Perturbing Smart Contract Execution Through the Underlying Runtime

Pinchen Cui(✉) and David Umphress

Computer Science and Software Engineering, Auburn University,
Auburn, AL 36849, USA
{pinchen,david.umphress}@auburn.edu

Abstract. Because the smart contract is the core element that enables blockchain systems to perform diverse and intelligent operations, the security of smart contracts significantly determines the reliability and availability of the blockchain applications. This work examines security from the perspective that, although a smart contract may be programmatically correct, the environment in which the smart contract is carried out is vulnerable. Adversaries do not need to necessarily concern themselves with how a smart contract is programmed or whether it is vulnerable; the integrity of the smart contract can be undermined by perturbing the output of smart contract execution. Such an approach does not rely on exploiting programming errors or vulnerabilities in smart contract verification and protection frameworks. Instead, it leverages the flaws in the underlying smart contract lifecycle and virtualization mechanisms. The Hyperledger Fabric platform is used to demonstrate the feasibility of the proposed attack.

Keywords: Blockchain · Hyperledger · Docker · Container · Smart contract · Security · Man in the middle

1 Introduction

A "smart contract" is a computation that is performed on a blockchain. The term is an oblique reference to the traditional notion of a legal contract in that it signifies signatories entering into some binding agreement regarding something. "Smart" signifies that software automatically triggered by the agreement carries out a series of actions that define the terms of the agreement; "contract" signifies that the results of the actions are recorded onto an indelible transaction ledger, such as a blockchain. The transactions themselves, once stored onto a blockchain, are considered, for the most part, secure. Executing the smart contract, on the other hand, raises questions. How open to vulnerabilities is the "smart" part of "smart contracts"?

On the one hand, the security of a smart contract relies on how formal and secure the contract has been programmed. Since the smart contract is designed

N. Park et al. (Eds.): SecureComm 2020, LNICST 336, pp. 336–349, 2020.
https://doi.org/10.1007/978-3-030-63095-9_22

to be a public application, any internal programming vulnerabilities can incur enormous influence on all the contract users. Therefore, several evaluation and verification frameworks at the programming level [1–6] have been proposed. These frameworks and tools evaluate the validity and security of smart contracts at the programming level by creating certain rules and boundaries for smart contract programming. With the examination of the smart contract code context, the smart contract can be converted, compiled, and regulated to a secure form. These frameworks focus on the security of the smart contract only in the Ethereum platform[1]. While there exists another platform, Hyperledger[2], which provides competitive smart contract functionalities, no similar smart contract security investigation is evident.

On the other hand, the environment in which the smart contract is carried out also determines security. We take the perspective of an adversary that does not care how the smart contract is programmed, as long as we can interfere or manipulate the output of smart contract execution and thus bypass any verification and protection frameworks that might be in place. We attempt to illustrate the flaws of the underlying smart contract life-cycle or virtualization mechanism (runtime), namely, the Ethereum Virtual Machine in Ethereum and the Docker container environment in Hyperledger Fabric. Since the smart contract installation and execution in Ethereum and Fabric are different, we mainly focus to investigate the potential security risks in the Hyperledger Fabric system.

The contribution of this paper can be summarized as follows:

- We elicit a new attack vector in the smart contract ecosystem. Instead of focusing on smart contract programming, we propose to perturb the smart contract execution at the life-cycle and runtime level.
- A detailed case study on Hyperledger Fabric has been performed, which proves the possibility and feasibility of the proposed attack.
- We briefly discuss and analyze the practicability of launching this attack on Ethereum and other platforms.
- A new threat model of smart contract systems has been created based on the findings in this paper.
- We summarize and demonstrate the limitations and countermeasures of the proposed attack.

2 Background

A smart contract is an automated agreement enforced by tamper-proof execution of computer code [7]. In the context of blockchain, smart contracts are scripts stored on a blockchain system and which enable users to perform general-purpose computations on the blockchain. Smart contracts can be applied to various fields,

[1] https://github.com/ethereum/wiki/wiki/White-Paper.
[2] https://www.hyperledger.org.

including B2B international transfers, central clearing, mortgages, and crowdfunding [8]. The use of smart contract enhances the integrity, traceability, and transparency of data, and further benefits the applications in many domains.

