École Polytechnique Fédérale de Lausanne

D-voting - Security audit

D-voting Security Report

Document owner	Chen Chang Lew (sciper: 321016)
Supervisor	Noémien Kocher, Pierluca Borsò-Tan, Simone
Responsible	Prof. Bryan Ford

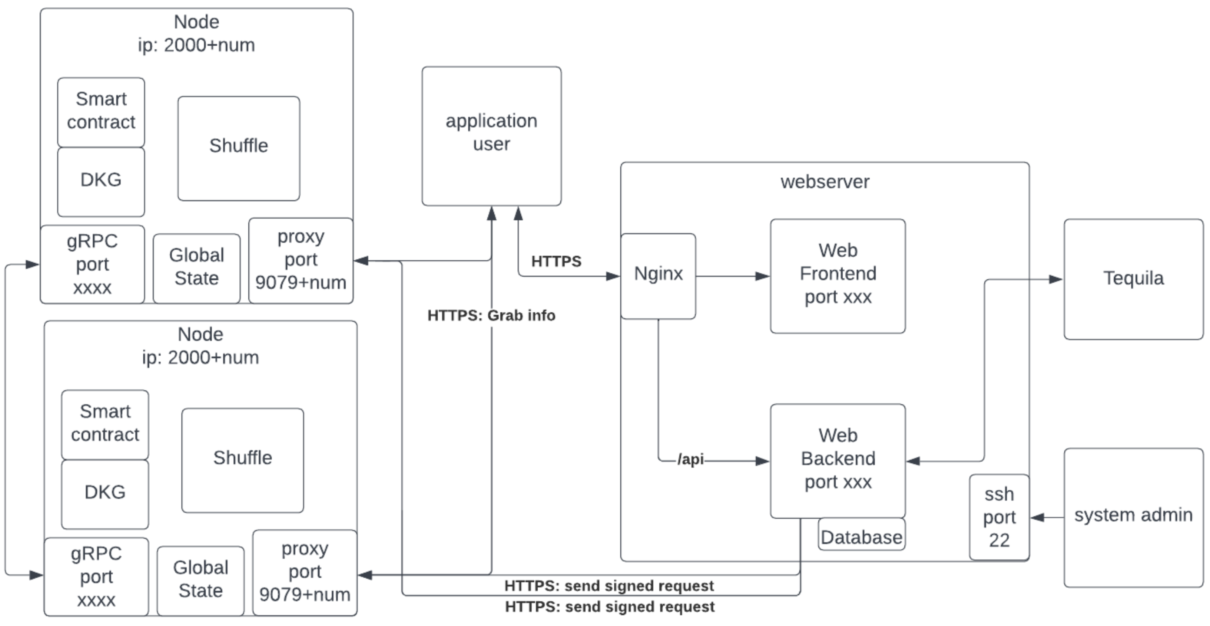
Table of content

Introduction

D-Voting is an e-voting platform based on the Dela blockchain. It uses state-of-the-art protocols that guarantee the privacy of votes and a fully decentralized process. This project was born in early 2021 and has been iteratively implemented by EPFL students under the supervision of DEDIS members.

This report analyzes components, trust zones, data flows, threat actors, controls, and findings of the Dvoting threat model version 1edd544. This was a point-in-time assessment and reflects the state of Dvoting, specifically version, at the time of the assessment, rather than any current or future state.

Component Descriptions



Component	Description
Web Frontend	The web frontend is a web app built with React. It offers a view for end-users to use the D-Voting system. The app is meant to be used by voters and admins. Admins can perform administrative tasks such as creating an election, closing it, or revealing the results. Depending on the task, the web frontend will directly send HTTP requests to the proxy of a blockchain node, or to the web backend. The end user needs a web-frontend to perform every action related to the voting services.
Web Backend	The web backend is built with Typescript and runs with NodeJs. The web backend handles authentication and authorization. Some requests that need specific authorization are relayed from the web frontend to the web backend. The web backend checks the requests and signs messages before relaying them to the blockchain node, which trusts the web backend. The web backend has a local database to store configuration

	data such as authorizations. Admins use the web frontend to perform updates. The blockchain node only trusts the backend as a source of data
Database	As mentioned above, the web-backend has a local database to store configuration data such as authorizations. The Database is run along with the backend server. This is the only part where the system records the user authorizations.
Proxy	A proxy offers the means for an external actor such as a website to interact with a blockchain node. It is a component of the blockchain node that exposes HTTP endpoints for external entities to send commands to the node. The proxy is notably used by the web clients to use the election system.
Blockchain node	A blockchain node is the wide definition of the program that runs on a host and participates in the election logic. The blockchain node is built on top of Dela with an additional d-voting smart contract, proxy, and two services: DKG and verifiable Shuffling. The blockchain node is more accurately a subsystem, as it wraps many other components. Blockchain nodes communicate through gRPC with the minogrpc network overlay. We sometimes refer to the blockchain node simply as a "node". With a decentralized network of blockchain nodes, which means no single party is able to break the system without compromising a Byzantine threshold of nodes. Which achieves no single point of failure.
Smart Contract	A smart contract is a piece of code that runs on a blockchain. It defines a set of operations that act on a global state (think of it as a database) and can be triggered with transactions. In the D-Voting system, a single D-Voting smart contract handles the elections. The smart contract is written in Golang. The smart contract ensures that elections follow a correct workflow to guarantee desirable properties such as privacy. For example, the smart contract won't allow ballots to be decrypted if they haven't been previously shuffled by a threshold of nodes.
Distributed Key Generation.	The DKG service allows the creation of a distributed key pair among multiple participants. Data encrypted with the key pair can only be decrypted with the contribution of a threshold of participants. This makes it convenient to distribute trust in encrypted data. In the D-Voting project, we use the Pedersen version of DKG. The DKG service needs to be set up at the beginning of each new election because we want each election to have its own key pair. Doing the setup requires two steps: 1) Initialization and 2) Setup. The initialization creates new RPC endpoints on each node, which they can use to communicate with each other. The second step, the setup, must be executed on one of the nodes. The setup step starts the DKG protocol and generates the key pair. Once done, the D-Voting smart contract can be called to open the election, which will retrieve the DKG public key and save it on the smart contract.
Verifiable	The shuffling service ensures that encrypted votes can not be linked to the user who cast them. Once the service is set up, each node can perform

Shuffling	what we call a "shuffling step". A shuffling step re-orders an array of elements such that the integrity of the elements is guaranteed (i.e no elements have been modified, added, or removed), but one can't trace how elements have been re-ordered. In D-Voting we use the Neff [2] implementation of verifiable shuffling.

