BIBIFI – Fix It

Remediation Report for Application Security Assessment

Prepared By: Jugal, Prasanna, Adith, Sarthak, Gitanshu, and Puru

Table of Contents

1. Introduction

This document outlines the security vulnerabilities identified in Group 2's secure file storage system by **Group 1** during the Break-it phase of the BIBIFI competition. It also describes the concrete fixes implemented and justifies certain design decisions based on trade-offs between security benefits and implementation complexity.

The vulnerabilities reported included:

1. **Weak Encryption – No Integrity Check (AES-256-CBC)**
2. **Partial Information Leak – File Names Are Exposed**

We addressed the first issue through a redesign of the authentication encryption mode by **replacing AES-256-CBC encryption with AES-256-GCM encryption**. For the second issue, we evaluated its practical impact and decided not to implement the fix due to its complexity and relatively lower severity.

2. Vulnerability Fixes

i) Weak Encryption with AES-256-CBC – No Integrity Check

Description:
Group 1 identified that our project used AES-256 in CBC mode for file encryption without any message authentication. This makes the system vulnerable to padding oracle attacks, bit-flipping, and ciphertext tampering, compromising both confidentiality and integrity.

Issue:

- Attackers could modify ciphertexts undetected.
- Lack of authentication allowed for corruption or partial leakage of plaintext.
- The system could behave unexpectedly under adversarially crafted inputs.

Fix:

We fully replaced AES-256-CBC with **AES-256-GCM**, an authenticated encryption mode that provides both confidentiality and integrity. The implementation includes:

- Storing the **authentication tag** along with the ciphertext.
- Using the `EVP_CIPHER_CTX_ctrl` interface to generate and verify the tag.
- Updating both encryption and decryption logic to use GCM's tag-based validation.

Security Benefits:

- Prevents tampering with ciphertext.
- Automatically rejects invalid ciphertexts during decryption.
- Eliminates the need for custom HMAC-based integrity checks.

Key Code Changes:

Old AES-256-CBC Encryption/Decryption - No Integrity Check:

Encryption before:

```
EVP_EncryptInit_ex(ctx, EVP_aes_256_cbc(), nullptr, key, iv);
EVP_EncryptUpdate(ctx, ciphertext.data(), &len, plaintext.data(), plaintext.size());
EVP_EncryptFinal_ex(ctx, ciphertext.data() + len, &len);
```

Decryption before:

```
EVP_DecryptInit_ex(ctx, EVP_aes_256_cbc(), nullptr, key, iv);
EVP_DecryptUpdate(ctx, plaintext.data(), &len, reinterpret_cast<const unsigned
char*>(ciphertext.data()), ciphertext.size());
EVP_DecryptFinal_ex(ctx, plaintext.data() + len, &len);
```

New AES-256-GCM Encryption/Decryption - With Integrity Check:

Encryption after:

```
EVP_EncryptInit_ex(ctx, EVP_aes_256_gcm(), nullptr, key, iv);
EVP_EncryptUpdate(ctx, ciphertext.data(), &len, plaintext.data(), plaintext.size());
EVP_EncryptFinal_ex(ctx, ciphertext.data() + len, &len);
EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GCM_GET_TAG, 16, tag);
```

Decryption After:

```
EVP_DecryptInit_ex(ctx, EVP_aes_256_gcm(), nullptr, key, iv);
EVP_DecryptUpdate(ctx, plaintext.data(), &len, ciphertext_data, ciphertext_len);
EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GCM_SET_TAG, 16, tag);
```

```
EVP_DecryptFinal_ex(ctx, plaintext.data() + len, &len);
```

