Toward a Zero-Trust Active Directory Architecture

AD Domain Services over the IPv6 Internet with end-to-end IPSec

Architecture & Reference Implementation
Ian Douglas, Stanford University

In highly distributed IT environments like Stanford, where technology groups employ varied cloud architectures without centralized coordination, integrations with centralized services require that the following practices be supported:

1. Use of auto-assigned IP addresses and DNS servers, as provided by the cloud vendor
2. Arbitrariness of the virtual network's private IP space
3. Exclusive reliance on the publicly routed internet for service discovery and communication

While employing these patterns confers significant benefits, complications arise when applying them to a service like Active Directory, which as a high-value attack target, is often cordoned on private networks, with its services unexposed to the public internet. So, a problem emerges: when a cloud-based server needs to join an AD domain, but the domain controllers (DCs) reside in a different virtual network, how are the DCs discovered and their services consumed over the public internet, without compromising security?

One Does Not Simply Expose AD to the Internet

A traditional approach would establish a network peering relationship between the different virtual networks, but this violates (2) and (3) of our desired design patterns, since peering requires that there be no overlapping IP space. Not only is micro-managing private IP space for distributed IT projects cumbersome and poorly scalable, it forces unnecessary design limitations on mutitier IAAS deployments, as teams are herded into centrally assigned network cidrs. Additionally, peering topologies are limited to a maximum of 125 active peering connections per VPC, with network performance significantly impacted at 16 connections, so they are not well-suited to a highly distributed model.

And the magic number is... 4,722,366,482,869,645,213,696

A more optimal solution is to provide AD services over IPv6, using transport-mode IPSec to ensure confidentiality. This confers global routablity and end-to-end encryption, and it satisfies all three design patterns: automatically assigned IPv6 addresses are leased for the lifetime of the server to DCs and domain member servers, allowing for stable, public DNS registration; virtual networks' private IP space layout is completely arbitrary, so that distributed IT teams may design their networks to fit their use-case --and may even have overlapping IP space with one another-- without the onus of managing/being managed by a central policy; the use of private networks is avoided, thereby reducing self-owned points of failure and leveraging the resilience of the internet. <diagram>

Decisions, Decisions

In implementing this solution, there are several critical decision points:

- Where to host SRV records, in public DNS or on private DNS servers?
- Which authority to use for the IPSec certificates,
- and by what mechanism bootstrap servers with certificates?
- Whether to use tunnel-mode or transport-mode IPSEC to secure DC-to-DC communication?

SRV Records: Dude, Where's My KRB?

The question of whether to host the AD-related SRV records in public DNS services like Route 53, or in firewall-restricted private DNS servers, raises the question of whether it is better security practice to restrict access to these records. While hiding the names and IP addresses of servers hosting services like LDAP and Kerberos may be a presumably stronger security posture, it comes at the cost of global discoverability. Also, knowing the address of a service does not grant access, assuming firewalls and access rules are configured, so it is more of a 'security by obscurity' approach to hide SRV records.

Whether or not hiding SRV records is truly more secure remains debatable, but there is an appreciable benefit to hosting the SRV records on the domain controllers themselves (since they also run the DNS service, in most cases), in that the SRV records can be automatically created by the DC and updated whenever the netlogon service is restarted, as well as every hour. There are usually at least 17 AD-related SRV records <link: https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-2000server/cc961719(v=technet.10) > (depending on configuration of Global Catalog, IP Sites, and subdomain topology, there can be many more), so scripting the creation of these records in AWS Route 53, although straightforward, introduces another process that is potentially subject to error, especially as servers reach the end of their lifecycle.

In evaluating both of these approaches, we found that while our use of a Terraform module to create SRV records reduced the operational errors of Route 53 maintenance, hosting the records on the DCs themselves was a better fit --at least in the immediate term-- to our accepted, albeit historical, security posture, because it allowed for applying network access controls. With the DCs behind a firewall, IP filtering can be used to allow port 53 only from known IP networks, which cloaks the SRV records from potential attackers. Since AWS offers stable /64 IPv6 address blocks for VPCs, both on-prem and cloud-based firewalls can be configured to allow these CIDRs.

