Security Assessment and Architecture & Risk Analysis of the Geph Desktop and Mobile Applications

TABLE OF CONTENTS

EXECUTIVE SUMMARY

Include Security (IncludeSec)

IncludeSec brings together some of the best information security talent from around the world. The team is composed of security experts in every aspect of consumer and enterprise technology, from low-level hardware and operating systems to the latest cutting-edge web and mobile applications. More information about the company can be found at www.IncludeSecurity.com.

Assessment Objectives

This assessment consisted of two main components: a security assessment and a holistic security analysis of the Geph project. The objective of the security assessment was to identify and confirm potential security vulnerabilities within targets in-scope of the SOW. The team assigned a qualitative risk ranking to each finding. Recommendations were provided for remediation steps which Geph could implement to secure its applications and systems.

The objective of the holistic security analysis was to formulate a set of prescriptive steps that the Geph team can implement to ensure that security is built into every facet of the Geph software development lifecycle (SDLC). Specific attention was paid to:

- Producing a high-level threat model, which can be used to drive future security initiatives for Geph.
- Assessing the security posture of Geph's application infrastructure and application architecture.
- Performing a review of Geph's cryptography and custom protocol.
- Identifying actionable process improvements to integrate security into Geph's Software Development Life Cycle (SDLC).

Scope and Methodology

Include Security performed a security assessment of Geph's Desktop and Mobile Applications on behalf of the Open Technology Fund. The assessment team performed a 16 day effort spanning from Jan 16th 2023 – Feb 6th 2023, using a Grey Box Standard assessment methodology which included a detailed review of all the components described in a manner consistent with the original Statement of Work (SOW).

Findings Overview

IncludeSec identified 17 categories of findings. There were 7 deemed to be "Medium-Risk," and 9 deemed to be "Low-Risk," which pose some tangible security risk. Additionally, 1 "Informational" level finding was identified that does not immediately pose a security risk.

IncludeSec encourages Geph to redefine the stated risk categorizations internally in a manner that incorporates internal knowledge regarding business model, customer risk, and mitigation environmental factors.

Next Steps

IncludeSec advises Geph to remediate as many findings as possible in a prioritized manner and make systemic changes to the Software Development Life Cycle (SDLC) to prevent further vulnerabilities from being introduced into future release cycles. This report can be used by as a basis for any SDLC changes. IncludeSec welcomes the opportunity to assist Geph in improving their SDLC in future engagements by providing security assessments of additional products. For inquiries or assistance scheduling remediation tests, please contact us at remediation@includesecurity.com.

RISK CATEGORIZATIONS

At the conclusion of the assessment, Include Security categorized findings into five levels of perceived security risk: Critical, High, Medium, Low, or Informational. **The risk categorizations below are guidelines that IncludeSec understands reflect best practices in the security industry and may differ from a client's internal perceived risk. Additionally, all risk is viewed as "location agnostic" as if the system in question was deployed on the Internet. It is common and encouraged that all clients recategorize findings based on their internal business risk tolerances. Any discrepancies between assigned risk and internal perceived risk are addressed during the course of remediation testing.**

Critical-Risk findings are those that pose an immediate and serious threat to the company's infrastructure and customers. This includes loss of system, access, or application control, compromise of administrative accounts or restriction of system functions, or the exposure of confidential information. These threats should take priority during remediation efforts.

High-Risk findings are those that could pose serious threats including loss of system, access, or application control, compromise of administrative accounts or restriction of system functions, or the exposure of confidential information.

Medium-Risk findings are those that could potentially be used with other techniques to compromise accounts, data, or performance.

Low-Risk findings pose limited exposure to compromise or loss of data, and are typically attributed to configuration, and outdated patches or policies.

Informational findings pose little to no security exposure to compromise or loss of data which cover defense-in-depth and best-practice changes which we recommend are made to the application. Any informational findings for which the assessment team perceived a direct security risk, were also reported in the spirit of full disclosure but were considered to be out of scope of the engagement.

The findings represented in this report are listed by a risk rated short name (e.g., C1, H2, M3, L4, and I5) and finding title. Each finding may include if applicable: Title, Description, Impact, Reproduction (evidence necessary to reproduce findings), Recommended Remediation, and References.

INTRODUCTION

Geph

Geph is an onion-routing network whose goal is to bypass Internet censorship. Desktop and mobile application clients are available, which are used to route traffic through Geph exit servers that are based in countries that do not censor websites. Traffic can first be forwarded through Geph-run bridges if direct connections to the Geph exit servers are themselves censored. Traffic is encrypted and obfuscated to appear as random data in transport. Unlike Tor and I2P, Geph optimizes for confidentiality, performance, and censorship circumvention rather than anonymity against strong network adversaries.

The assessment team performed a 16-day assessment beginning on January 16th 2023 and ending on February 6th 2023. The assessment consisted of reviewing the Geph network and clients for security vulnerabilities, with a particular focus on network integrity and risks to confidentiality.

The assessment consisted of first threat modelling, followed by a mix of source code review and dynamic testing, and concluding with software development lifecycle review, and protocol and cryptography review. Source code review was informed by static analysis tools including Semgrep, cargo audit, cargo geiger, and npm audit. Dynamic testing was performed with the assistance of Wireshark and BurpSuite to monitor, replay and tamper with protocol traffic. For the Android and iOS applications, the team additionally used the Drozer Mobile Security Toolkit, Mobile Security Framework (MobSF), and Android SDK tools to assist in finding mobile-specific vulnerabilities.

The following components were reviewed:

- geph4-client: command-line Geph client
- geph4-protocol: utilities for request-response protocols within Geph
- gephgui: Svelte (HTML/JS) app implementing a GUI for geph4-client
- gephgui-wry: Wrapper around gephgui for desktop platforms, using the "wry" WebView crate to display the HTML/JS
- geph-android: similar, but for android
- gephgui-ios: similar, but for iOS
- gephgui-pkg: build scripts for building gephgui-wry for all desktop platforms
- geph4-exit: daemon running on Geph exit servers. VPN traffic is end-to-end encrypted between clients and exits
- geph4-bridge: daemon running on Geph bridge servers.
- geph4-binder: central authentication server for Geph
- sosistab2: Geph transport protocol
- geph4-libs: helper libraries for shared functionality including the Mizaru authentication protocol

MEDIUM-RISK FINDINGS

M1: [binder] Passwords Cached in Plaintext

Description:

The **Geph Binder** service was found to store user passwords in plaintext. The purpose for doing so was to cache them to avoid performing costly hashing on the server.

Impact:

An attacker who was able to access the server's memory or compromise a **Binder** server would be able to access the plaintext credentials for up to 100,000 users who had logged in over the past hour.

Reproduction:

The password and authentication caches were defined in the file **geph4-binder/src/bindercore_v2.rs**, lines 88-95:

```
          pwd_cache: Cache::builder()
               .time_to_live(Duration::from_secs(3600))
               .max_capacity(100000)
               .build(),
           auth_cache: Cache::builder()
               .time_to_live(Duration::from_secs(3600))
               .max_capacity(100000)
               .build(),
```

On lines 480-506 of the same file, the **verify_password()** function first checked the cache for the user's plaintext password and compared it to the entered password, before computing a hash using libsodium if not. Finally, on a successful login, the password was entered into the password cache.

```
   async fn verify_password(&self, username: &str, password: &str) -> anyhow::Result<bool> {
        if self
            .pwd_cache
            .get(username)
            .map(|known| known == password)
            .unwrap_or_default()
        {
            return Ok(true);
        }
        let mut txn = self.postgres.begin().await?;
        let (pwdhash,): (String,) = if let Some(v) =
            sqlx::query_as("select pwdhash from users where username = $1")
                .bind(username)
                .fetch_optional(&mut txn)
                .await?
        {
            v
        } else {
            return Ok(false);
        };
        if verify_libsodium_password(password.to_string(), pwdhash).await {
            self.pwd_cache.insert(username.into(), password.into());
            Ok(true)
        } else {
            Ok(false)
        }
    }
```

Recommended Remediation:

The assessment team recommends designing the user authentication process such that plaintext passwords do not need to be stored on the server. As the team presumes that the caching was added due to high CPU load caused by hashing many user passwords, the team suggests an alternative solution would be to scale the number of **Binder** servers.

A different solution would be to generating an authentication token that is relayed to a user and can be used in place of credentials for a limited period.

References:

Libsodium Documentation - Password Hashing

M2: [bridge] No Binary Authentication in Auto Update

Description:

The **Geph** bridge code contained a routine to regularly auto-update the software. No cryptographic authentication or integrity checks were applied to the binary which was automatically downloaded.

Impact:

An attacker that could replace the file on the external Backblaze file storage service would be able to cause all **Geph** bridges to run arbitrary code. This would not be catastrophic for the **Geph** network since bridges are untrusted relays, however at the very least it could lead to downtime across the network until resolved.

Reproduction:

The auto update procedure was found in the file **geph4-bridge/src/autoupdate.rs**, lines 6-31:

```
pub fn autoupdate() {
    let current_exe = std::env::current_exe().unwrap();
    loop {
        let pre_sha256 = system(format!(
            "sha256sum {} | awk '{{ print $1 }}'",
            current_exe.display()
        ));
        log::debug!("*** CURRENT SHA256: {} ***", pre_sha256);
        system("wget --retry-on-http-error 500 --retry-connrefused --waitretry=1 --read-timeout=20 --timeout=15 -t
0 https://f001.backblazeb2.com/file/geph-dl/geph4-binaries/geph4-bridge-linux-amd64 -O /tmp/new-geph4-
bridge".to_string());
        // first make sure it even runs
        system("chmod +x /tmp/new-geph4-bridge".into());
        if system("/tmp/new-geph4-bridge -h".into()).contains("information") {
            let post_sha256 = system("sha256sum /tmp/new-geph4-bridge | awk '{ print $1 }'".into());
            if pre_sha256 != post_sha256 {
                log::debug!("*** NEW SHA256: {} ***", post_sha256);
                log::debug!("** UPDATING!!!! **");
                system(format!(
                    "mv /tmp/new-geph4-bridge {}",
                    current_exe.display()
                ));
                panic!("die to update")
            }
        }
        std::thread::sleep(Duration::from_secs_f64(fastrand::f64() * 3600.0))
    }
}
```

Recommended Remediation:

The assessment team recommends cryptographically signing binary releases so that they can be validated to originate from the **Geph** development team before being automatically downloaded and run.