Planes

our system itself is divided (roughly) into two “planes”, or groupings of components, the following table describes each plane, and groups the aforementioned components.

Plane	Description	Components
Web Plane	Web Plane is focused on the end-user journey. It controls the state of the election and the user authority.	Web Frontend, Web Backend, Database
Blockchain Plane	The blockchain plane guarantees the privacy of voters and a fully decentralized process to store the encrypted vote and the transactions. The nodes are public thus anyone can view and verify the states on the nodes.	Proxy, Blockchain node, Smart Contract, DPKG

Roles

We consider as actors parties of the system that are not part of the implementation. As we said in the previous part, the blockchain plane is public information thus anyone can view and verify the status on the nodes so all the users below can access the blockchain plane.

Roles	Description
Anonymous user / anyone	Users that are connected but not authenticated. Anyone can access the web proxy of the blockchain plane and the web frontend. component interact: web frontend, proxy
Unauthorized User	Users that are attempted to authenticate and failed. component interact: web frontend, tequila server.
Authenticated User (Voter)	Authenticated User, can access election information and vote at several events. component interact: web frontend, web backend, tequila server.
Operator	Authenticated User with operator level authority. - able to do what authenticated users can. - able to create an election - able to control the full lifecycle of an election (start election, end election, shuffle ballot, release result.) component interact: web frontend, web backend, tequila

Application Admin	Authenticated User with admin level authority. - able to do what operators can. - able to change the role of the users. - Able to add edit or remove the mapping between a node address and its proxy address. component interact: web frontend, web backend, tequila, backend database, proxy.
System admin	Admin that has terminal access to all the components mentioned above via SSH. Components interact: web frontend, web backend, backend db, node server.
Developer	D-voting application developer. Can edit d-voting GitHub code base. Component interact: GitHub codebase.
Blockchain operators	Operators that deploy/run our blockchain node. We can’t fully trust the blockchain operator and assume most of the threat models written to assume that adversaries are the blockchain operators. For example: if one of the blockchain nodes refuses to join dkg, dkg will be stuck.

Assets

Credentials	
User Credentials	User login details, password.
Operator Credentials	Operator login details, password
Application Admin Credentials	Application admin login details, password
System admin credentials	System admin details, password/private key to access the machine.
Server and secret	
backend server	The services that run in the backend
frontend server	The services that run in the frontend
node server	The services that run in the node server.
Secret key of Backend	Secret key that the backend uses to sign messages.
Application & Data	
Election information	The integrity of election information.
user ballot	The confidentiality and integrity of user’s ballot
Election result	The integrity of the election result
Authorization	
Application Admin’s ability	The ability to change roles of users and the proxy of the nodes
Operator’s ability	The ability to create an election and the full cycle flow of an election.
Voter ability	The ability to vote for a specific election and verify the ballot.

Blockchain	
Data	The data is maintained and stored inside the blockchain node.
DKG Key	The DKG key for each election.

Threat Actor

A threat actor, also called a malicious actor, is an entity that is partially or wholly responsible for a security incident that impacts the product or has the potential to impact an organization's security.

Actor	Descriptions
External Attacker	Attacker that can only interact with our website without authenticating via tequila. Their goal is to Steal credentials (voter credentials, operator credentials, …) using some vulnerabilities in the application (ex. Xss exploit) to become an authenticated attacker. Component interacts: web frontend
Authenticated Attacker	Authenticated attacker is someone who has authenticated (or stolen other user's session cookies) into the system. which allowed them to interact with the backend using the session cookie. It can launch an attack via sending requests to the frontend, backend, and blockchain node, and its goal is to break the security properties of our system. Components interact: web frontend, web backend.
Internal Attacker	Internal attack is someone who compromises some of our node servers, Their goal is to break the security properties of our system. Component interact: Node server.

Adversary goal

- Compromise a resource outside of their acceptable roles and privileges
    - Obtain voter ability.
    - Obtain operator ability.
    - Obtain application admin ability.
    - Obtain system admin ability.
- Read/Know the voter’s ballot choice.
- Change the voter’s ballot.
- Use server computation resources to run their own application (bitcoin mining etc.)
- Successfully launch a denial of service attack on the server. (with low budget)

Adversary Capabilities

What adversary can’t:

- We assume our network transport is safe (via HTTPS). Adversaries are not able to decrypt the TLS connection or spoof the TLS connection.
- Adversaries are not able to compromise our backend service.
- For this analysis Dela blockchain is out of scope, thus we assume adversaries are not able to attack Dela.
- For this analysis OS vulnerability is out of scope, adversaries are not able to attack the ubuntu system.
- Crypto library is safe …
- Ddos is safe …

What adversaries can do:

- Adversaries are external attackers (anonymous users), authenticated attackers (authenticated users), or internal attackers (compromised nodes).
- Adversaries have full control of the network. (he can view, remove and resend several packets.)
- Adversaries are able to compromise several nodes but not more than a Byzantine threshold of nodes.

Achieving security properties

- authorization
    - all the users are only able to do something they are authorized to.
- privacy
    - Ballot privacy of the voters, only voters themselves know their ballot (unlinkability)
    - Election privacy for the participants
- Confidentiality
    - Ballot confidentiality. The encrypted ballot stored on the blockchain will not leak any information about the voter's ballot.
- Integrity
    - voter's Ballot integrity, voter's Ballot not able to be changed by others, it can only be updated by the voter itself.
    - The system correctly counts the election results, which means the outcome of the election is correct and can't be changed by malicious nodes.
- Availability
    - no single-point-failure in nodes when several nodes (less than Byzantine Threshold) have been compromised. When some nodes are compromised it will not affect our system.
    - survive DoS attack in both frontend and backend web servers.
- Transparency/Verifiability/Auditability
    - Anyone can read and verify the log of events stored on the blockchain.

