Tineola: Taking a Bite Out of Enterprise Blockchain

Stark Riedesel, Parsia Hakimian, Koen Buyens, Travis Biehn

`{riedesel, hakimian, kbuyens, tbiehn}@synopsys.com`

Synopsys, Inc.

Abstract

Enterprise blockchain adoption has reached a fever pitch in 2018 and the security community has been late to the game of securing these platforms against attacks. To fill this gap, we explore the leading enterprise platform, Hyperledger Fabric, from the perspective of a penetration tester. To this end, we developed Tineola, a new red teamer's tool for interacting with Fabric deployments. The goal of Tineola is to make tangible and demonstrable the theoretical weaknesses of enterprise blockchain. With the help of Tineola, we detail six typical anti-patterns of composed systems with Fabric components.

Introduction to Blockchains in the Enterprise

Blockchain adoption has reached a fever pitch in 2018 and the security community has been late to the game of securing these platforms against attack. While the open source community has been enamored with the success of Ethereum, the enterprise community has been quietly building the next generation of distributed trust-less applications on permissioned blockchain technologies[1].

What started as pilot projects in 2016 are going live in late 2018 and will continue to grow significantly in the next five years. As of early 2018 an estimated 50% of private, permissioned enterprise blockchain projects rely on the HyperLedger Fabric[2] platform.

In this paper, we explore Fabric from the perspective of a penetration tester with the goals of exploring common misconfigurations, vulnerabilities, and pitfalls using concrete, repeatable processes. To this end, we demonstrate the capabilities and use of Tineola, a new red teamer's tool by the authors, for interacting with Fabric deployments.

Current Literature

Security of enterprise blockchain platforms is an unexplored topic. While public blockchains are constantly in the media spotlight, the narrative around enterprise blockchain has been drowned out by marketing hype. So far, the dialog around security has been focused on cryptocurrencies, specifically on market manipulation, exchanges, wallets, and Ethereum smart contracts.

Reflecting the lack of industry focus, Fabric in particular has undergone only limited public security audits. The Fabric Security Model document consists of two paragraphs[3] and has not

[1] https://www.gartner.com/newsroom/id/3873790
[2] https://arxiv.org/abs/1801.10228
[3] https://hyperledger-fabric.readthedocs.io/en/release-1.2/security_model.html.

been modified since release 1.0[4]. Fabric underwent one public security audit when version 1.0 was released in August 2017[5]. The focus of that assessment has been on smart contracts, their execution environment, and gRPC endpoints.

Public vs Private

Before elaborating on what enterprise blockchains are, we need to understand what they are not. Public blockchains allow everyone to participate in the network (with different degrees of access), while private blockchains regulate access via membership control (e.g. consortium consensus, IP whitelisting, certificate validation, use of a central authority, etc)[6].

While enterprise blockchains platforms have roots in established public blockchains like Bitcoin and Ethereum, they have significant and security-relevant differences. Unlike Bitcoin, the private blockchain's primary use case is not to store and transfer value, but to enforce arbitrarily complex business logic. While the terms "coin" or "wallet" may still be used, these are rarely treated as currency analogs in enterprise use cases. To separate themselves from the blockchain hype in the public sphere, enterprise practitioners often use the term Distributed Ledger Technology (DLT), referring to the general capabilities Fabric and other platforms provide.

The cryptocurrency scene is very diverse and has become one of the largest areas of active research. The enterprise space is relatively nascent, with three major projects working to establish themselves:

- Hyperledger project[7]: Started by the Linux foundation, Hyperledger is an ecosystem of different projects in the DLT space.
- Enterprise Ethereum Alliance[8] aims to increase adoption of systems based on the Ethereum blockchain.
- Corda[9] is an enterprise blockchain platform by R3.

Promises of the Blockchain

The promises made by blockchain platforms include the following:

Immutability is the guarantee that, once data has been written to the blockchain it will be tamper proof indefinitely. Compromised members of a network may have their local data modified but inconsistencies should be identified by the network. A small number of compromised peers should never be able to rewrite history.

Auditability is the direct result of the immutability guarantee combined with historical data kept by all members of a blockchain. The upshot is that, for any change made to the network (i.e. a transaction) the member who initiated the transaction, the inputs and outputs, and even the state of the network at that point in time, should be retrievable. Forensic and fraud investigators should have their jobs made much easier by DLT.

[4] https://hyperledger-fabric.readthedocs.io/en/release-1.0/security_model.html.
[5] https://wiki.hyperledger.org/_media/security/technical_report_linux_foundation_fabric_august_2017_v1.1.pdf.
[6] Although not exactly equivalent, in the context of this paper, public/permissionless and private/permissioned can be used interchangeably.
[7] https://www.hyperledger.org/
[8] https://entethalliance.org/
[9] https://www.r3.com

Tuneable-Trust promises that applications built on DLT do not need to trust all members of their network. Unlike traditional networks and distributed systems where all members ether must trust each other or a 3rd party network operator, DLT is designed from the ground up to have minimal trust requirements. Performance and scalability is still a problem, so DLT provides some knobs for adjusting these trust relationships while still providing partial guarantees against malicious or compromises parties.

Programmability means the rules of the network are codified into so called smart contracts. These blockchain programs describe precisely with code how data changes, ultimately comprising an enormous state machine. The programmability of private DLTs, especially platforms similar to Fabric, is greatly expanded over what is possible in Bitcoin or Ethereum networks. The state changes can be arbitrarily complex (i.e. are turing complete) and may even include arbitrary side effects outside the chain. This makes enterprise blockchain platforms extremely versatile compared to their counterparts.

Challenges of the Blockchain

Everything is not quite as it seems. Many of these strengths are also challenges to overcome, especially when it comes to security requirements:

Immutability means that not only can the data not be modified by the bad guys, but also that data will stick around indefinitely for eventual compromise. Creating large repositories of sensitive data just waiting to be compromised is a major risk and blockchain is no different. Generally, compromise of a blockchain leaks all historical transaction inputs, outputs, and intermediate values. Plain text sensitive information should ever be written to or output from blockchain systems.

Mutability creeps into the blockchain as the design diverges significantly from public chains. Removing proof of work and longest chain rules results in higher risk of counterfeit blocks; in proof of authority systems possession of a key is sometimes sufficient, in others, any participant may potentially generate arbitrary valid chains. Additionally, while the immutability property is provided in asset-transfer or UXTO[10] based cryptosystems like Bitcoin, smart contract developers must implement immutability in the domain of their application logic. Ultimately, expected immutability properties frequently fail to hold.

Privacy is an important factor during both the design and implementation phases. Due to the nature of a ledger, a lot of thought should go into what will end up on the chain. If at any time, a new privacy law comes into play (e.g. the European GDPR), removing data from the blockchain can be costly.

Correctness and Speed is a trade-off made by enterprise platforms necessary for scaling Blockchain technology to millions of transactions per day. For example, instead of running the smart contracts on each node independently, a platform may only execute its logic on just a small subset (as low as just one). If those nodes go rogue, it is possible to falsify the output of the smart contract and write invalid data to the ledger.

[10] Unspent Transaction Outputs

Execution Environments differ from non-blockchain software, sometimes, in the way code interacts with non-blockchain data[11], in all cases, where it executes, the order of execution, and what constitutes a valid execution and subsequent state transition. Due this novelty, developers and security professionals alike have difficulty understanding whether the finite state machine matches the intended finite state machine, leading to outsized risk that defects will be introduced during development, and not caught in review.

Platform Complexity is a confluence of all above factors, Enterprise Blockchain Platforms are new, poorly understood, involve many moving pieces. Further, as platform authors have focused on creating platforms that accomodate poorly understood, or unknown, use-cases. A lack of maturity in the understanding of application means that the idiomatic code, or platforms, that prevent insecure systems from being written in the first place do not exist. Difficult to understand, hard to develop for defensively, largely opaque to inspection, these systems create a perfect storm for the creation of insecure systems.

A Quick Fabric Primer

While we do not provide an in-depth description of Fabric, this section builds the foundation for analyzing the platform's security model. To get more information on the architecture, operation, and features of Fabric, please refer to the Fabric documentation[12].