Additionally, the team noted that the **Geph** bridge binary was a 15-megabyte download made about once every 30 minutes on average by every bridge, regardless of whether a new version was released or not. Across multiple bridges, this could lead to high bandwidth costs. This provides a justification for releasing a hash of an updated binary (potentially on a separate channel) so that the full binary only needs to be downloaded if it differs.

References:

Code Signing

M3: [Android] Application Executable Signed with v1 Signature Scheme (JANUS Vulnerability)

Description:

The assessment team found that the **Geph** Android application executable was signed with a v1 APK signature at the time of assessment.

Using a v1 signature makes the application prone to the Janus vulnerability on devices running Android 7 or below. The Janus vulnerability allows attackers to smuggle malicious code into the APK without breaking the signature.

At the time of writing, the application supported a minimum SDK version of 21 (Android 5), which also uses the v1 signature, thus being vulnerable to this attack. Android 5 devices no longer receive updates and are vulnerable to many security concerns. It can be assumed that any installed malicious application may trivially gain root privileges on those devices using public exploits.

Impact:

The existence of this vulnerability means that attackers could trick users into installing a malicious attacker controlled APK which matches the v1 APK signature of the legitimate Android application. As a result, a transparent update would be possible without warnings appearing on Android devices, effectively taking over the existing application and all its data.

Reproduction:

The following snippet from the **apksigner** tool shows that the application supported the v1 signature scheme at the time of assessment:

```
$ apksigner verify -v geph-android.apk
Verifies
Verified using v1 scheme (JAR signing): true
Verified using v2 scheme (APK Signature Scheme v2): true
Verified using v3 scheme (APK Signature Scheme v3): false
Verified using v3.1 scheme (APK Signature Scheme v3.1): false
Verified using v4 scheme (APK Signature Scheme v4): false
Verified for SourceStamp: false
Number of signers: 1
```

Recommended Remediation:

The assessment team recommends increasing the minimum supported SDK level to at least 24 (Android 7) to ensure that this vulnerability cannot be exploited on devices running older Android versions. In addition, future production builds should be signed only with v2 or greater APK signatures.

References:

Janus Vulnerability

M4: [Android] [iOS] Daemon Commands Exposed to All Applications on Device

Description:

The **Geph** Android and iOS applications were found to expose an Inter-Process Communication (IPC) mechanism that accepted commands from any application installed on the device. This design undermined the Android and iOS security models, which normally restrict how processes communicate with each other, and could expose users to additional risk from malicious applications installed on the device.

Impact:

A malicious application on the device could send commands to the **Geph** daemon to obtain information such as the current IP address, port, or protocol. The attacker could also terminate the VPN connection on behalf of the user, which could lead to deanonymization of the user's Internet activity in hostile environments.

Reproduction:

The following proof-of-concepts were created by leveraging the **Drozer** mobile security toolkit to install **busybox** and **wget** on an Android device, allowing the assessment team to simulate a malicious application.

The following snippet shows it was possible to query metadata about the current tunnel, including IP address, port, and protocol in use:

```
# busybox wget -O- -q --post-data='{"jsonrpc":"2.0","method":"basic_stats","params":[],"id":1}' --header='Content-Type: application/x-www-form-urlencoded' http://127.0.0.1:9809

{"jsonrpc":"2.0","result":{"address":"23.81.64.120:17814","last_loss":0.0,"last_ping":37.0,"protocol":"obfstls-1","total_recv_bytes":4162600.0,"total_sent_bytes":1492982.0},"id":1}
```

The following snippet shows it was possible to query the service to confirm if a user was connected to a tunnel:

```
# busybox wget -O- -q --post-data='{"jsonrpc":"2.0","method":"is_connected","params":[],"id":1}' --header='Content-Type: application/x-www-form-urlencoded' http://127.0.0.1:9809

{"jsonrpc":"2.0","result":true,"id":1}
```

The following command was used to terminate the VPN tunnel from another application:

```
busybox wget -O- -q --post-data='{"jsonrpc":"2.0","method":"kill","params":[],"id":1}' --header='Content-Type: application/x-www-form-urlencoded' http://127.0.0.1:9809
```

The following snippet from file **geph4-client/src/connect/stats.rs**, lines 19-40, shows that adding authentication to the RPC server was possible using the **GEPH_RPC_KEY** environment variable, however this key was not set when the daemon was launched by the Android and iOS applications:

```
pub static STATS_THREAD: Lazy<JoinHandle<Infallible>> = Lazy::new(|| {
    std::thread::spawn(|| loop {
        let server = tiny_http::Server::http(CONNECT_CONFIG.stats_listen).unwrap();
        for mut request in server.incoming_requests() {
            smolscale::spawn(async move {
                if let Ok(key) = std::env::var("GEPH_RPC_KEY") {
                    if !request.url().contains(&key) {
                        anyhow::bail!("missing rpc key")
                    }
                }
                let mut s = String::new();
                request.as_reader().read_to_string(&mut s)?;
                let resp = StatsControlService(DummyImpl)
                    .respond_raw(serde_json::from_str(&s)?)
                    .await;
                request.respond(tiny_http::Response::from_data(serde_json::to_vec(&resp)?))?;
                anyhow::Ok(())
            })
            .detach()
        }
    })
});
```

Recommended Remediation:

The assessment team recommends reconsidering the design of the mobile applications and using native IPC mechanisms offered by those platforms rather than exposing the daemon RPC server on a TCP socket. iOS offers IPC via application groups, which are already partially leveraged in the **Geph** iOS application, while Android typically uses intents. These APIs allow controlled sharing of data between applications without fundamentally undermining the security models of Android and iOS.

Alternatively, the RPC service could be launched with the **GEPH_RPC_KEY** populated to require authentication, or the service could be removed entirely if applications outside of the main **Geph** application do not legitimately need to interact with it.

References:

Apple Developer Documentation - Configuring App Groups
Android Developer Documentation - Security Tips - Interprocess Communication

M5: [Android] Security Relevant User Data Stored in Clear Text

Description:

The assessment team found that the application stored the user's username, plaintext password, and recently used exit servers within the shared preferences file.

The SharedPreferences API is commonly used to permanently save small collections of key-value pairs. Data stored in a SharedPreferences object is written to a cleartext XML file. It is possible to set the SharedPreferences file to be globally readable, making it accessible to all applications on the device.

Impact:

Misuse of the SharedPreferences API can often lead to disclosure of confidential data. In this case, an attacker who obtains access to the device could extract the account credentials and conduct further targeted attacks against this user. Knowledge of the account and exit server could also help sophisticated attackers, such as nation states, trace activity back to users.

Reproduction:

The following steps can be used to confirm this finding.

1. Install the application on a rooted device.
2. Navigate to the **/data/data/io.geph.android/shared_prefs** folder and open the **daemon.xml** file:

```
blueline:/data/data/io.geph.android # cat shared_prefs/daemon.xml
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <boolean name="listenAll" value="true" />
    <string name="password">[REDACTED]</string>
    <boolean name="forceBridges" value="false" />
    <string name="excludeAppsJson">[]</string>
    <string name="dnsServerPort">49983</string>
    <string name="socksServerAddress">127.0.0.1</string>
    <string name="forceProtocol">null</string>
    <string name="socksServerPort">9909</string>
    <string name="username">Incsectest1</string>
    <string name="exitName">2.mtl.ca.ngexits.geph.io</string>
</map>
```

The following snippet from the file **app/src/main/java/io/geph/android/MainActivity.kt**, line 457, shows that the username, password, and other information are recorded using the SharedPreferences API when the tunnel service is started:

```
    protected fun startTunnelService(context: Context?) {
        Log.i(TAG, "starting tunnel service")
        val startTunnelVpn = Intent(context, TunnelVpnService::class.java)
        Log.d(TAG, "*** GONNA SET PREFERENCES ***  "+ (context == null).toString())
        val prefs = context!!.getSharedPreferences("daemon", Context.MODE_PRIVATE);
        Log.d(TAG, "*** REALLY GONNA SET PREFERENCES ***  ")
        with (prefs.edit()) {
            putString(TunnelManager.SOCKS_SERVER_ADDRESS_BASE, mSocksServerAddress)
            putString(TunnelManager.SOCKS_SERVER_PORT_EXTRA, mSocksServerPort)
            putString(TunnelManager.DNS_SERVER_PORT_EXTRA, mDnsServerPort)
            putString(TunnelManager.USERNAME, mUsername)
            putString(TunnelManager.PASSWORD, mPassword)
            putString(TunnelManager.EXIT_NAME, mExitName)
            Log.d(TAG, "*** mForceBridges *** "  + mForceBridges.toString())
            putBoolean(TunnelManager.FORCE_BRIDGES, mForceBridges!!)
            Log.d(TAG, "*** mListenAll *** "  + mListenAll.toString())
            putBoolean(TunnelManager.LISTEN_ALL, mListenAll!!)
            putString(TunnelManager.FORCE_PROTOCOL, mForceProtocol)
            putString(TunnelManager.EXCLUDE_APPS_JSON, mExcludeAppsJson)
            commit()
        }

        Log.d(TAG, "*** SET PREFERENCES ***")

        if (startService(startTunnelVpn) == null) {
            Log.d(TAG, "failed to start tunnel vpn service")
            return
        }
        TunnelState.getTunnelState().setStartingTunnelManager()
    }
```

Recommended Remediation:

The assessment team recommends implementing Android KeyStore, which supports secure credential storage as of Android version 4.3 (API level 18). It provides public APIs for storing and using app-private keys. An application can use a public key to create a new private/public key pair for encrypting application secrets, and it can decrypt the secrets with the private key.

It's possible to protect keys stored in the Android KeyStore with user authentication in a credential flow. The user's lock screen credentials (e.g., PIN, Credentials, or fingerprint) could be used for authentication.

References:

OWASP MSTG Data Storage
OWASP MSTG - ANDROID KEYSTORE

M6: [client] Time-Based Client Deanonymization

Description:

The **Geph** homepage says “we don't log you — because we can't. We use zero-knowledge authentication so that we can never associate your browsing activity with your identity”. This refers to the **Mizaru** blind signature scheme that the Binder uses for giving authentication tokens to users. A description of the scheme is given in the **Protocol and Cryptography Review** section. In practice, deanonymization is easier than claimed.

