Global Permission Chain

Software Design Document

September 10, 2020

James P. Webster, Research Engineer, Penn State University, jpw5504@psu.edu

Executive Summary

As we increasingly rely on computers to make decisions and take action automatically, or give us guidance on the best course of action, it becomes increasingly important that the computers thwart attempts to gain unauthorized access. Computers usually have a list of authorized users, along with a representation of their password and a description of what each user has permission to do. Populating this list, however, remains a difficult problem that frequently requires human oversight.

In this document, we introduce the Global Permission Chain (GPC). The goal of this system is to give a computer the ability to evaluate a user it has never seen before and automatically derive which permissions the user should have. The basic strategy is for the user to provide documentation backing up their claim to a certain level of permission, and for the computer to check a distributed global ledger to confirm that the documents were published and never revoked.

In order to develop the necessary designs and prove out the concept, we have built a working prototype of the GPC. The prototype uses a Hyperledger Fabric blockchain to register digital certificates, which users can use to gain access to a protected system. Computers can learn about the state of blockchain by listening to broadcasts from an entity called a Relay. These broadcasts are highly efficient and would be suitable for computers with very limited bandwidth. Such computers would be analogous to GPS devices, but instead of keeping track of their location, they would keep track of the information necessary to evaluate a user's request for permission.

Our design optimizes for high scalability, reliability through distributed control, and minimal bandwidth requirements for the end-users.

Our results may have commercial applications well beyond tactical logistics. We see potential applications for this technology in situations where manual access control setup is inconvenient or infeasible, for example with Internet of Things (IoT) devices or autonomous systems. Another application could be situations where personnel are in a dynamic environment where it is hard to predict who they will work with, or what equipment they will work with. These personnel may have limited opportunity to communicate with the outside world to ask questions about the history of the people and equipment they are working with, and the level of access they should grant.

Contents

Introduction

In the future, we will increasingly rely on computers to make decisions and take action without direct human supervision. These computers could be vulnerable to clever attackers, who try to feed them false stories about what is going on in the outside world. You can think of such a computer as being like a person who must make decisions from inside a bunker deep underground. This person will make the best decisions possible, based on the inputs they are given, but they depend on the picture their instruments and communication links present to them. They cannot leave the bunker to verify the situation in person.

Events in the outside world cannot come into the bunker, but messages can enter the bunker. It is possible to give these messages cryptographic properties that provide direct evidence that a certain event occurred outside of the bunker. The computer can correctly reason that if the event had not happened, it would be impossible for the message to have the properties that it has.

The Global Permission Chain (GPC) acts as a foundation to help computers evaluate messages coming in from the outside world and make good judgements. Just as GPS helps a self-driving car know where it is in the world, GPC helps a computer decide whether a statement about the outside world is true or false.

GPC is an example of a Public Key Infrastructure (PKI) with the following distinctive properties:

1. Focus on verifying permission rather than identity.
2. Distributed control over the infrastructure. No single points of failure/trust/control. No single central authority.
3. Minimize bandwidth required by end-user. No uplink required. Works like a GPS receiver.
4. Learn about certificate revocations quickly. (Minutes, not months.)
5. System can promise end-users that it will protect them from having unachievable resource requirements by providing a way to limit how much traffic the system processes.

Using GPC, clients in a disconnected, intermittent, low-bandwidth (DIL) situation can nevertheless benefit from the trust-minimizing properties of a blockchain to obtain the crucial information they require to authenticate and check authorization.

Research Approach

A “disconnected blockchain” may seem like a contradiction in terms, since blockchains require that multiple peers connect to each other and constantly interact to share, validate, store and extend a data structure. When the Bitcoin whitepaper introduced the idea of using a blockchain to send electronic payments, it also introduced a strategy for helping people with connectivity

use the blockchain. This strategy appeared under the heading “Simplified Payment Verification” or SPV.

In the SPV concept, a disconnected user would be able to verify that a payment occurred without downloading and verifying the entire blockchain. Instead, the user would download just the block headers, which contain a Merkle root summarizing the rest of the block.

Our approach is similar to the SPV concept, but we are applying it to the problem of access control. We will create a blockchain that will give users who are disconnected from the broader community the cryptographic evidence they need to reason about what level of permission they should grant to someone who can produce a particular digital signature.

After considering several candidate frameworks for building our blockchain application, we chose Hyperledger Fabric because it is open source, well documented, has good support in industry, is relatively mature, and is very customizable.

We built a prototype Global Permission Chain to refine and test the idea and develop any necessary technology. We invented message formats, protocols, workflows and software as necessary to produce a working system.

We have taken inspiration from several technologies developed by the Bitcoin community. GPC stays structurally similar to Bitcoin in many ways. For example:

Bitcoin	GPC
Proves that a transaction was broadcast	Proves that a certificate was published.
Proves that a transaction has a valid chain all the way back to a mined coin.	Proves that an attribute has a valid chain back to a root attribute.
Proves that coins outputted from a transaction have not been spent yet.	Proves that a certificate has not been revoked yet.
“Correct” blockchain is the one with the most work.	“Correct” blockchain is the one Hyperledger Orderers agree on, based on RAFT consensus algorithm.
SPV User downloads the block headers.	Disconnected User downloads the Relay Block Messages.
Disconnected User downloads Bloom Filter representation of the block to see if a coin was spent in the block. (This protocol is called Neutrino.)	Disconnected User downloads the Bloom Filter Messages to see if a certificate has been revoked.

Overview

Hyperledger Fabric Network Overview

At the core, the GPC is a blockchain based on the Hyperledger Fabric platform. This blockchain provides us with an append-only ledger under the distributed control of several peers. Our peers are physically separate computers that do not share any infrastructure except the internet. These peers each store a copy of the ledger and use a service called an Order to come to consensus about the current state of the ledger using the Raft consensus algorithm. Every 60 seconds, the set of Orderers add a new block to the chain, and this block can contain transactions that add items to one of two sets:

1. The set of published certificates
2. The set of revoked certificates

Validator-Signer Overview

Our client wishes to know whether an x509 certificate enables the bearer to have a certain type of access. This will only be true if:

1. The certificate has been published
2. The certificate has not been revoked
3. The certificate establishes that the bearer should be granted access

The client uses a software component called “Validator-Signer” to examine a certificate, determine whether or not it has been published, and determine whether it has been revoked.

Relay Overview

Our low-bandwidth clients will not interact directly with the blockchain. We do not want to force them to become a fully validating Hyperledger peer, which would be costly in terms of administrative and communication overhead. Rather, we introduce the “Relay” which will observe each Fabric Block, discard information that is not helpful to the low-bandwidth client, compress what remains, and broadcast the compact representation of what was in the Fabric Block in a format called a Relay Block.

Relays are the only point of contact between the client and the blockchain, and clients to some degree do depend on access to an honest relay. Two facts strictly limit a relay’s ability to cause problems by failing to follow the rules:

1. The client has the ability to listen to more than one Relay.
2. There is only one way to compute each Relay Block correctly. All honest Relays will arrive at the same result.

Relays digitally sign the blocks they publish with publicly known certificates. The GPC clients can pick trustworthy Relays based on the historical performance of the relay and knowledge of who has control of the Relay. The Relay appends the Relay Signature to the Relay Block, creating a message called a Relay Message, and this is what the Relay forwards to the Client.

Relays can also listen for the broadcasts that other Relays create. This gives them an additional Relay Signature, which they can add to the Relay Message that they send out. Clients can listen to a single Relay while also getting enough information to verify that several Relays have signed the same message.

Permission Marshal Overview

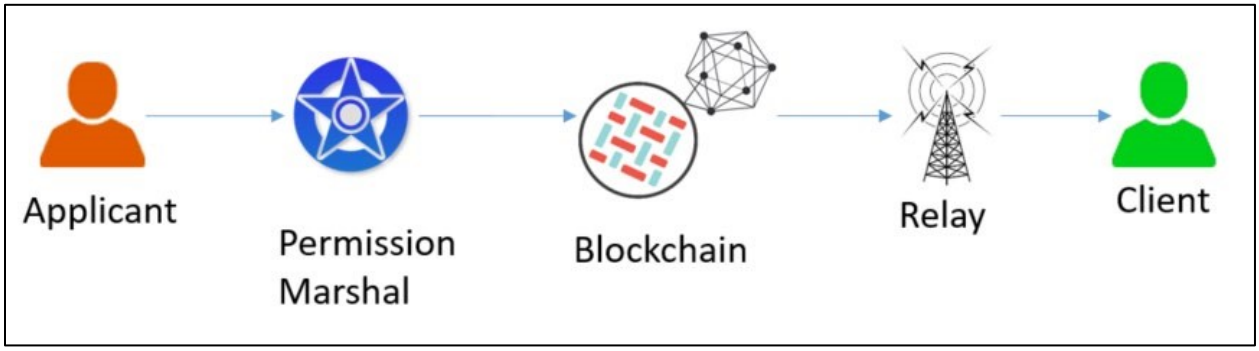
Permission Marshals are the role in the system that can publish a transaction to the Hyperledger blockchain. Users who wish to publish an x509 certificate (Applicants) or a revocation message must send it to a Permission Marshal. The Permission Marshal intermediates between the Applicants and the Hyperledger Network, performing several functions such as:

1. Eliminating the need for Applicants to implement their own Hyperledger client.
2. Filtering out requests that do not make sense so that they do not make it to the blockchain (e.g. malformed or redundant X509 messages, or unauthorized revocations.)
3. Imposing restrictions on the rate at which certificates can be published.
4. Bundling multiple certificates into a Merkle Tree, calculating the Merkle Tree Root, and sending this to the blockchain, which creates more privacy and places less demands on the globally shared resources.
5. Creating logs of certificates that the Permission Marshal publishes and sharing the log with authorized users.

Access Control Policy Overview

Users of the system who wish to coordinate on access control must define a common Access Control Policy. They must agree on a common language that defines what attribute text the certificate must contain to qualify it for a particular type of access.

Architecture Diagram:



Background Topics

Secure Hash

A Secure Hash is a function that takes any arbitrary document as an input and outputs a unique checksum that looks like a random number. This checksum is a compact representation of the original document, because it is not feasible to generate a different document that will hash to the same checksum.