Using smart contracts (a.k.a. chaincode), developers enforce complex business logic into rules for state change. Architecturally, Fabric deviates significantly from Ethereum-like systems; instead of all nodes executing contracts redundantly, only a subset of peers (endorsers) carry out the execution. The platform relies on carefully crafted endorsement policies to prevent any one party from breaking or changing the rules. Through the use of data partitions, organizations can maintain private information and logic on the same blockchain infrastructure. Finally, Fabric does not work in isolation; familiar components, web applications, services, and databases, are used to round out a typical tech stack.

Chaincode

The code running on the Fabric blockchain is known as chaincode (CC). Chaincode in Fabric is a program (written in Go, Java, or JavaScript) that processes the input and interacts with the chain via an internal Key/Value store known as the stateDB. The chaincode is given an API, known as the Shim, which has two basic commands for interacting with the stateDB: `GetState()` and `PutState()`; and two commands for passing back values to the caller: `Success()` and `Error()`. Additionally, most calls to Fabric chaincode are structured as a function name and a list of arguments. Figure 1 is a snippet from the fabcar example of the official example repository[13]. In this example, the chain function `changeCarOwner`[14] is called with two arguments: a car identifier, and a new owner for that car.

[11] Ethereum smart contracts, sandboxed to the EVM environment, depend on oracles for external state introduction, while Fabrics' chaincode, equipped with access to make kernel `syscalls`, can pull from local and remote environments.
[12] https://hyperledger-fabric.readthedocs.io/en/release-1.2/
[13] https://github.com/hyperledger/fabric-samples
[14] Routing is usually handled in the Invoke function of Fabric chaincode.

```
func (s *SmartContract) changeCarOwner(APIstub shim.ChaincodeStubInterface, args []string) sc.Response {

        if len(args) != 2 {
                return shim.Error("Incorrect number of arguments. Expecting 2")
        }

        carAsBytes, _ := APIstub.GetState(args[0])
        car := Car{}

        json.Unmarshal(carAsBytes, &car)
        car.Owner = args[1]

        carAsBytes, _ = json.Marshal(car)
        APIstub.PutState(args[0], carAsBytes)

        return shim.Success(nil)
}
```

Figure 1: `changeCarOwner` interacting with the state DB

One important property of chaincode is the complete isolation of its state values from external influence. Chaincode is designed to be the exclusive owner of its state and therefore should completely describe permitted changes. Chaincode should specify all validations and state restrictions without relying on trusted inputs.

Channels

Fabric supports partitioning by creating multiple isolated channels for data and chaincode. Each channel has its own isolated blockchain. Members of the network, known as **Peers**, are joined to channels by the channel's administrators and they become party to all transactions of that channel. Chaincode is installed to peers and then its stateDB is initialized to a specific channel. Peers each maintain a full history of **Blocks** (transaction inputs and outputs) for each of their channels. Peers and chaincode can each be members of multiple channels, but all state data, blocks, and transactions are isolated on channel boundaries.

Membership Service Provider (MSP)

The MSP is the authentication provider of Fabric. The design is pluggable, but the only currently supported implementation is a X.509 certificate-based authentication mechanism. Clients can authenticate to a **Certificate Authority** (CA) server which maintains a list of allowed usernames and passwords, known as **Enrollment IDs** and **Enrollment Secrets** respectively. Registered users can enroll new certificates bound to their identity. These certificates contain additional information known as **Attributes** which are used internally by Fabric for authorization decisions. Attributes may also be used by chaincode for business logic. To authenticate to a peer it is necessary to provide a MSP-signed certificate and sign messages with the corresponding private key. The message also includes the MSP ID which tells Fabric which root certificate to validate, and acts as an organization identifier.

Deployment Topology

Usually a fabric node consists of a docker daemon running on a local Unix socket. The peer container exposes two gRPC ports (usually wrapped in TLS). One is a messaging service for issuing commands and the other is an event service for subscribing to events. Each chaincode runs as its own isolated docker container, managed by the peer container through the docker socket. Chaincode interacts with the peer container over the peer's messaging gRPC port. The process of installing chaincode to a peer involves a client sending the peer source code and the peer building and deploying it to a new docker container. The same process is done for upgrading chaincode to a new version. If a chaincode container crashes or otherwise disappears, the peer will spin up the chaincode on demand when a transaction requires it.

Generally, each member organization in a Fabric network will operate one or more peers and a single Fabric CA server as an HTTPS service. By default, the Fabric CA server utilizes a Sqlite3 backend but may also rely on a MySQL or PostgreSQL backend, usually for high availability. Peers, by default, utilize a LevelDB backend for storing the current state values for all chaincodes it has initialized. LevelDB is file-based and will sit in the ephemeral peer container. Some advanced chaincode shim features, such as `GetQueryResult()`, require a "rich" database backend. The currently supported database for these features is CouchDB which usually runs as a container parallel to the peer container.

The State Machine

It is important to understand how transactions are handled by Fabric before delving into potential issues. The state machine responsible for handling transactions is non-trivial and contains multiple potential pitfalls. Figure 2 provides a graphical representation of this process.

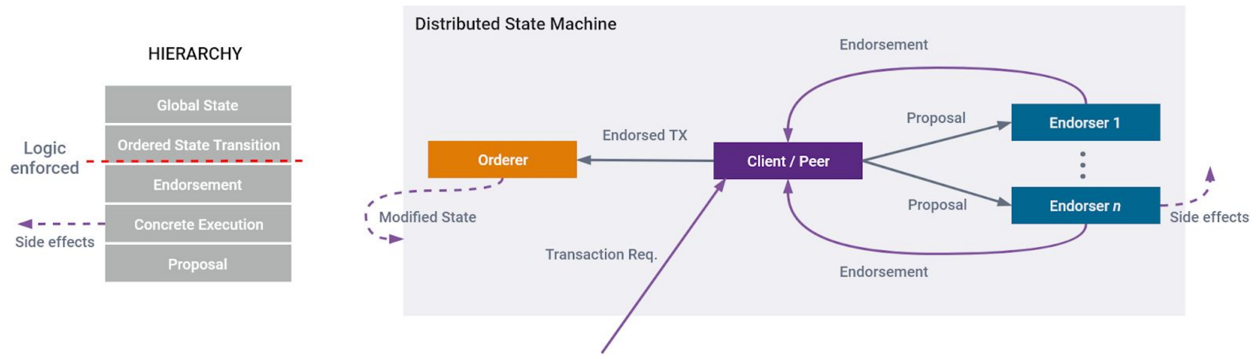


Figure 2: Fabrics' Distributed State Machine

Step 1: Proposal

A **Client**, usually using a Fabric Client SDK, submits a transaction **Proposal** to one or more peers known as **Endorsers**. The proposal includes Chaincode ID (name and version), Channel ID, inputs (usually a function name and zero or more arguments), MSP ID, user's MSP certificate, and a signature to validate the user's certificate. A proposal can also include **Transient Data** which is private data, accessible by chaincode, but only provided to endorsers. Transient inputs will not be included in the final block and therefore cannot be independently verified by non-endorsing peers.

Step 2: Concrete Execution

The endorsing peers process the transaction using their view of the chain (as described by their local LevelDB or CouchDB instance). There is no guarantee that their view of the current state database is consistent with the rest of the network. Instead, each endorser simply executes the transaction and records the resulting outputs.

Chaincode consists of arbitrary code which can have arbitrary functionality, rely on arbitrary libraries, or even communicate with network resources. Side effects and non-deterministic behavior are considered bad practice by the blockchain community, but Fabric never prevents this capability. It's often the case that blockchains need to reach out to an **Oracle** which provides some state data external to the blockchain (e.g. the current USD/EUR exchange rate). To prevent cheating by clients, endorsers may need to retrieve this oracle data independently. All endorsing peers must agree on outputs, so care must be taken by chaincode developers to eliminate sources of inconstancy when using oracles and other external data.

