Introducing ROADtools - The Azure AD exploration framework

Published: 2020-04-16 · Archived: 2026-04-06 01:04:12 UTC

Over the past 1.5 years I've been doing quite a lot of exploration into Azure AD and how it works under the hood. Azure AD is getting more and more common in enterprises, and thus securing it is becoming a bigger topic. Whereas the traditional Windows Server Active Directory already has so much research and community tooling available for it, Azure AD is in my opinion lagging behind in this aspect. In this post I'm introducing the ROADtools framework and it's first tool: ROADrecon. This framework was developed during my research and will hopefully serve as both a useful tool and an extensible framework for anyone that wants to analyse Azure AD, whether that is from a Red Team or a Blue Team perspective. This post is the first in part of a series in which I'll dive into more aspects of Azure AD and the ROADtools framework. Both ROADtools and ROADrecon are free open source tools and available on my GitHub. I also did a live stream of most things that are written here that you can watch on YouTube.

Why this framework

Whenever I find myself in a new network or researching a new topic, I want to know as much information as possible about it, in an easy to digest format. In Active Directory environments, information is relatively simple to query using LDAP, and many tools exist that query this information and transform it into a format that's easier to use for humans. Back when I started writing tools, I wrote a simple tool ldapdomaindump that tried to save all the information it could gather offline, so that I could quickly answer questions like "oh which groups is this user in again" or "do they have a group for system X that could be useful".

Fast forward a few years and companies are often using Microsoft 365 and moving their things to Azure, where there isn't really a tool that gives you quick insight into an environment. The Azure portal simply requires too many clicks to find what you're looking for, and it can be disabled for anyone but admins. The various Powershell modules, .NET libraries and other official ways to query Azure AD have varying degrees of support for information they can give you, ways to authenticate and restrictions that can be applied to them. While researching Azure AD I wanted to have a way to access all the possible information, using any authentication method (whether obtained legitimately or not) and have it available offline. Since none of the official methods offered this possibility, I quickly realized building a custom framework was the only way to achieve it. So I set myself a few goals:

- Provide tooling for both Red teams and Blue teams to explore all Azure AD data in an accessible way.
- Show the wealth of information available to anyone with just 1 valid credential set – from the internet.
- Improve understanding of how Azure AD works and what the possibilities are.
- Provide a framework that people can build upon and extend for their own use-cases.

I did learn a few things along the way from writing ldapdomaindump, which kept all information in memory until it had calculated all the recursive group memberships, at which point it would write it to disk. As one expects, this scales pretty bad in environments that have more than a few thousand users in them. I spent a lot of time thinking

how I wanted to do it (and even more writing the actual code), while ignoring all the ways one is supposed to access Azure AD, so today is the first release of the **R**ogue **O**ffice 365 and **A**zure (active) **D**irectory tools!

ROADrecon

The first (and likely most extensive) tool in this framework is ROADrecon. In short, this is what it does:

- Uses an automatically generated metadata model to create an SQLAlchemy backed database on disk.
- Use asynchronous HTTP calls in Python to dump all available information in the Azure AD graph to this database.
- Provide plugins to query this database and output it to a useful format.
- Provide an extensive interface built in Angular that queries the offline database directly for its analysis.

Where to get the data

Since Azure AD is a cloud service, there isn't a way to reverse engineer how it works, or a central repository where all the data is stored that you can access. Since Azure AD is completely different than Windows Server AD, there's also no LDAP to query the directory. While researching Azure and looking through the requests in the Azure Portal, at some point I noticed that the portal was calling a different version of the Azure AD Graph, the `1.61-internal` version.

Status	Method	Domain	File	Cause	Type	Transferred	Size	0 ms
200	GET	afd.hosting.portal.a...	zwB7yYcvLudD.js	fetch	js	6.03 KB	24.14 KB	44 ms
200	POST	portal.azure.com	DelegationToken?feature.refreshtokenbinding=true&featur...	xhr	json	2.93 KB	6.21 KB	317 ms
200	POST	portal.azure.com	DelegationToken?feature.refreshtokenbinding=true&featur...	xhr	json	3.48 KB	6.36 KB	152 ms
200	GET	graph.windows.net	roleDefinitions?api-version=1.61-internal&$top=500	xhr	json	68.92 KB	68.89 KB	143 ms
200	OPTIONS	main.iam.ad.ext.az...	CurrentContext	xhr	plain	39 B	0 B	95 ms
200	GET	main.iam.ad.ext.az...	CurrentContext	xhr	json	992 B	99 B	57 ms
200	OPTIONS	main.iam.ad.ext.az...	RoleAssignments?scope=undefined	xhr	plain	752 B	0 B	62 ms
200	GET	main.iam.ad.ext.az...	RoleAssignments?scope=undefined	xhr	json	2.08 KB	4.50 KB	395 ms