To date, only a few platforms provide a full implementation of a smart contract. Two of the most successful and widely deployed implementations are Ethereum and Hyperledger Fabric (or Hyperledger, since many other subprojects and blockchain applications are developed under the Hyperledger umbrella). Smart contract can be triggered by different nodes in blockchain network, these nodes can be different architecture-based and operating system-based. Ethereum and Hyperledger use different methods to ensure the smart contract can be run on all nodes and generate the same result. However, the core concept is the same: via virtualization. Ethereum adopts a virtual machine mechanism similar to the Java Virtual Machine, named Ethereum Virtual Machine (EVM). EVM is a stack machine that executes bytecode transformed from a high-level smart contract programming language (Solidity or Vyper). EVM is an embedded component of an Ethereum node client, which automatically runs in memory. In contrast, Hyperledger uses a Docker container to execute the smart contract. The smart contract in Hyperledger can be written in Go, Java, or NodeJS. The code is packaged and instantiated as a Docker container in the Hyperledger node's system. Each smart contract runs as a container, more details are described in Sect. 3.

3 Case Study: Attack on Hyperledger Fabric

In this section, we perform a case study on the Hyperledger Fabric platform to investigate the feasibility of perturbing the smart contract execution via runtime

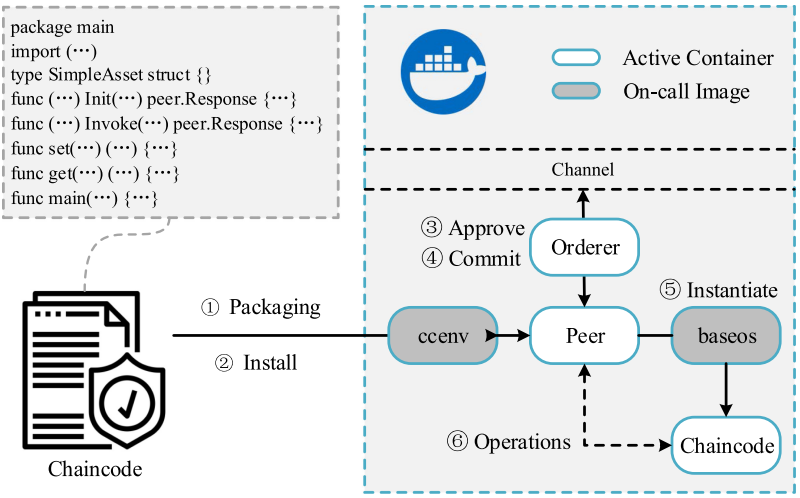

Fig. 1. Chaincode life cycle in hyperledger

vulnerabilities. The entire case study is based on the official Hyperledger network test example, *Byfn* network. We demonstrate the chaincode (the alias for the smart contract in Fabric) life-cycle and Docker environment of Hyperledger Fabric system before introducing the vulnerability itself.

3.1 Chaincode Life Cycle

Figure 1 illustrates the life-cycle of chaincode in the Docker containers that Hyperledger uses as its runtime environment. The following steps transpire when a Hyperledger blockchain peer tries to launch and test a piece of chaincode (starting in the top left corner of Fig. 1):

1. Packaging: The *Peer* first packages the chaincode into a tar format file.
2. Install: The compressed package is delivered to all the peers that need to run/endorse the chaincode. These peers build, compile, and install the chaincode locally.
3. Definition Approve: Corresponding channel members vote on and approve the definition of the chaincode, which includes such information as name, version, and endorsement policy (i.e., who can execute and validate).
4. Commit: Upon a success approval, a commit transaction proposal is submitted to the *Orderer*, which then commits the chaincode definition to the channel.
5. Instantiate: The complied chaincode is added into a base image to create the real instance of chaincode container.
6. Operations: Chaincode invoke and query operations are carried out by the communication between the peer container and chaincode container.

