Security Analysis of WireGuard

MIT 6.857 Final Project

Andrew He Baula Xu Jerry Wu

Spring 2018

1 Introduction

Virtual private networks (VPNs) are a relatively common networking tool that allows two networks to securely and privately connect to each other via an untrusted internet connection. These have various applications, including securely accessing a corporate network or evading censorship with a secure tunnel. The two most popular open source VPNs available today are OpenVPN and StrongSwan, which depend on OpenSSL or IPSec for security of IP packets. These systems are notoriously hard to configure and use securely; a common OpenVPN configuration requires the creation of a Certificate Authority, which must be handled with extreme care to preserve security.

WireGuard is a new, modern VPN [1]. It claims to be easy to use, cryptographically sound, and performant. It is purposely implemented in few lines of code ($\sim$ 4000 lines) in order to be easily auditable for security vulnerabilities. Single individuals should be able to comprehensively review WireGuard. In addition, its conservative choices have been reviewed by cryptographers. WireGuard's interface is basic and was designed with simplicity in mind; a simple SSH-style public/private keypair provides authentication, and WireGuard clients are simple to connect to each other. It is suitable for all types of devices, from smartphones to backbone routers. WireGuard has much less complexity than traditional solutions, and cuts out the intermediate IPsec/SSL-based encryption layers to ensure simplicity of the entire system.

We performed a partial security audit of WireGuard, focusing on the unverified portions of WireGuard, particularly the reconnection and session-management systems. We found some implementation errors related to these systems, and provided some recommendations for WireGuard to simplify the system.

We will begin by presenting a brief overview of the internals of WireGuard, focusing on the systems we analyzed. We will then discuss the steps we took to audit WireGuard, as well as the results we found.

2 Technical Background and Prior Work

In this section, we will introduce WireGuard's various components.

2.1 WireGuard High-Level User Interface

WireGuard is structured as a Linux kernel module, although some user-space implementations are also being developed. WireGuard exposes itself as a virtual network interface, similar to a Ethernet card or a wireless card. This allows the user to take advantage of standard IP routing configuration and tables to choose which packets to pass through the VPN. IP packets sent to the WireGuard interface are encrypted and then forwarded to the correct underlying wireless interface.

WireGuard instances connect in pairs of "peers" to form bidirectional channels. A single WireGuard instance can have multiple peers, allowing VPN forwarding to various other servers. WireGuard peers are identified simply by their static (ECDH) public key, and only one peer needs to know the IP address of the other; WireGuard infers peer addresses using the last successfully authenticated packet. This allows clients to roam freely between internet connections, similar to Mosh [1].

Overall, this results in a protocol which has a very simple interface and a simple outwards-facing security model (you only have to protect your static private key). The one-to-one peering scheme allows us to reason about security and performance on a single peer at a time, simplifying analysis. For the remainder of the paper, we will only consider a single WireGuard peer.

2.2 WireGuard Handshake and Encryption Protocol

The WireGuard protocol uses short-lived sessions with ephemeral keys in order to ensure perfect forward secrecy. Each session lasts for at most 3 minutes and at most $2^{64} - 2^4 - 1$ data packets transmissions. We'll describe the protocol for a single session in this subsection, and then discuss the session-management logic in the following subsection.

Each session begins with a handshake based in the Noise framework [3]. The handshake consists of only a single round trip: a handshake initiation and a handshake response. We will omit the details of the handshake, but they essentially run two instances of the Elliptic Curve Diffie-Hellman key exchange (authenticated using the static public/private keys) to derive a sending ephemeral symmetric key and a receiving ephemeral symmetric key. These symmetric keys are used to encrypt and authenticate data messages in the session.

There is a security caveat to this protocol: the handshake responder cannot assume the connection is authentic until they have received at least one valid data packet; otherwise, they are vulnerable to key-compromise impersonation (KCI), in which an attacker can set up a session with the responder as an arbitrary user given the responder's private key. In other words, the handshake initiation packet itself is not sufficient proof of authenticity, only the first data packet is. As a corollary, a WireGuard initiator always sends at least one possibly-empty packet immediately after a session is created.

This protocol has a small extension to allow the use of "cookie" messages for rate-limiting. If either peer is under load, they may respond to an incoming

handshake message with a rotating MAC-based cookie instead of the intended response. The sender should then (after some timeout) resend their message with the cookie attached. The cookie provides a short, computationally efficient way to verify proof of IP ownership, and a way to rate-limit incoming connections without performing Elliptic Curve Diffie-Hellman computations to verify authenticity. We did not test this protocol directly, but we do note that it may cause the network to take multiple round trips to establish a handshake, which can affect session management.