This internal version of the Azure AD graph exposes much more data than any of the official API's that are offered by Microsoft. I talked about some of the interesting things that you can find in this API in my BlueHat Seattle talk last year. Though one is probably not supposed to use this version, it is still available for any user. By default it is possible to query almost all the information about the directory as authenticated user, even when the Azure portal is restricted.

The next question was how to store this data in a structured way locally. The API streams everything as JSON objects, which is a useful format for transferring data but not really for storing and searching through data. So ideally we'd have a database in which objects and their relationships are automatically stored and mapped. For this ROADrecon uses the SQLAlchemy Object Relational Mapper (ORM). What this means is that ROADrecon defines the structure of the objects and their relationships, and SQLAlchemy determines how it stores and retrieves those from the underlying database. To create the object structure, ROADrecon uses the OData metadata definition that the Azure AD graph exposes. This XML document defines all object types, their properties and relationships in the directory.

```
-<edmx:Edmx Version="3.0">
  -<edmx:DataServices m:DataServiceVersion="3.0" m:MaxDataServiceVersion="3.0">
    -<Schema Namespace="Microsoft.DirectoryServices">
      -<EntityType Name="DirectoryObject" OpenType="true">
        -<Key>
           <PropertyRef Name="objectId"/>
         </Key>
         <Property Name="objectType" Type="Edm.String"/>
         <Property Name="objectId" Type="Edm.String" Nullable="false"/>
         <Property Name="deletionTimestamp" Type="Edm.DateTime"/>
         <NavigationProperty Name="createdOnBehalfOf" Relationship="Microsoft.DirectoryServices.Mic
         <NavigationProperty Name="createdObjects" Relationship="Microsoft.DirectoryServices.Microsc
         <NavigationProperty Name="manager" Relationship="Microsoft.DirectoryServices.Microsoft_Dir
         <NavigationProperty Name="directReports" Relationship="Microsoft.DirectoryServices.Microsof
         <NavigationProperty Name="members" Relationship="Microsoft.DirectoryServices.Microsoft_Dir
         <NavigationProperty Name="transitiveMembers" Relationship="Microsoft.DirectoryServices.Mic
         <NavigationProperty Name="memberOf" Relationship="Microsoft.DirectoryServices.Microsoft_D
         <NavigationProperty Name="transitiveMemberOf" Relationship="Microsoft.DirectoryServices.Mi
         <NavigationProperty Name="owners" Relationship="Microsoft.DirectoryServices.Microsoft_Direc
         <NavigationProperty Name="ownedObjects" Relationship="Microsoft.DirectoryServices.Microsof
       </EntityType>
      -<EntityType Name="ExtensionProperty" BaseType="Microsoft.DirectoryServices.DirectoryObject" Op
         <Property Name="appDisplayName" Type="Edm.String"/>
         <Property Name="name" Type="Edm.String"/>
         <Property Name="dataType" Type="Edm.String"/>
         <Property Name="isSyncedFromOnPremises" Type="Edm.Boolean"/>
         <Property Name="targetObjects" Type="Collection(Edm.String)" Nullable="false"/>
       </EntityType>
```


I wrote some quite ugly code which transforms this metadata XML (mostly) automatically into a neat and well-defined database structure, which for example looks like this:

```
class User(Base, SerializeMixin):
    __tablename__ = "Users"
    objectType = Column(Text)
    objectId = Column(Text, primary_key=True)
    deletionTimestamp = Column(DateTime)
    acceptedAs = Column(Text)
    acceptedOn = Column(DateTime)
    accountEnabled = Column(Boolean)
    ageGroup = Column(Text)
    alternativeSecurityIds = Column(JSON)
    signInNames = Column(JSON)
    signInNamesInfo = Column(JSON)
    appMetadata = Column(JSON)
    assignedLicenses = Column(JSON)
    assignedPlans = Column(JSON)
    city = Column(Text)
    cloudAudioConferencingProviderInfo = Column(Text)
    cloudMSExchRecipientDisplayType = Column(BigInteger)
    cloudMSRtcIsSipEnabled = Column(Boolean)
    cloudMSRtcOwnerUrn = Column(Text)
    cloudMSRtcPolicyAssignments = Column(JSON)
    cloudMSRtcPool = Column(Text)
    cloudMSRtcServiceAttributes = Column(JSON)
    cloudRtcUserPolicies = Column(Text)
    cloudSecurityIdentifier = Column(Text)
    cloudSipLine = Column(Text)
    cloudSipProxyAddress = Column(Text)
```