Throughout this procedure, the *Orderer* and *Peer* container are active all the time. The *ccenv* container, which is the chaincode environment container provides the functionalities of installation and instantiation. The *ccenv* is an offline docker image that only becomes an active container when there is a chaincode that needs to be processed. The *baseos* image is always offline.

3.2 Threat Model

In this case study, we identified two threats in the container runtime: insecure communication and loose image management. In the insecure communication threat, we assume the adversary may not have root privilege on the host machine, but he/she has access to the Docker network, images, and containers. This can be achieved using a pre-planted backdoor, malicious docker image, and/or remote control. The communication of the victim can be eavesdropped, intercepted, and modified by the adversary. The adversary can be an unrelated third party or an insider. The prerequisites of these attacks may enable the adversary to damage the system in a more severe and obvious way, but the major motivation of the adversary is to bias, perturb, and stop the service of the chaincode. We further

assume the adversary can deliver malicious docker images to victims in the loose image management threat.

The detailed attacks and consequences are presented in Subsects. 3.3, 3.4 and 5.1. For the chaincode invoke operations, the adversary can intercept and manipulate the original input to chaincode using a malicious image, and the faulty data would be added into the blockchain. The adversary can also modify the local chaincode execution result returned to the user using Man-in-the-Middle (MITM) channel (and/or malicious image), thus the transaction proposal of the chaincode invocation will be failed during the endorsement procedure. Namely, the DoS of the smart contract can be achieved.

On the other hand, the adversary can intercept the return results of the chaincode query operations using MITM and/or malicious image, so the query of chaincode data can be manipulated by the adversary.

3.3 Insecure Communication

The operations to an instantiated chaincode in Hyperledger are based on the communication between the *Peer* container and *Chaincode* container. We sniffed the communication traffic between these two containers while a query operation was taking place. As shown in Fig. 2, the communication was TLS v1.2 enabled, the encryption of the communication was based on ECDH Key Exchange, and the authentication was provided by mutual certificate verification. Normally, we should presume the communication is secure and reliable. However, the vulnerability came from a permissioned blockchain and Docker container.

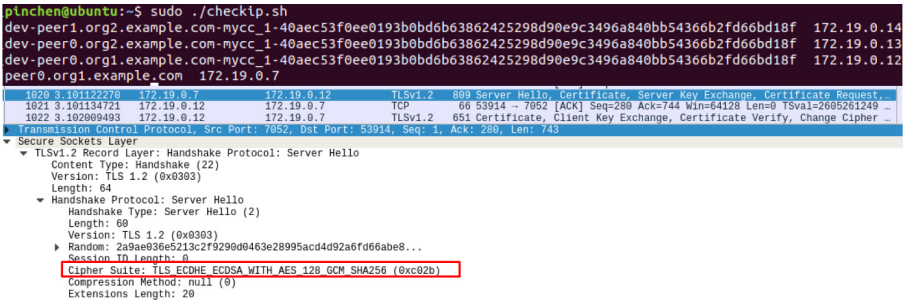

Fig. 2. Communication and handshake between the peer and chaincode containers

The permissioned blockchain requires the network to be a designated group of organizations and entities, and the enforcement of the network regulation relies on authentication. In other words, all the entities need to have corresponding certificates and keys for further verification. All the cryptographic materials are pre-generated and shared in the entire network *by loading them into the container at the point the container is created*. This creates a problem: all the keys and certificates are stored in the user space of both host and container. Figure 3

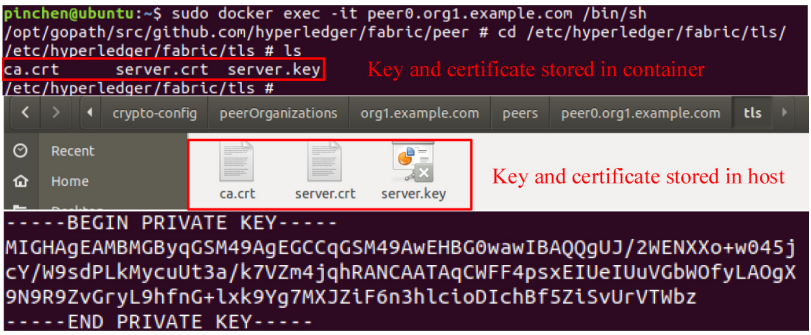

Fig. 3. Unencrypted key and certificates stored in host and containers

shows the accessible keys and certificates stored in host and container, which are also stored in an unencrypted manner.

