mnemonic

CUSTOMER CONFIDENTIAL

TECHNICAL REPORT

SOURCE CODE AUDIT:

EVA FOLKEAVSTEMNING

Valgdirektoratet

Place	Oslo
Date	2023-12-06
Version	1.0
Author	Erlend Leiknes, Tor Erling Bjørstad, Finn Johansen

Executive summary

mnemonic is carrying out a series of source code audits for the Norwegian Directorate of Elections (Valgdirektoratet). The current review is done after of the municipal and county council elections in 2023. The goal of the project is to provide an independent assessment of the security and quality of the source code and architecture across the EVA (*Elektronisk Valg*-Administrasjon) system portfolio given the changes from the previous iteration of source code reviews and respective changes done by Valgdirektoratet.

The current report describes the results for the assessment of EVA Folkeavstemning. This is a completely new application used to conduct referendums and plebiscites in a secure, flexible, and re-usable fashion over the Internet. Folkeavstemning consists of multiple sub-applications:

- "Folkeavstemning" is a small .NET application that allows eligible voters to generate digital ballots
- "Stemmemottak" is a small .NET application that receives the encrypted and signed ballots from Folkeavstemning. The results of the election will also be generated from this application.
- "Manntall" is a new integration with the National Registry (Folkeregisteret) in order to populate the electoral register with eligible voters
- The "frontend" application is written in TypeScript, and runs in the browser. It allows voters to encrypt and sign ballots.

The assessment has been carried out in November 2023, and is based on source code and documentation provided to mnemonic by Valgdirektoratet.

In order to assess the code base, mnemonic has relied on manual analysis in combination with open-source tools for Static Application Security Testing (SAST) tools and Software Composition Analysis (SCA) tool. Findings from the tools have been triaged and evaluated by mnemonic to avoid false positives and ensure that findings are correct and placed into context. We have focused on the overall system design, logical data flows, access control logic, and input handling, as these are areas where we would expect to find potential problems.

This report describes the technical findings and results from the audit. It contains the observations and findings that indicate security vulnerabilities, deviations from recommended practice, and potential hazards that have been identified in the source code.

mnemonic recommends that Valgdirektoratet should review the report with our findings and recommendations, in order to determine potential impacts, and evaluate the need to make either short- or long-term improvements or changes.

Summary of findings

The audit has led to 8 findings which are documented in this report. Based on mnemonic's initial evaluation, one finding is [4-critical], two is [2-medium] and the remaining [1-low] or [0-info]. The [4-critica] finding was discovered by the developers in Valgdirektoratet around the same time as mnemonic reported it.

One of the medium findings are related to insecure validation of password, and the storage of said password on the server. The second medium finding is related to the usage of the same key for encrypting and signing a vote.

In addition to the medium findings, we have observed several improvements in the code base that is not directly a security risk, however it can improve the solution by implementing the recommendations. These are related to logging key events in the application, and validation of ballot data and signatures of votes.

In addition to the specific findings, we provide some general discussion on the limitations by the chosen approach and the cryptographic security of the design and implementation.

Summary of recommendations

The main recommendation for EVA Folkeavstemning is to hash or encrypt all passwords that are stored on the servers. In addition, we recommend that different keys are used for signing and encrypting a vote – ensuring that it is not possible to recover the plaintext of a vote with a signed vote's key.

In addition to this, we have some general observations that are discussed in chapter 2.1, in which we recommend that Valgdirektoratet assess the integrity and trust that must be in place for EVA Folkeavstemning to be a successful solution for digital voting. From mnemonic's perspective we recommend supporting an open public discussion to build trust in the system, and that Valgdirektoratet does not give the appearance of having "something to hide".

Beyond this, we recommend that Valgdirektoratet validate and assess the report findings and associated recommendations one by one. While mnemonic's analysis is provided in the report, Valgdirektoratet's in-depth knowledge of the code base for Folkeavstemning will always exceed ours, and is likely to provide additional context about both impacts and proposed solutions.

Table of Contents

1 Introduction

1.1 Introduction to the report

mnemonic has performed a security assessment of Folkeavstemning and Stemmemottak, with basis in a source code review. The applications are made and maintained by Norwegian Directorate of Elections (Valgdirektoratet).

The current document describes the results of the activity. It includes mnemonic's analysis and recommendations for each of the observations and findings. mnemonic suggests using the document to identify and prioritize "quick-wins" and urgent remediation activities, as well as to support operational risk management and strategic security improvement initiatives.

A central goal of security testing is to discover and document as many security-related bugs as possible within the given scope, under constraints such as time and budget. The reader should keep in mind that security testing is a non-deterministic quality assurance activity which contains elements of both skill and luck. It is not likely that all bugs and vulnerabilities in any non-trivial software system will be discovered through testing. While the test report gives insight into the security, there may still exist additional latent vulnerabilities in the system.

1.2 Structure of the report

The report consists of 4 main parts, as well as appendices.

1. Chapter 1 (the current chapter) describes the overall context and structure of the report
2. Chapter 2 describes aggregated information about all findings and recommendations
3. Chapter 3 provides detailed information about each finding, sorted by mnemonic's evaluation of criticality.
4. Chapter 4 provides information about how the test was carried out

A. Appendix A provides supporting documentation regarding mnemonic's classification methodology for discovered vulnerabilities

2 Summary of findings and observations

2.1 General observations

2.1.1 Introduction

The apparent goal of EVA Folkeavstemning is to conduct referendums and plebiscites in a secure, flexible, and re-usable fashion through an on-line Internet-based solution. National referendums are very uncommon in Norway, but have been used in some cases, notably in the question of Norwegian membership of the European Union. However, local referendums have been held frequently over the last years, notably in municipalities considering whether to merge or split.