SQLAlchemy then creates the database for this model, which by default is an SQLite database, but PostgreSQL is also supported (in my testing the performance difference was minimal but SQLite seemed slightly faster). The main advantage of this is that it is really easy to query the data afterwards, without having to write any SQL queries yourself.

This database model is actually not part of ROADrecon but `roadlib`, the central library component of ROADtools. The reason for this is if you would want to build an external tool that interfaces with the database populated by ROADrecon you wouldn't actually need to import ROADrecon yourself and all its dependencies. Instead you could import the library containing the database logic, which doesn't depend on all the third party code that ROADrecon used to transform and display data.

Dumping the data

ROADrecon uses a process consisting of 3 steps to dump and explore the data in Azure AD:

1. Authenticate - using username/password, access token, device code flow, etc
2. Dump the data to disk
3. Explore the data or transform it into a useful format using plugins

Authenticating

Authenticating is the first step to start gathering data. ROADrecon offers quite some options to authenticate:

```
 usage: roadrecon auth [-h] [-u USERNAME] [-p PASSWORD] [-t TENANT] [-c CLIENT] [--as-app] [--device-code] [--ac
                      [--refresh-token REFRESH_TOKEN] [-f TOKENFILE] [--tokens-stdout]

optional arguments:
  -h, --help            show this help message and exit
  -u USERNAME, --username USERNAME
                        Username for authentication
  -p PASSWORD, --password PASSWORD
                        Password (leave empty to prompt)
  -t TENANT, --tenant TENANT
                        Tenant ID to auth to (leave blank for default tenant for account)
  -c CLIENT, --client CLIENT
                        Client ID to use when authenticating. (Must be a public client from Microsoft with user_
                        Default: Azure AD PowerShell module App ID
  --as-app              Authenticate as App (requires password and client ID set)
  --device-code         Authenticate using a device code
  --access-token ACCESS_TOKEN
                        Access token (JWT)
  --refresh-token REFRESH_TOKEN
                        Refresh token (JWT)
  -f TOKENFILE, --tokenfile TOKENFILE
```

```
                        File to store the credentials (default: .roadtools_auth)
  --tokens-stdout       Do not store tokens on disk, pipe to stdout instead
```

The most common ones that you will use are probably username + password authentication or device code authentication. Username + password is the easiest, but does not support (by design) any way of MFA since it's non-interactive. If your account requires MFA you can use the device code flow, which will give you a code to enter in the browser. There are more options here that you shouldn't need to use in most scenarios, but are for advanced usage or if you want to use tokens that were obtained via different methods. I am planning to do a future blog on Azure AD authentication and the options available for red teamers. ROADrecon will by default pretend to be the Azure AD PowerShell module and will thus inherit its permissions to access the internal version of the Azure AD graph. By default, ROADrecon will store the obtained authenticating tokens on disk in a file called `.roadtools_auth`. Depending on the authentication method this file contains long-lived refresh tokens, which prevent you from having to sign in all the time. This file is also compatible with any (future) tools using roadlib as authentication library. If you don't want to store tokens on disk you can also output them to stdout which allow you to pipe them into the next command directly.

Gathering all the data

The second step is data gathering, which the `roadrecon gather` command does. This has a few simple options:

```
 usage: roadrecon gather [-h] [-d DATABASE] [-f TOKENFILE] [--tokens-stdin] [--mfa]

optional arguments:
  -h, --help            show this help message and exit
  -d DATABASE, --database DATABASE
                        Database file. Can be the local database name for SQLite, or an SQLAlchemy compatible UR
                        postgresql+psycopg2://dirkjan@/roadtools. Default: roadrecon.db
  -f TOKENFILE, --tokenfile TOKENFILE
                        File to read credentials from obtained by roadrecon auth
  --tokens-stdin        Read tokens from stdin instead of from disk
  --mfa                 Dump MFA details (requires use of a privileged account)
```

By default it will dump it into an SQLite database called roadrecon.db in the current directory. Using postgresql requires some additional setup and the installation of `psycopg2`. The options for tokens depend on the settings you used in the authentication phase and are not needed if you didn't change those. The only other option for now is whether you want to dump data on Multi Factor Authentication, such as which methods each user has set up. This is the only privileged component of the data gathering and requires an account with membership of a role that gives access to this information (such as Global Admin/Reader or Authentication Administrator).