Blockchain

A blockchain is an append-only list of records that grows when a member of a set of peers **most** of the peers follow the rules.
"Block Height" is a way to identify a single block, by counting the number of blocks that come before it in the chain.

Smart Contract

A smart contract is a computer program that dispositions something valuable appropriately based on:

- A set of agreed-upon rules
- Facts about what events have occurred

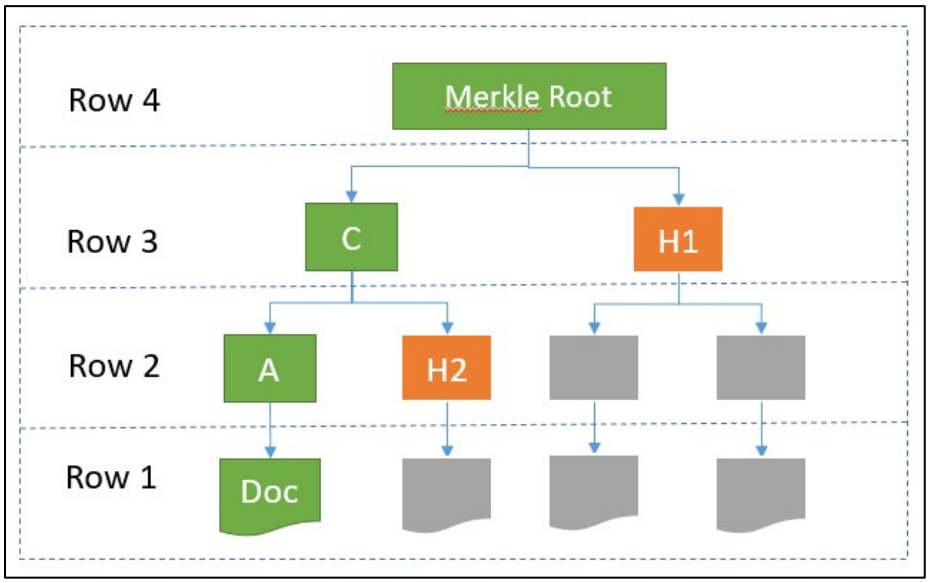
Once the computer takes action, it is difficult or impossible for participants in the smart contract to reverse the computer's action. Just like with a legal contract, users enter into a smart contract to precisely define and formalize a relationship and make sure that their counterparty lives up to their side of the agreement.

Smart contracts need to have a high security design, because one of their fundamental jobs is to prevent participants from taking action that they are motivated to take. A single hole can make a fence ineffective; a single vulnerability can defeat the purpose of a smart contract by giving the participants a way to trick the computer into taking the wrong action. Smart contracts rely heavily on digital signatures to provide evidence that the participants agree with some statement or approve of some proposed action.

Merkle Trees

A Merkle Tree is a data structure that organizes a set of documents in such a way that it is possible to prove that the document was included in the Merkle Tree using a very small proof.

Merkle Tree Structure:



This diagram explains the structure of a Merkle Tree. Row 1 represents the “Leaves” of the tree, or the documents that are included in the tree. To get Row 2, you hash each document in Row 1. To get Row 3, you start on the left side and first concatenate each pair of hashes in Row 2, then hash the concatenated result. If Row 2 has an odd number of elements, the rightmost element in Row 3 should just be the hash of the last element of Row 2. Repeat this step as many times as necessary until you generate a row with only a single hash. This hash is the “Merkle Root”.
The information you need to verify the inclusion of document “Doc” in this Merkle Tree includes:

1. The original document “Doc”
2. The Merkle Root
3. The partner-node hashes H1 and H2

Then verify as follows:

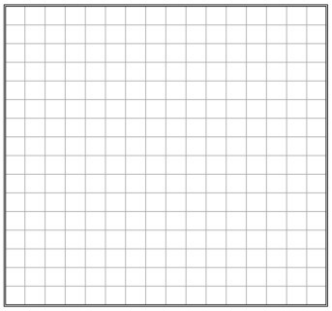
1. Let A = Hash of ‘Doc’
2. Let B = Concatenate A + H2
3. Let C = Hash of B
4. Let D = Concatenate C + H1
5. Let E = Hash of D
6. Compare E with the Merkle Root. If they are the same, then you have proved that the document is in the Merkle Tree.

Bloom Filters

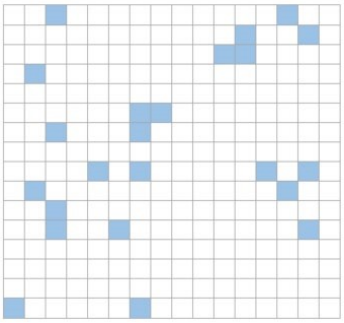
A Bloom Filter is a data structure that organizes a set of documents in such a way that it is possible to prove that a document is **not** included in the Bloom filter using a small proof.

To understand the operation of a Bloom Filter, consider a grid of cells:

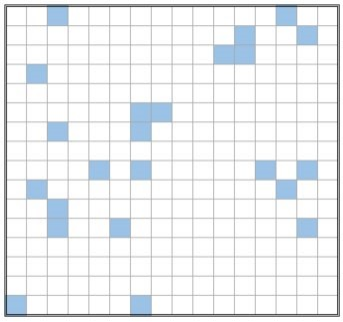
Empty Bloom Filter:



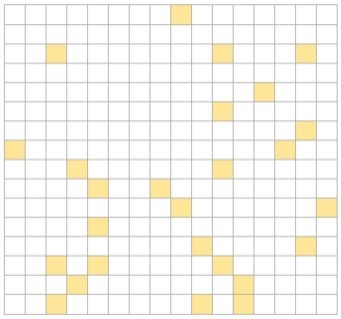
In order to add a document to the bloom filter, compute the hash of the document. Then map this hash to corresponding cells in the bloom filter. For example, our document might correspond to the following blue cells:



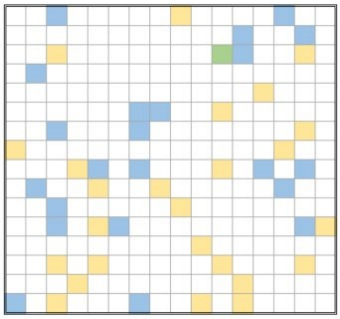
Add this document to the bloom filter by changing all of the corresponding cells in the bloom filter blue like this:



Add a second document in the same way. Hash the second document to get a distinctive pattern of cells:



Add the colored cells to the bloom filter:



Now there are two documents in the Bloom filter – one represented by blue cells, the other by yellow cells. One square is colored green because it was colored blue by the first document, and then colored yellow by the second document. In reality, the Bloom Filter cells do not have colors; they have a 1 or a 0:

0	1	0	0	0	1	0	1	0	0	1	0	0	0	1	0
1	1	0	1	0	0	1	0	0	0	0	1	1	1	1	1
1	0	1	1	1	0	0	0	1	1	0	0	0	1	0	0
0	0	1	0	1	0	0	0	0	1	0	0	1	0	1	1
0	1	0	0	0	1	0	1	1	0	1	0	0	1	1	0
0	1	0	0	0	0	0	0	0	1	1	0	1	0	0	0
1	0	1	1	1	1	1	0	0	1	0	1	0	0	0	1
1	1	0	1	1	0	0	1	0	0	1	0	1	1	0	1
0	1	1	0	0	0	1	1	0	0	0	0	1	0	1	1
0	1	1	0	0	1	0	1	1	1	0	1	0	0	0	0
0	0	0	1	1	0	0	0	0	1	1	0	1	1	0	1
0	0	0	0	0	0	1	0	1	0	0	0	1	1	1	0
0	0	1	0	0	0	1	0	1	1	0	0	0	0	0	1
1	0	0	1	0	0	0	0	1	1	1	1	0	1	1	0
1	1	0	0	0	0	0	1	1	0	1	1	1	1	1	0
0	1	1	0	1	0	1	0	1	1	1	0	0	0	1	1

To prove that a document was **not** added to the bloom filter, hash the document and examine the bloom filter for all of the cells that you would set to 1 when adding the document to the filter. If any of those cells are zero, it proves that the Bloom Filter does not contain the document.

If all of the cells that the document corresponds to are equal to 1, then the Bloom Filter *may* contain the document. There is a small chance that the filter does not contain the document, but unfortunately, the cumulative result of some other group of documents is producing a false positive. The chances that this will happen depend on the size of the Bloom Filter and the

number of documents that have been added to it. In round numbers, if you want to add 1 million documents to the Bloom Filter, and you want a positive inclusion test to be wrong no more than 1 time out of every million documents you test, you will need to use a Bloom Filter with about 29 million cells. In file form, this is about 3.4 megabytes in size. For a benchmark, if you MD5 hashed 1 million documents and transmitted the list of hashes, the size of the file would be 16 megabytes. Therefore, using the Bloom filter requires only 21% of the space required by the MD5 hash list.

