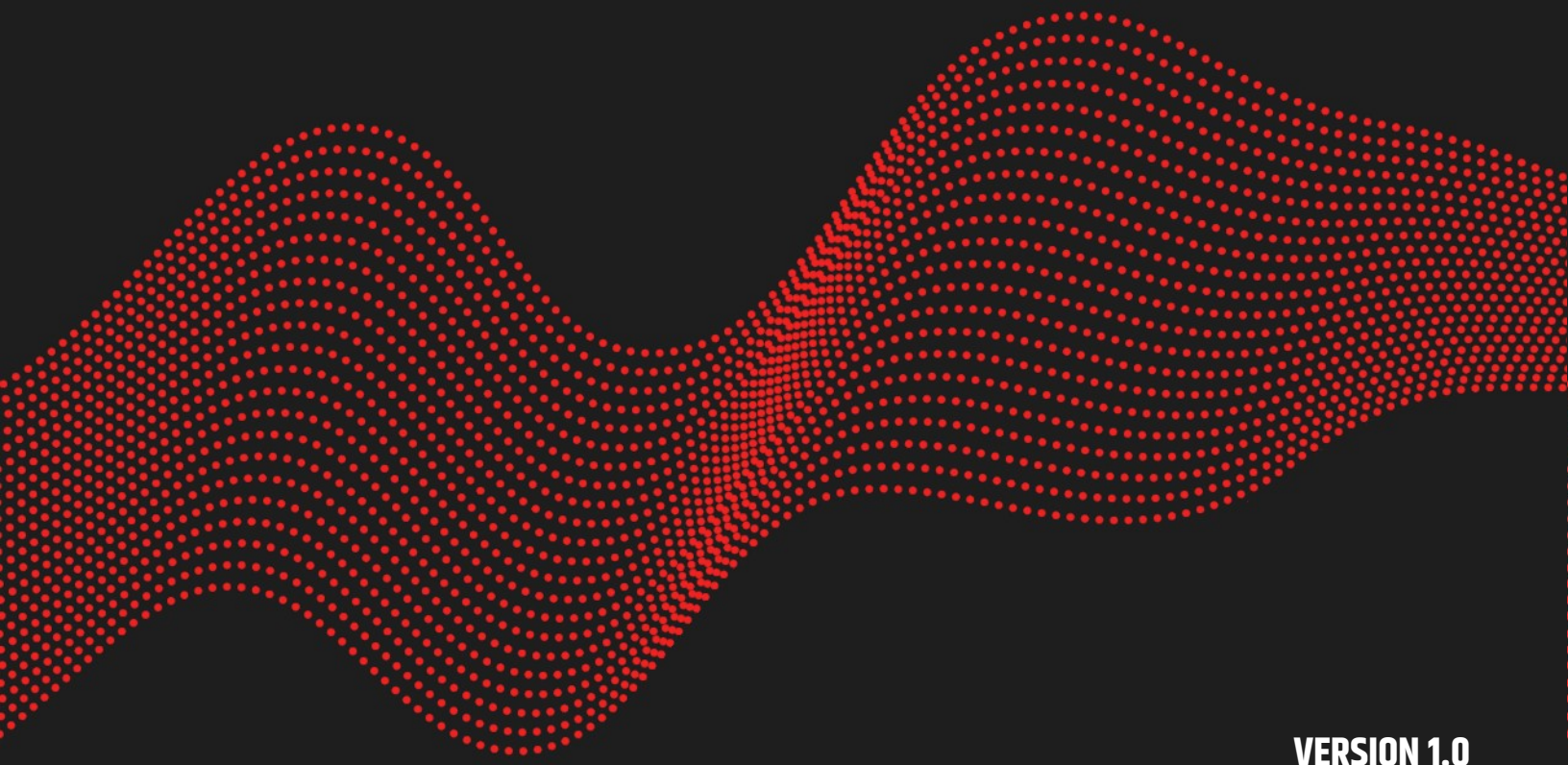
SYNACKTIV

ANSSI – SECURITY ASSESSMENT REPORT

Audit MLA

2026/01/30



VERSION 1.0

Contents

Introduction

Context and objectives

ANSSI has asked Synacktiv to perform a code audit on the Multi Layer Archive (MLA) utility. This application is an archive utility written in Rust, with support for traditional and post-quantum signing hybridization, and developed by ANSSI.