ROADrecon will request all the data available in two phases. The first phase requests all users, groups, devices, roles, applications and service principals in parallel using the `aiohttp` Python library. While requesting these objects is done in parallel, the Azure AD graph returns them in chunks of 100 entries and then includes a token to request the next page. This means that requesting the next 100 entries can only be performed after the result of the

first 100 is returned, effectively still making this a serial process. Each object type is requested in parallel, but it will still have to wait for the slowest parallel job to finish before continuing.

In the second phase all the relationships are queried, such as group memberships, application roles, directory role members and application/device owners. Because this is performed per individual group, there is a much larger number of parallel tasks here and thus the speed gains of using `aiohttp` become much larger. To limit the number of objects in memory, ROADrecon regularly flushes the database changes to disk (in chunks of ~1000 changes or new entries). This is not done asynchronously (yet) because in my testing the performance bottleneck seemed to be the HTTP requests rather than the database reads/writes.

Overall this whole process is pretty fast and for sure much faster than dumping everything in serial before I rewrote it to async code. Dumping an Azure AD environment of around 5000 users will take about 100 seconds. For really large environments that I’ve tested (~120k users) this will still take quite some time (about 2 hours) because of the number of objects that have to be requested in serial in the first phase of data gathering.

```
 (ROADtools) user@localhost:~/ROADtools$ roadrecon gather --mfa
Starting data gathering phase 1 of 2 (collecting objects)
Starting data gathering phase 2 of 2 (collecting properties and relationships)
ROADrecon gather executed in 7.11 seconds and issued 490 HTTP requests.
```

Exploring the data with the ROADrecon GUI

Now that we have access to all the data locally on disk in the database, we can start exploring it and convert it to a format that is easy to digest for humans. There are multiple options for this. ROADrecon is built with extensibility in mind, so it has a rudimentary plugin framework which allows for writing plug-ins that can take the data in the database and output this into something useful. For real simple use-cases, you don’t even need ROADrecon, but you can write a few lines of code that do what you want it to. Here is an example of a simple tool that only requires you to import the database definition from `roadlib` and then prints the names of all the users in the database:

```
 from roadtools.roadlib.metadef.database import User
import roadtools.roadlib.metadef.database as database
session = database.get_session(database.init())
for user in session.query(User):
    print(user.displayName)
```

You don’t need to write any code in most cases though, as ROADrecon already comes with some export plugins and a fully functional GUI. When running the `roadrecon-gui` or `roadrecon gui` commands, it will launch a local webserver through Flask which exposes a REST API that can be accessed by the single-page Angular JavaScript application.

It currently features:

- Listing of users / devices / groups

- Single-page directory role overview
- Applications overview
- Service Principal details
- Role / OAuth2 permissions assignments
- MFA overview

Some screenshots (or watch a [live demo here](#)):

ROADrecon

Home
Users
Groups
Devices
Directory roles
Applications
Service Principals
Application roles
OAuth2 Permissions
MFA

Database Stats

Users	6
Groups	8
Applications	7
ServicePrincipals	229
Devices	5

Tenant information

Name	iminyourcloud
Tenant ID	6287f28f-4f7f-4322-9651-a8697d8fe1bc
Syncs from AD	Yes

View Raw

Tenant Domains

Name	Type	Capabilities	Properties
iminyourcloud.mail.onmicrosoft.com	Managed	None	
iminyour.cloud	Managed	Email, OfficeCommunicationsOnline, Intune	Default
iminyourcloud.onmicrosoft.com	Managed	Email, OfficeCommunicationsOnline	Initial

ROADrecon

Home
Users
Groups
Devices
Directory roles
Applications
Service Principals
Application roles
OAuth2 Permissions
MFA