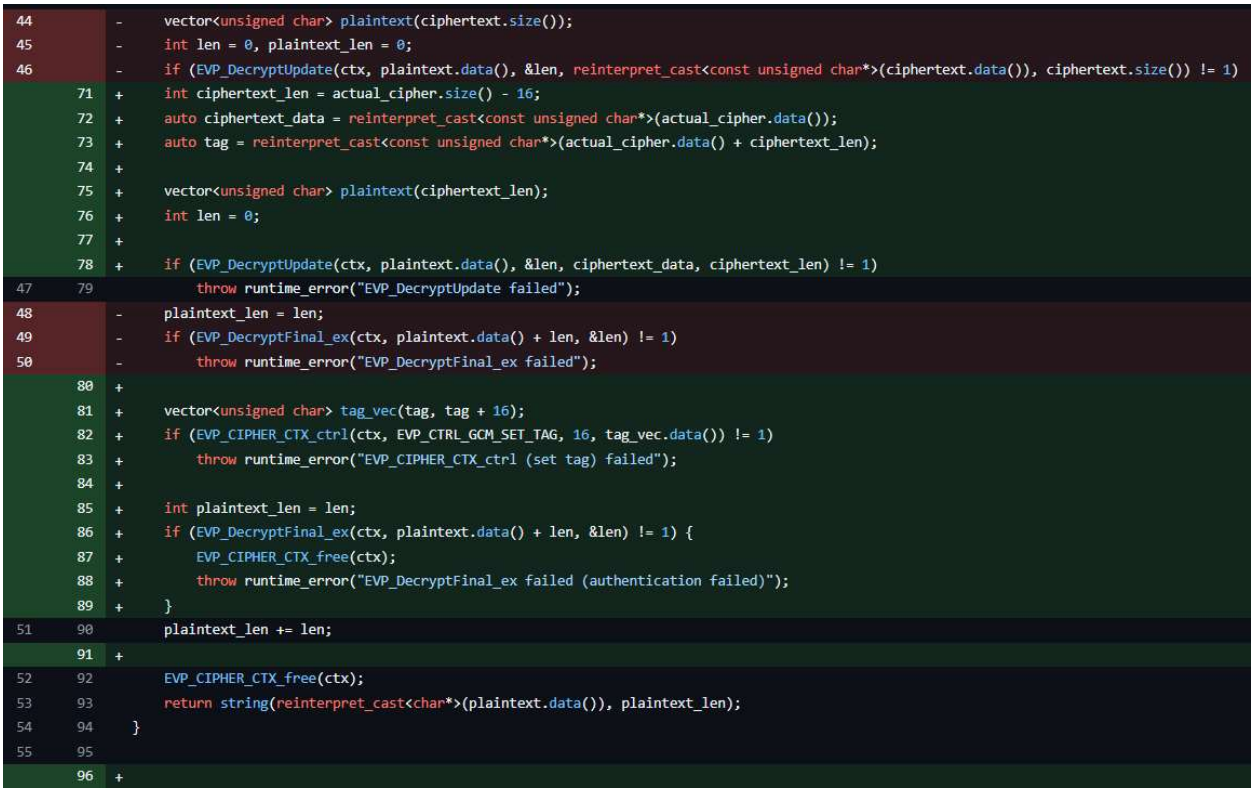
Full Code Changes:

```
+
 string aes_encrypt(const string &plaintext, const unsigned char *key, const unsigned char *iv) {
+    const string tag_prefix = "GCM";
     EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
     if (!ctx)
         throw runtime_error("Failed to create cipher context");
-    if (EVP_EncryptInit_ex(ctx, EVP_aes_256_cbc(), nullptr, key, iv) != 1)
+
+    if (EVP_EncryptInit_ex(ctx, EVP_aes_256_gcm(), nullptr, key, iv) != 1)
         throw runtime_error("EVP_EncryptInit_ex failed");

-    vector<unsigned char> ciphertext(plaintext.size() + EVP_CIPHER_block_size(EVP_aes_256_cbc()));
-    int len = 0, ciphertext_len = 0;
-    if (EVP_EncryptUpdate(ctx, ciphertext.data(), &len, reinterpret_cast<const unsigned char*>(plaintext.data()), plaintext.size()) != 1)
+    vector<unsigned char> ciphertext(plaintext.size());
+    int len = 0;
+
+    if (EVP_EncryptUpdate(ctx, ciphertext.data(), &len,
+                          reinterpret_cast<const unsigned char*>(plaintext.data()), plaintext.size()) != 1)
         throw runtime_error("EVP_EncryptUpdate failed");
-    ciphertext_len = len;
+
+    int ciphertext_len = len;
+
     if (EVP_EncryptFinal_ex(ctx, ciphertext.data() + len, &len) != 1)
         throw runtime_error("EVP_EncryptFinal_ex failed");
     ciphertext_len += len;
```

```
+
+    unsigned char tag[16];
+    if (EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GCM_GET_TAG, 16, tag) != 1)
+        throw runtime_error("EVP_CIPHER_CTX_ctrl (get tag) failed");
+
     EVP_CIPHER_CTX_free(ctx);
-    return string(reinterpret_cast<char*>(ciphertext.data()), ciphertext_len);
+
+    string final_result = tag_prefix;
+    final_result += string(reinterpret_cast<char*>(ciphertext.data()), ciphertext_len);
+    final_result += string(reinterpret_cast<char*>(tag), 16);
+    return final_result;
 }

+
+
 string aes_decrypt(const string &ciphertext, const unsigned char *key, const unsigned char *iv) {
+    if (ciphertext.size() < 3)
+        throw runtime_error("Invalid ciphertext length");
+
+    string mode_tag = ciphertext.substr(0, 3);
+    string actual_cipher = ciphertext.substr(3);
+
+    if (actual_cipher.size() < 16)
+        throw runtime_error("Ciphertext too short for GCM");
+
     EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
     if (!ctx)
         throw runtime_error("Failed to create cipher context");
-    if (EVP_DecryptInit_ex(ctx, EVP_aes_256_cbc(), nullptr, key, iv) != 1)
+
+    if (EVP_DecryptInit_ex(ctx, EVP_aes_256_gcm(), nullptr, key, iv) != 1)
         throw runtime_error("EVP_DecryptInit_ex failed");
```



```
-     vector<unsigned char> plaintext(ciphertext.size());
-     int len = 0, plaintext_len = 0;
-     if (EVP_DecryptUpdate(ctx, plaintext.data(), &len, reinterpret_cast<const unsigned char*>(ciphertext.data()), ciphertext.size()) != 1)
+     int ciphertext_len = actual_cipher.size() - 16;
+     auto ciphertext_data = reinterpret_cast<const unsigned char*>(actual_cipher.data());
+     auto tag = reinterpret_cast<const unsigned char*>(actual_cipher.data() + ciphertext_len);
+
+     vector<unsigned char> plaintext(ciphertext_len);
+     int len = 0;
+
+     if (EVP_DecryptUpdate(ctx, plaintext.data(), &len, ciphertext_data, ciphertext_len) != 1)
          throw runtime_error("EVP_DecryptUpdate failed");
-     plaintext_len = len;
-     if (EVP_DecryptFinal_ex(ctx, plaintext.data() + len, &len) != 1)
-         throw runtime_error("EVP_DecryptFinal_ex failed");
+
+     vector<unsigned char> tag_vec(tag, tag + 16);
+     if (EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GCM_SET_TAG, 16, tag_vec.data()) != 1)
+         throw runtime_error("EVP_CIPHER_CTX_ctrl (set tag) failed");
+
+     int plaintext_len = len;
+     if (EVP_DecryptFinal_ex(ctx, plaintext.data() + len, &len) != 1) {
+         EVP_CIPHER_CTX_free(ctx);
+         throw runtime_error("EVP_DecryptFinal_ex failed (authentication failed)");
+     }
      plaintext_len += len;
+
      EVP_CIPHER_CTX_free(ctx);
      return string(reinterpret_cast<char*>(plaintext.data()), plaintext_len);
  }
+
```

