Appgate SDP

Automation and orchestration – scripting of access controls

Type: Technical guide

Date: November 2020

Applies to: Appgate SDP v5.3 and newer

TABLE OF CONTENTS

INTRODUCTION

Like most other edge devices Appgate SDP can be configured with traditional static access rules. These can still be (quite advanced) conditional rules, but at the end of the day they are static; rules such as *allow access on Thursdays*. In an increasingly connected world where dev-ops is becoming the norm a more dynamic model is needed for access rules; rules such as *allow access if the data centre temperature is over 20 degrees.*

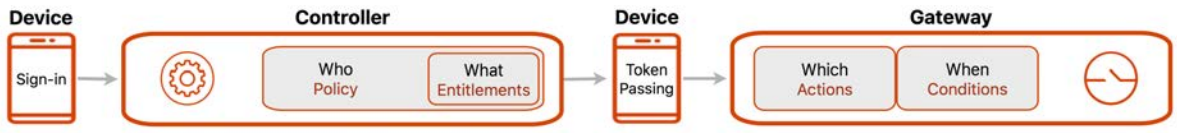
This paper takes you through the different points in the system where it is possible to use some form of scripting to dynamically configure the Appgate SDP system.

It does not cover the ability of the Appgate SDP to be modified using REST API calls (from scripts run by others). These makes it very easy for external systems to update the static (or dynamic) configuration settings within the Appgate SDP system. This further enhances the dynamic nature of the overall solution. For more information, please refer to the detailed API documentation.

By combining the scripting capabilities covered in this paper with the REST API support means it is now quite possible to utilize Appgate SDP as a foundation from which you can build a fully automated access control platform even encompassing the concept of ‘security as code’.

ACCESS RIGHTS MODEL USED IN APPGATE SDP

Appgate SDP uses a token-based architecture to pass information from the Controller, via the user's device to the Gateway. Tokens contain all the information needed for authentication, authorization and real-time access control.



- The user signs-in to the Client on their device.
- System, user and device claims are harvested.
- The Controller (the certificate authority) uses the claims to decide what Entitlements to assign to the user and creates signed Entitlement tokens for each Site.
- The Entitlement tokens and claims token are sent from the Controller to the user's device and the Client forwards them to the Gateway(s).
- The Gateway uses the Entitlement token to configure the firewall rules (Actions) and provide near real time access controls (Conditions) on a per-user basis.

Controller

When the first Controller in a new Collective is started, it establishes itself as the as the certificate authority for signing tokens and certificates and for establishing mutual TLS connections. It contains the configuration database but for the most part it is stateless.

The Controller is a REST based appliance which simply responds to any incoming requests.

For administrators it provides centralized administration over security Policies, assignment criteria, user Entitlements/Conditions, administrator privileges, network configuration, logging, monitoring, etc. The assignment criteria set for each Policy define who will be granted specific (access) rights. The Controller can also be used to manually revoke tokens at any time.

Whenever Clients require a token creating or updating, they make REST calls over mTLS to the Controllers – but otherwise remain disconnected. The Controller replies to the Client with Entitlement tokens - one per Site.

Gateways make REST calls to Controllers to report on health, fetch token revocation lists, check for any network changes, etc.

Client

The Client uses DNS round-robin to find an active Controller each time it requires tokens. After (re-)authentication, the required tokens are issued with a pre-defined lifetime.

The Entitlement token includes a list of available Gateways with load balance weightings for each Site. The Client will then use this information to establish a connection to one Gateway on each Site where the rights defined in the Entitlement are granted. Once established, the Client will be able to communicate with the protected network resource via the Gateway.

Gateway

The Gateway is the enforcement point, responsible for controlling user access to protected network resources. After initial seeding, it registers with the Controller via a REST call over mTLS and is then added to the list of available Gateways on the Site.

With every new Client connection, the Gateway will start a firewall service dedicated to that session. The Gateway uses the Entitlement tokens from each Client to manage firewall rules and uses the Claims token combined with other Conditions to provide real-time access control.

User interactions allow the Gateway to elevate user privilege levels on-demand.

Name resolvers automatically resolve any firewall rules based on real-time information from the Cloud (Cloud API based DNS) or hypervisor environments.