How this type of vote should be conducted is not strictly specified under Norwegian law, as opposed to municipal, regional, and national elections, which are regulated by the Elections Act (Valgloven). A trial solution for Internet voting was piloted in national elections in 2011 and 2013, but was later shelved, and there are (as far as we are aware) no plans to re-introduce this.

Local referendums have traditionally been handled differently by the relevant authorities, mostly using traditional paper-based voting, but sometimes also using on-line voting systems from commercial vendors. Having a common solution in place could help simplify future referendums.

Because of the lack of regulation, overall (security) requirements for a voting solution are open to some interpretation. As viewed from mnemonic's external perspective, the following are examples (non-exhaustive) of likely requirements:

- **Confidentiality**: Ballots cannot be linked to the voter who submitted it
- **Integrity**: Only people listed in the electoral register for the election are allowed to vote
- **Integrity**: Each voter is only allowed to vote once (in case multiple ballots are submitted, only one should be counted)
- **Integrity**: All votes are counted correctly according to voter intention
- **Integrity**: It is possible for an external observer to verify that the count is correct
- **Integrity**: Sufficient log data is available to determine that the voting system is operating correctly at all times
- **Availability**: Users interacting with the voting system should not be able to impact other users

Several of these goals are in some degree of conflict. For example, it is quite hard to build a system that is truly anonymous and at the same time also verifiable. Because of this, secure online voting systems is a long-standing topic of academic research, and there exists a large variety of existing literature and proposed schemes. A leading research group is based in Trondheim at NTNU.

It is important to understand that Folkeavstemning does **not** attempt to implement an advanced cryptographic voting scheme, with the associated security guarantees. Instead, it uses a fairly simple blind signature scheme, with an aim to provide some or partial privacy and end-user verifiability, while re-using existing digital infrastructure such as ID-porten for electronic identification and population data from the National Registry to build the electoral register.

2.1.2 Limitations of chosen approach

Using an on-line, Internet-based platform for referendums, as well as choosing to use a simple and straight-forward cryptographic protocol, has security consequences that are difficult to mitigate. These must be well understood and acceptable to stakeholders.

- The voting system, and by extension, Valgdirektoratet (who produces and operates it), must be both trustworthy and trusted by the voters and other stakeholders. This is particularly important because Valgdirektoratet controls the *entire* system; internal boundaries between the Folkeavstemning and Stemmemottak back-ends are softly enforced, and invisible to the end users.
    - The system must be trusted to maintain the electoral register, collect ballots, and count the votes, and it must do so fairly, correctly, and completely.
    - Since the system can access the secret key used to sign ballots, as well as the electoral registry, Valgdirektoratet would *in principle be able to* generate arbitrary votes in the election
    - Despite technical countermeasures, Valgdirektoratet will also most likely *in principle* be able to identify individual votes, based on e.g. log data
- Because the first (on-line) vote cast is the one that will be counted, the system opens for several potential abuse-cases with no clear mitigations:
    - The solution does not prevent vote-selling or coercion
    - Social manipulation and phishing techniques could be used to steal a user's vote, similarly to known cases of BankID fraud
    - If technical vulnerabilities such as clickjacking or XSS/CSRF are present in the application, they could also be exploited by malicious actors to submit ballots on behalf of a logged-in user
- Certain types of data may become unexpectedly sensitive in a voting system. This includes HTTP access logs, as they can be used to correlate between service calls to the Folkeavstemning and Stemmemottak service backends from a given IP address
    - We suggest making a separate assessment of the potential privacy implications of the solution, if this has not already been carried out

On a general note, the use of an Internet-based voting scheme will be controversial in some circles, and seems likely to attract a significant amount of public interest. This is doubly the case because the referendums that are planned for early 2024 are also *politically* controversial.

Valgdirektoratet should be prepared for public speculation about the security and trustworthiness of EVA Folkeavstemning, as well as data access requests from interested parties. mnemonic's suggestion would be to meet this with openness and transparency - ideally being proactive rather than reactive - to avoid giving even a suggestion that there is "something to hide", which could be used to create a controversy out of nothing and provide fertile ground for conspiracy theorists.

2.1.3 Description of cryptographic solution

The cryptographic solution in EVA Folkeavstemning is based on RSA *blind signatures*. This is a technique where messages are disguised ("blinded") before they are signed, in order to prevent the signer from learning the contents of the message. The overall concept was invented by David Chaum in 1983 and is well-known in the literature. It is a common building block in

cryptographic voting schemes, and is also used by some cryptocurrencies, in order to provide added privacy.

As indicated by the limitations discussed in the previous paragraph, as well as due to certain implementational weaknesses described in this report, it is important to consider whether the cryptographic solution produces sufficient added value in terms of security and privacy.