Threat & Mitigation

T1: The public/private key of the election is changed by the Adversary
Scenario
When a user wants to cast a ballot, the frontend server will request the election public key from a blockchain node. And the ballot will be encrypted using the public key. However, if the frontend requests the public key from a compromised blockchain node, the adversary can reply with a fake public key to the user. Then it can decrypt the ballot if the user uses the public key for the encryption.
Source
“web/frontend/src/pages/ballot/Show.tsx” function sendBallot “pubKey” is derived from the function “web/frontend/src/components/utils/useElection.tsx” to use pctx.getProxy() for the election info.
Assets
Node server, user ballot, election result
Threat actor
Internal Attacker
Category
Spoofing
Breaking Property
Confidentiality, Integrity
Risk
Base Score Metrics: 5.8/10
Mitigation
Frontend receives election public keys from at least ⅔ of the nodes. Frontend reports/sends alerts to the D-voting community when releasing a different public key.

T2: All the election stages can be ignored by malicious node
Scenario
Every request that is signed by the backend will send to the same node as described in config.env.template. However, if the node that is set in the backend env is a malicious node, it can selectively ignore some requests from the backend. For example, it can decide to execute a vote from certain sets of users while ignoring other users' vote that is not on their list.

Source

In the file “web/backend/Server.ts” function sendToDela.

```
function sendToDela(dataStr: string, req: express.Request, res: express.Response) {
  let payload = getPayload(dataStr);
  let uri = process.env.DELA_NODE_URL + req.baseUrl.slice(4);
```

And the value of process.env.DELA_NODE_URL is set to default = “http://localhost:9081”
In “web/backend/config.env.template”

Assets

Node server, user ballot, election result

Threat actor

Internal Attacker

Category

Denial of Service

Breaking Property

Availability

Risk

Base Score Metrics: 4.1/10

Mitigation

1. Instead of sending it to one node server, the backend will randomly pick one node server and send it to them. If the node happened to be malicious and drop the requests. The backend will pick a new random node server and send it again. However, this might cause a long wait time from frontend.
2. In order to mitigate the long response time from the backend we can just let the frontend check using get Election Info and then report failed at frontend pages and let the end user submit the request again. But this might cause bad user experiences because users might need to submit a request multiple times.
3. End users can choose which node to send to.
4. There is another way that we can solve this Threat is to redesign the system architecture, while the backend no longer sends data to the node while just being used as an authentication/authorization tool. The backend will now only sign the request from the frontend and then send the backend to the frontend and let the frontend handle the request sent to the node. However, this will introduce a new threat like a “replay attack” because the end user can record the signed msg from the backend and send it over and over again. In order to solve the replay attack we might need to have a nonce or counter for every signed request and the nodes should save the nonce or counter in the Dela global state which required lots of effort to mitigate the problem.

T3: A user can vote multiple times (count as multiple votes) in an election.

Scenario

Whenever the frontend sends an encrypted ballot to the backend to sign, the backend will include the userID of the user in the ballot and sign the msg. This design will allow every user to vote at most once. However, since we are using Tequila as our login server, thus it is hard to create multiple valid users in Tequila during backend testing. A workaround was created. Backend implements a function called “makeID” to create a random ID and sign it along with the encrypted ballot. This will allow a user to vote multiple times which counts as multiple votes. And this function is not yet removed until now. Thus an adversary can vote as many times as he wants to manipulate the election results.

Source

In the file “web/backend/Server.ts” function app.use(‘/api/evoting/*’)

```
app.use('/api/evoting/*', (req, res) => {
. . .
// special case for voting
  const regex = /\/api\/evoting\/elections\/.*\/vote/;
  if (req.baseUrl.match(regex)) {
    // We must set the UserID to know who this ballot is associated to. This is
    // only needed to allow users to cast multiple ballots, where only the last
    // ballot is taken into account. To preserve anonymity the web-backend could
    // translate UserIDs to another random ID.
    // bodyData.UserID = req.session.userid.toString();
    bodyData.UserID = makeid(10);
  }

function makeid(length: number) {
  let result = '';
  const characters =
'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789';
  const charactersLength = characters.length;
  for (let i = 0; i < length; i += 1) {
    result += characters.charAt(Math.floor(Math.random() * charactersLength));
  }
  return result;
}
```

