ASSURED

SECURITY CONSULTANTS

Report

Code review of GotaTun

Johanna Abrahamsson, Emilie Barse and Joachim Strömbergson

Project	Version	Date
MUL023	1	2026-02-17

ASSURED
SECURITY CONSULTANTS

REPORT

Project	**Version**	**Date**
MUL023	1	2026-02-17

Executive summary

Between 2026-01-19 and 2026-02-15 Assured Security Consultants performed a brief code audit of GotaTun, a user space implementation of the WireGuard® network tunnel protocol on behalf of Mullvad VPN AB. The audit was conducted by source code review as well as some testing.

The entire GotaTun implementation was in scope, with the exception of the command line interface (`gotatun-cli`) and the Mullvad specific DAITA code.

This report lists the security issues found, along with recommendations for remediation or mitigation. In our conclusions we discuss the issues and address apparent patterns in areas where security is lacking.

Observations were made with the following risk severity assessments (number of issues):

Critical 0 High 0 Medium 0 Low 2 Note 9 Good 1

Our recommendations can be summarized as follows:

- Ensure that GotaTun follows the WireGuard specification as closely as possible, in particular when it comes to padding and the generation of session identifiers
- Where GotaTun deviates from the WireGuard specification, or the behavior of the WireGuard kernel module, make sure to document this and provide the motivation for the deviation
- Replace the LFSR-and-counter index generation with random index generation similar to the WireGuard kernel module
- Rewrite and simplify the packet buffer implementation
- Consider a deeper code audit, or replacement, of select dependencies such as `tun` and `ring`

Assured would like to thank Joakim, Sebastian and Linus for their support during this audit. We are happy to answer any questions and provide further advice.

Contents

REPORT

Project	**Version**	**Date**
MUL023	1	2026-02-17

1 Introduction

1.1 Background

Assured AB (Assured) was contracted to conduct a brief code audit of GotaTun, a user space implementation of the WireGuard® network tunnel protocol on behalf of Mullvad VPN AB. The audit was conducted by source code review as well as some testing.

1.2 Constraints and disclaimer

This report contains a summary of the observations made during the project period. This report should not be considered as a complete list of all vulnerabilities, security flaws and/or misconfigurations.

1.3 Project period and staffing

Assured started the project on 2026-01-19 and finished on 2026-02-15.

This report was last reviewed on 2026-02-17.

Involved in the penetration testing were Assured consultants Johanna Abrahamsson, Emilie Barse and Joachim Strömbergson.

1.4 Assured Security Consultants

Assured Security Consultants (Assured AB[1]) was founded in 2015 with the mission to provide premier technical cybersecurity services as an independent consultancy, not affiliated with any vendor. Our team of experienced and dedicated cybersecurity specialists perform penetration testing, red team activities, secure design, embedded development, advisory, training and similar services.

We are committed to the security community as OWASP chapter leaders, event organizers and podcasters. We also take active part in security research projects into areas such as cryptography and automotive security.

[1]Assured AB, Org.nr. 556985-8276, registered in Sweden. www.assured.se

2 Scope and methodology

2.1 Scope

The scope of the test consisted of the GotaTun library. The scope also included mechanisms used by the library to include and use dependencies. One Dependency, the crate `tun`, was also included in the scope. Other dependencies were excluded from the scope. The command line interface part `gotatun-cli` was also excluded from the scope, as was the Mullvad specific DAITA code.

The audit was primarily carried out through source code review of the supplied repository `https://github.com/mullvad/gotatun`. The tag that was reviewed was `v0.2.0`, corresponding to the commit `67de7d50fa9d6b83e64c6cee279b856f874c7f49`.

Some verification of protocol parameters and packets was done by executing the code and establishing a Wireguard tunnel to a Mullvad VPN server.

2.2 Methodology

The audit was performed based on manual code analysis and execution in combination with tools used to audit the code. The following tools were used:

- Cargo Audit
- Cargo Clippy
- vscode with extensions rust-analyzer and CodeLLDB
- Wireshark

Where applicable, the WireGuard kernel module was used as reference implementation, in addition to the specification in the WireGuard whitepaper[2].

[2] https://www.wireguard.com/papers/wireguard.pdf

3 Observations

3.1 GOOD Nifty `size_must_be` sanity check

The file `gotatun/src/packet/util.rs` contains a nifty *const function* called `size_must_be`, which can be used to enforce size constraints on types during compile time. This is used throughout the code to make sure that the size of various network header structs (and even some whole packets) are correct.

This serves both to make sure that the structs behave as expected in all configurations and environments, as well as to reduce the risk of unintended effects of code changes. In a way they also serve to make the code easier to read and comprehend.

We recommend continuing to use the `size_must_be` function where it is applicable, and to, where possible, introduce similar mechanisms that makes the code more robust.

