Ethereum Foundation

(Draft) - Security assessment of P0TION

ctrlc03

Q1 2023

Contents

1 Overview

This document provides a summary of the internal security assessment conducted on P0TION, during Q1 2023.

P0TION is a set of tools (framework) that automates the creation and execution of zk-SNARK Phase2 ceremonies. The product provides three main components:

- an SDK
- the Firebase backend code
- an example client implementation as a command line interface

The assessment goal was to identify any security vulnerabilities present within P0TION, and provide appropriate recommendation steps for the developers to mitigate these risks. The penetration test was conducted in tandem with the final phase of the development, which include test writing and refactoring. Any identified issue was amended as soon as this was discovered (based on priorities).

Please note that this report is a work in progress, and that the security assessment has not been completed yet. As a result, certain issues described in this report are still present in the code. Please use P0T1ON with caution.

1.1 Scope

The assessment was conducted on the code available on GitHub, under the branch dev. The areas of the code that were audited are listed below:

- actions package (SDK)
- backend (Firebase cloud functions)
- phase2cli (a cli implementation)

The most up to date of the code can be found in the dev branch.

1.2 zk-SNARK Phase2 Ceremonies

A trusted setup ceremony is a procedure that is done once to generate a piece of data that must then be used every time some cryptographic protocol is run. With Groth16, a proving system that is still frequently used due to its fast proving time, phase2 ceremonies are required to generate the verifying and proving keys. These ceremonies are circuit specific, therefore must be run every time a circuit changes and for different parameter sets.

1.3 Threat model

P0TION has two user roles, a coordinator and a participant. Coordinators can setup and finalize ceremonies, whereas participants can contribute to them. Coordinators would usually manage the Firebase account, as well as the AWS infrastructure required (which at this time is only S3). Due to them controlling the Firebase infrastructure, they will have access to the participants' information.
Participant can log in with any OAuth2 provider (GitHub is the one of choice for the cli implementation), therefore any data associated with their GitHub account will be stored on Firebase. While this personal data is only the public data available on GitHub, users might want to keep it private that they contributed to a ceremony.

The main threats to P0T1ON are:

- Coordinators having participants download malware on their machines (participant trust coordinators that the downloaded files required for the ceremony are legitimate)
- Coordinators extracting the toxic waste of each contribution and defy the purpose of a phase2 trusted setup ceremony
- External users/Participants accessing the backend infrastructure and other participants' data
- External users/Participants launching denial of service attacks to disrupt a ceremony
- External users/Participants abusing the system to force the coordinator in to paying a large infrastructure bill (for instance by bypassing rate limiting)

1.4 Disclaimer

A security assessment can never verify the complete absence of vulnerabilities. This is an effort where security professionals try to find as many vulnerabilities as possible. 100% security cannot be guaranteed, even if multiple issues were found during the assessment.
As a result of this, P0TION should be considered experimental software and used with caution.

2 Summary

Overall, the code of P0TION is of good quality, with a solid test suite and the code having undergone several iterations of refactoring.
Despite the quality, the assessment identified several issues that could have resulted in the disclosure of sensitive data, as well as potentially executing malicious code into participants' computers.
To remediate the identified issues and improve the overall code quality, the following high level recommendations should be followed:

- Enforce strict access control to the Firestore database
- Enforce the correct access control on all Cloud functions
- Implement a sensible rate limiting mechanism
- Implement checks on downloaded files

2.1 Assessment Results

Description	Severity	Status
Arbitrary File Upload to S3	High	Not Fixed
Firestore Broken Access Control	High	Fixed
Arbitrary File Download from S3	Medium	Fixed
Lack of File Type Verification	Low	Not Fixed
Email Enumeration	Low	Not Fixed

3 Assessment Findings

3.1 Arbitrary File Upload to S3

3.1.1 Description

A malicious participant could leverage the available cloud functions to overwrite files in the S3 bucket for a ceremony and upload arbitrary files. This could result in the following scenarios:

- denial of service (Dos) on the ceremony - the previous contribution has been modified and cannot be verified. The ceremony will have to start over.
- a participant can add malicious files that will be downloaded by other participants
- a participant can add malicious files that will be downloaded by the coordinator and could potentially result in arbitrary code execution.

3.1.2 Risk Severity

Impact: High - an attacker could upload malicious files and potentially execute code on the coordinator or participants computers.
Likelihood: High - any authenticated user could be uploading arbitrary files.