2.3 WireGuard Session Management and Timers

We now introduce WireGuard's session management system, which controls how and when session handshakes should be reinitiated. We codify this in a simple set of rules.

1. **Sessions expire.** As stated above, a session is invalid after 3 minutes or after $2^{64} - 2^4 - 1$ packets are transmitted. Packets received from invalid sessions are silently dropped.

2. **Data to be sent triggers sessions.** When a data packet is queued to be sent, if no valid session is present, begin a session handshake. When a session is established, send all queued packets (or an empty packet as the initiator to confirm the session).

3. **Rekey rate limit.** Handshake initiations are *never* sent twice within 5 seconds, to prevent overload. This limit is only enforced at the sender, and includes reinitiations due to cookies. Thus, establishing a session while one server uses cookies takes at least 5 seconds.

4. **Handshake retransmission.** Handshake initiations are retried after $5 +$ jitter seconds if no session is established, either by receiving a valid handshake response as the initiator or by receiving the first data packet as the responder. Here, jitter is a randomized jitter (up to 333 ms) to reduce the chance of repeated handshake collisions. This is limited to 18 retries (90 seconds), after which the server gives up and drops all queued data packets.

5. **Keepalives.** Both servers, after receiving a non-empty data packet, guarantee that they will send a packet back within 10 seconds (the keepalive time). This response can either be a real data packet generated by an application, or, if the 10 second timer expires, may be an empty "keepalive" packet (which is still authenticated and encrypted).

   Conversely, if a peer does not receive a response after sending a non-empty data packet for more than the keepalive time plus the rekey timeout (15 seconds total), then it assumes the connection is broken and reinitiates a handshake to reestablish a session.

6. **Opportunistic rekeying.** After a packet is sent and at least $2^{64} - 2^{16} - 1$ packets have been sent already, the sender preemptively initiates a handshake to create a new session.

   Similarly, if any packet is sent by the initiator of the current session and the session is at least 2 minutes old, then the sender preemptively initiates a new handshake. This rule is limited to sends by the current initiator so as to prevent both peers from simultaneously retrying the handshake. Note that receives by the initiator will still trigger a reinitiation due to the keepalive.

   This actually creates an edge case: if the session is within 15 seconds of the 3-minute session timeout, the initiator will send the handshake immediately on (non-empty) data receive rather than waiting for the queued keepalive send, so that the session does not time out.

7. **Session keys storage.** At most two session keys are stored at any time. One key is always the current, latest confirmed session. The second is either the next unconfirmed key (as the receiver in an ongoing handshake), or is the previous session key (to allow receipt of out-of-order messages).

8. **Clearing data.** After 9 minutes (3 times the session timeout), if no new sessions are created or initiated, all keys are zeroed and erased.

9. **Optional persistent keepalive.** WireGuard does not normally establish sessions if no data is being sent. Users can optionally enable persistent keepalive, which periodically sends a keepalive packet regardless of data in order to keep the tunnel active at all times. This may be necessary to keep a NAT session alive, or for various other external reasons.

2.4 Prior Work - Formal Verification

The WireGuard handshake protocol has undergone rigorous formal verification of desired properties using the Tamarin proof system [2]. Many of the cryptographic primitive implementations have also been formally verified as correct. The remaining implementations have been carefully fuzzed against the verified implementations to ensure correctness.

2.5 Our approach

After consulting with the WireGuard authors, we learned that little work has been undertaken to inspect or verify the session management state. Thus, we have focused our analysis on this unverified portion of the protocol.

For this, we used two approaches: one is randomized fuzzing under adverse network conditions, and the other was careful analysis of the session management as a state machine, keeping care to document all transitions that could occur.

