### Technical overview

This document serves as an overview of the programs we will use during our Entra ID Conditional Access Toolbox project, which are Powershell and WhatIf, along with the requirements we’ve set up for the toolbox and testing environment.

[**Technical overview 1**](#_222af3jd4653)

[**What is DCToolbox and what kind of functionalities does it offer? 2**](#_mvub0i8exe0a)

[**Are there alternatives for DCtoolbox? 3**](#_lwk6nxu1sdkm)

[**WhatIf 3**](#_ywf194kborcj)

[**What is Microsoft Graph API? 4**](#_knpk95qxsmr1)

[**Our Test-environment 4**](#_vbxgnk82ncz7)

[**Requirements for the test environment 5**](#_47u6sjoe0qe9)

[**Policy structure 6**](#_7wf82i5iq24s)

[**JSON-Based Policy Definitions and Test Cases 6**](#_dlbhzrjt3zum)

[**Coding requirements for PowerShell modules 7**](#_wgvfoemuc952)

[**Development plan of the module 10**](#_3p7b0i3syyin)

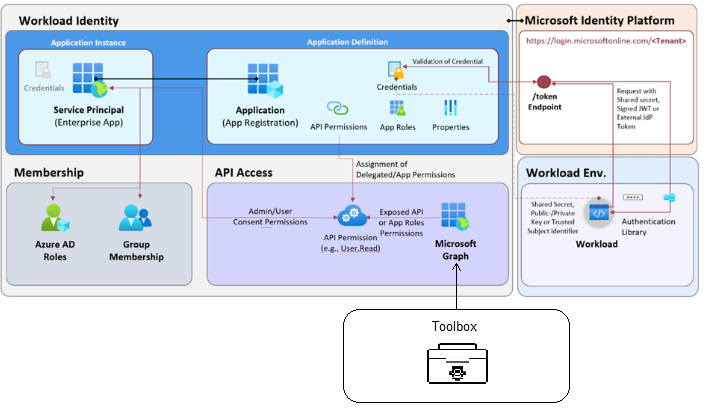
[**Requirements Test Cases 16**](#_r451b3aj73r3)

## 

### What is DCToolbox and what kind of functionalities does it offer?

DCtoolbox is a collection of PowerShell scripts and tools developed by Daniel Chronlund specifically designed to simplify the processes of Microsoft 365 environments. It provides a simple interface to perform common management tasks such as managing users, groups, Conditional Access Policies and other resources within Microsoft 365. We are going to be expanding on DCToolbox by adding our own methods and commands, to reach our goal.

A few examples of functionalities are that you can create templates of CA policies and integrate them right away on the Entra-ID network. You can assign CA policies to specific users, groups and/or departments to speed up the process. Other possibilities are to change names for groups of CA policies so they can be linked to specific users in an easy way, for example the marketing department gets a list of CA policies called MA. The nice thing about this toolbox is that you add all the changes live to your Entra-ID environment. So you have an interface where you can work easily and for that you don't have to work with the Entra-ID dashboard.



This model shows how Microsoft Entra ID works to manage access to applications and data. The toolbox is placed at API Access because this is the area where permission and access rights for APIs, such as Microsoft Graph, are managed.

Using this, administrators can set which applications and users have access to specific data or features, such as viewing user information. The toolbox serves as a tool to easily manage these permissions and roles so that the right people have the right access within the environment.

### Are there alternatives for DCtoolbox?

Another well-known toolbox that is similar to DCtoolbox is the Maester.Dev toolbox. This toolbox was created with the same vision in mind. Namely, to automate and speed up the process of managing Entra-ID. Maester.Dev does this in a different way, though, making it more accessible to most organizations. DCtoolbox, as mentioned earlier, is a collection of PowerShell scripts and IT administrators need to be familiar with PowerShell. Maester.Dev makes it easier for IT administrators and provides a user interface. You have the same capabilities, but this is now displayed in a simplified way.