In brief, creating a ballot in EVA Folkeavstemning is done as follows:

1. The voter creates a ballot consisting of their vote and a random fingerprint (which is used for randomization and verifiability), and encrypt it using the election system's public key
2. The voter creates a blinded message, by hashing their encrypted ballot and multiplying with a random element computed in a specific way. The resulting value is essentially random and leaks no information about the contents of the ballot.
3. The Folkeavstemning backend operates the attestation service. It takes the voter ID and blinded message, verifies that all formal requirements are satisfied (e.g. that the voter is eligible and has not already voted), and signs the blinded message with its private key. It then returns the signature.
4. By utilizing the malleability of (unpadded) RSA signatures, the voter unblinds the signature in a way that they now have a valid RSA signature on the encrypted ballot.
5. The Stemmegivning backend operates the verification services, and works as a ballot box. It is unauthenticated, and does not know who the end user is. However, when it receives an encrypted and signed ballot, it can verify that the signature is genuine.
6. After the election has ended, all ballots are decrypted and tallied by Stemmegivning.
7. The Fingerprint strings included with the ballots can be published, so that each individual voter can verify that their vote was counted.

As implemented, EVA Folkeavstemning follows the Wikipedia article[1] on blind signatures closely. To avoid the blinding attack described there, messages are hashed using SHA256. The 2048-bit RSA keys used for signing and encryption are included in the source code for Stemmegivning and Folkeavstemning.

In typical applications of blind signatures, the attestation service (Folkeavstemning) and verification service (Stemmegivning) would be independent parties that do not collude. This is not really the case here, since both are controlled by Valgdirektoratet and running in a shared environment. This implies that Valgdirektoratet needs to be highly trusted, and reduces the value of the blinding mechanism to provide voter privacy.

A limitation of blind signature schemes is a lack of standardization. Even though the technique is well known, there are few public standards to build on.

- ISO 18370:2016 standardizes blind signatures, but is not openly available.
- RFC 9474 standardizes blind signatures, but was only published in October 2023

In RFC 9474 they use a different blind signature algorithm, based on standard RSA-PSS signatures. This aligns with the most recent version of other standards including TLS 1.3, X.509, and PKCS #1.

[1] https://en.wikipedia.org/wiki/Blind_signature

Folkeavstemning simply hashes the (encrypted) ballot and performs the RSA operations directly on the result. This is the "traditional" hash-then-sign approach, which is generally not used in practice. Because there is no output padding (as in e.g. PKCS #1 v1.5 signatures) and the output range of the hash does not match the full domain of the RSA function (as in RSA-FDH signatures), the security proof for this construction is (probably) quite inexact[2]. Although we have not tried to quantify this, using different cryptographic primitive for blind signatures might give more confidence in the overall security of the blind signature scheme, and more explicit security guarantees as well.

We also note that Folkeavstemning uses the same RSA keypair for signing and encryption. With blind signatures, this is generally quite dangerous, because the attestation service is "signing" unknown inputs which may in fact be encrypted ciphertexts chosen by an attacker.

In sum, the added security and privacy of using blind signatures is somewhat limited as long as Valgdirektoratet is controlling both the attestation service and the ballot box, and is able to cross-correlate activity between the two. Use of blind signatures would make more sense in a situation where multiple independent parties were in charge of different parts of the election process. Other alternatives would be to look at a simpler (non-cryptographic) solution, with Valgdirektoratet acting as an explicitly trusted party to operate the election, as well as more complex end-to-end auditable cryptographic voting schemes.

2.2 Classification of technical findings

The following table gives an overview of which types of vulnerabilities that were found in the test. Categories are based on the OWASP Top 10 Application Security Risks (2021).

Table 1. Summary of findings, by category

Category	Finding	Status
A01:2021 – Broken Access Control Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification, or destruction of all data or performing a business function outside the user's limits. Access control is only effective in trusted server-side code or server-less API, where the attacker cannot modify the access control check or metadata.		Not found
A02:2021 – Cryptographic Failures Proper use of cryptography protects data in transit and at rest. Failures typically lead to exposure of confidential or sensitive information, and breach of privacy laws or regulatory compliance.	3.1, 3.3, 3.4, 3.5, 3.8	**Found**

[2] Several papers by Bellare and Rogaway explore the security of these schemes in the random oracle model, see e.g. "The Exact Security of Digital Signatures How to Sign with RSA and Rabin" from 1996.

Category	Finding	Status
A03:2021 – Injection Applications are vulnerable to Injection attacks when data is used raw in dynamic code, either in the application itself or when handing over data to some other component. Failure to use safe APIs or escaping/encoding output when safe APIs are not available typically leads to access to back-end component (for server-side Injections) or modification of user interface (for HTML Injection or Cross-Site Scripting).	3.3	**Found**
A04:2021 – Insecure Design Secure design is a culture and methodology that constantly evaluates threats and ensures that code is robustly designed and tested to prevent known attack methods. An insecure design cannot be fixed by a perfect implementation as by definition, needed security controls were never created to defend against specific attacks. One of the factors that contribute to insecure design is the lack of business risk profiling inherent in the software or system being developed, and thus the failure to determine what level of security design is required. Secure software requires a secure development lifecycle, some form of secure design pattern, paved road methodology, secured component library, tooling, and threat modeling.	0, 3.7	**Found**
A05:2021 – Security Misconfiguration Security misconfiguration is commonly a result of insecure default configurations, incomplete or ad hoc configurations, open cloud storage, misconfigured HTTP headers, and verbose error messages containing sensitive information.		Not Found
A06:2021 – Vulnerable and Outdated Components Components, such as libraries, frameworks, and other software modules, run with the same privileges as the application. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications and APIs using components with known vulnerabilities may undermine application defenses and enable various attacks and impacts.		Not found
A07:2021 – Identification and Authentication Failures Confirmation of the user's identity, authentication, and session management is critical to protect against authentication-related attacks. If application functions related to identification, authentication, and session management are implemented incorrectly, it allow attackers to compromise username-passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities temporarily or permanently.		Not found
A08:2021 – Software and Data Integrity Failures Enforcing software integrity is needed to protect against introduction of attacker-controlled code or configuration during development, build, and deployment, e.g. in CI/CD pipelines and infrastructure as code. Data integrity failures includes deserialization flaws, which can lead to remote code execution.		Not found

