**РАЗРАБОТКА VPN-СЕРВЕРА**

В данной главе описывается разработка ASP-приложения и docker-контейнера, реализующих функционал управляемого Wireguard-сервера. Почти весь представленный в данном документе программный код файлов, классов и функций - неполный, цель состоит в демонстрации идеи.

**Разработка WebAPI**

Начнем с разработки ASP WebAPI-приложения. Создадим структуру решения представленным ниже PowerShell-скриптом. Помимо API, код, управляющий Wireguard-сервисом вынесен в отдельный проект-библиотеку. Для них созданы проекты тестирования.

dotnet new classlib -o wireguard\_manipulator;  
dotnet new webapi -o vdb\_node\_api  
dotnet new xunit -o wireguard\_manipulator.tests;  
dotnet new xunit -o vdb\_node\_api.tests;

Для начала создадим базу для дальнейшей разработки. Из предыдущих проектов импортируем класс провайдера настроек[8], повышающий абстракцию использования файла appsettings.json и прочих, позволяя использовать его как набор C#-объектов в классах остальных сервисов. Дополнительно создадим класс, извлекающий переменные среды:

public sealed class EnvironmentProvider

{

// названия переменных среды

private const string ENV\_ALLOW\_NOAUTH = "REST2WG\_ALLOW\_NOAUTH";

// извлеченные значения переменных среды

public bool? ALLOW\_NOAUTH { get; init; } = null;

// логгер

private readonly ILogger<EnvironmentProvider> \_logger;

// конструктор

public EnvironmentProvider(ILogger<EnvironmentProvider> logger) {

\_logger = logger;

ALLOW\_NOAUTH = ParseBoolValue(ENV\_ALLOW\_NOAUTH);

}

// функция извлечения и парсинга переменных

private bool? ParseBoolValue(string EnvName)

{

string? str = Environment.GetEnvironmentVariable(EnvName);

if (str is not null) {

if (str.Equals("true", StringComparison.InvariantCultureIgnoreCase)) {

\_logger.LogInformation($"{EnvName}={true}."); return true;

}

if (str.Equals("false", StringComparison.InvariantCultureIgnoreCase)) {

\_logger.LogInformation($"{EnvName}={false}."); return false;

}

\_logger.LogWarning(GetIncorrectIgnoredMessage(EnvName));

}

\_logger.LogInformation($"{EnvName} was not present.");

return null;

} }

Теперь опишем сервис, отвечающий за выдачу адресов. Данный сервис должен реализовывать отношения рода Single-to-Single, т.е. один уникальный публичный ключ к одному уникальному IP-адресу. Для хранения адреса используется Int32, байты из которого извлекаются методом BitConverter.GetBytes. Само хранение реализовано словарем (ключ к адресу) и дополнительным HashSet’ом, хранящим только адреса. Сам по себе данный сервис способен адресовать до (250\*256\*256). Первая цифра адреса каждого клиента выбрана за 10 (согласно RFC1918[11]), вторая – за 6 в связи с техническими ограничениями Alpine Linux, особого значения это не имеет.

