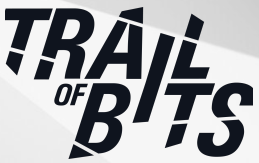

Discord DAVE Protocol Code Review

Security Assessment

September 5, 2024

Prepared for:
Maxime Nay
Discord

Prepared by: **Fredrik Dahlgren, Joe Doyle, and Artur Cygan**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
497 Carroll St., Space 71, Seventh Floor
Brooklyn, NY 11215
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

About Trail of Bits **1**
Notices and Remarks **2**
Table of Contents **3**
Project Summary **4**
Executive Summary **5**
Project Targets **8**
Project Goals **9**
Project Coverage **10**
Automated Testing **12**
Codebase Maturity Evaluation **14**
Summary of Findings **17**
Detailed Findings **19**
1. Undefined behavior in frame processor 19
2. Insufficient validation of unencrypted ranges 21
3. Insufficient Windows key data size validation 23
4. Call participants can send different media frames to lagging participants 25
5. Encrypted frames can be delivered multiple times or out-of-order 27
6. Unencrypted range offsets and sizes are not authenticated 29
7. Insufficient size validation in SerializeUnencryptedRanges 31
8. Integer overflow during encrypted frame validation 32
9. Out-of-bounds read in FindNextH26XNaluIndex 34
10. Insufficient validation of proposal type 36
11. Application UI does not distinguish causes of verification failure 38
12. Public key uploads are not bound to a specific account 39
13. Sensitive data is not cleared from memory 40
14. Session not closed on bad MLS binary input 43
15. Static key ratchet used in production code 44
A. Vulnerability Categories **46**
B. Code Maturity Categories **48**
C. Automated Testing **50**
D. Code Quality Recommendations **52**
E. Fix Review Results **54**
Detailed Fix Review Results 56

Project Summary

Contact Information

The following project manager was associated with this project:

Mike Shelton, Project Manager
mike.shelton@trailofbits.com

The following engineering director was associated with this project:

Jim Miller, Engineering Director, Cryptography
james.miller@trailofbits.com

The following consultants were associated with this project:

Artur Cygan, Consultant
artur.cygan@trailofbits.com

Fredrik Dahlgren, Consultant
fredrik.dahlgren@trailofbits.com

Joe Doyle, Consultant
joe.doyle@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
August 1, 2024	Pre-project kickoff call
August 12, 2024	Status update meeting #1
August 19, 2024	Delivery of report draft
August 19, 2024	Report readout meeting
September 16, 2024	Delivery of comprehensive report

Executive Summary

Engagement Overview

Discord engaged Trail of Bits to review the security of the implementation of its E2EE RTC protocol DAVE. The protocol is based on MLS (for group key agreement) and WebRTC (for audio and video delivery).

A team of three consultants conducted the review from August 12, 2024 to August 16, 2024, for a total of five engineer-weeks of effort. Our testing efforts focused on the client-side implementation and integration of the DAVE protocol, which handles media frame encryption and decryption, and Discord Voice, which implements the MLS delivery service. With full access to source code and design documentation, we performed static and dynamic testing of the target libraries, using automated and manual processes.

Additionally, we performed a fix review to assess the implemented fixes for the issues described in this report. All 15 issues identified during the engagement were completely addressed by the Discord team. The results from the fix review are described in appendix E.

Observations and Impact

The Discord E2EE RTC client is implemented by the DAVE C++ library, `libdave`. The library uses the `mlspp` library for group key agreement and the WebRTC library for video and voice communication. It is well-structured and defensively written, and we did not identify any issues with how DAVE's implementation interfaces with the underlying `mlspp` library.

The most significant issues identified during the review are related to input validation and memory unsafety. Since the library is written in C++, these types of issues could constitute a significant security risk to the stability and integrity of the client. This risk is exacerbated by the fact that the current test suite mostly tests the happy path and fails to sufficiently exercise the implementation on invalid inputs.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Discord take the following steps prior to deploying the DAVE protocol:

- **Remediate the findings disclosed in this report.** Many of these findings constitute a risk to either the security or stability of the native Discord clients. For this reason, we recommend addressing them as part of a direct remediation before the system is deployed into a production environment.

- **Improve randomized testing of the DAVE C++ library.** Many of the issues identified during this review could have been uncovered using either fuzzing or property testing. This is a clear indication that the current test suite mainly tests the happy path and fails to exercise the code on unexpected or invalid inputs. We recommend that Discord implement property testing across the entire `libdave` library using a framework like RapidCheck, focusing on components that handle untrusted inputs.

  The `libdave` library already contains a fuzzer based on AFL++ that tests the frame parser and decryptor, but we still found issues related to input validation in this part of the codebase that should have been identified by the fuzzer (see TOB-DISCE2EC-1 and TOB-DISCE2EC-2). This indicates that it would also be worthwhile to investigate and improve the fuzzer's code coverage. In particular, we note that it is difficult to fuzz code if reaching it requires a valid MAC or signature (which is true for parts of the frame processing logic); in this case, it may be necessary to use a dedicated fuzzer.

- **Encourage and adopt WebRTC support for encrypted media streams.** The current implementation works around several WebRTC media codecs that can mishandle ciphertext by treating it as encoded media. We recommend that Discord participate in industry standardization efforts to directly support encrypted media streams, and that they adopt these new standards once available.

- **Perform a comprehensive security review of the `mlspp` library.** Because of timing and scoping limitations, we did not perform an end-to-end review of the underlying MLS library `mlspp`. Even though the library was not in scope, we still identified a number of security issues in `mlspp` during the design review and code review phases of the engagement (e.g., TOB-DISCE2EC-14). We recommend that Discord performs a comprehensive security review of the library, focusing on security and correctness.

- **Work toward moving away from memory-unsafe languages.** This review uncovered a number of memory safety issues, such as out-of-bounds memory accesses and integer overflows (see TOB-DISCE2EC-1, TOB-DISCE2EC-2, TOB-DISCE2EC-4, TOB-DISCE2EC-8, TOB-DISCE2EC-9, and TOB-DISCE2EC-10). These types of issues are prevalent in memory unsafe languages like C and C++, and the only way to avoid them completely is to transition to more modern, memory-safe languages, like Rust, Swift, and Kotlin. Although we understand that the choice of language for the DAVE protocol implementation was at least partially influenced by external requirements, we still recommend that Discord work towards moving away from using C and C++ for security-critical projects.

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	1
Medium	4
Low	6
Informational	3
Undetermined	1

CATEGORY BREAKDOWN

Category	*Count*
Authentication	1
Cryptography	4
Data Validation	7
Denial of Service	1
Error Reporting	1
Undefined Behavior	1

Project Targets

The engagement involved a review and testing of the following target.

Discord monorepo

Repository	https://github.com/discord/discord
Version	`67a37e2dc8a5774c99c996a4ad49e39b072266d6`
Type	C++, JavaScript, and Elixir
Platform	Windows, MacOS, Android, and iOS

Project Goals

The engagement was scoped to provide a security assessment of the Discord E2EE RTC implementation. Specifically, we sought to answer the following non-exhaustive list of questions:

- Does `libdave` adhere to the design document provided for the DAVE protocol?
- Are MLS messages and encrypted frames validated correctly?
- Are return values from security sensitive functions checked correctly?
- Does the client correctly compute and display security codes?
- Are public keys correctly matched to user accounts?
- Is sensitive data like key material scrubbed from memory when it is no longer used?
- Can the Discord server break the confidentiality of the protocol?
- Can a malicious actor disrupt the Discord server in any way?

