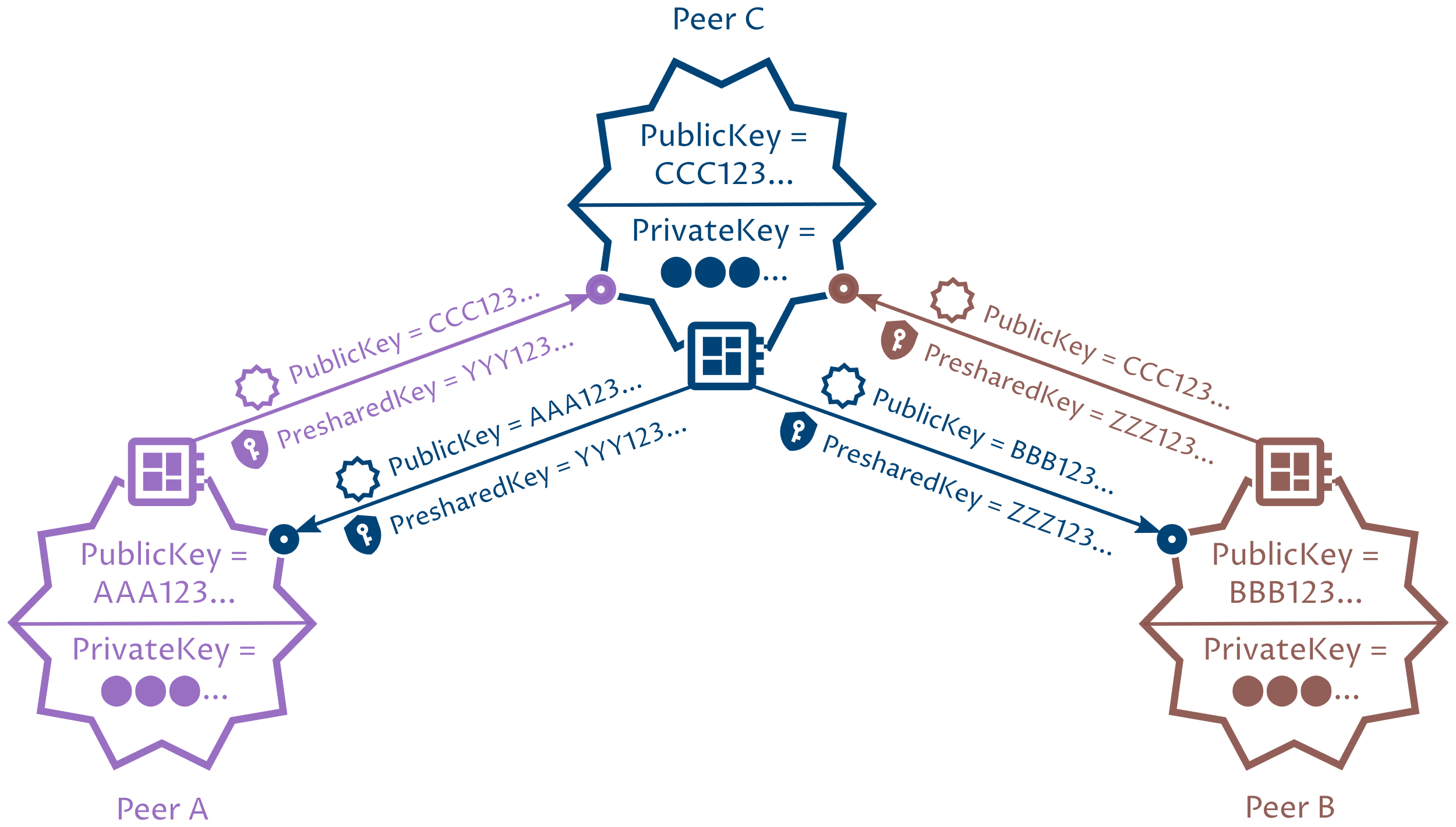
**ПРОТОТИПИРОВАНИЕ И РАЗРАБОТКА VPN-СЕРВЕРА**

В данном главе описывается разработка ASP-приложения и docker-контейнера, реализующих функционал управляемого Wireguard-сервера. Начнем разработку с ASP-приложения.