X509 with Extension Fields

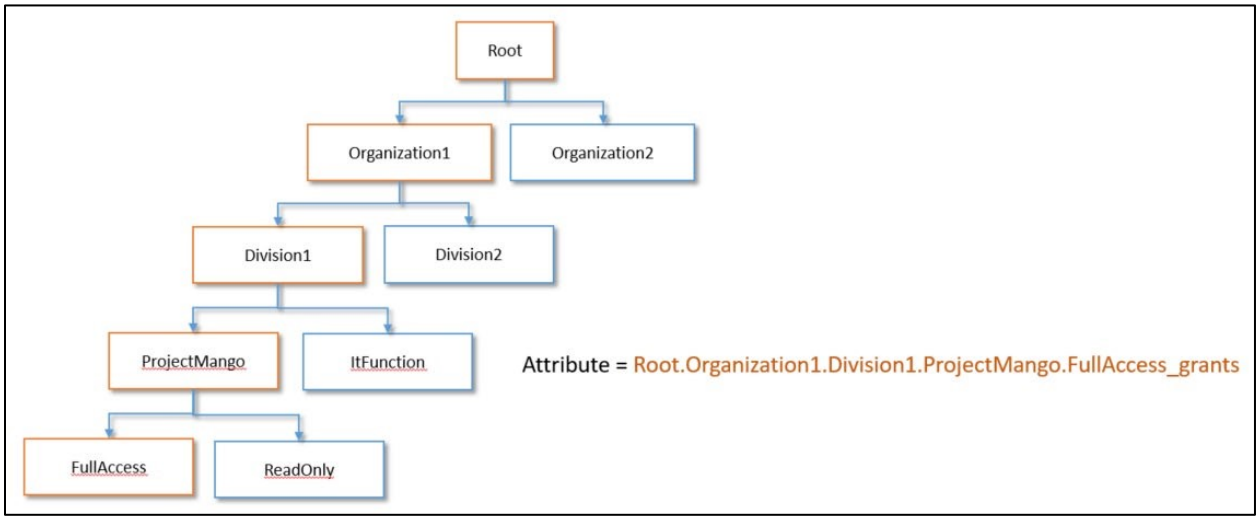
In our system, we use normal X509 certificates with two extension fields:

Attribute (OID 1.3.6.1.5.5.7.10)	Will contain a string describing what attribute the certificate confers to the bearer.
U2F Hash (OID 1.3.6.1.1.15.2)	Will contain a hash of a certificate that can be used as part of Universal 2nd Factor (U2F) authentication.

Attributes:

"Attribute" values correspond to nodes on a hierarchical access control graph that could, for example, mirror the organizational structure that is using it.

Example Attribute Tree:



The X509 Attribute Extension field represents attributes as a period-delimited list of identifiers that define a path through a hierarchy from the attribute that the certificate confers back to the root of the hierarchy.

If the attribute has the authority to grant permission to someone else by signing an X509, then the attribute will have the suffix "_grants". Attributes can only grant an attribute that is **longer**, which is to say, at least one level down the hierarchy.

Attribute Child Example:

Root.Organization1.Division1_grants can grant these attributes:

- Root.Organization1.Division1.ProjectMango
- Root.Organization1.Division1.ProjectMango_grants
- Root.Organization1.Division1.ProjectMango.FullAccess
- Root.Organization1.Division1.ProjectMango.FullAccess_grants
- Root.Organization1.Division1.ProjectMango.ReadOnly
- Root.Organization1.Division1.ProjectMango.ReadOnly_grants
- Root.Organization1.Division1.ItFunction
- Root.Organization1.Division1.ItFunction_grants

FIDO/U2F:

FIDO is a protocol for standardizing interaction with a hardware-based second-factor authentication token. GPC supports the use of a FIDO device to act as a second form of authentication.

Example FIDO U2F hardware:

Permission Chain (PCN) File Format

The PCN file is the standard format the Global Permission Chain uses to provide proof of permission of some kind. This file has the following structure:

1. A list of X509 certificates, in PEM format, separated by newline characters. The first certificate in the list contains the attribute that the PCN file is conferring. A chain of certificates follows, concluding with the Root certificate; each certificate is digitally signed by the certificate below.

2. A JSON Object with:

a. A list of Merkle Proofs. Merkle Proofs are in the same order as the PEM files above, and each Proof shows that the associated X509 PEM was indeed published on the blockchain. Each Merkle Proof contains:
    i. “height” = The block height in which the X509 was published
    ii. “numleaves” = The number of leaves in the Block-Level Merkle Tree.
    iii. “index” = The number of leaves that are to the left of the X509 in the tree.
    iv. “hashes” = A list of Partner Node Hashes needed for Merkle Proof Validation, starting with the hash that is farthest away from the Merkle Root and proceeding up the tree. The Merkle Root Hash is included at the end as the final hash, even though it does not have a partner.
b. Extra data as necessary for validating a particular request, demonstrating possession of private keys, etc.

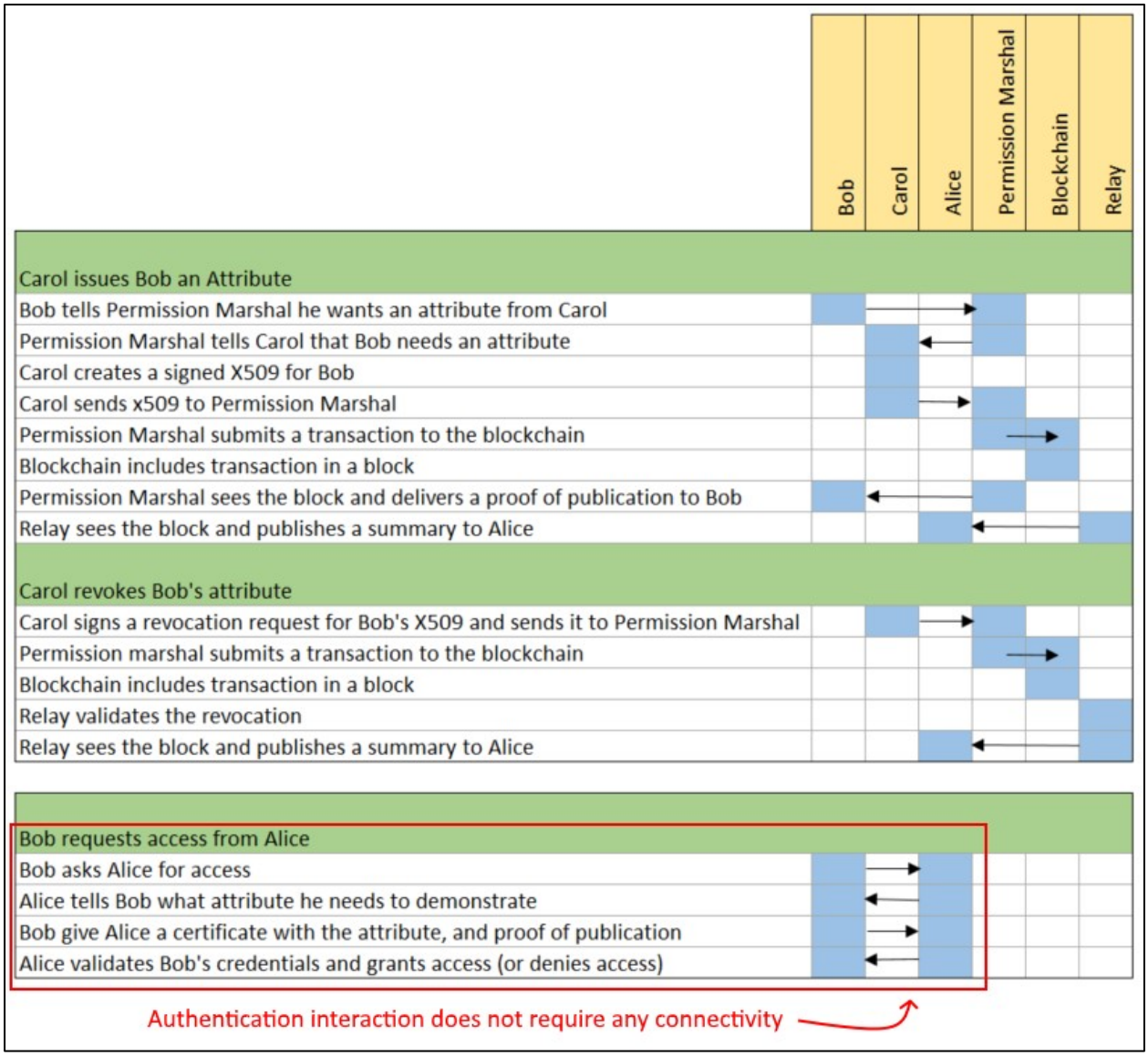
Sample PCN File:


```
-----BEGIN CERTIFICATE-----
MIIDyTCCArGgAwIBAgIUCQkgnMVRWn07RGFL1AAozSs5fwwwDQYJKoZIhvcNAQEL
BQAwXzELMAkGA1UEBhMCVVMxCzAJBgNVBAgMAlBBMRYwFAYDVQQHDA1TdGF0ZSBD
b2xsZWdlMQ0wCwYDVQQKDARSb290MQ0wCwYDVQQLDARSb290MQ0wCwYDVQQDDARS
b290MB4XDTIwMDIxMjE4MDU0NVoXDTIxMDIxMTE4MDU0NVowXzELMAkGA1UEBhMC
VVMxCzAJBgNVBAgMAlBBMRYwFAYDVQQHDA1TdGF0ZSBDb2xsZWdlMQ0wCwYDVQQK
DARSb290MQ0wCwYDVQQLDARSb290MQ0wCwYDVQQDDARSb290MIIBIjANBgkqhkiG
9w0BAQEFAAOCAQ8AMIIBCgKCAQEAw0s5T0R5EASJ+tOXILSgO40Wvkzyq/86Erbh
WgRPnAG3ppVYJts+b46YyL/4eSvDMAtLgMlyI5oLExr9L56v7WM7Ck7OEsmLrV3q
pvctf2T+SLYcDUB3TUDsXatSRizWtthi9UMayeKAxgBoromfKS7oFY7UNhM/aSWd
SmfdvTSCkMqdHNKZA2od7MikAgMP4DlK/l+OecAP/hLnh4QPB1ZF18+UvSyoaSbX
9D7VFpe/Sfl8/U9Of9m39eWmvmq8aFmpNCGGE6mjXEqP9bT/oklTuJMFZTI7omPS
UMIR+e00f6bbAeqNX8XUt4c2D/hgSoCbCP7EtzlimUq2VfFx7wIDAQABo30wezAY
BgcrBgEFBQcKBA0MC1Jvb3RfZ3JhbnRzMB0GA1UdDgQWBBQR5UHT7qy4mu+Gs5AG
cuqU0xmouTAfBgNVHSMEGDAWgBQR5UHT7qy4mu+Gs5AGcuqU0xmouTAPBgNVHRMB
Af8EBTADAQH/MA4GA1UdDwEB/wQEAwIBhjANBgkqhkiG9w0BAQsFAAOCAQEAuZks
zZ8PosSPzf8QjDaUOZShPEqmhtiwqcTHIYMFcH/olf9iSWP8uLqMIkFO58uc42YZ
f9KzaQmb8p8Pzq9W9A0a28lx/bR4X3PXh53YEspqJR8ssHypsjaEFtiKhTdKSKfA
F+OnXYv0jumOO5vF8wNhBKANiGLw1adM+UJTmaJrYztYJ4MkGMHzUltTdJFSRUOl
ovfl0smtvK4H94exFxX2rkzbTfurIstSuS+Cs7HaLmXsEc5mYnAD5xFsEAvlAiDC
tv8fjwMvk09ZB/kvmGIdevJaHgJZJ2je0vKnzOj73Yvq30PEEZk+6YxxbsO+XFb8
Z4OjWivvZhu5S0X9aQ==
-----END CERTIFICATE-----
{
  "proofList": [
  {
    "height": 0,
    "numLeaves": 1,
    "index": 0,
    "hashes": [
      "b8FBjhxq3LmFHtNpRyn7hneAKG\/OXujZ61IfM4Po3ic="
    ]
  }
  ]
}
```


Use Case Workflow

The following diagram describes the workflow for a typical use of the GPC. In this workflow, the names Bob, Carol and Alice personify the different roles. Bob will request some kind of access from Alice. The GPC must prove to Alice that Carol has given Bob an attribute that would qualify Bob for access. The GPC must also prove to Alice that Carol has not revoked the attribute from Bob.