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- **DAVE C++ library.** We manually reviewed the `libdave` library to assess whether the implementation follows the provided design document. We reviewed whether all cryptographic operations are performed securely and whether the library interfaces with the underlying MLS implementation correctly. During the manual review, we also looked for potential input-validation and memory-safety issues. We ran static-analysis tools like Semgrep, Flawfinder, and Cppcheck against the codebase to identify known-vulnerable code patterns, and we modified and ran the AFL++-based fuzzer implemented by the Discord team.
- **DAVE JavaScript library.** We manually reviewed the key fingerprint generation helper functions to assess whether key fingerprints are correctly generated from keys.
- **Discord App library.** We manually reviewed the fingerprint and chat verification workflow to assess whether fingerprints and keys are correctly calculated and whether fingerprint verification data is correctly stored.
- **Discord API.** We manually reviewed the public key upload and matching services to determine if public keys are correctly matched to user accounts.
- **Discord native library.** We manually reviewed the Discord native library, focusing on how the Discord client interfaces with the `libdave` library.
- **Discord Voice server.** We manually reviewed the parts of the `discord_voice` server that handle the MLS protocol.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- **Selective media frame encryption.** We did not have time to investigate each codec format to determine to which extent the selective media frame encryption leaks information about the underlying plaintext.
- **Underlying MLS library.** We did not have time to perform an end-to-end review of the underlying MLS library `mlspp`. Instead, we assumed that the library was

implemented correctly and focused our review on how the library was integrated by the `libdave` library.

- **Correct display of UI elements.** We did not confirm that in all cases, the fingerprint data displayed is the correct data for that conversation. When a user verifies a conversation fingerprint, they only have access to the data shown on their screen. While we did verify that the UI library generates HTML including the fingerprint, we did not confirm that related display data—such as the surrounding HTML and the involved stylesheets—ensures that the correct code is shown.
- **Delivery service.** While we did perform a best-effort review of the `discord_voice` server, some of its more complex parts, such as the `MLSManager`, would benefit from deeper review.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description	Policy
CodeQL	A code analysis engine developed by GitHub to automate security checks	Appendix C.1
Cppcheck	An open-source tool that detects undefined behavior and dangerous code patterns in C and C++ code	Appendix C.2
Flawfinder	An open-source tool that scans C and C++ source code and reports potential security issues	Appendix C.3
Semgrep	An open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code	Appendix C.4

Areas of Focus

Our automated testing and verification work focused on investigating the following question:

- Does the codebase contain known vulnerable code patterns or undefined behavior?

Test Results

The results of this focused testing are detailed below.

`libdave` C++ library. We ran Cppcheck, Flawfinder, and Semgrep on the `libdave` C++ library to search for known-vulnerable code patterns. Because of time constraints, we were not able to build the code as a standalone library, which means that we could not analyze the library using CodeQL.

Property	Tool	Result
The implementation does not contain known-vulnerable code patterns or undefined behavior	Cppcheck Flawfinder Semgrep	**Passed**

DAVE JavaScript Library. We used Semgrep and CodeQL on the JavaScript library to search for dangerous code constructs and known vulnerabilities.

Property	Tool	Result
The implementation does not contain known-vulnerable code patterns	CodeQL Semgrep	**Passed**

Discord Voice. We used Semgrep to analyze the server implementation located in the `discord_voice` directory. Apart from a private RSA key located in `discord_voice/priv`, this did not identify any issues.

Property	Tool	Script
The implementation does not contain known-vulnerable code patterns	Semgrep	**Passed**

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The `libdave` library does not use checked arithmetic when performing arithmetic operations on untrusted inputs, and we identified a number of issues (see TOB-DISCE2EC-2 and TOB-DISCE2EC-9) related to integer overflow.	**Moderate**
Auditing	The `libdave` library logs security relevant events using the `DISCORD_LOG` macro, which uses the underlying WebRTC logger.	**Satisfactory**
Authentication / Access Controls	Group membership is controlled by the Discord server, and since access to key material requires an `Add` proposal from the server, this is cryptographically guaranteed by the underlying MLS implementation. MLS ensures that all group members agree on the current state of the group (which includes group membership), and individual group members can verify each other's public keys out of band to prevent machine-in-the-middle attacks.	**Strong**
Complexity Management	The codebase is well-structured and easy to navigate. Functions are kept small and single-purpose, which helps reduce code complexity. The `libdave` library also uses interfaces to hide irrelevant low-level details in higher levels of the implementation. On the other hand, the requirement that encrypted frames must look like plaintext to the WebRTC packetizer and depacketizer, together with the requirement that the selective forwarding unit must be able to read parts of the encrypted frame, are both sources of complexity that	**Satisfactory**

	could be the cause of future security issues in the codebase.	
Configuration		**Not Considered**
Cryptography and Key Management	The DAVE protocol relies on MLS, a well-established group key agreement protocol that provides strong cryptographic guarantees. Symmetric keys are rotated and expired regularly by the implementation. The key rotation implementation allows for multiple simultaneously active keys, which introduces a risk of plaintext equivocation as described in TOB-DISCE2EC-5; however, because of the nature of the system, we believe that this risk is mostly theoretical. All encrypted messages are either implicitly authenticated using shared symmetric keys, or explicitly authenticated using signatures. However, we found one instance where plaintext message content is not included as authenticated data to the underlying AEAD (TOB-DISCE2EC-7). We also found a downgrade attack due to the inclusion of testing code in production builds where the Discord server could effectively disable encryption for all clients (TOB-DISCE2EC-16). Finally, we also found that neither the `libdave` library nor the underlying MLS library zeroizes sensitive data in memory when it is no longer used by the application (TOB-DISCE2EC-14).	**Satisfactory**
Data Handling	The `libdave` library defines dedicated types for a number of data types handled by the system, which helps prevent type confusion issues. Untrusted data is typically validated correctly as it enters the system, but there does not seem to be any structured thought behind how validation is performed. As an example, the underlying MLS library uses dedicated types (like `ValidatedContent` and `AuthenticatedContent`) to represent validated data, which provides compile-time guarantees that certain validation is performed before the data is released. This is not true of the DAVE protocol implementation.	**Moderate**

	We identified many input validation issues throughout the `libdave` library (TOB-DISCE2EC-2, TOB-DISCE2EC-4, TOB-DISCE2EC-6, TOB-DISCE2EC-8, TOB-DISCE2EC-9, TOB-DISCE2EC-10, and TOB-DISCE2EC-11). Since the library is written in a memory-unsafe language, the risk associated with this type of issue is particularly severe.	
Documentation	The design documentation provides a good resource for the overall design of the system and the different design choices related to MLS and the selective WebRTC frame encryption. However, the core APIs (including, e.g., the RPC API in `discord_api`) could benefit from more documentation.	**Satisfactory**
Maintenance		**Not Considered**
Memory Safety and Error Handling	The `libdave` library is written in a memory-unsafe language, and even though Discord has implemented a fuzzer for the library, we still identified multiple instances where missing or incomplete input validation could lead to memory-safety violations like out-of-bounds memory accesses. (See the Data Validation section above for references to individual issues.)	**Weak**
Testing and Verification	The `libdave` library has a number of unit tests that ensure that components like the BoringSSL encryptor, key management, and unencrypted range management work as expected. In addition, Discord has implemented a fuzzer that tests the encrypted frame parser and decryptor. However, most unit tests target expected behavior, and apart from the fuzzer, very few tests exercise the code on unexpected or invalid data. The issues we identified during the audit support this.	**Weak**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Undefined behavior in frame processor	Undefined Behavior	**Medium**
2	Insufficient validation of unencrypted ranges	Data Validation	**Medium**
3	Insufficient Windows key data size validation	Data Validation	**Low**
4	Call participants can send different media frames to lagging participants	Cryptography	**Low**
5	Encrypted frames can be delivered multiple times or out-of-order	Data Validation	**Informational**
6	Unencrypted range offsets and sizes are not authenticated	Cryptography	**Medium**
7	Insufficient size validation in SerializeUnencryptedRanges	Data Validation	**Informational**
8	Integer overflow during encrypted frame validation	Data Validation	**Low**
9	Out-of-bounds read in FindNextH26XNaluIndex	Data Validation	**Low**
10	Insufficient validation of proposal type	Data Validation	**Low**
11	Application UI does not distinguish causes of verification failure	Authentication	**Informational**
12	Public key uploads are not bound to a specific account	Denial of Service	**Low**
13	Sensitive data is not cleared from memory	Cryptography	**Medium**

14	Session not closed on bad MLS binary input	Error Reporting	**Undetermined**
15	Static key ratchet used in production code	Cryptography	**High**

Detailed Findings

1. Undefined behavior in frame processor

Severity: **Medium**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-DISCE2EC-1
Target: `discord_common/native/secure_frames/frame_processors.cpp`	

Description

The `InboundFrameProcessor::ParseFrame` method computes the offset of the magic marker as `frame.end() - sizeof(MagicMarker)` before checking that the frame buffer is large enough to contain the magic marker. The result of this expression is well-defined only if the result is inside (or one past the end) of the buffer that `frame.begin()` points to.

```
// Check the frame ends with the magic marker
auto magicMarkerBuffer = frame.end() - sizeof(MagicMarker);
if (frame.size() < sizeof(MagicMarker) ||
    memcmp(magicMarkerBuffer, &kMarkerBytes, sizeof(MagicMarker)) != 0) {
    Return;
}
```

Figure 1.1: The frame processor computes the offset of the magic marker before checking that the frame is large enough to contain the magic marker (discord_common/native/secure_frames/frame_processors.cpp#160-165)

If the frame buffer is smaller than `sizeof(MagicMarker)`, this is undefined behavior. (For additional details, see the C++ reference section on pointer arithmetic.)