The Hyperledger project provides a *Cryptogen* binary to help users tailor their cryptographic materials. Users can change the location of the keystore, change the key format and length, and create encrypted keys and certificates, but such information has to be loaded and stored into containers. Hyperledger containers come with root privileges, which provides access to keys. **As long as an adversary has access to any container, he/she has access to this material.** Hyperledger adopts ECDH key exchange, which is secure for

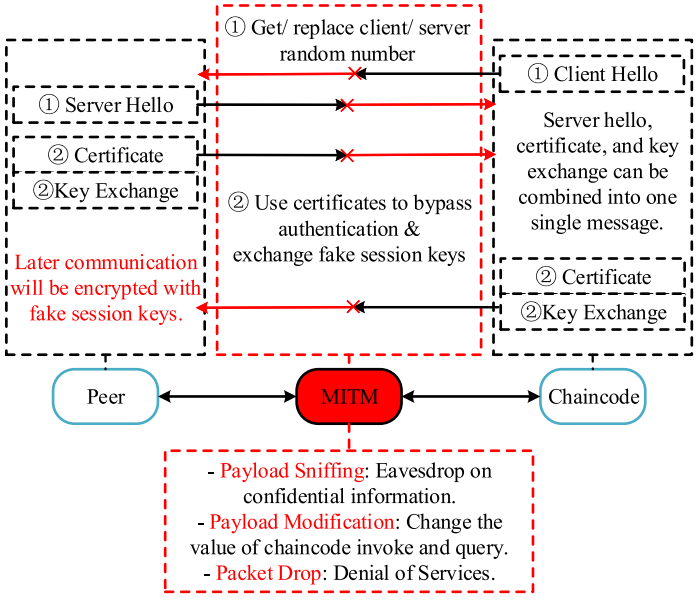

Fig. 4. Demonstration of MITM attack on chaincode communication

key exchange over an unencrypted channel. However, the overall "secure communication" relies on a mutual certificate authentication because the identity of the opposite entity can not be confirmed without it [9]. We discovered that the certificates are accessible on all the containers and can be compromised. The adversary can pretend to be a compromised node and set up a MITM channel to launch further attacks.

Traffic redirection tools (e.g. iptables and ARP spoofing) and TLS interception tools (e.g., SSLProxy[3]) make it relatively easy for an adversary to redirect the duo-direction communication between the *Peer* and *Chaincode* containers to a MITM agent. **All the operations to the chaincode can be then manipulated by the adversary**. An example attack scheme is described in Fig. 4. Note that, besides the *Peer-Chaincode* communication, there is another vulnerable point. **The communication between the *Client* (user command-line tool/ container that used to send the requests to Peer) and *Peer* can be the target of MITM attack as well.**

After this stage, the attack becomes an engineering task of forging all the malicious packets. This is obviously nontrivial, but feasible. We have plans to explore this further in the future; the aim of the current work is to suggest looking to the environment within which the secure contract executes for possible vulnerabilities.

3.4 Loose Image Management

As shown in Fig. 1, the chaincode container is created by loading chaincode binary file into the *baseos* image. One potential strategy for circumventing the integrity of the chaincode would be to poison the base image, thus ensuring subsequent chaincode containers would be vulnerable if the *baseos* image were modified or replaced. The *baseos* image, along with other Hyperledger container images, are ill protected. Containers are called via tags instead of hashes. This means that a benign image can be modified or replaced and still appear to be valid if it has a tag that corresponds with the original.