3.2 LOW LFSR used to generate peer identifiers

When establishing a WireGuard session, the specification calls for a 32-bit random number to be used for tying subsequent messages to this session, both when initiating a handshake and when replying to a handshake initiation (as described in section 5.4.2 and 5.4.3 in the WireGuard whitepaper). In GotaTun, this session identifier is generated through the use of a 24-bit Linear Feedback Shift Register (LFSR). The value of this 24-bit LFSR is then concatenated with an 8-bit counter to form the 32-bit number. According to comments in the GotaTun code "The index used is 24 bits for peer index, allowing for 16M active peers per server and 8 bits for cyclic session index". The concatenation is done in the function `inc_index` in `src/noise/handshake.rs` at line 472, the value of `next_index` will at first be initialised to be the output of the LFSR shifted 8 bits to the left (which is done in `src/noise/mod.rs` on line 85). The function that generates the pseudorandom sequence is `next` in `src/device/index_lfsr.rs` at line 33, this method is called in several places where a peer is added.

The reason for this deviation from the WireGuard specification is unclear. While it does not seem to weaken the protection of network tunnels, it could reveal information about the number of peers as well as the number of times handshakes have been exchanged with the peers to anyone who can eavesdrop on network traffic.

We recommend using a 32-bit random number as session identifier (in the kernel module referred to as `index`) in the same way that the kernel module and the wireguard-go implementation does, using a hashmap or similar data structure to ensure that each generated index is unique.

3.3 LOW FIXED Padding of payload not according to WireGuard specification

Verification note: This finding was fixed during the test time in git branch pad-data-payloads-to-16. The code now does padding up to a 16 byte boundary as described in the WireGuard specification. The device MTU is used for checking that the padded packet does not exceed the MTU.

The GotaTun code does not follow the WireGuard specification regarding padding of payload before encryption.

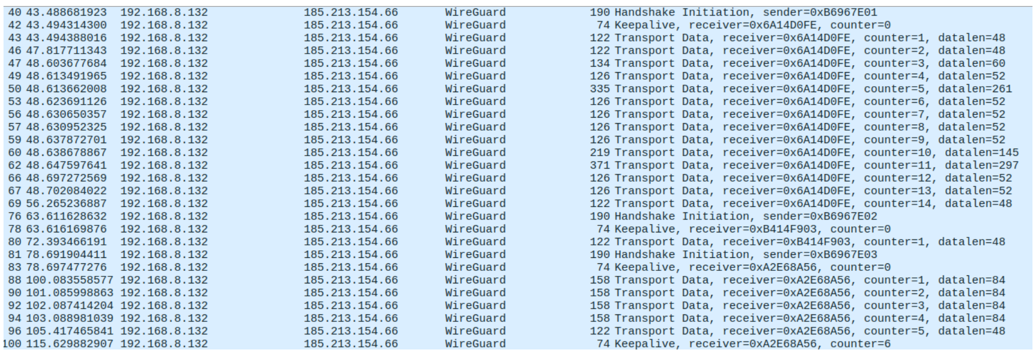
A TODO comment in the file `gotatun/src/noise/session.rs` states that specification requires padding to 16 bytes, but works fine without it. Example 1 shows a code listing of parts of the function `format_packet_data()` with the TODO comment. The comment probably refers to what is described in the WireGuard whitepaper[3], section 5.4.6. It is stated in the whitepaper that "The encapsulated packet itself is zero padded (without modifying the IP packet's length field) before encryption to complicate traffic analysis, though that zero padding should never increase the UDP packet size beyond the maximum transmission unit length."

Example 1: gotatun/src/noise/session.rs, format_packet_data()

```
        // TODO: spec requires padding to 16 bytes, but actually works fine without it
        let mut nonce = [0u8; 12];
        nonce[4..12].copy_from_slice(&sending_key_counter.to_le_bytes());
        data.encrypted_encapsulated_packet_mut()
            .copy_from_slice(&packet);
        self.sender
            .seal_in_place_separate_tag(
                Nonce::assume_unique_for_key(nonce),
                Aad::from(&[]),
                data.encrypted_encapsulated_packet_mut(),
            )
            .map(|tag| {
                data.tag_mut().copy_from_slice(tag.as_ref());
                packet.len() + AEAD_SIZE
            })
            .expect("encryption must succeed");
```