Note that Alice does not need a connection to the blockchain while she is authenticating Bob's request for access while they are operating in a disconnected, intermittent or low-bandwidth (DIL) environment. Furthermore, while Alice does receive messages broadcast from the GPC, she never sends messages back. Alice only sends messages locally to Bob. This is a key feature of our approach, and it not only supports the needs of disconnected users like Alice, it also makes the system globally scalable.



Detailed Description of Software Components

Hyperledger Architecture Detail

Hyperledger splits the task of constructing a blockchain into different roles that influence each other by passing messages defined by the Hyperledger protocol.

Peers:

Peers run programs called Chaincode to convert input values into a packaged set of changes to a database called a Ledger. After the Peer runs Chaincode and calculates the change set, it endorses the change set with a digital signature and sends it to an Orderer.

Orderers:

Orderers receive change set packages, batch them into a Block and append the block to the end of the Blockchain. Orderers use a consensus algorithm called Raft to decide which Orderer should generate the next block. After an Orderer produces a block, the Orderer broadcasts the block to all of the Peers. The Peers evaluate the transactions in the block. If a transaction is valid and has enough endorsements from enough Peers, each Peer will apply the change set to its local copy of the Ledger. If the transaction is invalid or does not have enough endorsements, each Peer will ignore it.

Clients:

Clients connect to Peers to submit transactions for publication to the Blockchain. Clients can query the Peer for the state of the Ledger or the contents of blocks. Clients can also subscribe to receive events, such as announcements that new block is available for download. In the GPC, there are two types of clients: Permission Marshals and Relays.

Hyperledger Fabric Blockchain Implementation Detail

With the configuration we implemented, Hyperledger runs on a network of two computers playing four roles:

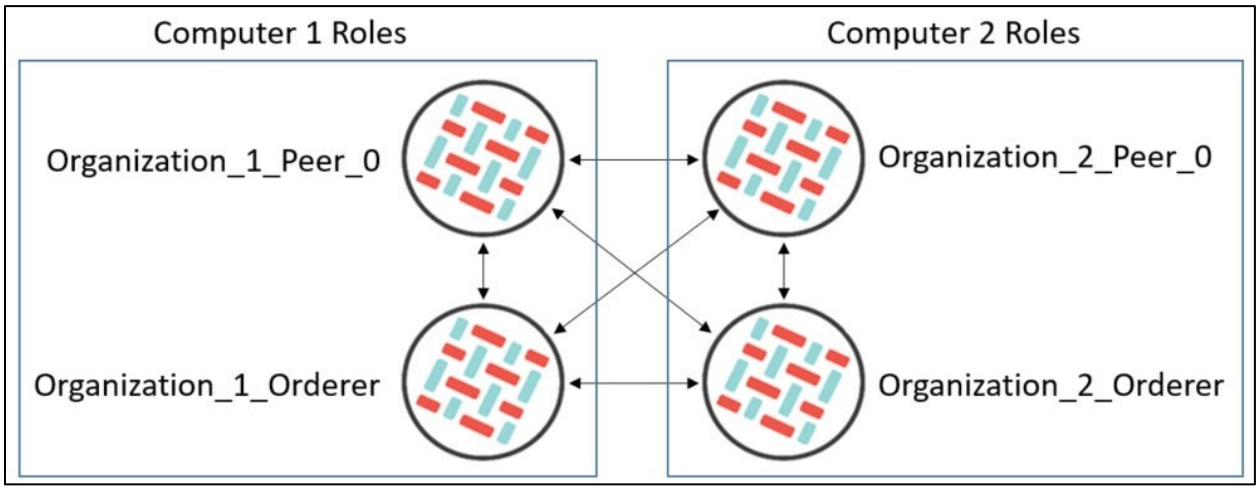
Computer 1 Roles:

1. Organization_1_Peer_0
2. Organization_1_Orderer

Computer 2 Roles:

3. Organization_2_Peer_0
4. Organization_2_Orderer

Hyperledger Fabric Computers and Roles:



Instantiating Hyperledger Fabric

Instructions and scripts for how to set up the development environment, build the project, and install the Hyperledger network are available in the “build-install-run.md” file in the “onrblockchain” repository.

Fabric runs as a set of Docker containers, launched by docker-compose on each computer, based on configuration in the “docker-compose.yml” file. This file configures hosts, services, ports, TLS keys, mappings to local files, and sets several environment variables that the blockchain uses when it starts up.

Here is a list of the containers that will start up:

Computer 1 Containers:

- ca1.example.com = Organization 1 Certificate Authority
- orderer0.example.com = Organization 1 Orderer
- peer0.org1.example.com = Organization 1 Peer
- cli = Command Line Interface client for controlling the network

Computer 2 Containers:

- ca2.example.com = Organization 2 Certificate Authority
- orderer1.example.com = Organization 2 Orderer
- peer0.org2.example.com = Organization 2 Peer
- cli2 = Command Line Interface client for controlling the network

After the containers start, the startup script on Computer1 creates a Channel called “mychannel” and then joins Organization_1_Peer_0 to the Channel.

The startup script on Computer2 fetches channel configuration data for “mychannel” and then joins Organization_2_Peer_0 to the channel.

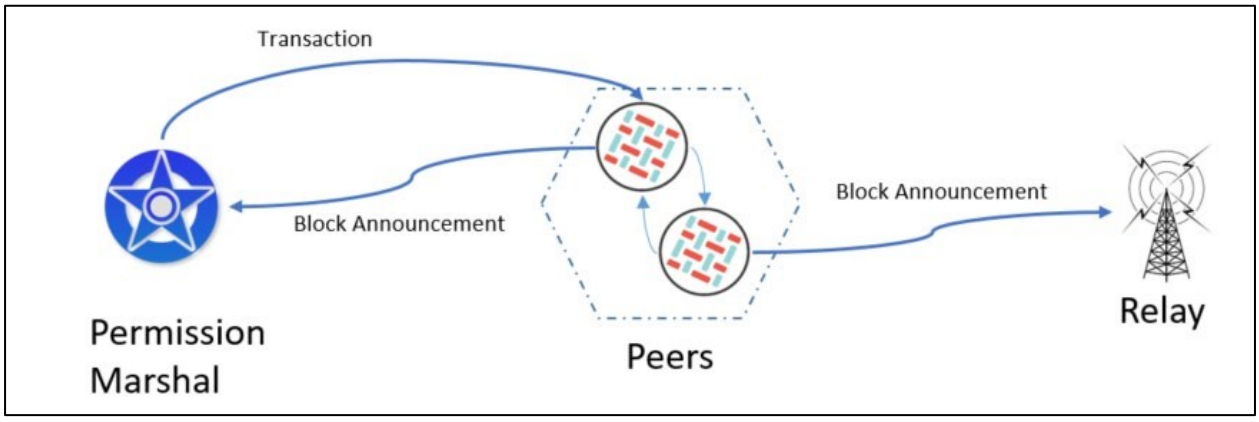
After Organization1 creates the channel, it installs the chaincode called “pubcc.go” and then initializes the chaincode, passing in a list of X509 certificates to the constructor. These X509 certificates will be the “Root” certificates and can issue ANY attribute in the Global Permission Chain.

After Organization 1 installs and instantiates the chaincode, Organization 2 installs the chaincode.

Hyperledger Fabric Blockchain in Action

After the Hyperledger Fabric Blockchain starts, Permission Marshals can connect to any of the Peers and submit transactions. Permission Marshals and Relays can use the Peers to create a subscription to receive a notification whenever the Orderers add a block to the ledger.

Relationship between Hyperledger Fabric Peers, Permission Marshal and Relay:



Submitting a Transaction:

Permission Marshal is a Hyperledger Client application. When it is time to submit a transaction, Permission Marshal invokes the chaincode “pubcc” and calls the function “pub” along with three arguments:

Argument 1: Root of a Merkle Tree of X509 Certificates Permission Marshal is issuing.
Argument 2: List of X509 Certificates that the transaction revokes. (JSON string formatted.)
Argument 3: Current Time

Permission Marshal Detail

Permission Marshal is a web application and Hyperledger Client written in Go to help accomplish the following functions:

Certificate Publishing:

1. Users can sign in, create Certificate Signing Requests (CSR) and submit them to another user for signing.
2. Users can sign in and see pending CSRs that they should sign. They can submit a signed X509 in response.
3. Permission Marshal will attach signed X509 to a Merkle Tree along with other signed X509s. Permission Marshal will periodically publish a transaction containing the Merkle Tree Root to the blockchain.
4. After Permission Marshal publishes the transaction to the blockchain, Permission Marshal will receive a Block Event from Hyperledger Fabric, which will enable it to compute a proof of publication for all of the X509s in the Merkle Tree.
5. Users can log in and obtain the Merkle Proof for X509s that they submitted to the Permission Marshal.