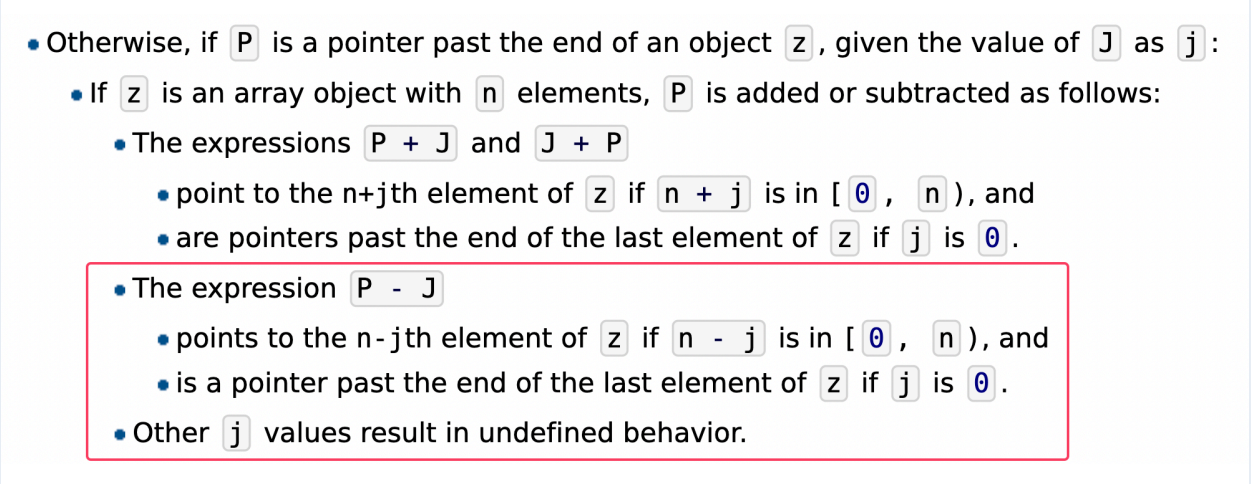

Figure 1.2: The expression P – J is well-defined only if the result points to an element of the array, or immediately past the end of the array.

Since the expression represents undefined behavior, the compiler is free to assume that this never happens and optimizes the surrounding code accordingly. This could potentially cause stability or security issues.

Exploit Scenario

A malicious Discord user discovers that short media frames cause some Discord clients to become unstable and crash. She uses this to kick users from video calls by sending short media frames to crash their clients.

Recommendations

Short term, check the media frame size before computing the offset of the magic marker.

Long term, fuzz the frame parser with undefined behavior sanitizer (UBSAN) enabled to identify any other instances of undefined behavior. To enable UBSAN when using AFL++, set the environment variable `AFL_USE_UBSAN` to 1 before building.

2. Insufficient validation of unencrypted ranges	
Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-DISCE2EC-2
Target: `discord_common/native/secure_frames/frame_processors.cpp`	

Description

The `ValidateUnencryptedRanges` function checks that the list of unencrypted ranges in the frame is ordered consecutively and that the ranges are contained within the frame. However, since the addition of the offset and size of a single range may overflow during validation, both properties could be invalidated by a maliciously crafted frame.

```
// validate that the ranges are in order and don't overlap
for (auto i = 1u; i < unencryptedRanges.size(); ++i) {
    auto prev = unencryptedRanges[i - 1];
    auto current = unencryptedRanges[i];


    if (prev.offset + prev.size > current.offset) {
        DISCORD_LOG(LS_WARNING)
          << "Unencrypted range may overlap or be out of order: prev offset: "
          << prev.offset << ", prev size: " << prev.size
          << ", current offset: " << current.offset
          << ", current size: " << current.size;
        return false;
    }
}

// validate that the last range doesn't exceed the frame size
auto last = unencryptedRanges.back();
if (last.offset + last.size > frameSize) {
    DISCORD_LOG(LS_WARNING)
      << "Unencrypted range exceeds frame size: offset: " << last.offset
      << ", size: " << last.size << ", frame size: " << frameSize;
    return false;
}
```

Figure 2.1: Computing the end of the range may overflow in `ValidateUnencryptedRanges` (`discord_common/native/secure_frames/frame_processors.cpp#76-96`)

For example, if the last range offset is equal to 0xFFFFFFFFFFFFFFFF and the corresponding size is 1, their sum will overflow and the result will be 0, which means that the final check above will pass even though the offset is outside the frame buffer.

By choosing the unencrypted ranges and offsets maliciously, an attacker could cause the frame processor to read out of bounds or allocate large amounts of memory in `AddCiphertextBytes` and `AddAuthenticatedBytes`, which use the unencrypted range offsets and sizes to copy data from the frame buffer.

```
// Split the frame into authenticated and ciphertext bytes
size_t frameIndex = 0;
for (const auto& range : unencryptedRanges_) {
    auto encryptedBytes = range.offset - frameIndex;
    if (encryptedBytes > 0) {
        assert(frameIndex + encryptedBytes <= frame.size());
        AddCiphertextBytes(frame.data() + frameIndex, encryptedBytes);
    }

    assert(range.offset + range.size <= frame.size());
    AddAuthenticatedBytes(frame.data() + range.offset, range.size);
    frameIndex = range.offset + range.size;
}
```

Figure 2.2: If the checks performed by the `InboundFrameProcessor::ParseFrame` method overflow as well, a maliciously crafted frame could cause large allocations inside `AddCiphertextBytes` or `AddAuthenticatedBytes`. (`discord_common/native/secure_frames/frame_processors.cpp#232-244`)

```
void InboundFrameProcessor::AddCiphertextBytes(const uint8_t* data, size_t size)
{
    ciphertext_.resize(ciphertext_.size() + size);
    memcpy(ciphertext_.data() + ciphertext_.size() - size, data, size);
}
```

Figure 2.3: The `AddCiphertextBytes` method allocates memory based on the unencrypted ranges. (`discord_common/native/secure_frames/frame_processors.cpp:279-283`)

Exploit Scenario

A malicious Discord user sends media frames with invalid unencrypted ranges. This causes other clients on the call to run out of memory and crash when they attempt to parse these frames.

Recommendations

Short term, use `__builtin_add_overflow` (which is available for both GCC and Clang) to detect overflows when validating the unencrypted ranges.

Long term, always use checked arithmetic when computing with untrusted inputs. Regularly review fuzz test coverage to ensure that security-relevant code is exercised by the fuzzer. Consider building the fuzzer with the `-fsanitize=unsigned-integer-overflow` flag to detect unsigned integer overflows.