Category	Finding	Status
A09:2021 – Security Logging and Monitoring Failures Insufficient logging and monitoring, coupled with missing or ineffective integration with incident response, allows attackers to further attack systems, maintain persistence, pivot to more systems, and tamper, extract, or destroy data. Most breach studies show time to detect a breach is over 200 days, typically detected by external parties rather than internal processes or monitoring.	3.6	**Found**
A10:2021 – Server-Side Request Forgery (SSRF) SSRF flaws occur whenever a web application is fetching a remote resource without validating the user-supplied URL. It allows an attacker to coerce the application to send a crafted request to an unexpected destination, even when protected by a firewall, VPN, or another type of network access control list (ACL).		Not found

2.3 Summary of technical recommendations

The following table summarizes all detailed recommendations made in the report. For additional context, please review the detailed finding descriptions which are given in the next chapter.

3 Detailed findings and observations – Folkeavstemning

The security test aims to discover, demonstrate, and document technical as well as logical vulnerabilities in the system, which can be exploited to cause a loss. This section describes all such findings in detail, as well as additional observations made during the test that are relevant for understanding the security posture of the system on a technical level.

All findings have been ranked based on our evaluation of criticality and impact, which takes into account the technical impact of each finding as well as our understanding of the system context, and gives a score on a scale from [4-critical] to [0-info]. For more information about our evaluation criteria, please see Appendix A:

Findings are generally limited by the scope and level of access given during the test. In particular, we do not actively try to find security issues outside the agreed scope.

3.1 Race conditions in vote casting and counting [4-critical]

Location

- Folkeavstemning
  - StemmegivningController.cs – function AvleggStemme
- Stemmemottak
  - StemmemottakEndpoint.cs - function RegistrerKryptertStemme

Category

- A02:2021 – Cryptographic Failures

Summary

Note: the same issues were discovered independently by Valgdirektoratet at the approximately same time as mnemonic reported this. The vulnerabilities were according to the developers in Valgdirektoratet fixed before this report was delivered.

Both Folkeavstemning and Stemmemottak have race condition vulnerabilities. Exploiting these races could lead to an attacker generating multiple valid signed ballots, and/or submitting multiple identical ballots which are accepted by the system.

Both are classical "time-of-check to time-of-use" (TOCTOU) vulnerabilities, where a condition is checked, before some computation is done, and an update is carried out asynchronously. This means that if multiple requests arrive at the same time, they will all be considered valid.

As an example, the code in Folkeavstemning first checks if a voter has voted, then computes a signature, and finally updates the database with the new voter state.

Detailed description

The vulnerable function AvleggStemme in Stemmemottak is found in the class StemmegivningController.cs. The underlying issue with this code is that the time of checking of a user has voted and saving the vote to the database are asynchronous checks. The timing attack

occurs when an attacker sends multiple duplicate votes to this endpoint, managing to get past the check for if the person has voted *before* the votes are added to the database.

Tabell 1. Vulnerable code in StemmegivningController.cs, function AvleggStemme

```
var manntallsnummer = await _manntallClient.FindManntallsnummerForPerson(folkeavstemningId, identity);
      var harStemt = await _context.Stemmegivninger.AnyAsync(x => x.Manntallsnummer ==
manntallsnummer && x.FolkeavstemningId == folkeavstemningId);

      if (harStemt)
      {
         _logger.LogDebug("Har stemt fra før");
         return Conflict();
      }

      if (manntallsnummer == null)
      {
         _logger.LogDebug("Har ikke stemmerett");
         return Forbid();
      }

      if (!_keys.Value.TryGetValue(folkeavstemningId.ToLinuxPortableCharacterSet(), out var keyLocations))
      {
         _logger.LogError("Mangler nøkkel-konfigurasjon for {FolkeavstemningId}", folkeavstemningId);
         return Problem("Mangler nøkkel-konfigurasjon");
      }

      var signature = await SignData(keyLocations, stemmepakke);

      await _context.Stemmegivninger.AddAsync(new Database.Stemmegivning
      {
         Manntallsnummer = manntallsnummer, FolkeavstemningId = folkeavstemningId
      });

      await _context.SaveChangesAsync();

      _logger.LogDebug("Har avlagt stemme");

      return Ok(signature);
```

Abusing this function would yield multiple valid blind signatures, which could be used by an attacker to create and submit multiple valid ballots. The Stemmemottak service is unable to detect this situation, as long as the ballots are not identical (i.e. if they have different fingerprint strings).

In the same manner in the function RegistrerKryptertStemme, the check to see if a vote exists in the database is not synchronous with adding the vote to the database. Again, an attacker can send multiple valid votes to this endpoint, and the check for if a vote exists will occur after the votes are added to the database.