Lastly, the side effects of chaincode must be carefully tracked. Chaincode that requests external resources may leave logs, or incur expense (such as hitting a rate limited API) for transactions rejected later in the state machine and therefore do not end up as part of the chain.

Step 3: Endorsement

After the endorsing peer executes the chaincode, it returns a signed payload known as the **Endorsement** back to the client. The endorsement contains both the transaction inputs, and the outputs known as the **Read/Write Set**. The read set contains the key name and version of every state value read during execution. The write set contains the key name and the value being written to the stateDB. Note that, at this point the peer's stateDB has not changed, the endorser has simply calculated the resulting state change if the transaction makes it into the blockchain and put its signature on it. Queries to the peer's state will still reflect the original values before the transaction was executed.

The client will collect endorsements from each peer and can validate that the chaincode has executed as expected (i.e. it has not errored). Clients can then choose to pass the set of endorsements to the orderer.

Step 4: Ordered State Transition

Up until this point, no guarantees have been made around the correct execution of chaincodes. The client could forge signatures of endorsements, the endorsers could lie about outputs, the outputs may conflict with each other or past events, or the endorsements may be insufficient. The orderer is responsible for validating incoming endorsed transactions to ensure a consistent state is maintained. The orderer does the following validations:

1. All read/write sets from the endorsements must match exactly. Transactions with endorsements which have read from different keys, or have written inconsistent values will be rejected.
2. The endorsements meet the **Endorsement Policy** for a given chaincode. For instance, the policy may require 3 peers in the network of 5 peers to agree on outputs for acceptance. Public chains require every peer to execute every transaction, while Fabric allows this trust model to be tunable for customizing the performance/security trade off.

3. The read set must show the most recent version for all keys. This prevents out of order computation and “double spending” attacks.

Importantly, the orderer does not independently validate the inputs and outputs of the transaction. In fact, the orderer lacks sufficient information to compute transaction outputs independently of the peers. The orderer trusts that any transaction payload with sufficient non-conflicting endorsement is valid and will blindly write it to the block. Although chaincode describes how states can change, it is ultimately the outputs of endorsed transactions which are responsible for final values. Thus, the security of the chain is rooted in the endorsement policy, not within the logic of chaincode.

Step 5: Global State

Assuming all the validations completed successfully in step 4, the transaction will be included in the next minted block and be broadcasted to all peers of the channel. Peers, upon receiving the new block, will update their state databases with new values. Peers can validate that the block was signed by a key matching the orderer’s root of trust (the orderer has its own MSP). Peers trust that the orderer will only have included blocks which have satisfied their endorsement policies and have no conflicts. Therefore they will not independently validate transactions. Regardless, due to transient data not being included in blocks, peers may not even be able to compute the transaction.

(In)Security in the State Machine

Analyzing this state machine for security properties is a very involved task. Furthermore, the configurability permitted by Fabric’s architecture requires understanding what guarantees are really provided by the platform. Figure 3 shows a few selected potential problems and pitfalls for developers using Fabric.

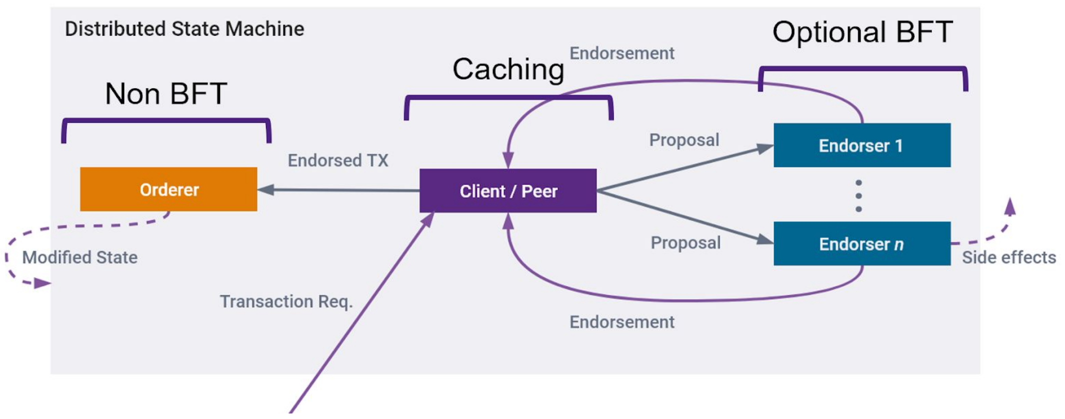


Figure 3: Problem areas in Fabric

Non Byzantine Fault Tolerance Ordering

Byzantine Fault Tolerance (BFT) is a property of distributed systems resilient to faulty, lying, or cheating nodes. The current design of ordering in Fabric does not have BFT and therefore a malicious orderer would have significant capabilities to disrupt network operation by having authority over the validations discussed in Step 4 of the State Machine section of this paper.

Fabric is currently limited to supporting only a single set of orderers for all channels, meaning they are party to all transactions. While transient data is not provided to orderers, the read/write set is necessarily exposed for all transactions. The downsides of relying on a central ordering party should be a major consideration when choosing Fabric as a platform.

Client Endorsement Caching

Clients can request endorsement of proposals and choose not to order them indefinitely. This may lead to interesting chaincode executions, especially when chaincode has side effects or non-deterministic behavior. For example, suppose a transaction that uses random values (a seed for a challenge/response protocol). A malicious client could request dozens of transactions to be endorsed until a predictable random value is chosen and then only submit that one transaction for ordering. Chaincodes which rely on side effects should carefully consider the impacts of endorsement caching.

Optional BFT from Endorsement Policy

The endorsement policy system used by Fabric is a great example of tunable trust in DLT. Sensitive chaincodes can insist on redundant endorsements for extra insurance against cheating, while less sensitive chaincodes can have weaker policies and shorter transaction times. Endorsement policies can also be written to require multiple organizations, not just multiple peers, thereby preventing Sybil style attacks. It is easy to write insecure policies and, by default, the endorsement policy is “any 1 peer on the channel.” The choice of the chaincode endorsement policy is a major consideration which often does not receive the necessary rigor. Further, this policy must be reconsidered whenever changes to channels membership happens or new, potentially more sensitive features are added to existing chaincodes.

Tineola

As applications built on Fabric reach production, security professionals are tasked with understanding what could go wrong with Fabric deployments. The authors of this paper created Tineola[15] to assist with penetration testing activities with the goal of creating concrete, repeatable, and documented steps for future tests. The following sections discuss the main use cases for Tineola. The ReadMe on the Github repository has additional information on features and steps for getting started.

The Strawman

To demonstrate Tineola’s capabilities this paper utilizes an open source demo application called Build Blockchain Insurance Application[16]. This application was chosen due to it being one of the few multi-peer demonstrations with non-trivial features. The four constituent peers are: Insurance, Shop, Repair Shop, and Police. Customers can enter into insurance contracts after buying items from the shop and self-report stolen or damaged items to the Insurance company. Insurers validate claims and pass contracts off to the Police (if theft has occurred) and Repair Shops. Repair Shops can mark items as repaired. Figure 4 shows the layout of the application.