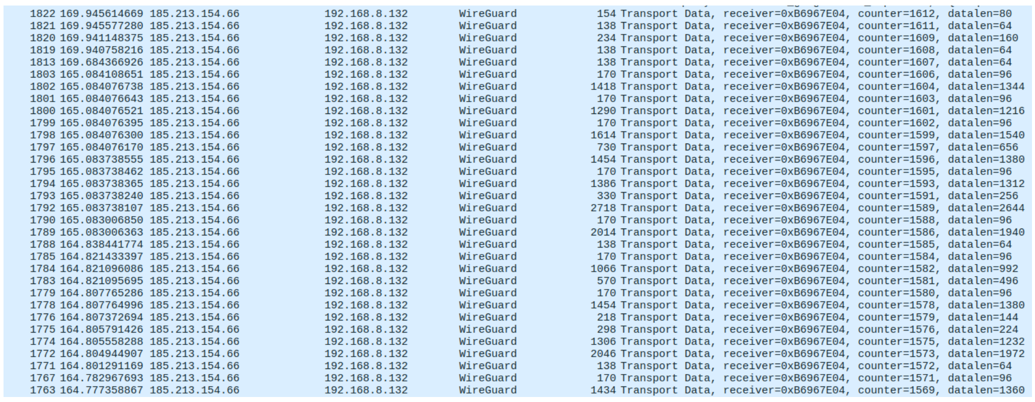
Figure 1 shows the encrypted WireGuard packets from the GotaTun client. The lengths of the packets are not padded to 16 bytes, and the length of the encrypted payload is exactly the same length as the original packet length. Figure 2 shows the lengths of the packets from the Mullvad WireGuard server which use padding to 16 bytes.

[3]https://www.wireguard.com/papers/wireguard.pdf



```
 40 43.488681923  192.168.8.132      185.213.154.66      WireGuard       190 Handshake Initiation, sender=0xB6967E01
 42 43.494314300  192.168.8.132      185.213.154.66      WireGuard        74 Keepalive, receiver=0x6A14D0FE, counter=0
 43 43.494388016  192.168.8.132      185.213.154.66      WireGuard       122 Transport Data, receiver=0x6A14D0FE, counter=1, datalen=48
 46 47.817711343  192.168.8.132      185.213.154.66      WireGuard       122 Transport Data, receiver=0x6A14D0FE, counter=2, datalen=48
 47 48.603677684  192.168.8.132      185.213.154.66      WireGuard       134 Transport Data, receiver=0x6A14D0FE, counter=3, datalen=60
 49 48.613491965  192.168.8.132      185.213.154.66      WireGuard       126 Transport Data, receiver=0x6A14D0FE, counter=4, datalen=52
 50 48.613662008  192.168.8.132      185.213.154.66      WireGuard       335 Transport Data, receiver=0x6A14D0FE, counter=5, datalen=261
 53 48.623691126  192.168.8.132      185.213.154.66      WireGuard       126 Transport Data, receiver=0x6A14D0FE, counter=6, datalen=52
 56 48.630650357  192.168.8.132      185.213.154.66      WireGuard       126 Transport Data, receiver=0x6A14D0FE, counter=7, datalen=52
 57 48.630952325  192.168.8.132      185.213.154.66      WireGuard       126 Transport Data, receiver=0x6A14D0FE, counter=8, datalen=52
 59 48.637872701  192.168.8.132      185.213.154.66      WireGuard       126 Transport Data, receiver=0x6A14D0FE, counter=9, datalen=52
 60 48.638678867  192.168.8.132      185.213.154.66      WireGuard       219 Transport Data, receiver=0x6A14D0FE, counter=10, datalen=145
 62 48.647597641  192.168.8.132      185.213.154.66      WireGuard       371 Transport Data, receiver=0x6A14D0FE, counter=11, datalen=297
 66 48.697272569  192.168.8.132      185.213.154.66      WireGuard       126 Transport Data, receiver=0x6A14D0FE, counter=12, datalen=52
 67 48.702084022  192.168.8.132      185.213.154.66      WireGuard       126 Transport Data, receiver=0x6A14D0FE, counter=13, datalen=52
 69 56.265236887  192.168.8.132      185.213.154.66      WireGuard       122 Transport Data, receiver=0x6A14D0FE, counter=14, datalen=48
 76 63.611628632  192.168.8.132      185.213.154.66      WireGuard       190 Handshake Initiation, sender=0xB6967E02
 78 63.616169876  192.168.8.132      185.213.154.66      WireGuard        74 Keepalive, receiver=0xB414F903, counter=0
 80 72.393466191  192.168.8.132      185.213.154.66      WireGuard       122 Transport Data, receiver=0xB414F903, counter=1, datalen=48
 81 78.691904411  192.168.8.132      185.213.154.66      WireGuard       190 Handshake Initiation, sender=0xB6967E03
 83 78.697477276  192.168.8.132      185.213.154.66      WireGuard        74 Keepalive, receiver=0xA2E68A56, counter=0
 88 100.083558577 192.168.8.132      185.213.154.66      WireGuard       158 Transport Data, receiver=0xA2E68A56, counter=1, datalen=84
 90 101.085998863 192.168.8.132      185.213.154.66      WireGuard       158 Transport Data, receiver=0xA2E68A56, counter=2, datalen=84
 92 102.087414204 192.168.8.132      185.213.154.66      WireGuard       158 Transport Data, receiver=0xA2E68A56, counter=3, datalen=84
 94 103.088981039 192.168.8.132      185.213.154.66      WireGuard       158 Transport Data, receiver=0xA2E68A56, counter=4, datalen=84
 96 105.417465841 192.168.8.132      185.213.154.66      WireGuard       122 Transport Data, receiver=0xA2E68A56, counter=5, datalen=48
100 115.629882907 192.168.8.132      185.213.154.66      WireGuard        74 Keepalive, receiver=0xA2E68A56, counter=6
```