Tabell 2. Vulnerable code in StemmemottakEndpoint.cs, function RegisterKryptertStemme

```
try
    {
        var kryptertStemme = ValidateStemmedata(folkeavstemningId, dto);

        if (!keys.Value.TryGetValue(folkeavstemningId.ToLinuxPortableCharacterSet(), out var keyLocations))
        {
            logger.LogError("Mangler nøkkel-konfigurasjon for {FolkeavstemningId}", folkeavstemningId);
            return Results.Problem("Mangler nøkkel-konfigurasjon");
        }
        await ValiderSignatur(keyLocations, kryptertStemme);

        var stemmeEksisterer = context.Stemmer.Any(x => x.Signatur == dto.Signatur);
        if (stemmeEksisterer)
        {
            return Results.Conflict();
        }

        await context.Stemmer.AddAsync(kryptertStemme);
        await context.SaveChangesAsync();

        return Results.Ok();
    }
```

In this case the result of submitting several votes at the same time would potentially be all or many of the votes being persisted.

Recommendations

As a general rule, TOCTOU vulnerabilities can be solved by making operations atomic. This can either be done on the application layer, or by enforcing constraints on the database.

We understand that Valgdirektoratet has addressed this issue by adding unique constraints on the database, ensuring that only the first update to reach the database will succeed. In the former case, the voter ID should be unique, while in the second case, the encrypted ballot should be unique.

This should be sufficient to solve the issue, particularly given good test coverage.

Recommendation 1. [4-critical] Avoid race conditions by making operations atomic or by enforcing uniqueness on the database layer

3.2 String literal comparison of credentials [2-medium]

Location

- Shared
    - BasicAuthenticationHandler.cs – function HandleAuthenticateAsync

Category

- A04:2021 – Insecure Design

Summary

The basic authentication method does a string literal check of credentials from the authentication header. This entails that the username and password are stored in clear-text in the configuration files.

Detailed description

In the BasicAuthenticationHandler.cs class the function for handling an authentication request is implemented in the HandleAuthenticateAsync function. This function extracts the authentication header – which contains a base64 encoded string of the username and password passed with the request, as seen below.

Tabell 3 HandleAuthenticateAsync function in BasicAuthenticationHandler class

```
protected override Task<AuthenticateResult> HandleAuthenticateAsync()
   {
      try
      {
         var authHeader = AuthenticationHeaderValue.Parse(Request.Headers["Authorization"]);
         if (authHeader.Parameter != null)
         {
            var credentials =
Encoding.UTF8.GetString(Convert.FromBase64String(authHeader.Parameter)).Split(':');
            var username = credentials.FirstOrDefault();
            var password = credentials.LastOrDefault();

            if (username == Options.Username && password == Options.Password)
            {
               var claims = new[] { new Claim(ClaimTypes.Name, username ?? "Anonymous user"), new
Claim(ClaimTypes.Role, "Admin")};
               var identity = new ClaimsIdentity(claims, "basic");
               var principal = new GenericPrincipal(identity, new []{"Admin"});
               var ticket = new AuthenticationTicket(principal, Scheme.Name);
               return Task.FromResult(AuthenticateResult.Success(ticket));
            }
         }
      }
      catch (Exception ex)
      {
         return Task.FromResult(AuthenticateResult.Fail($"Authentication failed: {ex.Message}"));
      }
```

```
        return Task.FromResult(AuthenticateResult.Fail("Invalid credentials"));
    }
```

The function does a string literal comparison of the provided username and password, meaning that the username and password are stored in clear-text in the configuration files on the web server.

In addition to being bad practice to have credentials stored in plain text, a string literal check can potentially be vulnerable to string attacks, such as encoding or other string based attacks.

Recommendations

mnemonic recommends that the credentials used for the backend servers are stored hashed or encrypted, and that the comparison is implemented to reflect this change.

In addition, Valgdirektoratet should consider implementing a time-constant comparison – to remove timing based password brute force attacks.

Recommendation 2. [2-medium] Store credentials hashed or encrypted for the basic authentication to backend servers

Recommendation 3. [2-medium] Consider implementing a time-constant password comparison to prevent timing based password attacks

3.3 Using the attestation service as a decryption oracle [2-medium]

Location

- Folkeavstemning
    - StemmegivningController.cs

Category

- A02:2021 – Cryptographic Failures

Summary

Because the same keypair is used for encryption and signing, an attacker is able to decrypt a single vote, at the cost of spoiling their ballot.

Assuming that finding 3.1 is patched, this can only be done once per user, which reduces the severity of the vulnerability. At the same time, we believe that this behavior is highly undesirable, and easily preventable by using distinct keys for signing and encrypting.

Detailed description

By submitting an unknown ciphertext c in a stemmepakke structure to the AvleggStemme endpoint, the attacker is able to compute c^d mod N, where d is the secret exponent.

If the ciphertext input is a kryptertStemme which was encrypted using RSA-OAEP, then the attacker can use the signature service to decrypt the OAEP-padded structure, and subsequently recover the plaintext.

Recommendations

Use separate keys for signing and encryption.

Recommendation 4. [2-medium] Use separate keys for signing and decryption

3.4 Stemmemottak is unable to validate ballot data [1-low]

Location

- Stemmemottak
    - ResultatEndpoint.ts - function DekrypterStemmer

Category

- A02:2021 – Cryptographic Failures
- A03:2021 – Injection

Summary

The ballots are encrypted, and they are only decrypted when the election is finished.

Since there are no restrictions on what data *could* be submitted, it is important to have sufficient input validation on the data that is received after decryption.

Validation appears to be implicitly enforced through the type system, but it might be more robust to check explicitly that decrypted data is of the expected format.

