Introduction

A need shared by many applications is the ability to authenticate a user and then bind a set of permissions to the user which indicate what actions the user is permitted to perform (i.e. authorization). A LocalSystem may have implemented it's own authentication and authorization and now wishes to utilize a federated Identity Provider (IdP). Typically the IdP provides an assertion with information describing the authenticated user. The goal is to transform the IdP assertion into a LocalSystem token. In it's simplest terms this is a data transformation which might include:

- renaming of data items
- conversion to a different format
- deletion of data
- addition of data
- reorganization of data

There are many ways such a transformation could be implemented:

1. Custom site specific code
2. Scripts written in a scripting language
3. XSLT
4. Rule based transforms

We also desire these goals for the transformation.

1. Site administrator configurable
2. Secure
3. Simple
4. Extensible
5. Efficient

Implementation choice 1, custom written code fails goals 1, 3 and 4, an admin cannot adapt it, it's not simple, and it's likely to be difficult to extend.

Implementation choice 2, script based transformations have a lot of appeal. Because one has at their disposal the full power of an actual programming language there are virtually no limitations. If it's a popular scripting language an administrator is likely to already know the language and might be able to program a new transformation or at a minimum tweak an existing script. Forking out to a script interpreter is inefficient, but it's now possible to embed script interpreters in the existing application. However sandboxing the execution of a script such that it cannot perform malicious operations is difficult. One could run a separate script process to provide sandboxing but this introduces the vagaries of managing a subordinate process and inter-process communication. Therefore scripts either fail goals 1 or 5, secure or efficient (depending on whether the interpreter is embedded or not).

Implementation choice 3, XSLT fails goals 3 and 4, it is not simple, it may not have the necessary features, and extending XSLT is difficult.

Implementation choice 4, rules based transformations offers the best combination of features to meet the goals. Site administrators can understand rules and edit them. There are no sandboxing concerns, rule evaluation can be crafted to be secure. With careful design the rule syntax can easily be extended. Rule execution should be efficient because it's implemented in a native library loaded into the application and does not depend on forking out to a separate process or using inter-process communication.

We describe a rule based system which offers much of the power and flexibility of a scripting language but without it's downsides. The rules are written much like a scripting language where you can call functions (e.g. verbs) and perform simple branching logic based on the result of a prior verb. Variables can be

assigned and referenced. As described in the Variables section a variable may be scalar, an associative array (key/value), or an indexed array (ordered list). A number of different data types are supported and are described in the Data Types section. Because the rules are written in JSON it should be easy for an administrator to understand and JSON provides the ability load site specific hardcoded data the rules can utilize. For example white lists, black lists, valid groups, invalid groups, etc. are all easy to express in JSON.

Implementation Status

The Mapping Rule Processor described in this document currently has implmentations in these languages:

- Java
- Python

An application simply needs to load the implementation and invoke it's entry points. It is expected the rules will reside in site specific configuration files.

General Data Flow

Figure 1. illustrates the processing inside the application when it needs to authenticate a user and provide a LocalSystem token identifying the user, providing attributes bound to the user, and a set of authorizations for the actions the user is permitted to perform (e.g. roles).

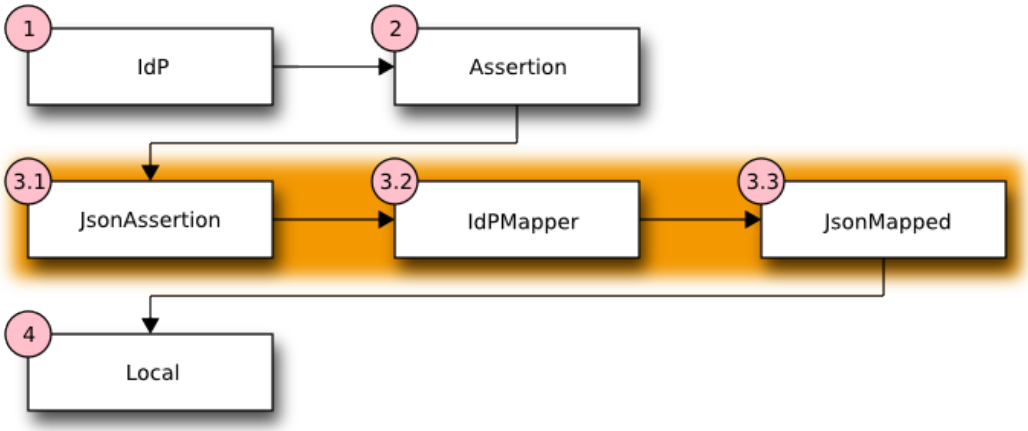

Figure 1.

