Security Assessment of VPN Generator's Application & Cryptography Architecture Review

OPEN TECHNOLOGY FUND

TABLE OF CONTENTS

Executive Summary 3
- Include Security (IncludeSec) 3
- Assessment Objectives 3
- Scope and Methodology 3
- Findings Overview 3
- Next Steps 3

Risk Categorizations 4
- Critical-Risk 4
- High-Risk 4
- Medium-Risk 4
- Low-Risk 4
- Informational 4

Low-Risk Findings 5
- L1: Secret Stored in Source Code Repository 5
- L2: Strict Host Key Checking Disabled 5
- L3: Telegram Chat IDs Stored in Database 7
- L4: Hosts Did Not Perform Automatic Updates 8

Appendices 10
- Statement of Coverage 10
- A1: Architecture Review 10

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

The objective of this 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 VPN Generator could implement to secure its applications and systems.

Scope and Methodology

Include Security performed a security assessment of VPN Generator's Application & Cryptography Architecture Review. The assessment team performed a 12 day effort spanning from Jul 11, 2023 – Jul 28, 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 a total of 4 findings. There were 0 deemed to be "Critical-Risk," 0 deemed to be "High-Risk," 0 deemed to be "Medium-Risk," and 4 deemed to be "Low-Risk," which pose some tangible security risk. Additionally, 0 "Informational" level findings were identified that do not immediately pose a security risk.

IncludeSec encourages VPN Generator 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 VPN Generator 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 VPN Generator 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.

LOW-RISK FINDINGS

L1: Secret Stored in Source Code Repository

Description:

An SSH private key was discovered in the codebase. Hardcoding credentials into the source code exposes security-relevant information to several people, including developers, administrators, and potentially other stakeholders. This practice can make controlling the data and managing access difficult, if not impossible. The artifacts may be stored (in addition to the repository) in other locations, such as on developers' laptops.

Impact:

Credentials committed together with the source code can remain in the repository for a long period of time, and even when deleted at some point, it can often still be possible to extract them from the repository's revision history. This means that any employee with access to the repository (currently, in the past, or in the future) could obtain control over various and perhaps critical parts of **VPN Generator** infrastructure, compromising the company and bringing risk to customer data.

A secret was identified in the codebase at the following location:

File	Description
ministry/conf/id_ecdsa	**Ministry Private Key**

Reproduction:

The following snippet from file **ministry/conf/id_ecdsa** shows a hardcoded SSH key used for administrating the **Ministry** server:

```
-----BEGIN OPENSSH PRIVATE KEY-----
b3BlbnNza[...]
```

Recommended Remediation:

The assessment team recommends invalidating all credentials and other secrets stored in the version history and implementing a secrets management service, such as Hashicorp Vault or AWS Secrets Manager, to retrieve credentials dynamically without checking them into version control.

If this is not possible, the assessment team recommends removing all confidential information from the git history (see References) and to rotate as many secrets as possible.

References:

Github: Removing Sensitive Data from a Repository
Git-secrets: Prevent Committing Secrets to the Repository
Hashicorp Vault
AWS Secrets Manager

L2: Strict Host Key Checking Disabled

Description:

Across the infrastructure, **VPN Generator** used SSH to perform management operations. SSH follows a trust-on-first-use (TOFU) model where when the first time an SSH client connects to a server, the server's host key is stored in the client's known hosts file. On subsequent connections, the client ensures that the server's host key still matches what is stored in this file, and if not, the connection is aborted. In **VPN Generator**, SSH's default strict host key checking was explicitly disabled when SSH connections were made.

Impact:

With the SSH strict host key checking setting disabled, SSH clients do not verify the host key of the host being connected to against the keys in the known hosts list. This means that, in the event of a server being replaced by an attacker, potentially through tampering of network traffic, connecting clients would ignore the changed fingerprint and connect to malicious servers.

The following instances of SSH connections being made with SSH strict host key checking disabled were found:

File	Line Number
dc-mgmt/cmd/stats-sync.sh	**24**
dc-mgmt/cmd/vpn-works-keydesks-sync.sh	**50**
keydesk/crutches/may1apr1/convert.sh	**10**
ministry/scripts/purge_never_visited.sh	**15, 40**
dc-mgmt/cmd/replacebrigadier/main.go	**319**
dc-mgmt/internal/kdlib/ssh.go	**42**
embassy-tgbot/ssh.go	**38**
ministry/cmd/checkbrigadier/main.go	**421**
ministry/cmd/createbrigade/main.go	**447**
partner-api/embapi/ssh.go	**34**

Reproduction:

As an example, at lines 313-321 of file **dc-mgmt/cmd/replacebrigadier/main.go**, an SSH configuration was created with the **ssh.InsecureIgnoreHostKey()** function set:

```
config := &ssh.ClientConfig{
                User: sshkeyRemoteUsername,
                Auth: []ssh.AuthMethod{
                        ssh.PublicKeys(signer),
                },
                // HostKeyCallback: ssh.FixedHostKey(hostKey),
                HostKeyCallback: ssh.InsecureIgnoreHostKey(),
                Timeout:        sshTimeOut,
        }
```

Recommended Remediation:

The assessment team recommends not disabling SSH strict host key checking. When hosts are provisioned, their host keys could be saved and loaded into connecting host's known hosts files using a tool such as **ssh-keyscan**. Alternately, SSH host key fingerprints could be stored in a centralized database which is periodically pulled from by management hosts.

References:

SSH Stricthostkeychecking
Managing Your SSH known_hosts Using Git

L3: Telegram Chat IDs Stored in Database

Description:

The **Embassy** service ran the **VPN Generator Telegram** bot that users communicated with to generate a brigade. The bot needed to store information about the current state of chats to select which message to send next. The bot was found to store **Telegram** Chat IDs in its database. The **Telegram** Chat IDs were identical to the **Telegram** User IDs of **VPN Generator** users in this context.