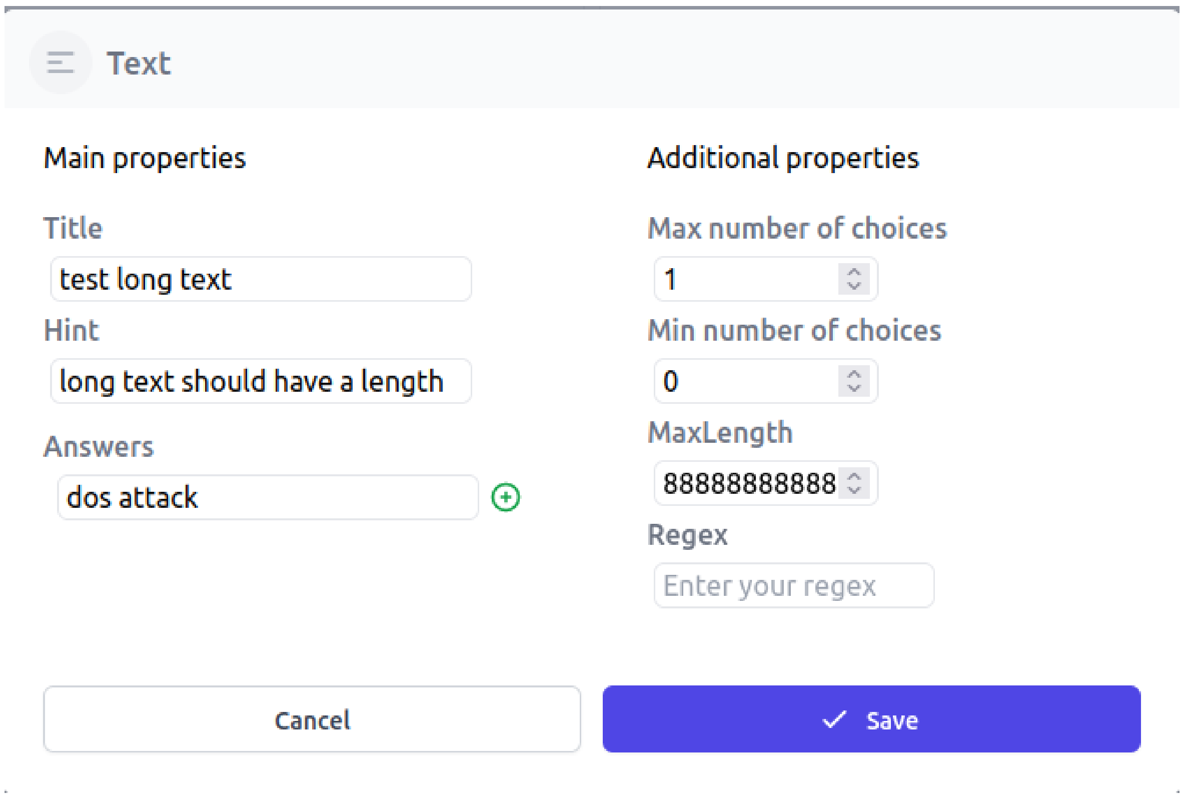
Assets

user ballot, election result
Threat actor
Authenticated Attacker
Category
Tampering
Breaking Property
Availability, Auditability
Risk
Base Score Metrics: 6.5/10
Mitigation
Remove the makeId function. If the developer prefers to keep the function for testing, we can set an env variable in config.env to determine this environment, which production environment would use User Id and the Dev environment would use Fake Id.

T4: Dkg public key will not be successfully created if there exists a malicious node.
Scenario
During the node initialization phase, the master node will run dkg init and collect all the dkgPubKeys to make sure all the keys are the same or the process will fail. However, a malicious node can just always return a false key then will always let the dkg initialization process fail. And right now there is not a monitor system or log to determine which node has the different output thus we are not able to track malicious users.
Source
“services/dkg/pedersen/mod.go” function setup() The code here will check that all the dkgPubKeys returned from the nodes are the same, and will output false if any of those fails.
Assets
Node server, election availability
Threat actor
Internal Attacker

Category
Denial of service
Breaking Property
Availability
Risk
Base Score Metrics: 4.4/10
Mitigation
The DKG server will proceed to the next phase once it receives more than ⅔ of the correct dkg pubkey. The DKG server will also report the nodes that return fake keys and raise an alarm to notify there exist some malicious node.

T5: Frontend create form didn't check for the maximum length of the form
Scenario
When creating a form, an operator/admin got a choice to select the "text" option, however the text option maxlength doesn't have a limit, which means a malicious user can create a huge form. And this will increase the load of the node & server when they encoded the cast vote. Since all encrypted ballots should be the same length to avoid leakage, thus the frontend will pad the ballot before the encryption. Thus with a huge form created it will increase the load of the node & server to process the result even if the vote size itself is small. This will potentially be a denial of service attack.
Source
Frontend create form.



Text

Main properties

Title: test long text

Hint: long text should have a length

Answers: dos attack

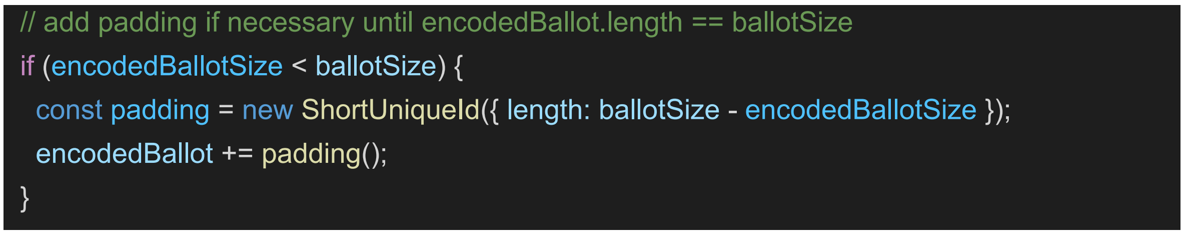
Additional properties

Max number of choices: 1

Min number of choices: 0

MaxLength: 88888888888

Regex: Enter your regex



```
// add padding if necessary until encodedBallot.length == ballotSize
if (encodedBallotSize < ballotSize) {
  const padding = new ShortUniqueId({ length: ballotSize - encodedBallotSize });
  encodedBallot += padding();
}
```

Assets
Node server, election availability
Threat actor
Authenticated attacker
Category
Denial of services
Breaking Property
availability
Risk
Base Score Metrics: 4.5/10

Mitigation

This can be mitigated by setting a maximum size of ballot for each form.
(Note this will also need to check in the smart contract because we will not trust the request from end-user)

T6: Election will not be able to reveal the result if anyone submits a fake vote.

Scenario

If a ballot fails to decrypt during the decryption process, it will return an error and make the whole decryption step fail. This will make the smart contracts will not accept the transaction which means the decryption process will never succeed.

Source

contracts/evoting/evoting.go combineShares()

```
for j := 0; j < ballotSize; j++ {
    chunk, err := decrypt(i, j, allPubShares, form.PubsharesUnits.Indexes)
    if err != nil {
        return xerrors.Errorf("failed to decrypt (K, C): %v", err)
    }
...
}
```

Node server, user ballot, election result

Threat actor

Authenticated Attacker

Category

Denial of service

Breaking Property

Availability

Risk

Base Score Metrics: 5.7/10

Mitigation

Ignore the ballot which has a decryption error, just assume it is an empty ballot during the reveal result.