[15] https://www.github.com/tineola/tineola
[16] https://github.com/IBM/build-blockchain-insurance-app

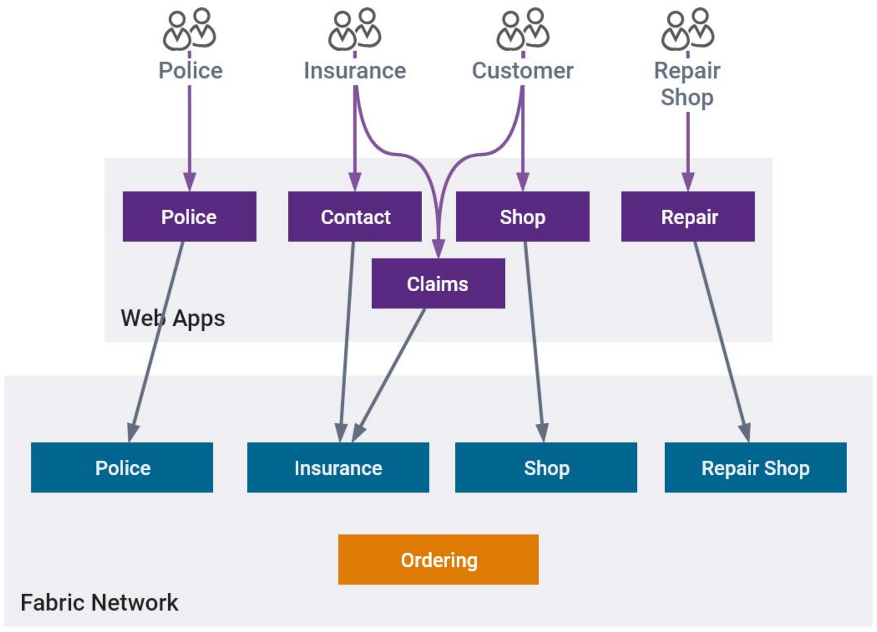


Figure 4: The composition of the Build Blockchain Insurance Application

Structure of Tineola

Tineola is an interactive command line application built in JavaScript and requires NodeJS 8.x or newer[17]. Tineola ships with help features which can be found by using the "`help`" command or by passing the "`--help`" flag to any Tineola command. Commands are broken down into modules as follows:

- User - setting up user context and managing the local X.509 keys
- CA - interacts with a Fabric CA server including user enrollment
- Peer - interacts with Fabric peer nodes including invoking chaincode
- Orderer - interacts with Fabric orderer nodes when submitting transactions
- Channel - attaching to channels and querying block history
- Tineola - installs and interacts with the Tineola

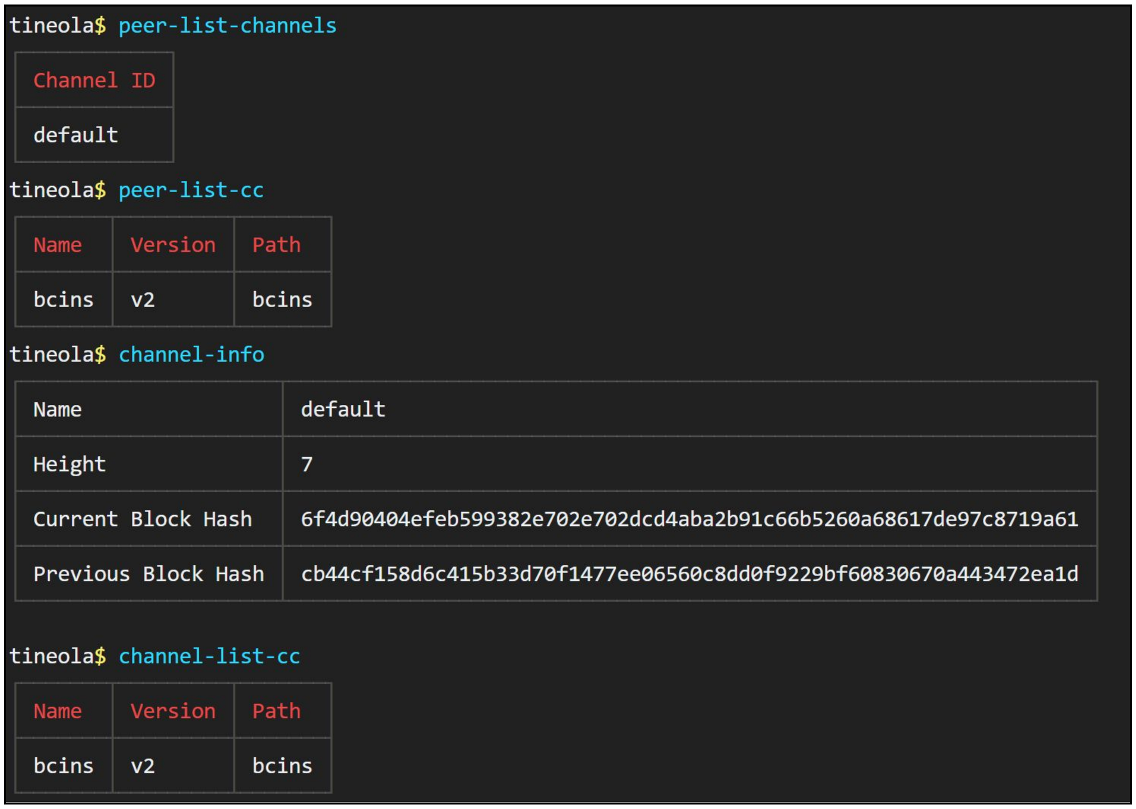
A typical Tineola connection looks like:

```
ca-set https://ca-server.org1:7050
user-set admin
ca-enroll adminpw Org1MSP
peer-set https://peer1.org1:7051
orderer-set https://orderer01.ordererOrg:7051
channel-set myChannel
```

[17] See the NodeJS docs for install instructions: https://nodejs.org/en/download/package-manager/

Enumeration

Step 1 for most red team activities is getting a lay of the land. Credentials may be stolen or guessed[18] and the attacker needs to know what the system does and how to interface with it. Tineola comes with a handful of useful commands to determine how peers are configured:



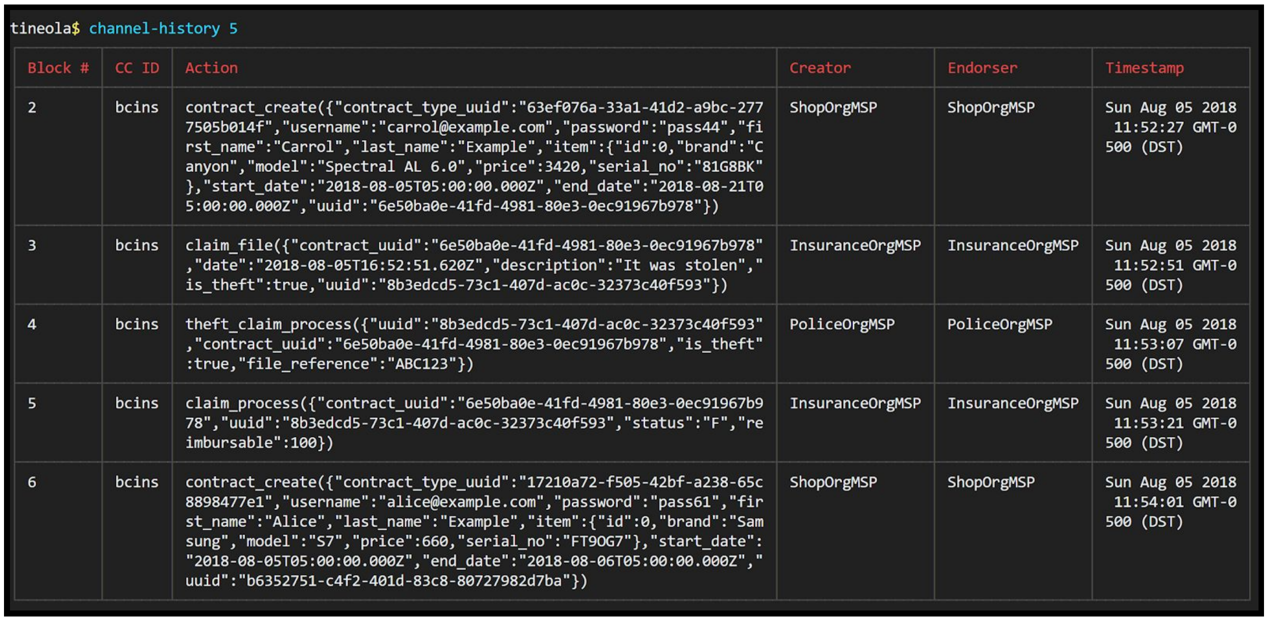
```
tineola$ peer-list-channels
 Channel ID
 default
tineola$ peer-list-cc
 Name    Version   Path
 bcins   v2        bcins
tineola$ channel-info
 Name                  default
 Height                7
 Current Block Hash    6f4d90404efeb599382e702e702dcd4aba2b91c66b5260a68617de97c8719a61
 Previous Block Hash   cb44cf158d6c415b33d70f1477ee06560c8dd0f9229bf60830670a443472ea1d

tineola$ channel-list-cc
 Name    Version   Path
 bcins   v2        bcins
```

Fgiure 5: Tineola commands to determine how peers are configured

The output in Figure 5 shows the name of the channels (default case of the insurance strawman), the name of the peer installed chaincodes (only bcins in this case), some information about the current channel including the block height, and the list of channel chaincodes. Note that peers are not required to execute all the chaincodes which have been initialized to their channels. Chaincodes not on a channel cannot be invoked on that channel. Chaincodes not installed to a peer cannot be endorsed by that peer.