Impact:

If the reality of the protocol does not match the security claims on the site, this could lead to reputational damage. If an outside agency were to gain access to **Geph** binder infrastructure or compel the operators to identify users, it could be possible to deanonymize them.

On the GitHub wiki, it is stated that “Geph is optimized for confidentiality, performance, and censorship circumvention rather than anonymity against strong network adversaries.” The statement on the homepage may give users a false sense of security relating to anonymity.

Reproduction:

Practical deanonymization was possible with the binder, since directly after issuing a client a blind token from the binder, the client needs to make a second request to get exits and bridge descriptors from the binder.

For instance, in the **get_closest_exit()** function in the file **geph4-protocol/src/binder/client.rs**, lines 71-73, includes calls **get_auth_token()**:

```
   pub async fn get_closest_exit(&self, destination_exit: &str) -> anyhow::Result<ExitDescriptor> {
        let token = self.get_auth_token().await?.1;
        let summary = self.get_summary().await?;
```

If not already cached, **get_auth_token()** will send the user's authentication data to the binder, in exchange for a blind token. The purpose of the **Mizaru** blind signature scheme is to break the link between the authentication details and the token. However, in the **get_summary()** function, the client immediately makes another request to the binder using the blind token. It would therefore be possible for the binder to store a mapping of authentication details to tokens just based on timing linkage. Time-based client deanonymization is described in section 8.4 of the Privacy Pass paper, which is a similar protocol to **Mizaru**.

The **Geph** team noted some mitigating factors after the draft report was delivered. The client caches tokens for 24 hours and refreshes them when they expire meaning that clients do not authenticate too often.

Further, the large amount of traffic that **Geph** servers receive means that there will likely be intervening requests between the authentication and subsequent requests by the same user, making linkage somewhat less trivial.

Recommended Remediation:

The assessment team recommends changing the wording of the homepage so that the security properties of the **Mizaru** protocol are more accurately represented, since out-of-band deanonymization is difficult to mitigate.

References:

Privacy Pass: Bypassing Internet Challenges Anonymously

M7: [client] Client Did Not Validate Mizaru Keys

Description:

The **Geph** homepage refers to **Mizaru** as a "zero-knowledge authentication" scheme. However, the scheme is not technically zero-knowledge, and this can be shown straightforwardly as the client does not validate Mizaru public keys.

Impact:

If the reality of the protocol does not match the security claims on the site, this could lead to reputational damage. If an outside agency were to gain access to **Geph** binder infrastructure or compel the operators to identify users, it could be possible to deanonymize them.

Reproduction:

The binder could send connecting clients a different Mizaru keypair for every authentication. Then, the binder server could link identities by checking which key had been used in the validation step.

It appeared that steps had been taken to prevent this form of attack on the client by hardcoding Mizaru Merkle tree roots into the client config contained in **geph4-client/src/config.rs**, however in practice these were not used. In **get_auth_token()** (line 149), the **get_mizaru_pk()** function is called, which fetches the key from the binder:

```
    /// Obtains an authentication token.
    pub async fn get_auth_token(&self) -> anyhow::Result<(UserInfo, BlindToken)> {
        if let Some(auth_token) = (self.load_cache)("auth_token") {
            if let Ok(auth_token) = serde_json::from_slice(&auth_token) {
                return Ok(auth_token);
            }
        }
        let digest: [u8; 32] = rand::thread_rng().gen();
        for level in [Level::Free, Level::Plus] {
            let mizaru_pk = self.get_mizaru_pk(level).await?;
```

Recommended Remediation:

The assessment team recommends that the client checks the Mizaru public keys from the binder against the Merkle roots stored in the config to validate that the keys are part of the Merkle tree.

References:

Privacy Pass: Bypassing Internet Challenges Anonymously

LOW-RISK FINDINGS

L1: [client] Registration Captcha Bypass

Description:

The **Geph** user registration process required solving a captcha. There were multiple approaches which could circumvent this captcha validation.

Impact:

A captcha bypass could be used as part of a denial-of-service attack against the **Geph** network, either to exhaust the resources of the binder server, or to overwhelm the network with too much traffic.

Reproduction:

In the first instance, the team noted that the captcha generated was not complex. The following script, using the tesseract library for image-to-text, could be used to produce correct answers:

```
from PIL import Image
from pytesseract import pytesseract
import io
import numpy as np

with open('img.png', 'rb') as f:
    data = f.read()

img = Image.open(io.BytesIO(data)).convert('L')

import cv2
ret,img = cv2.threshold(np.array(img), 0, 255, cv2.THRESH_BINARY)
img = cv2.blur(np.array(img), (5,5))
img = Image.fromarray(img.astype(np.uint8))

img.show()

text = pytesseract.image_to_string(img, config='--psm 6 -c tessedit_char_whitelist=0123456789')
print(text)
```

The team also observed that once solved, captchas did not expire. Therefore, a registration request could reuse a captcha solution that had previously been used. Although the team did not have access to the source code of the captcha service, it was observed that captchas used at the beginning of the assessment were still valid two weeks later. This would bypass the need to solve any captchas when mass registering accounts.

Captchas were consumed in the **create_user()** function of **geph4-binder/src/bindercore_v2.rs** file:

```
    /// Creates a new user, consuming a captcha answer.
    pub async fn create_user(
        &self,
        username: &str,
        password: &str,
        captcha_id: &str,
        captcha_soln: &str,
    ) -> anyhow::Result<Result<(), RegisterError>> {
        // // EMERGENCY
        // return Ok(Err(RegisterError::Other("too many requests".into())));
        if !verify_captcha(&self.captcha_service_url, captcha_id, captcha_soln).await? {
            log::debug!("{} is not soln to {}", captcha_soln, captcha_id);
            return Ok(Err(RegisterError::Other("incorrect captcha".into())));
        }
```

Recommended Remediation:

The assessment team recommends increasing the complexity of captchas so they cannot easily be defeated by off-the-shelf Python libraries. Additionally, captchas should expire once solved.

References:

Breaking Simple Captchas with Tesseract OCR and OpenCV in Python

L2: [binder] Password Complexity Policy Does Not Follow Industry Guidelines

Description:

Password complexity policies require users to select passwords that conform to specific complexity requirements to make it more difficult for attackers to guess them using brute force attacks. In this case, **Geph** was found not to require users to select passwords that meet industry guidelines for complexity requirements.

Impact:

Users could select passwords that could be easily guessed by attackers, which would allow them to compromise the targeted users' accounts. In the context of **Geph**, this has limited impact, as access to a user's account should not expose their browsing history. Still, this could incentivize attackers to compromise premium users' accounts.

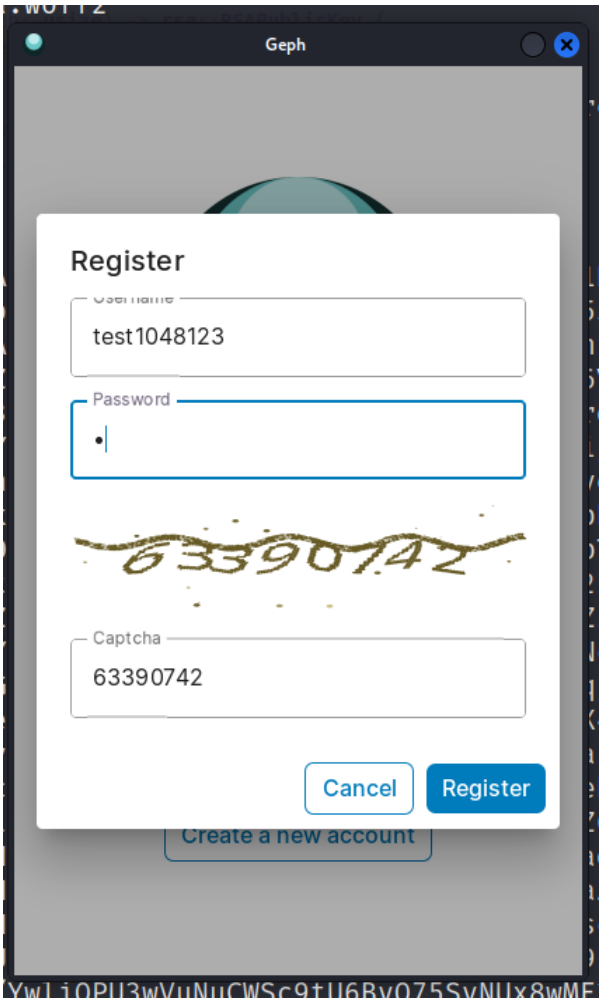
Reproduction:

The following code from the file **geph4-binder/src/bindercore_v2.rs**, lines 187-217, show the user registration function:

```
   /// Creates a new user, consuming a captcha answer.
    pub async fn create_user(
        &self,
        username: &str,
        password: &str,
        captcha_id: &str,
        captcha_soln: &str,
    ) -> anyhow::Result<Result<(), RegisterError>> {
        // // EMERGENCY
        // return Ok(Err(RegisterError::Other("too many requests".into())));
        if !verify_captcha(&self.captcha_service_url, captcha_id, captcha_soln).await? {
            log::debug!("{} is not soln to {}", captcha_soln, captcha_id);
            return Ok(Err(RegisterError::Other("incorrect captcha".into())));
        }
        // TODO atomicity
```

```
        if self.get_user_info(username).await?.is_some() {
            return Ok(Err(RegisterError::DuplicateUsername));
        }
        let mut txn = self.postgres.begin().await?;
        sqlx::query(
            "insert into users (username, pwdhash, freebalance, createtime) values ($1, $2, $3, $4) on conflict do
nothing",
        )
        .bind(username)
        .bind(hash_libsodium_password(password).await)
        .bind(1000i32)
        .bind(Utc::now().naive_utc())
        .execute(&mut txn)
        .await?;
        txn.commit().await?;
        Ok(Ok(()))
    }
```

As shown, there is no code for setting password strength. This was confirmed in practice by registering an account with the credentials **test1048123:a**:



Recommended Remediation:

The assessment team recommends enforcing a more complex password policy on the server. For example, OWASP defines the following guidelines for complexity rules:

A password should satisfy at least 3 of 4 complexity rules:

- at least 1 uppercase character (A-Z)
- at least 1 lowercase character (a-z)
- at least 1 digit (0-9)
- at least 1 special character (punctuation), including space.