Revoking Certificates:

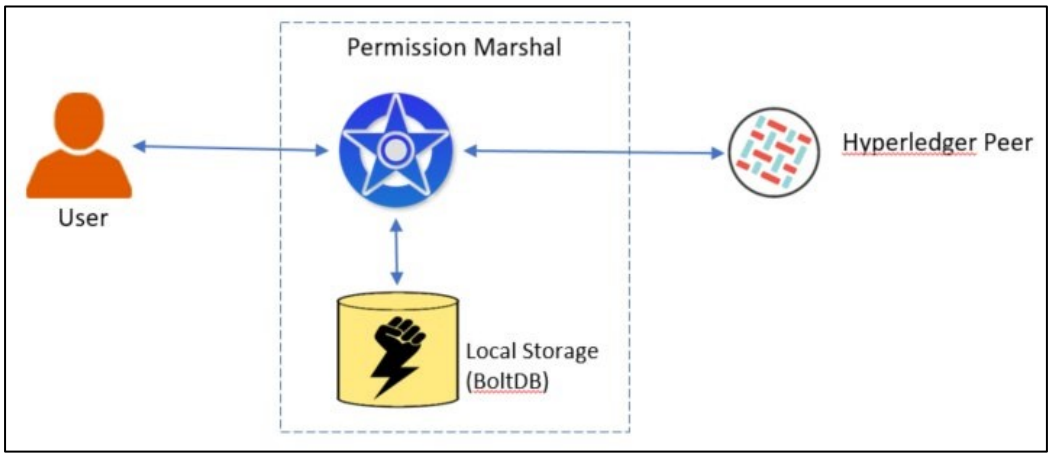
1. Users can sign in and mark an X509 for revocation.
2. Users can download a list of certificates that are marked for revocation to their Validator-Signer for evaluation.
3. Users can submit a signed revocation request.
4. Permission Marshal will combine revocation requests into a list.
5. Permission Marshal will periodically publish the list of revoked certificates to the blockchain.

Validity Checks:

Permission Marshal will perform validity checks to stop incorrect or redundant information from entering the blockchain. Although incorrect information would not cause downstream clients to make errors, it is a waste of scarce resources, so it is worthwhile for the Permission Marshal to screen it out. For example:

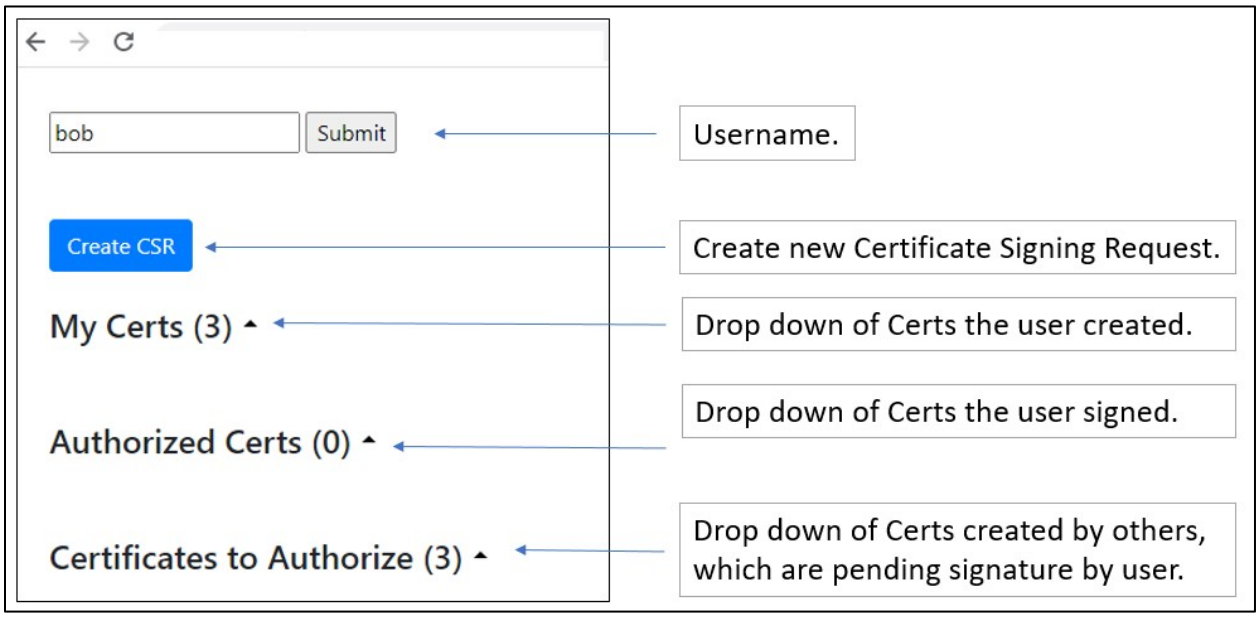
- Permission Marshal will not publish PCNs unless the chain of attributes is correct.
- Permission Marshall will not revoke a certificate if the applicant is attempting to use a revoked certificate as proof that they have permission to perform the revocation.

Permission Marshal Interactions:

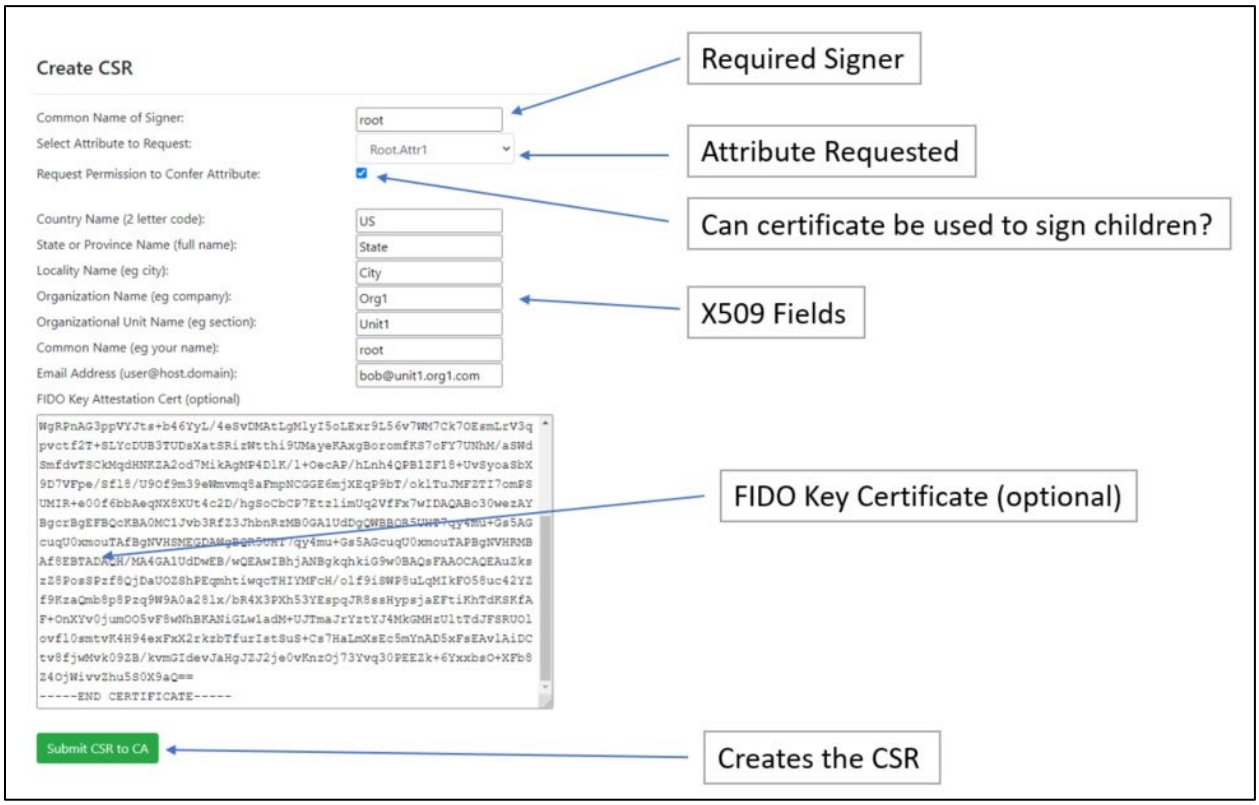


Permission Marshal User Interface:

Permission Marshal User Interface (1):

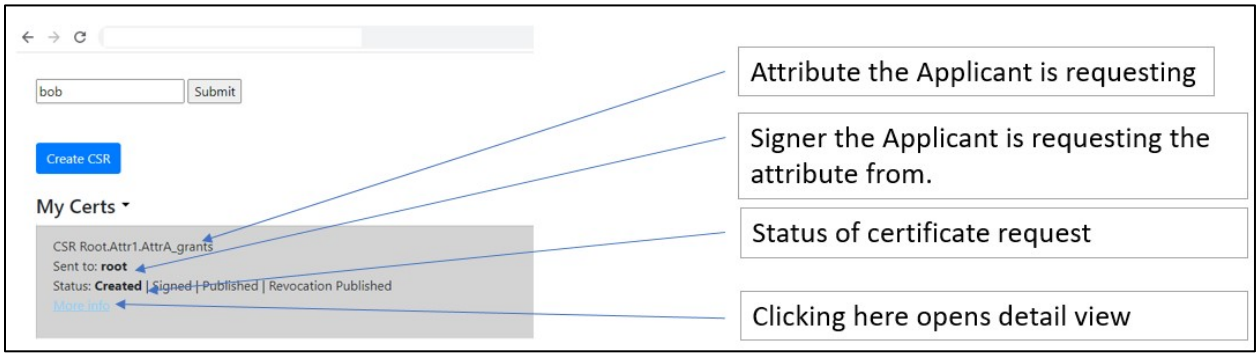


Permission Marshal User Interface: Create CSR Detail.