The tests were performed using a white-box approach, where the GitHub repository with a specific commit hash was given to Synacktiv consultants.

The objectives of these tests were to:

- Identify vulnerabilities and their associated risks.
- Exploit vulnerabilities.
- List remediations that will improve the security level of the application.

Scope and limits

The audit focused on the following security features for MLA:

- Ensure that a crafted archive cannot lead to arbitrary writes either through path traversal or silent file overwrite
- Ensure that cryptography provides state-of-the-art confidentiality, integrity and authentication
- Ensure that mlar command line interface tool provides sensible default security settings
- Implementation of secret zeroization
- Ensure that rust best practices are followed, including review of unsafe code

The audit scope only includes the following assets :

Assets	
Asset name	https://github.com/ANSSI-FR/MLA version 2.0.0-beta, commit hash 6a08b16c1f132e8ce5d92f047f711d56afcc277e

Black-box	Grey-box	White-box
No information provided.	Authenticated access provided with dedicated accounts	Access to the application source code and documentation.

Team

Name	Role
Clients and Auditees	
ANSSI	Technical Advisor SDO
ANSSI	SDO CISO
ANSSI	MLA Development Team
Audit team	
Synacktiv	Auditor
Synacktiv	Auditor

Timeline

The security assessment was performed from the Synacktiv offices, from the 5th to the 16th of January 2026.

Date	Description
2026/11/24	Kick-off
2026/01/05	Beginning of the tests
2026/01/07	Follow-up meeting
2026/01/09	Follow-up meeting
2026/01/16	Closing meeting

Version history

Version	Comment
V1	Initial version

Metrics

Security level rating

Synacktiv experts determine a global security level of the audited target given the audited scope, corresponding observations and state of the art.

HIGH

No sensitive asset was compromised or impacted during the audit period. A few improvements were nonetheless identified.

SATISFACTORY

No sensitive asset was compromised or impacted during the audit period. However, several identified vulnerabilities may help to impact these assets given specific prerequisites (publication of a new vulnerability, different network setup, etc.).

INSUFFICIENT

Some attack vectors impacting sensitive assets were identified during the audit period.

LOW

Sensitive assets of the audited scope were compromised during the audit period.

Vulnerability rating

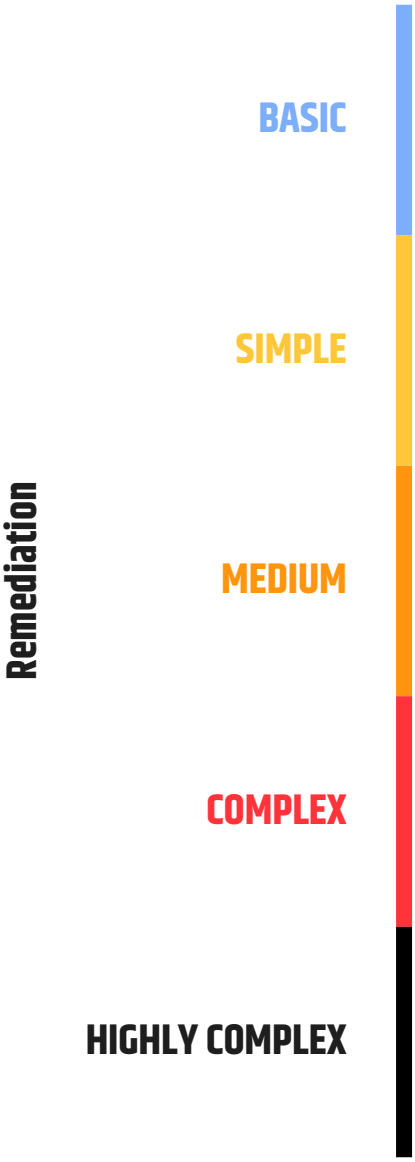
Synacktiv experts classify the sensitivity of the identified vulnerabilities and determine a grade of Severity (S), resulting from the product of two intermediate scores Probability (P), and Impact (I).

This scoring system is close to the concept of probabilistic risk assessment used in the industrial sector.