Claims Token

A claims token is provided by the Controller to the Client on successful user authentication. The claims token is signed by the Controller and contains validated trusted claims related to the identity (and context of the Client and device). Information in the claims token can be encrypted thus protecting any sensitive claims that may have been returned in scripts.

The Client will keep the claims token in memory for as long as it needs but not save it to disk. This allows the Client to connect to the appliances in the Collective without any further authentication.

An example of the information included in a claims token might be username, AD group membership, etc.

Entitlement Token

An Entitlement token contains a list of all the user's Entitlements for a specific Site as well a list of the Gateways associated with that Site and the defined weighting of each Gateway.

The Entitlement token is created by the Controller following a request from the Client. Once it is connected, the Client sends the relevant Entitlement token to the appropriate Gateway. An example of the information included in an Entitlement token would be Site, list of Gateways, access rights (Actions and Conditions).

Claims based access rights model

Before exploring how to add orchestration and automation to the Appgate SDP system it is worth remembering that this is just an extension of the traditional static access rules capabilities of the system. The system already uses claims in Boolean expressions to assign Policies and allow Actions. By adding orchestration and automation we are enhancing the number/type of claims available, extending how claims can be used within the system and allowing more complex expressions to evaluate claims. Whether using the normal capabilities of Appgate SDP or embarking on an orchestration and automation project, the same care needs to be taken when designing access controls.

The claims used throughout fall into two categories:

- Trusted - typically collected from authoritative sources such as LDAP
- Un-trusted - typically collected from exposed environments such as the user’s device

The latter should therefore not be relied on for making critical security decisions; but could be used for finessing access decisions such as removing an app on mobile devices which may not work on small screens.

From a trust (and therefore security) point of view it can sometimes be better to think at the overall system level and use a combination of things to achieve the desired outcome. If you needed to rely on claims about connecting devices, then rather than just relying on a claim from a device script it might be better to combine the use of and external database with a device claim script and a user claims script, all working together. A device script could be used to collect a secure random ID which was assigned to the connecting device as part of an enterprise build process. This ID itself can't be trusted because it has come from the device. But this ID could now be passed in a user claim script to an authoritative external device database. In the first instance the ID has to match a record, confirming it as a legitimate device; and then the script could return some trusted claims relating to that device ID.

Unlike some edge devices, Appgate SDP can be configured with quite advanced claims based access rules but at the end of the day they are static - rules such as allow access on Thursdays. In an increasingly connected world where dev-ops is becoming the norm a more dynamic model is needed for access rules - rules such as allow access to a specific data center when the inside temperature is over 20 degrees centigrade.

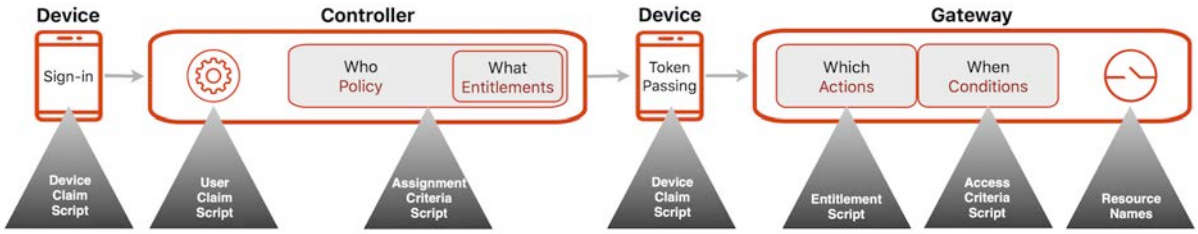
ADDING ORCHESTRATION AND AUTOMATION

The Appgate SDP system has evolved to allow almost all parts of the access control process flow to be made scriptable. This allows the system to interoperate with countless external systems and network environments. Effectively the system can learn who might want access and what is potentially available for them. It can then decide what Actions are allowed at the current time and find the available live hosts.

Dynamic configuration offers many advantages over static configurations:

- The system configuration is always up to date (no stale firewall rules).
- The actual number of rules required is likely to be drastically reduced (making for streamlined rule audits).
- A well configured dynamic access system requires very little day to day maintenance (reducing the opportunity to make mistakes).
- The use of metadata to control the configuration allows specific user access decisions to be informed by authoritative parties (outside of the networking and security teams).
- The access rules within the system can respond in near real time (such as when a threat is detected).
- The system can automatically adapt to the environments in which it is deployed (such as the use of autoscaling).