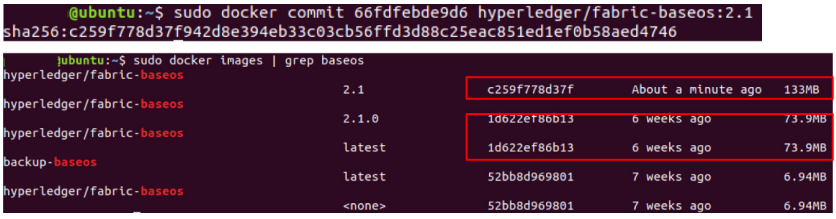

Fig. 5. Replace the Hyperledger baseos image

Figure 5 illustrates a simple example of such an image alteration. The original on Hyperledger v2.1 *baseos* image was Linux Alpine based, with an image size of

[3] https://github.com/sonertari/SSLproxy.

6.94 MB. We replaced all the versions of *baseos* image to a Ubuntu-based image of 73.9 MB. In addition, we modified the clean Ubuntu image with Python and some other libraries installed, which also has been committed to *baseos* image version 2.1. **We observed that all the chaincode containers created after this image alteration were installed with Python and additional libraries**.

In this case, if the adversary can either redirect the user to download a malicious image or somehow modify/replace the image, the entire system can be corrupted. The adversary can load customized code, autorun rootkit, revert channel backdoor, and cryptocurrency miner program into the malicious image. These malicious images can lead to denial of services, abuse of resources, unauthorized access, and information leaking. One can also use the malicious image to launch the aforementioned MITM attacks.

Fig. 6. Dev-chaincode containers on-the-fly

This naive attack can work because the Hyperledger container life-cycle environment calls all the images with tags. Image integrity is guaranteed in the Docker container system using hashes. The system can ensure a particular image has not been modified using hash but it cannot stop the tag from being written another image. The adversary can re-assign the corresponding tag to malicious images. Although the hashes are inherently provided by Docker engine, it is apparently more convenient, albeit incautious, to use the human-readable but less secure tags in the life-cycle. Moreover, besides the *baseos* image, all the other Hyperledger images are vulnerable as well. The orderer, CA, and peer images are all free for modification and replacement. Each installed and instantiated chaincode will be mapped with a container image as well (shown in Fig. 6), these images that generated on-the-fly can be also attacked.

3.5 Risks Behind Docker

Hyperledger outsources Docker as the chaincode runtime, and its security is then bounded with Docker container security, which basically doubles the attack surface. The adversary can utilize the vulnerability of Docker to perform much more severe attacks. For example, one can combine a malicious image with reverse shell and Docker escape attack to gain the root access of the host system[4]. This can lead the attacker to have full control of the entire system. An example set up is shown and described in Fig. 7, where upon the invocation of the malicious image, a reverse shell establishes and returns with the root access of the victim's host. If the adversary replaces the *baseos* image with this malicious one,

[4] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-5736.

the entire chaincode system is disrupted. Note that the malicious docker images problem has been of concern for some time. By uploading malicious images into Docker Hub, an unscrupulous actor can generate $90,000 dollars from the million downloads and deployments of these malicious images in 10 months[5].