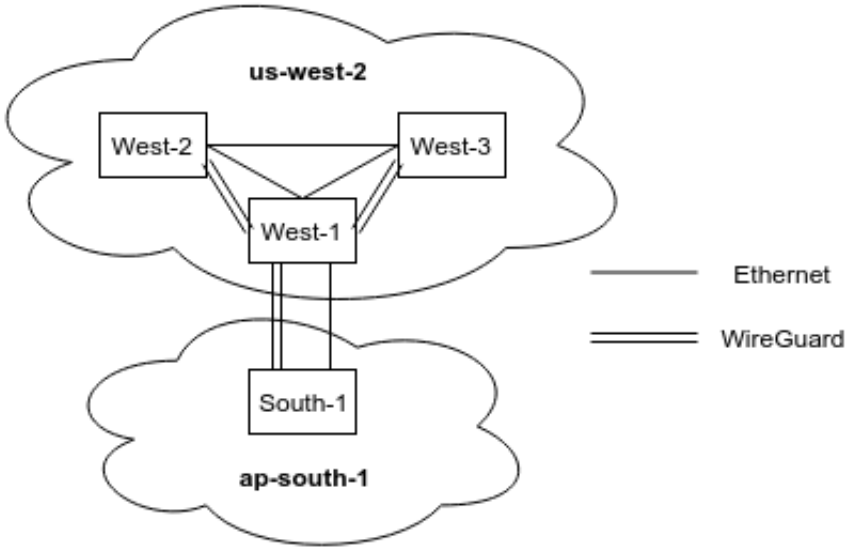

Figure 1: Network Setup. We have three WireGuard nodes in the us-west-2 AWS region in Oregon and one in the ap-south-1 region in Mumbai. Single edges denote connections over regular Ethernet, and double edges denote a WireGuard peering relationship. The ping time between servers in the us-west-1 region is negligible, and the ping time between West-1 and South-1 is about 212ms.

3 Approach 1: Fuzzing

3.1 Audit Methodology

In order to fuzz WireGuard, we set up four real WireGuard nodes in AWS, compiled in debug mode so we could access logs. In order to test a variety of network conditions, we set up three WireGuard nodes in the same region, us-west-2 (in Oregon), as well as one in a faraway region, ap-south-1 (in Mumbai). See Figure 1 for the network connection setup and the server names. We performed fuzzing by sending packets between these WireGuard nodes, both between nodes in the same region, in which the connection is very fast, and between nodes in two faraway regions, in which the connection is somewhat slower (around 212ms ping time). In both of these cases, we sent packets between the two WireGuard nodes and verified that the pings were able to go through without substantial latency or any packet drop rate.

However, in order to test the timer interface, we decided the most fruitful fuzzing would be under very adverse network conditions. We used `netem` to add an artificial 2s latency and 80% packet drop rate for outgoing packets sent by West-1 to West-2 and West-3. We performed a test in which the West-2 server

sent ping messages to South-1, which were routed first over the WireGuard connection with West-1, then over the Ethernet connection to South-1. Observe that the added latency and packet drops are only observed on the return trip, as West-1 forwards South-1's response back to West-2, as well as in handshake or other messages sent from West-1 to West-2. A total of 2974 ping messages were sent. We observed the performance on the pings, both in terms of ping time and packet losses, and also read through the WireGuard logs to ensure WireGuard was behaving as we expected it to.

3.2 Results and Analysis

Overall, throughout the fuzzing process, we found that WireGuard achieved its stated goals in being simple, easy to use, performant, and robust. When setting up WireGuard nodes for fuzzing, we (who had never manually set up a VPN connection before) found the online documentation a little confusing, but altogether the process was simple and straightforward, especially after we had finished setting up the first node and needed to repeat the process for additional nodes. In the tests without artificially imposed latency and packet drops, we observed that WireGuard performed as expected; it did not noticeably increase the ping time, nor did it cause any packets to be dropped.

In the test with between West-2 and South-1 with artificial latency and packet drops, we observed that the packet drop rate was 80% and the ping time was typically very close to 2212ms, which is the total RTT between West-2 and South-1 without WireGuard. However, in one case, the ping time was increased to 4s, and in another case, the ping time was increased to 13s. Additionally, the pings immediately after these packets were also delayed by progressively smaller amounts of time. This demonstrates that in these two situations, WireGuard was unable to establish a new session due to repeated handshake failures before the current session expired; it correctly queued up the unsendable packets and sent them all at once when the WireGuard connection was re-established. In the remainder of the cases, WireGuard was able to re-establish sessions because sessions are re-established 60 seconds in advance of the next session expiring, which is sufficient time under most circumstances. We consider this result to be good performance under these highly adverse network conditions.

However, fuzzing did reveal a bug in the timer system implementation. When reading through the WireGuard logs, we discovered that in several instances, the logs indicated that after sending a handshake initiation, WireGuard had received no response within five seconds and needed to send another handshake initiation. However, no handshake initiation was actually sent, despite the claim that retries should occur up to 18 times. Upon closer inspection, WireGuard had tried to retry the handshake initiation in slightly less than 5 seconds after the previous initiation, but handshake initiations are rate-limited to occur at most once per 5 seconds, which prevented the handshake from being sent and cancelled the whole retry process. See Figure 2 for an example log of this occurring.