ii) Partial Information Leak – File Names Are Exposed

Description:
Group 1 reported that the file and folder names in the file system (e.g., under `filesystem/<user>/personal`) were stored in plaintext, leaking sensitive contextual metadata such as file names and user activity patterns.

Issue:

- Exposure of document names like `secret_plan.pdf` or `private_key.pem`.
- Inference of user activity from timestamps or naming conventions.
- Potential Information leak by inspecting directory structures.

Not Fixed – imbalanced Implementation Trade-Offs:

While this is a legitimate, though non-critical, concern, we opted **not to encrypt filenames** due to:

- Significant **refactoring complexity** across many modules (file I/O, directory traversal, user commands).

- Trade-offs between **usability and security** — encrypting filenames would require a separate mapping system, impacting ease of access and maintainability.
- File name encryption is a **lower severity issue** compared to plaintext content leakage.

We explicitly note this limitation in our system documentation and acknowledge the trade-off.

3. Future Improvements

(i) Address Vulnerability – Partial Information Leak

Theoretical Architectural Solution for File/Directory Name Obfuscation:

If we were to address the issue of file and directory names being human readable, we would adopt a design that integrates deterministic name encryption and user-specific secure bootstrapping. The goal would be to fully obscure filesystem metadata from local

adversaries by encrypting paths based on a global key and securely binding them to user credentials.

Our proposed approach would include the following:

- **Deterministic Path Obfuscation**: All file and directory names would be encrypted using `encryptPath()` and accessed via the virtual filesystem layer (`virtual_fs`) to produce consistent, hashed paths derived from a global sharing key.

- **Global Key Retrieval First**: Before any user-specific directories are created, the global key would be initialized and retrieved. This key would be required for all future filesystem interactions.

- **Startpoint Bootstrapping**: Each user would have a visible directory under `filesystem/startpoint/`, where the subdirectory name is derived from the user's password (via `deriveKeyFromPassword()`). Within this, an `init.enc` file would store encrypted references to the user's private key path and the obfuscated global key path.

- **Secure Initialization**: On first login or key generation, the system would encrypt the true (obfuscated) paths using the password-derived key and store them in `init.enc`. These paths could then be decrypted on future logins to retrieve and activate the global key and private key.

- **Virtual Filesystem Enforcement**: All other files and directories (keyfiles, metadata, user files) would be created **only after** the global key has been retrieved, and always via `virtual_fs` functions to ensure consistency and prevent leakage.

Due to the **complexity, time constraints, and the risk of introducing new issues**, we chose **not** to implement this enhancement in the current system. However, we acknowledge its value and consider it a viable direction for future improvements to protect against metadata leakage.

4. Conclusion

In this Fix-it Report, we addressed the most critical vulnerability identified by Group 1—namely, the lack of integrity checks in our encryption logic. We replaced AES-256-CBC with AES-256-GCM to provide authenticated encryption, significantly improving the system's resistance to ciphertext tampering and ensuring message authenticity and confidentiality.

For the second issue—human readable file names and directory names—we conducted a thorough assessment and developed a robust theoretical design to address it. While the proposed solution would greatly enhance metadata confidentiality, its implementation would require extensive architectural changes across the system, including reworking all file I/O, user commands, and metadata handling to pass through a deterministic virtual file system.

Given the complexity of such a change, the limited time available, and the non-critical nature of the vulnerability, we chose to defer its implementation. However, our exploration of a secure bootstrapping mechanism and virtual name obfuscation illustrates a clear path forward for future development.

Overall, we prioritized impactful security improvements while carefully considering the cost-benefit trade-offs of more extensive changes. This report reflects our commitment to improving the security posture of the system, both through concrete fixes and forward-looking design.