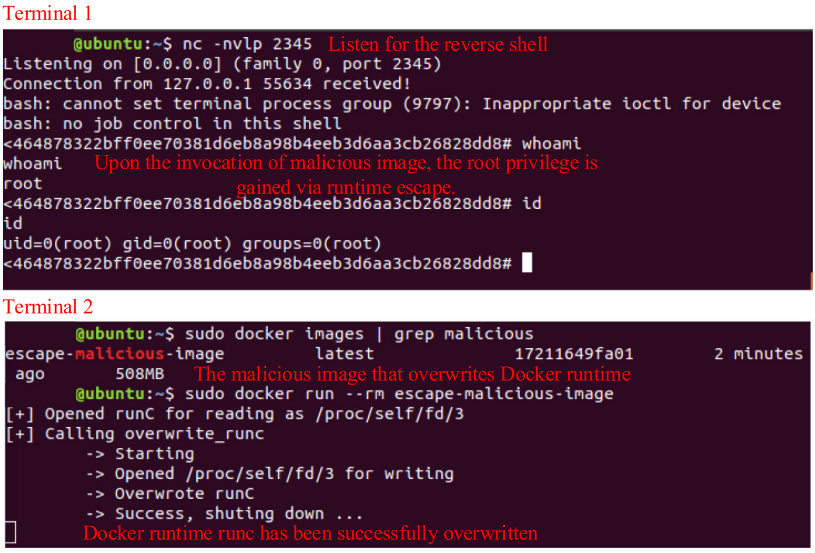

Fig. 7. Gain root access on host via malicious image: this attack allows the adversary to inject any code in docker runtime (runc), and the code will be executed on host with root privilege. This example simply injects "$bash - i > \&/dev/tcp/0.0.0.0/23450 > \&1\&$" into runc, and a reverse shell with root access on victim's host will be created.

Complicating things further, Docker and Docker Compose are normally configured to be user-space applications. Since it is not practical, efficient, and secure in a production environment to ask all the persistent container operations for root privileges, the Docker environment can be significantly manipulated even without root access on the host system. Issuing Docker commands on the host can be another threat to the chaincode system. One can stop and re-run a chaincode container to break the established TLS connection with *Peer* container, thus, the sniffing and MITM can be launched at any time. One can run the containers with privileged mode, so the *iptables* and all the other kernel-related system calls are enabled in the container for further malicious objectives (e.g., setup network forwarding rules and divert channels, and load malicious kernel modules).

[5] https://arstechnica.com/information-technology/2018/06/backdoored-images-downloaded-5-million-times-finally-removed-from-docker-hub/.

4 What About Ethereum and Others?

Generally, it may seem that Ethereum is more secure than Hyperledger since its runtime, EVM, is a customized in-memory stack machine. We focus on Hyperledger is not only because it is less discussed or arguably less secure, but also due to the fact that EVM has already been attacked in a similar form. Both EVM stack overflow attacks[10,11] and CVE-2018-18920 (an internal flaw of python implementation of EVM)[6] attacks perturb the normal execution of smart contract based on the vulnerabilities in EVM implementation. These two attacks can infinitely trigger the smart contract functions without corresponding gas and payments, which also do not rely on any programming faults in the smart contract. Although these two vulnerabilities have been already fixed, the concept of our proposed attack is verified.

A research question is whether it is possible to attack the EVM without a zero-day vulnerability. Because EVM is a program running in the local system, it can be perturbed if the system owner (or the adversary with the same privilege) decides to do so. Indeed, any program and applications can be attacked in this manner as well, this type of attack is beyond the scope of "attack on the runtime". A smart contract is different from the other application scenarios. **Since the contracts are immutable and reliable hardcoded programs in the blockchain, altering the execution or the result of smart contracts even in a more general manner would still be interesting**. One potential direction is to locate the EVM stack and memory locations in physical memory space and use the *process_vm_writev()* system call to transfer and inject data into that memory location, thus altering execution.

We note that there exist some other smart contract platforms, such as EOSIO and NEO EOSIO adopts a customized web assembly virtual machine (EOS-VM) to transform a C++ smart contract, making the code executable across platforms. NEO uses an enriched EVM-style stack machine to execute the smart contract. Generally, the threats and concerns of all other VM-like runtimes would be similar to the EVM.

5 Lessons Learned

5.1 Limitation and Impact of the Proposed Attack

Altering smart contract execution requires access to the runtime environment and/or certain vulnerabilities present at runtime. Access to the runtime environment may lead to other security concerns and make the perturbation unnecessary. For example, the adversary can simply remove the Docker engine from the victim's system, thus achieving DoS. Similarly, one can block the EVM implementation and Ethereum client from normal functioning by setting up certain network rules or checkpoints (via debugger). We notice that the assumption of having partial access to the system is a strong assumption, but note that all the

[6] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-18920.

aforementioned attacks in Hyperledger can still be performed without access to the system, as long as a malicious image is delivered.