Impact:

To increase privacy of **VPN Generator** users, a minimal amount of information about them should be stored. Compromise of the **Embassy** service or its **BadgerDB** store would reveal the **Telegram** User IDs that had created brigades in the past three days on **VPN Generator**. With a list of User IDs obtained, by joining common groups it would be possible to link those User IDs to identities of real people using **Telegram**.

Reproduction:

The **setSession()** function at line 51 of file **embassy-tgbot/session.go** was used to serialize chat sessions of users. The **sessionID()** function was used as the database key:

```
func setSession(dbase *badger.DB, chatID int64, msgID int, update int64, stage int, state int, payload []byte)
error {
        session := &Session{
                OurMsgID:   msgID,
                Stage:      stage,
                UpdateTime: update,
                Payload:    payload,
        }

        data, err := json.Marshal(session)
        if err != nil {
                return fmt.Errorf("parse: %w", err)
        }

        key := sessionID(chatID)
        err = dbase.Update(func(txn *badger.Txn) error {
```

The **sessionID()** function at line 40 of the same file contained a salt and digest mechanism; however, the direct SHA256 hash of Chat IDs was stored in the key, and the salt was not used in a cryptographic way:

```
func sessionID(chatID int64) []byte {
        var int64bytes [8]byte

        binary.BigEndian.PutUint64(int64bytes[:], uint64(chatID))

        digest := sha256.Sum256(int64bytes[:])
        id := append([]byte(sessionPrefix), append([]byte(sessionSalt), digest[:]...)...)

        return id
}
```

Line 14 of the file showed the constant salt and prefix values:

```
const (
        sessionSalt   = "$Rit5"
        sessionPrefix = "session"
)
```

In messages with **Telegram** bots, the Chat IDs were identical to the **Telegram** User IDs of the users communicating with the bot. This was confirmed dynamically by inspecting the **Embassy** logs:

```
[i] User:  ChatID: 264670827 Message: /start
```

This Chat ID was the same as the assessment team's User ID which sent the message.

Since Chat IDs are a nine-digit number, it is straightforward to generate the SHA256 hash of all possible numbers. This requires calculating a billon SHA256 hashes; an Nvidia GTX 1080 GPU can calculate the whole list in less than a second.

With a lookup table precalculated, any **Embassy** session key could be mapped back to a User ID in **Telegram**. While there was no **Telegram API** to directly show the account associated with a User ID, there were several ways in the documentation to obtain more information about users by knowing their ID, such as the getUserProfilePhotos API.

Recommended Remediation:

The assessment team recommends using a HMAC construction if Chat IDs must be stored. The identifiers would be hashed in the **sessionID()** function using a key known only to the application and not the database. Then, compromise of the database would not lead to the Chat IDs being reversable.

References:

Telegram API: Available Types
What is Difference between `msg.chat.id` and `msg.from.id` in Telegram Bot?
Tapping Telegram Bots

L4: Hosts Did Not Perform Automatic Updates

Description:

Control and endpoint servers in the **VPN Generator** infrastructure did not automatically install security updates, as the **unattended-upgrades** package was not installed on those servers. Administrative and management servers such as **Ministry** did have the **unattended-upgrades** package installed. The **unattended-upgrades** package automatically retrieves and installs security patches and other essential upgrades for servers with the package installed.

Impact:

Security vulnerabilities are frequently published for components such as the Linux kernel, and this could expose **VPN Generator** to risk of public exploits if not patched for several months.

Reproduction:

When authenticating to the staging endpoint virtual machine, the assessment team observed that 117 security updates were missing:

```
# ssh -o StrictHostKeyChecking=no -i ~/.ssh/id_ecdsa_staging ubuntu@10.255.0.5
[...]

202 updates can be applied immediately.
117 of these updates are standard security updates.
```

The assessment team checked for packages, and the **unattended-upgrades** package was not installed:

```
ubuntu@staging-vm-ep-0:~$ dpkg -l | grep unattended
rc  unattended-upgrades                   2.8ubuntu1                              all          automatic
installation of security upgrades
```

Administrative servers such as **Ministry** did have the **unattended-upgrades** package installed, but they did not have a cronjob or other method by which to reboot the server after a Linux kernel, systemd, or other system update had been applied.

Recommended Remediation:

The assessment team recommends installing the **unattended-upgrades** package by default on all servers, which regularly installs security updates. The assessment team also recommends adding a crontab to reboot the servers on some cadence to ensure the latest kernel security updates are applied.

An example of such a crontab that reboots on every first Sunday of the month is shown below, if the **unattended-upgrades** package has flagged that a reboot is required:

```
0 9 1-7 * */7  [ -f /var/run/reboot-required ] && reboot
```

References:

AutomaticSecurityUpdates
Schedule Cronjob for the First Monday of Every Month, the Funky Way

APPENDICES

Statement of Coverage

The **VPN Generator** network was subjected to a grey box application assessment. **VPN Generator** is a tool which aims to make VPN access easy for people in countries where Internet censorship is widespread. By messaging a Telegram bot, a user can obtain a working Wireguard configuration. When connected, additional Wireguard configurations can be created for friends and family members of the user.