Probability	RARE	Hidden attack vector and/or needing high prerequisites hard to obtain.
	LOW	Vulnerability difficult to identify, the attacker must have technical information on the target or must exploit intermediate vulnerabilities.
	MEDIUM	Vulnerability identifiable by an average attacker.
	HIGH	Vulnerability easy to identify by an attacker, attack vector accessible without any particular constraint.
	FREQUENT	Vulnerability trivial to identify and potentially already identified.
Impact	MINIMAL	Exploitation of the vulnerability makes it possible to obtain non-sensitive technical information on the target.
	LOW	Exploiting the vulnerability provides technical information about the target.
	MEDIUM	The vulnerability allows an attacker to partially compromise the security of the target.
	HIGH	The attacker can access and/or modify sensitive information compromising the security of the target and its environment.
	MAXIMAL	The attacker can compromise the majority of the information system or the most sensitive data through the vulnerability.
Severity	REMARK	Negligible risk, non-compliance with hardening procedures. The vulnerability does not pose a significant risk to the target.
	LOW	Vulnerability remediation is used to comply with good security practices.
	MEDIUM	Vulnerability presents a risk to the target and needs to be fixed in the short term.
	HIGH	Vulnerability presents a significant risk for the target and must be fixed in the very short term.
	CRITICAL	Vulnerability presents a major risk for the target and requires immediate consideration.

Remediation rating level

Synacktiv provides an indicative level of complexity for vulnerability remediation. Due to limited visibility across the entire information system, this level may differ from the actual complexity of remediation.



BASIC — Instant, simple remediation requiring no specific security skills and without side effects.

SIMPLE — Low-complexity remediation that can be quickly performed by the development or administration team of the information system without side effects.

MEDIUM — Remediation requiring a preliminary study and a minimal monitoring of implemented measures by the administration team of the information system.

COMPLEX — High-complexity remediation involving structural changes to the information system and requiring an implementation by a dedicated team.

HIGHLY COMPLEX — A highly complex remediation involving a partial overhaul of the information system as well as a substantial human and financial investment.

Executive summary

Global security level

The security assessment performed by Synacktiv on MLA revealed a high security level.

Indeed, no compromise scenarios, allowing to alter or recover a properly encrypted and signed archive have been identified.

Each layer is properly implemented independently of others, ensuring that each properly performs a single task and can be composed with any other. Authenticity of the archive is guaranteed by the signature layer leveraging state-of-the-art cryptography and hybrid public key exchange for post-quantum cryptography. The derived symmetric key and nonce are then passed to the encryption layer leveraging AES-GCM for encryption providing confidentiality and authentication of ciphertext and key validation using key commitment blocks.

Moreover, even though cryptographic algorithms are set for each layer, hampering crypto agility, the archive format uses magic bytes to identify each layer independently. It then allows the software to implement new layers to introduce alternative algorithm for each feature, ensuring long term adaptability.

Nonetheless, few, issues also have been identified. The first one lies in the zeroization process that was only partial due to buffer regrows, and the second one is related to file permission on generated private keys allowing world readable access. Moreover, two informational points have been identified linked to non-exploitable weaknesses.

Eventually it should be noted that all issues, even informational ones, identified were fixed during the assessment.

Synacktiv identified 4 security issues: **2 of low severity** and **2 remarks**.

Strengths and areas of improvements

Robust cryptography

The application uses state-of-the-art cryptography.

Memory Safety

The application is written in safe rust, which prevents entire categories of bugs.

Protection against common attacks

Default privileges on private key files

When private keys are created, they inherit the default permissions. They are for example readable by every user on Linux.

Incomplete memory zeroization

In some instances, the application failed to zeroize memory where secrets were stored. This issue was corrected by ANSSI during

The application was built to prevent common attacks such as Zip Slip and command injections.

the audit.

Vulnerabilities summary

2	2	0	0	0
Remark	Low	Medium	High	Critical

ID	Name and remediation	P	I	S
V-01	INCOMPLETE MEMORY ZEROIZATION			
[p°14]	Use the zeroize() function on every variable holding secrets before freeing the memory.			
V-02	INAPPROPRIATE DEFAULT PRIVILEGES ON PRIVATE KEY FILES			
[p°17]	Enforce strict permission on MLA key files to only allow owner read and write.			
V-03	POSSIBLE INITIALIZATION VECTOR REUSE IF THE SIZE OF DATA CHUNKS IS EXTENDED			
[p°21]	Document the security consideration associated with the size of the chunks, to avoid any dangerous refactoring.			
V-04	MULTIPLE INTEGER OVERFLOW			
[p°24]	An efficient way of identifying possible integer overflow in Rust is to use a Clippy linter such as arithmetic_side_effects. When the linter is activated, cargo build will list all identified integer overflow.			