Figure 1: Encrypted WireGuard traffic from the gotatun client



```
1822 169.945614669 185.213.154.66      192.168.8.132      WireGuard       154 Transport Data, receiver=0xB6967E04, counter=1612, datalen=80
1821 169.945577280 185.213.154.66      192.168.8.132      WireGuard       138 Transport Data, receiver=0xB6967E04, counter=1611, datalen=64
1820 169.941148375 185.213.154.66      192.168.8.132      WireGuard       234 Transport Data, receiver=0xB6967E04, counter=1609, datalen=160
1819 169.940758216 185.213.154.66      192.168.8.132      WireGuard       138 Transport Data, receiver=0xB6967E04, counter=1608, datalen=64
1813 169.684366926 185.213.154.66      192.168.8.132      WireGuard       138 Transport Data, receiver=0xB6967E04, counter=1607, datalen=64
1803 165.084108651 185.213.154.66      192.168.8.132      WireGuard       170 Transport Data, receiver=0xB6967E04, counter=1606, datalen=96
1802 165.084076738 185.213.154.66      192.168.8.132      WireGuard      1418 Transport Data, receiver=0xB6967E04, counter=1604, datalen=1344
1801 165.084076643 185.213.154.66      192.168.8.132      WireGuard       170 Transport Data, receiver=0xB6967E04, counter=1603, datalen=96
1800 165.084076521 185.213.154.66      192.168.8.132      WireGuard      1290 Transport Data, receiver=0xB6967E04, counter=1601, datalen=1216
1799 165.084076395 185.213.154.66      192.168.8.132      WireGuard       170 Transport Data, receiver=0xB6967E04, counter=1602, datalen=96
1798 165.084076300 185.213.154.66      192.168.8.132      WireGuard      1614 Transport Data, receiver=0xB6967E04, counter=1599, datalen=1540
1797 165.084076170 185.213.154.66      192.168.8.132      WireGuard       730 Transport Data, receiver=0xB6967E04, counter=1597, datalen=656
1796 165.083738555 185.213.154.66      192.168.8.132      WireGuard      1454 Transport Data, receiver=0xB6967E04, counter=1596, datalen=1380
1795 165.083738462 185.213.154.66      192.168.8.132      WireGuard       170 Transport Data, receiver=0xB6967E04, counter=1595, datalen=96
1794 165.083738365 185.213.154.66      192.168.8.132      WireGuard      1386 Transport Data, receiver=0xB6967E04, counter=1593, datalen=1312
1793 165.083738240 185.213.154.66      192.168.8.132      WireGuard       330 Transport Data, receiver=0xB6967E04, counter=1591, datalen=256
1792 165.083738107 185.213.154.66      192.168.8.132      WireGuard      2718 Transport Data, receiver=0xB6967E04, counter=1589, datalen=2644
1790 165.083006850 185.213.154.66      192.168.8.132      WireGuard       170 Transport Data, receiver=0xB6967E04, counter=1588, datalen=96
1789 165.083006363 185.213.154.66      192.168.8.132      WireGuard      2014 Transport Data, receiver=0xB6967E04, counter=1586, datalen=1940
1788 164.838441774 185.213.154.66      192.168.8.132      WireGuard       138 Transport Data, receiver=0xB6967E04, counter=1585, datalen=64
1785 164.821433397 185.213.154.66      192.168.8.132      WireGuard       170 Transport Data, receiver=0xB6967E04, counter=1584, datalen=96
1784 164.821096086 185.213.154.66      192.168.8.132      WireGuard      1066 Transport Data, receiver=0xB6967E04, counter=1582, datalen=992
1783 164.821095695 185.213.154.66      192.168.8.132      WireGuard       570 Transport Data, receiver=0xB6967E04, counter=1581, datalen=496
1779 164.807765286 185.213.154.66      192.168.8.132      WireGuard       170 Transport Data, receiver=0xB6967E04, counter=1580, datalen=96
1778 164.807764996 185.213.154.66      192.168.8.132      WireGuard      1454 Transport Data, receiver=0xB6967E04, counter=1578, datalen=1380
1776 164.807372694 185.213.154.66      192.168.8.132      WireGuard       218 Transport Data, receiver=0xB6967E04, counter=1579, datalen=144
1775 164.805791426 185.213.154.66      192.168.8.132      WireGuard       298 Transport Data, receiver=0xB6967E04, counter=1576, datalen=224
1774 164.805558288 185.213.154.66      192.168.8.132      WireGuard      1306 Transport Data, receiver=0xB6967E04, counter=1575, datalen=1232
1772 164.804944907 185.213.154.66      192.168.8.132      WireGuard      2046 Transport Data, receiver=0xB6967E04, counter=1573, datalen=1972
1771 164.801291169 185.213.154.66      192.168.8.132      WireGuard       138 Transport Data, receiver=0xB6967E04, counter=1572, datalen=64
1767 164.782967693 185.213.154.66      192.168.8.132      WireGuard       170 Transport Data, receiver=0xB6967E04, counter=1571, datalen=96
1763 164.777358867 185.213.154.66      192.168.8.132      WireGuard      1434 Transport Data, receiver=0xB6967E04, counter=1569, datalen=1360
```