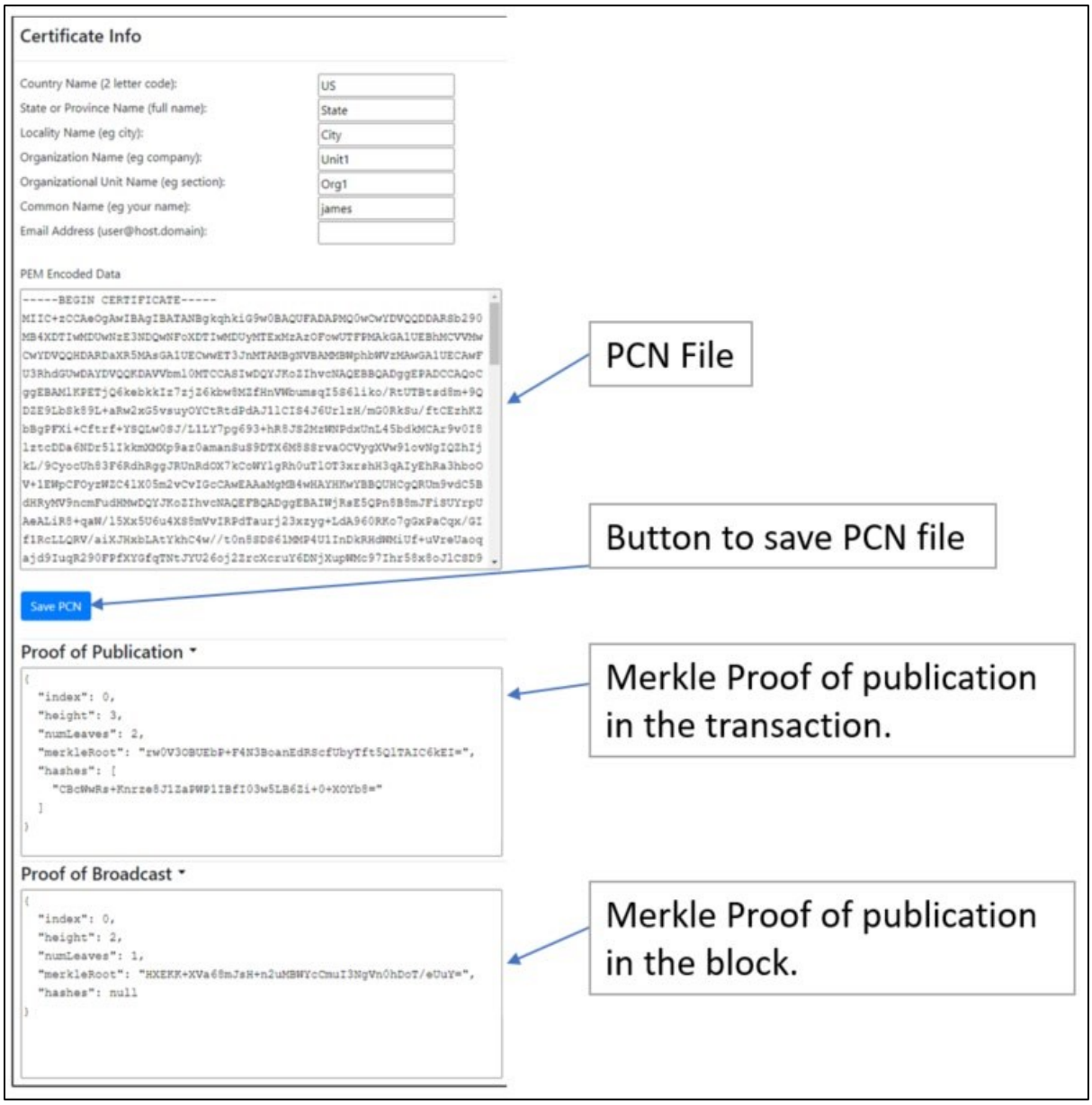
The Validator-Signer can read the “FIDO Key Attestation Cert” from a FIDO key plugged into a USB port. If this field is blank, no second factor is required to use this certificate. If this field has a certificate, then FIDO authentication will be necessary whenever an applicant presents the certificate.

Permission Marshal User Interface (2):



Permission Marshal User Interface (3) Detail view of CREATED CSR:

Permission Marshal User Interface (4) Detail view of Published CSR:

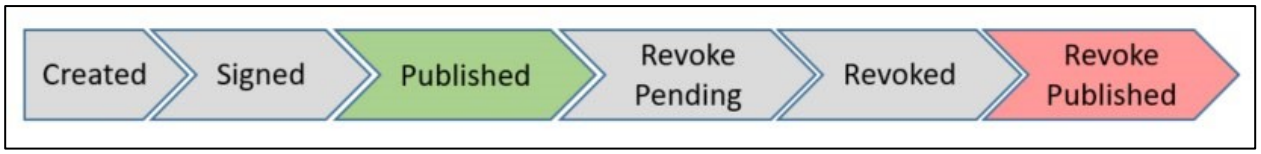


After a user submits a CSR for signing, the recipient must sign it, the Permission Marshal must publish it to the blockchain, and the Hyperledger Orderer must include it in a block. After Permission Marshal receives notification that the X509 has been published in a block, the Permission Marshal will create a PCN file, which the user can obtain by clicking the “Save PCN” button above.

Permission Marshal User Interface (4) Marking a Certificate for Revocation:

If a user clicks the “Mark for Revocation”, then the Permission Marshal will mark the certificate as needing to be revoked. Later, the Validator-Signer would do a GET /revoke/get to obtain a list of marked certificates.

Certificate lifecycle:



Certificates pass through several stages during their life. In this example, Bob is the user wishing to receive a certificate attaching an attribute to Bob. Carol is the user who is qualified to grant this attribute to Bob. Here is an explanation of the lifecycle phases of the certificate:

- **Created:** Bob has logged into Permission Marshal and created a Certificate Signing Request (CSR). He submits the CSR to the Permission Marshal. Permission Marshal stores the CSR locally, using BoltDB. Now the certificate is “CREATED.”
- **Signed:** Carol logs into Permission Marshal and sees that there is a “Certificate to Authorize” or in other words, a “Created” certificate, which Bob sent her. Carol uses her Validator-Signer application to sign the CSR, creating a Certificate (X509). Carol’s Validator-Signer combines this X509 with other information proving that Carol has authority to confer the attribute, creating a file called a Permission Chain File (PCN). Carol uploads this PCN file to Permission Marshal. Permission Marshal stores this PCN file in its local BoltDB. Now the certificate is “SIGNED”.

- **Published:** Periodically, the Permission Marshal batches all SIGNED certificates that it has in its database, attaching them as leaves in a Merkle Tree. Permission Marshal then publishes the root of this tree to the blockchain. Now the certificate is “PUBLISHED”.
- Revoke Pending: Carol decides that she should revoke Bob’s certificate. She logs into the Permission Marshal and expands the “Authorized Certs” dropdown, where she sees Bob’s certificate, which she previously signed. Carol clicks on the certificate to see a detailed view. At the bottom of the form, Carol clicks the button that says, “Mark for Revocation.”

Relay Detail

Relay is a Hyperledger client written in Go with the following functions:

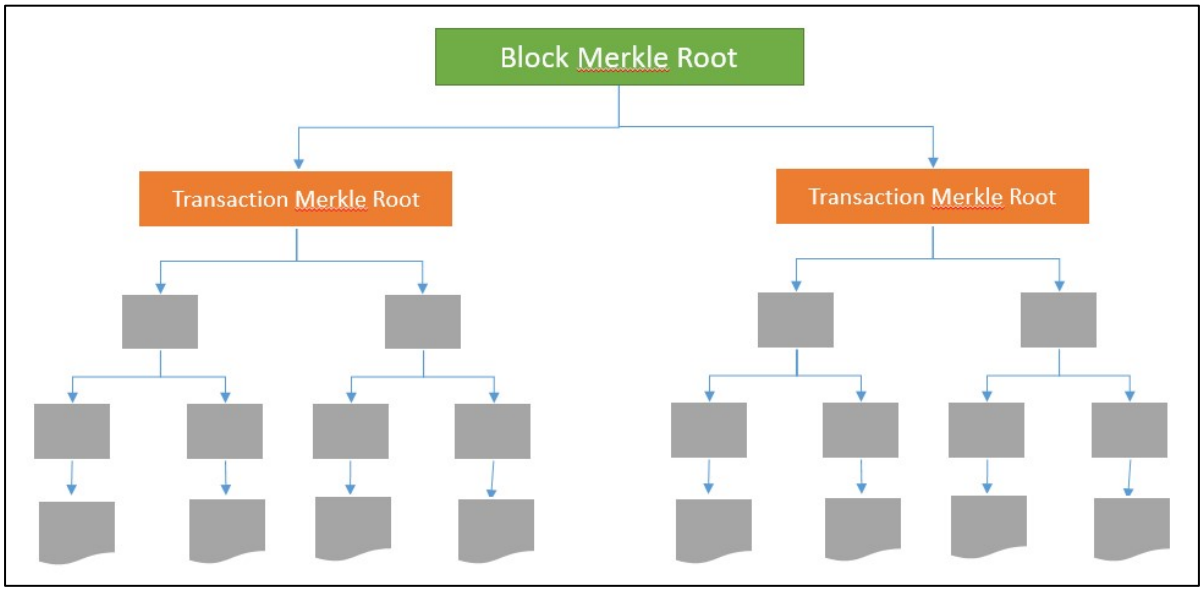
Receive Hyperledger Blocks:

Relay receives Block Announcement events from Hyperledger. These events indicate to Relay that a new block is available at height N. Relay queries a Hyperledger Peer to receive the contents of the block.

Compress Hyperledger Blocks into Relay Blocks:

Relay iterates through the block contents, finding all of the transactions in the block. For each transaction, Relay checks the block metadata to confirm that the transaction was “valid”, meaning that enough Peers endorsed the transaction. If the transaction is valid, Relay reads the Merkle Root of the transaction and begins construction of a Block-Level Merkle Tree.

Relay Constructing a Block-Level Merkle Tree:



Revocations:

Relay scans through the revoked certificates present in every valid transaction in the block and if they have the correct format, adds them to a bloom filter. For testing, we configured the Bloom filter with a size of 1000 items and a false positive probability of 1/100,000.