Detailed description

Decryption happens in the ResultatEndpoint service.

```
        var krypterteStemmer = await GetKrypterteStemmer(context, folkeavstemningId);
        var dekrypterteStemmer = DekrypterStemmer(krypterteStemmer, key);
        var stemmepakker = DeserialiserStemmer(dekrypterteStemmer);
        var resultat = LagResultat(stemmepakker);
```

It is supported by the following convenience functions.

```
    private static ICollection<byte[]?> DekrypterStemmer(KryptertStemme[] krypterteStemmer, RSA
privateKey)
    {
        var dekrypterteStemmer = krypterteStemmer
            .AsParallel()
            .Select(x => Decrypt(privateKey, x))
            .ToArray();
        return dekrypterteStemmer;
    }
```

```
private static ICollection<Stemmepakke?> DeserialiserStemmer(ICollection<byte[]?> dekrypterteStemmer)
    {
        var stemmepakker = dekrypterteStemmer.Select(x => x == null ? null :
JsonSerializer.Deserialize<Stemmepakke>(x)).ToArray();
        return stemmepakker;
    }
```

```
private static Dictionary<string, int> LagResultat(IEnumerable<Stemmepakke?> stemmepakker)
    {
```

```
        var resultat = stemmepakker
            .Select(x => x?.Valg ?? "<UGYLDIGE STEMMER>")
            .GroupBy(x => x)
            .Select(x => new { x.Key, Count = x.Count() })
            .ToDictionary(x => x.Key, x => x.Count);
        return resultat;
    }
```

It is not 100% clear what happens if the contents of the encrypted ballots are in fact something else than what it should be.

It *seems* to us that it is not possible to successfully inject malicious data in the counting step, but it would be more robust if each ballot is explicitly validated, with anything on the wrong format explicitly logged and thrown out of the count. This would also serve as a safety valve to ensure that unexpected numbers of invalid ballots cause an alert.

Recommendations

Make validation explicit during ballot counting, and ensure that all ballots on the wrong format are logged and thrown out.

Recommendation 5. [1-low] Make validation explicit during ballot counting

Recommendation 6. [1-low] Ensure that ballots of the wrong format are explicitly logged and rejected

3.5 Signatures are not checked during counting [0-info]

Location

- Stemmemottak
    - ResultatEndpoint.ts - function DekrypterStemmer

Category

- A02:2021 – Cryptographic Failures

Summary

When Stemmemottak decrypts the ballots, it does not check the signatures. The signature is only checked when the ballot is received.

This means that if anyone has altered the ballots after they were stored in the database and before they are counted, this will not be detected by the system.

In effect, the signature is only used by Stemmemottak to verify that the ballot was signed, but not to provide integrity later on in the process.

Recommendations

Consider verifying signatures at the time of counting, not only when the ballots are received.

Recommendation 7. [0-info] Consider verifying signatures at the time of counting, not only when the ballots are received.

3.6 Use of signing key is not logged [0-info]

Location

- Folkeavstemning
    - StemmegivningController.cs - functions SignData and GetKey

Category

- A09:2021 – Security logging and monitoring failures

Summary

When the secret key is used (e.g. to perform blind signatures), this usage is not logged. Access to this key should be tracked explicitly, and signing actions should be logged in a way that makes it possible to review the total number of signatures made.

Recommendations

Ensure that use of signing key is logged and audited.

Recommendation 8. [0-info] Ensure that use of signing key is logged and audited.

3.7 Unclear logic to generate manntallsnummer [0-info]

Location

- Manntall
    - GenerateManntallsnummerEndpoint.cs - GenerateManntallsnummer

Category

- A04:2021 – Insecure Design

Summary

Rationale for generating manntallsnummer is not clear.

Detailed description

The code claims that the probability of guessing a valid manntallsnummer is 4%. This is unclear.

```
        logger.LogInformation("Genererer manntallsnummer for {Folkeavstemning}", folkeavstemning);

        var personer = await context.Personer.GetAllPersonerIFolkeavstemning(folkeavstemningId, token);

        var manntallsnummer = folkeavstemning.ManntallsnummerStart + Random.Shared.Next(0, 10000);

        foreach (var person in personer.OrderBy(_ => Random.Shared.Next()))
        {
            if (StemmerettChecker.HarStemmerett(person, folkeavstemning))
```

```
            {
                // ingen garantert sekvens - sannsynlighet for å gjette et gyldig manntallsnummer er 4%
                manntallsnummer += 1 + Random.Shared.Next(9);
                person.Manntallsnummer = $"{manntallsnummer}";
            }
        }

        await context.SaveChangesAsync(token);

        logger.LogInformation("Manntallsnummer er generert for {Folkeavstemning} - siste manntallsnummer
er {Manntallsnummer}", folkeavstemning, manntallsnummer);
```

As far as we can tell, the code gets the list of people in the electoral registry, chooses a random starting integer between 0 and 10000, and allocates electoral identifies in a random order with a random increment between 1 and 9.

Given one known manntallsnummer (e.g. your own), the chance of guessing a valid number will be much higher than 4% by guessing nearby integers in both directions.

Given no known manntallsnummer, the density (given a random increment between 1 and 9) will still be about 20%, which means that guessing random numbers above the minimum threshold will be likely to succeed.

On the other hand, it is not clear what security purpose keeping these identifiers secret provides.

Recommendations

Consider security / privacy rationale for how manntallsnummer are generated, and whether existing implementation supports it.