Automatic SRV and host record creation on the part of the Domain Controllers does come with one potential problem, although easily solved. Since the DCs in AWS are dual-stack IPv4 and IPv6, they will register both their 'A' and 'AAAA' host records. The 'AAAA' records in DNS are the servers' actual, globally-routable IPv6 address, assigned to the VM instance's network interface, so this is exactly what is desired. However, the 'A' records that the server updates in DNS are in fact the private RFC1918 address assigned from the the VPC's pool of local addresses, which is not globally reachable and not appropriate for serving as the destination of a SRV record. This is because the IPv4 address that is assigned to the instance's network interface always comes from the the private subnet where the instance resides. But to advertise AD Domain services via SRV records, a globally routable, public IPv4 address is required (assuming we are to support both IPv4 and IPv6, otherwise the IPv6 address alone would suffice).

In the case of AWS, a static public IP address is provided as an Elastic IP, however, while this address can be can be assigned to a server, it isn't configured into the server's network interface, so automatic DNS registration continues to use the private, rather than the public, IPv4 address. Global service reachability via the public IPv4 address relies on NAT to map the assigned Elastic IP to the server's private IPv4 address, so the operating system is never aware of its own public IP address. To solve this problem, and to prevent the server from continually updating its 'A' host record with its private IPv4 address, we elected to set the value in DNS to the public IP, then create an ACL on the DNS record that denies 'write' privilege to the server. While this requires a one-time DNS configuration to set the 'A' record and its ACL, it otherwise allows the server to create automatically its many SRV records, its CNAME record (more on this later), and its 'AAAA' host record.

When a server is born in AWS with a IPv6 address and attempts to find the AD Domain, it queries DNS for the SRV records for ldap_.tcp_.<domain name> and receives a set of domain controller names. Windows server will prefer IPv6 when available, so will attempt to contact all of the IPv6 addresses associated with the returned set of SRV records (if their SRV records are weighted equally), sending and LDAP request on UDP port 389 to each of them; the first Domain Controller to answer will then be used to continue the domain join process. (To control which DCs answer, Active Directory Sites and Services should be configured appropriately, as well as the SRV weights). For more background and additional links, see https://blogs.msmvps.com/acefekay/2010/01/03/the-dc-locator-process-the-logon-processcontrolling-which-dc-responds-in-an-ad-site-and-srv-records/ I am the Keymaster; Are You the Gatekeeper?

Once clients in discrete virtual networks can find the AD Domain services over the internet, how do we ensure the confidentiality and integrity of the client-to-domain-controller communications? IPSec transport mode provides exactly this security assurance, and additionally, serves as a mulitfactor authentication mechanism, since in addition to requiring passwords, communications with DCs are blocked unless both parties have mutually trusted x509 certificates.

Microsoft officially supports the use of IPSec to encrypt network traffic in 'domain isolation' models that include client-to-server, client-to-client, server-to-server, and additionally, DCto-DC replication traffic, as well as global catalog-to-global catalog replication traffic. However, it is not possible to require IPSec from member server to a domain controller when Kerberos is used as the authentication method for the IPSec negotiation --only certificate-based authentication will work in this scenario.

Futhermore, using the typical Certificate Authority found in most MS environments, Active Directory Certificate Services, presents a similar complication, since auto-enrollment for server certificates requires that the server be joined to the domain, presupposing that communications between server and DC are already allowed. Since we want to secure servertoDC communications before, during, and after the domain join process, we need a certificate enrollment process that has no dependencies on Active Directory.

IPSEC Certificates:

I don't often x509, but when I do, I prefer auto-enrollment

We evaluated two different solutions for providing x509 certificates automatically during the server bootstrap process: Let's Encrypt (with DNS validation) and Hashicorp Vault. Both provide certificates that can be used for IPSec, and an enrollment process that can be entirely automated.