*The Internet Assigned Numbers Authority (IANA) has reserved the*

*following three blocks of the IP address space for private internets:*

*10.0.0.0 - 10.255.255.255 (10/8 prefix)*

*172.16.0.0 - 172.31.255.255 (172.16/12 prefix)*

*192.168.0.0 - 192.168.255.255 (192.168/16 prefix).*

Маска подсети выбрана за /32, т.е. весь адрес относится к узлу, подсети отсутствуют, что является стандартной практикой конфигурирования Wireguard, ибо мы хотим развернуть туннель, а не сеть. Когда сервису необходимо выдать клиенту адрес – он в первую очередь пытается сгенерировать адрес на основании текущего количества клиентов, проще говоря, пытается угадать, какой адрес свободен. К сожалению, одни клиенты могут оставаться подключенными длительное время, другие же – отключаться спустя секунды, что, в действительности, порождает случайное распределение занятых адресов по адресному пространству. Это приводит к необходимости каждый раз проверять свободность того или иного адреса. В целом, данные операции не должны существенно снижать производительность, так как и поиск, и добавление, и удаление из HashSet’а имеет сложность О(1). При удалении пира его публичный ключ и выделенный адрес также должны удаляться из соответствующих коллекций, таким образом происходит освобождение адресов. Отдельным вопросом является синхронизация реального состояния Wireguard-сервиса с WebAPI. Например, клиент может отключиться от сервера по причине потери у клиента электропитания машины, что приведет к тому, что де-факто пир будет отключен, но об этом не будет сообщено ASP-приложению. Тогда ASP-приложение должно регулярно проверять список пиров и удалять те, что уже не подключены. Код сервиса представлен ниже в кратком варианте.

public sealed class IpDedicationService{

private const byte FirstIpByteStart = 10, SecondIpByteStart = 6, NetworkMask = 32;

private const int MaxClients = 100 \* 255 \* 255;

private Dictionary<string, int> \_dedicatedAddresses; // ключ к выделенному его адресу

private readonly HashSet<int> \_usedAddresses; // все занятые адреса

public int IpsAvailable => MaxClients - \_usedAddresses.Count;

public IpDedicationService() { \_dedicatedAddresses = new(); \_usedAddresses = new(); }

/// <returns> address dedicated for the pubKey in the format '10.6.0.0/32' </returns>

public string EnsureDedicatedAddressForPeer(string pubKey) // tested

{

if (\_dedicatedAddresses.TryGetValue(pubKey, out int address)) {

return IndexToString(address);

}

else

{

if (\_dedicatedAddresses.Count >= MaxClients) {

throw new IndexOutOfRangeException("Max clients reached.");

}

int addr = \_dedicatedAddresses.Count; // firstly, try count as address

if (\_usedAddresses.Contains(addr)) {

for (addr = 0; addr < MaxClients; addr++) {

if (!\_usedAddresses.Contains(addr)) {

AddPeer(pubKey, addr);

return IndexToString(addr);

} } }

AddPeer(pubKey, addr);

return IndexToString(addr);

}

}

/// <returns> true if the pubKey was successfully added, false otherwise. </returns>

private bool AddPeer(string pubKey, int addressIndex) // tested

{

\_dedicatedAddresses.Add(pubKey, addressIndex);

return \_usedAddresses.Add(addressIndex);

}

/// <returns> true if the pubKey was found and removed, false otherwise. </returns>

public bool DeletePeer(string pubKey) // tested

{

if (!\_dedicatedAddresses.TryGetValue(pubKey, out int address)) {

return false;

}

else {

\_usedAddresses.Remove(address);

return \_dedicatedAddresses.Remove(pubKey);

}

}

public void SyncState(Dictionary<string, int> keyToIpActulState)

{

\_dedicatedAddresses = keyToIpActulState;

\_usedAddresses.Clear();

foreach (int ip in \_dedicatedAddresses.Values)

\_usedAddresses.Add(ip);

}

Теперь перейдем к описанию службы пиров, которая непосредственно будет внедряться в контроллеры. Она служит связующим звеном между контроллерами, сервисом выделения адресов и кодом манипуляции Wireguard-сервисом. Функции сервиса состоят в добавлении и удалении пиров, получении списка пиров, синхронизации состояния. Код сервиса представлен ниже в кратком варианте. В нем уже применен класс WgCommandsExecutor, который будет описан позднее.

public sealed class PeersBackgroundService : BackgroundService

{

private readonly PeersBackgroundServiceSettings \_settings;

private readonly IpDedicationService \_ipService;

private readonly ILogger<PeersBackgroundService> \_logger;

private DateTime \_lastUpdateUtc = DateTime.MinValue;

public string? InterfacePubkey { get; private set; }

public PeersBackgroundService(IpDedicationService ipService,

SettingsProviderService settingsProvider, ILogger<PeersBackgroundService> logger){