Figure 2: Encrypted WireGuard traffic from a Mullvad WireGuard server

This comment and code was inherited from the fork of BoringTun, and is still present in the BoringTun code too.

This is not a major cryptographic issue, though the reason for adding the padding in the WireGuard whitepaper seems valid.

We recommend adding the padding to follow the specification.

3.4 NOTE Endpoint address only updated on handshake initiation

It seems that the only place where the IP address of an endpoint is updated is when receiving a handshake initiation message, in the method `handle_incoming` in `src/device/mod.rs` on line 655. This is not adhering to the WireGuard specification, which states that "[...] WireGuard receives a correctly authenticated packet from a peer, it will use the outer external source IP address for determining the endpoint". In BoringTun, from which GotaTun is forked, IP addresses of endpoints are updated as per the specification but in GotaTun this method call (`set_endpoint(addr)`) has been moved to occur only on handshake initiation messages.

The reason that WireGuard is specified to update IP addresses of peers in this way is to support roaming, where a WireGuard tunnel can persist even if one peer changes their external IP address, either through switching networks or through routing changes.

We recommend following the WireGuard specification in this regard, although GotaTun seems primarily used in a setting where the endpoint address is statically configured, implementing this part of the protocol properly would open up for GotaTun to be used in the server like setting of listening for connections from any address as well.

3.5 NOTE Sensitive data not protected in memory

Sensitive data such as encryption keys, both static and ephemeral, are stored in memory with no apparent protection. While there are no methods to completely protect program memory against all kinds of readout methods, there are some preventable ways that memory contents might unintentionally end up on disk.

One such way, that have been exploited in the past, is when sensitive memory contents are included in a core dump. While Rust normally does not create a core dump, there may be situations where it does. To limit the information exposure of core dumps on a UNIX system, the system call `setrlimit` may be used to set the limit of `RLIMIT_CORE` to 0 which disables the creation of a core dump. There is also the syscall `madvise` that can set the flag `MADV_DONTDUMP` can be applied to a memory region to exclude it from core dumps. On Windows, the system call `WerRegisterExcludedMemoryBlock` can likewise be used to ask the operating system to exclude a memory region from any crash dump.

Another way that sensitive memory contents may be written to storage is through memory swapping. On a UNIX system the system call `mlock` may be used to prevent swapping for a given memory region. The corresponding system call on Windows is `VirtualLock`.

While it is possible to reduce the risks of inadvertent disclosure of sensitive memory contents through the above mentioned means, it is sometimes unfeasible. Another way to protect sensitive memory contents is to encrypt the data in memory. In Windows there is a system call, `CryptProtectMemory` that purports to do this, although in some cases the encryption method in use is 3DES and the key generation is possibly vulnerable to some attacks. Many crypto libraries also include some sort of memory encryption method.

We recommend looking into where memory protection mechanisms may be used, preferably through the use of reputable crypto libraries rather than by own implementation. For inspiration, secure coding standards such as SEI CERT C[4] may be consulted.

[4] `https://wiki.sei.cmu.edu/confluence/spaces/c/pages/87152115/MEM06-C.+Ensure+that+sensitive+data+is+not+written+out+to+disk`

3.6 NOTE FIXED Crypto dependencies can be updated

Verification note: This finding was fixed during the test time in git commits 98dca59 and dd2066e. The version of `x25519-dalek` that is specified in `gotatun/Cargo.toml` is now 2.0.1 and the version of `chacha20poly1305` is now 0.10.1.

The `gotatun/Cargo.toml` file specifies versions for the cargo dependency crates `x25519-dalek` and `chacha20poly1305` which can be updated.

As seen in the listing in Example 2, the `gotatun/Cargo.toml` file specifies that the version of `x25519-dalek` installed will be at least 2.0.0 and for `chacha20poly1305` at least version 0.10.0-pre.1.

Versions 2.0.1 respectively 0.10.1 are available and preferred. These versions are in fact already used in the Cargo.lock file provided with the GotaTun repo.