A somewhat complex example which would have to use several of these scripts might be:
allow support guys weekday only access to Azure systems exhibiting high CPU load tagged with 'Dev'. They must have a correctly patched device and there must be a 'Dev' in the office (to check any fixes).

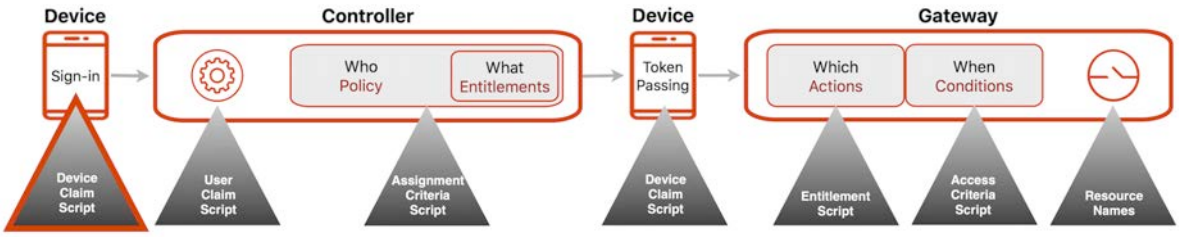


The triangles show all the different points in a typical sign-in process flow where it is possible to apply some sort of scripting. At most of these points the scripts can perform complex logic, string manipulations or calculations but an important part of the capability offered is to allow limited calls to be made to external systems.

Remember it is also possible for external systems to make calls to Appgate SDP to modify the system configuration. The REST API functions allow you to create Entitlements, Policies, and Conditions (among many other settings). This means it is even possible to have external systems create some of the scripts which are then used to dynamically update the configuration settings within the system!

The following sections of this document takes you through each of the points in the system shown in the diagram above. It explains briefly what each script does and its usage. The system requires you to set things up in a number of different places to allow a script to be used – so all these steps are highlighted. Finally, a code example is provided for each.

Device Claim Script (Controller)



After the user signs in, specified native executable device claim scripts are pushed to the user's device. The Client runs these executables which should be specifically designed to harvest and return additional claims (using stdout). The appropriate executables are automatically pushed to the specified platform. Each OS must be able to support native execution of the file type pushed.

Why use?

Over 20 built-in claims and on-demand device claim commands are available. When the specific information required is not available using the built-in commands, you may write and upload native executable scripts. The supplemental device information provided by the script will be saved as a claim by the Controller and can be used as basis for assignment decisions in the (Assignment) Criteria Scripts within Policies.

Usage example:

Return the device's IT asset number from some corporate device identifier.

Configure:

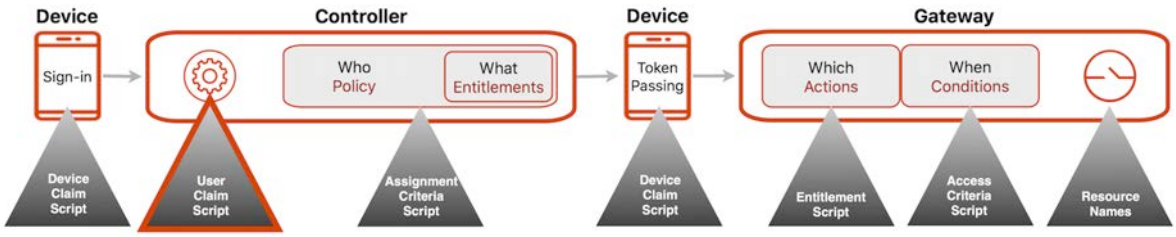
- Device Claim Scripts are uploaded in **Scripts>Device Claim Scripts**
- They are enabled by Identity Provider. To run a script, go to **System>Identity Providers>Configure On-demand Device Claims** and select **Run Device Script** from the **Command** drop down. Here you specify where (platform) the script should be run. The output from the script will be mapped to the device claim specified.
- The on-demand Device Claims are then available at the bottom of the **Assignment Criteria** drop downs in **Operations>Policies** and **Scripts>Criteria Scripts**.