Recommendation 9. [0-info] Consider the security and privacy rationale for how manntallsnummer are generated.

3.8 Functions do not draw from complete range [0-info]

Location

- frontend
    - blindvote.ts

Category

- A02:2021 – Cryptographic Failures

Summary

Weaknesses in how certain values are computed may reduce the theoretical security level of the blind signature scheme.

Detailed description

The randomizer is computed as a 512-bit secure random element:

```
        r = new BigInteger(secureRandom(64)).mod(N);
```

The message to be blinded is hashed to a 256-bit random element:

```
async function messageToHash(message: string) {
    const utf8 = new TextEncoder().encode(message);
    const hashBuffer = await crypto.subtle.digest('SHA-256', utf8);
    const hashArray = Array.from(new Uint8Array(hashBuffer));
    const hashHex = hashArray
        .map((bytes) => bytes.toString(16).padStart(2, '0'))
        .join('');
    return hashHex;
}
```

Ideally both of these should be uniformly random group elements, i.e. from the full 2048-bit range. This would give the RSA Full Domain Hash signature, which is well studied. Note that hashing / mapping to a group element will be key-dependent, and, while constructions to do so based on a regular hash or RNG exist, it introduces additional complexity. The easiest way to address this might be to change blind signature schemes to something that has been standardized and does not require customized hashing, e.g. to RFC 9474 based on RSA-PSS.

We have not tried to quantify how much this non-uniformity reduces the overall security level of the signature, but we note that the security argument for textbook hash-and-sign RSA signatures typically assumes uniformity over the whole group. For further discussion, the original security proofs for hash-and-sign / Full-Domain Hash constructions (in the random oracle model) are given in the papers by Bellare and Rogaway from the mid-1990s[3,4].

Recommendations

Consider changing the blind signature scheme to RSA PSS as defined in RFC 9474. Alternately ensure that group elements are selected uniformly at random from the correct distribution.

Recommendation 10. [0-info] Consider changing the blind signature scheme to RSA PSS as defined in RFC 9474.

[3] Bellare, Rogaway: Random Oracles are Practical: A Paradigm for Designing Efficient Protocols (1993). https://cseweb.ucsd.edu/~mihir/papers/ro.pdf
[4] Bellare, Rogaway: The Exact Security of Digital Signatures -- How to Sign with RSA and Rabin (1996). https://web.cs.ucdavis.edu/~rogaway/papers/exact.pdf

4 About the report

4.1 Test execution

Project name	Folkeavstemning – Source Code Review
Client	**Valgdirektoratet** Ann Karin Pedersen (Ann.Karin.Pedersen@valg.no)
Conducted by	mnemonic AS
Consultants	Erlend Leiknes, Tor Erling Bjørstad, Finn Johansen
Internal QA	Peder Grundvold
Start date	2023-11-13
End date	2023-12-01

4.2 Document version control

The current revision of this document is **1.0**. All major revisions are documented in the table below.

Table 2. Document version control

Version	Date	Consultant(s)	Comment
1.0	2023-12-06	Erlend Leiknes, Tor Erling Bjørstad, Finn Johansen	Final report delivered to Valgdirektoratet.

4.3 Test scope

The test has covered the following scope:

- Folkeavstemning source code

4.4 Test scenarios and mis-use cases

No specific scenarios and mis-use cases were discussed for the test. The test has simulated a knowledgeable and skilled actor attempting to explore the Valgresultat.no codebase and identify potential weak areas or vulnerabilities which could be used to impact the security properties of the solution.

4.5 Methodology and tools

mnemonic has conducted security and penetration testing, source code audits, and related services, ever since the company was founded in 2000. Our security testing methodology is based on the combination of open standards and collections of “industry best practice”, together with our own experience accumulated over the last 20 years. In addition to this, testing is supported by an extensive knowledge base, as well as internally developed tools and scripts.

Our methodology is supported by the processes “*P3003 Procedure for security testing*” and “*P3006 Use and maintenance of testing platform*” in mnemonic’s ISO 9001 / ISO 27001 certified quality and security management system, as well as associated templates and documentation.

Security testing conducted by mnemonic consist of the following phases:

1. **Preparation**: establish test context and scope, agree on terms of engagement and escalation routines, plan assessment activities, perform a functional review of the test objects
2. **Reconnaissance**: gather information about the test objects, based on available documentation, open source intelligence (OSINT), relevant technical standards, own research, and other sources if applicable
3. **Mapping**: scan and explore the relevant systems, using a combination of automated tools and manual or guided exploration, attempting to discover vulnerabilities or “trouble spots” along the way
4. **Verification and analysis**: evaluation and verification of initial findings, in-depth manual vulnerability assessment of selected areas, analysis of possible countermeasures and recommendations
5. **Reporting**: internal QA and presentation of findings, through a structured written report, and a joint debrief. Raw data can also be extracted and presented as an attachment to the report.
6. **(Re-test)**: if desirable, we also provide re-testing after remediation to verify that findings are closed

Central tools used in the current assessment include:

- OWASP Dependency Check
- SpotBugs
- Semgrep
- Visual Studio Code
- Visual Studio Community

4.6 Problems encountered

We discovered that Valgdirektoratet had changed their internal VPN and access policy, which meant that mnemonics clients could not access Valgdirektoratet’s systems. This led to the project starting somewhat later than originally planned, related to the access issues and seasonal flu.