Example 2: gotatun/Cargo.toml

```
[dependencies]
base64 = "0.13"
hex = "0.4"
libc = "0.2"
parking_lot = "0.12"
ipnetwork = "0.21"
ip_network = "0.4.1"
ip_network_table = "0.2.0"
ring = "0.17"
x25519-dalek = { version = "2.0.0", features = [
    "reusable_secrets",
    "static_secrets",
] }
rand_core = { version = "0.6.4", features = ["getrandom"] }
chacha20poly1305 = "0.10.0-pre.1"
aead = "0.5.0-pre.2"
blake2 = "0.10"
hmac = "0.12"
...
```

As a note, the `cloundflare/boringtun` repo has updated the version of `x25519-dalek` to version 2.0.1, but `chacha20poly1305` version 0.10.0-pre.1 is still used. `x25519-dalek` was updated in the BoringTun repo from 2.0.0-pre3 because it used a vulnerable dependency version of `curve25519-dalek` as described in BoringTun issue #451 [5].

We recommend updating the version of `x25519-dalek` to 2.0.1, and `chacha20poly1305` to 0.10.1 in the `gotatun/Cargo.toml` file.

[5] https://github.com/cloudflare/boringtun/issues/451

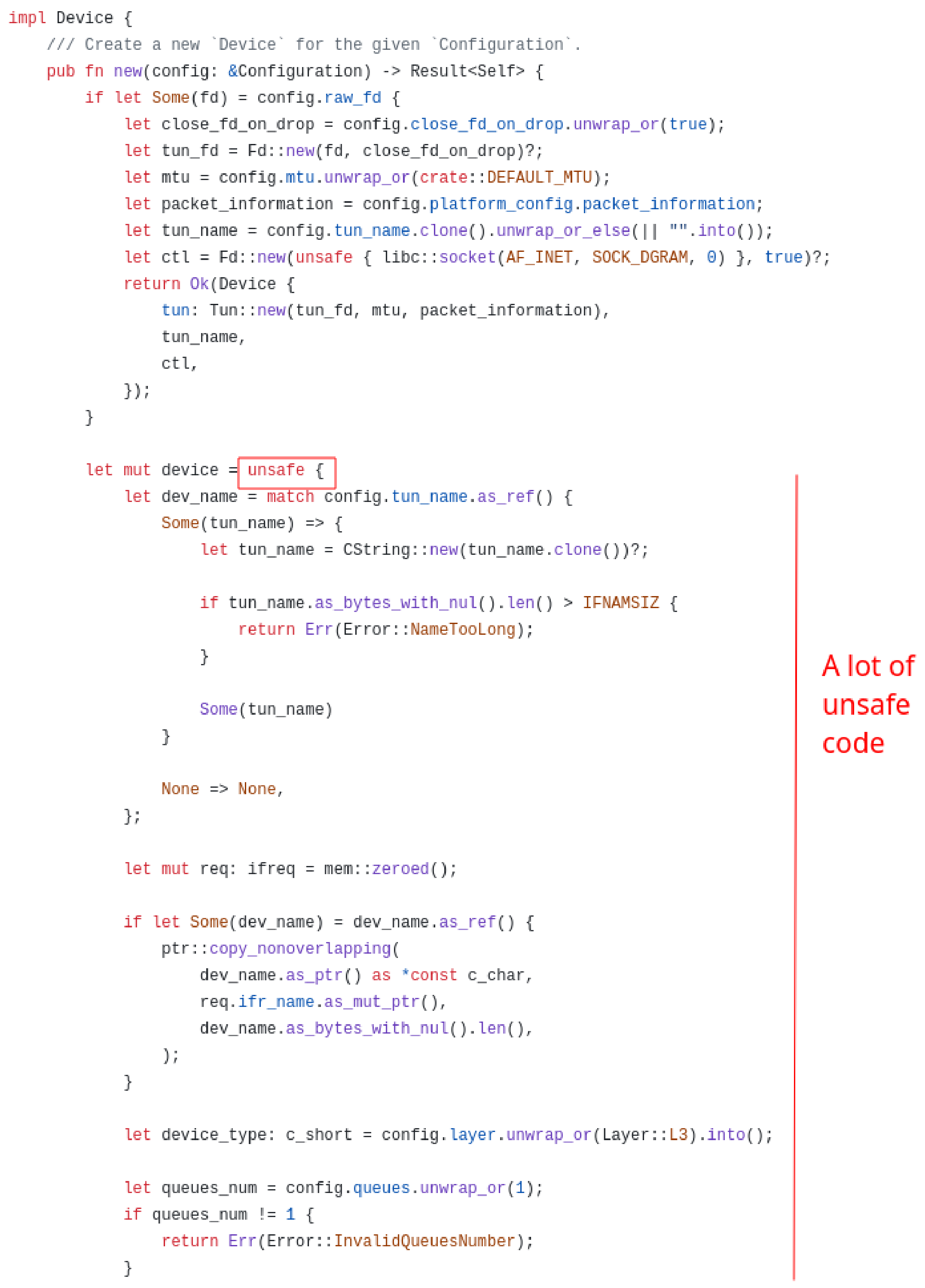
3.7 NOTE Extensive usage of unsafe in crate tun

The dependency crate `tun v0.8.5` contains many long blocks annotated with the `unsafe` keyword and no comments on why `unsafe` is used and what is required to make it safe. Good security practice for using `unsafe` in rust is to minimize the blocks and comment on what is needed to make the code safe.