Code example:

Checking MDM firewall is enabled:

```
@requires_authorization
def somefunc(param1='', param2=0):
    r'''A docstring'''
    if param1 > param2: # interesting
        print 'Gre\'ater'
    return (param2 - param1 + 1 + 0b10l) or None

class SomeClass:
    pass

>>> message = '''interpreter
... prompt'''
```

User Claim Script



The user Claim script is designed to collect additional user centric claims. User claim scripts execute AFTER user sign-in but BEFORE Policy assignment. The resultant claims generated by the user claim script may be re-used elsewhere in the system; within the Controller the new user claims can be used as assignment criteria within Policies. These new claims are also added to the claims token so within the Gateway, they can be used as access criteria in Conditions and even to define protected hosts in Entitlement scripts.

The JavaScript expression runs in a sandboxed JavaScript engine which supports external http Get/Post/Put calls. There is no time-out, as such so user claim scripts should be built as efficiently as possible as the user/device sign-in process will be waiting for this script to finish. If the script takes too long, after 10 seconds the Client will fail-over to the next Controller and the process will be repeated - possibly indefinitely!

Why use?

This type of script would typically be used to query external systems in order to collect additional information about the user and/or their device. It should be used for making external calls in lieu of doing this in the (assignment) criteria script. The user claim script will be run just once whereas the assignment criteria script will be run once for every Policy.

This type of script can also help to reduce the load on your Gateways considerably. Claim values (from the script) may be used straight from the claims token; rather than the Gateway having to query the external system (again) for every new TCP stream initiated by every user.

Usage example:

Returns a key value pair after checking user and/or device claims against an external database. The device's asset number could be used to check if the patching is up to date by returning the last patch date.

Configure:

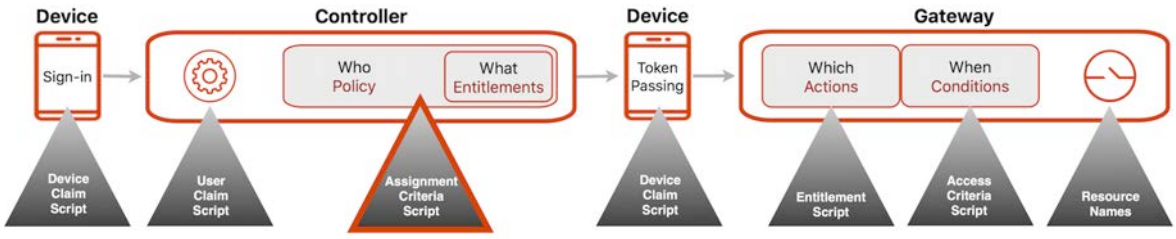
- User Claim Scripts are added in **Scripts>User Claim Scripts**
- They are enabled by Identity Provider. To run a script, go to **System>Identity Providers>User Claim Scripts.** The output from the script must include a specific user claim and its value.
- The results from the User Claim Scripts can be included by selecting **User Claim Script** from the top section of the criteria drop downs in **Operations>Policies**, **Operations>Conditions** and **Scripts>Criteria Scripts**.
- A JavaScript expression is required to evaluate the returned value as the system has no way of knowing the type of value that has been returned. The new claim will appear in the system as *claims.user.agScripted.claim_name:claim_value*

Code example:

Returns a risk score for the user/device:

```
// Create HTTP body to send
var body = {
  username: claims.user.username,
  deviceId: claims.device.id, // generated by a device claim script
};var response = httpPost('https://posture.mycompany.com/appgate-posture', body);if
(!response) {
  console.log('Connection failed');
  return {};
}if (response.statusCode !== 200) {
  console.log('Posture check returned error: ' + response.data);
  return {};
}return { riskScore: response.data.score };
```

(Assignment) Criteria Script



The Appgate SDP system evaluates claims using assignment criteria expressions to make Policy assignment decisions. Normally Policy assignment criteria expressions can be configured within the Policy itself using the assignment tool options and claims pick-list. In such cases there is no requirement to write a separate script as the simplified UI will generate a JavaScript expression for you.

When a custom script is needed, it may use existing user, device and system claims. This script also executes within the native sandboxed JavaScript engine which supports external http Get/Post/Put calls. This script should be as efficient as possible because after sign-in every Policy will be checked (and will run the script) to determine if the related user rights apply.