\_ipService = ipService;

\_settings = settingsProvider.PeersBackgroundServiceSettings;

\_logger = logger;

if (\_settings.HandshakeAgoLimitSeconds <= 0)

\_settings.HandshakeAgoLimitSeconds = int.MaxValue;

}

public async Task<string> EnsurePeerAdded(string pubkey) {

string ip = \_ipService.EnsureDedicatedAddressForPeer(pubkey);

string result = await WgCommandsExecutor.AddPeer(pubkey, ip);

return !string.IsNullOrWhiteSpace(result) ?

throw new AggregateException(result) : ip;

}

public async Task<bool> EnsurePeerRemoved(string pubkey) {

string output = await WgCommandsExecutor.RemovePeer(pubkey);

return \_ipService.DeletePeer(pubkey) && string.IsNullOrWhiteSpace(output);

}

public async IAsyncEnumerator<WgShortPeerInfo> GetPeersAndUpdateState() {

var enumer = await WgCommandsExecutor.GetPeersListEnumerator();

var syncedDictionary = new Dictionary<string, int>();

while (enumer.MoveNext())

{

var peer = enumer.Current;

if (peer.HandshakeSecondsAgo > \_settings.HandshakeAgoLimitSeconds)

await EnsurePeerRemoved(peer.PublicKey);

else { syncedDictionary.Add(peer.PublicKey,

\_ipService.StringToIndex(peer.AllowedIps));

yield return peer;

}

}

\_ipService.SyncState(syncedDictionary);

\_lastUpdateUtc = DateTime.UtcNow;

InterfacePubkey = WgCommandsExecutor.LastSeenInterfacePubkey;

yield break;

}

protected override async Task ExecuteAsync(CancellationToken stoppingToken) {

if (\_settings.PeersRenewIntervalSeconds <= 0) return;

while (!stoppingToken.IsCancellationRequested){

if ((DateTime.UtcNow - \_lastUpdateUtc).TotalSeconds

> \_settings.PeersRenewIntervalSeconds) {

var enumer = GetPeersAndUpdateState();

while (await enumer.MoveNextAsync()) ;

int delayS = \_settings.PeersRenewIntervalSeconds

- (int)(DateTime.UtcNow - \_lastUpdateUtc).TotalSeconds;

await Task.Delay(delayS \* 1000);

} } }

Далее необходимо создать middleware авторизации. Для него создается специальный сервис, валидирующий мастер-аккаунты. В приложение различными путями передаются хеши ключей аккаунтов. Хранение ключей в открытом виде нецелесообразно. Краткий код мидлваре и сервиса представлен ниже.

public sealed class MasterAccountsService {

private readonly List<byte[]> \_mastersKeyHashes;

public MasterAccountsService(SettingsProviderService settingsProvider) {

\_mastersKeyHashes = settingsProvider.MasterAccounts

.Select(x => Convert.FromBase64String(x.KeyHashBase64)).ToList();

}

public bool IsValid(string keyBase64) {

var search = SHA512.HashData(Convert.FromBase64String(keyBase64));

return \_mastersKeyHashes.Any(k => k.SequenceEqual(search));

}

}

public sealed class AuthorizationMiddleware : IMiddleware

{

private readonly MasterAccountsService \_accountsService;

private readonly ILogger \_logger;

public AuthorizationMiddleware(MasterAccountsService accountsService, ILogger logger) {

\_accountsService = accountsService;

\_logger = logger;

}

public Task InvokeAsync(HttpContext context, RequestDelegate next)

{

var header = context.Request.Headers.Authorization; string? key;

try { key = header.Single(); } /\* Acc. to RFC, Authorize may appear only once \*/

catch (InvalidOperationException) {

\_logger.LogWarning(GetRejectionMessage(context,

"Authorization header seen more than once."));

return Task.FromResult(new StatusCodeResult(Status400BadRequest));