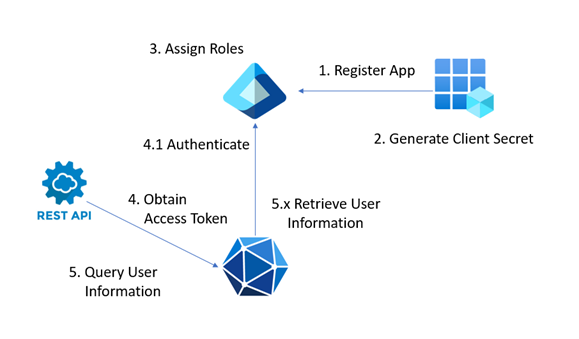
We chose to expand on the DCtoolbox rather than Maester.Dev because our project requires the flexibility and scriptability provided by a PowerShell-based approach. DCtoolbox allows us to directly manipulate and automate Entra ID operations through scripts, making it an ideal choice for implementing automated testing of Conditional Access (CA) policies using JSON test cases.

### WhatIf

The "What If" tool in Entra ID helps administrators test and predict the effects of Conditional Access policies on user access scenarios without enforcing them in real time. By simulating various conditions, such as specific users, devices, or applications, administrators can analyze how policies impact access, detect potential issues, and optimize security configurations. This tool aids in fine-tuning policies to ensure that security requirements are met while maintaining a smooth user experience. It serves as a risk-free environment for proactive policy management.

If you look on the right you see an example of the What If tool. You can put in your own conditions manually to check if certain Conditional Access policies are “hit”. This can take a lot of time if you want to do this for every user or scenario in an organization.

### What is Microsoft Graph API?

The toolboxes use the Microsoft Graph API, which provides access to a wide range of data within the Microsoft 365 environment, such as user information, SharePoint sites, OneDrive files and even Teams data. Microsoft Graph acts as a central API that connects the various services within the Microsoft 365 ecosystem. Toolboxes use this API to automate complex tasks and manage them more efficiently, which is especially useful for larger organizations that need to optimize and manage their Microsoft 365 environment.

In short, all information that is on Entra-ID is shared with the toolboxes. Examples for our project are users, groups and CA policies. This information can then be modified in the toolbox and is sent back by Microsoft Graph.

### Our Test-environment

To develop and test our toolbox for Conditional Access (CA) policies, we have set up a mock Entra ID environment. This environment connects via the Microsoft Graph API, enabling us to access existing information within Entra ID, including Policies, Users, and Groups. The toolbox itself will be a PowerShell module; a collection of tools for Microsoft Graph functions, Entra ID management, and CA policy testing.

The **main purpose** of this mock environment is to simulate and test CA policies rather than to mimic the exact structure of a real organization. The CA policies are key to our testing, as we’re developing a "what-if" type of method within the toolbox. This method will allow us to run test cases in various scenarios, assessing how CA policies would apply without needing to change live settings.

We chose to create 64 users and organize them into 8 groups to create balance between complexity and manageability. The structure of 64 users provides scalability for running test cases with varying group sizes and policy configurations. By dividing users into 8 groups, we can test how policies apply to different organizational units, roles, or access levels.

Each user has been created with a first name, last name, and an e-mail address.In our case, the user and group setup serves mainly as a structure for us to simulate our own policies and find out how Entra-ID truly works, rather than replicating an actual organizational model. The focus for the mock environment lies on the Conditional Access (CA) policies themselves and the functionality of the test-method we’re developing within the toolbox.

#### Requirements for the test environment

In order to test CA policies and develop our “what-if” testing method, our mock environment must meet the following requirements:

1. **Entra ID sandbox setup**
   1. The environment must work as a non-production sandbox, allowing safe experimentation with CA policies. This setup makes sure we can test without any risk to live data or actual company policies.
   2. A designated breakglass account and group must be configured to make sure we keep access to the environment in case any CA policies lock out users or administrators. This is key to prevent lockouts during testing.
2. **Policy requirements**
   1. The test environment must have a variety of CA policies and policy settings (e.g., access requirements based on user risk, location, device compliance, multi-factor authentication, etc.) to make sure we are able to test a variety of different scenarios once exported.
3. **Connectivity to Microsoft Graph API**
   1. The environment must support automated data interactions through the Microsoft Graph API. This capability will allow our toolbox to interact with Entra ID programmatically, which is important for testing CA policies.
4. **User and group setup**
   1. A simple user and group structure is necessary to simulate how policies work. For this project, a setup of 64 users across 8 groups gives us a varied but manageable structure for testing CA policies.