Filter

Name	UserPrincipalName	Enabled	Email	Department	Last password change	Job title	Mobile	Account type	MFA
HJ M	dirkjan@iminyour.cloud	✓	dirkjan@iminyour.cloud		2020-02-13T09:13:41			Cloud	
Joe Biz	joebiz@iminyour.cloud	✓	joebiz@iminyour.cloud		2019-11-22T22:26:25			AD	
mobiel	mobiel@iminyour.cloud	✓	mobiel@iminyour.cloud		2020-04-08T12:35:59			AD	
Rob Dark	rdark@iminyour.cloud	✓	rdark@iminyour.cloud		2019-11-22T22:26:52			AD	
On-Premises Directory Synchronization Service Account	Sync_iyc-app-server_fd4d8b246454@iminyourcloud.onmicrosoft.com	✓			2019-11-22T22:07:05			AD	
yubi	yubi@iminyour.cloud	✓			2020-02-26T11:55:17			AD	

Items per page: 50 1 – 6 of 6

ROADrecon

Home
Users
Groups
Devices
Directory roles
Applications
Service Principals
Application roles
OAuth2 Permissions
MFA

Filter

Name	Manufacturer	Enabled	Model	OS	OS Version	Trust type	Compliant	Managed	Rooted
insider-tpm		✓		Windows	10.0.19041.84	ServerAd		✓	
mobiel_Android_4/9/2020_10:13 AM	LGE	✓	Nexus 5X	Android	8.1.0	Workplace	✓	✓	
insider				Windows	10.0.19569.1000	ServerAd		✓	
iyc-client		✓		Windows	10.0.17134.0	ServerAd		✓	
iyc-app-server		✓		Windows	Windows 10	ServerAd		✓	

Items per page: 50 1 – 5 of 5

A recurring component of these listings is that the most important properties are displayed in a table, which supports pagination and a quick filter option. If you want to know more details of an object or how it relates to other components, most of the objects are clickable. When clicked, more detailed information will be shown in a pop-up.

Joe Biz

Overview Raw

Display name	Joe Biz
UserPrincipalName	joebiz@iminyour.cloud
ObjectId	7c38e062-7411-469d-a317-fb6667ee78f6
Email	joebiz@iminyour.cloud
Last password change	2019-11-22T22:26:25
Account type	Synced with AD
Status	Enabled

Groups (1)

Roles (1)

DisplayName	Description
User Account Administrator	Can manage all aspects of users and groups, including resetting passwords for limited admins.

For every object there is also the "raw" view, which displays all the available properties in a collapsible JSON structure (these properties come directly from the Azure AD internal API).

Joe Biz

Overview | Raw


```
▾ Object
  acceptedAs: null
  acceptedOn: null
  accountEnabled: true
  ageGroup: null
  ▾ alternativeSecurityIds: Array[0]
    []
  appMetadata: null
  ▾ assignedLicenses: Array[1]
    ▾ 0: Object
      ▸ disabledPlans: Array[0]
      skuId: "c42b9cae-ea4f-4ab7-9717-81576235ccac"
  ▸ assignedPlans: Array[37]
  city: null
  cloudAudioConferencingProviderInfo: null
  cloudMSExchRecipientDisplayType: 1073741824
  cloudMSRtcIsSipEnabled: true
  cloudMSRtcOwnerUrn: null
  ▾ cloudMSRtcPolicyAssignments: Array[6]
    0: "VoicePolicy=Host:HybridVoice"
    1: "MobilityPolicy=Host:MobilityEnableOutsideVoice"
    2: "MeetingPolicy=Host:BposSAllModality"
    3: "LocationProfile=Host:NL"
    4: "HostedVoicemailPolicy=Host:BusinessVoice"
    5: "ExternalAccessPolicy=Host:FederationAndPICDefault"
  cloudMSRtcPool: "sippoolAM42E12.infra.lync.com"
```


One of my favourite views is the Directory Roles view, since this view gives a really quick overview of which user or service accounts has a privileged role assigned. If you performed collection using a privileged account (Blue Team!) and collected MFA info, you can instantly see which accounts have MFA methods registered and which ones don't have this.