3. Insufficient Windows key data size validation

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-DISCE2EC-4
Target: `discord_common/native/secure_frames/mls/detail/persisted_key_pair_win.cpp`	

Description

The `convertBlob` lambda expression defined in the `GetNativePersistedKeyPair` function for Windows checks the size of the input blob before casting it to a `BCRYPT_ECCKEY_BLOB` pointer. However, the size is checked against the size of a `BCRYPT_ECCKEY_BLOB` *pointer* (which is either four or eight bytes depending on the architecture) rather than the size of an actual `BCRYPT_ECCKEY_BLOB` structure (which is eight bytes).

```
if (blob.size() < sizeof(PBCRYPT_ECCKEY_BLOB)) {
    DISCORD_LOG(LS_ERROR)
      << "Exported key blob too small in GetPersistedKeyPair/convertBlob: "
      << blob.size();
    return false;
}

PBCRYPT_ECCKEY_BLOB header = (PBCRYPT_ECCKEY_BLOB)blob.data();
```

Figure 3.1: The size check mistakenly uses PBCRYPT_ECCKEY_BLOB instead of BCRYPT_ECCKEY_BLOB. (discord_common/native/secure_frames/mls/detail/persisted_key_pair_win.cpp#69-76)

Since the correct size check is performed before `convertBlob` is called, this is currently not a problem, but could become an issue in the future if the calling function is refactored.

Exploit Scenario

The `GetNativePersistedKeyPair` function for Windows is refactored and the duplicate size check outside `convertBlob` is removed. Since the check in `convertBlob` checks the blob size against the pointer size rather than the structure size, the function could end up reading out of bounds on 32-bit architectures.

Recommendations

Short term, unify the key data validation in the `GetNativePersistedKeyPair` function and ensure that the size of the blob is checked against the size of the actual structure.

Long term, use a tool like CodeQL to identify locations where the size of a buffer is checked against the size of a pointer rather than a structure.

4. Call participants can send different media frames to lagging participants

Severity: **Low**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-DISCE2EC-5
Target: `discord_common/native/secure_frames/`	

Description

When responding to issues of abuse in a group call, it is important for participants to be able to correctly identify who is acting abusively. If an abusive participant is able to make their actions appear differently to different parties, they can "muddy the waters." They could evade consequences, or even directly abuse a target, by attacking the credibility of their accusers.

The Discord E2EE call protocol does not guarantee that each participant receives the same video feed from any given participant, and furthermore does not have a mechanism to allow participants to detect when different clients have received conflicting data from one participant.

Since audio and video traffic is forwarded through the selective forwarding unit, it should not be possible to send a different ciphertext to each participant, but it is possible to send different video to participants who are lagging behind due to the flexibility of the decryption logic (shown below in figure 4.1).

```
// Try and decrypt with each valid cryptor
// reverse iterate to try the newest cryptors first
bool success = false;
for (auto it = cryptorManagers_.rbegin(); it != cryptorManagers_.rend(); ++it) {
    auto& cryptorManager = *it;
    success = DecryptImpl(cryptorManager, mediaType, *localFrame, frame);
    if (success) {
        break;
    }
}
```

Figure 4.1: Active entries in the `cryptorManagers` vector are tried in newest-to-oldest order (`discord/discord_common/native/secure_frames/decryptor.cpp#93–102`)

This is possible since AES-GCM is not key committing, which means that it is possible to create AES-GCM ciphertexts that decrypt to different valid plaintexts under different keys.

In one attack, the victim does not yet have the most recent `cryptorManager`. The attacker creates a malicious encrypted frame that successfully decrypts to two different messages: a

benign frame for the newest key, and an abusive frame for the previous key. Participants who have the most recent encryption key will decrypt the benign frame, while those who have not yet adopted it will decrypt the abusive frame.

In another attack, the victim has not yet removed a recently expired `cryptorManager`, but other participants have. The attacker sends the malicious frames using the oldest key, and only the victim decrypts it successfully.

If Discord E2EE calls eventually allow direct peer-to-peer connections, or if an attacker can manipulate the selective forwarding unit to choose which packets are sent to which recipient, this attack becomes much more powerful. In that case, an attacker would be able to send entirely different ciphertexts to each participant.

Exploit Scenario

Alice and Bob participate in competitive "Go Fish" tournaments with their friends over Discord calls. Bob wishes to get Alice disqualified, so he attempts to make it seem like Alice is cheating. Bob modifies his Discord client so that different audio is sent to Alice than is sent to anybody else. Bob asks Alice for twos, while sending a request for sevens to the remainder of the call. Alice responds "Go Fish," and after a few more turns, Alice gives sevens to another player, after which Bob accuses Alice of lying about not having sevens. Since everyone except Alice heard that Bob asked for sevens, Alice is disqualified for cheating despite Bob being the culprit.

Recommendations

Short term, document this attack and ensure that the Trust and Safety team is aware that sophisticated users may be able to send different data to each participant without those participants' knowledge.

Long term, design mitigations to detect and flag potential inconsistencies in data that each client receives from each participant.

References

- How to Abuse and Fix Authenticated Encryption Without Key Commitment

5. Encrypted frames can be delivered multiple times or out-of-order

Severity: **Informational**	Difficulty: **Not Applicable**
Type: Data Validation	Finding ID: TOB-DISCE2EC-6
Target: `discord/discord_common/native/secure_frames/decryptor.cpp`	

Description

When decrypting a received frame, the client does not perform any specific validation to ensure that the received frames are distinct or delivered in order. Figure 5.1 shows how the client determines the nonce and encryption key based on data in the packet. Any packet whose corresponding key and nonce have not expired will be accepted, including packets that have already been received, or packets that were created before the most recently received frame. This allows a Discord insider to manipulate the video and audio data received by a particular client by rearranging it.

```
bool Decryptor::DecryptImpl(CryptorManager& cryptorManager,
                            MediaType mediaType,
                            InboundFrameProcessor& encryptedFrame,
                            ArrayView<uint8_t> frame)
{
    auto tag = encryptedFrame.GetTag();
    auto truncatedNonce = encryptedFrame.GetTruncatedNonce();
    // ... [redacted]

    // expand the truncated nonce to the full sized one needed for decryption
    auto nonceBuffer = std::array<uint8_t, kAesGcm128NonceBytes>();
    memcpy(nonceBuffer.data() + kAesGcm128TruncatedSyncNonceOffset,
           &truncatedNonce,
           kAesGcm128TruncatedSyncNonceBytes);

    auto nonceBufferView = MakeArrayView<const uint8_t>(
           nonceBuffer.data(),
           nonceBuffer.size());

    auto generation = cryptorManager.ComputeWrappedGeneration(
        truncatedNonce >> kRatchetGenerationShiftBits);

    // Get the cryptor for this generation
    ICryptor* cryptor = cryptorManager.GetCryptor(generation);
    // ... [redacted]

    // perform the decryption
    bool success = cryptor->Decrypt(
           plaintext,
```

```
            ciphertext,
            tag,
            nonceBufferView,
            authenticatedData);
    stats_[mediaType].decryptAttempts++;

    if (success) {
        cryptorManager.ReportCryptorSuccess(generation);
    }

    return success;
}
```

Figure 5.1: The nonce and key are derived only from data in the packet (discord/discord_common/native/secure_frames/decryptor.cpp#139–179)

It is hard to effectively manipulate video and audio by rearranging it without knowing the underlying data, so the exploit scenario below does not include the creation of targeted misleading content. However, with the assistance of a call participant, an adversary could create "cheapfakes" by rearranging and selectively removing the audio and video some clients receive.

Since the underlying video codecs have their own sequence numbering, it does not appear to be possible to make video cheapfakes. According to the Discord team, the current audio codecs may not always have sequence numbering, which may still allow the creation of audio-only cheapfakes.

Recommendations

Short term, add replay protection to the DAVE protocol. This can be done by, for example, adding a sequence number to packets, tracking the most recent sequence number received from each remote user, and rejecting packets with sequence numbers that are older than the most recent frame. The existing truncated nonce can be used as a sequence number, in lieu of a sequence number, to reduce packet size.

Long term, extend the design goals of integrity and authenticity to clearly specify what users should expect from both the within-frame and across-frame behavior of an E2EE call.