#### Policy structure

As for policies, we are using all 18 template-policies that Microsoft provides us with. Microsoft recommends these policies as the base for all organizations. Microsoft supplies us with policy templates for 5 different categories:

* **Secure Foundation**: Policies within this category cover basics like blocking legacy authentication or requiring MFA for all users. These are essential for reducing common vulnerabilities and improving baseline security.
* **Zero Trust**: These policies enforce stricter controls, like requiring MFA based on sign-in risk or making sure access is only granted from compliant devices. They align with the "verify explicitly" principle, making access decisions based on conditions.
* **Remote Work**: Policies in this category, like protecting access on unmanaged devices or requiring frequent sign-ins, focus on securing remote and hybrid work scenarios.
* **Protect Administrator**: Admin accounts are high-value targets, so these policies enforce stronger security measures like phishing-resistant MFA or limiting admin access to specific conditions.
* **Emerging Threats**: This category focuses on addressing newer risks, like insider threats or requiring devices to meet compliance standards with MDM.

The reason we included all 18 policies is that they cover a wide range of situations, letting us configure CA policies in as many areas as possible. Each category represents a different type of security challenge, and testing all of them ensures we can try out our "what-if" tool against different scenarios. This way, we’re not just testing random policies—we’re making sure we hit every major area that organizations would care about.

### JSON-Based Policy Definitions and Test Cases

Each policy in Entra ID can be exported as a JSON (JavaScript Object Notation) file. JSON is a standard format for storing and transporting data, known for being "self-describing" and easily readable. These JSON definitions can be edited and then imported on the main Conditional Access policies page using the Upload policy file option. This could be useful for bulk-editing, but we’re going to make use of the JSON definitions to create specific test cases designed to simulate different access scenarios. These test cases will be stored in a JSON file, allowing us to define hypothetical users, devices, and locations to observe how CA policies respond to varying conditions.

The "what-if" test method within the toolbox reads these test case JSON files, simulating CA policy outcomes without affecting live Entra ID settings. This method allows controlled, repeatable testing and troubleshooting of policies, making it much more efficient than the manual testing currently required.

The following parameters, excluding the device filters we’ve made, were used:

UserPrincipalName can be either "All", an email, or "GuestsOrExternalUsers". Country is a Country code, I want mostly NL and other EU countries. Platform is the OS, so windows, macOS etc. ApplicationDisplayName can be The display name of the application targeted by Conditional Access policies (same display name as in Entra ID Portal when creating Conditional Access policies). Example 1: 'Office 365'. Example 2: 'Microsoft Admin Portals'. Default: 'All'. Client app is The client app type used during sign-in. Possible values: 'browser', 'mobileAppsAndDesktopClients', 'exchangeActiveSync', 'easSupported', 'other'. Default: 'browser'. TrustedIPaddress can be true/false, signinrisk and userrisk can be High, medium or low.

Here is an example:

{

"UserPrincipalName": "emilydavis@hfsaoiufhaosiudhf.onmicrosoft.com",

"Country": "SE",

"Platform": "macOS",

"ApplicationDisplayName": "Microsoft Admin Portals",

"ClientApp": "browser",

"TrustedIPAddress": false,

"SignInRiskLevel": "low",

"UserRiskLevel": "low"