To get more information about previous transactions, use the `channel-history` command:

[18] The bootstrap administrator account name is `admin`, and the password suggested by Fabric documentation is `adminpw`. Almost all open source sample code reuse this password.

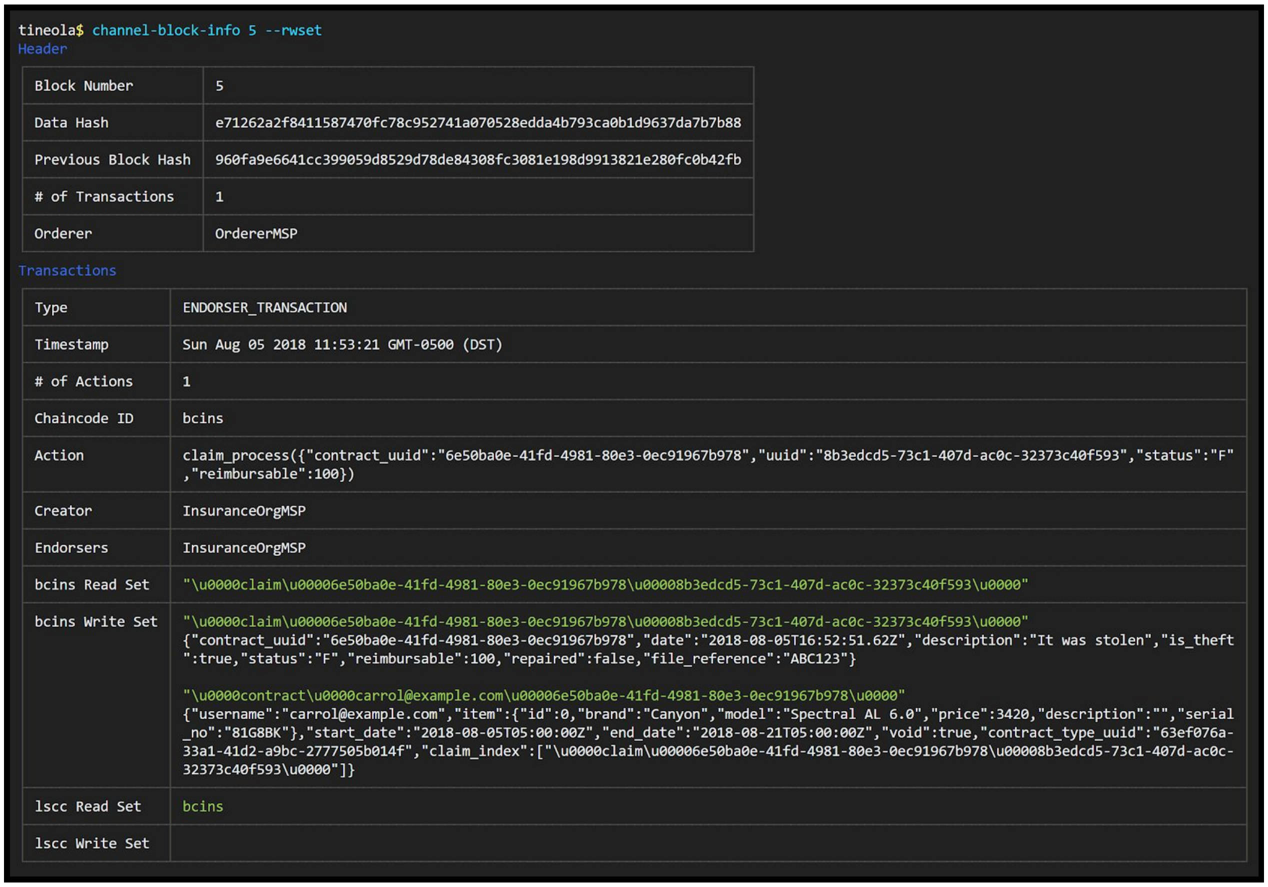


```
tineola$ channel-history 5
 Block #   CC ID   Action                                                              Creator           Endorser          Timestamp
 2         bcins   contract_create({"contract_type_uuid":"63ef076a-33a1-41d2-a9bc-277  ShopOrgMSP        ShopOrgMSP        Sun Aug 05 2018
                   7505b014f","username":"carrol@example.com","password":"pass44","fi                                      11:52:27 GMT-0
                   rst_name":"Carrol","last_name":"Example","item":{"id":0,"brand":"C                                     500 (DST)
                   anyon","model":"Spectral AL 6.0","price":3420,"serial_no":"81G8BK"
                   },"start_date":"2018-08-05T05:00:00.000Z","end_date":"2018-08-21T0
                   5:00:00.000Z","uuid":"6e50ba0e-41fd-4981-80e3-0ec91967b978"})
 3         bcins   claim_file({"contract_uuid":"6e50ba0e-41fd-4981-80e3-0ec91967b978"  InsuranceOrgMSP   InsuranceOrgMSP   Sun Aug 05 2018
                   ,"date":"2018-08-05T16:52:51.620Z","description":"It was stolen","                                      11:52:51 GMT-0
                   is_theft":true,"uuid":"8b3edcd5-73c1-407d-ac0c-32373c40f593"})                                        500 (DST)
 4         bcins   theft_claim_process({"uuid":"8b3edcd5-73c1-407d-ac0c-32373c40f593"  PoliceOrgMSP      PoliceOrgMSP      Sun Aug 05 2018
                   ,"contract_uuid":"6e50ba0e-41fd-4981-80e3-0ec91967b978","is_theft"                                      11:53:07 GMT-0
                   :true,"file_reference":"ABC123"})                                                                     500 (DST)
 5         bcins   claim_process({"contract_uuid":"6e50ba0e-41fd-4981-80e3-0ec91967b9  InsuranceOrgMSP   InsuranceOrgMSP   Sun Aug 05 2018
                   78","uuid":"8b3edcd5-73c1-407d-ac0c-32373c40f593","status":"F","re                                      11:53:21 GMT-0
                   imbursable":100})                                                                                     500 (DST)
 6         bcins   contract_create({"contract_type_uuid":"17210a72-f505-42bf-a238-65c  ShopOrgMSP        ShopOrgMSP        Sun Aug 05 2018
                   8898477e1","username":"alice@example.com","password":"pass61","fir                                      11:54:01 GMT-0
                   st_name":"Alice","last_name":"Example","item":{"id":0,"brand":"Sam                                     500 (DST)
                   sung","model":"S7","price":660,"serial_no":"FT9OG7"},"start_date":
                   "2018-08-05T05:00:00.000Z","end_date":"2018-08-06T05:00:00.000Z","
                   uuid":"b6352751-c4f2-401d-83c8-80727982d7ba"})
```

Figure 6: The `channel-history` command gives more information about previous transactions

Here we see the last 5 transactions on this channel, the chaincode they invoked, the action (usually a function name and list of arguments), a creator, list of endorsing peers, and a timestamp. The insurance strawman (similar to most example applications) uses the default Fabric endorsement policy whereby the transaction creator may be the sole endorser, as is the case for all of transactions above. Furthermore, the insurance application utilizes a common technique of passing JSON objects back and forth with chaincode. This is the prefered rich data format due to CouchDB having JSON object querying capabilities.