Appendix A: Vulnerability classification

The purpose of this section is to describe how mnemonic ranks vulnerabilities. Unless otherwise agreed, all findings will be ranked based on mnemonic's evaluation of criticality and impact, according to the criteria described below. In the technical report, findings will be sorted based on this ranking (and optionally grouped per test object, for tests that cover large or complex scopes).

Alternate measures, such as CVSS v3 score, may be used as supplementary criteria, or as a primary ranking on request.

It is worth pointing out that vulnerability classification and ranking is far from an exact science. Because of this, we always encourage clients to review all findings that mnemonic reports, in order to understand their potential impacts and prioritize for remediation. Preferably, findings can be debriefed together with mnemonic's team as part of delivery. This allows both sides to clear up potential misunderstandings, and ensures that any loose ends are tied up.

A.1 Classification matrix

Rating	Description	Examples
4-critical	Finding or vulnerability with potentially critical impact on the system's confidentiality, integrity and/or availability, and with few or no mitigating factors. Critical findings are "showstoppers" that mnemonic thinks should be subject to immediate / emergency remediation, based on their potential impact.	• Remote code execution or similar vulnerabilities that enable a threat agent to break in through the system and target back-end resources • Authentication or authorization bypass leading to administrative or privileged access to essential functionality • Vulnerability that may lead to a total breach of key security properties of the system, such as a database compromise
3-high	Finding or vulnerability with potentially high impact on the system's confidentiality, integrity and/or availability. High-severity findings should normally be prioritized for analysis and possible remediation within an urgent time-frame.	• Otherwise critical findings with partial mitigation in place, e.g. requiring higher effort to exploit or leading only to partial loss of security • Systematic issues that have significant impact across multiple solutions or components • Persistent cross-site scripting (XSS) that may be used to target privileged users without user interaction
2-medium	Finding or vulnerability with a medium-level impact on the system's confidentiality, integrity and/or availability.	• Vulnerabilities that require significant effort, user interaction, and/or luck to exploit, such as reflected XSS or CSRF

	Medium-severity findings should be subject to remediation following usual vulnerability management and triage processes.	• Findings that provide a non-trivial benefit to a threat agent, but do not meet the criteria for high or critical impact
1-low	Finding or vulnerability with a potential low impact on the system's confidentiality, integrity and/or availability. Low-severity findings should be subject to remediation following usual vulnerability management and triage processes, but the initial evaluation is that they may be of lower priority.	• Findings that have a limited and localized security impact, such as a small information leak of non-sensitive data • Deviation from recommended security practice, or lacking defense-in-depth
0-info	In addition to the above, we may use a classification called 0-info, sometimes pointing out robust / well-designed solutions, unexpected behavior, bugs that do not have an obvious security impact, or to provide tips and hints for improvements or optimizations. Informational findings should be reviewed by relevant stakeholders.	• Findings that do not have a clear or proven security impact, but which we feel should be reviewed • Findings that do not have an impact within the test scope, but may be relevant elsewhere • Bugs without a clear security impact • Suggested future improvements • Unusual or particularly clever solutions

A.2 Rationale

While the impact of a *vulnerability* can often be classified purely in terms of technical impact, it is usually desirable to enrich the vulnerability with additional context. For example, the *practical exploitability* of a vulnerability may range from simple to unproven. A vulnerability may be directly accessible from the Internet, or located on an isolated internal network segment. Finally, the data or functionality that is being impacted may range from trivial to mission-critical.

As external penetration testers working on a limited-duration assessment, mnemonic will rarely gain a complete understanding about the business impact of a complex vulnerability, whereas its direct technical impact is easier to assess based on objective criteria. At the same time, our experience is that building on the penetration testers' experience and judgement to rank findings in context yields results that are more aligned with our customer's business need, than a purely generic framework such as CVSS.

Frameworks such as in CVSS v3 attempt to encode the impact of a vulnerability in purely technical terms. A CVSS score is an ordinal score from 0 to 10, which provides a ranking. In doing so, a lot of functional context is lost. Because it uses an ordinal scale, it is generally not useful (although quite common) to do arithmetic on CVSS scores; concepts such as the "*average CVSS*" is largely meaningless, and a CVSS 10.0 vulnerability is generally not "*10 times as bad*" as a CVSS 1.0 vulnerability. It is our experience that CVSS scores by themselves are often given too much weight, when prioritizing remediations.

To give an example, a CVSS 10.0 vulnerability in a system which is isolated from threats by its environment may pose a lot less risk to the business than a CVSS 7.5 vulnerability in an Internet-facing webserver with a published exploit. The derived measures of CVSS temporal and CVSS environmental scores try to enrich the base scoring with this type of context, but we feel that these measures are unwieldly in practice.

In our rankings, mnemonic tries to strike a balance between these perspectives. We use the technical impact of our findings as a starting point, but take into account the knowledge we have about the system, and how vulnerabilities may be exploited, when we make our evaluation.

Nevertheless, this is not an exact science. Because of this, we always encourage clients to review every finding that is documented in the report, based on their knowledge of their own business context, internal and external requirement, technical solution, risk appetite, and other constraints.

A.3 See also

Information on CVSS from the NIST National Vulnerability Database and FIRST:

- https://nvd.nist.gov/vuln-metrics/cvss
- https://www.first.org/cvss/user-guide

For an alternate approach, see BugCrowd's Vulnerability Rating Taxonomy (VRT):

- https://bugcrowd.com/vulnerability-rating-taxonomy