Step 1: A federated Identity Provider (IdP) is asked to authenticate a user and provide additional information (attributes) concerning the user.

Step 2: Upon successful authentication the IdP provides an assertion for the user which also contains extra information (attributes) bound to the user (e.g. group membership, authorizations, etc.)

Step 3: The mapper is invoked to transform the external IdP assertion into a local representation. The transformation occurs according to the rules and mapping templates loaded into the rule processor.

The Mapping Rule Processor is designed to accept a JSON object (set of key/value pairs) as input and emit a different JSON object as output. Thus the Mappgin Rule Processor effectively operates as a transformation engine on key/value pairs with the ability to add or delete key/value pairs.

Step 3.1: The input assertion is rewritten as a JSON object in the format required by the Mapping Rule Processor. The JSON assertion is then passed into the Mapping Rule Processor.

Step 3.2: The Mapping Rule Processor identified as `IdPMapper` evaluates the input JSON assertion in the context of the mapping rules defined for the site deployment. If `IdPMapper` is able to successfully transform the input it will return a JSON object which is called the *mapped* result. If the input JSON assertion is not compatible with the site specific rules loaded into the `IdPMapper` then NULL is returned by the `IdPMapper`.

Step 3.3: If the mapping was successful the `IdPMapper` returns the transformed assertion as a JSON object.

Step 4: The mapped JSON object is converted for use in the local system.

Example Assertion Transformation

A federated IdP supplies metadata in a form unique to the IdP. This is called an assertion. That assertion must be transformed into a format and data understood by LocalSystem. More importantly that assertion needs to yield *authorization roles specific to LocalSystem*. In Figure 1. Step 3.2 the `IdPMapper` provides the transformation from an external IdP assertion to a LocalSystem specific token. It does this via a Mapping Rule Processor which reads a site specific set of transformation rules. These mapping rules define how to transform the external IdP assertion into a LocalSystem token. The mapping rules also are responsible for validating the external IdP assertion to make sure it is consistent with the site specific requirements. The operation of the Mapping Rule Processor and the syntax of the mapping rules are defined in Structure Of Rule Definitions.

Below is an example mapping rule which might be loaded into the Mapping Rule Processor to support a fictional FOOBAR application. The IdP provides assertions using the `REMOTE_USER` CGI style. It is assumed there are two LocalSystem roles which may be assigned:

`user`

A role granting standard permissions for normal FOOBAR users.

`admin`

A special role granting full FOOBAR administrative permissions.

In this example assigning the `user` and `admin` roles will be based on group membership in the following groups:

`foobar_users`

Members of this group are normal FOOBAR users with restricted permissions.

`foobar_admin`

Members of this group are FOOBAR administrators with permission to perform all operations.

Granting of the `user` and/or `admin` roles based on membership in the `foobar_users` and `foobar_admin` is illustrated in the follow mapping rule example which also extracts the user principal and domain information in the preferred format for the site (e.g. usernames are lowercase without domain suffixes and the domain is uppercase and supplied separately).

Mapping Rule Example 1.

```
1   [
2     {"mapping": {"ClientId": "$client_id",
3            "UserId": "$user_id",
4            "User": "$username",
5            "Domain": "$domain",
6            "roles": "$roles",
7           },
```

```
8       "statement_blocks": [
9         [
10          ["set", "$groups", []],
11          ["set", "$roles", []]
12        ],
13        [
14          ["in", "REMOTE_USER", "$assertion"],
15          ["exit", "rule_fails", "if_not_success"],
16          ["regexp", "$assertion[REMOTE_USER]", "(?<username>\\w+)@(?<domain>.+)"],
17          ["exit", "rule_fails", "if_not_success"],
18          ["lower", "$username", "$regexp_map[username]"],
19          ["upper", "$domain", "$regexp_map[domain]"],
20        ],
21        [
22          ["in", "REMOTE_USER_GROUPS", "$assertion"],
23          ["exit", "rule_fails", "if_not_success"],
24          ["split", "$groups", "$assertion[REMOTE_USER_GROUPS]", ":"],
25        ],
26        [
27          ["in", "foobar_users", "$groups"],
28          ["continue", "if_not_success"],
29          ["append", "$roles", "user"],
30        ],
31        [
32          ["in", "foobar_admin", "$groups"],
33          ["continue", "if_not_success"],
34          ["append", "$roles", "admin"]
35        ],
36        [
37          ["unique", "$roles", "$roles"],
38          ["length", "$n_roles", "$roles"],
39          ["compare", "$n_roles", ">", 0],
40          ["exit", "rule_fails", "if_not_success"],
41        ],
42      ]
43    }
44  ]
```