The assessment team performed a source code review of all relevant code, as well as dynamic testing of a staging environment, with a focus on network architecture. The following source code repositories were reviewed:

- cert-vpn-works-builder
- control-endpoint-vms-deploy
- dc-mgmt
- dc-vpnapi-access
- embassy-tgbot
- encrypted-logger
- endpoint-setup-files
- keydesk
- keydesk-backup
- keydesk-spawner-access
- keydesk-stats-access
- keydesk-web
- ministry
- partner-api
- vpngine
- wordsgens

Exclusions
At the time of the assessment, code to set up ipsec VPNs was being added; however, this code was incomplete and not part of the application yet, so the assessment team did not review it. Similarly, the Partner API was in development; the assessment team recommends another review when it is finished since it will be a significant public-facing component of the system.

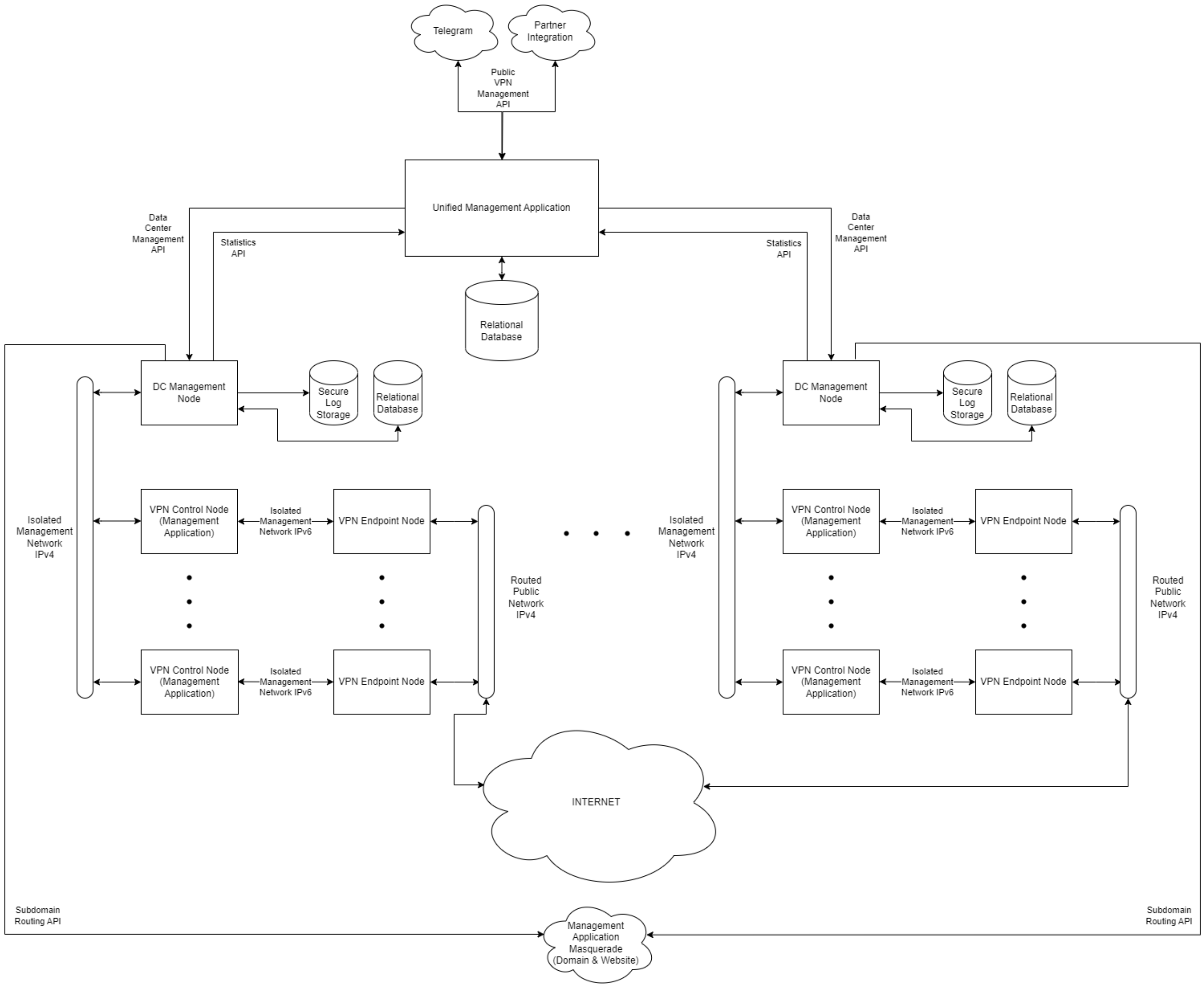
A1: Architecture Review

The **VPN Generator** architecture was found to be comprised of five main layers:

- A Telegram bot and “Partner API” frontend which were the public-facing entry points to **VPN Generator**.
- A set of backend management and provisioning services (**Ministry**) that administered all **VPN Generator** network deployments.
- An individual datacenter management plane, with a management node and log storage for administrator access only.

- Within each datacenter, individual **VPN Generator** network deployments, each with a control and endpoint node.
- Multiple “brigades” hosted on each control-endpoint node pair; a brigade was controlled by a “brigadier” and was defined as a collection of VPN configurations associated to a group of users who knew each other.

The high-level architecture is shown in the following image:


Telegram
Partner Integration
Public VPN Management API
Unified Management Application
Data Center Management API
Statistics API
Statistics API
Data Center Management API
Relational Database
DC Management Node
Secure Log Storage
Relational Database
DC Management Node
Secure Log Storage
Relational Database
Isolated Management Network IPv4
VPN Control Node (Management Application)
Isolated Management Network IPv6
VPN Endpoint Node
Routed Public Network IPv4
Isolated Management Network IPv4
VPN Control Node (Management Application)
Isolated Management Network IPv6
VPN Endpoint Node
Routed Public Network IPv4
VPN Control Node (Management Application)
Isolated Management Network IPv6
VPN Endpoint Node
VPN Control Node (Management Application)
Isolated Management Network IPv6
VPN Endpoint Node
INTERNET
Subdomain Routing API
Subdomain Routing API
Management Application Masquerade (Domain & Website)

The workflow for a user wishing to browse the Internet anonymously was established as follows:

1. The user would communicate with the **VPN Generator** Telegram bot, sending it a photo of a receipt to indicate the user was real.
2. A member of the **VPN Generator** team would approve the request and create a deployment (“brigade”) for that user.
3. The Telegram bot would then send an individual Wireguard config file to the user.