T7: ShuffleThreshold shouldn’t be used in the nbrSubmissions

Scenario

In general we define that ShuffleThreshold > f, while DKGThreshold > 2f. This is because we only need a shuffler to shuffle 1 time more than the malicious node (f). And for computePubshares, we will need to have more than 2f to protect against the byzantine node. However, the current system only has “ShuffleThreshold” which didn’t differentiate between “shuffle” and “compute pubshares”. Which will cause problems in the future if we tighten the shuffle threshold.

Source

For checking if we have shuffled enough, we can use shuffleThreshold
contracts/evoting/evoting.go line 396~399

```
// in case we have enough shuffled ballots, we update the status
if len(form.ShuffleInstances) >= form.ShuffleThreshold {
    form.Status = types.ShuffledBallots
    PromFormStatus.WithLabelValues(form.FormID).Set(float64(form.Status))
}
```

But for checking the threshold
contracts/evoting/evoting.go line 583~586

```
if nbrSubmissions >= form.ShuffleThreshold {
    form.Status = types.PubSharesSubmitted
    PromFormStatus.WithLabelValues(form.FormID).Set(float64(form.Status))
}
```

Assets

Node server, user ballot, election result

Threat actor

Internal Attacker

Category

Tampering

Breaking Property

Integrity

Risk

Base Score Metrics: 6.6/10

Mitigation

We can create a new threshold “DKGThreshold” for combine Pubshares and only use “ShuffleThreshold” for shuffler related function.

T8: Logout will not clear all the sessions in the browser
Scenario
When a user logs out and the user is not closing his browser (closing a single tab will not solve this issue). A malicious user can just log in without the need to input their credentials. The adversary can change the user’s previous vote or cast another ballot under user name. The worst thing that can happen is if the previous login account is an admin account, the malicious user can assign themselves as admin and then get admin ability.
Source
It is easy to try to reproduce this error by logging in immediately after logout. Since the browser keeps the tequila cookie then the next user can log in without inputting their credentials.
Assets
User Credentials, Operator Credentials, Application Admin Credentials, Application Admin’s ability, Operator’s ability, Voter ability.
Threat actor
Unauthorized User, Authenticated User
Category
Spoofing, Elevation of privilege
Breaking Property
Authentication, Availability, Authorization
Risk
Base Score Metrics: 6.4/10
Mitigation
Use OpenID connect instead of tequila

T9: A user who is not an admin or operator cannot vote.

Scenario

Every action from a user who is not an admin or operator will become unauthorized, including casting a vote. An authenticated user should able to cast a vote even if they are not an operator or admin

Source

web/backend/src/Server.ts

```
// Secure /api/evoting to admins and operators
app.use('/api/evoting/*', (req, res, next) => {
  if (!isAuthorized(req.session.userid, SUBJECT_ELECTION, ACTION_CREATE)) {
    res.status(400).send('Unauthorized - only admins and operators allowed');
    return;
  }
  next();
});
```

Assets

Voter ability.

Threat actor

Authenticated user

Category

Denial of service

Breaking Property

Availability, Authorization

Risk

Base Score Metrics: 5.7/10

Mitigation

Backend should let an authenticated user have the ability to vote.

Technical Debt

Other than threat I have found several Technical Debt that might cause problems or reduce user readability in the future. I have created issues in Github for all the Technical debt as follows:

1. Should verify signature before executing the request.
   a. Proxy dkg.go EditDKGActor
   b. Proxy election.go EditForm
   c. Proxy election.go DeleteForm
2. Structure legacy not change
   a. Contracts evoting types transactions.go CreateForm struct (No more require AdminID)
3. Unclear Comment
   a. Proxy election.go line 117, comment msg said sign response but it didn't (should change sign to encoding)
4. Variable name not consistent
   a. proxy / xxx.go. formIDBuf, formIdBuff, Buf … (formIDBuf)
5. Should check lenaddrs before sending getPeerKey
   a. services/dkg/pedersend/mod.go setup()
6. Encrypt function in DKG && Marshall ballot function in smart contract is not used anymore
   a. services/dkg/pedersen/mod.go Encrypt
7. Duplicate function getForm
   a. Services/dkg/pedersen/mod.go getForm()
   b. Services/dkg/pedersen/handler.go getForm()
   c. Services/shuffle/neff/mod.go getForm()
8. Refactor in dkg & shuffler
   a. services/dkg/pedersen/handler.go handleDecryptRequest() Line 505-519
   b. services/shuffle/neff/handler.go handleStartShuffIt() Line 134-148
9. Change loop and sleep to channel + ctx timeout to increase readability of code
   a. Services/shuffle/neff/mod.go waitAndCheckShuffling()
   b. services/shuffle/neff/handler.go handleStartShuffle()
10. Shouldn't use the fingerprint function for pseudorandomness. (not efficient)
    a. Services/shuffle/neff/handler.go makeTx()
    b. contracts/evoting/evoting.go shuffleBallots()
11. Remove point & public key in backend since it is not used.
    a. web/backend/src/Server.tx app.delete()
    b. web/backend/src/Server.tx getPayload()