Vulnerabilities details

V-01 Incomplete memory zeroization

Probability	Impact	Severity	Remediation
LOW	LOW	LOW	SIMPLE

Observations

Although extensive efforts were made to zeroize secret-bearing variables after use, Synacktiv consultants discovered several instances where memory remained uncleared.

In the file src/crypto/mlakey.rs, in the function `serialize_decryption_private_key` (line 207).

```
fn serialize_decryption_private_key<W: Write>(&self, mut dst: W) -> Result<(), Error> {
    dst.write_all(MLA_PRIV_DEC_KEY_HEADER)?;
    let mut b64data = vec![];
    b64data.extend_from_slice(DEC_METHOD_ID_0_PRIV);
    b64data.extend_from_slice(EMPTY_OPTS_SERIALIZATION);
    b64data.extend_from_slice(&self.private_key_ecc.to_bytes());
    b64data.extend_from_slice(self.private_key_seed_ml.to_d_z_64().as_ref());
    let mut encoded = base64_encode(&b64data);
    dst.write_all(&encoded)?;
    encoded.zeroize();
    dst.write_all(b"\r\n")?;
    Ok(())
}
```

The variable `b64data` is not zeroized. The function `base64_encode` does not take ownership of the `b64data` variable.

The call to the function `to_bytes()` will also result in the data being copied in memory without being wiped.

Another issue is the multiple calls to the function `extend_from_slice`. If the appended data does not fit inside the current structure, `extend_from_slice` will allocate a new larger buffer, and free the previous buffer without zeroing it.

The issue of extending a vector is also present in the `base64_encode` function, present in the file `mla/src/base64.rs`.

```
pub(crate) fn base64_encode(data: &[u8]) -> Vec<u8> {
    let mut encoded = Vec::new();
    let mut i = 0;
    while i < data.len() {
        let mut val: u32 = 0;
```

```
        let mut n = 0;
        while n < 3 && i < data.len() {
            val = (val << 8) | (data[i] as u32);
            n += 1;
            i += 1;
        }
        if n == 1 {
            val <<= 16;
        } else if n == 2 {
            val <<= 8;
        }
        for j in 0..4 {
            if j < (n + 1) {
                let mut idx = ((val >> ((3 - j) * 6)) & 0x3F) as usize;
                encoded.push(BASE64_CHARS[idx]);
                idx.zeroize();
            } else {
                encoded.push(b'=');
            }
        }
        val.zeroize();
    }
    encoded
}
```

Risks

Secrets may still be present in memory after running the application.

An attacker having physical access to the device may access to secrets freed by the allocator but not properly zeroed. This may be accessed either with a physical or privileged access to the host system. Eventually unprivileged access may occur depending on the host configuration hardening.

Recommendations

Use the `zeroize()` function on every variable holding secrets before freeing the memory.

To avoid cloning then borrowing with `&value.to_bytes()`, without having a reference to bytes to be zeroed, it is recommended to use direct borrowing of bytes with `value.as_bytes()`.

To avoid reallocation on growth, if size cannot be predicted or must be heap allocated it can be constructed with either of these functions:

```
/// Concatenate buffers leveraging iters for declarative code.
///
/// Given that .size_hint() is known for each of the iterator, chains sums them and
/// .collect() will only allocate required memory once
pub fn concat_iter(first: &[u8], second: &[u8], third: &[u8]) -> Vec<u8> {
    first.iter().chain(second).chain(third).cloned().collect()
}
```

```
/// Pre-allocate memory and extend slice for imperative code.
pub fn concat_extend(first: &[u8], second: &[u8], third: &[u8]) -> Vec<u8> {
    let mut res = Vec::with_capacity(first.len() + second.len() + third.len());
    res.extend_from_slice(first);
    res.extend_from_slice(second);
    res.extend_from_slice(third);
    res
}
```

Otherwise, if the final buffer size is known at compile time and small enough, it can be stack allocated and returned as such:

```
pub fn concat_buffers(first: &[u8; 32], second: &[u8; 32]) -> [u8; 64] {
    let mut out = [0u8; 64]
    out[..32].copy_from_slice(first);
    out[32..].copy_from_slice(second);
    out
}
```

Note that this approach is compatible with no_std crates.