ROADrecon

Home
Users
Groups
Devices
Directory roles
Applications
Service Principals
Application roles
OAuth2 Permissions
MFA

User Account Administrator (1)

Directory Writers (1)

Directory Synchronization Accounts (1)

Principal Name	Principal Type	userPrincipalName	Account type	Status	MFA
On-Premises Directory Synchronization Service Account	User	Sync_iyc-app-server_fd4d8b246454@iminyourcloud.onmicrosoft.com	AD		

Company Administrator (1)

Principal Name	Principal Type	userPrincipalName	Account type	Status	MFA
HJ M	User	dirkjan@iminyour.cloud	Cloud		

Directory Readers (1)

Another one is the Application Roles page, which shows all the privileges that Service Principals have in for example the Microsoft Graph and which users/groups are assigned to a role in applications.

ROADrecon

Home
Users
Groups
Devices
Directory roles
Applications
Service Principals
Application roles
OAuth2 Permissions
MFA

This page shows all application roles granted in the tenant. These roles are often used to manage access to applications or APIs. If the role is granted to a user/group, it is often used to grant access to an application or to give the user an (administrative) role in an application. If the role is assigned to a ServicePrincipal, the role is often used to manage API permissions, such as for the Microsoft Graph (the Azure Portal calls this an application role).

Filter

Principal Name	Principal Type	Application	Role	Description
HJ M	User	Office 365 Service Trust Portal	Default	Default Role
testapp	ServicePrincipal	Microsoft Graph	Directory.Read.All	Read directory data
testapp	ServicePrincipal	Microsoft Graph	Calls.AccessMedia.All	Access media streams in a call as an app
Office 365 Service Trust Portal	ServicePrincipal	Windows Azure Active Directory	Directory.ReadWrite.All	Read and write directory data
b	Group	testapp4	Default	Default Role

There are some things still in development in the GUI and I plan to add more advanced filtering features later, but the basics are there and overall it feels pretty snappy barring some loading times in large environments.

ROADrecon plugins - Parsing conditional access policies

I already mentioned plugins and that the goal is to make it easy for others to also write their own plugins or tools interacting with ROADrecon. An example plugin that I developed together with my colleague Adrien Raulot which has not made its way to the GUI yet is the conditional access policies plugin. As I discussed during my BlueHat talk, conditional access policies are not visible for regular users in the Azure Portal. The internal Azure AD API allows anyone to list them, but their raw format is full of GUIDs that have to be resolved manually. The “policies” plugin for ROADtools parses them into readable format and outputs them to a single static HTML page. Since Conditional Access policies are a pain to explore in Azure AD and require way too many clicks, this file is one of my favourite methods of exploring them. From a Red Team perspective, Conditional Access Policies are the most valuable resource to determine which applications do have stricter access controls such as requiring MFA or a managed device.

MFA for office

Applies to	**Including**: Users in groups: mfa for office
Applications	**Including**: All Office 365 applications
Controls	**Requirements**: Mfa

appserver require hybrid

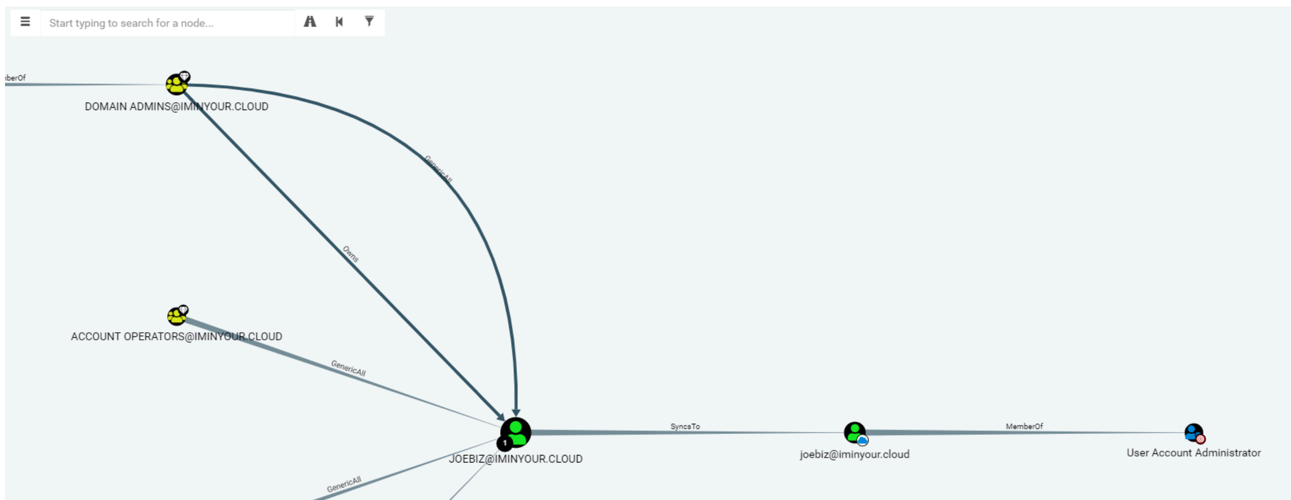
Applies to	**Including**: All users **Excluding**: Users: HJ M
Applications	**Including**: Applications: appserver
On platforms	**Including**: All platforms **Excluding**: Android, iOS
Using clients	**Including**: Browser, Native
Controls	**Requirements**: RequireCompliantDevice, RequireDomainJoinedDevice