}

if (string.IsNullOrEmpty(key)) {

\_logger.LogWarning(GetRejectionMessage(context,

"Authorization key was null or empty."));

return Task.FromResult(new StatusCodeResult(Status400BadRequest));

}

try {

if (!\_accountsService.IsValid(key)) {

\_logger.LogWarning(GetRejectionMessage(context,

"Authorization key was not found on the server."));

return Task.FromResult(new StatusCodeResult(401));

}

}

catch{

\_logger.LogWarning(GetRejectionMessage(context,

"Authorization key format was not valid."));

return Task.FromResult(new StatusCodeResult(Status400BadRequest));

}

return next(context); /\* key was successfully validated \*/

}}

Теперь создадим манипулятор WG-сервисом. В целом, он не должен иметь в себе никакой существенной логики, а только выполнять переданные ASP-приложением команды и парсить их выводы. Краткий код представлен ниже.

public static class WgCommandsExecutor

{

public static string? LastSeenInterfacePubkey

=> WgStatusStreamParser.LastSeenInterfacePubkey;

private static async Task<string> RunCommand(string command, string fileName = @"wg") {

return await (await RunCommandStream(command, fileName)).ReadToEndAsync();

}

private static async Task<StreamReader> RunCommandStream(string command, string fileName = @"wg") {

var psi = new ProcessStartInfo();

psi.FileName = fileName;

psi.Arguments = command;

psi.RedirectStandardOutput = true;

psi.UseShellExecute = false;

psi.CreateNoWindow = true;

var process = Process.Start(psi);

if (process is null) {

throw new AggregateException("Unable to perform the command");

}

await process.WaitForExitAsync();

return process.StandardOutput;

}

private static string GetAddPeerCommand(string pubKey, string allowedIps) {

return $"set wg0 peer \"{pubKey}\" allowed-ips {allowedIps}";

}

private static string GetRemovePeerCommand(string pubKey) {

return $"set wg0 peer \"{pubKey}\" remove";

}

private static string GetWgShowCommand(string wgInterfaceName = null!) {

return wgInterfaceName is null ? "show" : $"show {wgInterfaceName}";

}

public static async Task<string> AddPeer(string pubKey, string allowedIps) {

return await RunCommand(GetAddPeerCommand(pubKey, allowedIps));

}

public static async Task<string> RemovePeer(string pubKey) {

return await RunCommand(GetRemovePeerCommand(pubKey));

}

public static async Task<IEnumerator<WgShortPeerInfo>> GetPeersListEnumerator()

{

return WgStatusStreamParser.ParsePeersFromStreamShortly(

await RunCommandStream(GetWgShowCommand()));

} }

В коде применен класс WgStatusStreamParser, код которого не будет представлен в связи с большим размером. Данный класс пошагово выполняет разбор потока вывода команды ‘wg show’, получая по итогу список экземпляров класса WgShortPeerInfo, содержащих открытый ключ, выделенный адрес и количество секунд, прошедших с последнего рукопожатия (‘latest handshake’).

**Конфигурация NGINX**

Поскольку ASP имеет возможность авторизации по API-ключу, который передается в заголовке Authorize в формате base64, что является кодированием, а не шифрованием, необходимо защитить подключение. К тому же, в будущем можно представить сценарии, в которых необходимо ограничить частоту запросов или разрешить запросы только с определенного адреса. Для этого опишем конфигурацию NGINX, инкапсулирующего подключения к ASP.

worker\_processes  auto;

http {

ssl\_protocols TLSv1.2 TLSv1.3;  
ssl\_certificate /etc/ssl/certs/nginx-selfsigned.crt;  
ssl\_certificate\_key /etc/ssl/private/nginx-selfsigned.key;  
add\_header Strict-Transport-Security "max-age=63072000; includeSubdomains";

server {

    listen              51821 ssl;

listen              51822;

limit\_req zone=first\_zone nodelay;

location / {

proxy\_pass <http://0.0.0.0:5000>;

} } }

**Конфигурация Wireguard**

В конфигурации WG применим наиболее простую из обеспечивающих полную функциональность конфигурацию. Замечу, что PrivateKey оставлен пустым и генерируется при запуске уже непосредственно в контейнере.