A small crate with no_std can be used to define functions that will interact with sensitive data to ensure that no allocation will occur in their code. If size cannot be predicted, the function can still take a mutable reference to a pre allocated buffer:

```
#[no_std]

pub fn concat_buffers_no_std<'a>(first: &[u8], second: &[u8], out: &'a mut [u8]) -> &'a
[u8] {
    // Start by the end to trigger a single boundary check on the whole buffer.
    out[first.len()..first.len()+second.len()].copy_from_slice(second);
    out[..first.len()].copy_from_slice(first);
    out
}
```

Eventually, the global allocator could be replaced to enforce zeroization of every buffer upon free. Such allocators can be found in the **`zeroizing-alloc`** crate for instance. However, this may be the responsibility of projects calling the **`mla`** library to make such choices.

Bibliography

- https://docs.rs/zeroizing-alloc/latest/zeroizing_alloc/

V-02 Inappropriate default privileges on private key files

Probability	Impact	Severity	Remediation
LOW	LOW	LOW	SIMPLE

Observations

Private key files generated with `mlar` use default file permissions, meaning they honor UMASK on Linux, and they inherit permission from parent folder on Windows.

```
$ ./target/release/mlar keygen user
$ ls -l user.mla*
-rw-r--r-- 1 user user  452 16 janv. 10:45 user.mlapriv
-rw-r--r-- 1 user user 5870 16 janv. 10:45 user.mlapub
```

By default, private key files are thus world readable on Linux and may be readable if created in the wrong folder on Windows.

Risks

An attacker having access on the system may use world readable permission of MLA private key files to extract them without having to compromise the owning user account. Such a scenario may happen with a compromised service exposed over the network.

Recommendations

Enforce strict permission on MLA key files to only allow owner read and write.