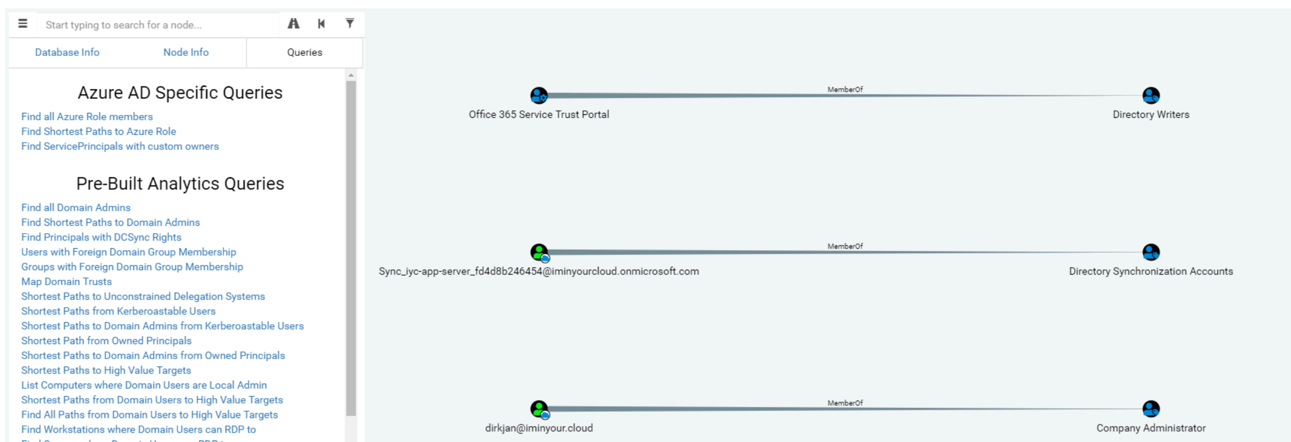
device stuff

Applies to	**Including**: Users in groups: ca_hybrid_device
Applications	**Including**: All Office 365 applications
Controls	**Requirements**: Mfa, RequireDomainJoinedDevice

BloodHound - with a twist of cloud

Another plugin that has a lot of potential is the BloodHound plugin. This plugin reads the objects of the Azure AD tenant in the database and writes them into a (local) neo4j database containing BloodHound data. When using a custom [fork](#) of the BloodHound interface, you can explore users, groups and roles visually, including links with on-prem Active Directory users if is a synchronized environment.





The BloodHound fork is still in an alpha version and will require some knowledge of Cypher to really get all the information out of it. I know that other people (such as Harmj0y and tifkin_) have also been working on an Azure AD supporting version of BloodHound, so my hope is that this can be developed further and maybe even merged back into the official BloodHound project.

Getting the tools

ROADtools is available on [GitHub](#) under an MIT open source license. Easiest way to install is using PyPi, automatic builds from Git are available in [Azure Pipelines](#).

The fork of BloodHound is available at https://github.com/dirkjanm/BloodHound-AzureAD.

I do also have a lot of stickers with the ROADtools logo (thanks for the design help [Sanne](#)!), which I'll be handing out as soon as we can safely do conferences again!

Defense

In my opinion enumeration is not an attack technique that blue teamers should focus their defense efforts on. The best way to prevent unauthorized users from accessing this information is by having strict conditional access policies which govern how and from where users are allowed to use their Azure AD credentials. That being said, there is a setting in the deprecated `MSOnline` PowerShell module which prevents enumeration using the Azure AD graph, which is documented here. I haven't personally looked into bypassing this or if other functionality in Azure breaks if you enable this.

Source: https://dirkjanm.io/introducing-roadtools-and-roadrecon-azure-ad-exploration-framework/