There are 74 `unsafe` blocks in the tun crate, and many occurrences contains multiple statements. It is unlikely that all `unsafe` blocks are minimal in size.

Figure 3 shows one example of a long `unsafe` block in the `impl Device` block in the file `rust-tun/src/platform/linux/device.rs`.

We recommend going through the tun crate and help the developer minimize and comment the `unsafe` blocks.



```
impl Device {
    /// Create a new `Device` for the given `Configuration`.
    pub fn new(config: &Configuration) -> Result<Self> {
        if let Some(fd) = config.raw_fd {
            let close_fd_on_drop = config.close_fd_on_drop.unwrap_or(true);
            let tun_fd = Fd::new(fd, close_fd_on_drop)?;
            let mtu = config.mtu.unwrap_or(crate::DEFAULT_MTU);
            let packet_information = config.platform_config.packet_information;
            let tun_name = config.tun_name.clone().unwrap_or_else(|| "".into());
            let ctl = Fd::new(unsafe { libc::socket(AF_INET, SOCK_DGRAM, 0) }, true)?;
            return Ok(Device {
                tun: Tun::new(tun_fd, mtu, packet_information),
                tun_name,
                ctl,
            });
        }

        let mut device = unsafe {
            let dev_name = match config.tun_name.as_ref() {
                Some(tun_name) => {
                    let tun_name = CString::new(tun_name.clone())?;

                    if tun_name.as_bytes_with_nul().len() > IFNAMSIZ {
                        return Err(Error::NameTooLong);
                    }

                    Some(tun_name)
                }

                None => None,
            };

            let mut req: ifreq = mem::zeroed();

            if let Some(dev_name) = dev_name.as_ref() {
                ptr::copy_nonoverlapping(
                    dev_name.as_ptr() as *const c_char,
                    req.ifr_name.as_mut_ptr(),
                    dev_name.as_bytes_with_nul().len(),
                );
            }

            let device_type: c_short = config.layer.unwrap_or(Layer::L3).into();

            let queues_num = config.queues.unwrap_or(1);
            if queues_num != 1 {
                return Err(Error::InvalidQueuesNumber);
            }
```

Figure 3: Example of long unsafe block in crate tun

3.8 NOTE Complex non-working pre-allocation of packet buffers

The file `gotatun/src/packet/pool.rs` contains a `PacketBufPool`, which is supposed to be a pool of buffers residing in a contiguous block of memory. To achieve this, a large buffer is allocated when instantiating a `PacketBufPool` and this buffer is then split into a number of smaller buffers with a length of 4096 bytes. However, due to a small error it seems that only the very last buffer will be used.

The method for acquiring a buffer from the pool contains a fallback to new allocation of memory through `BytesMut::zeroed(len)`, which on release will become a part of the pool of buffers (unless the pool is at capacity, in which case the memory will be freed upon release). In the end, the `PacketBufPool` will work as a pool of buffers, albeit not with a contiguous block of memory as backing.

Even though the implementation of `PacketBufPool` does not work as intended and might have a slight negative effect on performance, it likely does not impact the security of GotaTun negatively. Still, the complexity of the implementation, and the use of the `unsafe` keyword at one point to manipulate the length value of the buffer, increases the risk of vulnerabilities either present now but missed or introduced unknowingly in future development.

One of the design principles of WireGuard is to not be dependent on dynamic allocation of memory for handling received packets, but rather use static allocations. Because of this, there should be little to no need for GotaTun to dynamically allocate memory for handling received packets.

We recommend rewriting the `PacketBufPool`, trying to keep the implementation as simple as possible and preferably avoiding use of the `unsafe` keyword. We recommend removing the fallback to dynamic memory allocation, since it serves mostly to mask the failure of the pre-allocation.

3.9 NOTE Usage of the crate ring

GotaTun depends, due to it's origin as a fork of BoringTun, on the crate `ring`, which is described by the author as "An experiment". It seems to be maintained mainly by one person, Brian Smith.