Appendix

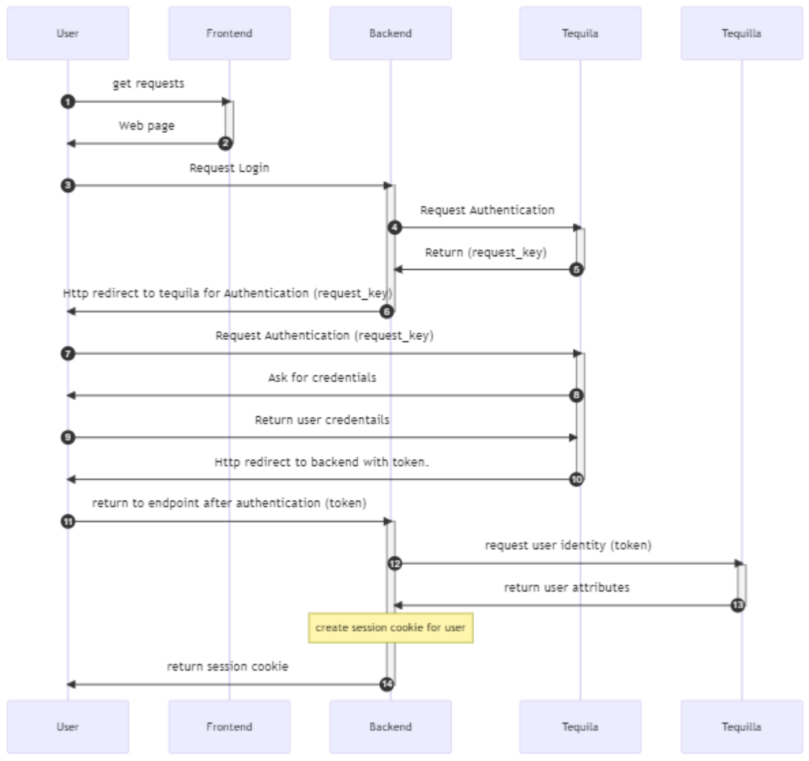
Reference

[1] kubernetes threat model:
https://github.com/trailofbits/audit-kubernetes/blob/master/reports/Kubernetes%20Threat%20Model.pdf
[2] A verifiable secret shuffle and its application to e-voting.
https://dl.acm.org/doi/10.1145/501983.502000
[3] CVSS https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator

All User Flow Graph

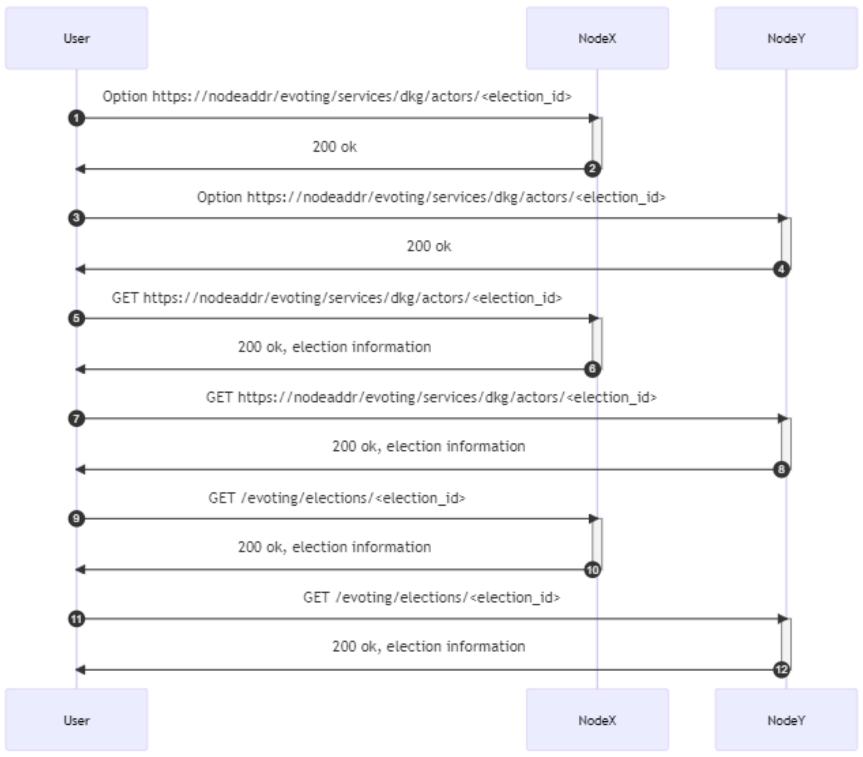
I created several user flow graphs to help visualize the steps that users would take to perform several actions. I followed a process of defining the task or goal, identifying the necessary steps, creating a visual representation using a flowchart tool, and reviewing and revising the graph to ensure accuracy and effectiveness. These user flow graphs were a valuable resource for understanding how users navigate and interact with our product, and they helped inform the design and development process.

Login


User
Frontend
Backend
Tequila
Tequilla
get requests
Web page
Request Login
Request Authentication
Return (request_key)
Http redirect to tequila for Authentication (request_key)
Request Authentication (request_key)
Ask for credentials
Return user credentails
Http redirect to backend with token.
return to endpoint after authentication (token)
request user identity (token)
return user attributes
create session cookie for user
return session cookie

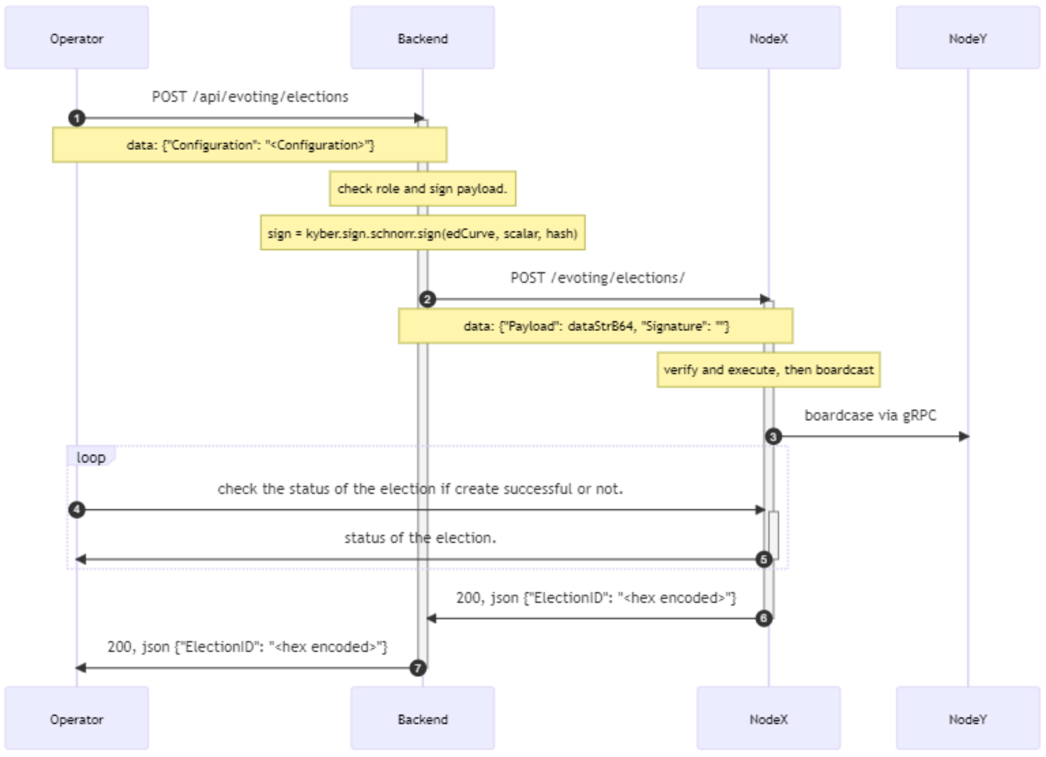
election get info

```
GET api/evoting/elections/{ElectionID}
frontend directly grab from the node
```