The file could be created leveraging OpenOptions and settings mode before opening. The following Rust snippet demonstrates how to create a file with restricted owner-only permissions:

```
use std::fs::File;
use std::io::Error;
use std::path::Path;

/// Create a file with owner only rw access.
pub fn create_private_key<P: AsRef<Path>>(p: P) -> Result<File, Error> {
    create_private_key_os(p)
}

#[cfg(target_family = "unix")]
fn create_private_key_os<P: AsRef<Path>>(p: P) -> Result<File, Error> {
    unix::create_private_file(p)
```

```
}

#[cfg(target_family = "unix")]
mod unix {
    use std::fs::{File, OpenOptions};
    use std::io::Error;
    use std::os::unix::fs::OpenOptionsExt;
    use std::path::Path;

    pub fn create_private_file<P: AsRef<Path>>(p: P) -> Result<File, Error> {
        OpenOptions::new()
            .create_new(true)
            .read(true)
            .write(true)
            .mode(0o600)
            .open(p)
    }
}

#[cfg(target_family = "windows")]
fn create_private_key_os<P: AsRef<Path>>(p: P) -> Result<File, Error> {
    windows::create_private_file(p)
}

#[cfg(target_family = "windows")]
mod windows {
    use std::fs::File;
    use std::io::Error;
    use std::os::windows::ffi::OsStrExt;
    use std::os::windows::io::FromRawHandle;
    use std::path::Path;
    use std::ptr;
    use windows::core::PCWSTR;
    use windows::Win32::Foundation::{GENERIC_WRITE, INVALID_HANDLE_VALUE, HLOCAL,
LocalFree};
    use windows::Win32::Security::{SECURITY_DESCRIPTOR, PSECURITY_DESCRIPTOR};
    use windows::Win32::Security::Authorization::{
        ConvertStringSecurityDescriptorToSecurityDescriptorW,
        SDDL_REVISION_1,
    };
    use windows::Win32::Security::SECURITY_ATTRIBUTES;
    use windows::Win32::Storage::FileSystem::{
        CreateFileW, CREATE_NEW, FILE_ATTRIBUTE_NORMAL, FILE_SHARE_READ,
    };

    pub fn create_private_file<P: AsRef<Path>>(p: P) -> Result<File, Error> {
        let filename_wide: Vec<u16> = p
            .as_ref()
            .as_os_str()
            .encode_wide()
            .chain(std::iter::once(0))
            .collect();

        // Create a SDDL disabling inheritance
        let mut sd_ptr: *mut SECURITY_DESCRIPTOR = ptr::null_mut();
        create_security_descriptor_from_sddl("D:P(A;;FA;;;OW)", ptr::from_mut(&mut
sd_ptr).cast())?;

        // Wrap it in a struct ensuring proper free on drop
```

```
        struct SDWrapper(*mut SECURITY_DESCRIPTOR);

        impl Drop for SDWrapper {
            fn drop(&mut self) {
                if ! self.0.is_null() {
                    // SAFETY: pointer is not null
                    unsafe { LocalFree(Some(HLOCAL(self.0.cast()))) };
                }
            }
        }

        let _wrapped = SDWrapper(sd_ptr);


        // Create SA with SDDL
        let mut sa = SECURITY_ATTRIBUTES {
            nLength: std::mem::size_of::<SECURITY_ATTRIBUTES>() as u32,
            lpSecurityDescriptor: sd_ptr.cast(),
            bInheritHandle: false.into(),
        };

        let file = unsafe {
            // Create file with SDDL
            let handle = CreateFileW(
                PCWSTR(filename_wide.as_ptr()), // File name
                GENERIC_WRITE.0,                // Access rights
                FILE_SHARE_READ, // Share mode (allow others to read? No, if ACL works)
                Some(&mut sa),   // Security Attributes (The ACL)
                CREATE_NEW,      // Creation disposition (Fail if exists)
                FILE_ATTRIBUTE_NORMAL, // Flags
                None, // Template file
            )?;

            // Check handle
            if handle == INVALID_HANDLE_VALUE {
                // Get the last error code for debugging
                let error = windows::core::Error::from_thread();
                return Err(error.into());
            }

            // Create file from handle
            File::from_raw_handle(std::mem::transmute(handle))
        };
        // Return file
        Ok(file)
    }

    // Creating a SD from SDDL.
    fn create_security_descriptor_from_sddl(
        sddl: &str, sd_ptr: *mut PSECURITY_DESCRIPTOR
    ) -> Result<*mut PSECURITY_DESCRIPTOR, std::io::Error> {
        let wide: Vec<u16> =
sddl.encode_utf16().chain(std::iter::once(0u16)).collect();

        // SAFETY: sd_ptr points to a pointer for SECURITY_DESCRIPTOR allocation
        unsafe {
            let _ = ConvertStringSecurityDescriptorToSecurityDescriptorW(
                PCWSTR(wide.as_ptr()),
                SDDL_REVISION_1,
```

```
                sd_ptr,
                None,
            )?;
        }
        Ok(sd_ptr)
    }
}
```

Note that for Windows, some unsafe calls are required due to the lack of safe abstraction with OpenOptions. Moreover, the following dependencies must be added to Cargo.toml:

```
[target.'cfg(windows)'.dependencies]
windows = { version = "0.62.2", default-features = false, features = [
   "Win32_Storage",
   "Win32_Storage_FileSystem",
   "Win32_Security",
   "Win32_Security_Authorization",
   "Win32_System",
   "Win32_System_Memory"
] }
windows-result = "0.4.1"
```

Bibliography

- https://doc.rust-lang.org/std/fs/struct.OpenOptions.html

V-03 Possible Initialization Vector reuse if the size of data chunks is extended

Probability	Impact	Severity	Remediation
RARE	MINIMAL	REMARK	BASIC

Observations

There is a risk of nonce reuse in the usage of the AES-GCM cipher if the chunks of data is extended beyond 232 blocks of 16 bytes (around 64 GiB). Note that during the assesment the chunk size is hardcoded to 128KiB, preventing exploitation of this vulnerability in AES-GCM implementation in MLA.

Nonces are calculated using the `compute_nonce` function.

```
/// Compute the nonce for a given sequence number (RFC 9180 §5.2)
pub(crate) fn compute_nonce(base_nonce: &Nonce, seq: u64) -> Nonce {
    // RFC 9180 §5.2: seq must not be superior to 1 << (8*Nn)
    // As we use AES-256-GCM, Nn = 12 (RFC 9180 §7.3), so u64 is always enough

    // Nonce = nonce ^ 0...seq
    let mut nonce = *base_nonce;
    let seq_be = seq.to_be_bytes();
    for i in 0..seq_be.len() {
 let nonce_idx = i + nonce.len() - seq_be.len();
        nonce[nonce_idx] ^= seq_be[i];
    }
    nonce
}
```

