Fine penetration tests for fine websites

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Audit-Report Obscura Protocol & MacOS VPN Client 06.2025

Cure53, Dr.-Ing. M. Heiderich, Dr. N. Kobeissi, Dr. D. Bleichenbacher, J. Ginesin

Index

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Introduction

“Privacy that’s more than a promise Meet Obscura: the first VPN that can’t log your activity and outsmarts internet censorship. VPNs know more about you than they should. So we made one that doesn’t. Even “no-log” VPNs can track you, since they see both who you are and what you do. Obscura is built such that we can’t see your traffic in the first place.”

From https://obscura.net/#faq

This report describes the results of a security assessment of the Obscura complex, covering the Obscura MacOS application, the Obscura network extension, as well as the Obscura protocol design. The project, which included a penetration test and a dedicated source code audit, was conducted by Cure53 in May 2025.

The audit, registered as *OBS-01*, was requested by Sovereign Engineering Inc. in April 2025 then scheduled to start shortly after, i.e., in May 2025. The project initiates the cooperation between Cure53 and Sovereign Engineering Inc. / Obscura maintainers.

In terms of the exact timeline and specific resources allocated to *OBS-01*, the Cure53 team has completed their research in CW21 and CW22 of 2025. In order to achieve the expected coverage for this task, a total of twenty days were invested. A team consisting of three senior testers was formed and assigned to the preparation, execution, documentation, and delivery of this project.

For optimal structuring and tracking of tasks, the assessment was divided into three separate work packages (WPs):

- **WP1**: White-box penetration tests & code audits against Obscura MacOS application
- **WP2**: White-box penetration tests & code audits against Obscura network extension
- **WP3**: White-box penetration tests & code audits against Obscura protocol design

As the titles of the WPs indicate, the white-box methodology was used across all three components of this *OBS-01* project. Cure53 was provided with documentation, as well as all further means of access required to complete the tests. In addition, all sources corresponding to the test targets were shared to ensure that the project could be executed in accordance with the agreed framework.

The project was completed without any major issues. To facilitate a smooth transition into the testing phase, all preparations were completed in CW20. Throughout the engagement, communications were conducted through a private, dedicated, and shared Slack channel. Stakeholders - including Cure53 testers and the internal staff from Obscura - were able to participate in discussions in this space.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Cure53 did not need to ask many questions, and the quality of all project-related interactions was consistently excellent. The testers offered frequent status updates on the examination and emerging findings. What is more, the issues were live-reported during this project, with the aforementioned Slack channel used for this purpose.

Continuous communication contributed positively to the overall results of this project. Significant roadblocks were avoided thanks to clear and careful preparation of the scope, as well as through subsequent support.

The Cure53 team achieved very good coverage of the WP1-WP3 objectives. Of the six security-related discoveries, two were classified as security vulnerabilities and four were reported only as general weaknesses, i.e., flaws with lower potential for exploitation. Given this small array of shortcomings, it can be argued that the results of this *OBS-01* project are positive.

More generally, the inspected Obscura stack revealed sophisticated engineering and high code quality throughout its components. This is highlighted by an innovative 2-party relay system and a seamless integration of WireGuard and QUIC protocols. The audited system demonstrates strong security practices, proper privilege separation and the use of state-of-the-art cryptography while maintaining user anonymity and prioritizing low latency within a well-defined adversarial model

As noted, the assessment found just a handful of issues, predominantly of very limited severity, pointing towards calling for incremental improvements rather than critical design alterations. Recommendations include enhancing credential storage security, improving connection resilience, addressing potential memory exhaustion issues, and mitigating risks of client fingerprinting and traffic analysis.

The following sections first describe the scope and key test parameters, as well as how the work packages were structured and organized.

Then, what the Cure53 team did in terms of attack attempts, coverage, and other test-related tasks is explained in a separate, detailed and extensive chapter on test methodology.

Next, all findings are discussed in grouped vulnerability and miscellaneous categories. The problems are then discussed chronologically within each category. In addition to technical descriptions, PoC and mitigation advice is provided where applicable.