After examining the code, we determined this was due to the rounding behav-

ior of the randomized retransmit timer: when `timer_retransmit_handshake` is set, the timeout is rounded by the `slack_time` function, which can actually cause the rounded time to be slightly less than 5 seconds, triggering the rate limit. This should be fixed by always rounding the timeout upwards. Additional measures like increasing the retry time or relaxing the rate limit may also prevent future errors of this form.

```
[291608.816746] wireguard: wgtest: Sending handshake initiation
    to peer 9 (172.31.28.65:51820)
[291613.680094] wireguard: wgtest: Handshake for peer 9
    (172.31.28.65:51820) did not complete after 5 seconds,
    retrying (try 7)
[291669.536100] wireguard: wgtest: Retrying handshake with peer 9
     (172.31.28.65:51820) because we stopped hearing back after
    15 seconds
```

Figure 2: Log indicating retry rate limiting. Note that the time between the first two lines is only 4.9s, and after the second log entry, no handshake initiation was sent.

4 Approach 2: State Machine Analysis

4.1 Audit methodology

We began our audit by examining the code structure. Most of the relevant code represented the session management system as a set of timers, a set of event triggers which could modify/cancel the timers (such as data sends or handshake completion), and a set of callbacks to perform on timer expiration (such as handshake reinitiation). These callbacks also modified other timer states, which led to a fairly complex state machine.

We attempted to catalogue all the timers and state transitions which were used. Our result follows.

4.2 State machines identified

This is a list of the separate state machines used, organized from lowest level (and fewest dependencies) to highest.

We will use *set* or *reset* to mean setting a timer's timeout back to the full time period, and *clear* to mean cancelling the timer and its callback entirely.

1. **Data clearing and zeroing.** The `timer_zero_key_material` timer zeros sensitive materials.

   Transitions are as follows:

- On expiration, delete all derived ephemeral keys and partial handshake materials.
- Reset the timer when an ephemeral key is derived, i.e. a handshake response is constructed or received.
- Reset the timer when the handshake retransmission gives up to clear partial handshake data.

2. **Handshake initiation and retransmission.** The `timer_retransmit_handshake` timer and the `packet_send_queued_handshake_initiation` function cooperatively manage the initiation of handshakes and its retransmission. They have the following transitions:

   - The queued send enforces the 5 second rate limit. On a non-rate-limited send, reset the timer to retransmit after $5 +$ jitter seconds.
   - On timer expiration, call the queued send to begin a new handshake, unless the retry limit has been reached.
   - The timer is cleared when an authentic session has been established (as signaled by `timers_handshake_complete`).
   - The queued send is also called by higher level systems when a send is necessary, such as opportunistic rekeying.

   This feedback loop is fairly simple and robust.

3. **Opportunistic rekeying.** Opportunistic rekeying is performed on the receive/send paths directly without any timers. This code is difficult to phrase as a state machine, but it is worth mentioning here because it affects session management and calls the handshake initiation system.

4. **Sending keepalives.** Keepalives are sent by the `packet_send_keepalive` function and managed by the `timer_send_keepalive` timer. The system has the following transitions:

   - The timer is reset when an authenticated (non-empty) data packet is received. If a second packet is received before the timer expires, set a flag and do not reset the timer.
   - The timer is cleared when a (non-empty) data packet is sent.
   - The timer is cleared when a handshake is initiated.
   - On timer expiration, send a keepalive packet (or queued data). If a second packet was received before the keepalive expired, reset the keepalive timer to send another keepalive.
   - Separately from the timer, when a handshake is completed as the initiator (a valid handshake response is received), immediately send a keepalive packet (or queued data).
   - When a new device or network is established, send an initial keepalive packet (or queued data).

- To send a keepalive packet, add a keepalive packet queue if the queue is empty, and then attempt to send the packets in the queue. Be sure to perform the usual logic of initiating a handshake if no session exists.

5. **Receiving/checking keepalives.** Checking for keepalive messages is handled by the `timer_new_handshake` timer.

   - The timer is reset when a (non-empty) data packet is sent, if the timer is not currently running.
   - The timer is cleared when any authenticated (possibly empty) data packet is received.
   - When the timer expires, queue a new handshake initiation.

6. **Persistent keepalives.** The last timer, `timer_send_persistent_keepalive` manages persistent keepalives.

   - The timer is reset when any packet is sent or received, including handshakes, keepalives, or data.
   - On timer expiration, send a keepalive packet. Also, clear the standard send-keepalive timer.

4.3 Analysis

The state machines used here seem generally quite simple and independent, and each machine has relatively few transitions, as desired. We identified one state machine to be a clear outlier: keepalive sending. Much of the additional state machine logic is in fact necessary, as keepalives involve the send path much more than other packets, and keepalives are used to preemptively establish sessions with new peers.