6. Unencrypted range offsets and sizes are not authenticated

Severity: **Medium**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-DISCE2EC-7
Target: `discord_common/native/secure_frames/encryptor.cpp`	

Description

The unencrypted range offsets and sizes are not included in the associated data passed to AES-GCM, which means that they can be tampered with by an attacker with privileged network access who is able to modify encrypted frames.

```
const auto& unencryptedRanges = frameProcessor->GetUnencryptedRanges();
auto unencryptedRangesSize = UnencryptedRangesSize(unencryptedRanges);

auto additionalData = MakeArrayView(unencryptedBytes.data(),
unencryptedBytes.size());
auto plaintextBuffer = MakeArrayView(encryptedBytes.data(), encryptedBytes.size());
auto ciphertextBuffer = MakeArrayView(ciphertextBytes.data(),
ciphertextBytes.size());
// ... [redacted]

// encrypt the plaintext, adding the unencrypted header to the tag
bool success = cryptor->Encrypt(
  ciphertextBuffer, plaintextBuffer, nonceBufferView, additionalData, tagBuffer);
// ... [redacted]
```

Figure 6.1: The unencrypted range offsets and sizes are not included in the associated data passed to the underlying AEAD.
(discord_common/native/secure_frames/encryptor.cpp#79-120)

We note that encrypted frames are sent using a dedicated encrypted and authenticated channel between each client and Discord server. It follows that the ability to carry out this attack requires a high level of access to network traffic. For this reason, we have rated the difficulty of exploiting this issue as high.

Discord is aware of this issue and has already considered switching to an unauthenticated cipher mode like AES-CTR, and using truncated HMACs for authentication. This would allow them to decouple encryption and authentication, and authenticate the entire encrypted frame before it is parsed. However, Discord found that this solution did not provide sufficient performance for their use case.

Exploit Scenario

A privileged attacker with the ability to modify encrypted frames changes the unencrypted range offsets and sizes in an encrypted frame. Since unencrypted ranges are not validated correctly (see finding TOB-DISCE2EC-2), this allows the attacker to cause receiving clients to crash when they attempt to parse the frame.

Recommendations

Short term, document this issue together with any potential solutions considered in the protocol design documentation. This ensures that the risk remains known and that the problem can be revisited and potentially solved in the future.

Long term, continue to investigate ways to decouple authentication from encryption in a way that provides sufficient performance for the DAVE protocol. Work toward a solution that avoids the need for codec-aware encryption, which also removes the need for the unencrypted ranges.

7. Insufficient size validation in SerializeUnencryptedRanges

Severity: **Informational**	Difficulty: **Not Applicable**
Type: Data Validation	Finding ID: TOB-DISCE2EC-8
Target: `discord_common/native/secure_frames/frame_processors.cpp`	

Description

When the unencrypted range offsets and sizes are written to the output message buffer, the `SerializeUnencryptedRanges` function checks that the current range will fit within the buffer by asserting that the serialized size of each range is less than or equal to the buffer size. However, since there is typically more than one unencrypted range, this check should be performed against the remaining buffer size rather than the total buffer size.

```
uint8_t SerializeUnencryptedRanges(const Ranges& unencryptedRanges,
                                   uint8_t* buffer,
                                   size_t bufferSize)
{
    auto writeAt = buffer;
    for (const auto& range : unencryptedRanges) {
        assert(Leb128Size(range.offset) + Leb128Size(range.size) <= bufferSize &&
               "Buffer is too small to serialize unencrypted ranges");
        writeAt += WriteLeb128(range.offset, writeAt);
        writeAt += WriteLeb128(range.size, writeAt);
    }
    return writeAt - buffer;
}
```

Figure 7.1: The `SerializeUnencryptedRanges` function checks the serialized size of each range against the total buffer size.
(`discord_common/native/secure_frames/frame_processors.cpp#27-39`)

This means that the function can still write out of bounds if multiple unencrypted ranges are serialized to the given buffer. However, since the buffer used to hold the unencrypted ranges is ensured to be large enough to hold all the serialized ranges by the calling function (`Encryptor::Encrypt`), this issue is never triggered.

Recommendations

Short term, track the remaining buffer size and check the required size against the remaining size in each iteration of the for-loop in `SerializeUnencryptedRanges`.

Long term, use a property testing framework like RapidCheck to test individual functions on randomized inputs. Document the invariants assumed by each internal function.

8. Integer overflow during encrypted frame validation

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-DISCE2EC-9
Target: `discord_common/native/secure_frames/codec_utils.cpp`	

Description

Before an encrypted frame is released, the client checks that the frame does not contain any H.264 NAL unit start sequences that could confuse the H.264 packetizer. For each encrypted range, the `ValidateEncryptedFrame` function checks that the range does not contain any start sequences. To ensure that no sequence overlaps with the previous unencrypted range, the function starts the search two bytes before the start of the encrypted range.

```
size_t encryptedSectionStart = 0;
for (auto& range : unencryptedRanges) {
    if (encryptedSectionStart == range.offset) {
        encryptedSectionStart += range.size;
        continue;
    }

    auto padding = kH26XNaluShortStartSequenceSize - 1;
    auto start = std::max(size_t{0}, encryptedSectionStart - padding);
    auto end = std::min(range.offset + padding, frame.size());
    if (FindNextH26XNaluIndex(frame.data() + start, end - start)) {
        return false;
    }

    encryptedSectionStart = range.offset + range.size;
}
```

Figure 8.1: If the encrypted section start is less than 2, the computation of the `start` value will overflow. (`discord_common/native/secure_frames/codec_utils.cpp#397-412`)

If the first encrypted range starts at offset 0, the search should start at offset 0. However, because of the C++ implicit conversion rules, `padding` (which has `int` type) is converted to `size_t` before the subtraction when the `start` offset is computed. Since `padding` is 2 and `encryptedSectionStart` is 0 in the first iteration of the loop, this results in an unsigned integer overflow and sets `start` to the maximum value of `size_t` minus 1 (typically 2^{64} - 2).

```cpp
#include <cstdlib>
#include <iostream>

constexpr uint8_t kH26XNaluShortStartSequenceSize = 3;

int main() {
  size_t encryptedSectionStart = 0;

  auto padding = kH26XNaluShortStartSequenceSize - 1;
  auto start = std::max(size_t{0}, encryptedSectionStart - padding);

  std::cout << start << std::endl;
  return 0;
}
// clang++ -std=c++17 test.cpp -o test && ./test
// 18446744073709551614
```

Figure 8.2: Extracting the relevant code snippet and running it with an start offset equal to 0 demonstrates the overflow.

This will cause both of the parameters to the `FindNextH26XNaluIndex` function to overflow as well, which means that the function will read out of bounds. It follows that the validation performed in `ValidateEncryptedFrame` is ineffective in this special case, and could lead to segmentation faults if the function attempts to access unmapped memory.

Exploit Scenario

The Discord client is updated to support a new video format where the initial bytes of the frame header are encrypted. When the client validates encrypted frames in this format, the client reads out of bounds, causing crashes that are hard to diagnose for the development team.

Recommendations

Short term, use an explicit check to ensure that `encryptedSectionStart` is greater than `padding` when computing the `start` offset.

Long term, use a static-analysis tool like CodeQL to detect calls to unsigned versions of `std::max` where one argument is zero. Alternatively, use a property testing framework like RapidCheck to test individual functions on randomized inputs. Document the invariants assumed by each internal function.

9. Out-of-bounds read in FindNextH26XNaluIndex

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-DISCE2EC-10
Target: `discord_common/native/secure_frames/codec_utils.cpp`	

Description

The `FindNextH26XNaluIndex` function searches through a buffer for the next H.264 NAL unit start sequence and returns the offset of the first start sequence found. To avoid reading out of bounds, the main loop considers only offsets that are less than `bufferSize - kH26XNaluShortStartSequenceSize`.

```
// look for NAL unit 3 or 4 byte start code
for (size_t i = searchStartIndex; i < bufferSize - kH26XNaluShortStartSequenceSize;)
{
    if (buffer[i + 2] > kH26XStartCodeHighestPossibleValue) {
        // third byte is not 0 or 1, can't be a start code

        // ... [redacted]
    }
    else if (buffer[i + 2] == kH26XStartCodeEndByteValue) {
        // third byte matches the start code end byte, might be a start code

        // ... [redacted]
    }
    else {
        // third byte is 0, might be a four byte start code

        // ... [redacted]
    }
}
```

Figure 9.1: The computation of the upper bound in the main loop in `FindNextH26XNaluIndex` could overflow (`discord_common/native/secure_frames/codec_utils.cpp#81-109`)

However, if the buffer size is smaller than `kH26XNaluShortStartSequenceSize`, the subtraction computing the upper bound will overflow, leading to an out-of-bounds read.