[Interface]

Address = 10.0.0.0/8

PostUp = iptables -I FORWARD -i wg0 -j ACCEPT;

PostUp = iptables -t nat -I POSTROUTING -o eth0 -j MASQUERADE;

PostDown = iptables -D FORWARD -i wg0 -j ACCEPT;

PostDown =  iptables -t nat -D POSTROUTING -o eth0 -j MASQUERADE;

ListenPort = 51820

PrivateKey =

Применение iptables потребует выдачи контейнеру некоторых привилегий, ибо изначально контейнер не имеет доступа к настройкам сети. Директива MASQUERADE означает маскировку траффика, как будто он исходит от сервера, а не от удаленного клиента посредствам туннеля.

**Конфигурация Docker и сборка образа**

Как и планировалось, мы базируем свой образ на Alpine Linux – одной из самый легковесных разновидностей Linux, полноценный образ которой веси 5 МБ. Мы применяем так называемую multi-stage build. Мы создаем контейнер, в который устанавливаем SDK, там производим компиляцию, после чего создаём финальный контейнер, куда копируем уже собранное приложение, а в нём добавляем только Runtime, что уменьшает финальную сборку. Рассмотрим dockerfile:

FROM alpine:3 AS base

FROM mcr.microsoft.com/dotnet/sdk:7.0 AS build

COPY ./vdb\_node\_api /vdb\_node\_api

COPY ./vdb\_node\_wireguard\_manipulator /vdb\_node\_wireguard\_manipulator

RUN dotnet publish /vdb\_node\_api/vdb\_node\_api.csproj -c "Release" -r linux-musl-x64 --no-self-contained -o /app/publish

FROM base AS final

COPY --from=build /app/publish /app

RUN apk add -q --no-progress nginx

RUN apk add -q --no-progress openssl

RUN apk add -q --no-progress wireguard-tools

RUN apk add -q --no-progress aspnetcore7-runtime

COPY ./build\_alpine/pre-setup.sh ./etc/rest2wg/pre-setup.sh

COPY ./build\_alpine/pre-wg0.conf ./etc/rest2wg/pre-wg0.conf

COPY ./build\_alpine/pre-nginx-limit\_req.conf.template ./etc/rest2wg/pre-nginx-limit\_req.conf.template

COPY ./build\_alpine/pre-nginx.conf/ ./etc/nginx/nginx.conf

COPY ./build\_alpine/pre-ssl-params.conf ./etc/nginx/snippets/ssl-params.conf

COPY ./build\_alpine/pre-self-signed.conf ./etc/nginx/snippets/self-signed.conf

ENV ASPNETCORE\_ENVIRONMENT=Production

ENV REST2WG\_LIMIT\_REQ=100000;

CMD ["bash", "-c", "chmod +x /etc/rest2wg/pre-setup.sh && /etc/rest2wg/pre-setup.sh"]

**Точка входа контейнера**

Рассмотрим точку входа контейнера, которой является bash-скрипт ‘pre\_setup.sh’. Предшествующая ему команда ‘chmod +x /etc/rest2wg/pre-setup.sh’ исправляет потенциальную неспособность оболочки распознать файл как исполняемый. Данный скрипт выполняет несколько задач. Он генерирует приватный ключ Wireguard, если таковой отсутствует; генерирует self-signed x509 сертификат, если таковой отсутствует, для шифрования траффика посредствам NGINX; исправляет значения переменной среды REST2WG\_LIMIT\_REQ, если пользователь задал её некорректно; генерирует файл, конфигурирующий ограниче запросов NGINX, если таковой отсутствует; запускает NGINX; запускает Wireguard; запускает ASP WebAPI. Последняя команда служит для предотвращения выхода из контейнера по завершению скрипта. Рассмотрим полный код.