After creating the Block-Level Merkle Tree and Bloom Filter, Relay makes a JSON message called the Relay Block. This message has the following structure:

Relay Block:

```
{
  “index”:0,
  ”root”:”HASH_OF_BLOCK_MERKLE_ROOT”,
  “bloom”:”HASH_OF_BLOOM_FILTER”,
  “previous”:””
}
```

The fields have the following meanings:

- index = How many blocks are in the chain before this block.
- root = Merkle Root of Block-Level Merkle Tree of published certificates
- bloom = Hash of Bloom Filter, with all active revocations through the current block • previous = Hash of prior Relay Block

Sign Relay Blocks:

Relay signs a hash of the Relay Block using its private key, which Relay finds in “relay_deploy/go/src/blockchain-service/relay/certs/key.pem”. Relay creates a new JSON object called a “Relay Block Message”. This message has the following structure:

Relay Block Message:

```
{
  “block”: {
    “index”:0,
    ”root”:”HASH_OF_BLOCK_MERKLE_ROOT”,
    “bloom”:”HASH_OF_BLOOM_FILTER”,
    “previous”:””
  },
  “siglist”:[
    “SIGNATURE_OF_RELAY_BLOCK_HASH_1”
    ],
  “blockhash”:”HASH_OF_RELAY_BLOCK”
}
```

The fields have the following meanings:

- block = The Relay Block being transmitted.
- siglist = List of signatures that result from signing the hash of the Relay Block. Relay will include its own signature and may optionally bundle signatures from other relays.
- blockhash = Hash of Relay Block (which is what is signed)

Broadcast Relay Block Messages:

Relay will broadcast each Relay Block Message, and this is what the tactical clients consume to use as a basis for evaluating claims that a particular certificate was present in the blockchain. Relay publishes these messages to ActiveMQ, under a topic with a name derived as "RELAY_NAME-relayblocks".

Multiple relays can use the same ActiveMQ broker, so long as they choose unique RELAY_NAME values. If they do not cooperate to pick unique RELAY_NAME values, tactical clients will receive Relay Block Messages that they did not ask for, but this is not a serious problem, as tactical clients will be discarding relay block messages with a signature from a key they do not recognize.

Broadcast Bloom Messages:

As long as tactical clients have the Relay Block Messages, plus a list of Relay certificates that correspond to Relays with a good reputation, they have everything they need to validate information received from untrusted sources. One such piece of information is the Bloom Message, which communicates the contents of the Bloom filter. This message has the following structure:

```
{
  "index": 0,
  "filter": "BODY_OF_BLOOM_FILTER"
}
```

The fields have the following meanings:

- index = How many blocks are in the chain before the block that yields this Bloom Filter.
- filter = Bytes of the Bloom Filter

The Relay publishes these messages to ActiveMQ, under a topic with a name derived as "RELAY_NAME-bloomfilters".

To verify the integrity of this message, tactical clients can calculate the hash of the value of the "filter" field, look up the associated Relay Block Message (by index) and compare the "bloom" field value with the calculated hash.

Serving Relay Block Messages

Relay provides a REST endpoint for serving old Relay Block Messages, to help Validators get Relay Block Messages that they missed because they were offline. This service runs on port 8081 and includes two endpoints.

Relay REST Endpoints:

relayhost:8081/blocks	Pass in "blockNumber" and receive the Relay Block Message at that block height
relayhost:8081/currentHeight	Returns current height of the blockchain

Validator-Signer Detail

Validator-Signer is a Java desktop application with a Swing GUI and the following functions:

Receive Relay Broadcasts and Calculate the State of the Blockchain:

Trusted Relays:

Validator-Signer has a list of trusted relays at "/ValidatorSigner/relays/trusted-relays.txt". This file has a tab-delimited list of relays with the following columns:

- Nickname
- IP_ADDRESS:PORT where Validator-Signer can request blocks
- Hash of relay signing certificate

Relay signing certificates must be located in the same directory. The certificate file has a name equal to HASH_OF_RELAY_SIGNING_CERTIFICATE.crt

Getting Relay Blocks:

When Validator-Signer starts, it looks for previously downloaded blocks at "/ValidatorSigner/blocks/saved-blocks.txt" and loads these into memory. Then the Validator-Signer asks a Relay what is the height of the current highest block. Validator-Signer then requests all of the blocks from the last block it has up to the current height.

As Validator-Signer encounters Relay Blocks, before it accepts them as part of the chain, it performs validity checks:

1. Is the block height correct?
2. Is the value of the "previous" field equal to the hash of the prior block in the chain?
3. Does the enclosing Relay Block Message contain a "siglist" field with a signature of the block hash from one of the trusted relays?

After getting historical blocks, Validator-Signer switches to listening to an ActiveMQ broker for the latest Relay Block on the "RELAY_NAME-relayblocks" topic. Validator-Signer finds a list of ActiveMQ brokers and their URLs in "/ValidatorSigner/config/config.json". This file contains a

JSON object with an array named “brokers”. Each element of this array is a JSON object similar to this:

```
{
  "name":"broker1",
  "url":"10.0.1.22:61616",
  "relays":["relay1"]
}
```

The “relays” field here is a set of relay names that define the ActiveMQ topic name that Validator-Signer should subscribe to. (e.g. “relay1-relayblocks”)

Getting the latest Bloom Filter:

Validator-Signer receives the latest Bloom Filter by subscribing to the ActiveMQ topic RELAY_NAME-bloomfilters. Validator-Signer compares the Bloom Filter to the associated Relay Block Message by hashing the bloom filter and comparing the value to the “bloom” field in the Relay Block Message.

Granting Attributes:

Validator-Signer finds a Permission-Marshal by looking for a URL in the config file defined in “/ValidatorSigner/config/config.json” which contains an array named “permission-marshals”. Each element of this array is a JSON object similar to this:

```
{
  "url":"10.0.1.22:8080"
}
```

Validator-Signer can contact Permission Marshal and download a list of CSRs that are awaiting signature by calling GET “/csr/get/to_sign”. The CSR details will appear one at a time on the Validator-Signer, along with a message indicating whether the Validator-Signer has the resources to sign the CSR. In order to be able to sign the CSR:

1. Validator-Signer must have a PCN file giving them an attribute that can sign the CSR. This attribute must be shorter than the attribute being signed, and must be otherwise identical. This attribute must also have the “_grants” suffix. Validator-Signer finds PCN files in the “/ValidatorSigner/permission-files” folder.
2. The private key to this certificate must be present in “/ValidatorSigner/permissionfiles/keys/” with a filename equal to “HASH_OF_CERT_PUBLIC_KEY.key”
3. The PCN file must be valid. The chain of PEMs must extend attributes down from the root attribute to the attribute given by the PCN.

4. A The PCN file must be published. The Merkle Proof must show that it was included in a block.
5. The PCN file must not be revoked. The Bloom Filter must not contain any of the certificates in the PCN file.

If Validator-Signer can sign the CSR, it does so by creating a new Partial PCN file. The ValidatorSigner cannot make a complete PCN file, because is not published and therefore it is missing a Block Merkle Proof. To create the Partial PCN file, Validator-Signer does the following:

1. Begins with the PCN file that will be conferring an attribute.
2. Creates a signed X509 using the certificate at the top of the PCN file.
3. Adds the signed X590 to the top of the PCN file.

By doing these steps, Validator-Signer creates a Partial PCN file for the new attribute. ValidatorSigner then pushes the Partial PCN to Permission Marshal by calling “POST /csr/post/signed.”

Revoking Certificates:

Validator-Signer can contact Permission Marshal and download a list of certs that are awaiting revocation by calling GET “/revoke/get”. The certs will appear one at a time on the ValidatorSigner, along with a message indicating whether the Validator-Signer can actually revoke the cert. In order to be able to revoke the cert:

1. The user must have a PCN file giving them the certificate that signed the cert.
2. The private key to this certificate must be present in “/ValidatorSigner/permissionfiles/keys/” with a filename equal to “HASH_OF_CERT_PUBLIC_KEY.key”
3. The PCN file must be valid. The chain of PEMs must extend attributes down from the root attribute to the attribute given by the PCN.
4. The PCN file must be published. The Merkle Proof must show that it was included in a block.
5. The PCN file must not be revoked. The Bloom Filter must not contain any of the certificates in the PCN file.

Validator-Signer revokes the CSR by taking the PCN file that enables it to perform the revocation, and attaching an extra field to the JSON object at the bottom. The extra field has the name “revoke” and has a value with this structure:

```
{
  "message”:”REVOKE\n-----BEGIN CERTIFICATE---CERT_TO_REVOKE -----END CERTIFICATE-----“,
  “signature”:”SIGNED_WITH_PRIVATE_KEY_OF_FIRST_CERT_IN_PCN”
}
```

Validator-Signer then pushes the Revoking PCN to Permission Marshal by calling POST “/revoke/post”

Creating Invitations:

The first step in an interaction between Bob, who wishes to gain access, and Alice, who evaluates Bob's access request, is for Bob to indicate what access he wants. Alice will convert that request into an invitation by telling Bob what he will need to provide in order for her to be willing to grant access. Alice decides what attribute will satisfy her that Bob is qualified to get access. Alice also generates a random number (nonce) that Bob must sign using the same certificate that gives him the attribute. This will prove that Bob knows the private key belonging to the certificate.

Invitations have the following structure:


```
{

"attribute":"Root.Attr1",
“nonce":"0p8l\/Q88RA\/06TCDiS\/\/mowT\/lEwZkbzwwI5y0w3W0Ew2FABxoXQh0Wzbnx2uuENWE5Lh
mzGzwLCM8vBxpmP1CmO0bXQmFS1X\/E0NRk9ggEFihFYtsRkJ7KA+h7cmDH5IVTPqWr9xkfsUZFM1OyQe
kU7cXoCNI\/++M264UP0nqw="

}
```


Alice selects which attribute to require of Bob based on her particular situation and the nature of the access she is guarding. In practice, Alice would probably simply provide a list of the different kinds of access she can grant, along with the attributes each one will require.