4. The first user of the brigade would then be identified as a “brigadier” who, when connected to the VPN endpoint, could also access the **Keydesk** dashboard running on the control node.
5. The brigadier could then access the **Keydesk** dashboard to add new users to the brigade; each of these new users would get their own Wireguard config. These subsequent users created could not connect to the **Keydesk** dashboard; they could only use the VPN endpoint.

Therefore, the key requirements of this architecture were identified as follows:

1. It should not be possible to breach user privacy by determining who is accessing content using **VPN Generator**.
2. An external user should only be able to interact with the Telegram bot and Partner API, as nothing else was intended to be public facing.
3. Users must not be able to access any management functionality intended for **VPN Generator** administrators, including datacenter management nodes or the backend management application.
4. Non-brigadier users should not be able to access the **Keydesk** dashboard.
5. Users in a brigade must not be able to interfere or tamper with the experience of users in other brigades, even if hosted on the same node.

Each of these points is addressed in a separate section below.

#1. It should not be possible to breach user privacy by determining who is accessing content using VPN Generator.

Initial setup on **VPN Generator** was performed by messaging a Telegram bot. This raises some privacy concerns, as messages between users and bots in **Telegram** are not end-to-end encrypted. In **Telegram**, only secret chats between users are end-to-end encrypted using Telegram's MTProto 2.0 protocol.

This would mean that individuals who can access Telegram servers could see a list of Telegram users who have signed up to use **VPN Generator** as well as those users' Wireguard configurations. To prevent this, a different chat service that enforces end-to-end encryption in all communications (such as Signal) would have to be used; otherwise, a scheme to add encryption on top of chat messages to the Telegram bot would be needed (requiring third-party software). Neither are attractive options, as one of the core goals of the project is to provide easy VPN access to primarily Russian-speaking users.

Adding to this, **VPN Generator** infrastructure itself stored Telegram user identifiers, creating another central point where VPN users could be potentially linked to real-world identities. This is elaborated on further in the finding **Telegram Chat IDs Stored in Database**. Finally, as shown in previous security research, if the Telegram bot's API key is ever disclosed, this information could be used together with Telegram Chat IDs to replay past user conversations. This means there is a third-party record of Telegram IDs linked to the IP addresses of VPNs they have used. Ideally this linkage should be never be permanently stored but that is not possible with the current Telegram setup.

Aside from this, within the **VPN Generator** infrastructure itself, a small number of statistics were collected about users. A statistics service running on control nodes regularly fetched Wireguard traffic statistics, such that the amount of daily, weekly, monthly, and yearly bytes used were known and stored in the control node's brigade database. Additionally, a Zabbix agent ran on each VPN endpoint, and the control node forwarded traffic from the management node to that Zabbix agent on the endpoint node; however, it did not appear that this configuration was used to gather any user-specific data at the time of assessment.

Beyond this, no user traffic was found to be logged, and the assessment team did not find any user's “real” IP address to be logged anywhere. Though, as with all VPN providers, an element of trust in the administrators of the service is required to assume they will not introduce user monitoring at some point in time.

*Recommendations:*The assessment team recommends reconsidering the use of Telegram as the entry point to the **VPN Generator** network, as messages are known to not be end-to-end encrypted; this lack of end-to-end encryption creates a privacy risk and vulnerability to the whole architecture of the application.

#2. An external user should only be able to interact with the Telegram bot and Partner API, as nothing else was intended to be public facing.

The assessment team was given access to staging environment versions of all key servers in the **VPN Generator** architecture. All network links and iptable rules were checked to ensure that network segments were enforced correctly. No servers were found to have public-facing IP addresses besides endpoint nodes.

Endpoint nodes did host a publicly accessible nginx page, as shown below:

Request:

```
GET / HTTP/1.1
Host: 195.133.0.116
```

Response:

```
HTTP/1.1 200 OK
Server: nginx/1.18.0 (Ubuntu)
Date: Fri, 28 Jul 2023 14:41:07 GMT
Content-Type: text/html
Content-Length: 612
Last-Modified: Fri, 28 Jul 2023 14:41:07 GMT
Connection: keep-alive
ETag: "f1cd6dce"
Accept-Ranges: bytes

<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
[...]
```