Additionally, the team recommends requiring that all passwords be at least 10 characters long. For applications containing confidential data or with stricter security requirements, consider a minimum of 12 characters.

Finally, ensure that passwords are not easily guessable (e.g., identical to a user's username), and do not include more than two identical characters in a row (so **111** should not be accepted).

References:

OWASP Authentication Cheatsheet: Password Strength Controls

L3: [binder] Public Captcha Endpoint

Description:

Geph users had to solve a captcha to register an account. The backend captcha service was found to be publicly exposed on the Internet.

Impact:

In this case, public access was not required since captcha requests were proxied through **Binder** which was also controlled by **Geph**. Direct access to backend services could allow attackers to find vulnerabilities, as well as to simplify brute-force registration attacks by attacking endpoints that are not rate-limited directly (see **[client] Registration Captcha Bypass**).

Reproduction:

For example, the **/solve** endpoint of the Captcha service, hosted on Appspot, could be called directly.

Request:

```
GET /solve?id=fcofRN4ac8hXInJhukU6&soln=12658357 HTTP/2
Host: single-verve-156821.ew.r.appspot.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110
Safari/537.36
Accept: */*
```

Response:

```
HTTP/2 200 OK
X-Cloud-Trace-Context: 7c2e673b60717bd11eb23e0370367e81;o=1
Date: Wed, 18 Jan 2023 01:43:00 GMT
Content-Type: text/html
Server: Google Frontend
Content-Length: 0
Alt-Svc: h3=":443"; ma=2592000,h3-29=":443"; ma=2592000,h3-Q050=":443"; ma=2592000,h3-Q046=":443"; ma=2592000,h3-
Q043=":443"; ma=2592000,quic=":443"; ma=2592000; v="46,43"
```

Recommended Remediation:

The assessment team recommends firewalling off backend services that do not need to be externally exposed.

References:

OWASP API security – 7: Security misconfiguration

L4: [Android] Application Allows Backups

Description:

The **Geph** Android application was configured to allow backups at the time of assessment. Application backups may contain security relevant or confidential data such as user credentials, authentication tokens, or personal information.

Impact:

Allowing application backups enables an attacker with access to a device to extract application data from the backup file, which in this case would include usernames, plaintext passwords, unblinded keys and recently used bridge servers. The attacker could use this information to conduct further attacks against the targeted user, particularly if the credentials were re-used for other services.

Reproduction:

The following snippet from the file **app/src/main/AndroidManifest.xml** shows that Android backups were enabled for the **Geph** application:

```
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:usesCleartextTraffic="true"
        android:theme="@style/AppTheme">
```

Recommended Remediation:

The assessment team recommends disallowing backups for any Android applications that contain security relevant or confidential information in the application data directory. This can be achieved by setting the following directive in the application manifest:

```
android:allowBackup="false"
```

Additionally, the assessment team recommends storing security relevant information in the Android Keystore instead of the shared preferences file. This would prevent credentials from being included in backups and make it more difficult in general to extract this data.

References:

OWASP Mobile Application Security Testing Guide - Backups
Android Keystore

L5: [Android] [iOS] Native Code Not Compiled with Stack Canary Exploit Mitigation

Description:

The **Geph** Android and iOS applications included native ARM executables that were not compiled with stack canary exploit mitigation. This feature helps reduce the impact of stack-based memory corruption vulnerabilities by inserting values onto the stack that are verified when the function returns. The program then aborts if the canary has changed, which indicates memory corruption has occurred.

Impact:

An attacker who has identified a stack-based memory corruption vulnerability in the mobile applications could more easily exploit it to achieve memory disclosure or remote code execution depending on application internals.

Reproduction:

The assessment team extracted the production APK and iOS applications and leveraged the **checksec** tool to verify compiler mitigations on the native executables included with these builds. The following snippet shows that stack canary protection is disabled on the 32 and 64 bit builds of **libjnidispatch.so** included with the Android application:

```
$ /apk/lib/arm64-v8a$ checksec libjnidispatch.so
[*] 'apk/lib/arm64-v8a/libjnidispatch.so'
    Arch:     aarch64-64-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
$ /apk/lib/armeabi-v7a$ checksec libjnidispatch.so
[*] 'apk/lib/armeabi-v7a/libjnidispatch.so'
    Arch:     arm-32-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

The main **Geph** executable in the iOS build was also affected, as shown in the following snippet from the file **/private/var/containers/Bundle/Application/CF4F7810-DDE2-4686-B2F5-D9FBB4A5D79E/Geph.app/Geph**:

```
$ checksec -f Geph
MachO64: | ARC: false Canary: false Code Signature: true  Encrypted: false Fortify: true  Fortified: 4 NX Heap:
false NX Stack: true  PIE: true  Restrict: false RPath: true
```

Recommended Remediation:

The assessment team recommends adding stack canary protection to the existing exploit mitigations on all native executables. For Android, compiler and linker options (including **-fstack-protector-all**) can be added to **Android.mk** and **Application.mk** files which are processed by the NDK when the application is compiled. For iOS, these options can be added to XCode under “Other C Flags” in the project build settings.

References:

checksec.rs
OWASP Mobile Application Security Testing Guide - XCode Project Settings
Stack Canaries

L6: [client] Non-Encrypted HTTP request

Description:

The assessment team noted that some HTTP requests in **Geph** libraries did not use the HTTPS protocol.

Impact:

Information traveling in cleartext is susceptible to Man-in-the-Middle (MitM) Attacks, in which data is intercepted by an attacker with access anywhere along the network path between the user and the target server. In this case, a network-level attacker could replace an IP address with one of their own, causing **Geph** clients to connect to attacker-controlled infrastructure. However, the full implications of this were not explored during the assessment since the team could not dynamically test with a binder server.

The following instances were noted:

File	Line(s)
geph4-bridge/src/main.rs	**325**
geph4-client/src/china/mod.rs	**54,56**
geph4-exit/src/asn.rs	**41**

Reproduction:

For instance, in the file **geph4-exit/src/asn.rs**, lines 39-47, the **Geph** exit servers obtained their own IP addresses by contacting Amazon AWS over HTTP:

```
/// my own IP address
pub static MY_PUBLIC_IP: Lazy<Ipv4Addr> = Lazy::new(|| {
    let resp = ureq::get("http://checkip.amazonaws.com").call();
    resp.into_string()
        .expect("cannot get my public IP")
        .trim()
        .parse()
        .expect("got invalid IP address for myself")
});
```

The IP address returned was then used when uploading a bridge descriptor to the binder in the file **geph4-exit/src/listen.rs**, lines 517-530:

```
           // Upload a "self-bridge". sosistab2 bridges have the key field be the bincode-encoded pair of bridge
key and e2e key
            let mut _task = None;
            if let Some(client) = ctx.binder_client.clone() {
                let ctx = ctx.clone();
                _task = Some(smolscale::spawn(async move {
                    loop {
                        let fallible = async {
                            let mut unsigned_udp = BridgeDescriptor {
                                is_direct: true,
                                protocol: "sosistab2-obfsudp".into(),
                                endpoint: SocketAddr::new(
                                    (*MY_PUBLIC_IP).into(),
                                    listen_addr.port(),
                                ),
```

Recommended Remediation:

The assessment team recommends that all transport is encrypted including use of TLS for confidentiality and authentication across all HTTP requests.

References:

Transport Layer Protection Cheat Sheet

L7: [Android] Main Activity Configuration Enables Task Hijacking on Older Versions of Android

Description:

The assessment team found that **launchMode** for the app-launcher activity was set to **singletop**, which mitigates task hijacking via the StrandHogg exploit and other older techniques documented since 2015, while leaving the app vulnerable to the newer StrandHogg 2.0 exploit.

StrandHogg exploits a design flaw in the multitasking system of Android to enable malicious applications to masquerade as any other application on the device.

This vulnerability affects Android versions 3-9.x but was patched by Google in Android 8 – 9. Since the application supports devices as old as Android 5 (API level 21), this leaves users running Android 5-7.x vulnerable, as well as users running unpatched Android 8-9.x devices.

Impact:

A malicious application could leverage this vulnerability to manipulate the way in which users interact with the **Geph** application. More specifically, this would be instigated by relocating a malicious attacker-controlled activity in the screen flow of the user, which may be useful to perform Phishing and Denial-of-Service attacks or capturing user-credentials. Strandhogg has been successfully exploited by high profile banking malware trojans in the past.

Reproduction:

The following snippet from the file **app/src/main/AndroidManifest.xml**, lines 17-29, shows that the **launchMode** setting is set to **singleTop**, potentially enabling Standhogg 2.0 exploits:

```
        <activity
            android:exported="true"
            android:name=".MainActivity"
            android:label="@string/app_name"
            android:launchMode="singleTop"
            android:screenOrientation="portrait"
            android:theme="@style/AppTheme.NoActionBar">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
```

Recommended Remediation:

The assessment team recommends implementing the following countermeasures:

- The task affinity of exported application activities should be set to an empty string in the Android manifest. This will force the activities to use a randomly generated task affinity instead of the package name and prevent task hijacking, as malicious apps will not have a predictable task affinity to target.

- The **launchMode** should be changed to **singleInstance** (instead of **singleTop**).
- A custom **onBackPressed()** function could be implemented to override the default behavior.
- The **FLAG_ACTIVITY_NEW_TASK** should not be set in activity launch intents. If deemed required, one should use the aforementioned in combination with the **FLAG_ACTIVITY_CLEAR_TASK** flag.

References:

Android Developer Documentation - android:launchMode
StrandHogg
StrandHogg 2.0
Android Task Hijacking
Android phones under active attack by bank thieves
StrandHogg PoC

L8: Out-of-Date Libraries in Use

Description:

Geph was found to use outdated Rust libraries which are affected by publicly known vulnerabilities.

Impact:

The assessment team found a number of libraries used by the application to be out of date. These components have publicly known vulnerabilities, and an attacker who discovers out-of-date software within the application could use them to focus exploit attempts. Note that these vulnerabilities require very specific conditions to be exploitable; the extent to which the out-of-date components can be exploited depends largely on how these libraries are used within the application.

The following table lists out-of-date components with known vulnerabilities which were found during assessment, listed by subproject:

geph4-binder

Package	Description	Severity	Current Version	Patched Versions
time	Potential segfault in the time crate	CVSS:3.1/AV:L/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H	0.1.43	**>=0.2.23**
tokio	reject_remote_clients Configuration corruption	None	1.23.0	**>=1.18.4, <1.19.0,>=1.20.3, <1.21.0,>=1.23.1**

geph4-bridge

Package	Description	Severity	Current Version	Patched Versions
chrono	Potential segfault in `localtime_r` invocations	None	0.4.19	**>=0.4.20**
regex	Regexes with large repetitions on empty sub-expressions take a very long time to parse	CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H	1.5.4	**>=1.5.5**
thread_local	Data race in `Iter` and `IterMut`	None	1.1.3	**>=1.1.4**
time	Potential segfault in the time crate	CVSS:3.1/AV:L/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H	0.1.43	**>=0.2.23**
tokio	reject_remote_clients Configuration corruption	None	1.20.1	**>=1.18.4, <1.19.0,>=1.20.3, <1.21.0,>=1.23.1**
zeroize_derive	zeroize(drop) doesn't implement `Drop` for `enum`s	None	1.1.0	**>=1.1.1**

geph4-client

Package	Description	Severity	Current Version	Patched Versions
rustc-serialize	Stack overflow in rustc_serialize when parsing deeply nested JSON	None	0.3.24	
time	Potential segfault in the time crate	CVSS:3.1/AV:L/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H	0.1.45	**=0.2.23**
tokio	reject_remote_clients Configuration corruption	None	1.23.0	**=1.18.4, 1.19.0,=1.20.3, 1.21.0,=1.23.1**

geph4-exit

Package	Description	Severity	Current Version	Patched Versions
time	Potential segfault in the time crate	CVSS:3.1/AV:L/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H	0.1.43	**=0.2.23**
tokio	reject_remote_clients Configuration corruption	None	1.23.0	**=1.18.4, 1.19.0,=1.20.3, 1.21.0,=1.23.1**

geph4-libs

Package	Description	Severity	Current Version	Patched Versions
time	Potential segfault in the time crate	CVSS:3.1/AV:L/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H	0.1.45	**=0.2.23**

geph4-protocol

Package	Description	Severity	Current Version	Patched Versions
time	Potential segfault in the time crate	CVSS:3.1/AV:L/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H	0.1.45	**=0.2.23**

gephgui-wry

Package	Description	Severity	Current Version	Patched Versions
chrono	Potential segfault in `localtime_r` invocations	None	0.4.19	**=0.4.20**
regex	Regexes with large repetitions on empty sub-expressions take a very long time to parse	CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H	1.5.4	**=1.5.5**
rust-embed	RustEmbed generated `get` method allows for directory traversal when reading files from disk	None	6.2.0	**=6.3.0**
thread_local	Data race in `Iter` and `IterMut`	None	1.1.3	**=1.1.4**

Reproduction:

The following snippet from **cargo audit** shows that the version of **chrono** library in use was 0.4.19:

```
geph4-bridge$ cargo audit
    Fetching advisory database from `https://github.com/RustSec/advisory-db.git`
      Loaded 488 security advisories (from /home/kali/.cargo/advisory-db)
    Updating crates.io index
    Scanning Cargo.lock for vulnerabilities (348 crate dependencies)
Crate:     chrono
Version:   0.4.19
Title:     Potential segfault in `localtime_r` invocations
Date:      2020-11-10
ID:        RUSTSEC-2020-0159
URL:       https://rustsec.org/advisories/RUSTSEC-2020-0159
Solution:  Upgrade to >=0.4.20
Dependency tree:
chrono 0.4.19
└── simple_asn1 0.4.1
    └── rsa 0.3.0
        ├── rsa-fdh 0.5.0
        │   ├── mizaru 0.1.3
        │   │   └── geph4-binder-transport 0.2.0
        │   │       └── geph4-bridge 0.1.0
        │   └── geph4-binder-transport 0.2.0
        ├── mizaru 0.1.3
        └── geph4-binder-transport 0.2.0
```

Recommended Remediation:

The assessment team recommends updating all out-of-date components to their most recent releases. If this is not possible, the assessment team recommends updating all dependencies to at least the earliest version that addresses all publicly known vulnerabilities.

References:

Cargo Audit

L9: [ui] User Credentials Exposed to All Processes

Description:

The **Geph** GUI application launched the **geph4-client** in a way that revealed user credentials on the process command line.

Impact:

Low-privileged users on multi-user systems would be able to view and extract the credentials of a **Geph** user on the system.

Reproduction:

The process tree shown in this reproduction was from a Linux system running the Flatpak release of **Geph**. **gephgui-wry** spawned an instance of **geph4-client** with the command line arguments for username and password visible:

```
user      398659  4.5  0.9 104679780 78404 pts/2 Sl+  20:22   0:02  |           \_ gephgui-wry
user      398695  2.4  0.6 87337036 52744 pts/2  SLl+ 20:22   0:01  |               \_ /usr/libexec/webkit2gtk-
4.0/WebKitNetworkProcess 7 25
user      398701  9.4  2.4 87893588 201444 pts/2 SLl+ 20:22   0:04  |               \_ /usr/libexec/webkit2gtk-
4.0/WebKitWebProcess 11 15
user      399132  0.1  0.0   7740  3848 pts/2    S+   20:23   0:00  |               \_ /bin/bash /app/bin/pkexec
```

```
geph4-client connect --username Aas2d3 --password Adasd --exit-server 1.tor.ca.ngexits.geph.io --v
pn-mode tun-route
user      399138  0.0  0.0 158412  4900 pts/2    Sl+  20:23   0:00  |                  \_ flatpak-spawn --host
pkexec /home/user/.geph-blobs/geph4-client connect --username Aas2d3 --password [REDACTED] --exit-
server
 1.tor.ca.ngexits.geph.io --vpn-mode tun-route
```

The code that setup these command-line arguments was found in the file **gephgui-wry/src/daemon.rs**, lines 50-64:

```
impl DaemonConfig {
    /// Starts the daemon, returning a death handle.
    pub fn start(self) -> anyhow::Result<std::process::Child> {
        let common_args = Vec::new()
            .tap_mut(|v| {
                v.push("--username".to_string());
                v.push(self.username.clone());
                v.push("--password".into());
                v.push(self.password.clone());
                v.push("--exit-server".into());
                v.push(self.exit_hostname.clone());
                if let Some(force) = self.force_protocol.clone() {
                    v.push("--force-protocol".into());
                    v.push(force);
                }
```

Recommended Remediation:

The assessment team recommends using an alternate mechanism to pass credentials to the client that does not involve displaying credentials on the command line. This could be via a UNIX pipe, environmental variable, or a configuration file that other users do not have permission to read. This would be favored in place of argv process introspection and sanitization.

References:

How to Handle Secrets on the Command Line

INFORMATIONAL FINDINGS

I1: [Android] Network Security Configuration Allows Cleartext Communication to Servers in Older Versions of Android

Description:

The **Geph** Android application did not include a Network Security Configuration. As a result, the default values for network security options from the Android OS where the application was installed were used.

The default configuration for versions below Android 9 (API level 28) allows the application to communicate with servers in cleartext.

Impact:

Users with the application installed on Android 8.1 or lower are susceptible to a protocol downgrade attack, or to a Man-in-the-Middle (MitM) attack if the attacker has a way to force the application to navigate to a desired URL.

Note that in this case, the only HTTP traffic observed was between services running locally on the device, so the risk is considered nominal. This finding could present a risk if future code changes or other vulnerabilities cause the **Geph** application to communicate with remote servers over HTTP, so the assessment team has reported this finding for Informational purposes.

Reproduction:

This finding was observed by auditing the source code of the **Geph** Android application as well as the production APK and noting the absence of the **res/xml/network_security_config.xml** file.

Recommended Remediation:

The assessment team recommends adding a Network Security Configuration file with the **cleartextTrafficPermitted="false"** directive. Exceptions can be added if required.

References:

Network Security Config

PROTOCOL AND CRYPTOGRAPHY REVIEW

Protocol, Cryptography, and Architecture Overview

Geph introduces several new cryptographic protocols:

- **Binder Protocol** – used between binder and all other Geph components in order to authenticate or retrieve and update network topology data.
- **Mizaru** – used between the client and the binder in order to obtain an authentication token that has no link to the user's credentials.
- **Sosistab2** – the encryption and obfuscation protocol which wraps forwarded traffic.

Generally, these protocols build on modern and audited primitives. An important point about the architecture is that the binder servers are run by **Geph** and considered trusted – malicious behavior from these servers would break core trust assumptions of the network since they store a list of all users, exit servers, bridges, and associated cryptographic material. This is similar to exit servers: they are operated by **Geph** and contain keys that allow them to alter the topology of the network.

However, the threat model is not too straightforward as the Mizaru blind signature authentication protocol is designed to hide client identities from the binder (if just hiding identities from exit servers was desired, the scheme would not be necessary and the binder could just sign random tokens and hand those out to clients). The scheme aims to prevent accidental linkage of identities to traffic, or deanonymization of past sessions should the binder be compromised.

Meanwhile, the main job of the Binder protocol and Sosistab2 is to encrypt, authenticate, and obfuscate traffic to protect from network-level attackers.

Binder Protocol

The binder protocol is used in a number of network interactions as listed in the threat model. For instance, **Geph** clients initially connect to the binder by sending an authentication request using this protocol (via domain fronting). All clients have the hard-coded “master” public key of the binder, and when communicating with the binder, send an ephemeral public key followed by an encrypted serialized message. Keys are used in a x25519 Diffie Hellman key exchange to obtain a shared secret. ChaCha20Poly1305 encryption is then used, with a zero nonce, however a new ephemeral key was generated for every message. The main concern with the protocol is if the “master” secret key of the binder is ever compromised. Not only would it be difficult to update clients that have the hardcoded master public key; past captured messages could be decrypted since only one side of the key exchange uses ephemeral keys.

Mizaru

Mizaru is a cryptographic scheme designed to increase the privacy of users from **Geph** infrastructure. As part of the scheme, 65536 (2**16) RSA keypairs are generated on the binder server, one for each day after the UNIX epoch. A Merkle tree is created from hashes of the public keys. The RSA keys are used as part of a Full Domain Hash blind signature scheme. The majority of the code is located at **geph4-libs/mizaru/src/keypair.rs**.

When a user authenticates to the binder, they generate a string of random bytes and create a full domain hash digest from it. They then blind that digest and send it to the binder as part of the standard authentication procedure; the binder uses its daily Mizaru private key to blind sign the digest. The client receives the blinded signature back, verifies, and unblinds it. The user now has an authentication token, permitting either “free” or “plus” use of the service for up to 24 hours, with a signature from the binder, yet the binder did not see that token during the authentication process. The binder server verifies these tokens by checking the epoch,

ensuring that the used public key hash is in the correct position of the Merkle tree, and that the public key verifies the unblinded signature. The client- and server-side aspects of the protocol are implemented at **geph4-protocol/src/binder/client.rs** and **geph4-binder/src/bindercore_v2.rs**.

This authentication process is similar to the Privacy Pass framework proposed by Cloudflare. In practice, this protocol does have the problem of time-based client deanonymization, which is described in section 8.4 of the Privacy Pass paper and in the finding **[client] Time-Based Client Deanonymization**.

The assessment team observed that the third-party RSA-FDH library was used correctly according to its API docs. The assessment assumes that the library itself provides a secure and correct implementation of the scheme and notes that the library was not requested to be in scope for security audit.

Sosistab2 Symmetric Encryption

The Sosistab2 protocol relies on two symmetric schemes, NonObfsAead and ObfsAead. Both are based on ChaCha20-Poly1305. The main difference between them is that NonObfsAead does not hide the fact that it is encrypted data since it uses a sequentially consistent nonce, where ObfsAead uses a random nonce to obfuscate the encrypted payload. In NonObfsAead, only the first 8 bytes of a 12 byte nonce are used to form a u64 counter, which does decrease the entropy of nonces, but this should be sufficient in practice.

Sosistab2 Key Exchanges

Sosistab2 key exchanges are made between the **Geph** client and an exit server. Somewhat confusingly, there are actually two different key exchanges here at two different layers of the protocol, Obfuscated pipes and Multiplex. Obfuscated pipes are concerned with bypassing censorship and Multiplex is the higher layer, which has its own authenticated encryption but isn't concerned about obfuscation. A solid set of cryptographic primitives are used, with x25519 keypairs, Blake3 hashes and the AEAD ciphers described above. Where appropriate, labelling of keys derived via Blake3 hashes is used to prevent reuse of keys in the wrong context.

Sosistab2 Obfuscated Pipes

Obfuscated pipes (ObfsTlsPipe and ObfsUdpPipe) are the underlying layer which encrypt and obfuscate packets in such a way that they should not be distinguishable as **Geph** traffic. The Obfuscated pipe interface makes no promise about any security properties other than bypassing censorship (e.g. ObfsTlsPipe in practice uses unauthenticated self-signed TLS certs). Obfuscated pipes use the ObfsAead symmetric cipher to disguise the handshake as random bytes. Note that the key for the ObfsAead handshake encryption is derived on the exit server from the current time and a bridge-specific key (**geph4-exit/src/listen/control.rs** line 186). This appears to have been designed to make active-probing of bridges difficult, but note that a sufficiently-resourced network-level attacker which actively made legitimate **Geph** connections using all circuits could learn all such keys and would be able to deobfuscate captured handshake traffic made to these bridges. Active man-in-the-middle of application traffic sent over the connection is aimed to be prevented by the Multiplex transport layer one level higher.

Sosistab2 Multiplex

In the Sosistab2 Multiplex key exchange, clients and exit servers exchange long-term and ephemeral Diffie Hellman keys. Clients check that the long-term key sent by the exit server they are connecting to matches the one received from the binder for that server. The keys are used to calculate a “triple Diffie-Hellman” handshake, similar to that specified in the X3DH protocol. The X3DH protocol is simplified here due to the binder server being trusted to have an authentic long-term key for the exit server, and the fact that ephemeral keys can be generated online.

Replay protection for the Multiplex protocol consists of a “replay filter” on the nonces of the symmetrically encrypted messages, which drops messages that contain a recently-seen nonce or a nonce that does not appear in a window of 10,000 sequential values. There is potential here for a race condition of two identical messages being accepted since the filter is not synchronized, however in practice the encapsulated protocol is likely to detect this. The handshake itself is not authenticated nor timestamped, but the checking of the servers long-term public key means message forgery should produce an incorrect shared secret.

Assuming key material was deleted, compromise of a long-term exit servers private key should not compromise past sessions due to the use of ephemeral keys. However, sessions from the client with a particular exit server could last for long periods. A mechanism for periodic rekeying to prevent long-lived sessions from compromise would be a possible improvement.

Future Work
Within the scope of this timeboxed assessment, the team had time to validate the primitives and overall operation of the protocol by code review and dynamic testing. In future, the team recommends that a formal specification be written for the **Geph** cryptographic protocols, at which point a more in-depth review of protocol considerations could be carried out.

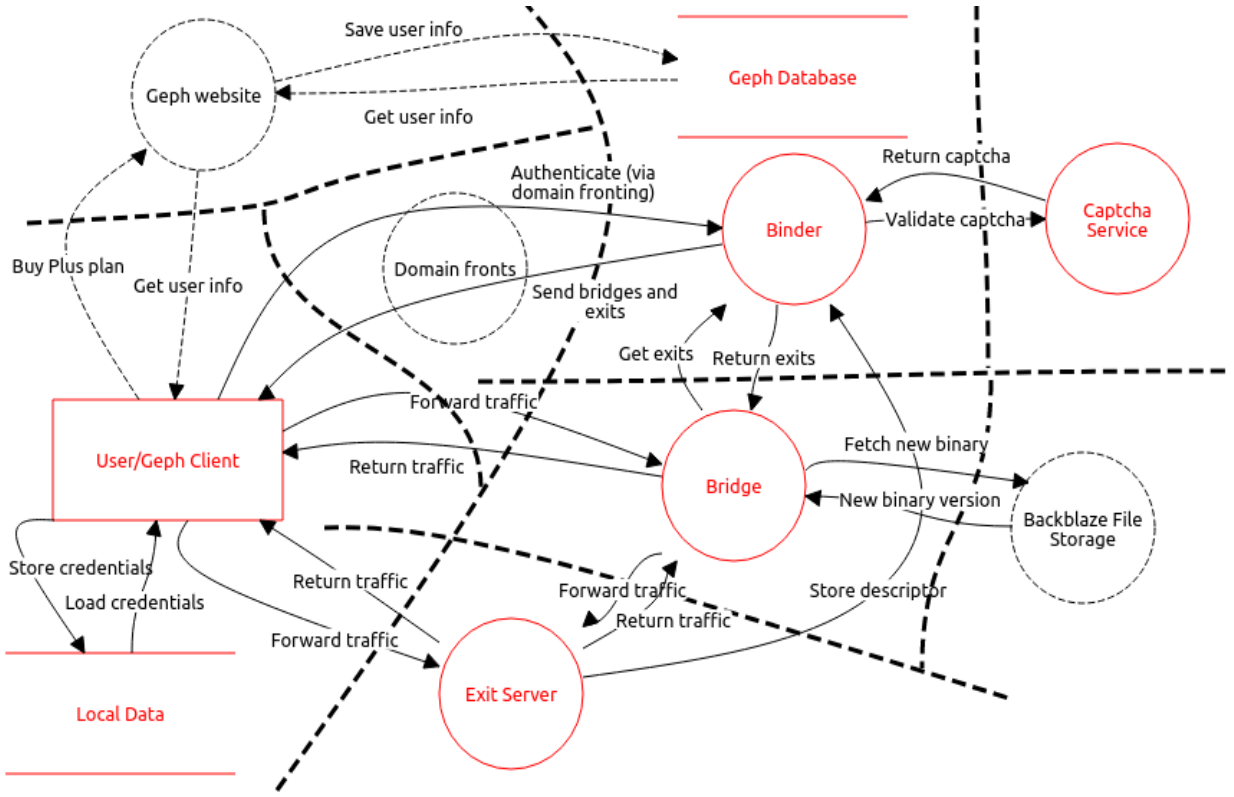
THREAT MODEL

To outline the attack surface and discover the applicable threats, the first step in drawing up **Geph's** threat model was identifying its main components, links, and trust boundaries, and consequently its security requirements.

Please note: What follows is the assessment team's interpretation of attack surface area and threats. Threat models are often subjective and two different expert security practitioners may have a difference in subjective thinking in how they are defined.

A useful tool to graphically depict this is a Data Flow Diagram (DFD). This type of diagram should assist analysts, helping them better understand the system and identify applicable threats using the STRIDE approach.

The team drew up the following diagram for **Geph** architecture:



Partial Application Decomposition

The following are the process entities involved in the architecture:

- **Bridge**: “Untrusted” part of Geph network which simply forwards traffic from client to exit to circumvent blocking of exit server IPs
- **Binder**: The binder service is the critical central server for the Geph network. It stores the private keys that authenticate all users, and communicates with all other Geph infrastructure elements
- **Exit Server**: Last part of Geph network which forwards traffic to requested servers
- **Captcha Service**: Web service running on Appspot that creates and validates captchas
- **Geph website**: Geph.io frontend, with download links and payment options for Plus subscription
- **Domain fronts**: External CDNS (e.g. Netlify) used to proxy request to blocked binder servers
- **Backblaze File Storage**: External servers hosting Geph binaries

Of these, the **Binder** servers are considered the main trusted component, but with some caveats – see the **Protocol and Cryptography Review** for a full discussion of this. **Bridges**, **Exit Servers**, the **Captcha Service**, and the **Geph website** are all operated by the **Geph** team, so are trusted to some extent, however the network could be recovered if they were compromised. In particular, **Bridges** are quickly spun up and down on AWS LightSail and just hold a single low-value secret and are designed not to have any insight into users' traffic other than metadata.