User
NodeX
NodeY
Option https://nodeaddr/evoting/services/dkg/actors/<election_id>
200 ok
Option https://nodeaddr/evoting/services/dkg/actors/<election_id>
200 ok
GET https://nodeaddr/evoting/services/dkg/actors/<election_id>
200 ok, election information
GET https://nodeaddr/evoting/services/dkg/actors/<election_id>
200 ok, election information
GET /evoting/elections/<election_id>
200 ok, election information
GET /evoting/elections/<election_id>
200 ok, election information
User
NodeX
NodeY

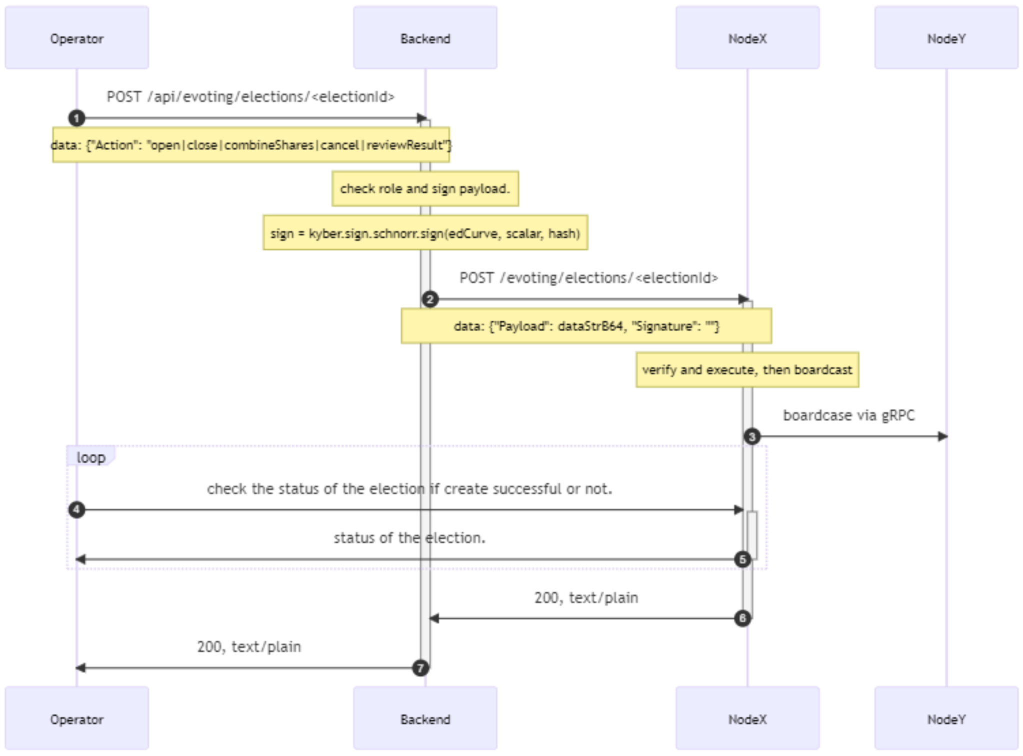
Election create

```
POST api/evoting/elections, data: {"Configuration": "<Configuration>"}
response 200 OK application/json {"ElectionID": "<hex encoded>"}
```


Operator
Backend
NodeX
NodeY
POST /api/evoting/elections
data: {"Configuration": "<Configuration>"}
check role and sign payload.
sign = kyber.sign.schnorr.sign(edCurve, scalar, hash)
POST /evoting/elections/
data: {"Payload": dataStrB64, "Signature": ""}
verify and execute, then boardcast
boardcase via gRPC
loop
check the status of the election if create successful or not.
status of the election.
200, json {"ElectionID": "<hex encoded>"}
200, json {"ElectionID": "<hex encoded>"}
Operator
Backend
NodeX
NodeY