Upon further investigation, the assessment team found this to be a deliberate ploy to perhaps distract potential attackers. The page was not being served by nginx, but by netcat, as seen in **endpoint-setup-files/etc/systemd/system/fakehttp-ns@.service**:

```
[Unit]
Description=FakeHTTPs for namespaced interface %I
Requires=network.target
After=network.target
StartLimitIntervalSec=0

[Service]
Type=simple
Restart=always
Environment=FAKEPAGE=KLUv/QRY7[...]
Environment="ifwg=%i"
ExecStartPre=/bin/bash -c 'shuf -i 0-1 -n 1 > /tmp/fakehttps-%i-random-seed'
ExecStartPre=/bin/bash -c "/usr/bin/ip netns exec ns${ifwg##*:} iptables -A INPUT -i ${ifwg%%:*} -p tcp --dport 80
-j ACCEPT || true"
ExecStart=/bin/bash -c "while true; do if [[ `cat /tmp/fakehttps-%i-random-seed` == "0" ]]; then echo -n ; else
echo $FAKEPAGE | base64 -d | unzstd | sed \"s/_ETAG_/`openssl rand -hex 4`/g\" | sed \"s/_DATE_/`date -u '+%%a, %%d
%%b %%Y %%H:%%M:%%S GMT'`/g\" ; fi | /usr/bin/ip netns exec ns${ifwg##*:} timeout 10 nc -q 1 -nl -s `/usr/bin/ip
netns exec ns${ifwg##*:} /usr/bin/ip -4 -o a | fgrep ${ifwg%%:*} | cut -d \  -f 7 | cut -d \/ -f 1` -p 80 ; done"
ExecStopPost=rm -f /tmp/fakehttps-%i-random-seed
```

```
ExecStopPost=/bin/bash -c "/usr/bin/ip netns exec ns${ifwg##*:} iptables -D INPUT -i ${ifwg%%:*} -p tcp --dport 80
-j ACCEPT || true"

[Install]
WantedBy=multi-user.target
```

However, the assessment team questioned the effectiveness of this technique. The unusual hosting setup (for instance, port 443 was open but returned a specific SSL error) and the timing variances between real nginx servers and the custom one described here could create additional data points by which all **VPN Generator** endpoints could be targeted for blocking by active censorship.

The **VPN Generator** team said that the Partner API running on **Ministry** would also be exposed publicly in production. The Partner API was limited in functionality at the time of assessment, with a single handler function, **PostAdminHandler()**, at line 363 of file **partner-api/cmd/embsrv/main.go**:

```
api.PostAdminHandler = operations.PostAdminHandlerFunc(func(params operations.PostAdminParams, principal
interface{}) middleware.Responder {
                return embapi.AddAdmin(params, principal, sshConfig, addr)
        })
```

The **AddAdmin()** function in file **partner-api/embapi/admin.go** did not use any parameters provided by the user—all VPN configuration was generated server-side, including the username, which was that of a random Nobel prize winner. Further, access to the API required a JWT which had to be generated by the **VPN Generator** team.

A web application assessment was performed of the **Keydesk** application, which brigadiers could access once connected to their specified Wireguard network. To some extent **Keydesk** could be considered an external attack surface, since anyone could access it after chatting with the Telegram bot and having their request to use **VPN Generator** approved. The assessment team looked for opportunities to tamper with the data of other brigadiers or to escalate privileges to other points in the network. Again, the attack surface was found to be small as the application was designed with a minimal set of functionalities, with the following routes as observed on lines 527-542 of file **keydesk/cmd/keydesk/main.go**:

```
api.PostTokenHandler = operations.PostTokenHandlerFunc(keydesk.CreateToken(brigadeID, TokenLifeTime))

        api.PostUserHandler = operations.PostUserHandlerFunc(func(params operations.PostUserParams, principal
interface{}) middleware.Responder {
                return keydesk.AddUser(db, params, principal, routerPublicKey, shufflerPublicKey)
        })

        api.PostUserngHandler = operations.PostUserngHandlerFunc(func(params operations.PostUserngParams, principal
interface{}) middleware.Responder {
                return keydesk.AddUserNg(db, params, principal, routerPublicKey, shufflerPublicKey)
        })

        api.DeleteUserUserIDHandler = operations.DeleteUserUserIDHandlerFunc(func(params
operations.DeleteUserUserIDParams, principal interface{}) middleware.Responder {
                return keydesk.DelUserUserID(db, params, principal)
        })

        api.GetUserHandler = operations.GetUserHandlerFunc(func(params operations.GetUserParams, principal
interface{}) middleware.Responder {
                return keydesk.GetUsers(db, params, principal)
        })

        api.GetUsersStatsHandler = operations.GetUsersStatsHandlerFunc(func(params operations.GetUsersStatsParams,
principal interface{}) middleware.Responder {
                return keydesk.GetUsersStats(db, params, principal)
        })
```

All routes were found to require JWT authentication by the brigadier. API calls eventually led to system commands; however, almost all of the parameters interpolated into those commands were generated server-side. The assessment team did not find ways for attacker input to flow from source to sink.

Of the route handlers, the **DeleteUserUserIDHandler()** function was the only one that accepted user input parameters, i.e., a validated UUID; but even if it was not validated, the string parameters were base32- or base64-encoded before being used in the **endpoint-setup-files/wg-mng.sh** management script. For instance, on lines 41-48 of file **keydesk/vpnapi/wgvpn.go** the **WgPeerDel()** function base64 encoded the public key of the user to be deleted:

```
// WgPeerDel - peer_del endpoint-API call.
func WgPeerDel(actualAddrPort, calculatedAddrPort netip.AddrPort, wgPub, wgIfacePub []byte) error {
	query := fmt.Sprintf("peer_del=%s&wg-public-key=%s",
		url.QueryEscape(base64.StdEncoding.WithPadding(base64.StdPadding).EncodeToString(wgPub)),
		url.QueryEscape(base64.StdEncoding.WithPadding(base64.StdPadding).EncodeToString(wgIfacePub)),
	)

	_, err := getAPIRequest(actualAddrPort, calculatedAddrPort, query)
```

Recommendations: The assessment team determined the external attack surface to be minimal and encourages the **VPN Generator** team to continue the practice of accepting as little external user input as possible when creating and configuring VPN connections. However, the assessment team suggests evaluating the **fakehttp** service running on endpoints to determine if it is effective as a disguise.

#3. Users must not be able to access any management functionality intended for administrators, including datacenter management nodes or the backend management application.