Even more data about past transactions can be found by using the `channel-block-info` command with the `--rwset` flag:



```
tineola$ channel-block-info 5 --rwset
Header
 Block Number          5
 Data Hash             e71262a2f8411587470fc78c952741a070528edda4b793ca0b1d9637da7b7b88
 Previous Block Hash   960fa9e6641cc399059d8529d78de84308fc3081e198d9913821e280fc0b42fb
 # of Transactions     1
 Orderer               OrdererMSP
Transactions
 Type              ENDORSER_TRANSACTION
 Timestamp         Sun Aug 05 2018 11:53:21 GMT-0500 (DST)
 # of Actions      1
 Chaincode ID      bcins
 Action            claim_process({"contract_uuid":"6e50ba0e-41fd-4981-80e3-0ec91967b978","uuid":"8b3edcd5-73c1-407d-ac0c-32373c40f593","status":"F"
                   ,"reimbursable":100})
 Creator           InsuranceOrgMSP
 Endorsers         InsuranceOrgMSP
 bcins Read Set    "\u0000claim\u00006e50ba0e-41fd-4981-80e3-0ec91967b978\u00008b3edcd5-73c1-407d-ac0c-32373c40f593\u0000"
 bcins Write Set   "\u0000claim\u00006e50ba0e-41fd-4981-80e3-0ec91967b978\u00008b3edcd5-73c1-407d-ac0c-32373c40f593\u0000"
                   {"contract_uuid":"6e50ba0e-41fd-4981-80e3-0ec91967b978","date":"2018-08-05T16:52:51.62Z","description":"It was stolen","is_theft
                   ":true,"status":"F","reimbursable":100,"repaired":false,"file_reference":"ABC123"}

                   "\u0000contract\u0000carrol@example.com\u00006e50ba0e-41fd-4981-80e3-0ec91967b978\u0000"
                   {"username":"carrol@example.com","item":{"id":0,"brand":"Canyon","model":"Spectral AL 6.0","price":3420,"description":"","serial
                   _no":"81G8BK"},"start_date":"2018-08-05T05:00:00Z","end_date":"2018-08-21T05:00:00Z","void":true,"contract_type_uuid":"63ef076a-
                   33a1-41d2-a9bc-2777505b014f","claim_index":["\u0000claim\u00006e50ba0e-41fd-4981-80e3-0ec91967b978\u00008b3edcd5-73c1-407d-ac0c-
                   32373c40f593\u0000"]}
 lscc Read Set     bcins
 lscc Write Set
```

Figure 7: The `channel-block-info` command provides detailed information on a block

All the information from before is shown for the particular block being queried, as well as the read/write set of the endorsed transaction. Object keys can be found in green. Note that the null byte `\u0000` is used as a separator by Fabric and is not a valid character in a key. Looking up full read/write set data is useful for tracing functionality of unknown chaincode or for looking for revealing information written to the chain.

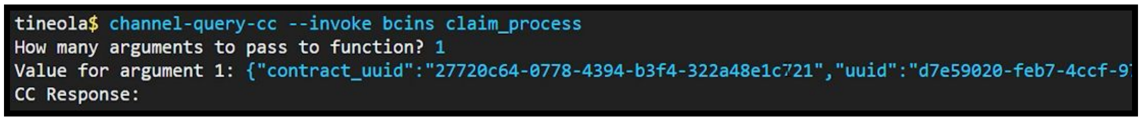
Anti-Pattern 1: Sensitive Information Disclosure

In the above example we find a common vulnerability in Fabric-based applications: sensitive information exposure. Specifically in blocks 2 and 6 the user's password is passed to the chaincode in the "password" JSON field. While passing obviously sensitive information in the clear like this is a problem, the authors of this paper recognize that real world applications need to operate on actual data which is oftentimes sensitive (hashed passwords, PI data, internal identifiers, etc). A few solutions can be envisioned:

1. Hash or encrypt sensitive information before transmitting to chaincode. This works well for applications that needs to store information long term but not directly work with it in the chaincode logic. Organizations which want to decrypt sensitive data outside the blockchain can use out of band secret sharing.
2. Move sensitive information out of band. If information is not needed by other organizations then it should not be stored on the blockchain. Having secondary,

non-shared, databases is a good practice to isolate organizational-specific data from 3rd parties. Remember, creating a private channel still exposes private data to the orderer so a channel of 1 peer is still shared data.
3. Use transient data inputs and private data collections for sensitive information. The order is never made aware of these two data sources and therefore data isolation is achieved. Remember that this data will still be shared with members of your channel.

Invoking Chaincode
Once an attacker has discovered the Chaincode calls made by the application, the next step is to abuse the chaincode logic. Tineola can be used for both querying (making chaincode read-only calls without ordering) and invoking (making read/write calls by submitting to the orderer) existing chaincode on a peer. Both are accomplished with the `channel-query-cc command` with the optional "`--invoke`" flag.



```
tineola$ channel-query-cc --invoke bcins claim_process
How many arguments to pass to function? 1
Value for argument 1: {"contract_uuid":"27720c64-0778-4394-b3f4-322a48e1c721","uuid":"d7e59020-feb7-4ccf-9
CC Response:
```

Figure 8: The channel-query-cc command can invoke chaincode

Here, a repairshop peer is invoking the claim_process function on the bcins chaincode. This function is usually used by the insurance peer to approve insurance claims before passing the claim to the repairshop for repair. In this case, the chaincode is not performing authorization allowing arbitrary peers to invoke arbitrary function calls. Note that response here is blank since the chaincode being called doesn't happen to respond with any data. Errors would be clearly shown if there was an issue during invocation of the function.

Anti-Pattern 2: Missing Authorization Controls
By communicating directly with Fabric, an attacker can bypass potential controls implemented in the user-facing Web or Thick-client application. As Fabric is inherently a shared computation platform, chaincode must be responsible for vetting incoming requests. The Fabric development community has not yet codified best practices for establishing identity within chaincode. The shim alone provides only limited capabilities with the `GetCreator()` function call. Creator IDentity [19] (CID) is a new API shipping with Fabric which aims to solve this identity problem by parsing the `GetCreator()` output into useful information like certificate subject and issuer common names. This library, however, is very new and has very limited documentation and features. Furthermore, relying on common names is dangerous in the face of CA server compromise.

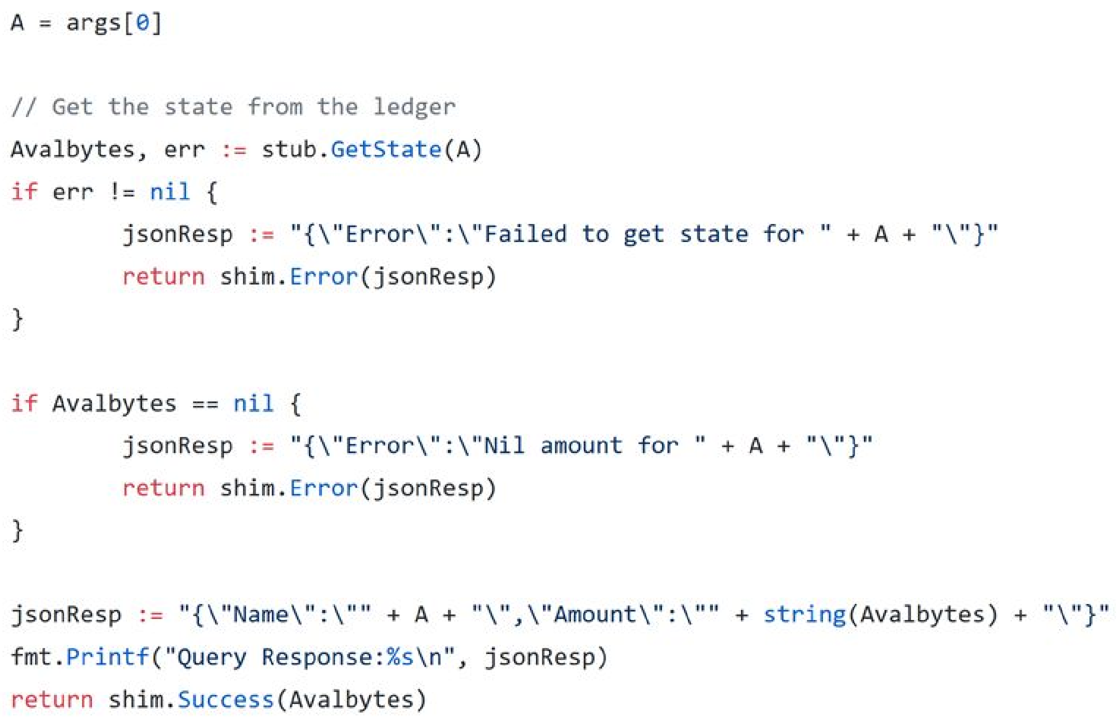
The authors suggest commiting public key fingerprints to the chaincode state and validating subsequent requests against these known keys. Additional work must be done to support secure key rotation and maintenance, but this provides far better security guarantees than relying on certificates to be vetted by a potentially compromised CA server.