3.1.3 Proof of Concept

The following code, part of the backend package, is used to enforce checks to which users can upload files to S3 using a multi part upload. The only checks in this code are that the caller is a participant in the ceremony, but they could be launching this attack while contributing. The attacker would have a time window before they are locked out to launch this attack, however this is highly likely to be enough time to launch an automated attack.

```
const checkPreConditionForCurrentContributorToInteractWithMultiPartUpload =
↪  async (
    contributorId: string,
    ceremonyId: string
) => {
    // Get ceremony and participant documents.
    const ceremonyDoc = await
    ↪  getDocumentById(commonTerms.collections.ceremonies.name, ceremonyId)
    const participantDoc = await
    ↪  getDocumentById(getParticipantsCollectionPath(ceremonyId),
    ↪  contributorId!)

    // Get data from docs.
    const ceremonyData = ceremonyDoc.data()
    const participantData = participantDoc.data()

    if (!ceremonyData || !participantData)
    ↪  logAndThrowError(COMMON_ERRORS.CM_INEXISTENT_DOCUMENT_DATA)

    // Check pre-condition to start multi-part upload for a current
    ↪  contributor.
    const { status, contributionStep } = participantData!

    if (status !== ParticipantStatus.CONTRIBUTING && contributionStep !==
    ↪  ParticipantContributionStep.UPLOADING)
        logAndThrowError([..snip])
}
```

3.1.4 Recommendations

In the first place, each call to functions dealing with S3 buckets, aside from creation, should check that the bucket name is associated with a ceremony. This can be achieved by using the following code:

```
// Extract ceremony prefix from bucket name.
const ceremonyPrefix =
↪  bucketName.replace(String(process.env.AWS_CEREMONY_BUCKET_POSTFIX), "")

// Query the collection.
const ceremonyCollection = await firestoreDatabase
    .collection(commonTerms.collections.ceremonies.name)
    .where(commonTerms.collections.ceremonies.fields.prefix, "==",
    ↪  ceremonyPrefix)
    .get()

if (ceremonyCollection.empty) logAndThrowError([..snip])
```

Furthermore, given that the only file that a contributor is supposed to be uploading is the new zKey (after their contribution), the code should ensure that only the zKey with the correct index is uploaded. For instance, the 13th contributor should be uploading the zKey number 12. While a simple check on the file name will prevent any overwriting of files, this will not suffice to prevent users to upload malicious data, which might later be downloaded by another contributor. Therefore, on top of the file name check, the zKey file should be verified server side before being stored on the S3 bucket for the ceremony. This can be achieved with the use of the ‘snarkJS‘ library.

3.2 Firebase Broken Access Control

3.2.1 Description

During the assessment it was noted that it was possible to enumerate data about other users from any authenticated accounts. This could lead to the disclosure of sensitive contributors' details such as email addresses and names.

3.2.2 Risk Severity

Impact: High - sensitive data could be exposed.
Likelihood: High - this is a trivial attack to execute.

3.2.3 Proof of concept

The following TypeScript code presents an instance where, by using the **getDocumentById** (a wrapper around Firestore functionality that allows to query a Firestore document by its id), it was possible for user2 to retrieve the details of user1.

```
 await createNewFirebaseUserWithEmailAndPw(
    userApp,
    user.data.email,
    generatePseudoRandomStringOfNumbers(24)
)

const user2 = fakeUsersData.fakeUser2
await createNewFirebaseUserWithEmailAndPw(
    userApp,
    user2.data.email,
    generatePseudoRandomStringOfNumbers(24)
)

// Retrieve the current auth user in Firebase.
const currentAuthenticatedUser = getCurrentFirebaseAuthUser(userApp)
user2.uid = currentAuthenticatedUser.uid

await sleep(5000) // 5s delay.
console.log('Currently logged in user', currentAuthenticatedUser.email)
const userDoc = await getDocumentById(userFirestore, "users", user.uid)
const data = userDoc.data()
console.log(`Retrieved data for user ${data?.email}`, data)
// Clean user from DB.
await adminFirestore.collection("users").doc(user2.uid).delete()

// Remove Auth user.
await adminAuth.deleteUser(user2.uid)
```

As shown below, we are logged in as user2 and were able to retrieve details about other users.

```
console.log
Currently logged in user user2@user.com

console.log
Retrieved data for user user1@user.com {
  emailVerified: false,
  photoURL: '',
  email: 'user1@user.com',
  name: null,
  lastUpdated: 1674916402088,
  displayName: null,
  creationTime: '2023-01-28T14:33:20Z',
  lastSignInTime: '2023-01-28T14:33:20Z'
}
```

3.2.4 Recommendations

It is recommended to enforce the following Firestore security rules to prevent unauthorised access to arbitrary users' data. The rule below prevents users from accessing documents of other users, and only allow read access to their data. On the other hand, the coordinator is allowed to create new users only.
For ceremonies data, which is not sensitive, this can be accessed by any authenticated user, however only managed programmatically by an user with coordinator privileges:

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    // Define which users can read and write to the database
    match /users/{userId} {
      // users can read update and delete their own data
      allow read, update, delete:
        if request.auth != null &&
        request.auth.uid == userId;
    }
    // applies to the ceremonies collection and nested collections
    match /ceremonies/{ceremonyId=**} {
      // any authenticated user can read
      allow read: if request.auth != null;
      // only coordinator can create, and update ceremonies
      allow create, update:
        if request.auth != null &&
        request.auth.token.coordinator;
    }
  }
}
```

3.2.5 Remediation

The issue was addressed with the following pull request.
Supporting test cases where added to prove the effectiveness of the newly implemented rules, and can be found in the repository.

3.3 Arbitrary File Download From S3

3.3.1 Description

The Cloud Function **generateGetObjectPreSignedUrl** allows any user to generate a pre-signed URL for objects within the coordinator's S3 buckets. This functionality is not restricted to any specific bucket used only for a ceremony, neither an object (such as the next zKey to be downloaded), and therefore could result in an attacker being able to download sensitive files from S3 out of the coordinator's AWS account.

The vulnerable code can be found on GitHub within the mpc-tool repository at this link. The code is presented below.