The crate `ring` is, according to it's description, mostly built on code from BoringSSL. It does not *use* BoringSSL as a library, but rather methods in C and assembly have been ported from BoringSSL to the Rust crate `ring`. This, along with seemingly depending on one sole maintainer, comes with the risk of vulnerabilities discovered in BoringSSL being applicable also to `ring` and possibly being publicly disclosed before any fixes have been ported to `ring`.

The library BoringSSL itself comes with the following disclaimer.

> BoringSSL README.MD
>
> *BoringSSL is a fork of OpenSSL that is designed to meet Google's needs.*
>
> *Although BoringSSL is an open source project, it is not intended for general use, as OpenSSL is. We don't recommend that third parties depend upon it. Doing so is likely to be frustrating because there are no guarantees of API or ABI stability.*

We recommend considering replacing the crate `ring`. If the crate is not to be replaced, we recommend performing an audit of the crate, as well as establishing a process for monitoring publicly disclosed vulnerabilities in BoringSSL to ascertain their potential impact on `ring` and GotaTun.

3.10 NOTE Simultaneous decryption and authentication

According to the Cryptographic Doom Principle[6], it is advisable to perform authentication (as in validation of an authentication tag or MAC) of any received data *before* performing other operations such as decryption.

GotaTun makes use of the crate `ring` and it's method `open_in_place`, which performs in-place decryption and authentication in one method call. By analyzing the code we can see that it performs decryption and calculation of the correct authentication tag simultaneously, after which it compares the calculated authentication tag with the received tag. If they don't match, the newly decrypted plaintext is overwritten with zeroes and an error is returned. The method `open_in_place` does not follow the Cryptographic Doom Principle, but it does make sure that there is no way to mistakenly use plaintext originating from data with an incorrect authentication tag.

It should be noted that the WireGuard kernel module also does not follow the Cryptographic Doom Principle, opting instead to perform the decryption in place in a similar manner to `ring`.

We recommend, mainly for reasons described in observation 3.9, to consider replacing the `ring` crate. When looking for a replacement, adherence to common best practices such as authentication before decryption may be one aspect to consider.

[6]Moxie Marlinspike "The Cryptographic Doom Principle" `https://moxie.org/2011/12/13/the-cryptographic-doom-principle.html`

3.11 NOTE About the validation of checksums

The file `gotatun/src/packet/mod.rs` contain three comments that begin with "TODO: validate checksum", where the last instance actually has a question mark at the end. These seem to indicate a need to validate the checksums of IP packets arriving through the tunnel.

The WireGuard kernel module, which could be regarded as the reference implementation, chooses to forgo checksum validation with the following reasoning.

Comment in drivers/net/wireguard/receive.c

We've already verified the Poly1305 auth tag, which means this packet was not modified in transit. We can therefore tell the networking stack that all checksums of every layer of encapsulation have already been checked "by the hardware" and therefore is unnecessary to check again in software.

We recommend taking the same stance as the WireGuard kernel module, removing the TODO comments and possibly, for clarity, leaving a comment about why checksum validation is not necessary.

3.12 NOTE FIXED Vulnerability in bytes (CVE-2026-25541)

Verification note: This finding was fixed during the test time in git commit c594f34ecaee647279e8c206aecd2411343f212c. The version of `bytes` that is specified in `gotatun/Cargo.toml` is now 1.11.1.

Towards the end of the audit a vulnerability in the crate `bytes` was published with the CVE number 2026-25541, also referenced as RUSTSEC-2026-0007. The vulnerability affect versions of `bytes` from 1.2.1 up to and including the version in use in GotaTun, 1.11.0.

While we do recommend updating `bytes` to version 1.11.1 where the vulnerability has been fixed, a brief analysis found no instances where this vulnerability could be triggered or exploited within GotaTun.

4 Conclusions and recommendations

Based on our code review, GotaTun has no major vulnerabilities. However, the parts where it deviates from the WireGuard specification (as described in Observations 3.2 and 3.3) are cause for concern. Furthermore, the code contains some instances of "TODO" comments which should be addressed.

Some of the dependencies seem to have single maintainers, or are maintained by a small group of individuals. This increases the risk of vulnerabilities present now or introduced in the future since projects with one or only a few developers typically have less review of the source code both before and after release. For some of these dependencies, it may be possible to find replacements that are more actively maintained by an organization or a larger group of people, whereas for some dependencies it may be prudent for Mullvad to take a more active part in further development and maintenance, including performing or commissioning code audits.

Our recommendations can be summarized as follows:

- Ensure that GotaTun follows the WireGuard specification as closely as possible, in particular when it comes to padding and the generation of session identifiers
- Where GotaTun deviates from the WireGuard specification, or the behavior of the WireGuard kernel module, make sure to document this and provide the motivation for the deviation
- Replace the LFSR-and-counter index generation with random index generation similar to the WireGuard kernel module
- Rewrite and simplify the packet buffer implementation
- Consider a deeper code audit, or replacement, of select dependencies such as `tun` and `ring`