The initialization vectors used for AES-GCM can be observed by adding a print statement to the `AesGcm256::new()` function.

```
impl AesGcm256 {
    // errors are from mla/src/error.rs
    #[allow(clippy::unnecessary_wraps)]
    pub fn new(key: &Key, nonce: &Nonce, associated_data: &[u8]) -> Result<AesGcm256,
Error> {
        // Convert the nonce (96 bits) to the AES-GCM form
        let mut counter_block = [0u8; BLOCK_SIZE];
        counter_block[..12].copy_from_slice(nonce);
        counter_block[15] = 1;

        println!("AES-GCM counter block: {:02x?}", &counter_block);
```

The Initialization vectors used can then be observed by creating an archive with the **mlar create** command.

```
$ head -n 4096 /dev/random | cargo run --package mlar -- create --output archive.mla --
stdin-data -k tests/keys.mlapriv -p tests/keys_dest.mlapub | sort

    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.04s
     Running `/home/olivier/projets/MLA/target/debug/mlar create --output archive.mla
--stdin-data -k tests/keys.mlapriv -p tests/keys_dest.mlapub`

AES-GCM counter block: [a4, e6, 9d, b1, 91, 6d, ea, 56, b6, 59, b4, 41, 00, 00, 00, 01]
AES-GCM counter block: [ae, 81, 83, 90, 37, e0, 42, 76, 28, 08, 73, f4, 00, 00, 00, 01]
AES-GCM counter block: [ae, 81, 83, 90, 37, e0, 42, 76, 28, 08, 73, f5, 00, 00, 00, 01]
AES-GCM counter block: [ae, 81, 83, 90, 37, e0, 42, 76, 28, 08, 73, f6, 00, 00, 00, 01]
AES-GCM counter block: [ae, 81, 83, 90, 37, e0, 42, 76, 28, 08, 73, f7, 00, 00, 00, 01]
AES-GCM counter block: [ae, 81, 83, 90, 37, e0, 42, 76, 28, 08, 73, f8, 00, 00, 00, 01]
AES-GCM counter block: [ae, 81, 83, 90, 37, e0, 42, 76, 28, 08, 73, f9, 00, 00, 00, 01]
AES-GCM counter block: [ae, 81, 83, 90, 37, e0, 42, 76, 28, 08, 73, fa, 00, 00, 00, 01]
AES-GCM counter block: [ae, 81, 83, 90, 37, e0, 42, 76, 28, 08, 73, fb, 00, 00, 00, 01]
AES-GCM counter block: [ae, 81, 83, 90, 37, e0, 42, 76, 28, 08, 73, fc, 00, 00, 00, 01]
AES-GCM counter block: [ae, 81, 83, 90, 37, e0, 42, 76, 28, 08, 73, fd, 00, 00, 00, 01]
AES-GCM counter block: [ae, 81, 83, 90, 37, e0, 42, 76, 28, 08, 73, fe, 00, 00, 00, 01]
AES-GCM counter block: [ae, 81, 83, 90, 37, e0, 42, 76, 28, 08, 73, ff, 00, 00, 00, 01]
```

The CTR component on AES-GCM is limited to a 32-bit unsigned integer. When AES-GCM is used, the counter-block is incremented for each AES block (128 bits) encrypted.

If enough blocks are created, the CTR component will start to overflow on the base nonce. Given the base nonce sequence, it could lead to nonce reuse.