Для начала импортируем в проект класс провайдера настроек[8], повышающий абстракцию использования файла appsettings.json и прочих, позволяя использовать его как набор C#-объектов в классах остальных сервисов. Этот класс был разработан значительно ранее.

На этапе прототипирования возник вопрос использования pre-shared key в пирах Wireguard. Данные ключи обеспечивают симметричное шифрование, в дополнение к асимметричному (по открытому ключу). По большому счету, данный функционал предназначен для защиты от атак квантовых компьютеров [9] и не будет добавлен в решение:

*« Importantly, you don’t* need *to use preshared keys with WireGuard — there are no known practical attacks against the WireGuard cryptosystem, even without preshared keys. However, because WireGuard key pairs use elliptic-curve cryptography (ECC), in the future it might be possible for quantum computers to “crack” those key pairs (where the quantum computer would be able to calculate the private key from the public key). This would enable an organization with a quantum computer to decrypt any encrypted WireGuard traffic that it had previously captured and stored.*

*Using preshared keys means that even if an attacker calculated (or was able to steal) the private keys used by a WireGuard connection, she still wouldn’t be able to decrypt the connection’s encrypted WireGuard traffic. To do so, she would also have to steal the preshared key (a randomly-generated preshared key would not be “crackable” by quantum computers, nor by any other means). »*

**Теперь перейдем непосредственно к написанию кода.** Как было описано ранее, необходимо реализовать аутентификацию/авторизацию по предопределенным секретным ключам. Для этого реализован сервис, извлекающий ключи из файла и производящих валидацию. Также реализовано мидлваре, при прохождении конвейера проверяющее наличие в заголовке запроса Authorize валидного ключа.

public sealed class MasterAccountsService /\* Singleton service \*/

{

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 \*/

}}

 На текущем этапе одна из основных функций ASP-приложения – выделение адресов для публичных ключей. Для этого необходимо написать сервис, формирующий таковой адрес. Для начала экспериментально проверим, что Wireguard может работать с адресами от 10.0.0.0 до 10.255.255.255. Для этого на тестовом сервере была задана следующая конфигурация, после чего все соединения были успешно протестированы:

Теперь опишем сервис, отвечающий за выдачу адресов. Данный сервис должен реализовывать отношения рода PK-to-PK, т.е. один уникальный публичный ключ к одному уникальному IP-адресу. Для хранения адреса используется Int32, байты из которого извлекаются методом BitConverted.GetBytes. Само хранение реализовано словарем (ключ к адресу) и дополнительным HashSet’ом, хранящим только адреса. Сам по себе данный сервис способен адресовать до (245\*256\*256\*256) клиентов, первая цифра адреса по негласному соглашению выбрана за 10, особого значения это не имеет. Маска подсети выбрана за /32, т.е. весь адрес относится к узлу, подсети отсутствуют, что является стандартной практикой конфигурирования Wireguard. 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. Ниже представлен код тестирования разработанного сервиса на способность к правильной итерации адресов, способность к выдаче адресов, способность к удалению адресов из списка занятых.

Основные методы самого сервиса выглядят следующим образом:

[Fact] public void **CanEnumerateAddresses**()

{

var address1 = IndexToString(0); // first value

var address255 = IndexToString(255); // 1 byte is full

var address255\_p1 = IndexToString(255 + 1); // 1 byte is full + 1 started

var address256\_256\_m1 = IndexToString(256 \* 256 - 1); // 2 bytes are full

var address256\_256 = IndexToString(256 \* 256); // 2 bytes are full + 1

var address256\_256\_256\_m1 = IndexToString(256 \* 256 \* 256 - 1); // 3 r full

var address256\_256\_256 = IndexToString(256 \* 256 \* 256); // 3 r full + 1

Assert.Equal("10.0.0.0/32", address1);