We believe that this should typically never happen in practice, but could occur due to some edge case or implementation error in the H.264 encoder.

Exploit Scenario

An edge case in the H.264 codec causes the encoder to release frames shorter than three bytes. This causes the `FindNextH26XNaluIndex` function to read out of bounds and crash the client.

Recommendations

Short term, exit early in `FindNextH26XNaluIndex` if the buffer is too short to contain the start sequence.

Long term, use a property testing framework like RapidCheck to test individual functions on randomized inputs. Document the invariants assumed by each internal function.

10. Insufficient validation of proposal type

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-DISCE2EC-11
Target: `discord_common/native/secure_frames/mls/session.cpp`	

Description

The DAVE protocol only recognizes MLS `Add` and `Remove` proposals sent by the delivery service, which is added as an external sender to each group. All proposals are validated by the `mlspp` library, and the client then performs additional validation in the `ValidateProposalMessage` function.

```
bool Session::ValidateProposalMessage(
    ::mlspp::AuthenticatedContent const& message,
    ::mlspp::State const& targetState,
    std::set<std::string> const& recognizedUserIDs) const
{
    if (message.wire_format != ::mlspp::WireFormat::mls_public_message) {
        // ... [redacted]
    }
    if (message.content.epoch != targetState.epoch()) {
        // ... [redacted]
    }
    if (message.content.content_type() != ::mlspp::ContentType::proposal) {
        // ... [redacted]
    }
    if (message.content.sender.sender_type() != ::mlspp::SenderType::external) {
        // ... [redacted]
    }

    const auto& proposal =
        ::mlspp::tls::var::get<::mlspp::Proposal>(message.content.content);
    if (proposal.proposal_type() == ::mlspp::ProposalType::add) {
        // ... [redacted]
    }

    return true;
}
```

Figure 10.1: The Discord client fails to validate the proposal type (`discord_common/native/secure_frames/mls/session.cpp#201-246`)

This function ensures that the sender is the delivery service, but fails to ensure that the proposal is either an `Add` or a `Remove`.

According to the MLS RFC section on external proposals, the only proposal types allowed from an external sender are:

- `Add`
- `Remove`
- `PSK`
- `ReInit`
- `GroupContextExtensions`

However, we could not find anywhere in `mlspp` where this is actually checked. This essentially means that the server is free to send invalid proposals to selected clients, which could cause stability issues and potentially crash clients. However, we note that since the server is free to deny service or remove clients from a group at any time, we do not think that this represents a significant risk to the system.

Exploit Scenario

An attacker with a privileged network position sends invalid `Update` proposals to selected Discord clients, causing these clients to crash.

Recommendations

Short term, restrict the proposal type to Add and Remove proposals in the `ValidateProposalMessage` function.

Long term, consider performing a security review of the `mlspp` library to ensure that it conforms to the MLS RFC.

11. Application UI does not distinguish causes of verification failure

Severity: **Informational**	Difficulty: **Not Applicable**
Type: Authentication	Finding ID: TOB-DISCE2EC-12
Target: `discord_app/modules/rtc/hooks/useIsSecureFramesVerified.tsx`	

Description

Encrypted calls are considered verified if each participant's public key has been verified previously. However, the UI does not distinguish between users that have never been verified and users who are using a different key than was verified previously. This may make it difficult for users to correctly identify the cause of an unverified call. Since the most common reason for a conversation to become unverified is when a user adds a new device, some users may become complacent and re-verify people they know without going through the whole process. Although these users will be vulnerable to person-in-the-middle attacks, they can still be protected from scammers who use simple impersonation—for example, someone who copies an avatar and uses a lookalike username.

The UI should distinguish between failures indicating a new device or compromise within Discord—i.e., an unverified key for an account where some keys have been verified—and failures indicating a potentially malicious contact.

Recommendations

Short term, modify the UI to show more information when a conversation is not verified, especially whether there are already verified keys associated with the account.

Long term, account for the possibility of deceptive scammers in the E2EE threat model. Although encryption cannot protect against scam tactics in general, E2EE-related features such as verification may change a user's expectations of trustworthiness and make them more vulnerable in some cases.

12. Public key uploads are not bound to a specific account	
Severity: **Low**	Difficulty: **Medium**
Type: Denial of Service	Finding ID: TOB-DISCE2EC-13
Target: `discord_api/discord/views/voice.py`	

Description
The association between accounts and public keys is confirmed via a public key database. When uploading a public key, the API confirms that the key comes with a self-signature, and the RPC service confirms that the key has not been uploaded before. However, the self-signature includes only the key itself and the key's version, so if an attacker can intercept the signature for upload, they may be able to instead register the key to their own account. Without knowing the private key, the attacker cannot compromise E2EE sessions, but they can disrupt the target by causing their public key match queries to fail.

Exploit Scenario
Whenever Alice attempts to upload a public key to the matching service, Mallory intercepts the packet, extracts the self-signature, and instead registers it to her own account. Alice is unable to upload and match her keys, and gives up on Discord E2EE calls.

Recommendations
Short term, include account identifiers in the initial signature used for public key uploads.

Long term, always ensure that all relevant context is included in cryptographic signatures to prevent attackers from reusing them in other situations without having access to the secret key.

13. Sensitive data is not cleared from memory

Severity: **Medium**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-DISCE2EC-14
Target: Multiple	

Description

The `libdave` library does not make an effort to zeroize key material or other sensitive data when it is no longer used. This means that sensitive data will remain in memory and could potentially be accessed by a local attacker with the ability to read application memory.

```
CryptorManager::ExpiringCryptor CryptorManager::MakeExpiringCryptor(
    KeyGeneration generation)
{
    // Get the new key from the ratchet
    auto encryptionKey = keyRatchet_->GetKey(generation);
    auto expiryTime = TimePoint::max();

    // If we got frames out of order, we might have to create a cryptor for an
    // old generation. In that case, create it with a non-infinite expiry time
    // as we have already transitioned to a newer generation
    if (generation < newestGeneration_) {
        DISCORD_LOG(LS_INFO)
            << "Creating cryptor for old generation: " << generation;
        expiryTime = clock_.Now() + kCryptorExpiry;
    }
    else {
        DISCORD_LOG(LS_INFO)
            << "Creating cryptor for new generation: " << generation;
    }

    return {CreateCryptor(encryptionKey), expiryTime};
}
```

Figure 13.1: The symmetric encryption key (which has type `std::vector<uint8_t>`) is not cleared when a new `ExpiringCryptor` is created. (`discord_common/native/secure_frames/cryptor_manager.cpp#90-108`)

Many parts of the codebase use the `bytes` type provided by `mlspp`. This type clears the underlying buffer in the destructor using `std::fill`. However, this call will typically be removed by an optimizing compiler.

```
bytes keyData(keySize);

status = NCryptExportKey(key,
                         NULL,
                         BCRYPT_PRIVATE_KEY_BLOB,
                         NULL,
                         keyData.data(),
                         keyData.size(),
                         &keySize,
                         NCRYPT_SILENT_FLAG);
if (status != ERROR_SUCCESS) {
    DISCORD_LOG(LS_ERROR)
        << "Failed to export key in GetPersistedKeyPair: " << status;
    return nullptr;
}

if (keyData.size() < sizeof(BCRYPT_KEY_BLOB)) {
    DISCORD_LOG(LS_ERROR)
        << "Exported key blob too small in GetPersistedKeyPair/convertBlob: "
        << keyData.size();
    return nullptr;
}

BCRYPT_KEY_BLOB* header = (BCRYPT_KEY_BLOB*)keyData.data();
if (header->Magic != keyBlobMagic) {
    DISCORD_LOG(LS_ERROR)
        << "Exported key blob has unexpected magic in GetPersistedKeyPair: "
        << header->Magic;
    return nullptr;
}

if (!convertBlob(keyData)) {
    DISCORD_LOG(LS_ERROR) << "Failed to convert key in GetPersistedKeyPair";
    return nullptr;
}

return std::make_shared<::mlspp::SignaturePrivateKey>(
    ::mlspp::SignaturePrivateKey::parse(suite, keyData));
```