For instance with larger chunk size the following pattern may occur:

```
AES-GCM counter block: [a4, e6, 9d, b1, 91, 6d, ea, 56, b6, 59, b4, 41, 00, 00, 00, 01]
// used for key commitment encryption on chunk[0]
AES-GCM counter block: [a4, e6, 9d, b1, 91, 6d, ea, 56, b6, 59, b4, 41, 00, 00, 00, 02]
// used for chunk[0] block[0] encryption
[...]
AES-GCM counter block: [ae, 81, 83, 90, 37, e0, 42, 76, 28, 08, 73, f4, 00, 00, 00, 01]
// used for key commitment encryption on chunk[1]
AES-GCM counter block: [ae, 81, 83, 90, 37, e0, 42, 76, 28, 08, 73, f4, 00, 00, 00, 02]
// used for chunk[1] block[0] encryption
[...]
AES-GCM counter block: [ae, 81, 83, 90, 37, e0, 42, 76, 28, 08, 73, f4, ff, ff, ff, ff]
// used for chunk[1] block[2**32] encryption
AES-GCM counter block: [ae, 81, 83, 90, 37, e0, 42, 76, 28, 08, 73, f5, 00, 00, 00, 00]
// used for chunk[1] block[2**32 + 1] encryption, CTR overflow
AES-GCM counter block: [ae, 81, 83, 90, 37, e0, 42, 76, 28, 08, 73, f5, 00, 00, 00, 01]
// used for chunk[1] block[2**32 + 2] encryption, CTR overflow
[...]
// up to the next chunk
AES-GCM counter block: [ae, 81, 83, 90, 37, e0, 42, 76, 28, 08, 73, f5, 00, 00, 00, 01]
// used for key commitment encryption on chunk[2], nonce reuse on known plaintext
AES-GCM counter block: [ae, 81, 83, 90, 37, e0, 42, 76, 28, 08, 73, f5, 00, 00, 00, 02]
```

```
// used for key commitment encryption on chunk[2]
[...]
```

Risks

If the initialization vector is reused to encrypt data, it becomes possible to decrypt the data, and forge new messages. Moreover, it should be noted that key commitment blocks provide a known clear text for first blocks which may be reused.

Recommendations

Document the security consideration associated with the size of the chunks, to avoid any dangerous refactoring.

Note that the 64 gigabytes per AES-GCM invocation is documented in NIST Special Publication 800-38D and RustCrypto's implementation enforces this limit on single chunk decryption:

- https://github.com/RustCrypto/AEADs/blob/3021c464d3c2174d250e2b0e967283f9cfbd3cc0/aes-gcm/src/lib.rs#L112
- https://github.com/RustCrypto/AEADs/blob/3021c464d3c2174d250e2b0e967283f9cfbd3cc0/aes-gcm/src/lib.rs#L275

This limit could be enforced with a similar check on `self.bytes_encrypted` value at https://github.com/ANSSI-FR/MLA/blob/e57802fc945be013a585ec838994526b233ecbd5/mla/src/crypto/aesgcm/mod.rs#L85.

Bibliography

- https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-38d.pdf

V-04 Multiple integer overflow

Probability	Impact	Severity	Remediation
RARE	MINIMAL	REMARK	BASIC

Observations

The software performs a calculation that can produce an integer overflow or wraparound. In Rust, arithmetic operations are wrapped around by default on release profile.

Multiple case of unchecked arithmetic operations leading to an integer overflow were identified in the source code. Some examples are demonstrated below.

entry.rs:710

```
SeekFrom::Current(asked_seek_offset) => match self.state {
    ArchiveEntryDataReaderState::InEntryContent(remaining) => {
        let offset_from_start = self.offsets_and_sizes
            [..=self.current_offsets_and_sizes_index]
            .iter()
            .map(|p| p.1)
            .sum::<u64>()
```

entry:675

```
// look for block containing asked_seek_offset from end (.rev())
for (index, (offset, size)) in
    self.offsets_and_sizes.iter().copied().enumerate().rev()
{
    sum += size;
    if sum > offset_from_end {
        // return found info and index
        found = Some((offset, size, index));
        break;
    }
}
```

Risks

Integer overflows generally lead to undefined behavior and therefore crashes. In case of overflows involving loop index variables, the likelihood of infinite loops is also high.

While the auditors did not identify a scenario where an integer overflow had a security implication, using checked operations will ensure a correct and predictable behavior of the application.

Recommendations

An efficient way of identifying possible integer overflow in Rust is to use a Clippy linter such as arithmetic_side_effects. When the linter is activated, `cargo build` will list all identified integer overflow.

`https://rust-lang.github.io/rust-clippy/master/index.html#arithmetic_side_effects`

Perform input validation on any numeric input by ensuring that it is within the expected range. Enforce that the input meets both the minimum and maximum requirements for the expected range.

Use unsigned integers where possible. This makes it easier to perform sanity checks for integer overflows. If you must use signed integers, make sure that your range check includes minimum values as well as maximum values.

SYNACKTIV

+33 1 45 79 74 75

contact@synacktiv.com

5 boulevard Montmartre

75002 – PARIS

www.synacktiv.com