Creating Requests for Permission:

Bob needs access to an asset that Alice controls, and Alice has provided Bob with an invitation describing what attribute will be necessary to gain access. Bob uses his Validator-Signer to read the invitation. Validator-Signer tells Bob whether it has a PCN file that it can use to demonstrate possession of the attribute. If Bob has a qualifying PCN file, Validator-Signer will create a Permission Request. The Permission Request consists of the qualified PCN file, and an extra field added to the JSON object. The field has the name “signedInvitation” and has the following structure:


```
{
"signature":"IC\/tXD9IpsLvvb7mJ5LQX4chUGyiRUWiixus2Rv17eQtPFMy987ZVo4SdTDVJXxEM1WHJrf04\
/cBhvJbYCy\/\/8zlDkQrvKC8ySTzYcR2P1vJZwucvD54zT3w7\/3a1YGKmmDYwmIoYMXm2O00IoE2N60ue
ME9e78fZz3wFtgbO\/PYibl3YEmYYj5TbXsKxvF34SOP26HSu526CRgwGVP+gpqE2m5SZWPYDPF+4cawME
DKV9nRgHGVBHCT4bkIqZNdJvsdJep+4FdwzCsbNkxlmBsrdJM9hlEDHwHcVNsOo0H+tzVYspflalsP0HgfOb9
OsoKAUuQ9bzoMKewAl8znqQ==",

“fidosig”: “OPTIONAL_U2F_SIGNATURE”,

"attribute":"Root.Attr1.AttrA",
```

"nonce":"9TK9BcNtXWX0\/AP9qt\/J92i1guvqTgFu2qFkgSpD9i6rkE43dSHlBaF7HpzJ0tImPhQMYX73poeSPMJB83Hjhbe4aH4buMORku+QDd7BhzPpoxHXUi7SAOHAR9R\/+15DYPfo1psBty5Sqm9wN\/uLTIt0eIQ2opHDcucGTOYkBjY="

}

Where:

- “signature” is a signature over the “nonce” using the private key of the certificate at the top of the PCN file.
- “attribute” is the attribute from the Invitation, which matches the extension field in the certificate at the top of the PCN file.
- “nonce” is the nonce from the Invitation.

Evaluating Requests for Permission:

After Bob gives Alice a Permission Request, Alice’s Validator-Signer will evaluate whether or not it is a satisfactory response to the Invitation. The Permission Request must satisfy the following:

1. The PCN file has the proper format.
2. Alice generated the nonce.
3. The first certificate in the PCN file has an attribute extension that matches the required attribute the Invitation specified.
4. The first certificate in the PCN file signed the nonce.
5. Each certificate in the PCN file signed the one above it.
6. Each certificate in the PCN has an attribute, which qualifies it to issue the certificate above.
7. The last certificate in the PCN file has the root attribute.
8. All of the certificates have valid Merkle Proofs, showing that they were published in the blockchain.
9. None of the certificates are expired.
10. None of the certificate are revoked.

Interface with FIDO/U2F device

In the case where a certificate has a hash value in the extension field OID 1.3.6.1.1.15.2, Validator-Signer will require a FIDO authentication to prove ownership of the certificate.

To demonstrate the use of the FIDO key, the Validator-Signer sends a registration request message to the FIDO key that contains, among other fields, the nonce from Alice’s invitation

message. The key will begin to blink, alerting Bob that he must touch the key to permit it to create a signature. After Bob touches the Fido key, it will sign the message and return the signature to Validator-Signer. Validator-Signer will append the signature in a JSON object called “fidosig” which appears in the "signedInvitation" section of the Permission Request.

Normal Workflows

Bob requests an attribute from Carol

1. Bob logs into Permission Marshal and creates a Certificate Signing Request.
2. Bob saves the private key on his computer.
3. Permission Marshal saves Bob’s request to BoltDB.
4. Carol opens her Validator-Signer and asks if she has any pending certificates to authorize.
5. Validator-Signer contacts Permission Marshal, which notifies Validator-Signer about Bob’s pending CSR.
6. Carol reviews Bob’s requested attribute. If Bob should have this attribute, Carol authorizes it.
7. Carol’s Validator-Signer app pushes the signed X509 to the Permission Marshal.
8. Permission Marshal bundles the X509 with others and creates a Transaction Merkle Root.
9. Permission Marshal sends transaction to Hyperledger.
10. Hyperledger Peers endorse transaction.
11. Hyperledger Orderer adds transaction to a block and broadcasts block to Peers.
12. Permission Marshal Workflow Branch
    a. Peer notifies Permission Marshal about a block event.
    b. Permission Marshal downloads the block contents and constructs a Block Merkle Tree out of all valid transactions.
    c. Permission Marshal updates BoltDB database with information indicating that the X509 has been published to the blockchain, along with the Merkle Proof.
    d. Bob logs into Permission Marshal and sees that his CSR was published.
    e. Bob saves the PCN file to his local machine.
13. Relay Workflow Branch
    a. Peer notifies Relay about a block event.
    b. Relay downloads the block contents and constructs a Block Merkle Tree out of all valid transactions.
    c. Relay creates and signs a Relay Block.
    d. Relay broadcasts Relay Block Message.

Carol revokes Bob’s attribute

1. Carol decides to revoke Bob’s certificate.
2. Carol logs into Permission Marshal and views Bob’s certificate.

3. Carol clicks the “Mark for Revocation” button.
4. Permission Marshal changes the status of the certificate to be Revoke Pending.
5. Carol starts her Validator-Signer.
6. Validator-Signer pulls down list of revoked certificates from Permission Marshal (GET /revoke/get).
7. Validator-Signer displays certificate to revoke.
8. Carol presses the Revoke button, creating a signed PCN file revoking the certificate.
9. Validator-Signer sends the PCN file to Permission Marshal (POST /revoke/post).
10. Permission Marshal evaluates PCN file to determine whether the author has permission to revoke.
11. If PCN file has permission to revoke, Permission Marshal batches the revoked PEM with other revoked PEMs into a list and submits the list in a transaction to Hyperledger.
12. Hyperledger Peers endorse transaction.
13. Hyperledger Orderer adds transaction to a block and broadcasts block to Peers.
14. Permission Marshal Workflow
    a. Peer notifies Permission Marshal about a block event.
    b. Permission Marshal downloads the block contents. For PEMS that the block revokes, Permission Marshal changes their status to Revoked_Published in the BoltDB database.
15. Relay Workflow
    a. Peer notifies Relay about a block event.
    b. Relay downloads the block contents and adds revoked certificates to its list of revoked certificates.
    c. Relay constructs a Bloom Filter based on the list of revoked certificates.
    d. Relay calculates the hash of the Bloom filter.
    e. Relay creates and signs a Relay Block.
    f. Relay broadcasts Relay Block Message.

Alice gets the current state of the blockchain

1. Alice installs Trusted Relay Certificates to her Validator-Signer.
2. Alice turns on Validator-Signer, which contacts a Relay and obtains the current blockchain height.
3. Validator-Signer requests any Relay Block Messages that it does not currently have from the Relay over REST.
4. Relay sends needed Relay Block Messages to Validator-Signer.
5. Validator-Signer authenticates and validates Relay Block messages and constructs its local copy of the blockchain.
6. Validator-Signer listens for new Relay Block Message events and Relay Bloom Messages.
7. Validator-Signer obtains the most recent Bloom Filter from Relay.
8. Validator-Signer compares the Bloom Filter Hash to the hash in the associated Relay Block at the same height.

Bob requests permission from Alice

1. Bob asks Alice for an invitation.
2. Alice uses Validator-Signer to create an invitation and gives it to Bob
3. Bob opens the Invitation with Bob's Validator-Signer and sees whether he has the attribute necessary to satisfy the Invitation.
4. If he has the attribute, Bob uses his Validator-Signer to create a Permission Request.
5. Bob gives the Permission Request to Alice.
6. Alice evaluates the permission request for validity.
    a. Did Bob prove that he has the relevant private keys?
    b. Did Bob show that he has certificates that grant him the required attribute?
    c. Did a chain of authorized certificates starting with the root certificate issue Bob the required attribute?
    d. Is Bob's certificate not expired?
    e. Is Bob's certificate not revoked?
7. Alice grants or denies Bob the permission he requested, based on Validator-Signer analysis.

Conclusions

We have built a system called the Global Permission Chain (GPC) to provide users operating in a disconnected, intermittent, and low bandwidth environment with crucial infrastructure they need to build smart contracts and distributed applications. Smart contracts rely on digital signatures to provide evidence that a participant agrees with a statement or requests an action. The challenge for a disconnected computer system running a smart contract is in the initial binding of a public key to a set of attributes that go along with the public key. If Bob requests a classified document, the fact that he can produce a digital signature is not enough to release the document. Somehow, the validating computer must also learn that Bob's digital signature gives Bob permission to receive the document.

GPC is a type of Public Key Infrastructure (PKI) that uses a blockchain to eliminate single points of control, remove the need to rely on lists of trusted certificate authorities and broadcast certificate revocations within minutes.

Our solution consists of four main components:

- A Hyperledger blockchain with a smart contract to record published certificates and certificate revocations.
- Permission Marshal, which batches certificates and revocations and sends them to the blockchain.
- Relay, which compresses Hyperledger blocks into a Merkle Root and Bloom Filter Hash, signs the resulting message with the relay's public key, and broadcasts to lowbandwidth users.

- Validator-Signer, which selects a set of Relays and obtains several independent but identical descriptions of the contents of the blockchain.

Users wishing to determine whether someone should receive access may do so using the GPC in the following way:

- Obtain a Permission Chain (PCN) file from the applicant, which contains a certificate granting whichever attribute is needed based on the access the applicant is requesting.
- Obtain a digital signature from the applicant, showing that they are the person who originally requested the certificate.
- Confirm that a Permission Marshal published the PCN file to the blockchain by verifying that the Block Merkle Root is present in the blockchain the Validator-Signer obtained from a Relay.
- Confirm that none of the certificates in the PCN file have ever been revoked by confirming that each certificate is not present in the latest Bloom Filter the ValidatorSigner obtained from a Relay.