The detailed attack consequences depend on the type of container/communication compromised by the adversary:

- Orderer Container: Gaining control of the orderer container allows an adversary to i) partially DoS the blockchain network by dropping transactions sent to the orderer node; or ii) perturb the overall transaction propagation in the orderer cluster.
- Peer Container and/or Chaincode Container: The adversary can use a compromised container to send malicious chaincode invocations (the input to chaincode is manipulated), so the final data uploaded into blockchain can be manipulated. Although the users can find out the wrong values added in the blockchain and examine the local system afterward, the faulty data has been already uploaded.
- Communication between Peer and Chaincode Containers: When the user invokes/queries a chaincode, an adversary can DoS or perturb this chaincode execution procedure using the MITM channel. The modification on invocation execution results leads to the failed transaction endorsements, thus DoS of chaincode can be achieved. The modification on query execution results enables the adversary to manipulate the chaincode query functionality and fool the victim.

5.2 Countermeasure to the Proposed Attack

Aforementioned problems can be solved in the following way:

- Malicious Image: The Hyperledger system should regulate all the invocation of containers to be bound with hashes instead of tags. Each time a container image is called, the hash needs to be compared, including the on-the-fly generated chaincode images.
- Access Control: If it is not necessary, all the Hyperledger containers should be run in root-less mode. This may need additional libraries installed on the host and further supports from Docker community [7]. This can limit the adversary from accessing the keys and certificates.
- Communication Security: The adversary can obtain access to the private keys for further communication manipulation due to the confidential cryptographic material loaded into containers. This is a side-effect of malicious image and loose access control, and the problem can be fixed only if the previous two are properly handled. However, the use of containers also adds an additional communication layer between the peer and installed chaincode. Note that, the chaincode is installed locally per Peer, which means the *Chaincode* container and *Peer* container are running in the same machine. For the local system data exchanging between *Chaincode* container and *Peer* container, network-based

[7] https://docs.docker.com/engine/security/rootless/.

communication is not the only option. Docker supports shared memory (inter-process communication (IPC) namespace) for inter-container data exchange. Using IPC between *Chaincode* container and *Peer* container can eliminate the risk in network-based communication.

5.3 Threat Model of Smart Contract Systems

Based on the findings in this paper, a new threat model of the smart contract system is established and shown in Fig. 8. The threats can be divided into two levels, smart contract programming level [11], and runtime level. The threats in smart contract programming level are either caused by programming flaws or backdoors. The programming flaws including external dependence (e.g., re-entrancy, delegatecall injection), improper validations (e.g., integer overflow and underflow), and inadequate authentication or authorization (e.g., erroneous visibility, unprotected suicide). The backdoors are intentionally planted malicious functions, they may not violate any programming rules or fall in any programming flaw definitions. However, they can be used to trigger blockchain operations that can potentially prejudice the interests of others.

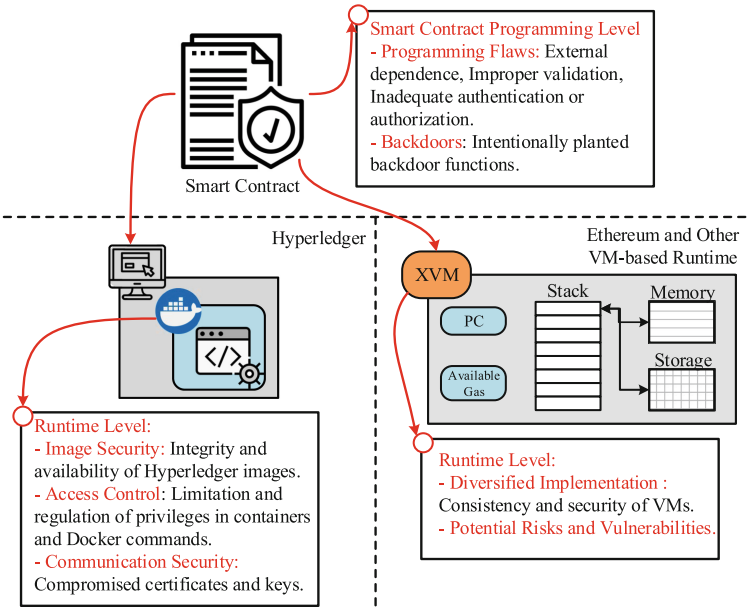

Fig. 8. Threat model of smart contract system

The runtime level threats in Hyperledger-based blockchain are bound to Docker container security. The integrity and availability of Hyperledger images

significantly determine the security of the chaincode system. If the access control in the Docker environment is not appropriately configured, keys and certificates can be accessed without authorization. The Docker commands that can be issued as a non-root user can also harm the chaincode system. For example, an evil insider can easily stop the containers to perturb and even DoS the blockchain system. The insecure communication and loose image management threats presented in this paper also need concerns.