The following data stores were identified:

- **Geph Database**: Main database storing all Geph data: users, exits, bridges.
- **Local Data**: Local configuration files for client

The following actor object was identified, as the user in the system. This user is assumed to have a single use-case, which is to bypass Internet censorship:

- **User/Geph Client**: The user runs a copy of geph4-client locally, whether on desktop or mobile. This authenticates with the binder, retrieves a list of bridges and exits, then exposes interfaces (SOCKS/HTTP proxy/VPN) to allow traffic to be forwarded over the Geph network.

The following list details the data flows between entities:

- **Forward traffic**: Client can be configured to directly forward traffic to exit server without going via bridge (Sosistab2)
- **Forward traffic**: SOCKS proxy, HTTP proxy, or VPN sends client traffic to bridge over encrypted & obfuscated sosistab2 protocol (Sosistab2)
- **Forward traffic**: Bridge forwards client traffic to exit server using iptables rules (Sosistab2)
- **Validate captcha**: Binder validates captcha challenge against server (HTTPS)
- **Return captcha**: Binder fetches captcha challenge to send to client (HTTPS)
- **Send bridges and exits**: After authenticating with credentials, the client sends the binder its signed token to get a list of bridges and exits it can connect to (Bridge protocol)
- **Authenticate (via domain fronting)**: The first stage of a Geph connection; client sends an authentication message to the Binder using encrypted binder protocol, via a domain front (Binder protocol)
- **Store descriptor**: Exit server sends its data to the binder so it can be selected by the client (Binder protocol)
- **Save user info**: Website marks user as Plus user after payment details sent (HTTPS)
- **Buy Plus plan**: User can use Stripe to purchase subscription on website (HTTPS)
- **Get user info**: User details returned to frontend after authenticating on website (HTTPS)
- **Return traffic**: Bridge forwards client traffic to exit server using iptables rules (Sosistab2)
- **Return traffic**: SOCKS proxy, HTTP proxy, or VPN sends client traffic to bridge over encrypted & obfuscated sosistab2 protocol (Sosistab2)
- **Return traffic**: Client can be configured to directly forward traffic to exit server without going via bridge (Sosistab2)
- **Get user info**: User details returned to frontend after authenticating on website (HTTPS)
- **Store credentials**: User credentials cached to file on local filesystem.
- **Load credentials**: User credentials cached to file on local filesystem.
- **Get exits**: Bridge polls binder for exit servers to form connections with (Binder protocol)
- **Return exits**: Binder returns some number of exit servers to bridge (Binder protocol)
- **Fetch new binary**: Bridge runs a loop which periodically fetches new version from Backblaze and launches it (HTTPS)
- **New binary version**: New version of geph4-bridge executed on bridge (HTTPS)

Attacker Behavioral Summary

Since **Geph** is an anti-censorship tool, the main attackers that come to mind are nation-states implementing censorship. While it is often thought that nation-state adversaries are hopeless to defend against, in practice they may only deploy a limited number of resources to attacking a network such as **Geph**, meaning that raising the bar and denying easy ways to disrupt the network is an important goal to strive towards.

1. The attacker is assumed to be interested in the list of websites and metadata that users are accessing through the **Geph** network (e.g., through compromising an **Exit** server).
2. The attacker is also assumed to be interested in correlating personal identifiers (e.g., IP addresses) with particular traffic flows.
3. The attacker would like to gain system access to high-value servers such as the **Binder** service to access the database.
4. Similarly, an attacker would like to carry out a supply chain attack in order that either clients or servers are running backdoored code that sends data to the attacker.
5. The attacker would like to disable the network by a high traffic denial of service attack.
6. The attacker would alternatively like to disable the network by censoring all bridges and making it impossible for users to connect to a network entry point.

High Priority Threats and Mitigations

Threat	Component	Description	Mitigation
Sosistab2 Information Leak ***Information disclosure***	User/Geph Client (Actor)	Bug in the transport protocol causes request data or metadata to leak, identifying the user seeking censorship circumvention	**Increase testing of protocol on all platforms to ensure all traffic is obfuscated and encrypted**
Sosistab2 cryptography failure ***Information disclosure***	User/Geph Client (Actor)	A cryptographic failure (e.g. nonce reuse) allows decryption of traffic by passive adversary	**Cryptographic audit and testing**

Medium Priority Threats and Mitigations

Threat	Component	Description	Mitigation
Unencrypted requests ***Tampering***	User/Geph Client (Actor)	The Geph client makes requests over unauthenticated protocols (e.g. HTTP)	**All traffic goes over confidential and authenticated protocols**
Unauthenticated updates ***Tampering***	Bridge (Process)	The bridge fetches auto updates from external servers which aren't cryptographically signed, which could lead to an attacker executing code on bridge servers	**Cryptographically authenticate updates**

User deanonymized ***Information disclosure***	Binder (Process)	Geph network is able to track network requests for a particular identity	**Robust authentication protocol**
Membership upgrade ***Elevation of privilege***	Binder (Process)	User is able to upgrade to Plus plan without paying	**Separate logic to prevent membership upgrades**
Unauthenticated users ***Spoofing***	Binder (Process)	Users can connect to the Geph network without authentication	**All interactions require time-limited tokens**
Silent compromise of binder ***Tampering***	Binder (Process)	The binder service represents an attractive target to compromise via e.g. exploit in OS package, after which traffic would be monitored	**Intrusion and detection, EDR to catch persistent compromise**
User deanonymization*Information disclosure*	Exit Server (Process)	Exit server can deanonymize user	**Geph authentication protocol**
Modify user traffic ***Tampering***	Exit Server (Process)	Exit server modifies user's traffic	**HTTPS**
Spoofed exit server ***Spoofing***	Exit Server (Process)	An attacker is able to create a fake exit server on the network	**Private key to authenticate**
Bandwidth exhaustion ***Denial of service***	Bridge (Process) Exit Server (Process)	Exit servers can no longer pass new traffic due to many high bandwidth streams	**Ability to scale up exit servers and terminate/limit abusive traffic flows**
Replay attacks ***Tampering***	Exit Server (Process)	Attackers can replay captured traffic	**Monitor timestamp and nonces in connections**
Credential cracking ***Information disclosure***	Geph Database (Store)	See OWASP Automated Threat #7: Brute force, dictionary and guessing attacks used against authentication processes of the application to identify valid account credentials	**Defences include restriction of excessive authentication attempts, control of interaction frequency and enforcement of a single unique action**
Account creation ***Elevation of privilege***	Geph Database (Store)	See OWASP Automated Threat #19: Bulk account creation, and sometimes profile population, by using the	**Defences include control of interaction frequency, enforcement of a single unique a action and enforcement of behavioral workflow**

		application's account signup processes	

Low Priority Threats and Mitigations

Threat	Component	Description	Mitigation
Spoof bridge server ***Spoofing***	Bridge (Process)	An attacker is able to create a fake bridge server on the network	**Bridge private key**
DoS of binder servers ***Denial of service***	Binder (Process)	Geph network knocked over by repeated application of intensive operations on Binder servers.	**Anti-abuse mechanisms**
Captcha bypass ***Spoofing***	Captcha Service (Process)	Attackers are able to register accounts without legitimately solving captcha	**Remove bypasses and increase captcha difficulty**
User credentials exposed to all processes ***Information disclosure***	Local Data (Store)	Local users can learn credentials of other users via the command line / open permissions on config files	**Restrict permissions**

SOFTWARE DEVELOPMENT LIFECYCLE REVIEW

In this section, individual findings related to the **Geph** SDLC process are discussed and recommendations related to software development practices, secret management processes, build/signing/release processes, software updates, and potential automation solutions are included.

SDLC Background:
The SDLC is a structured process for creating high-quality software. A typical SDLC methodology includes the following phases of software development:

- Requirement analysis
- Design
- Development
- Testing
- Deployment
- Maintenance

A notable insight from industry SDLC research is that bugs or design flaws are easier and cheaper to fix early in the process, as compared to after software is deployed. This is particularly true in the case of security vulnerabilities, which can cause significant business impact if exploited. In particular, SDLC review with a focus on security aims to identify places where processes can be improved so that security vulnerabilities can be systematically caught long before reaching production.

Overview:
As a fully open-source project with a small team of maintainers, **Geph's** SDLC process was fairly informal and many typical SDLC recommendations which are aimed at larger organizations did not apply. The assessment team took these factors into consideration when performing this phase of the engagement and focused on recommendations that would provide a tangible security benefit at the present time as well as longer term improvements that would make the project more robust as it continues to mature.

Overall, the **Geph** developers paid careful attention to security concerns and followed many best practices regarding supply chain security, account protection and password hygiene. All user accounts were protected with TOTP Multi-Factor Authentication where possible, all passwords were randomly generated by a publicly audited password manager, and full disk encryption was leveraged on all workstations containing security relevant data for the project.

Despite these best practices, the assessment team identified some opportunities for improvement in the overall SDLC process. Automated source code analysis tools for Rust are continuing to mature, and other technologies used by the project such as Kotlin and JavaScript are well-supported. Tracking of vulnerable dependencies is relatively trivial to implement using tools like GitHub's **dependabot** or **cargo-audit** and would prevent vulnerabilities such as the **Out-of-Date Libraries in Use** finding in this report.

One of the most urgent recommendations involves integrity verification for application binaries and updates. Given that **Geph's** most significant adversaries are nation states, the assessment team believes it would be within their capability to tamper with precompiled binaries to compromise end users or server components such as bridges. Modifying the bridge's automatic update functionality and providing end users with high security needs the ability to verify the integrity of application code and would help mitigate these kinds of attacks. While the assessment team believes this recommendation to be urgent, it was considered a Medium-term recommendation due to the complexity involved in implementing it.

Overview of Recommendations:
The assessment team formulated eight recommendations for how **Geph** could further improve its SDLC. These recommendations have been divided into three suggested timeframes for implementation:

- Short-term goals, which can be implemented within a few months.
- Medium-term goals, which can be implemented in the next year.
- Long-term goals, which should be implemented as the project team matures or when higher level of security assurance is required.

Many of these goals are practices that should be completed periodically and/or at significant milestones.

SDLC Short-Term Goals

SDLC.S1. Increase Use of Static Analysis Tools
Static analysis tools scan codebases, matching code patterns against a library of rules to detect security hotspots and potential vulnerabilities. While they often produce false positives and do not catch more sophisticated vulnerabilities, they are invaluable at preventing several common bugs at an early stage of the SDLC. The ideal way to set up the tools is to run them on every pull request and commit as part of a CI/CD pipeline. Tools should also be configured carefully so that alerts are high signal and cannot be easily ignored (e.g., by preventing push if the CI/CD pipeline fails). **Geph** leveraged CircleCI to run unit tests and check for build errors when commits are pushed to GitHub, so extending the CI configuration to integrate tools such as **Semgrep** should be straightforward.

SDLC.S2. Publish Disclosure Policy for Reporting Security Concerns