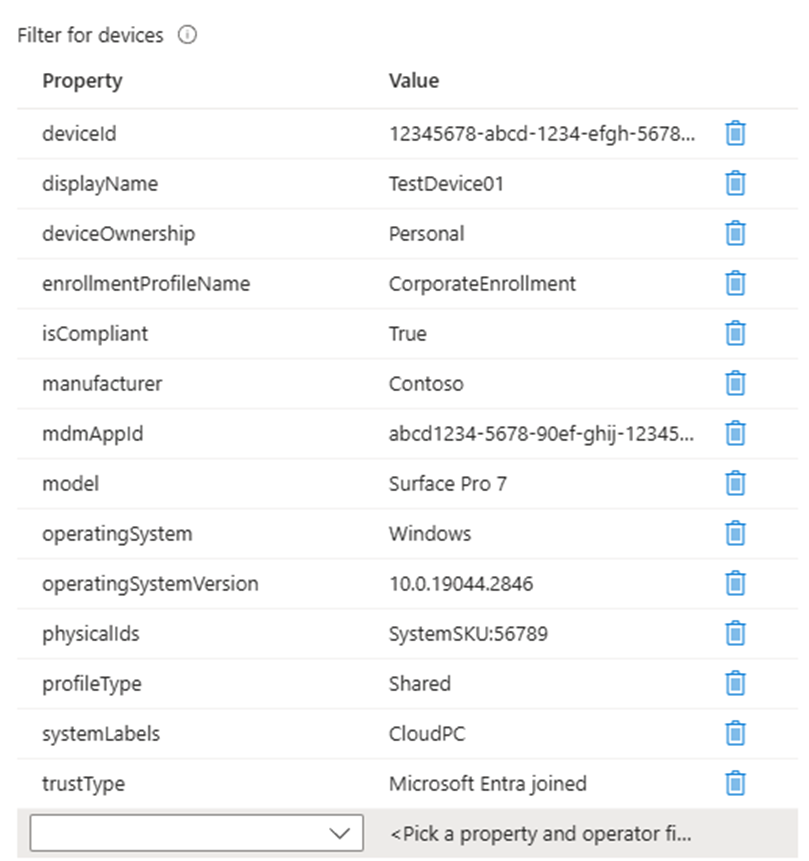
}

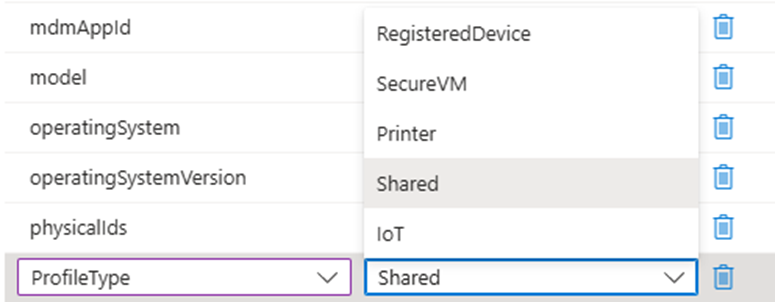
### Coding requirements for PowerShell modules

**New code must follow the format of existing code.**  
The DC-Toolbox contains two files. One file is used to create all the functionalities of the methods, and the other one organizes these methods. Any code added to the DC-Toolbox must follow the style of the existing code to maintain consistency and structure. This is beneficial for both developers and clients.

**New code must not conflict with existing code.**  
To ensure new code does not interfere with already existing code, new objects and methods are created. These methods are used to test new functionalities. Although this might result in some code duplication, it helps keep the current code as intact as possible, which helps to minimize the risk of mistakes in the current code.

**Filter for Devices must be added.**The Filter for Devices is a feature that allows specific criteria to be applied to devices. This filter consists of 14 variables, each of which can specify the type of filter being used. These variables can take values such as booleans (true/false), strings or predefined options from a list. The Filter for Devices is already supported in the What-If tool and can be tested there. The figures below demonstrate how these filters look in the What-If tool.  
  
Currently, Filter for Devices is not implemented in the DC-Toolbox, making testing with it in this environment impossible. To create this functionality the filters need to be implemented in the DC-Toolbox and added to the parameters of the method: ‘Invoke-DCConditionalAccessSimulationWithDevices @Parameters | Format-List’. This will make it possible to test conditional access policies using the Filter for Devices.

  
*Example of all the properties of Filter for Devices with filled in values.*



*Example of a value with a list.*

**Conditional access policies that match must be correct.**  
The parameters in the method ‘Invoke-DCConditionalAccessSimulationWithDevices @Parameters | Format-List’ provides information to determine whether the conditional access policies match them. The methods responsible for verifying these matches must function correctly. To ensure this unit tests should be created or they can be manually tested using the What-If tool.

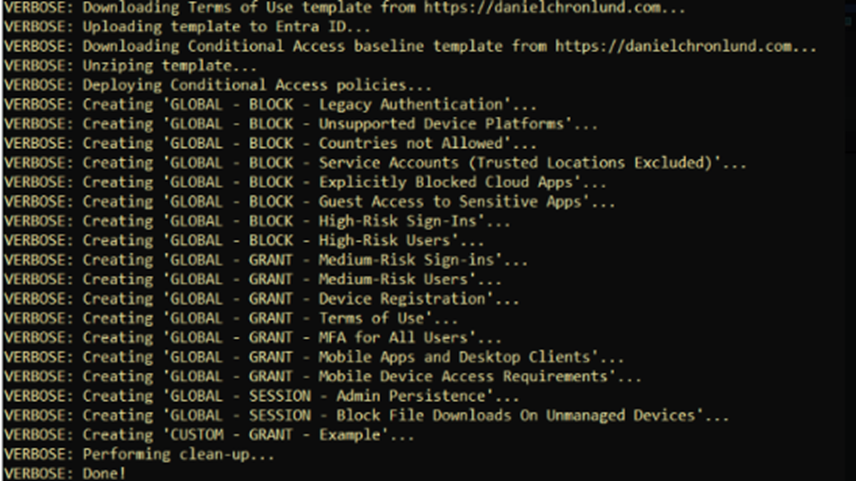
**Output interface has to be correct.**When conditional access policies match the given parameters, the results must be displayed correctly in the interface. This can be verified through unit tests and by printing it in the console in a clear way.