Assert.Equal("10.0.0.255/32", address255);

Assert.Equal("10.0.1.0/32", address255\_p1);

Assert.Equal("10.0.255.255/32", address256\_256\_m1);

Assert.Equal("10.255.255.255/32", address256\_256\_256\_m1);

Assert.Equal("11.0.0.0/32", address256\_256\_256);

}

[Fact] public void **CanRegisterKey**()

{

var key1 = Convert.ToBase64String(RandomNumberGenerator.GetBytes(256 / 8));

var key2 = Convert.ToBase64String(RandomNumberGenerator.GetBytes(256 / 8));

var ip1 = service.EnsureDedicatedAddressForPeer(key1);

var ip2 = service.EnsureDedicatedAddressForPeer(key2);

var ip11 = service.EnsureDedicatedAddressForPeer(key1);

var ip22 = service.EnsureDedicatedAddressForPeer(key2);

Assert.Equal(ip1, ip11);

Assert.Equal(ip2, ip22);

}

[Fact] public void **CanDeleteKey**()

{

var key1 = Convert.ToBase64String(RandomNumberGenerator.GetBytes(256 / 8));

var key2 = Convert.ToBase64String(RandomNumberGenerator.GetBytes(256 / 8));

var key3 = Convert.ToBase64String(RandomNumberGenerator.GetBytes(256 / 8));

var ip1 = service.EnsureDedicatedAddressForPeer(key1);

var ip2 = service.EnsureDedicatedAddressForPeer(key2);

var del1 = service.DeletePeer(key1);

Assert.True(del1);

var ip3 = service.EnsureDedicatedAddressForPeer(key3);

Assert.Equal(ip1, ip3);

ip1 = service.EnsureDedicatedAddressForPeer(key1);

Assert.StartsWith("10.0.0.0", ip3);

Assert.StartsWith("10.0.0.1", ip2);

Assert.StartsWith("10.0.0.2", ip1);

}

[Fact] public void **CanParseAddress**()

{

var Ids = new int[] { 0, 255, 255 + 1,

256 \* 256 - 1, 256 \* 256, 256 \* 256 \* 256 - 1, 256 \* 256 \* 256 };

var converted = Ids.Select(x => StringToIndex(IndexToString(x)));

Assert.True(Ids.SequenceEqual(converted));

}

using System;

namespace vdb\_node\_api.Services;

/\* Данный Singleton-сервис служит для нумерации IP-адресов клиентов.

\* Кажому серверу предлагается адресовать до 16 777 216 клиентов,

\* путем итерации по трём байтам следующего рода

\* 10.0.0.0 -> +1 -> 10.0.0.1

\* 10.0.0.1 -> +255 -> 10.0.1.0

\* 10.0.0.0 -> +16581375 -> 10.255.255.255

\* Согласно RFC 1918 (https://datatracker.ietf.org/doc/html/rfc1918#section-3),

\* существуют и другие пространства приватных адресов, однако

\* 10.0.0.0 -> 10.255.255.255 является общепринятым и достаточным.

\*

\* Данный сервис должен заниматься хранением публичного ключа и

\* адреса, выделенного для него. Хранение осуществляется в формате

\* словаря <string pubKeyBase64, int dedicatedAddress>, где

\* int dedicatedAddress служит для хранения IP-адреса в виде 4 байтов.

\*

\* Максимальное количество клиентов предлагается ограничить

\* пространством 10.10.255.255, а равно 10\*255\*255 = 650к клиентов.

\*

\* Длинна ключа клиента WG составляет 256 бит, а равно 32 байта.

\* Длинна IP-адреса состаавляет 4 байта. Маска - константа, /32.

\* Поскольку Base64 представляет строку длинной равной 4/3 символов

\* от количества байт в изначальной последовательности, то для

\* хранения в таком виде потребуется 4/3\*32\*650\*1000/1024/1024 Мб

\* памяти, что составляет менее 27 мб и является приемлемым.

\* Для хранения адресов потребуется 4\*4\*650\*1000/1024/1024 Мб

\* памяти, что составляет менее 10 мб и является приемлимым.