The report ends with general conclusions relevant to this May 2025 project. Based on the test team's observations and the evidence collected, Cure53 elaborates on the overall impressions and reiterates the verdict. The final section also includes tailored hardening recommendations for the Obscura complex.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Scope

- **Penetration tests & code audits against Obscura protocol & VPN client for MacOS**
  - **WP1:** White-box penetration tests & code audits against Obscura MacOS application
    - **Source code URL:**
      - https://github.com/Sovereign-Engineering/obscuravpn-client
    - **Commit hash:**
      - 6c79c298320946e6c5be0506801d32b81d884602
  - **WP2:** White-box penetration tests & code audits against Obscura network extension
    - **Source code URL:**
      - https://github.com/Sovereign-Engineering/obscuravpn-client
    - **Commit hash:**
      - 6c79c298320946e6c5be0506801d32b81d884602
  - **WP3:** White-box penetration tests & code audits against Obscura protocol design
    - **Design Overview:**
      - https://obscura.net/blog/bootstrapping-trust/
      - An additional Threat-Modeling PDF to review was shared with Cure53.
  - **Test-supporting material was shared with Cure53**
  - **All relevant sources were shared with Cure53**

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Test Methodology

This section outlines the comprehensive test methodology employed during the security audit of the Obscura Protocol & VPN Client for macOS. The methodology was designed to provide thorough coverage across three work packages spanning application security, network extension security and protocol design analysis. The testing approach combined white-box security assessment techniques with architectural analysis to ensure a complete evaluation of the system's security posture.

Target overview

The Obscura VPN client represents a sophisticated multilayered architecture that implements a two-party relay system. This system is similar to other modern networking privacy protocols such as iCloud Private Relay. The technical stack is designed to separate concerns across components, as reflected by three distinct work packages, each handling different aspects of the VPN functionality while maintaining security boundaries.

Obscura's technical architecture is built around two core elements that address fundamental problems in existing VPN solutions:

- The implementation of a two-party relay system that eliminates the inherent trust problem of traditional VPNs by ensuring that no single entity can correlate user identity with browsing activity.
- The usage of QUIC as the transport protocol for tunneling WireGuard packets, which provides both obfuscation benefits by blending with HTTP/3 traffic and performance advantages by avoiding TCP-over-TCP issues.

The system employs a separation of responsibilities across multiple architectural layers, each implemented using the most appropriate technology stack for its specific requirements, i.e., Swift for macOS-level APIs, TypeScript for the user-interface, Rust for low-level network adapters. This separation is maintained through communication channels between components using Foreign Function Interface (FFI) boundaries.

WP1: macOS application security testing

The macOS application security testing phase focused on analyzing the Swift frontend components, as well as its function as the "glue" between the TypeScript frontend and the Rust low-level network adapter/backend. The testing methodology examined component interaction patterns to identify potential security boundaries and privilege escalation vectors within the application architecture.

Particular attention was paid to FFI boundaries between Swift and Rust components, as these represent critical security transition points where type safety and memory safety guarantees may be compromised.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

WP2: Network extension security testing

The network extension's security testing focused on the core network components that handle sensitive user traffic and implement the custom VPN protocol. The QUIC protocol implementation analysis examined the security of connection establishment procedures in *quicwg.rs*, including detailed review of the handshake procedures and certificate validation logic. This analysis identified the issue documented in OBS-01-003.

The message handling analysis examined message parsing and validation logic to identify injection vulnerabilities and protocol compliance issues. The custom certificate verified *VerifyVpnServerCert* was analyzed for potential bypass techniques and cryptographic weaknesses that could compromise the security of relay authentication. Error handling mechanisms were tested to identify information disclosure vulnerabilities and to ensure that error propagation does not leak sensitive information about internal system state.

WireGuard integration testing revealed some potential security improvements, as discussed in OBS-01-001. The analysis revealed that long-term cryptographic keys are stored unencrypted in plain text JSON configuration files, exposing them to potential compromise by malicious processes or unauthorized users with filesystem access. The testing examined key generation, rotation, and lifecycle management procedures to ensure cryptographic best practices are followed and that key material is properly protected throughout its lifecycle.

The relay selection and management analysis led to the OBS-01-004 vulnerability being discovered. Cure53 found that unbounded parallel QUIC handshakes can be triggered by malicious API responses containing arbitrarily large lists of relay servers. The testers examined the implementation to understand resource consumption patterns during relay selection, parallelly evaluating the effectiveness of connection abandonment and cleanup procedures.

WP3: Protocol design security analysis

The Obscura protocol is designed to improve user privacy by employing a two-party relay-exit architecture, preventing any single entity from linking user identities to their traffic. *QUIC* is used as the transport layer for the WireGuard packets, offering performance and obfuscation benefits. The adversarial model assumes a passive or active network attacker capable of observing, modifying, inserting, or replaying traffic. Post-compromise scenarios of relay servers should also be considered.