**Being able to test multiple sets of parameters with a method.**Currently the method ‘Invoke-DCConditionalAccessSimulation’ can only be used for one specific user with a single set of parameters. This means that you must test conditional access policies for each user individually by adjusting the parameters manually and calling the method again. This process becomes very time-consuming, especially when there is a large number of users.  
  
To address this issue, a new method is being developed to enable testing for multiple users simultaneously. Test cases for different users will be stored in a single JSON file. The new method will read this file and automatically test all test cases in sequence. This removes the need to call the method manually multiple times, significantly reducing time and making the testing process more efficient.   
 **Clear and accurate output for multiple sets of test cases.**When testing multiple users, the method will return a set of conditional access policies for each user. It is important that these policies are presented correctly for each user and in a way that is easy to interpret. This can be achieved by making unit tests. Additionally it should be discussed with the client what they consider a clear and concise representation of the output.

### Development plan of the module

**Current module**The DC-Toolbox contains 32 different methods that can be applied to the Entra-ID environment. First a connection must be established with Entra-ID, which is achieved using an API called Microsoft Graph. Once the connection is established various methods can be executed with different functionalities. In this project the focus is placed on the methods necessary for conditional access policies and their testing.

**Deploy-DCConditionalAccessBaselinePoC**This method is used to create conditional access policies for the test environment. It generates a template of 18 conditional access policies in report-only mode, which allows them to be tested without having to enforce them in practice.

  
*Output of the console after invoking the method ‘Deploy-DCConditionalAccessBaselinePoC’.*

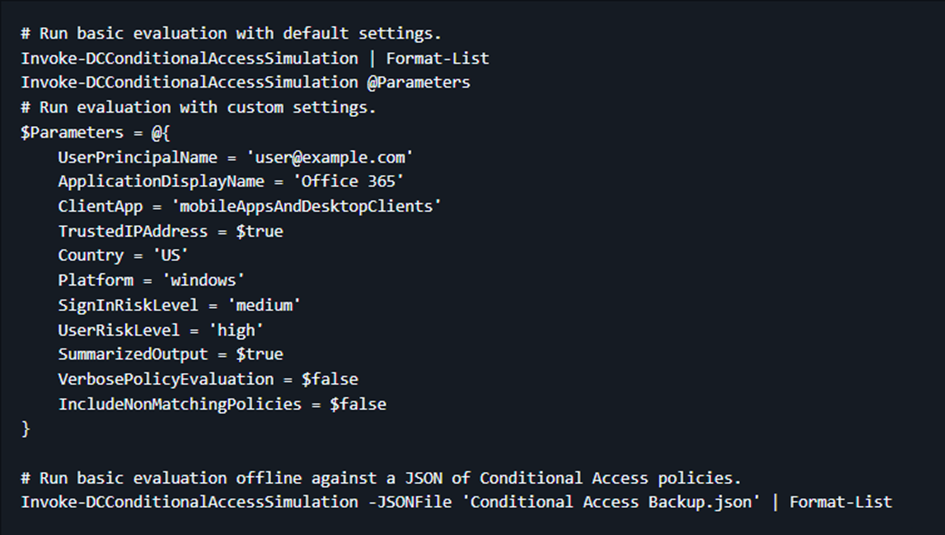
**Set-DCConditionalAccessPoliciesReportOnlyMode**This method allows you to change the mode of policies. The template policies created earlier were activated using the method ‘Set-DCConditionalAccessPoliciesReportOnlyMode -PrefixFilter 'GLOBAL - ' –SetToEnabled’. With this all policies with "GLOBAL" in their names have been activated for testing purposes.