Why use?

There are several reasons why you might want to use a criteria script (as opposed to just a Boolean expression).

- The complexity of the Boolean expression required goes beyond what the builder can support.
- You switched from *custom logic is met* to *script returns true* and made an edit.
- An external call is required to harvest claims from some other third-party system.
- The same criteria expression is to be re-used in several Policies.

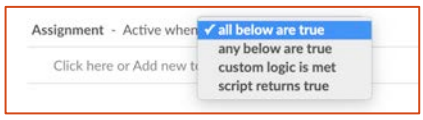
Usage example:

Returns true if the user is in LDAP Group X AND the device is fully patched AND device is on the office network.

Configure:

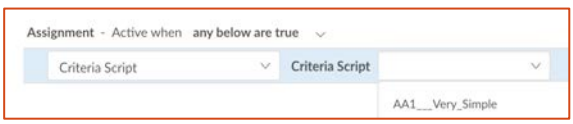
Either

- Enter the script directly in **Operations>Policy>Assignment** using **script returns true**.



Or

- In **Scripts>Criteria Scripts** under **Assignment** select **script returns true** and create and save a re-usable script.
- In **Operations>Policy** under **Assignment** by choosing **Criteria Script** from the drop down select the script from the list.

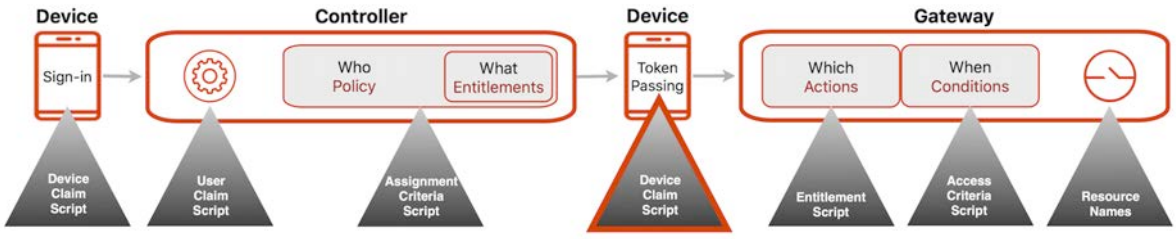


Code example:

Typical multi-claim criteria script:

```
var result = false;
if/*ip*/(claims.system.isClientInNetwork('10.10.1.0/24'))/*end ip*/ { result = true; } else { return
false; }
if/*claims.user.groups*/(claims.user.groups && claims.user.groups.indexOf('ITAdmin') >=
0)/*end claims.user.groups*/ { result = true; } else { return false; }
if/*claims.device.os.family*/(claims.device.os && claims.device.os.family === "Windows")/*end
claims.device.os.family*/ { result = true; } else { return false; }
if/*claims.device.Symantec_chk*/(claims.device.Symantec_chk === true)/*end
claims.device.Symantec_chk*/ { result = true; } else { return false; }
if/*claims.device.isFirewallEnabled*/(claims.device.isFirewallEnabled === true)/*end
claims.device.isFirewallEnabled*/ { result = true; } else { return false; }
return result;
```

Device Claim Script (Gateway)



Every 5 mins, the Client runs native executable scripts that have previously been pushed to the device. Each OS must be able to support native execution of the file type pushed.

These should be specifically designed to harvest and return additional claims (using stdout). They will typically check things which may have changed since the user signed in. Metrics which change all the time (such as CPU usage) SHOULD BE AVOIDED because updated device claims are sent to Gateway and the access criteria is re-evaluated every time a change is detected, this will result in significant system and network overhead.

Why use?

Over 20 built-in claims and on-demand device claim commands are available. When the specific information required is not available (using the built-in commands), you may write and upload native executable scripts. The supplemental device information provided by the script will be treated as a claim by the Gateways and can be used in a Condition's access criteria scripts to allow (or block) a specific Entitlement.

Usage example:

Return true/false value based on whether the AV process is installed, updated and running.