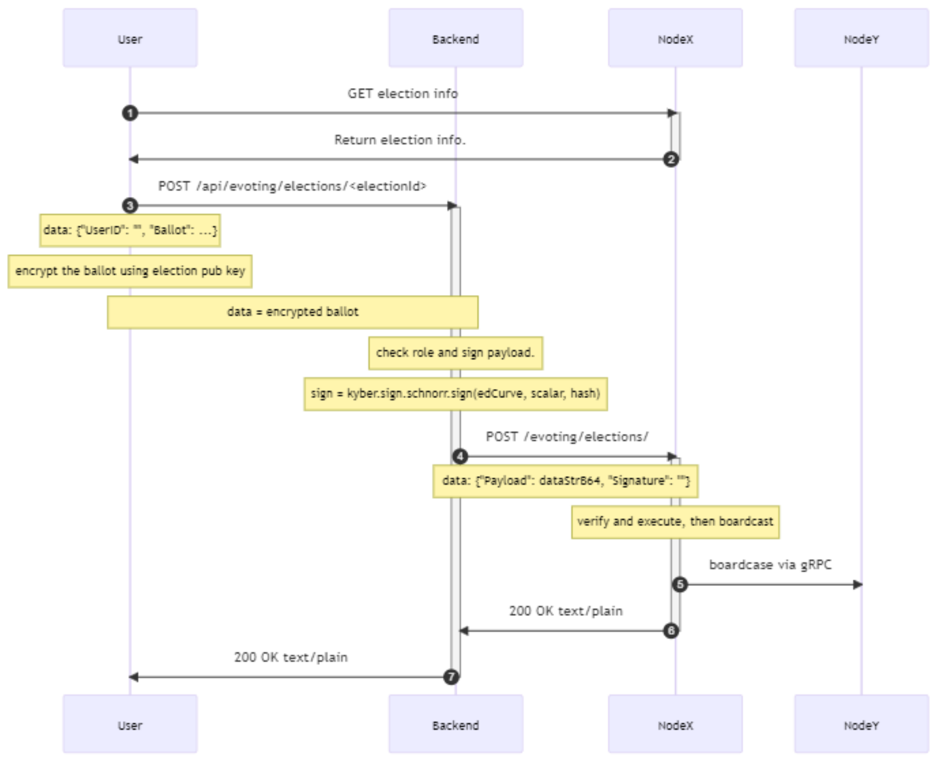
Election flow (open/close/combineShares/cancel/reviewresult)

```
PUT api/evoting/elections/{ElectionID},
data: {"Action":"open|close|combineShares|cancel|reviewResult"}
```


Operator
Backend
NodeX
NodeY
POST /api/evoting/elections/<electionId>
data: {"Action": "open|close|combineShares|cancel|reviewResult"}
check role and sign payload.
sign = kyber.sign.schnorr.sign(edCurve, scalar, hash)
POST /evoting/elections/<electionId>
data: {"Payload": dataStrB64, "Signature": ""}
verify and execute, then boardcast
boardcase via gRPC
loop
check the status of the election if create successful or not.
status of the election.
200, text/plain
200, text/plain
1
2
3
4
5
6
7

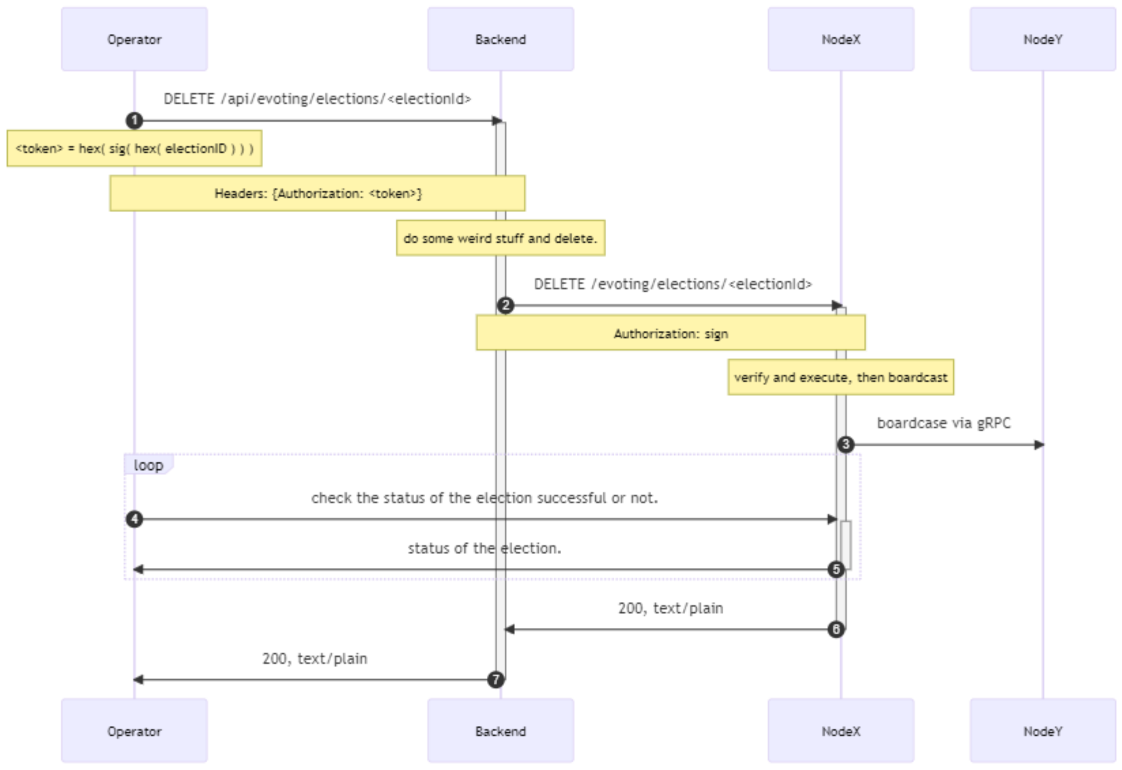
Cast Vote

```
POST /evoting/elections/{ElectionID}/vote
data: {"UserID": "", "Ballot": [{"K": "<bin>", "C": "<bin>"}]}
```


User
Backend
NodeX
NodeY
GET election info
Return election info.
POST /api/evoting/elections/<electionId>
data: {"UserID": "", "Ballot": ...}
encrypt the ballot using election pub key
data = encrypted ballot
check role and sign payload.
sign = kyber.sign.schnorr.sign(edCurve, scalar, hash)
POST /evoting/elections/
data: {"Payload": dataStrB64, "Signature": ""}
verify and execute, then boardcast
boardcase via gRPC
200 OK text/plain
200 OK text/plain
1
2
3
4
5
6
7

Election delete

```
DELETE /evoting/elections/{ElectionID}
{Authorization: <token>}, <token> = hex( sig( hex( electionID ) )
```


Operator
Backend
NodeX
NodeY
DELETE /api/evoting/elections/<electionId>
<token> = hex(sig(hex(electionID)))
Headers: {Authorization: <token>}
do some weird stuff and delete.
DELETE /evoting/elections/<electionId>
Authorization: sign
verify and execute, then boardcast
boardcase via gRPC
loop
check the status of the election successful or not.
status of the election.
200, text/plain
200, text/plain
1
2
3
4
5
6
7