**Get-DCConditionalAccessPolicies**This method is used to retrieve and print all policies in the console. It is a convenient alternative to navigating through Entra-ID to check which policies exist.

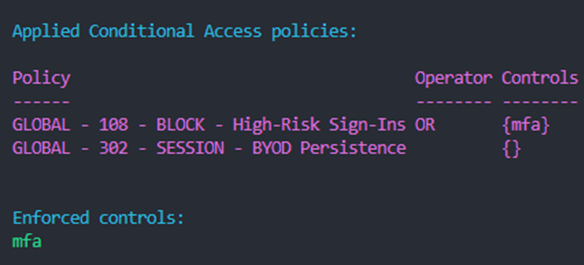
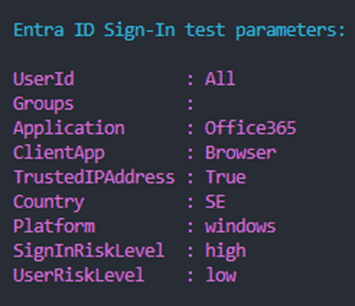
**Rename-DCConditionalAccessPolicies**This method allows you to rename policies. This can be useful for improving clarity of policy names. It also offers the option to rename all policies with the same prefix simultaneously. Which is not possible inside of Entra-ID.

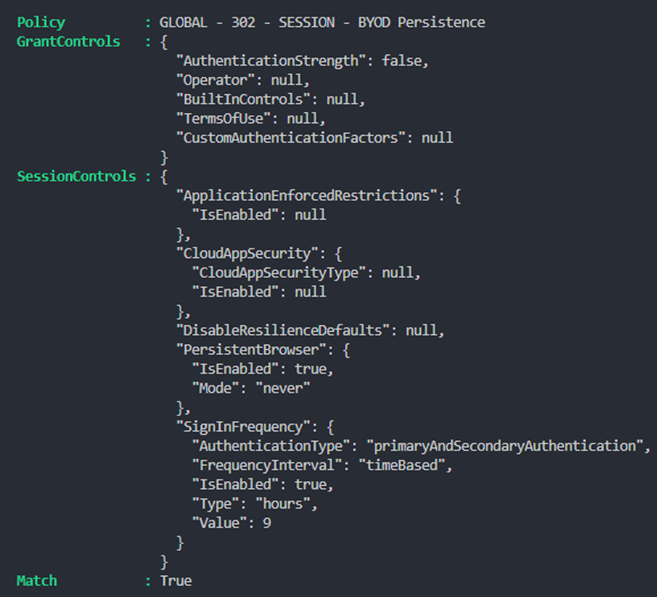
**Remove-DCConditionalAccessPolicies**This method can be used to delete all policies or specific ones based on a prefix. This saves time by avoiding the need to delete policies individually in Entra-ID.

**Invoke-DCConditionalAccessSimulation**This method allows you to simulate which conditional access policies match for a single user. It has the same functionality as the What-If tool in Entra-ID, but it does not include the Filters for Devices.