The Let's Encrypt solution required that the servers be launched in an EC2 Role (https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles_use_switch-role-ec2.html#rolesusingrole-ec2instance-roles) that allows creation of DNS TXT records in a specific Route 53 zone, so that the CA can validate the certificate request. While scripting this to run as part of machine startup and at regular intervals was achieved relatively painlessly with PowerShell and the aws cli, the short validity periods of the certificates (90 days), combined with the public nature of the CA, as well as allowing DNS write priviliges to distributed IT teams, resulted in a solution that did not meet the desired operational and security practice.

A solution using Hashicorp Vault's PKI engine was favored, since a highly available Vault cluster was available in-house. This soltuion required that the Vault PKI backend be configured with a role to issue certificates, an authorization policy be written to grant access to the certificate issuer endpoint, and that a specific approle be created to authenticate api requests to assume said policy's role. The specifics of these configuration requirements are well documented on Vault's website (https://www.vaultproject.io/docs/).

The Secret of the Chicken and the Egg

One particular problem that needs to be solved when implementing solutions using Hashicorp Vault (or any other secrets engine) is the problem of secure introduction (https://www.vaultproject.io/guides/identity/secure-intro.html) . How does a secret consumer prove that it is the legitimate recipient for a secret so that it can acquire a token? Vault's AppRole method provides a means for programmatic authentication of machines or applications, whereby an identifying RoleID and a corresponding SecretID are used to obtain a short-lived access token, which in turn confers some well defined set of priviliges that may be invoked in subsequent api requests. In our use-case, we have created an approle in Vault that allows applications to request certificates. So how are are the the initial RoleID and SecretID distributed in a secure manner?

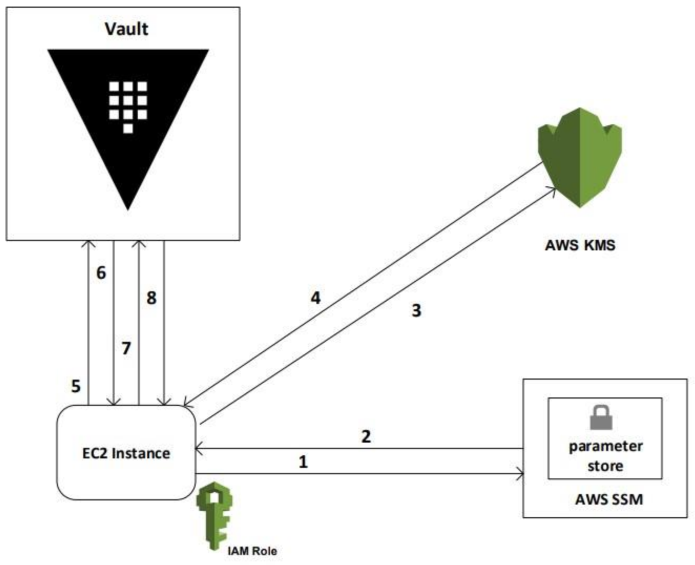
For simplicity, we chose a 'platform integration' approach, essentially delegating secure introduction to the trusted mechanisms available to AWS EC2, namely Systems Manager Parameter Store, KMS, and IAM (https://aws.amazon.com/systems-manager/features/#Parameter_Store). This resource provides an encrypted key-value store with fine-grained access controls using IAM polices. When a server is launched in an EC2 role with a certain IAM Policy, it is allowed to read and decrypt the particular Parameter Store values containing the RoleID and SecretID.

```
{
"Version": "2012-10-17",
"Statement": [
{
"Effect": "Allow",
"Action": [
"ssm:DescribeParameters"
],
"Resource": "*"
},
{
"Sid": "Stmt1482841904000",
"Effect": "Allow",
"Action": [
"ssm:GetParameters"
],
"Resource": [
"arn:aws:ssm:${AWS_DEFAULT_REGION}:${AWS_ACCOUNT}:parameter/${AWS_PROFILE}-*"
]
},
{
"Sid": "Stmt1482841948000",
"Effect": "Allow",
"Action": [
"kms:Decrypt"
],
"Resource": [
"arn:aws:kms:${AWS_DEFAULT_REGION}:${AWS_ACCOUNT}:key/${AWS_KMS_KEYID}"
]
}
] }
```

Can I Haz Cert?