Figure 13.2: Loading persistent keys on Windows could leave a copy of the private key in memory. (discord_common/native/secure_frames/mls/detail/persisted_key_pair_win.cpp#166-200)

```
bytes
KeyScheduleEpoch::do_export(const std::string& label,
                            const bytes& context,
                            size_t size) const
{
  auto secret = suite.derive_secret(exporter_secret, label);
  auto context_hash = suite.digest().hash(context);
  return suite.expand_with_label(secret, "exported", context_hash, size);
}
```

Figure 13.3: The MLS key exporter provided by the `mlspp` library could leave a copy of the HMAC key in memory. (`discord_common/native/third_party/mlspp/src/key_schedule.cpp#406-414`)

Exploit Scenario

An update to the Discord client introduces a memory disclosure vulnerability where uninitialized heap memory is revealed to all other members of the group. In rare instances, this includes parts of the persisted private identity key. A malicious user discovers the vulnerability and uses it to recover other group members' identity keys, allowing her to impersonate them in the future.

Recommendations

Short term, determine and document whether local memory access falls within the threat model for the DAVE protocol. Consider disabling crash dumps in the event of an application crash to prevent sensitive data being leaked to disk if the application crashes.

Long term, use a dedicated type for sensitive data like key material and ensure that the underlying storage buffer is cleared when the variable goes out of scope. The `memset_explicit`, `explicit_bzero`, `SecureZeroMemory` and `OPENSSL_cleanse` functions can all be used to zeroize memory in a way that should prevent the compiler from removing the operation.

14. Session not closed on bad MLS binary input

Severity: **Undetermined**	Difficulty: **Low**
Type: Error Reporting	Finding ID: TOB-DISCE2EC-15
Target: `discord_voice`	

Description

The `mls_key_package`, `mls_commit_welcome`, and `mls_invalid_commit_welcome` opcodes handlers (figure 14.1) incorrectly handle bad data. The error handling branch is ineffective since the handlers return `{:noreply, state |> schedule_timeout}`, which should be returned only when there was no error.

```
defp handle_incoming_message(Opcodes.mls_key_package(), data, state) do
  with {true, key_package_binary} <- sanitize({:binary, data}) do
    Server.update_key_package(state.server_pid, state.user_id, key_package_binary)
  else
    _ ->
      do_close(:bad_request, state)
  end

  {:noreply, state |> schedule_timeout}
end
```

Figure 14.1: Incorrect error handling that does not close the connection (discord_voice/lib/discord_voice/session/session.ex#L904-L939)

Recommendations

Short term, move `{:noreply, state |> schedule_timeout}` under the first branch of the `with` statement. Implement a test case that exercises bad inputs on all handlers.

Long term, ensure that tests cover all branches of the program, and consult test coverage reports to identify untested code paths.

15. Static key ratchet used in production code

Severity: **High**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-DISCE2EC-16
Target: `discord_native_lib/src/media/connection.cpp`	

Description

The static key ratchet is an insecure keying algorithm that uses the user ID to key the underlying AEAD. It is unproblematic to use for testing purposes, but would be catastrophic to use in a production environment since it completely breaks all confidentiality and integrity guarantees provided by the protocol.

The `Connection::MakeUserKeyRatchet` method returns a static key ratchet instance if the given protocol version is less than `kTempMLSProtocolVersionSeparator`.

```
std::unique_ptr<secure_frames::IKeyRatchet> Connection::MakeUserKeyRatchet(
    std::string const& userId,
    secure_frames::ProtocolVersion protocolVersion) const
{
    if (protocolVersion < secure_frames::kTempMLSProtocolVersionSeparator) {
        return MakeSimpleKeyRatchet(userId, protocolVersion);
    }
    else {
        if (!mlsSession_) {
            DISCORD_LOG(LS_WARNING)
                << "Cannot make user key ratchet: MLS session not initialized";
            return nullptr;
        }

        return mlsSession_->GetKeyRatchet(userId);
    }
}
```

Figure 15.1: The Discord native library will use an insecure keying algorithm for protocol versions below `kTempMLSProtocolVersionSeparator`. (discord_native_lib/src/media/connection.cpp#700-715)

```
std::unique_ptr<secure_frames::IKeyRatchet> MakeSimpleKeyRatchet(
    const std::string& userID,
    uint32_t version)
{
    if (version == 0) {
        return nullptr;
```

```
    }

    return std::make_unique<secure_frames::StaticKeyRatchet>(userID);
}
```

Figure 15.2: The static key ratchet algorithm is insecure and should not be exposed in production builds. (discord_native_lib/src/media/frame_cryptors.cpp#29-37)

The `MakeUserKeyRatchet` method is called when the connection is first set up, as well as when the client receives a `secure-frames-prepare-transition` control message from the server. We could not find any logic in the client that would prevent the Discord server from sending a `secure-frames-prepare-transition` message with a protocol version lower than `kTempMLSProtocolVersionSeparator`; this effectively disables encryption for all clients.

Exploit Scenario

The static key ratchet is not removed from the production version of the Discord client before the system is deployed. This allows a Discord insider to disable encryption and eavesdrop on selected end-to-end encrypted calls.

Recommendations

Short term, refactor the Discord native library and `libdave` library to include the static key ratchet only in debug builds.

Long term, use the static key ratchet only for testing purposes.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Configuration	The configuration of system components in accordance with best practices
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Data Handling	The safe handling of user inputs and data processed by the system
Documentation	The presence of comprehensive and readable codebase documentation
Maintenance	The timely maintenance of system components to mitigate risk
Memory Safety and Error Handling	The presence of memory safety and robust error-handling mechanisms
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Automated Testing

This section describes the setup of the automated analysis tools used during this audit.

C.1. CodeQL

CodeQL can be installed from the CodeQL release page. To build a CodeQL database for a C++ or JavaScript library, navigate to the root folder of the library and run `codeql database create --language <LANGUAGE> --command <BUILD COMMAND> codeql.db`, where the language parameter is given by either "`cpp`" or "`javascript`", depending on the library. (For interpreted languages like JavaScript, the build command parameter can be left out.)

To analyze a database for security issues, use the `codeql database analyze` command. For more information on CodeQL, refer to the Trail of Bits Testing Handbook.

Since we did not have access to other private repositories under the Discord organization, we were not able to build the `libdave` library as a standalone library under CodeQL during the engagement. However, we did use CodeQL to analyze the DAVE JavaScript library under `discord_common/js/packages/secure_frames`. We ran all queries included in the `codeql/security-extended` query suite on the codebase. This analysis did not identify any issues in the implementation.

C.2. Cppcheck

Cppcheck can be installed using Homebrew on MacOS and Linux systems using the command `brew install cppcheck`. To run Cppcheck on a C++ library, navigate to the root folder of the library and run the command `cppcheck .`.

We ran Cppcheck on the implementation of the DAVE protocol using the flag `--check-level=exhaustive`. This analysis did not identify any issues.

C.3. Flawfinder

Flawfinder can be installed using Homebrew on MacOS and Linux systems using the command `brew install flawfinder`. To run Flawfinder on a C++ library, simply run the command `flawfinder --sarif .` in the root folder of the library.

We ran Flawfinder on the DAVE protocol implementation and triaged the results. This analysis did not result in any issues or code quality recommendations.

C.4. Semgrep

Semgrep can be installed using `pip` by running `python3 -m pip install semgrep`. To run Semgrep on a codebase, run `semgrep --config "<CONFIGURATION>"` in the root directory of the project. Here, `<CONFIGURATION>` can be a single rule, a directory of rules,

or the name of a rule set hosted on the Semgrep registry. For more details on Semgrep, refer to our Testing Handbook.