  
*Documentation of the method ‘DCConditionalAccessSimulation’.*

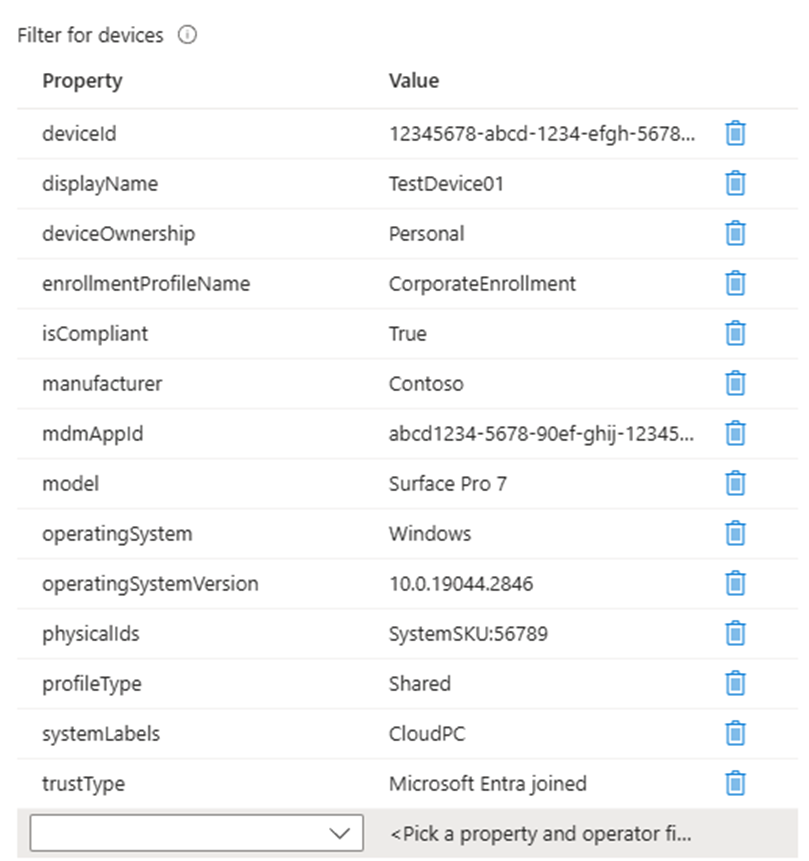
First you need to provide the parameters, and then you can call the command ‘Invoke-DCConditionalAccessSimulation @Parameters’ to see which access policies match. If you use the command ‘DCConditionalAccessSimulation @Parameters | Format-List’ you can get additional information such as GrantControls and SessionControls, which provide information about the output. If you do not use @Parameters the default parameters shown in the figure above will be used.

  
*Partial output in the console after invoking the method ‘DCConditionalAccessSimulation’.*

  
*Partial output in the console after invoking the method ‘Invoke-DCConditionalAccessSimulationWithDevices @Parameters | Format-List’.*

In the PowerShell console you can see which conditional access policies match for the specified users and parameters. Under the controls section, you can view the measures applied by each policy. For instance in this case, MFA is enforced under the second policy. While no measures are applied under the first policy. However under the sessionControls section, in the SignInFrequency category, it is shown that the user will automatically be signed out 9 hours after logging in.

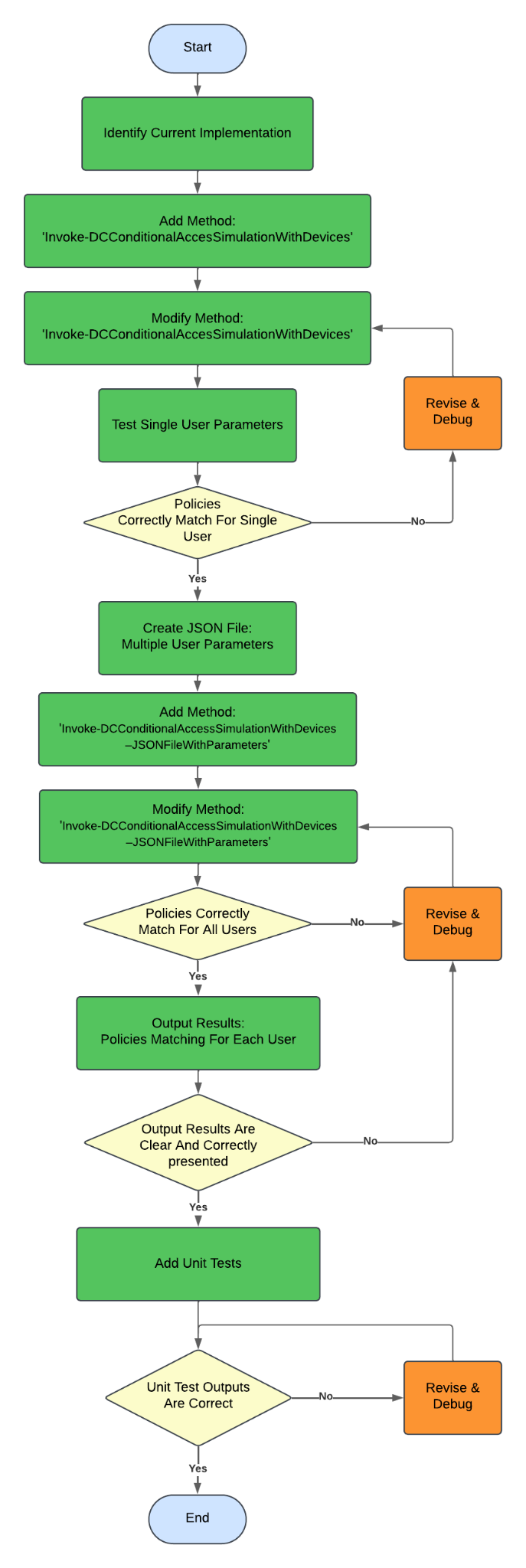
**Implementation of the current module and the new implementation**