Line 1: Starts a list of rules. In this example only 1 rule is defined. Each rule is a JSON object containing a `mapping` and a required list of `statement_blocks`. The `mapping` may either be specified inside a rule as it is here or may be referenced by name in a table of mappings (this is easier to manage if you have a large number of rules and small number of mappings).

Lines 2-7: Defines the JSON mapped result. Each key maps to LocalSystem token. The value is a rule variable whose value will be substituted if the rule succeeds. Thus for example the LocalSystem token value `User` will be assigned the value from the `$username` rule variable.

Line 8: Begins the list of statement blocks. A statement must be contained inside a block.

Lines 9-12: The first block usually initializes variables that will be referenced later. Here we initialize `$groups` and `$roles` to empty arrays. These arrays may be appended to in later blocks and may be referenced in the final `mapping` output.

Lines 13-20: This block sets the user and domain information based on `REMOTE_USER` and exits the rule if `REMOTE_USER` is not defined.

Lines 14-15: This test is critical, it assures `REMOTE_USER` is defined in the assertion, if not the rule is skipped because we depend on `REMOTE_USER`.

Lines 16-17: Performs a regular expression match against `REMOTE_USER` to split the username from the domain. The regular expression uses named groups, in this instance `username` and `domain`. If the regular expression does not match the rule is skipped.

Lines 18-19: These lines reference the previous result of the regular expression match which are stored in the special variable `$regexp_map`. The username is converted to lower case and stored in `$username` and the domain is converted to upper case and stored in `$domain`. The choice of case is purely by convention and site requirements.

Lines 21-35: These 3 blocks assign roles based on group membership.

Lines 21-25: Assures `REMOTE_USER_GROUPS` is defined in the assertion; if not, the rule is skipped. `REMOTE_USER_GROUPS` is colon separated list of group names. In order to operate on the individual group names appearing in `REMOTE_USER_GROUPS` line 24 splits the string on the colon separator and stores the result in the `$groups` array.

Lines 27-30: This block assigns the `user` role if the user is a member of the `foobar_users` group.

Lines 31-35: This block assigns the `admin` role if the user is a member of the `foobar_admin` group.

Lines 36-41: This block performs final clean up actions for the rule. First it assures there are no duplicates in the `$roles` array by calling the `unique` function. Then it gets a count of how many items are in the `$roles` array and tests to see if it's empty. If there are no roles assigned the rule is skipped.

Line 43: This is the end of the rule. If we reach the end of the rule it succeeds. When a rule succeeds the mapping associated with the rule is looked up. Any rule variable appearing in the mapping is substituted with its value.

Using the rules in Mapping Rule Example 1. and following example assertion in JSON format:

Assertion Example 1.

```
{
  "REMOTE_USER": "TestUser@example.com",
  "REMOTE_AUTH_TYPE": "Negotiate",
  "REMOTE_USER_GROUPS": "foobar_users:foobar_admin",
  "REMOTE_USER_EMAIL": "test.user@example.com",
  "REMOTE_USER_FIRSTNAME": "Test",
  "REMOTE_USER_LASTNAME": "User"
}
```

Then the mapper will return the following mapped JSON document. This is the `mapping` defined on line 2 of Mapping Rule Example 1. with the variables substituted after the rule successfully executed. Note any valid JSON data type can be returned, in this example the `null` value is returned for `ClientId` and `UserId`, normal strings for `User` and `Domain` and an array of strings for the `roles` value.