Fuzzing Chaincode Outputs
Fabric deployments cannot be considered in isolation to their greater application use case. Specifically, Fabric is usually found as a backend to HTTP-based interfaces (either GUIs or APIs)

[19] https://godoc.org/github.com/hyperledger/fabric/core/chaincode/lib/cid

which end users interact. As such, Tineola has features for targeting these upstream applications via chaincode abuse. Specifically, Tineola ships with the `tineola-http-proxy` command which opens a localhost port on the attacker's machine which translates HTTP-formatted requests into chaincode function calls.

Web application scanning tools, such as BurpSuite Scanner[20], can be used to target chaincode functions for scanning and fuzzing. An example target, extracted from the example02 chaincode in fabric-sample repository[21], is shown below in Figure 9.



```
A = args[0]

// Get the state from the ledger
Avalbytes, err := stub.GetState(A)
if err != nil {
        jsonResp := "{\"Error\":\"Failed to get state for " + A + "\"}"
        return shim.Error(jsonResp)
}

if Avalbytes == nil {
        jsonResp := "{\"Error\":\"Nil amount for " + A + "\"}"
        return shim.Error(jsonResp)
}

jsonResp := "{\"Name\":\"" + A + "\",\"Amount\":\"" + string(Avalbytes) + "\"}"
fmt.Printf("Query Response:%s\n", jsonResp)
return shim.Success(Avalbytes)
```

Figure 9: Chaincode vulnerable to JSON injection

The chaincode shown above demonstrates a trivial JSON injection vulnerability in the `jsonResp` variables. Upstream applications which ingest data generated by chaincode may react unexpectedly to such injections. By scanning chaincode and committing changes back to the stateDB, Tineola can target the upstream applications through outputs of vulnerable chaincode.

Anti-Pattern 3: Trusting Chaincode Data

Unlike typical backend databases, the Fabric stateDB represents data shared across organizational boundaries. Any member organization can modify the data within the bounds of the defined chaincode. Further, vulnerabilities in Fabric configuration such as poor Endorsement policies, may make even chaincode-based controls ineffective. The authors suggest never trusting outputs from chaincode to be pre-sanitized.

[20] https://portswigger.net/burp

[21] https://github.com/hyperledger/fabric-samples/blob/release-1.2/chaincode/chaincode_example02/go/chaincode_example02.go

Anti-Pattern 4: Lack of Input Validation and Encoding in Chaincode

Chaincode should make every effort to ensure inputs match expected formats and are properly encoded. Given that the upstream client to the chaincode may be any member of the channel, it cannot be assumed that the client is performing validation or encoding on behalf of chaincode. Furthermore, once a peer or client is compromised, arbitrary data can be provided directly to chaincode by a malicious party.

Installing the Tineola Chaincode

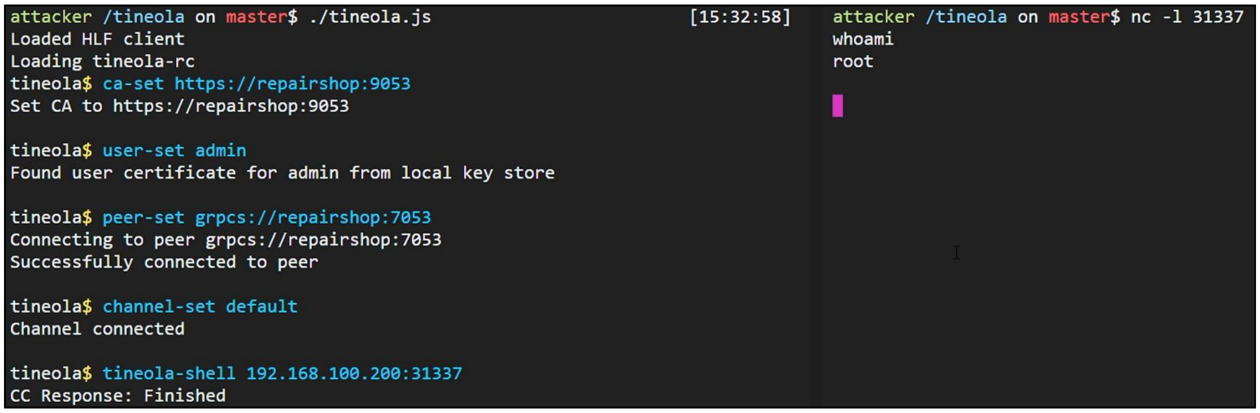
Tineola ships with specialized chaincode for executing attacks from the peer node itself. The following two parts, Pivoting Through Chaincode and Direct StateDB Manipulation, require the `tineolacc` chaincode to be installed on at least one peer and initialized onto at least one channel.

To install chaincode to a peer, the tineola user must acquire a peer Administrator certificate and private key. This is not the same as the bootstrap "admin" account; instead the Administrator is manually configured to the peer as one or more allowed certificates in the "admincerts" section of the MSP directory. Only the keys corresponding to those certificates will be allowed to install new chaincode. Once these keys have been gathered, the `user-load-keys` tineola command is used to load the keys into TIneola and the `tineola-install` command is used to deploy the chaincode to the active peer and channel.

The purpose of the attacks described below are to demonstrate the capabilities of an evil or compromised peer on the shared Fabric network.

Pivoting Through Chaincode

Fabric peers are often deployed to private networks, isolated from other network resources, with only the needed gRPC endpoints exposed. By deploying the tineola chaincode we can invoke a remote shell session and have it connect back to the attacker's machine (i.e. a reverse TCP shell). The `tineola-shell` command is used to accomplish this.



```
attacker /tineola on master$ ./tineola.js                [15:32:58]
Loaded HLF client
Loading tineola-rc
tineola$ ca-set https://repairshop:9053
Set CA to https://repairshop:9053

tineola$ user-set admin
Found user certificate for admin from local key store

tineola$ peer-set grpcs://repairshop:7053
Connecting to peer grpcs://repairshop:7053
Successfully connected to peer

tineola$ channel-set default
Channel connected

tineola$ tineola-shell 192.168.100.200:31337
CC Response: Finished
```

```
attacker /tineola on master$ nc -l 31337
whoami
root
```

Figure 10: The `tineola-shell` command enables attackers to invoke a remote shell session

Figure 10 shows a tineola on the left, invoking the `tineola-shell` command with the IP and port of the attacker's machine. The tineola chaincode then execs the `sh` binary inside the chaincode container and connects the input/output pipe to the desired remote IP/port. On the right a simple `netcat` listener is used to wait for incoming connections. After connection is established the `whoami` linux command is sent over to the container and it responds with "root" indicating the chaincode is executing as the root user.

Note that, chaincode is isolated from the peer node via an Ubuntu container. The container, however, is not fully isolated from the peer's network. It is possible to scan the network, using `nmap`[22] for example, to discover additional peers and services accessible from the peer's perspective.