We ran Semgrep using the automatically selected rule sets (obtained by running the command `semgrep -c auto .`), as well as the following rulesets from the Semgrep registry.

```
p/trailofbits
p/security-audit
p/cpp-audit
p/c
p/c-audit-banned-functions
p/elixir
p/javascript
```

Figure C.1: Semgrep rulesets used during the review

Additionally, we ran a number of internal Semgrep rules against the codebase. This analysis flagged a private key file in the `discord_voice/priv` directory, but did not identify any other issues.

D. Code Quality Recommendations

The following section contains code quality recommendations for issues we identified that do not have any immediate security implications.

- **Add code-level documentation of the encrypted frame format.** The `InboundFrameProcessor::ParseFrame` method would benefit from a comment describing the frame format assumed by the parser. Currently, this has to be reverse engineered from the source code, which makes reviewing and refactoring the method unnecessarily difficult.
- **Call to `memcpy` should use the size of the destination buffer.** The calls to `memcpy` in `Decryptor::Decrypt` should use the size of the destination buffer rather than the source buffer to ensure that the function call does not write outside the destination buffer.
- **The size of the truncated nonce should be validated.** The frame parser uses the `ReadLeb128` function to read the truncated nonce. The function returns a 64-bit integer and the return value is assigned to a 32-bit integer. The return value should be validated to ensure that it fits in 32 bits.

```
truncatedNonce_ = ReadLeb128(readAt, end);
```

Figure D.1: The truncated nonce from `ReadLeb128` may not fit in 32 bits. (discord_common/native/secure_frames/frame_processors.cpp#L205)

- **Validate user credential strings.** The `CreateUserCredential` function uses `std::stoull` to parse Discord user IDs. However, it fails to check that the input string is a valid 64-bit integer. This should be checked to ensure that user IDs follow the correct format.
- **Use the dedicated types consistently throughout the code.** The `MLS.LeafNode.BasicCredentialUserID` module uses the `pos_integer()` type instead of the `MLS.Type.user_id()` type for user ID.
- **Document or remove unused function arguments.** The `key_context` argument is unused and it is not clear why it was left in the code.
- **Avoid using magic values.** The ciphersuite values hard-coded here are already named in the `MLS.Ciphersuite` module.
- **Fix incorrect type signatures.** The `MLS.Parameters.signature_algorithm` function also returns the `:secp256r1` atom.
- **Remove unreachable branches.** This code path is unreachable because the `signature_algorithm` function already throws an unsupported algorithm.

- **Remove code duplication.** The `MLS.Signature.do_verify_signature` function contains repeated code for different signature algorithms. The `MLS.Parameters.signature_algorithm` function can return all of the arguments required by the `:crypto.verify` function, removing the need for different branches and code duplication.

E. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

From September 4 to September 6, 2024, Trail of Bits reviewed the fixes and mitigations implemented by the Discord team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the 15 issues described in this report, Discord has resolved 12 issues, has partially resolved one issue, and has not resolved the remaining two issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	Undefined behavior in frame processor	Resolved
2	Insufficient validation of unencrypted ranges	Resolved
3	Insufficient Windows key data size validation	Resolved
4	Call participants can send different media frames to lagging participants	Resolved
5	Encrypted frames can be delivered multiple times or out-of-order	Resolved
6	Unencrypted range offsets and sizes are not authenticated	Resolved
7	Insufficient size validation in SerializeUnencryptedRanges	Resolved
8	Integer overflow during encrypted frame validation	Resolved
9	Out-of-bounds read in FindNextH26XNaluIndex	Resolved
10	Insufficient validation of proposal type	Resolved
11	Application UI does not distinguish causes of verification failure	Resolved

12	Public key uploads are not bound to a specific account	**Resolved**
13	Sensitive data is not cleared from memory	**Resolved**
14	Session not closed on bad MLS binary input	**Resolved**
15	Static key ratchet used in production code	**Resolved**

Detailed Fix Review Results

TOB-DISCE2EC-1: Undefined behavior in frame processor
Resolved in commit 301f58369a285f8fa8107352f565d43f7d93d63b. The pointer arithmetic has been moved to occur after the buffer's size is checked.

TOB-DISCE2EC-2: Insufficient validation of unencrypted ranges
Resolved in commit ac5117f66bed96fa8821f6541becb7b11aa23a0a. The range checks now use compiler intrinsics that indicate if an overflow has occurred, and the checks fail in the case of an overflow.

TOB-DISCE2EC-4: Insufficient Windows key data size validation
Resolved in commit d735f4b3f92741ea6483bd351f0176979457673b. The blob size is now correctly checked against `BCRYPT_ECCKEY_BLOB` instead of `PBCRYPT_ECCKEY_BLOB`.

TOB-DISCE2EC-5: Call participants can send different media frames to lagging participants
Resolved. Documentation cautioning about the lack of key commitment has been added to the design document.

TOB-DISCE2EC-6: Encrypted frames can be delivered multiple times or out-of-order
Resolved in commit 3bc779728c485c0393528e16616b31d373ddf921. The `CryptorManager` type now tracks already seen nonces which prevents replay attacks.

TOB-DISCE2EC-7: Unencrypted range offsets and sizes are not authenticated
Resolved. This issue has been documented, and TOB-DISCE2EC-1 has been fixed, making this finding no longer exploitable. The Discord team should still consider adding an authentication layer to the ranges, and should thoroughly test and validate any functions that deserialize or otherwise process unauthenticated data to defend against exploitation.

TOB-DISCE2EC-8: Insufficient size validation in SerializeUnencryptedRanges
Resolved in commit eb9cb0b8bf14acd055c543271d0ad92ea429eb27. The function `SerializeUnencryptedRanges` now checks the serialized size of each range against the remaining buffer size, by checking that `rangeSize > end - writeAt`, which cannot overflow.

TOB-DISCE2EC-9: Integer overflow during encrypted frame validation
Resolved in commit ac9e2cd03e2c62caa6d202311608bf0c92d2985a. The expression `std::max(size_t{0}, encryptedSectionStart - padding)` has been replaced with `encryptedSectionStart - std::min(encryptedSectionStart, size_t{Padding})`, which does not have problematic overflow behavior.

TOB-DISCE2EC-10: Out-of-bounds read in FindNextH26XNaluIndex
Resolved in commit 09ff623326c42bada57204cc13cf2fbcce3d5d26. The function `FindNextH26XNaluIndex` now immediately returns if the buffer is too small.

TOB-DISCE2EC-11: Insufficient validation of proposal type
Resolved in commit 99204180f2674a957dce2b2b1800e3b123b8debe. The function `Session::ValidateProposalMessage` has been modified to only accept `Add` and `Remove` proposals.

TOB-DISCE2EC-12: Application UI does not distinguish causes of verification failure
Resolved in commit 460184d8ca6b8486359f82ad1e8ab8bed134dcb9. The verification code UI now also shows the number of previously verified devices.

TOB-DISCE2EC-13: Public key uploads are not bound to a specific account
Resolved in commit 99204180f2674a957dce2b2b1800e3b123b8debe. The signature now contains the session ID in addition to the key itself.

TOB-DISCE2EC-14: Sensitive data is not cleared from memory
Resolved in commit 99204180f2674a957dce2b2b1800e3b123b8debe and pull request 432 to `mlspp`. The pull requests to `mlspp` ensures that the compiler cannot optimize away calls to `std::fill` by casting the corresponding pointers to volatile pointers. The team has also updated the `EncryptionKey` type to use the `bytes` type defined by the `mlspp` library.

TOB-DISCE2EC-15: Session not closed on bad MLS binary input
Resolved in commit 4294dc9c7295b87c2a380c2e4dfca3eb2ec084bc. The non-error returns have been moved into the non-error branch, and regression tests have been added.

TOB-DISCE2EC-16: Static key ratchet used in production code
Resolved in commit 9c252427bb6b65f7c559bd7f2015cce6cbd9e53e. The function `MakeSimpleKeyRatchet` has been removed, and the corresponding branch of the `MakeUserKeyRatchet` function now returns `nullptr` instead of returning a static key ratchet.

F. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.