Mapped Result Example 1.

```
{
  "ClientId": null,
  "UserId": null,
  "User": "testuser",
  "Domain": "EXAMPLE.COM",
```

```
  "roles": ["user", "admin"]
}
```

Operation Model

The assertions from an IdP are stored in an associative array. A sequence of rules are applied, the first rule which returns success is considered a match. During the execution of each rule values from the assertion can be tested and transformed with the results selectively stored in variables local to the rule. If the rule succeeds an associative array of mapped values is returned. The mapped values are taken from the local variables set during the rule execution. The definition of the rules and mapped results are expressed in JSON notation.

A rule is somewhat akin to a function in a programming language. It starts execution with a set of predefined local variables. It executes statements which are grouped together in blocks. Execution continues until an exit statement returning a success/fail result is executed or until the last statement is reached which implies success. The remaining statements in a block may be skipped via a continue statement which tests a condition, this is equivalent to an "if" control flow of logic in a programming language.

Rule execution continues until a rule returns success. Each rule has a mapping associative array bound to it which is a template for the transformed result. Upon success the mapping template for the rule is loaded and the local variables from the successful rule are used to populate the values in the mapping template yielding the final mapped result.

If no rules returns success authentication fails.

Pseudo Code Illustrating Operational Model

```
mapped = null
foreach rule in rules {
    result = null
    initialize rule.variables with pre-defined values

    foreach block in rule.statement_blocks {
        for statement in block.statements {
            if statement.verb is exit {
                result = exit.status
                break
            }
            elif statement.verb is continue {
                break
            }
        }
        if result {
            break
        }
    if result == null {
        result = success
    }
if result == success {
    mapped = rule.mapping(rule.variables)
}
return mapped
```

Structure Of Rule Definitions

Rules are loaded by the rule processor via a JSON document called a rule definition. A definition has an *optional* set of mapping templates and a list of rules. Each rule has specifies a mapping template and has a list of statement blocks. Each statement block has a list of statements.

In pseudo-JSON (JSON does not have comments, the ... ellipsis is a place holder):

```
{
    "mappings": {
                    "template1": "{...}",
                    "template2": "{...}"
                },
    "rules": [
                 {   # Rule 0. A rule has a mapping or a mapping name
                     # and a list of statement blocks

                     "mapping": {...},
                     # -OR-
                     "mapping_name": "template1",

                     "statement_blocks": [
                                             [   # Block 0
                                                 [statement 0]
                                                 [statement 1]
                                             ],
                                             [   # Block 1
                                                 [statement 0]
                                                 [statement 1]
                                             ],

                                          ]
                 },
                 {   # Rule 1 ...
                 }
            ]

}
```

Mapping

A mapping template is used to produce the final associative array of name/value pairs. The template is a JSON Object. The value in a name/value pair can be a constant or a variable. If the template value is a variable the value of the variable is retrieved from the set of local variables bound to the rule thereby replacing it in the final result.

For example given this mapping template and rule variables in JSON:

template:

```
{
    "organization": "BigCorp.com",
    "user: "$subject",
    "roles": "$roles"
}
```

local variables:

```
{
    "subject": "Sally",
    "roles": ["user", "admin"]
}
```

The final mapped results would be:

```
{
    "organization": "BigCorp.com",
    "user: "Sally",
    "roles": ["user", "admin"]
}
```

Each rule must bind a mapping template to the rule. The mapping template may either be defined directly in the rule via the `mapping` key or referenced by name via the `mapping_name` key.

If the `mapping_name` is specified the mapping is looked up in a table of mapping templates bound to the Rule Processor. Using the name of a mapping template is useful when many rules generate the exact same template values.

If both `mapping` and `mapping_name` are defined the locally bound `mapping` takes precedence.

Syntax

The logic for a rule consists of a sequence of statements grouped in blocks. A statement is similar to a function call in a programming language.

A statement is a list of values the first of which is a verb which defines the operation the statement will perform. Think of the verbs as function names or operators. Following the verb are parameters which may be constants or variables. If the statement assigns a value to a variable left hand side of the assignment (lhs) is always the first parameter following the verb in the list of statement values.

For example this statement in JSON:

```
["split", "$groups", "$assertion[Groups]", ":"]
```

will assign an array to the variable `$groups`. It looks up the string named `Groups` in the assertion which is a colon (:) separated list of group names splitting that string on the colon character.