Anti-Pattern 5: Unrestricted Chaincode Container Networks

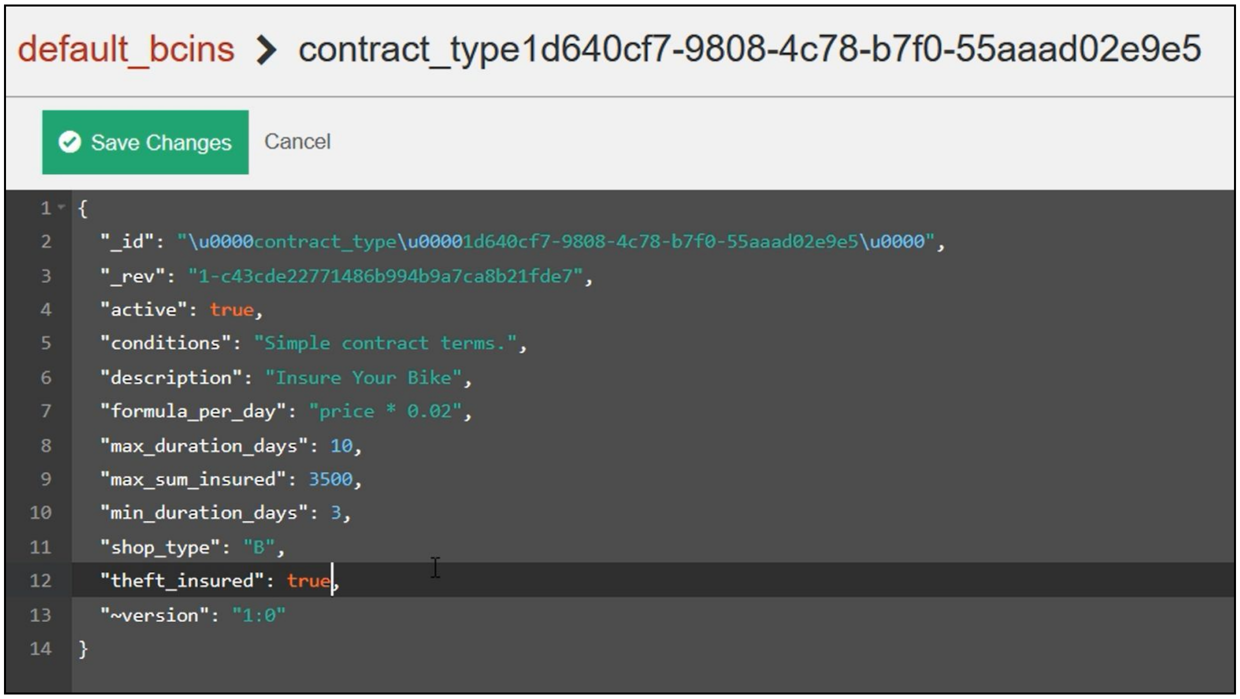
By default in Fabric all chaincode containers have unrestricted access to network resources. Fabric version 1.2 does have configurations[23] to limit the docker container capabilities, but this feature is not documented and is likely not supported due to impacting its ability to communicate to the peer node. Instead, steps should be taken to firewall access to sensitive resources such as CouchDB from non-local peers.

Direct StateDB Manipulation

While the state of a blockchain is idealized as the sum of all transaction outputs, the peers really use databases to represent the set of current state values. This stateDB is therefore the ground truth for that peer with respect to chain data. If an attacker can modify the data stored in this database, the chaincode will be unable detect the change. This is of particular interest because chaincode is idealized as a description of allowed state changes. By bypassing chaincode, we can invalide invariants described by code. For example, we may write a chaincode that represents a deed to a property and say, "once a deed has been created, the physical address of that deed never changes." By modifying the value in the stateDB, subsequent queries to the chaincode would return tainted data even though the chaincode is correct and no bugs exist. Furthermore, no chain history would exist demonstrating the change. Together, this attack breaks the core promises of blockchain: immutability and auditability.

For Fabric, this attack is most likely to occur when CouchDB is in use as the stateDB backend. CouchDB does not have authentication enabled by default and even the fabric-sample repository suggests exposing the CouchDB port externally on the peer host[24]. Even when the network isolates the CouchDB instance from the network, Tineola's chaincode can proxy traffic from the peer's internal network to the attacker's machine using the `tineola-ssh-proxy` command. Once connection has been established, CouchDB's web interface[25] can be used to directly modify JSON documents. See Figure 11 for an example document interface.

[22] `nmap` can be installed using the `apt` package manager. If network access is not available, the `tineola-http-drop` command can be used to drop the `nmap` binary onto the container image.
[23] https://github.com/hyperledger/fabric/blob/release-1.2/core/container/dockercontroller/dockercontroller.go#L144
[24] https://github.com/hyperledger/fabric-samples/blob/release-1.2/basic-network/docker-compose.yml#L97
[25] Usually located at the `/_utils` path


default_bcins > contract_type1d640cf7-9808-4c78-b7f0-55aaad02e9e5
Save Changes
Cancel

```
{
  "_id": "\u0000contract_type\u00001d640cf7-9808-4c78-b7f0-55aaad02e9e5\u0000",
  "_rev": "1-c43cde22771486b994b9a7ca8b21fde7",
  "active": true,
  "conditions": "Simple contract terms.",
  "description": "Insure Your Bike",
  "formula_per_day": "price * 0.02",
  "max_duration_days": 10,
  "max_sum_insured": 3500,
  "min_duration_days": 3,
  "shop_type": "B",
  "theft_insured": true,
  "~version": "1:0"
}
```

Figure 11: Couch DBs interface can be used to directly modify JSON documents

Note that modifying the stateDB does not propagate changes by itself to other peers. Only the peer with the compromised stateDB would reflect the changes made by direct manipulation. In the case of the default, any one peer policy, this doesn't matter since all subsequent requests taking advantage of the change can be directed at the compromised peer. In the example above, the `theft_insured` flag is never modified by the chaincode after the contract type is created. But, other parts of the same JSON document can be updated. When such updates occur, the entire JSON document must be read from the stateDB and then written back[26] with the desired and allowed change. Any tainted data read off from the stateDB would then persist into the endorsed write set and be propagated to other peers. An endorsement policy of 2 or more peers would detect this change since the write set of the peers would differ and be rejected by the orderer.

Anti-Pattern 6: Unauthenticated CouchDB

The CouchDB container provided by Fabric does support authentication, but it is disabled by default. In fact, the fabric-sample repo contains a comment saying "Populate the `COUCHDB_USER` and `COUCHDB_PASSWORD` to set an admin user and password for CouchDB. This will prevent CouchDB from operating in an "Admin Party" mode." Users who reuse existing examples without updating values will miss this advice which results in highly insecure Fabric deployments. Further, this username and password must then be provided to Fabric peers as well, either in their docker environment variables (`CORE_LEDGER_STATE_COUCHDBCONFIG_USERNAME` and `CORE_LEDGER_STATE_COUCHDBCONFIG_PASSWORD`) or in the peer configuration YAML file.

[26] This is due to Fabric abstracting changes as key/value assignment where the value is often an entire JSON document. There are no primitives for modifying individual JSON keys. Importantly, this use of JSON is encouraged by Fabric due to Shim's support for the JSON query capability provided by CouchDB.

Note that solving authentication for CouchDB only prevents unauthorized access, and does not limit peers from modifying their own stateDBs in order to bypass chaincode logic. The endorsement policy must be configured such that the possibility and impact of rouge and compromised peers on the network is minimized.

Conclusion and Future Work

The blockchain craze has reached the security community. The focus has been crypto-currencies and public blockchains. Meanwhile, enterprises have developed systems based on these technologies. As applications reach the release phase, the security debt is showing. In developers' rush to the market, security has been an afterthought in both the blockchain platform and the applications built on top of them. Blockchain is here to stay. The security community should start getting involved in the design and implementation of these systems. While it might be late to get involved in the design of current generation of blockchain platforms, we can achieve some semblance of security and cut our losses. By learning what's wrong now, we can prevent the same mistakes in the next iteration.

There is a lot to be done in this space. On the offensive side, the tool can be expanded to include pivoting, compatibility with Metasploit, and support more chaincode invocations. Defense could use secure standards for Fabric deployment and operation, chaincode development, and network operation. Other enterprise blockchain platforms such as Quorum[27] are good targets for security research.

[27] https://github.com/jpmorganchase/quorum