The assessment team examined the mechanisms used to perform administrative and logging functions across the architecture. One example of a logging function was found where the host running the **Embassy** bot also ran a statistics service to get information from control nodes about the number of remaining slots for new VPN users. This was performed by SSH'ing into control nodes from the **Embassy/realm** host with the **marina** user. The **marina** user's SSH key was added to control nodes in file **keydesk-stats-access/debpkg/nfpm.yaml**:

```
contents:
- dst: /home/_marina_/.ssh
  type: dir
  file_info:
    mode: 0700
    owner: _marina_
    group: _marina_
- src: authorized_keys
  dst: /home/_marina_/.ssh/authorized_keys
  file_info:
    mode: 0400
    owner: _marina_
    group: _marina_
```

The **authorized_keys** file template at **keydesk-stats-access/debpkg/examples/keydesk_stats_access_authorized_keys.examples** limited the user to running only the **/opt/vgkeydesk/ssh_stats_command.sh** command over SSH:

```
command="/opt/vgkeydesk/ssh_stats_command.sh ${SSH_ORIGINAL_COMMAND}",no-port-forwarding,no-X12-forwarding,no-
agent-forwarding,no-pty ecdsa-sha2-nistp256
AAAAE2VjZHNhLXNoYTItbmlzdHAyNTYAAAAIbmlzdHAyNTYAAABBBKiNrvFERQPcvvSMC8RuHcRtrH9tnkUO1ltMmC0zjcPxJ+XJzajVk1t/YpGQ7Uf
uxAy/WtHxn21DDJvrYl9l1lk= phil@office
```

As shown on lines 22-26, **keydesk/cmd/sshcmd/ssh_stats_command.sh** prevented command injection by quoting the arguments and ensuring that only the **fetchstats** command could be run remotely:

```
if [ "xfetchstats" = "x${cmd}" ]; then
        ${basedir}/fetchstats "$@"
else
    echo "Unknown command: ${cmd}"
    printdef
fi
```

Besides the **marina** user, the **serega** user was used to add and remove brigades on the control node, and the **valera** user was used by the **Ministry** server to run commands on the **Embassy** server.

The assessment team checked that internal administrative servers could not be reached directly through the Wireguard network. The endpoint network configuration bound the incoming Wireguard interface to an interface **ens161** inside the network namespace:

```
root@staging-vm-ep-0:/home/ubuntu# ip netns exec nswg1 ip r
default via 195.133.0.113 dev ens161
100.64.0.0/24 dev wg1 proto kernel scope link src 100.64.0.40
195.133.0.112/29 dev ens161 proto kernel scope link src 195.133.0.116
```

The **ens161** link was not connected to other entities in the **VPN Generator** network architecture.

```
root@staging-vm-ep-0:/home/ubuntu# ip netns exec nswg1 ip a
[...]
11: ens161: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 00:50:56:01:28:54 brd ff:ff:ff:ff:ff:ff
    altname enp4s0
    inet 195.133.0.116/29 scope global ens161
       valid_lft forever preferred_lft forever
    inet6 fe80::250:56ff:fe01:2854/64 scope link
       valid_lft forever preferred_lft forever
```

The iptables rules on the control endpoint also aimed to prevent any Wireguard traffic from accessing the rsyslog or apt cache services listening on ports 514 and 3142 (see **control-endpoint-vms-deploy/setup-files-ct/etc/iptables/rules.v6**):

```
[...]
-A INPUT ! -i wg+ -s fc00::/7 -p tcp --dport 3142 -m state --state NEW -j ACCEPT
-A INPUT ! -i wg+ -s fc00::/7 -p tcp --dport 514 -m state --state NEW -j ACCEPT
[...]
```

However, the assessment team did note opportunities for improved network segmentation. Endpoint nodes were capable of making SSH connections to other nodes on the same subnet, such as 10.255.0.3, the management node:

```
root@staging-vm-ep-0:/home/ubuntu# nc 10.255.0.3 22
SSH-2.0-OpenSSH_8.9p1 Ubuntu-3ubuntu0.3
```

In practice, an endpoint node should never make SSH connections to other services this way, so to mitigate the potential of lateral movement, management services are recommended to be firewalled off unless accessed from a higher-privileged host. Additionally, network alerts would help detect if such an event was occurring as it would be a strong signal that a segment of the network had been compromised.

Returning to the **Embassy/realm** host, this host was found to perform several duties, and the architecture could be improved by splitting the duties up. At the time of assessment, the host ran the Telegram bots, hosted a database, and collected stats from all control endpoints, among other duties. Ideally, the same service hosting the bots would not be able to SSH into other parts of the infrastructure and would authenticate to an API whenever it needed to make infrastructure changes. The assessment team understands the **Ministry** service is currently being developed to achieve this goal.

Recommendations: The assessment team suggests reviewing firewall rules to allow only expected SSH and management traffic interactions. Network monitoring tools installed on management machines would help detect breaches if they occurred. Hosts with user-facing functionality (e.g., bots) are recommended to be separate from hosts running databases or performing management operations.

#4. Non-brigadier users should not be able to access the Keydesk dashboard.

The assessment team checked dynamically and found that non-brigade users could not load the **Keydesk** dashboard. The IPv6 address of the **Keydesk** dashboard for the assessment team's brigade could be seen by making a DNS lookup for **vpn.works**:

```
$ host vpn.works
vpn.works has address 0.0.0.0
vpn.works has IPv6 address fd22:b00c::e41f
```

The endpoint server's interface for routing requests to this address was linked to the **nswg1** network namespace.:

```
19: wg1veth0@if18: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether b6:0a:6d:29:cf:c4 brd ff:ff:ff:ff:ff:ff link-netns nswg1
    inet6 fd22:b00c::e41f/128 scope global
       valid_lft forever preferred_lft forever
    inet6 fe80::d49d:63ff:fe46:bed4/64 scope link
       valid_lft forever preferred_lft forever
```

The assessment team aimed to identify the access control mechanism that was allowing brigadiers to send packets through this network namespace while restricting other users.