Statements **must** be grouped together in blocks. Therefore a rule is a sequence of blocks and block is a sequence of statements. The purpose of blocks is allow for crude flow of control logic. For example this JSON rule has 4 blocks.

```
[
    [
        ["set", $user, ""],
        ["set", $roles, []]
    ],
    [
        ["in", "UserName", "$assertion"],
        ["continue", "if_not_success"],
        ["set", "$user", "$assertion[UserName"],
    ],
    [
```

```
        ["in", "subject", "$assertion"],
        ["continue", "if_not_success"],
        ["set", "$user", "$assertion[subject]"],
    ],
    [
        ["length", "$temp", "$user"],
        ["compare", "$temp", ">", 0],
        ["exit", "rule_fails", "if_not_success"]
        ["append" "$roles", "unprivileged"]
    ]
]
```

The rule will succeed if either `UserName` or `subject` is defined in the assertion and if so the local variable `$user` will be set to the value found in the assertion and the "unprivileged" role will be appended to the roles array.

The first block performs initialization. The second block tests to see if the assertion has the key `UserName` if not execution continues at the next block otherwise the value of UserName in the assertion is copied into the variable `$user`. The third block performs a similar operation looking for a `subject` in the assertion. The fourth block checks to see if the `$user` variable is empty, if it is empty the rule fails because it didn't find either a `UserName` nor a `subject` in the assertion. If `$user` is not empty the "unprivileged" role is appended and the rule succeeds.

Data Types

There are 7 supported types which equate to the types available in JSON. At the time of this writing there are 2 implementations of this Mapping specification, one in Python and one in Java. This table illustrates how each data type is represented. The first two columns are definitions from an abstract specification. The JSON column enumerates the data type JSON supports. The Mapping column lists the 7 enumeration names used by the Mapping implemenation in each language. The following columns list the concrete data type used in that language.

JSON	Mapping	Python	Java
object	MAP	dict	Map<String, Object>
array	ARRAY	list	List<Object>
string	STRING	unicode (Python 2)	String
		str (Python 3)	
number	INTEGER	int	Long
	REAL	float	Double
true	BOOLEAN	bool	Boolean
false			
null	NULL	None	null

Rule Debugging and Documentation

If the rule processor reports an error or if you're debugging your rules by enabling DEBUG log tracing then you must be able to correlate the reported statement to where it appears in your rule JSON source. A message will always identify a statement by the rule number, block number within that rule and the statement number within that block. However once your rules become moderately complex it will become increasingly difficult to identify a statement by counting rules, blocks and statements.

A better approach is to tag rules and blocks with a name or other identifying string. You can set the Reserved Variables `rule_name` and `block_name` to a string of your choice. These strings will be reported in all messages along with the rule, block and statement numbers.

JSON does not permit comments, as such you cannot include explanatory comments next to your rules, blocks and statements in the JSON source. The `rule_name` and `block_name` can serve a similar purpose. By putting assignments to these variables as the first statement in a block you'll both document your rules and be able to identify specific statements in log messages.

During rule execution the `rule_name` and `block_name` are initialized to the empty string at the beginning of each rule and block respectively.

The above example is augmented to include this information. The rule name is set in the first statement in the first block.

```
[
    [
        ["set", "$rule_name", "Must have UserName or subject"],
        ["set", "block_name", "Initialization"],
        ["set", $user, ""],
        ["set", $roles, []]
    ],
    [
        ["set", "block_name", "Test for UserName, set $user"],
        ["in", "UserName", "$assertion"],
        ["continue", "if_not_success"],
        ["set", "$user", "$assertion[UserName"],
    ],
    [
        ["set", "block_name", "Test for subject, set $user"],
        ["in", "subject", "$assertion"],
        ["continue", "if_not_success"],
        ["set", "$user", "$assertion[subject]"],
    ],
    [
        ["set", "block_name", "If not $user fail, else append unprivileged to roles"],
        ["length", "$temp", "$user"],
        ["compare", "$temp", ">", 0],
        ["exit", "rule_fails", "if_not_success"]
        ["append" "$roles", "unprivileged"]
    ]
]
```

Variables

Variables always begin with a dollar sign ($) and are followed by an identifier which is any alpha character followed by zero or more alphanumeric or underscore characters. The variable may optionally be delimited with braces ({}) to separate the variable from surrounding text. Three types of variables are supported:

- scalar
- array (indexed by zero based integer)
- associative array (indexed by string)

Both arrays and associative arrays use square brackets ([]) to specify a member of the array. Examples of variable usage:

```
$name
${name}
$groups[0]
${groups[0]}
$properties[key]
${properties[key]}
```

An array or an associative array may be referenced by it's base name (omitting the indexing brackets). For example the associative array array named "properties" is referenced using it's base name `$properties` but if you want to access a member of the "properties" associative array named "duration" you would do this `$properties[duration]`

This is not a general purpose language with full expression syntax. Only one level of variable lookup is supported. Therefore compound references like this

```
$properties[$groups[2]]
```

will not work.