```
/**
 * Generate a new AWS S3 pre signed url to upload/download an object (GET).
 */
export const generateGetObjectPreSignedUrl = functions.https.onCall(async
↪  (data: any): Promise<any> => {
    if (!data.bucketName || !data.objectKey)
    ↪  logMsg(GENERIC_ERRORS.GENERR_MISSING_INPUT, MsgType.ERROR)

    // Connect w/ S3.
    const S3 = await getS3Client()

    // Prepare the command.
    const command = new GetObjectCommand({ Bucket: data.bucketName, Key:
    ↪  data.objectKey })

    // Get the PreSignedUrl.
    const url = await getSignedUrl(S3, command, { expiresIn:
    ↪  Number(process.env.AWS_PRESIGNED_URL_EXPIRATION!) })

    logMsg(`Single Pre-Signed URL ${url}`, MsgType.LOG)

    return url
})
```

3.3.2 Risk Severity

Impact - High - an attacker could download sensitive data.
Likelihood - Low - an attacker would first need to enumerate valid bucket and object names.

3.3.3 Proof of Concept

```
it("should not be possible to call this function when not authenticated",
↪  async () => {
    // @todo enforce auth check in the cloud function
    await signOut(userAuth)
    console.log(await generateGetObjectPreSignedUrl(userFunctions,
    ↪  bucketName, objectName))
    expect(generateGetObjectPreSignedUrl(userFunctions, bucketName,
    ↪  objectName)).to.be.rejected
})
```

This provides a pre-signed URL which allows us to download the file.

3.3.4 Recommendations

It is recommended to enforce access control onto the Cloud Function. First of all, this should only be callable by authenticated users, and each of the bucket/object combination requested, should be cross-checked against opened or closed ceremonies.

Furthermore, as a recommendation to the coordinator, AWS accounts for running ceremonies should be segregated from accounts used for other activity.

3.3.5 Remediation

This issue was fixed with commit 388caac7bb8a574f804639733738adcc3d730978.

3.4 Lack of file type verification

3.4.1 Description

Contributors to a ceremony are required to download the previously computed zKey before they can start their contribution. This zKey file is never verified to be the expected file. This could allow a malicious coordinator to trick contributors into downloading malicious files, and potentially executing code on their machines.

3.4.2 Risk Severity

Impact: Medium - a malicious coordinator might be able to coerce a contributor to download malicious files.
Likelihood: Low - exploitation of this issue would require a user to execute any downloaded files.

3.4.3 Code Location

The downloadContribution helper function downloads a file from S3, and is used within the contribution command.

3.4.4 Recommendations

It is recommended that the cli code is amended to ensure that correct verification of the downloaded file is enforced. This would ensure that a contributor who downloads the cli from GitHub, is safe against a malicious coordinator which attempts to have participants download and execute malicious code.
Verification can be achieved using 'snarkJS' verify function on the download zKey file. Additionally, a public record of the SHA256 hash of the previously computed zKey could be created after each contribution, and used to verify that the zKey downloaded by the next contributor is legitimate.

3.5 Email Enumeration

3.5.1 Description

Firebase email and password authentication does not prevent email enumeration by default. This could allow a user to generate a list of valid email addresses of registed users, and later launch further attacks, such as phishing campaigns on password bruteforce attacks to gain access to their accounts.

3.5.2 Risk Severity

Impact: Medium - an attacker would have access to email addresses of other users.
Likelihood: Low - this would require a coordinator enabling username/password authentication to Firebase.

3.5.3 Proof of Concept

The following code can be used to authenticate to Firebase using username and password, and as shown in the error, the server confirms that the password is wrong. On the other hand, providing a wrong email will result in an error message such as: "Firebase: Error (auth/wrong-email).".

```
it("should not let anyone authenticate with the wrong password", async () =>
↪  {
    const wrongPassword = "wrongPassword"
    expect(signInWithEmailAndPassword(userAuth, users[0].data.email,
    ↪  wrongPassword))
    .to.be.rejectedWith("Firebase: Error (auth/wrong-password).")
})
```

```
"Firebase: Error (auth/invalid-email)."
```

An automated tool could be easily developed to exploit this issue and compile a list of valid email addresses.

3.5.4 Recommendations

It is recommended that the coordinator, should they enable username/password authentication to Firebase, enable the email enumeration protection.
A guide can be found on the official documentation.