Once 'secret zero' has been delivered to the server in the form of a RoleID and SecretID, obtaining a certificate from Vault is straightforward, and requires no special client software, since all of Vault's capabilities are accessible through the REST API. The entire certificate request process can be performed using simply curl or its equivalent. Having received and decrypted the SecretID and RoleID from the Parameter Store, the server then authenticates to Vault using an HTTP POST request, and once the request is validated and assigned to an approle, Vault returns a limited scope token to the client. This token is placed into the header of a subsequent POST request against the certificate issuer endpoint, and if the request meets all required criteria, a certificate is issued in PEM block format. The PEM block contains the requestor's new certificate, the associated private key, and the certificate of the issuing CA.

Fetching Machine Certificates for IPSec



1. EC2 instance in appoved IAM Role calls AWS SSM Parameter Store
2. SSM issues ciphertext response
3. Decrypt is requested based on IAM Role policy
4. AWS KMS returns decrypted data containing Vault Approle role_id and secret_id
5. role_id and secret_id are used to authenticate to Vault using approle auth method
6. Vault issues short-lived token based on approle
7. Token is used to make a POST request to the PKI issuer endpoint
8. Vault returns a PEM bundle containing issuer cert, requestor cert and requestor key

Since Windows Server does not use PEM encoded certificates natively, a further step is required to split out the elements of the PEM file and convert them into PKCS12 format, before importing them into the certificate store. Note that the private key cannot be imported without assigning it a password, which must be converted from plain text to SecureString. The value of this password is also retrieved from the Parameter Store.

For a code example, see the following, a function for retrieving data from the Parameter Store, requesting a certificate from Vault, and importing it into the certificate store. Note that this runs in a userData script on the target AWS EC2 instance, as the server is being launched for the first time. Since we use Terraform to define and deploy our IAAS infrastructure, we leverage Terraform's template abstraction to render the userdata dynamically and to provide the required values to the function parameters.

Keep Your Eyes off My Packets

Having obtained a certificate, the server may now establish an IPSec policy, so that communications with Domain Controllers require end-to-end IPSec. Transport mode IPSec has improved significantly in Windows Server 2012 and 2016, as IKEv2 with certificate-based authentication is now supported. IKEv2 provides a robust mechanism for key exchange that establishes a security association that persists despite changes in the underlying network connection. However, it can only be configured through PowerShell, as its configuration parameters are not exposed through the Windows Firewall with Advanced Security user interface. Once IPSec profiles are established, they can in fact be monitored via the 'wf.msc' UI.

IPSec profiles are scoped to a set of remote IP Address ranges. Since this solution uses IPv6, the remote IP scope for which IPSec is required is defined as the entire /64 CIDR of the Active Directory virtual network. Similarly, on the domain controllers, an IPSec profile is established for each IT groups' IPv6 network. Presuming that the DCs and the member servers both have machine certificates signed by the Vault CA, and that the CA's certificate has been imported into their trusted store, a cryptographic set should be negotiated and an IPsec channel established.