To meet the requirement for the Filter for Devices the following method was created: ‘Invoke-DCConditionalAccessSimulationWithDevices’. This is a modified version of the method ‘Invoke-DCConditionalAccessSimulation’ where Filter for Devices is added to the parameters. These are the 14 variables shown here:   
These parameters must then be tested against the conditional access policies to determine which ones match.

To implement the Filter for Devices and the functionality for testing conditional access policies, the way how the current existing code has been written is reviewed. For this objects from the parameters and checks for the policies are used as examples. Since no documentation has been written for the current code, extensive testing is required to understand how the code is structured and functions. The code will also be tested using unit tests.

To meet the requirement ‘being able to test multiple sets of parameters with a method’, a JSON file will be created that contains multiple users that each have their own parameters. Next the method ‘Invoke-DCConditionalAccessSimulationWithDevices’ will be modified to test a JSON file with multiple users with their respective parameters. It is meant that this method will loop through all the users and check the conditional access policies once for each user by only being called once. Afterward it will print in a clear format which conditional access policies match for each individual. To use the JSON file the command ‘Invoke-DCConditionalAccessSimulationWithDevices –JSONFileWithParameters’ must be called.

This creates two types of methods. Method: 'Invoke-DCConditionalAccessSimulationWithDevices' where the Filter for Devices is added and it tests which conditional access policies match for one user. Method: 'Invoke-DCConditionalAccessSimulationWithDevices –JSONFileWithParameters', where the filter for devices is added and it tests which conditional access policies match for multiple users.

  
*Flowchart of the development process of the module.*

More readable version can be found with the following link: <https://lucid.app/lucidchart/f04571aa-a05f-4914-b84d-5e951706ffc3/edit?viewport_loc=-3798%2C-41%2C6451%2C2681%2C0_0&invitationId=inv_ea2a63a7-373e-4823-9955-c77cee10767a>

### Requirements Test Cases

For testing our toolbox, we will be creating several test cases based on our test environment. To ensure the quality of the test cases can be justified, we establish requirements that they must meet before we proceed with them.

**1.** **Coverage of Test Environment**Test cases must include scenarios for:

* All users: Ensure policies behave correctly for a variety of roles and identities.
* All groups: Validate policies to end up covering/targeting all groups.
* Role differentiation: Verify that policies correctly distinguish between regular users and privileged roles.
* Differences in Location (Country, IP, Office)
* Differences in devices (OS, type of device (mobile/desktop) etc.)

**2.** **Policy Template Validation**Test cases must evaluate the behavior of all 18 Microsoft-provided template policies, and all other policies an organization may have implemented.

**3. Comprehensive Scenarios**Test cases must include:

* Positive scenarios: Verify expected access under compliant conditions (compliant device, trusted location, valid MFA).
* Negative scenarios: Test for blocked access under non-compliant conditions (legacy authentication, untrusted location).
* Edge cases: Validate uncommon but possible situations.

**4. Conditional Logic and Overlap**Test cases must verify how policies interact, especially when multiple policies apply to a single user or group. For example: Verify that a 'Protect Administrator' policy takes precedence over a 'Remote Work' policy when an admin logs in remotely.

**5.** **Sign-In Risk Evaluation**Test cases must validate the handling of risk levels:

* Low risk: Verify normal access behaviour.
* Medium or high risk: Confirm MFA is required or access is blocked based on policy settings.

**6.** **Usability for Remote Work**Test cases must validate policies specifically aimed at remote work.

**7. Administrative Role Protection**Test cases must validate stronger security for admin accounts:

* MFA.
* Restricted login conditions (trusted locations and/or devices).

**8. Automation Compatibility**Test cases must be structured for automation, so it matches the purpose of our toolbox. This means that test cases need to be able to be triggered from a command line terminal.

**9.** **Preconditions and Dependencies**Test cases must document:

* Any pre-configuration required (policy creation, user assignments).
* Any dependencies (configuration of devices).