When checking the ip6tables rules for that namespace, it was observed that the default forwarding policy was to drop packets; however, HTTP and HTTPS packets from source **fd11:beaf::c160:7ec6:f62a:69ad** were enabled to be forwarded to **fd22:b00c::e41f**. Note that **fd11:beaf::c160:7ec6:f62a:69ad** was the address of the brigadier user for this brigade, while **fd22:b00c::e41f** was the address of the endpoint node itself:

```
root@staging-vm-ep-0:/# ip netns exec nswg1 ip6tables -L -v
Chain FORWARD (policy DROP 89 packets, 8202 bytes)
 pkts bytes target     prot opt in     out     source               destination
    0     0 DROP       tcp      any    any     anywhere             anywhere             multiport dports
smtp,137,netbios-ssn
    0     0 DROP       udp      any    any     anywhere             anywhere             multiport dports netbios-
ns,netbios-dgm
    0     0 REJECT     all      any    any     anywhere             anywhere             mark match 0x1 reject-with
icmp6-adm-prohibited
    0     0 DROP       all      any    any     anywhere             anywhere             state INVALID
    0     0 SET        tcp      any    any     anywhere             anywhere             state NEW ! match-set
ScannedPorts6 src,dst,dst limit: above 10/min burst 10 mode srcip-dstip htable-expire 10000 add-set PortScanners6
src exist
  266 21280 SET        tcp      any    any     anywhere             anywhere             state NEW add-set
ScannedPorts6 src,dst,dst
    0     0 DROP       all      any    any     anywhere             anywhere             state NEW match-set
PortScanners6 src
    0     0 ACCEPT     all      wg1    ens161  anywhere             anywhere
    0     0 ACCEPT     all      ens161 wg1     anywhere             anywhere
 6213   552K ACCEPT    tcp      any    any     fd11:beaf::c160:7ec6:f62a:69ad  fd22:b00c::e41f      tcp dpt:http
 1630   206K ACCEPT    tcp      any    any     fd11:beaf::c160:7ec6:f62a:69ad  fd22:b00c::e41f      tcp dpt:https
 7711    16M ACCEPT    tcp      any    any     fd22:b00c::e41f      fd11:beaf::c160:7ec6:f62a:69ad
```

This only allowed forwarded packets to the endpoint host itself, so additional ip6tables rules in the non-namespaced network allowed forwarding the specific traffic on to the control host at **fdcc:c385:74::2**:

```
# ip6tables -L -v
Chain FORWARD (policy DROP 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
    0     0 ACCEPT     tcp      any    any     fd22:b00c::631       fdcc:c385:73::2      tcp dpt:http
    0     0 ACCEPT     tcp      any    any     fd22:b00c::631       fdcc:c385:73::2      tcp dpt:https
    0     0 ACCEPT     tcp      any    any     fdcc:c385:73::2      fd22:b00c::631
 6213  552K ACCEPT     tcp      any    any     fd22:b00c::e420      fdcc:c385:74::2      tcp dpt:http
 1630  206K ACCEPT     tcp      any    any     fd22:b00c::e420      fdcc:c385:74::2      tcp dpt:https
 7712   16M ACCEPT     tcp      any    any     fdcc:c385:74::2      fd22:b00c::e420
```

The forwarding itself was performed by pre-routing and post-routing rules in the NAT table:

```
# ip6tables -L -v -t nat
Chain PREROUTING (policy ACCEPT 104K packets, 8344K bytes)
 pkts bytes target     prot opt in     out     source               destination
    0     0 DNAT       tcp      any    any     fd22:b00c::631       anywhere             tcp dpt:http
to:[fdcc:c385:73::2]:80
    0     0 DNAT       tcp      any    any     fd22:b00c::631       anywhere             tcp dpt:https
to:[fdcc:c385:73::2]:443
  127 10160 DNAT       tcp      any    any     fd22:b00c::e420      anywhere             tcp dpt:http
to:[fdcc:c385:74::2]:80
   70  5600 DNAT       tcp      any    any     fd22:b00c::e420      anywhere             tcp dpt:https
to:[fdcc:c385:74::2]:443
[...]
Chain POSTROUTING (policy ACCEPT 287 packets, 22984 bytes)
 pkts bytes target     prot opt in     out     source               destination
    0     0 SNAT       tcp      any    any     fd22:b00c::631       fdcc:c385:73::2      tcp to:fdcc:c385:73::3
  197 15760 SNAT       tcp      any    any     fd22:b00c::e420      fdcc:c385:74::2      tcp to:fdcc:c385:74::3
```

Non-brigadier users did not get such rules set up for them, so they could not forward through the namespace. When users were created using the **WgPeerAdd()** function at lines 20-39 of file **keydesk/vpnapi/wgvpn.go**, the **control-host** parameter was added to the API call only if the user was a brigadier.

```
// WgPeerAdd - peer_add endpoint-API call.
func WgPeerAdd(actualAddrPort, calculatedAddrPort netip.AddrPort, wgPub, wgIfacePub, wgPSK []byte, ipv4, ipv6,
keydesk netip.Addr) error {
	query := fmt.Sprintf("peer_add=%s&wg-public-key=%s&wg-psk-key=%s&allowed-ips=%s",
		url.QueryEscape(base64.StdEncoding.WithPadding(base64.StdPadding).EncodeToString(wgPub)),
		url.QueryEscape(base64.StdEncoding.WithPadding(base64.StdPadding).EncodeToString(wgIfacePub)),
		url.QueryEscape(base64.StdEncoding.WithPadding(base64.StdPadding).EncodeToString(wgPSK)),
		url.QueryEscape(ipv4.String()+","+ipv6.String()),
	)

	if keydesk.IsValid() {
		query += fmt.Sprintf("&control-host=%s", url.QueryEscape(keydesk.String()))
	}

	_, err := getAPIRequest(actualAddrPort, calculatedAddrPort, query)
	if err != nil {
		return fmt.Errorf("api: %w", err)
	}

	return nil
}
```