Configure:

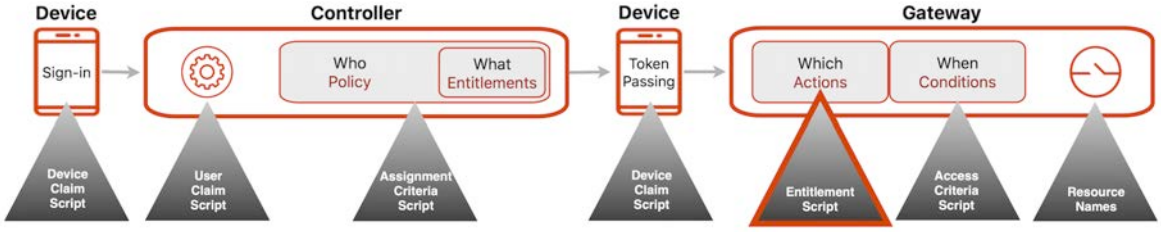
- Device Claim Scripts are uploaded in **Scripts>Device Claim Scripts**
- They are enabled by Identity Provider. To run a script, go to **System>Identity Providers>Configure On-demand Device Claims** and select **Run Device Script** from the **Command** drop down. Here you specify where (platform) the script should be run. The output from the script will be mapped to the device claim specified.
- The On-demand Device Claims are then available at the bottom of the **Access Criteria** drop downs in **Operations>Conditions**.

Code example:

Checks the wifi signal strength on a macOS.

```
#!/bin/bash

x=$(/System/Library/PrivateFrameworks/Apple80211.framework/Versions/Current/Resources/
airport -I|grep agrCtlRSSI|awk '{print $2}')
if(($x > -60))
then
        result="strong"
elif(($x > -70))
then
        result="acceptable"
elif(($x <= -70))
then
        result="weak"
fi

echo $result
```

Entitlement Scripts



When an Entitlement is defined, it comprises app shortcuts (shown in the Client app launcher) and the related Actions (defining the firewall rules). It is possible to use Entitlement scripts to define these elements. These scripts can utilize any existing user, device and system claims including those provided by the Controller in the claims token. Again, it executes in a sandboxed JavaScript engine with http Get/Post/Put calls allowed.

The 3 types are:

- App Shortcut – defines the name, description, url and Icon color for the app shortcut that will be displayed in the Client.
- Host – defines a resolvable name or IP address. If resource names are returned, then later the name resolver will resolve these to IP addresses.
- Port or Type – defines one or more port numbers or ICMP types.

Scripts may request information from other systems that require credentials (which will have to be included in the JavaScript). Scripts used in Entitlements and Conditions are therefore passed in the encrypted portion of the Entitlement token.

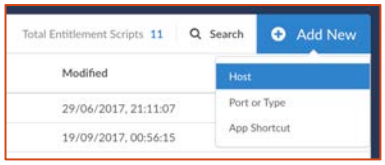
Why use?

Entitlements that contain scripts (based on user/device claims) are initially run when the user connects to each Gateway. The script might be designed to create a resource name by concatenating together a claim value (such as an AD attribute) with some static syntax or a value returned from an external system. If the result was *esx://vm:<VALUE>* then this will subsequently be resolved to IPs using the vSphere name resolver. This ability to resolve different VMs based on some *<VALUE>* allows an Entitlement to effectively adapt at use-time, which can dramatically reduce the number of Policies and Entitlements that need to be configured in the system.

Usage example:

Returns a resolvable expression (aws://tag-value….) made by concatenating claims values with the value returned from an external api call.

Configure:



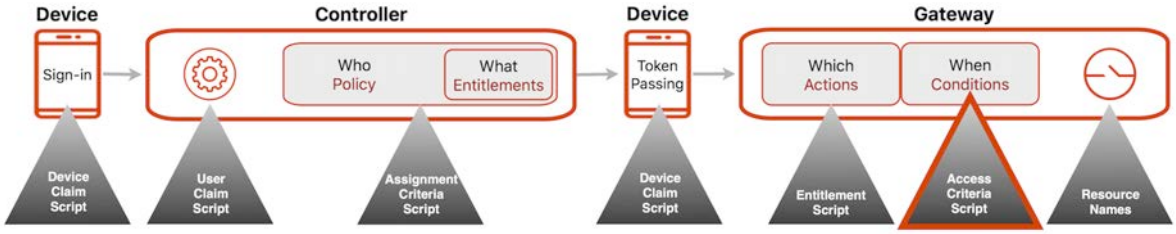
- Entitlement Scripts are added in **Scripts>Entitlement Scripts** and then selecting between the three options:
  Host - Port or Type - App Shortcuts
- Specify the scripts in **Operations>Entitlement***:*
  - under **Client App Shortcuts - using Entitlement Scripts** select the required script from the dropdown
  - under **Actions** enter the syntax *script://script_name* in either the **protected hosts (target)** and/or **ports** fields.