if ! test -e "/etc/wireguard/wg0.conf"; then

    echo "Wireguard configuration file not detected. Generating..."

    cp /etc/rest2wg/pre-wg0.conf /etc/wireguard/wg0.conf

    wg genkey >> /etc/wireguard/wg0.conf

fi

if ! ((test -e /etc/ssl/private/nginx-selfsigned.key)

&& (test -e /etc/ssl/certs/nginx-selfsigned.crt)); then

echo "Self-signed x509 sertificate files not detected. Generating..."

openssl req -x509 -nodes -days 36500 -newkey rsa:2048 -subj "/CN=US/C=US/L=San Fransisco" -keyout /etc/ssl/private/nginx-selfsigned.key   
-out /etc/ssl/certs/nginx-selfsigned.crt

fi

if !((REST2WG\_LIMIT\_REQ > 0)) && ((REST2WG\_LIMIT\_REQ <= 9999999)); then

    echo "Incorrect value of REST2WG\_LIMIT\_REQ environment variable was ignored."

    echo "REST2WG\_LIMIT\_REQ was set to 100k."

    REST2WG\_LIMIT\_REQ=100000;

fi

if ! test -e "/etc/nginx/snippets/nginx-limit\_req.conf"; then

    echo "Nginx limit\_req configuration file not detected. Generating..."

    cp /etc/rest2wg/pre-nginx-limit\_req.conf.template

/etc/nginx/snippets/nginx-limit\_req.conf

    echo "${REST2WG\_LIMIT\_REQ}r/s;" >> /etc/nginx/snippets/nginx-limit\_req.conf

fi

echo "Spinning up the Nginx reverse-proxy..."

nginx

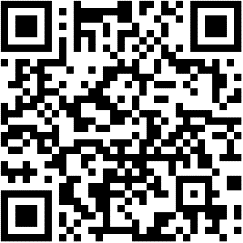
echo "Spinning up the Wireguard service..."

wg-quick up wg0 && wg show wg0

echo "Spinning up the ASP WebAPI..."

dotnet /app/vdb\_node\_api.dll -no-launch-profile

tail -f /dev/null



github.com/LuminoDiode  
/rest2wireguard

===========================>

hub.docker.com/repository/docker  
/luminodiode/rest2wireguard

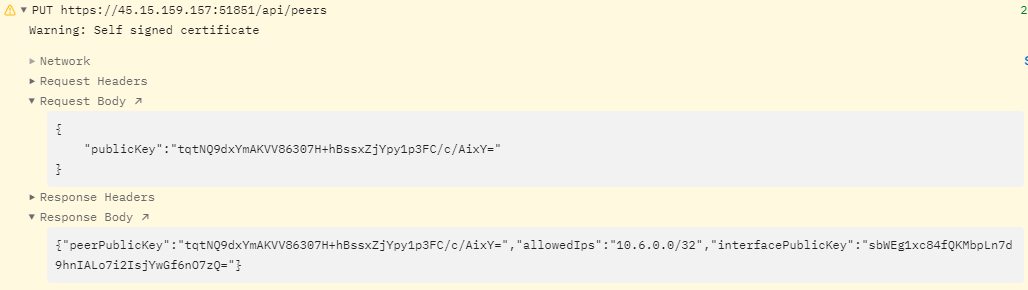
===========================>

**Тестирование практической работоспособности**

Запустим контейнер на имеющемся VPS, открыв порт для Wireguard и порт для TLS-защищенного подключения к ASP-приложению.

**docker run** --cap-add NET\_ADMIN -p 51850:51820/udp -p 51851:51821/tcp

--env REST2WG\_ALLOW\_NOAUTH=true **luminodiode/rest2wireguard**:0.0.92-rc

С помощью Postman осуществим запрос и добавим имеющийся локально Wireguard клиент.

Теперь проверим создание туннеля.