For the example code below, a function for establishing an IPSec policy, note that the required parameters are generated dynamically while deploying the server via Terraform.

```
function Get-VaultCert
{
[CmdletBinding()]
[OutputType([System.Security.Cryptography.X509Certificates.X509Certificate2])]
Param
(
$cert_cn = "$env:COMPUTERNAME",
$cert_ttl = "43800h",
$vault_uri,
$vault_login_uri,
$vault_issue_uri,
$aws_cli_path = "$env:ProgramFiles\Amazon\AWSCLI\aws.exe",
[Parameter(ParameterSetName='PRESEED')]$vault_role_id,
[Parameter(ParameterSetName='PRESEED')]$vault_secret_id,
[Parameter(ParameterSetName='PRESEED')]$vault_pem_pwd,
[Parameter(ParameterSetName='AWS_SSM')]$aws_default_region = "us-west-2",
[Parameter(ParameterSetName='AWS_SSM')]$role_id_key,
[Parameter(ParameterSetName='AWS_SSM')]$secret_id_key,
[Parameter(ParameterSetName='AWS_SSM')]$pem_pwd_key
 )
 [Net.ServicePointManager]::SecurityProtocol = [Net.SecurityProtocolType]::Tls12
```

```
    switch ($PsCmdlet.ParameterSetName)
    {
    “AWS_SSM”  {
    Write-Verbose "Getting machine certificate for $cert_cn using secret from AWS Parameter Store"
    $sts_caller_identity = Try { $(& $aws_cli_path sts get-caller-identity) } Catch [System.Management.Automation.CommandNotFoundException]{ Write-Error -Message "aws cli not in PATH. Exiting."; exit }
    $env:AWS_DEFAULT_REGION = $aws_default_region
    $vault_role_id = (& $aws_cli_path ssm get-parameters --names "$role_id_key" --withdecryption | ConvertFrom-Json).Parameters.Value
    $vault_secret_id = (& $aws_cli_path ssm get-parameters --names "$secret_id_key" -with-decryption | ConvertFrom-Json).Parameters.Value
    $vault_pem_pwd = (& $aws_cli_path ssm get-parameters --names "$pem_pwd_key" --withdecryption | ConvertFrom-Json).Parameters.Value
    }
    “PRESEED”  {
    Write-Verbose "Getting machine certificate for $cert_cn using secret from supplied parameters"
    }
    }
    $body_auth = @{
    "role_id" = "$vault_role_id"
    "secret_id" = "$vault_secret_id"
    } | ConvertTo-Json
    # get token
    $response_auth = Invoke-RestMethod -Method Post -Uri $vault_login_uri -Body $body_auth ContentType "application/json"
    $token = $response_auth.auth.client_token
    $body_issue = @{
    "common_name" = "$cert_cn"
    "ttl" = "$cert_ttl"
    } | ConvertTo-Json
    $headers = @{
    "X-Vault-Token" = "$token"
    }
    # get cert bundle
    $response_pem = Invoke-RestMethod -Method Post -Uri $vault_issue_uri -Body $body_issue Headers $headers
    $response_pem.data.private_key | Set-Content client.KEY
    $response_pem.data.certificate | Set-Content client.cer
    $response_pem.data.issuing_ca | Set-Content vault-ca.crt
    # convert cert to pkcs#12
    certutil -p "$vault_pem_pwd,$vault_pem_pwd" -f -MergePFX client.cer client.pfx
    # import into LocalMachine cert store
    $SecureString = "$vault_pem_pwd" | ConvertTo-SecureString -AsPlainText -Force
    Import-PfxCertificate -Password $SecureString -FilePath client.pfx -CertStoreLocation Cert:\LocalMachine\My -Exportable
    Import-Certificate -FilePath vault-ca.crt -CertStoreLocation Cert:\LocalMachine\Root
"client.KEY,client.cer,client.pfx,vault-ca.crt" -split "," | Remove-Item
}

```

DC-to-DC comms: to Tunnel or Not to Tunnel

Conventional hybrid architectures employ the use of VPN tunnels to connect on-premises networks to virtual networks in the cloud. This approach, while widely used, is incompatible with (2) and (3) of our desired design patterns. With respect to (3), while virtual networks' route tables can be configured to selectively route some traffic bound for on-premises resources through the tunnel and some through the internet, those that go through the tunnel do not enjoy the built-in resilience of the internet, and suffer from more potential points of failure, as well as bandwidth limitations. As for (2), the private IP space of the distributed virtual networks can no longer be considered arbitrary, since the VPN router needs to advertise specific routes, and thus none of the many distributed virtual networks may have overlapping IP space. This is poorly scalable for a distributed IT model, as it often results in teams being granted a relatively small swathe of IP space, managed by a central policy. This enforces undesirable and unneccesary limitations on teams designing their multitier cloud-based solutions.