The **Geph** project did not publish a security policy to guide developers and researchers on how to report security vulnerabilities. Publishing clear guidance on how to notify developers of security concerns helps streamline the process and, in some cases, may reduce the amount of time between vulnerability discovery and remediation. Public guidelines on contributing and reporting security vulnerabilities also help encourage contributions from the community, which can help expand the bandwidth for development of the project.

SDLC Medium-Term Goals

SDLC.M1. Track Vulnerable Dependencies
The **Geph** project tracked security vulnerabilities in libraries and software it uses on an ad-hoc basis.

This may lead to vulnerabilities that stem from outdated components, as discussed in the finding **Out-of-Date Libraries in Use** in this report. The **Geph** team should work to automate the tracking of security vulnerabilities in the software components its business depends on. Specifically for Rust, the **cargo audit** tool could be integrated in the CircleCI setup. Alternatively, GitHub's **dependabot** solution offers support for Rust **Cargo.lock** files.

SDLC.M2. Implement Integrity Verification for Application Updates
The bridge servers contained code to automatically update the software by downloading precompiled binaries from Backblaze B2, however no software integrity verification was performed on these updates. This could allow an attacker who has managed to compromise the B2 bucket or Backblaze account to deploy malicious code on the bridges. The impact in this scenario would mainly be limited to Denial-of-Service (DoS) since the **Geph** architecture treats bridges as untrusted.

An attacker could also target precompiled desktop clients (also hosted in B2) that are served to end users to compromise their devices as it wasn't possible to verify the integrity of the **Geph** desktop clients. Testing builds were distributed via Telegram, which provides another potential avenue for attack if an adversary were able to compromise the **Geph** developer's account or attempt social engineering attacks against users in the channel aimed at delivering them malicious versions of the client.

The assessment team recommends implementing signature verification on all software updates and aborting the update process if this verification process fails. Note that at the time of assessment, the binder and exit servers were manually updated, and the desktop clients simply instructed users to download the latest builds from Backblaze. Implementing automatic updates from within the application could handle integrity verification and also make it easier for users and servers to remain up to date with the latest versions of the software.

As a short-term mitigation, the assessment team recommends giving users the option of manually verifying the precompiled desktop binaries via GPG. This is often done by hosting a signed file containing the SHA256 hashes of the binaries that users can then verify with the maintainer's public key.

SDLC.M3. Protect Accounts With Hardware Two-Factor Authentication
The **Geph** team followed secure practices regarding password hygiene and Two-Factor Authentication (2FA) by opting for TOTP on a GrapheneOS-based mobile device and randomly generated unique passwords using Bitwarden. While TOTP is a significant improvement over email and SMS-based 2FA methods, this method is susceptible to phishing attacks as demonstrated in many high-profile breaches. An attacker who can phish a TOTP token could potentially compromise the **Geph** supply chain with malicious code targeting **Geph** infrastructure and end users.

As an improvement to TOTP, the assessment team recommends that **Geph** adopt FIDO2 hardware-backed 2FA. The FIDO2 protocol provides protection against phishing attacks by cryptographically authenticating the service the user is logging into. GitHub, AWS, and other providers used by the **Geph** team support FIDO2 2FA, and the team's Crates.io account would additionally be protected as the service uses GitHub for authentication.

SDLC Long-Term Goals

SDLC.L1. Regular Security Assessments
The assessment team recommends that consistent periodic security audits of new production code are integrated into the **Geph** SDLC.

Security assessments will inform and govern the other recommendations in this document. For every vulnerability that is discovered in an assessment:

- The vulnerability should be mitigated, or risk accepted.
- The **Geph** team should determine that no more vulnerabilities of its class exist, especially in code or infrastructure that was out of scope for review.
- Automated processes should be put in place to prevent vulnerabilities of this class from appearing again.

SDLC.L2. Incorporate Code Review Security Checklists Into Existing Manual Code Review Processes
A reliable code review process is one hallmark of a checks and balances system found within mature SDLCs. Regular code reviews are amongst the most powerful tools available for reducing defects during the software engineering process.

However, while code reviewers are generally aware of bad development practices, security defects are often subtle and complex. They can often require a wide array of knowledge that may not always be available to any particular code reviewer.

The assessment team's recommended improvement to this situation is to implement the use of checklists during the code review process. Checklists are proven to allow people and organizations to significantly reduce the burden of coping with overwhelming complexity. These checklists reduce the cognitive load even in projects like **Geph** where there is a small number of developers and only one reviewer, and the benefit will be multiplied if more maintainers join the project in the future.

Incorporating a security checklist, employed by every maintainer during their code reviews, can have a profound impact in preventing the most common types of security vulnerabilities from being deployed to production.

SDLC.L3. Sign All Git Commits and Tags
The Git version control system offers the ability to cryptographically sign commits and tags within a repository. These signatures allow developers and end users to verify that code was not maliciously added to the repository by an attacker who has managed to compromise a maintainer's GitHub account.

The assessment team recommends signing all commits and tags with the maintainer's private key. This would also increase security during the update process on the binder and exit servers, as these components were updated by manually checking out the repository and building from source locally.

APPENDICES

OWASP Mobile Top 10

Category	Description	Assessment Observations
M1. Improper Platform Usage	This category covers misuse of a platform feature or failure to use platform security controls. It might include Android intents, platform permissions, misuse of TouchID, the Keychain, or some other security control that is part of the mobile operating system. There are several ways that mobile apps can experience this risk.	Multiple findings related to Improper Platform Usage were identified in the **Geph** Android application. For example, the application stored user credentials in plaintext using the SharedPreferences API rather than the Android KeyStore. The application allowed backups, which could be exploited to obtain user credentials. The application was signed with the v1 signature scheme, exposing users with the application installed on older versions of Android to various attacks. Finally, **launchMode** setting in the application manifest enabled task hijacking attacks against users on vulnerable devices.
M2. Insecure Data Storage	This new category is a combination of M2 + M4 from Mobile Top Ten 2014. This covers insecure data storage and unintended data leakage.	One finding related to Insecure Data Storage was identified during the mobile application assessment. Specifically, the **Geph** Android application stored user credentials in plaintext using the SharedPreferences API, which could make it easier for an attacker with access to the device to compromise the user's account.
M3. Insecure Communications	This covers poor handshaking, incorrect SSL versions, weak negotiation, cleartext communication of sensitive assets, etc.	No findings related to Insecure Communication were identified during the mobile assessment. The **Geph** mobile applications leveraged various encrypted protocols to achieve obfuscation from censorship authorities.
M4. Insecure Authentication	This category captures notions of authenticating the end user or bad session management. This can include: • Failing to identify the user at all when that should be required • Failure to maintain the user's identity when it is required • Weaknesses in session management	No findings related to Insecure Authentication were identified during the mobile assessment. Authentication occurred over an encrypted protocol and the overall design did not leverage traditional session management.
M5. Insufficient Cryptography	The code applies cryptography to a sensitive information asset. However, the cryptography is	No findings related to Insufficient Cryptography were identified during this

	insufficient in some way. Note that anything and everything related to TLS or SSL goes in M3. Also, if the app fails to use cryptography at all when it should, that probably belongs in M2. This category is for findings where cryptography was attempted, but it wasn't done correctly.	assessment. The application used custom obfuscated and encrypted protocols for all outbound communication. Client side cryptography was handled by the native Rust client and covered as part of the Cryptography Review component of this assessment.
M6. Insecure Authorization	This is a category to capture any failures in authorization (e.g., authorization decisions in the client side, forced browsing, etc.). It is distinct from authentication findings (e.g., device enrolment, user identification, etc.). If the app does not authenticate users at all in a situation where it should (e.g., granting anonymous access to some resource or service when authenticated and authorized access is required), then that is an authentication failure not an authorization failure.	No findings related to Insecure Authorization were identified during the mobile assessment. As stated previously, the application did not use traditional session management and did not have separate user roles, other than premium accounts in addition to free accounts. The team attempted to tamper with local data to bypass these controls and gain access to premium features but was unsuccessful.
M7. Client Code Quality	This was the "Security Decisions Via Untrusted Inputs", one of our lesser-used categories. This would be the catch-all for code-level implementation problems in the mobile client. That's distinct from server-side coding mistakes. This would capture things like buffer overflows, format string vulnerabilities, and various other code-level mistakes where the solution is to rewrite some code that's running on the mobile device.	No findings related to Client Code Quality were identified during the mobile assessment. The mobile applications contained relatively minimal platform-specific code and instead relied on the client, written in Rust, to implement most of the logic.
M8. Code Tampering	This category covers binary patching, local resource modification, method hooking, method swizzling, and dynamic memory modification. Once the application is delivered to the mobile device, the code and data resources are resident there. An attacker can either directly modify the code, change the contents of memory dynamically, change or replace the system APIs that the application uses, or modify the application's data and resources. This can provide the attacker a direct method of subverting the intended use of the software for personal or monetary gain.	One finding related to Code Tampering was identified during the mobile assessment. Specifically, the v1 signature included in the Android application APK could allow attackers to inject malicious code which poses as an application update, and the verification of the update would succeed on older Android devices.
M9. Reverse Engineering	This category includes analysis of the final core binary to determine its source code, libraries, algorithms, and other assets. Software such as IDA Pro, Hopper, otool, and other binary inspection tools give the attacker insight into the inner workings of the application. This may be used to exploit other nascent vulnerabilities in the application, as well as revealing information about back end servers, cryptographic constants and ciphers, and intellectual property.	No findings related to Reverse Engineering were identified during the mobile assessment. All **Geph** components, including the mobile applications, are open source and freely available on GitHub, so the assessment team considered this class of vulnerability out-of-scope for the mobile assessment.

M10. Extraneous Functionality	Often, developers include hidden backdoor functionality or other internal development security controls that are not intended to be released into a production environment. For example, a developer may accidentally include a password as a comment in a hybrid app. Another example includes disabling of 2-factor authentication during testing.	One finding related to Extraneous Functionality was identified during the mobile assessment. The iOS and Android applications both exposed an RPC server to other applications on the device over a TCP socket. This allows malicious applications to query information about the current VPN tunnel and to totally shut off the tunnel. This could in-turn lead to Denial-of-Service (DoS) or disclosure of the user's identity to adversaries in hostile network environments.