On the other hand, the Ethereum-based smart contract system suffers from the diversified EVM implementations. Hyperledger adopts a single standardized runtime Docker as the universal chaincode runtime, whereas Ethereum provides different implementations of EVMs. As stated in [12], the gas and opcode consistency problem have been already found in different EVM implementations. Moreover, maintaining and ensuring the security of all the EVM implementations are non-trivial and challenging tasks. The attack in CVE-2018-18920 just utilizes the flaws in the Python version of EVM implementation. Some of the other potential risks are also indicated in the work [13]. As the development and maintenance of EVM continue, the security of Ethereum runtime needs more concern. Other VM-based runtime blockchains such as EOSIO and NEO may confront the same problems, however, the security analysis on these two platforms is limited.

6 Conclusion

Smart contract is the core functionality enabler for blockchain, thus it needs to be secure and reliable. This paper differs from the state-of-art works, which instead of analyzing the security of smart contract based on programming validation, proposes to investigate smart contract security at runtime level. A case study has been performed on Hyperledger blockchain, and we demonstrate the potential risks that exist in the Hyperledger chaincode runtime environment. The analysis and countermeasures are elaborated in detail. In addition, combining with other prior attacks on EVM mechanism, we propose a new threat model for smart contract systems which includes both programming security and runtime security.

References

1. Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Buenzli, F., Vechev, M.: Securify: practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, ACM (2018)
2. Park, D., Zhang, Y., Saxena, M., Daian, P., Roşu, G.: A formal verification tool for Ethereum VM bytecode. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 912–915. ACM (2018)

3. Hildenbrandt, E. et al.: Kevm: a complete formal semantics of the ethereum virtual machine. In: 2018 IEEE 31st Computer Security Foundations Symposium (CSF), pp. 204–217. IEEE (2018)
4. Wohrer, M., Zdun, U.: Smart contracts: security patterns in the ethereum ecosystem and solidity. In: 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE), pp. 2–8. IEEE (2018)
5. Perez, D., Livshits, B.: Smart contract vulnerabilities: Does anyone care? (2019). arXiv preprint arXiv:1902.06710
6. Brent, L. et al.: Vandal: a scalable security analysis framework for smart contracts (2018)
7. Clack, C.D., Bakshi, V.A., Braine, L.: Smart contract templates: foundations, design landscape and research directions (2016)
8. Cui, P., Guin, U., Skjellum, A., Umphress, D.: Blockchain in IoT: current trends, challenges, and future roadmap. J. Hardware Syst. Secur. **3**(4), 338–364 (2019). https://doi.org/10.1007/s41635-019-00079-5
9. Adrian, D. et al.: Imperfect forward secrecy: how diffie-hellman fails in practice. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 5–17 (2015)
10. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on Ethereum smart contracts (SoK). In: Maffei, M., Ryan, M. (eds.) POST 2017. LNCS, vol. 10204, pp. 164–186. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54455-6_8
11. Chen, H., Pendleton, M., Njilla, L., Xu, S.: A survey on Ethereum systems security: vulnerabilities, attacks and defenses. ACM Comput. Surv. **53**(3), 1–43 (2020). https://doi.org/10.1145/3391195
12. Fu, Y., Ren, M., Ma, F., Jiang, Y., Shi, H., Sun, J.: Evmfuzz: differential fuzz testing of Ethereum virtual machine (2019)
13. Fu, Y. et al.: Evmfuzzer: detect EVM vulnerabilities via fuzz testing. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1110–1114 (2019)