With the use of IPv6 and end-to-end IPSec with certificate based authentication, VPN tunnels are no longer required for hybrid architectures, since the goals of inter-network routability are provided via IPv6, and the security requirements of confidentiality, integrity and authenticity are provided by virtue of IPSec with Certificate-based authentication (availability may also be improved, in that the VPN device(s) are no longer a potential point of failure). The potential use-cases for this architecture are various and certainly not limited to Active Directory.

But what about domain controller-to-domain controller communication? In AD, data replication and managment is carried out over the Directory Replication Service Remote Protocol (MSDRSR), an RPC protocol whose functionality is a superset of the capabilities exposed by the LDAP protocol. DC-to-DC connections require mutual authentication and encryption of protocol traffic, where mutual authentication is provided by Kerberos. Although DC-to-DC conversations are thusly protected from eavesdropping and replay attacks, Microsoft recommends that domain controller traffic be carried out solely on private networks, or, that it be protected by IPSec in either tunnel (VPN gateway) or transport (end-to-end) modes.

We evaluated using a VPN tunnel to carry DC-to-DC traffic, as well as using transport IPSec. Motivated by the desired design patterns, the end-to-end approach was favored as the optimal solution. However, since our on-prem servers were not using globally routable IPv6 addresses, this required that the IPSec connections between cloud-based DCs and our onprem DCs use IPv4. While the on-prem servers' IPv4 addresses happen to be public addresses from Stanford's /14 range, the cloud DC's IPv4 addresses were, as detailed above, assigned using AWS Elastic IPs, which poses complications, since resolution of Elastic IP addresses require NATing to translate the public IP to the DC's private IP.

This use of NAT is problematic because the optimal configuration of transport mode IPSec employs both AH and ESP (Authentication Header and Encapsulated Security), and AH breaks when used over NAT, because the hash used to check AH integrity uses the source IP and destination IP, which is altered in transit by NAT. It can be argued that using ESP without AH in this scenario is of negligible security risk, given the fact that the certificate based authentication of IPSec guarantees authenticity. In <code example> provided above, the '-natTraversal' switch can be used when either of the IP scopes rely on NAT. The most desirable solution would be to configure all the on-prem DCs, or at least a subset of them desiginated as bridgehead servers <link https://docs.microsoft.com/en-us/windows-server/identity/ad-ds/get-started/replication/active-directory-replication-concepts>, with IPv6, so that no NATing would occur.

YOU get a CIDR and YOU get a CIDR and...

In the interim to IPv6 being deployed on-prem, lest IPSec transport mode be configured between domain controllers with AH disabled, a VPN tunnel was employed. This violated the desired design patterns, but a compromise solution was found to address (2). In AWS virtual networks, multiple, disjoint CIDRs can be assigned to a single VPC, so that, in effect, the majority of the VPC address space be treated as arbitrary, with only a small subset reserved for inclusion in the routes advertised back through the tunnel gateway.

For example, the VPC may be assigned two CIDRs, say, 10.32.16.0/24 and 10.10.0.0/16. The 10.10/16 can be treated as arbitrary and subnets carved out as needed for any other locally required services that provide backend services to AD, like security service subnets, management and monitoring subnets, or bastion subnets. The other CIDR, 10.32.16.0/24, can be treated as part of the on-prem address space, and carved into 3 subnets (one for each availability zone) containing the Domain Controllers. In the <diagram above>, subnets Public2A, Public-2B, and Public-2C are formed from the 10.32.16.0/24, while all other subnets are arbitrary. The routing tables for the AD subnets in AWS are configured such that any traffic bound for the on-prem AD subnets go through the VPN gateway, while any other destination, either on campus or elsewhere, go through the internet gateway.