\*

\* Одним из вариантов является не хранение адресов, а их формирование

\* на основании индекса в массиве хранимых публичных ключей.

\* Заметим, что List<T> в С# представляет инкапуляцию Array<T>

\* и имеет скорость случайного доступа равную О(1). Однако

\* такой вариант плох проверкой ключа на существование, ибо

\* 650к - большое число элементов. В данном случае предлагается

\* использовать словарь.

\*

\* Словарь элементов представляет собой публичный ключ и адрес,

\* выделенный для него. Поскольку ключи словаря представляют

\* объект типа HashSet, то проверка уникальности ключа является

\* высокопроизводительной и позволяет легко избегать выделения

\* нескольких адресов одному ключу.

\*

\* Предполагается, что данный сервис должен гарантировать полный

\* контроль выделенных адресов wireguard и на этапе разработки

\* не видится препятствий к этому.

\*/

public sealed class IpDedicationService

{

private void Swap<T>(ref T v1, ref T v2) { var t = v1; v1 = v2; v2 = t; }

private const int MaxClients = 10 \* 255 \* 255;

private const int NetworkMask = 32;

private Dictionary<string, int> \_dedicatedAddresses { get; set; }

private HashSet<int> \_usedAddresses { get; set; }

public IpDedicationService()

{ dedicatedAddresses = new(); \_usedAddresses = new(); }

private int StringToIndex(string address) // tested

{

var slashIndex = address.LastIndexOf('/');

var actualAddress = slashIndex == -1 ?

address : address.Substring(0, slashIndex);

return BytesToIndex(actualAddress.Split('.').Select(byte.Parse).ToArray());

}

private int BytesToIndex(byte[] bytes) // tested

{

Swap(ref bytes[3], ref bytes[0]);

Swap(ref bytes[2], ref bytes[1]);

bytes[3] -= 10;

return BitConverter.ToInt32(bytes);

}

private byte[] IndexToBytes(int index) // tested

{

var address = BitConverter.GetBytes(index);

address[3] += 10; // BitConverter return bytes right to left!

return address;

}

private string IndexToString(int index) // tested

{

var address = IndexToBytes(index);

// concat methods performance compared https://imgur.com/a/5dGE8xE

// BitConverter return bytes from smaller to bigger! (right to left)

return $"{address[3]}.{address[2]}.{address[1]}.{address[0]}/{NetworkMask}";

}

public string EnsureDedicatedAddressForPeer(string pubKey) // tested

{

if (\_dedicatedAddresses.TryGetValue(pubKey, out var address))

return IndexToString(address);

else {

if (this.\_dedicatedAddresses.Count >= MaxClients)

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

// firstly, try to use count as new address

int addr = \_dedicatedAddresses.Count;

if (\_usedAddresses.Contains(addr)) {

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

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

\_dedicatedAddresses.Add(pubKey, addr);

\_usedAddresses.Add(addr);

return IndexToString(addr);

} } }

\_dedicatedAddresses.Add(pubKey, addr);

\_usedAddresses.Add(addr);

return IndexToString(addr);

}

}

public bool DeletePeer(string pubKey) // tested

{

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

return false;

else {

\_usedAddresses.Remove(address);

return \_dedicatedAddresses.Remove(pubKey);

} }

Теперь создадим прототип манипулятор WG-сервисом. В целом, он не должен иметь в себе никакой существенной логики, а только выполнять переданные ASP-приложением команды. Пока-что заложим в него небольшой набор таковых. Заметим, что, вероятнее всего, bash будет заменен на sh:

private async Task<string> RunCommandWithBash(string command)

{

var psi = new ProcessStartInfo();

psi.FileName = "/bin/bash";

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();

var output = process.StandardOutput.ReadToEnd();

return output;

}

private string GetAddPeerCommand(string pubKey, string allowedIps)

{

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

}

private string GetRemovePeerCommand(string pubKey)

{

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

}

public async Task<string> ExecuteArbitraryCommand(string command)

{

return await RunCommandWithBash(command);

}

public async Task AddPeer(string pubKey, string allowedIps)

{

await ExecuteArbitraryCommand(GetAddPeerCommand(pubKey, allowedIps));

}

public async Task RemovePeer(string pubKey)

{

await ExecuteArbitraryCommand(GetRemovePeerCommand(pubKey));

}

Таким образом, данные методы обеспечивают удаление и добавление пиров. В будущем будет добавлено получение списка пиров, однако это – прототип.