The protocol design security analysis focused on evaluating the fundamental cryptographic and architectural assumptions underpinning the Obscura protocol. Particular attention was paid to the distribution and validation of cryptographic key material, metadata leakage resistance, the resilience of the protocol against passive and active network adversaries within the defined threat model, potential information leakage, and evaluating handshake flows.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

The design of the protocol changes the security requirements of the WireGuard protocol, since the WireGuard server no longer can learn (and verify) the sender's identity. This presents additional attack vectors. The design of the WireGuard protocol has been analyzed against these added requirements.

Risk assessment methodology

The vulnerability classification system employed a five-tier risk assessment framework ranging from *Critical* system compromise scenarios to *Informational* security improvement recommendations.

Specifically, *Critical* vulnerabilities were defined as those enabling direct system compromise or complete privacy breach. Going down the spectrum, *High*-severity issues represent significant security weaknesses with clear exploitation paths, while *Medium*-severity vulnerabilities require specific conditions for successful exploitation.

On the end of limited exploitation, *Low*-severity issues signify minor security concerns with limited potential of endangering or harming the components and complex. Finally, *Info*-findings offer security improvement recommendations, but the patterns described within these items carry no immediate risk.

Impact analysis considered multiple dimensions of potential harm, including confidentiality impact through user traffic exposure and metadata leakage, integrity impact through data tampering and protocol manipulation capabilities, availability impact through service disruption and Denial-of-Service potential, and privacy impact through user identification and traffic correlation vulnerabilities. This comprehensive impact assessment framework ensured that all potential consequences of the identified vulnerabilities could be properly evaluated and prioritized.

Methodology: effectiveness and results

Adopting the test methodology described, Cure53 successfully identified **six** distinct security findings, including **two** exploitable vulnerabilities and **four** general security weaknesses that could impact the overall security posture of the system. The white-box approach combined with multilayered testing provided deep insights into both implementation-level vulnerabilities and design-level security considerations.

The systematic approach to security testing balanced breadth of coverage with depth of analysis, ensuring that critical security properties were thoroughly validated while also identifying practical implementation issues that could impact real-world deployment security. The methodology's effectiveness was demonstrated through the identification of previously unknown vulnerabilities that could have significant impact on user security and privacy, as well as the validation of security controls and the identification of areas for improvement.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Identified Vulnerabilities

The following section lists all vulnerabilities and implementation issues identified during the testing period. Notably, findings are cited in chronological order rather than by degree of impact, with the severity rank offered in brackets following the title heading for each vulnerability. Furthermore, each ticket has been given a unique identifier (e.g., *OBS-01-001*) to facilitate any follow-up correspondence.

OBS-01-003 WP2: Unbounded vector allocation on message receipt *(Medium)*

Fix Note: *This issue has been fixed by the development team and verified by Cure53 to be working as expected. The described issue no longer exists.*

The client *QUIC* message handler, *recv_resp,* allocates a vector of length *msg_header.payload_length*. However, *msg_header.payload_length* is never sanitized before usage and can be set to be arbitrarily large by a malicious API server. Allocating a large enough vector may exhaust the memory resources of the client, therefore causing the client to crash or become unresponsive.

Affected file:
rustlib/src/quicwg.rs

Affected code:

```
async fn recv_resp(&mut self) -> Result<(), QuicWgRelayHandshakeError> {
	loop {
	let mut buffer = [0u8; 8];
	self.recv
		.read_exact(&mut buffer)
		.await
		.map_err(QuicWgRelayHandshakeError::ControlStreamReadError)?;
	let msg_header = MessageHeader::from(buffer);
	const_assert!(size_of::<usize>() >= size_of::<u32>());
	let mut payload = vec![0u8; msg_header.payload_length as usize];
	self.recv
		.read_exact(&mut payload)
		.await
		.map_err(QuicWgRelayHandshakeError::ControlStreamReadError)?;
```

It is recommended to implement robust validation of any value used to determine memory allocation. Before a vector is allocated for the buffer, *msg_header.payload_length* should be checked to be within a desirable range.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

OBS-01-004 WP2: Unbounded parallel *QUIC* handshakes on relay selection *(Low)*

Fix Note: *This issue has been fixed by the development team and verified by Cure53 to be working as expected. The described issue no longer exists.*

When selecting an Obscura relay for a new VPN connection, the client fetches a list of Obscura relay servers from the API and triggers a *QUIC* handshake to each relay in parallel. However, the list of relays is never validated nor constrained to be within certain limits before the client triggers parallel *QUIC* handshakes with each relay.