<route table figure>

Configuring multiple CIDRs produced some undesirable side effects. Since the VPN gateway in AWS and the on-prem routers use BGP to propogate route advertisements to each other, it became evident that both CIDRs configured for the VPC were being advertised back to campus, which was unacceptable, since only the 10.32.16.0/24 (in our example) should be advertised. AWS did not yet expose the configuration of the VPN gateway that would allow fine-grained control of which routes were advertised, so the on-prem VPN router needed to be set to drop all traffic except the expected 10.32.16.0/24.

Similarly, due to BGP automatic route propogation, every on-prem subnet was being advertised to the AWS VPN gateway, so that any AWS subnet whose route table association contained the VPN gateway, advertised routes to every on-prem network, which was also undesirable, since the intention was for the VPN tunnel to carry only the DC-to-DC traffic. This problem was fixed by preventing automatic route propogation in our terraform configuration, by explicitly defining the DC subnets' route tables, <see terraform snippet>, rather than simply associating the VPN gateway with the subnet in its default configuration

```
resource "aws_route_table" "vpn-us-west-2a-public-2" {

  vpc_id = "${aws_vpc.main.id}"

    route {
    cidr_block = "0.0.0.0/0"
    gateway_id = "${aws_internet_gateway.main.id}"
    }

    route {
    ipv6_cidr_block = "::/0"
    gateway_id      = "${aws_internet_gateway.main.id}"
    }

    route {
    cidr_block = "${var.onprem_ad_nets}"
    gateway_id = "${aws_vpn_gateway.vpn_gw.id}"
    }

    tags {
    Name = "vpn-us-west-2a-public-2 | ${aws_vpc.main.id}"
    }
    }

```

What's in a CNAME

Another untoward consequence of using the VPN tunnel soon emerged. When performing directory replication using DRS <link: https://msdn.microsoft.com/en-us/library/cc228086.aspx> , Domain controllers do not use each other's fully qualified hostname to find their corresponding IP address, but instead use a CNAME of the form <GUID>._msdcs. <domain name> . As discussed above, the public IPv4 addresses of the AWS domain controllers are assigned using Elastic IPs, which must be set as an 'A' record in DNS instead of the private IPv4 address, otherwise the resolution of SRV records would in some cases return an unreachable private IPv4 address. Essentially, the intention is for on-prem DCs to use the private IPv4 addresses of the AWS DCs, and for everyone else to use their public IPv6 addresses, but to provide public IPv4 addresses in their SRV records for edge cases.

Split IP Soup

This results in a 'split IP space' problem: an on-prem IPv4 DC attempts replication from a DC in AWS, looks up its CNAME in DNS, gets a pointer to a public IPv4 address, and attempts to initiate communication over the internet, rather than through the tunnel, since only destinations to 10.32.16.0/24 (in our example) are routed through the VPN gateway. This would be undesirable, since DC-to-DC communications are meant to occur over some form of IPSec, and will fail due to the AWS DCs' firewall restrictions and IPSec policy requirements.

In order to ensure that all traffic from the on-prem DCs goes through VPN tunnel, while still maintaining public IP addresses in DNS, the solution is to provide a HOSTS file on each onprem DC, containing entries associating every AWS DC's DSA GUID name with its private IPv4 address. This can be implemented using a group policy that targets only the on-prem DCs, so that the AWS DCs continue to talk to each other over IPv6.

This further highlights the fact that the optimal solution avoids using the VPN tunnel and employs instead the pure IPv6 design with end-to-end IPSec.

Blast off to adventure in the amazing year four sextillion!

As the use of IAAS solutions continues to increase, with existing on-premises servers being rebuilt in the cloud, and new servers being spun up in AWS in preference to on-prem datacenters, the boundaries of our networks will become ever more distributed across the internet. Using IPv6 for clustered services that span diverse IP networks (like Active Directory), along with end-to-end IPSec for enhanced security, offers a highly scalable mechanism for achieving the resilience of geodiversity and the loose coupling of interdependent services.

71.