Code example:

Returns a list of IP addresses.

```
return JSON.parse(httpGet("https://dev-ops.int.appgate.com/test-runners/executions-ips").data);
```

Access Criteria Script



The Gateways evaluate access criteria expressions included in Conditions to make access decisions. If true, then the Actions are allowed; if false: a user interaction is initiated and sent to the Client.

There is no requirement to write a script - expressions can be configured within the Condition itself using the criteria tool options and claims pick-list. This will generate the resulting JavaScript expression for you. However, when an access criteria script is used, it can combine existing user, device and system claims with extra dynamic information and make complex access decisions in near real-time. Again, it executes in a sandboxed JavaScript engine with http Get/Post/Put calls allowed.

As noted previously, when writing a script, it should be as efficient as possible as every time a user opens a new TCP stream the script will be run. Instead of using http calls to get static claims (i.e. user group), consider using a user claim script as this will only be run once per sign-in and the Controller will add any new claims to the claims token. This will reduce the latency caused by the Gateways having to make frequent (unnecessary) external API calls.

Scripts may request information from other systems that require credentials (which will have to be included in the JavaScript). Scripts used in Entitlements and Conditions are therefore passed in the encrypted portion of the Entitlement token.

Why use:

Every entitlement must include a Condition which by default is set to *always*. The default Condition can be replaced by more specific Conditions in order to be able to make conditional access decisions.

There are several reasons why you might want to use an access criteria script (as opposed to just a Boolean expression).

- The complexity of the Boolean expression required goes beyond what the builder can support.
- You switched from *custom logic is met* to *script returns true* and made an edit.
- An external call is required to harvest claims from some other third-party system.

Usage example:

Allows the use of real-time user interactions - such as requesting MFA to access a specific protected host because an intrusion detection system is at warning level 3.

Configure:

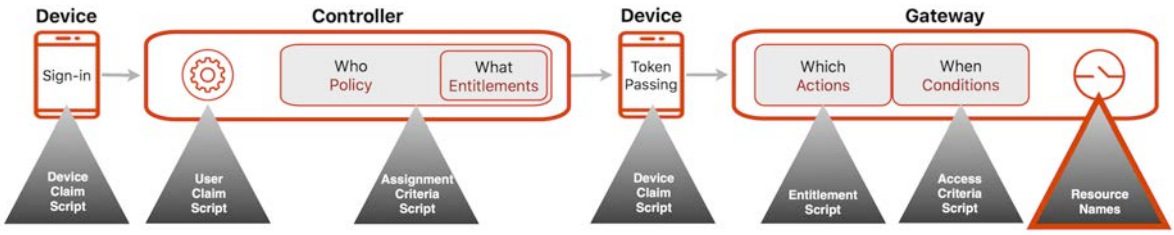
- Enter the script directly in **Operations>Conditions>Access Criteria** using **script returns true** and save the Condition.
- In **Operations>Entitlements>Condition** Select the saved Condition from the drop down.

Code example:

Get devices by hostname then request the scan history of the device:

```
var log2Console = true; //Visible UI edit mode test panel
var log2Audit = false;  //Visible in audit logs

function log(msg) {
   var prefix = "myScript_ref: ";
   msg = prefix + msg;

   if (log2Console)
      console.log(msg + "; ");
   if (log2Audit)
      auditLog(msg);
}
function GetDeviceId(){
        var claimId=null;
        var hostname = claims.device.os.hostname
        var requestDeviceId=skorpionUri+"?filter_hostname="+hostname+"&limit=1"

        var deviceIdResponse=httpGet(requestDeviceId,headers);
        if(!deviceIdResponse){
                log("No response from httpGet device id");
        }else{
                var deviceRecord = JSON.parse(deviceIdResponse.data);
                claimId = deviceRecord["id"];
        }
        return claimId;
}

function GetDeviceStatus(deviceId){
        var scanresult=null;
        var requestDeviceHealth=skorpionUri+"/"+claimId+"/scan_history?limit=1";
        var deviceHealthResponse=httpGet(requestDeviceHealth,headers);
        if(!deviceHealthResponse){
                log("No response from httpGet device health")
        }else{
                var healthRecord = JSON.parse(deviceHealthResponse.data);
                scanresult =healthRecord["result"];
                log("scan result found");
                log(scanresult);
        }
        return scanresult;
}

var deviceId = GetDeviceId();
//claims.device.SKID=deviceId;
if(!deviceId){
          log("No profile found")
}else{
          //deviceId = claims.device.SKID;
          SKscanresult = GetDeviceStatus(deviceId);
          claims.device.SKscanresult = SKscanresult;
          devicePassed = (SKscanresult == "pass")
}
return devicePassed;
```

Resource Names



Instead of managing firewall rules directly, administrators control access by defining Entitlements comprising Actions (to network resources) and Conditions (access criteria). Within an Action, the network resources can be defined by a number of means:

- IP addresses - need no further action
- Hostnames - sent to the configured DNS server to be resolved.
- Resource names - comprise special syntax that can be resolved using Cloud APIs.

Appgate SDP supports AWS, Azure, GCP and VSphere. Once configured correctly, the Gateway will automatically adjust the allowed access rules in response to changing assets and IP addresses within any virtualized environment. The name resolver re-checks the results every 60 seconds to see if they have changed. If so, then the Entitlement and access criteria scripts are re-run and the firewall rules are updated accordingly. If a hostname is returned, then the Name Resolver will then use DNS to resolve this. API calls are rate limited and cached to ensure the respective host platforms do not block these queries at busy times.

Why use?

When resource names are used it becomes possible to automate the configuration within Entitlements. This allows the network resources to be populated by external systems - something which is much more suitable for today's agile dev-ops world.

The Appgate SDP system makes API calls that return the actual instances currently active in a specific hypervisor or Cloud environment. This allows the firewall rules to be populated in near real-time and avoids any need to manually update Entitlements (firewall rules) every time a new server instance is created or removed.

Usage example:

If the resource name aws://security-group:Example was used, the Gateway will query the appropriate provider (in this case, AWS) to get the current IP addresses for all the network resources in the security group "Example". The resolvers work recursively, so if this first query does not return IP addresses (as might be the case when querying a Load Balancer) then the process is repeated. Once it returns IPs, these will be used to create the required firewall rules.

When a new host is started and tagged with "Example", it will automatically be added to the results the next time the provider is queried for information.

Configure

- Within the hosting environment (eg. AWS), use tags/security groups/namespaces/etc to identify the instances of servers, interfaces, load balancers, etc.
- Set permissions in or obtain credentials from the hosting environment that grants the Appgate SDP API user the rights to use all relevant REST API calls.
- Set up the different types of name resolvers in **System>Sites>Name Resolvers** so that they are ready to be used by the Gateway.
- Define the protected hosts using the special resource name syntax in **System>Entitlements>Actions**.

- You can test your name resolver syntax in **System>Sites**. There is a test button for each site. The syntax for testing the resolver will be exactly the same as the syntax that you configure your Actions.

Code example:

To get all instances in project ‘appgate-dev’ which are in the USA:

```
gcp://project=id:appgate-dev;instance='zone: "${ZONE_PROJECT_URL}us-*"'
```

MORE INFORMATION

This paper is focused on the overall scripting capabilities of the Appgate SDP system. It does not explore the wider aspects of how to configure an Appgate SDP system. Many parts of this paper have been integrated into the admin guide to better link these scripting and configuration aspects together. The best starting point in the manual is the topic on Automation & orchestration.

There is also the Appgate GitHub site where there is more information about scripts, extensions and customizations

Appgate delivers secure access solutions that thwart complex threats, reduce costs, boost operational efficiency and secure the lives of the people that rely on them. Through a suite of network security and fraud protection solutions - including the world’s leading Zero Trust Software-Defined Perimeter architecture.

Appgate is relied on by global enterprises seeking to protect their digital assets. Start your secure access journey with confidence by visiting https://www.appgate.com/software-defined-perimeter