The relevant network namespace and iptables rules were then set up at lines 218-251 of file **endpoint-setup-files/wg-mng.sh**, bound to port 8080 of the endpoint host. The lines are not reproduced here as they show similar commands to what was listed previously, and the file was a long shell script with many variables that was hard to read.

Recommendations: The isolated IPv6 network design with network namespaces for restricting non-brigadiers worked well. Consider rewriting the file **endpoint-setup-files/wg-mng.sh** in Golang, as the script was difficult to review and may contain bugs.

#5. Users in a brigade must not be able to interfere or tamper with the experience of users in other brigades, even if hosted on the same node.

In the current architecture, multiple brigades may be hosted on the same endpoint and control node pair. While an ideal architecture might see each brigade have its own deployment, this would likely be expensive.

Each brigade had its own user on the control node, with its own home directory:

```
root@staging-vm-ct-0:/home# ls -lah
total 28K
drwxr-xr-x  7 root                       root                       4.0K Jul 11 11:00 .
drwxr-xr-x 19 root                       root                       4.0K May 31 10:05 ..
drwx------  2 FH3AHFR3VFELJJME4WGCY3LGBM FH3AHFR3VFELJJME4WGCY3LGBM 4.0K Jul 28 14:19 FH3AHFR3VFELJJME4WGCY3LGBM
drwx------  2 GFHPNZG7RRE5TKJ66VXU35SM7E GFHPNZG7RRE5TKJ66VXU35SM7E 4.0K Jul 28 14:19 GFHPNZG7RRE5TKJ66VXU35SM7E
drwxr-x---  4 _marina_                   _marina_                   4.0K May 31 18:00 _marina_
drwxr-x---  4 _serega_                   _serega_                   4.0K May 31 17:49 _serega_
```

Each home directory contained a Brigade database. The relevant base32-encoded named brigade user account ran a unique **Keydesk** application for each Brigade database with its own socket:

```
root@staging-vm-ct-0:/home/FH3AHFR3VFELJJME4WGCY3LGBM# systemctl list-units | grep vgkeydesk
  vgkeydesk@FH3AHFR3VFELJJME4WGCY3LGBM.service                                                       loaded active
running   VPNGen Keydesk
  vgkeydesk@GFHPNZG7RRE5TKJ66VXU35SM7E.service                                                       loaded active
running   VPNGen Keydesk
  system-vgkeydesk.slice                                                                             loaded active
active    Slice /system/vgkeydesk
  vgkeydesk@FH3AHFR3VFELJJME4WGCY3LGBM.socket                                                        loaded active
running   vgkeydesk@FH3AHFR3VFELJJME4WGCY3LGBM.socket
  vgkeydesk@GFHPNZG7RRE5TKJ66VXU35SM7E.socket                                                        loaded active
running   vgkeydesk@GFHPNZG7RRE5TKJ66VXU35SM7E.socket
```

To test if this isolation was working correctly, the **Keydesk** route handler **DeleteUserUserIDHandler()** function was used by the assessment team to try to delete a UUID referring to a user in a different Brigade database. This was forbidden, as expected:

Request:

```
DELETE /user/f333a4c7-793f-4670-aac6-1591779243e8 HTTP/1.1
Host: [fd22:b00c::e41f]
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101 Firefox/102.0
Accept: application/json, text/plain, */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://[fd22:b00c::e41f]/
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5c[...]
Origin: http://[fd22:b00c::e41f]
Connection: close
```

Response:

```
HTTP/1.1 403 Forbidden
Date: Fri, 28 Jul 2023 13:16:24 GMT
Content-Length: 0
Connection: close
```

As mentioned previously, each brigade had its own network namespace on the endpoint node:

```
root@staging-vm-ep-0:/home/ubuntu# ip a
[...]
17: wg0veth0@if16: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether aa:67:b5:ba:3b:c1 brd ff:ff:ff:ff:ff:ff link-netns nswg0
    inet6 fd22:b00c::630/128 scope global
       valid_lft forever preferred_lft forever
    inet6 fe80::502f:fbff:fe1d:deb4/64 scope link
       valid_lft forever preferred_lft forever
19: wg1veth0@if18: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether b6:0a:6d:29:cf:c4 brd ff:ff:ff:ff:ff:ff link-netns nswg1
    inet6 fd22:b00c::e41f/128 scope global
       valid_lft forever preferred_lft forever
    inet6 fe80::d49d:63ff:fe46:bed4/64 scope link
       valid_lft forever preferred_lft forever
```

In practice, one brigade could likely disrupt the service by using a large amount of traffic, causing a denial of service for other co-located brigades. Each created user received a 100GB allowance, but a brigadier could keep generating new users to refresh the limit. The file **endpoint-setup-files/wg-mng.sh** contained a facility to set bandwidth for particular users, but it needed to be enforced by an administrator.

Recommendations: The assessment team recommends setting a bandwidth limit that applies across a brigade. For further isolation of individual brigades, and mitigating privilege escalation from potential application vulnerabilities, the assessment team suggests configuring AppArmor profiles for **VPN Generator** services to ensure they can perform only expected functionalities.