We did notice a particular set of keepalive-send transitions which seemed unnecessary and was actually asymmetric with keepalive receiving/checking: the keepalive timer is cleared when a handshake is initiated, while the receiver continues to expect a keepalive even if it receives a handshake initiation. The receiver behavior is correct, as a handshake initiation cannot be guaranteed as authentic, so the receiver should not act on it. The sender in fact generally does send a keepalive anyways, when the handshake is completed (to confirm the handshake). Thus, in practice, this behavior is acceptable. We did construct an edge case in which the handshake-completion keepalive arrives too late, so the keepalive checker also initiates a handshake, causing duplicate work, though this case is rare and requires somewhat specific conditions (see Appendix A).

Although this rarely causes error conditions, we still propose a simplification of the state machine, to make these conditions easier to reason about. In particular:

- Eliminate the transition to clear the keepalive timer on handshake initiation.

- For consistency, clear the keepalive timer on keepalive sends in addition to data sends. This transition may be rare, but there is little reason not to include it.
- We're not sure if the flag to send a second keepalive packet is necessary. More investigation may be necessary, but we believe it can be eliminated.

These changes cause the send-keepalive logic to be an almost exact mirror if the receive-keepalive logic, which makes correctness simpler to reason about and understand. It also removes the dependency of the state machine on the particular case of sending a keepalive on a handshake completion.

5 Conclusions

Overall, we found that WireGuard generally functions as it should. The protocol indeed was very simple, and we found out how easy it was to set up. The design philosophy indeed led to a concise and simple design. We did not identify any critical bugs, but we did find some areas of WireGuard that could be improved. To recap, we recommend that:

1. The handshake retransmission timer should round upwards, rather than downwards, to ensure that the rate limit is not triggered.
2. The send-keepalive timer should not be cleared on handshake initiation and should be generally simplified.

We also hope to incorporate some of the documentation from this report into the WireGuard code base or website for future reference.

We'd like to thank Jason Donenfeld for meeting with us and helping us understand WireGuard, as well as Professor Rivest, Professor Kalai and the 6.857 course staff for their resources and support.

References

[1] Jason A. Donenfeld. *WireGuard: Next Generation Kernel Network Tunnel*. 2018. URL: `https://www.wireguard.com/papers/wireguard.pdf` (visited on 05/16/2018).

[2] Jason A. Donenfeld and Kevin Milner. *Formal Verification of the WireGuard Protocol*. 2018. URL: `https://www.wireguard.com/papers/wireguard-formal-verification.pdf`.

[3] Trevor Perrin. *The Noise Protocol Framework*. 2017. URL: `http://noiseprotocol.org/noise.html`.

A Scenario for Double-Handshaking

We present a scenario in which both parties initiate a handshake due to a cancelled keepalive message. This requires either relatively high (second-scale) latency, cookies enabled on the server, or both. For the sake of this discussion, we proceed with a 0.25 second latency between the two peers, which is high but not unheard of; lower latency can still trigger this behavior, though with lower probability.

Consider a scenario in which two peers I and R are operating on an open WireGuard session (the session's initiator and receiver, respectively). Due to inactivity, the session has not been recreated up until this point. We give time stamps relative to the start of the session. Also, assume that the receiver is under load and is thus using cookies.

Consider the following sequence of events:

- At 2:37.00, R sends a data packet $D1$ to I. It also sets `timer_new_handshake` to expire at 2:52.00 to begin a handshake if no authenticated data response arrives.
- The packet $D1$ arrives at I at 2:37.25. I sets the `timer_send_keepalive` to send a keepalive at time 2:47.25.
- At 2:46.75, R sends a data packet $D2$ to I. It does not update `timer_new_handshake`, as the timer is still pending.
- At 2:47.00, I receives the data packet $D2$. Because this is late in the session, I sends a handshake initialization to R, and also cancels its `timer_send_keepalive`.
- At 2:47.25, R receives handshake initialization, and responds with a cookie. I receives the cookie and stores it.
- At 2:52.00, I resends its handshake initialization to R, with the cookie attached.
- At 2:52.00, R's `timer_new_handshake` expires, and R initiates a handshake to I.

Thus, both servers have sent a handshake to each other, duplicating the work necessary. If, on the other hand, I had not cancelled its `timer_send_keepalive`, then I would have actually responded with a keepalive and R would not initiate a new handshake. This is a small edge case, but occurs with nontrivial probability in a real system with some latency.

Similar cases do show that there is some nontrivial interaction in sending keepalives in almost-expired sessions, especially if latency is high or cookies are in use. The only simple solution we have come up with is extending the keepalive receive timeout to account for longer-latency handshakes, which does not seem to fit the common case. This requires additional consideration.