Теперь необходимо разобраться, как конкретно будет запускаться WG-сервер. Мы понимаем, что ASP должно работать в одной операционной системе с WG. В качестве образа используем Alpine Linux – полноценной операционной системой, которая весит всего 5 МБ. Поскольку разрабатываемый сервер реализует концепцию stateless, при каждом запуске предполагается полностью удалять контейнер, де-факто, выполняя переустановку операционной системы. Когда мы запускаем контейнер –необходимо установить Wireguard, сгенерировать приватный ключ, если таковой отсутствует, запустить сервис. Рассмотрим код docker-compose.yml и использованных в нем файлов. Замечу, что по неизвестным причинам Alpine Linux отказался адресовать столь же произвольные адреса, какие способен адресовать Ubuntu, поэтому клиенты адресуются в пространстве 10.6.0.0/32 -> 10.128.255.255/32 (описанный ранее IpDedicationService был скорректирован). Итак, содержание файла docker-compose, «поднимающего» WG-сервер под Alpine’ой:

version: '3.4'

services:

my\_alpine:

image: alpine

tty: true

ports:

- "51830:51820/udp"

volumes:

- ./pre-wg0.conf:/etc/wireguard/pre-wg0.conf:ro

- ./pre-setup.sh:/etc/wireguard/pre-setup.sh

cap\_add:

- NET\_ADMIN

sysctls:

- net.ipv4.ip\_forward=1

command: [sh, -c, "

umask 077 &&

chmod +x /etc/wireguard/pre-setup.sh &&

/etc/wireguard/pre-setup.sh"]

Рассмотрим данный код. Он создает сервис докер-контейнер с именем my\_alpine (временное имя), использует за изначальный образ (image) Linux Alpine. Директива tty указывает, что контейнер должен продолжать исполнения после выполнения директивы command. Директива ports указывает перенаправление с внешнего порта 51830 на внутренний 51820 (оба порта являются стандартными портами Wireguard, однако внешний порт 51820 чаще всего окажется занят уже работающим в системе WG-сервисом). Директива cap\_add: NET-ADMIN разрешает контейнеру манипулировать командой iptables, которая требуется для настройки туннелирования Wireguard. Директива sysctls служит для задания системных переменных, что позволяет задать их изначально, а не запускать контейнер в привилегированном режиме с возможностью изменить их в процессе работы. Переменная системы ‘net.ipv4.ip\_forward=1’ является обязательной для корректной работы Wireguard. Наконец, директива command выполняет определенные команды в уже запущенном контейнере. Теперь рассмотрим содержание sh-скрипта, предполагаемого к выполнению:

apk add wireguard-tools

if ! test -e "/etc/wireguard/privatekey"; then

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

wg genkey > /etc/wireguard/privatekey

cat /etc/wireguard/privatekey >> /etc/wireguard/wg0.conf

fi

wg-quick up wg0 && wg show wg0

Данный скрипт, что очевидно, устанавливает Wireguard, проверяет существование файла секретного ключа, если он отсутствует – создает его и записывает с ним новую конфигурацию wg0. Далее идет запуск с помощью команды ‘wg-quick up wg0’. Пиры для тестирования добавляются командой ‘wg set wg0 peer PUB\_KEY allowed-ips 10.6.0.0/32’, выполняемой внутри контейнера с помощью ‘docker exec CONTAINER …command’. Упомянутый выше файл pre-wg0.conf содержит предварительную конфигурацию Wireguard, требующую только добавления приватного ключа, что и выполняется в вышеупомянутом скрипте командой ‘cat PRIV\_KEY >> wg0.conf’:

[Interface]

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 =

Теперь перейдем к тому, чтобы запустить в этом же контейнере наше ASP-приложение. По большому счету, есть 2 варианта сделать это: скомпилировать приложение заранее и передать в контейнер исполняемые файлы, либо передать в контейнер исходный код и скомпилировать его прямо там. С учетом того, что наш репозиторий является публичным, второй вариант является даже более простым, т.к. мы можем использовать git clone, но… Если в будущем появится желание сделать репозиторий приватным – появится необходимость всё переписывать. Поэтому принято решение передавать в контейнер уже скомпилированное приложение.

Но для начала попробуем просто скомпилировать приложение прямо в контейнере чтобы на начальном этапе избавиться от проблем с совместимостью. Выполним следующий shell-скрипт используя docker exec:

/bin/sh -c "cd /home && git clone https://github.com/LuminoDiode/vdb\_vpn\_server"

Cloning into 'vdb\_vpn\_server'...

dotnet run --project /home/vdb\_vpn\_server/vdb\_node\_api/vdb\_node\_api.csproj

Building...

info: Microsoft.AspNetCore.Server.Kestrel[0]

Unable to bind to http://localhost:5001 on the IPv6 loopback interface.

info: Microsoft.Hosting.Lifetime[14]

Now listening on: http://localhost:5001

info: Microsoft.Hosting.Lifetime[0]

Application started. Press Ctrl+C to shut down.

info: Microsoft.Hosting.Lifetime[0]

Hosting environment: Development

info: Microsoft.Hosting.Lifetime[0]

Content root path: /home/vdb\_vpn\_server/vdb\_node\_api

Однако, сервер не способен принимать запросы извне. Если в самом контейнере воспользоваться пакетом curl или wget (лично я предпочитаю curl) и отправить запрос командой ‘docker exec 4f2775fde7a8 curl <http://localhost:5001/api/peers>’ – получим закономерный ответ – Unauthorized, что соответствует ожидаемому поведению. Однако при попытке осуществить такой же запрос с помощью Postman’а с моего собственно локального компьютера – запрос отобразиться в tcpdump (tcpdump port 5001 -i eth0), однако не будет перехвачен ASP-приложением. Данная ситуация является странной, однако уже описана в интернете[10] (QR). По сути, нам требуется использовать не loopback адрес, который представляет собой localhost, а любой конкретный адрес. Какой? – вернемся к упомянутому выше RFC 1918[11], материал которого мы учли при написании IpDedicationService для ASP-приложения:

*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).*

Учитывая то, что пространство 10/8 мы использовали для адресации Wireguard-клиентов, выберем произвольный адрес из любого другого пространства, пусть таковым будет 192.168.6.1. Для его использования применим к команде dotnet run аргумент --urls "http://192.168.6.1:5001". Но изначально Kestrel (стандартный сервер ASP) предполагает прослушивание localhost, данное поведение следует изменить на IPAddress.Any, он же 0.0.0.0, в противном случае приложение не будет доступно из внешних сетей, для этого пропишем builder.WebHost.UseKestrel(opts => opts.ListenAnyIP(5001)). Теперь дополним docker-compose перенаправлением с порта 51831 на порт 5001. Для работы всего вышеперечисленного необходимо разрешить системному UFW (брандмауэру) подключения на соответствующие порты, в нашем случае это команды ‘ufw allow 51830/udp; ufw allow 51831/tcp’, т.к. 30 порт у нас слушает WG, инкапсулирующий свой протокол под UDP, а 31 – ASP приложение, используй стандартный для веб-приложений TCP.

Отлично, теперь ASP-приложение, работая в контейнере, может принимать произвольные запросы. Сейчас, перед переходом к развертыванию предсобранных приложений взамен компиляции непосредственно в контейнере, необходимо доработать и протестировать полноценную функциональность и продумать передачу в ASP-приложение файла ключей.

Теперь перейдем от сборки непосредственно в контейнере к предсобранному приложению.