A malicious API server could send a client an arbitrarily large number of phony relay servers, causing the client to attempt an arbitrarily large number of *QUIC* handshakes simultaneously. Each *QUIC* handshake requires allocating buffers for TLS; therefore, attempting an arbitrary number of *QUIC* handshakes may exhaust the memory or computational resources of the client, causing the client to crash or become unresponsive.

Affected file:
rustlib/src/client_state.rs

Affected code:

```
pub async fn select_relay(&self) -> Result<(OneRelay,
QuicWgConnHandshaking), TunnelConnectError> {
       let relays = self.api_request(ListRelays {}).await?;
       tracing::info!("relay candidates: {:?}", relays);

       let racing_handshakes = race_relay_handshakes(relays)?;
       let mut relays_connected_successfully = BTreeSet::new();
       let mut best_candidate = None;
```

It is recommended to more thoroughly validate the list of relays received from the API server, or to batch *QUIC* handshake attempts instead of triggering every single one right away parallelly. Additionally, since the client only uses *QUIC* handshakes in relay selection to estimate RTT, it may be better to switch it for a more lightweight protocol such as UDP or ICMP.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Miscellaneous Issues

This section covers any and all noteworthy findings that did not incur an exploit but may assist an attacker in successfully achieving malicious objectives in the future. Most of these results are vulnerable code snippets that did not provide an easy method by which to be called. Conclusively, while a vulnerability is present, an exploit may not always be possible.

OBS-01-001 WP2: Long-term key storage improvements via Keychain API *(Info)*

Fix Note: *This issue has been fixed by the development team and verified by Cure53 to be working as expected. The described issue no longer exists.*

The WireGuard private keys used for VPN connections are stored unencrypted in plain text JSON configuration files on disk. The *WireGuardKeyCache* struct contains sensitive cryptographic material including the current *secret_key* and historical *old_public_keys*, which are serialized and saved directly to the filesystem via the configuration persistence mechanism. This approach exposes long-term cryptographic keys to potential compromise by malicious processes, malware or unauthorized users who gain filesystem access in another way.

While this sort of unauthorized access is rendered less likely due to the JSON file being permissioned towards the root macOS user, it remains an unnecessary risk. This should especially be revised given the existence of more secure persistent key storage alternatives, such as the macOS Keychain API.[1]

Affected file:
rustlib/src/config/persistence.rs

Affected code:

```
#[serde_with::serde_as]
#[derive(Serialize, Deserialize, Clone, PartialEq, Eq)]
pub struct WireGuardKeyCache {
    #[serde_as(as = "serde_with::base64::Base64")]
    secret_key: [u8; 32],
    #[serde_as(as = "Option<serde_with::TimestampSeconds>")]
    first_use: Option<SystemTime>,
    #[serde_as(as = "Option<serde_with::TimestampSeconds>")]
    registered_at: Option<SystemTime>,
    #[serde_as(as = "Vec<serde_with::base64::Base64>")]
    old_public_keys: Vec<[u8; 32]>,
}

[...]
```

[1] https://developer.apple.com/documentation/security/keychain-services

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
pub fn save(config_dir: &Path, config: &Config) -> Result<(),
ConfigSaveError> {
    let json = match serde_json::to_vec_pretty(config) {
        Ok(json) => json,
        Err(error) => {
            tracing::error!(
                    config.dir =? config_dir,
                    ?error,
                    "config::save could not encode config"
            );
            return Err(ConfigSaveError::SerializeError(error));
        }
    };
```

It is recommended to integrate platform-specific secure storage APIs such as the macOS Keychain API, Windows Credential Manager, or Linux Secret Service API. These solutions can be used to optimally store the WireGuard private keys separately from the main configuration file.

Moreover, the configuration file should only contain non-sensitive metadata and references to securely stored keys. This would provide hardware-backed encryption, where available, and can ensure that cryptographic material is protected by the operating system's credential management infrastructure.

OBS-01-002 WP2: Connection retry logic fails after fixed number of retries *(Info)*

Note: *This issue was confirmed to be a false positive after the delivery of the audit report, since the connection logic referenced below was later determined to be part of an infinite outer loop.*

Obscura’s WireGuard handshake retry mechanism uses a fixed number of retry attempts defined by the *WG_FIRST_HANDSHAKE_RESENDS* constant. Once these are exhausted, the connection permanently fails without any recovery mechanism to be seen.

This approach can cause connection failures in scenarios where temporary network conditions or server-side issues might resolve themselves given additional time or retry attempts. The fixed retry limit does not account for varying network conditions, temporary server unavailability, or transient connectivity issues that are common in mobile and unstable network environments.

Affected file:
rustlib/src/quicwg.rs

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Affected code:

```
const WG_FIRST_HANDSHAKE_RESENDS: usize = 25; // 2.5s per handshake.
const WG_FIRST_HANDSHAKE_TIMEOUT: Duration = Duration::from_millis(100);
[...]
// Simple retry code omitted for brevity.
```

The current implementation likely uses a simple counter-based retry mechanism that permanently abandons the connection attempt once the fixed retry limit is exceeded. This can lead to unnecessary connection failures when users experience temporary network disruptions, particularly affecting mobile users who frequently transition between different networks.

It is recommended to implement an adaptive retry strategy that considers factors such as the type of error encountered, network conditions, and elapsed time. This approach supersedes relying solely on a fixed retry count. Obscura should consider implementing exponential backoff with jitter, distinguishing between retryable and non-retryable errors, and potentially allowing for longer-term retry attempts with appropriate user feedback. Additionally, the retry logic should gracefully handle scenarios where connectivity might be restored after extended periods.

OBS-01-005 WP2: Client-relay RTT leakage on client relay selection *(Info)*

When selecting an Obscura relay for a new VPN connection, the client fetches a list of Obscura relay servers from the API and triggers a *QUIC* handshake to each relay in parallel to attempt to discover the fastest relay server relative to itself. An attacker observing network traffic between the client and one or more Obscura relay servers, or an attacker controlling one or more Obscura relay servers, could observe the RTTs on the *QUIC* UDP packets. This could assist fingerprinting or attempts to geolocate the client.

In particular, an attacker observing the network traffic may timestamp each *QUIC* UDP handshake message - as it passes - to calculate RTT. Similarly, an attacker controlling an Obscura server may calculate the difference between the first *QUIC* client “hello” and the client’s second handshake message, again gaining the capacity to calculate RTT.

Affected file:
rustlib/src/relay_selection.rs

Affected code:

```
pub fn race_relay_handshakes(relays: Vec<OneRelay>) ->
Result<Receiver<(OneRelay, u16, Duration, QuicWgConnHandshaking)>,
RelaySelectionError> {
	let mut tasks = JoinSet::new();
	let udp = new_udp(None).map_err(RelaySelectionError::UdpSetup)?;
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
        let quic_endpoint =
new_quic(udp).map_err(RelaySelectionError::QuicSetup)?;

        for relay in relays {
        for &port in &relay.ports {
                let quic_endpoint = quic_endpoint.clone();
                let relay_addr = (relay.ip_v4, port).into();
                let relay_cert = relay.tls_cert.clone().into();
                let relay = relay.clone();
                tasks.spawn(async move {
                let result: Result<(QuicWgConnHandshaking, Duration),
QuicWgConnectError> = async {
                        let mut handshaking =
QuicWgConnHandshaking::start(relay.id.clone(), &quic_endpoint, relay_addr,
relay_cert).await?;
                        let rtt = handshaking.measure_rtt().await?;
                        Ok((handshaking, rtt))
                }
                .await;
                (result, relay, port)
                });
        }
}
```

It is recommended to add additional latency on the second *QUIC* client handshake packet, or to switch to a stateless protocol such as UDP or ICMP for measuring latency to potential relay servers. This way, relay servers are kept uninformed of the RTT to the client, and potential leakage of client-server RTT to any observer is minimized.

OBS-01-006 WP2: *QUIC* data packets are unpadded *(Info)*

Fix Note: *This issue has been fixed by the development team and verified by Cure53 to be working as expected. The described issue no longer exists.*

Throughout the client implementation, *QUIC* messages are transmitted without padding, thereby increasing susceptibility to network traffic analysis based on packet size variabilities. Additionally, because WireGuard does not offer any padding nor obfuscation features by default, packets appear as variable in length on the wire[2].

Therefore, an attacker could more easily fingerprint client traffic between Obscura relays and Mullvad exit nodes. Similar traffic analysis techniques have been shown to be effective in closely related scenarios[3].

[2] https://www.wireguard.com/known-limitations/
[3] https://www.usenix.org/conference/usenixsecurity24/presentation/xue-fingerprinting

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Affected file:
rustlib/src/quicwg.rs

Affected code:

```
// logging code removed for brevity
pub fn send(&self, packet: &[u8]) {
       let mut wg_state = self.wg_state.lock().unwrap();
       let WgState { buffer, wg, traffic_stats, last_send_err_logged_at,
tick_stats, .. } = &mut *wg_state;
       match wg.encapsulate(packet, buffer) {
              TunnResult::WriteToNetwork(datagram) => {
              tick_stats.wg_tx_count += 1;
              match self.quic.send_datagram(datagram.to_vec().into()) {
                     Ok(()) => tick_stats.quic_tx_count += 1,
                     Err(error) => // error code
              }
        }
}
```

To obfuscate client traffic, it is highly recommended to employ padding frames[4] in *QUIC.* This would extend the packet sizes to a fixed, consistent length. It is also advisable to introduce random background traffic and random packet sending intervals to further prevent traffic analysis.

[4] https://datatracker.ietf.org/doc/html/rfc9000#section-19.1

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Conclusions

As noted in the *Introduction,* Obscura is a well-engineered privacy solution with no major security vulnerabilities within its defined threat model. Cure53 is happy to report that the findings of this *OBS-01* project only represent opportunities to further strengthen the already robust security posture.

This examination, performed by three members of the Cure53 team in May 2025, demonstrated that the Obscura stack boasts sophisticated engineering across its TypeScript frontend, Rust backend, and Swift macOS network extension components. The Obscura complex is characterized by an observably consistent high code quality and appropriate use of each language's safety features.

The novel 2-party relay system effectively eliminates traditional VPN trust requirements while maintaining strong security boundaries between user identity and browsing activity, representing a significant architectural achievement.

Similarly, the innovative merging of WireGuard and *QUIC* protocols into a singular stack is clean and impressive, showcasing strong software engineering skills with well-segmented, idiomatic Rust implementation. The multilayered security approach includes proper privilege separation between components and well-defined FFI boundaries, reflecting mature software engineering practices and deep understanding of system security principles.

The underlying cryptographic primitives are state-of-the-art and suitable for the implemented protocol, utilizing robust and well-maintained cryptographic libraries. The protocol's handling of public key authenticity is crucial for preventing traffic analysis that could reveal both sender and receiver identities as well as transaction content. While WireGuard typically assumes mutual public key authentication, the proposed solution maintains user anonymity by modifying this assumption without enabling new attack vectors such as man-in-the-middle attacks.

The protocol prioritizes low latency over maximum protection against global attackers, with detailed documentation of the adversarial model and underlying assumptions. The assessment identified several issues that are predominantly minor in nature, totaling two vulnerabilities and four general weaknesses.

In the absence of fundamental or major problems, Cure53 advises focusing on minor improvements, which have been made across five areas. First, in terms of storage of credentials, OBS-01-001 shows how WireGuard private keys and sensitive cryptographic material are stored unencrypted in plain text JSON files. As this exposes them to potential compromise, migration to platform-specific secure storage APIs like macOS Keychain is recommended.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Second, connection resilience was pointed out in OBS-01-002: The fixed 25-retry limit for WireGuard handshake attempts can cause unnecessary failures during temporary network issues. An adaptive retry strategy with exponential backoff would improve reliability.

Two memory exhaustion vulnerabilities - OBS-01-003 and OBS-01-004 concern the fact that the *QUIC* message handler and relay selection process can be exploited by malicious API servers. Thus, it could lead to forcing arbitrary memory allocation, potentially causing resource exhaustion. Additional validation of payload lengths and limiting parallel connection attempts is recommended.

In the second-to-last realm, client fingerprinting and geolocation was seen as unnecessarily facilitated by OBS-01-005. The relay selection process, which uses paralle*l QUIC* handshakes, can leak RTT information, could be used for this purpose. Adding latency to handshake packets or switching to stateless protocols like UDP/ICMP would mitigate this risk.

Finally, OBS-01-006 presents the results linked to traffic analysis. Unpadded *QUIC* data packets allow potential network traffic analysis based on packet size variability. Implementing fixed-length packet padding using *QUIC* padding frames would enhance privacy protection.

To conclude, the implementation does not seem to suffer from any major security issues within the defined threat model of Obscura. Most vulnerabilities occur in post-compromise scenarios and therefore carry lower severity. The comprehensive evaluation confirms that Obscura represents a well-engineered privacy solution with strong foundational security practices.

Cure53 would like to thank Carl Dong, Florian Uekermann and Kevin Cox from the Sovereign Engineering Inc. team for their excellent project coordination, support and assistance, both before and during this assignment.