Escaping

If you need to include a dollar sign in a string (where it is immediately followed by either an identifier or a brace and identifier) and do not want to have it be interpreted as representing a variable you must escape the dollar sign with a backslash, for example "$amount" is interpreted as the variable `amount` but "\$amount" is interpreted as the string "$amount" .

Reserved Variables

A rule has the following reserved variables:

assertion

The current assertion values from the federated IdP. It is a dictionary of key/value pairs.

regexp_array

The regular expression groups from the last successful regexp match indexed by number. Group 0 is the entire match. Groups 1..n are the corresponding parenthesized group counting from the left. For example regexp_array[1] is the first group.

regexp_map

The regular expression groups from the last successful regexp match indexed by group name.

rule_number

The zero based index of the currently executing rule.

rule_name

The name of the currently executing rule. If the rule name has not been set it will be the empty string.

block_number

The zero based index of the currently executing block within the currently executing rule.

block_name

The name of the currently executing block. If the block name has not been set it will be the empty string.

statement_number

The zero based index of the currently executing statement within the currently executing block.

Examples

Split a fully qualified username into user and realm components

It's common for some IdP's to return a fully qualified username (e.g. principal or subject). The fully qualified username is the concatenation of the user name, separator and realm name. A common separator is the @ character. In this example lets say the fully qualified username is `bob@example.com` and you want to return the user and realm as independent values in your mapped result. The username appears in the assertion as the value `Principal`.

Our strategy will be to use a regular expression identify the user and realm components and then assign them to local variables which will then populate the mapped result.

The mapping in JSON is:

```
{
    "user": "$username",
    "realm": "$domain"
}
```

The assertion in JSON is:

```
{
    "Principal": "bob@example.com"
}
```

Our rule is:

```
[
    [
        ["in", "Principal", "assertion"],
        ["exit", "rule_fails", "if_not_success"],
        ["regexp", "$assertion[Principal]", (?P<username>\\w+)@(?P<domain>.+)"],
        ["set", "$username", "$regexp_map[username]"],
        ["set", "$domain", "$regexp_map[domain]"],
        ["exit, "rule_succeeds", "always"]
    ]
]
```

Rule explanation:

Block 0:

1. Test if the assertion contains a Principal value.
2. Abort the rule if the assertion does not contain a Principal value.
3. Apply a regular expression the the Principal value. Use named groupings for the username and domain components for clarity.
4. Assign the regexp group username to the $username local variable.
5. Assign the regexp group domain to the $domain local variable.
6. Exit the rule, apply the mapping, return the mapped values. Note, an explicit exit is not required if there are no further statements in the rule, as is the case here.

The mapped result in JSON is:

```
{
    "user": "bob",
    "realm": "example.com"
}
```

Build a set of roles based on group membership

Often one wants to grant roles to a user based on their membership in certain groups. In this example let's say the assertion contains a `Groups` value which is a colon separated list of group names. Our strategy is to split the `Groups` assertion value into an array of group names. Then we'll test if a specific group is in the groups array, if it is we'll add a role. Finally if no roles have been mapped we fail. Users in the group "student" will get the role "unprivileged" and users in the group "helpdesk" will get the role "admin".

The mapping in JSON is:

```
{
    "roles": "$roles",
}
```

The assertion in JSON is:

```
{
    "Groups": "student:helpdesk"
}
```

Our rule is:

```
[
    [
        ["in", "Groups", "assertion"],
        ["exit", "rule_fails", "if_not_success"],
        ["set", "$roles", []],
        ["split", "$groups", "$assertion[Groups]", ":"],
    ],
    [
        ["in", "student", "$groups"],
        ["continue", "if_not_success"],
        ["append", "$roles", "unprivileged"]
    ],
    [
        ["in", "helpdesk", "$groups"],
        ["continue", "if_not_success"],
        ["append", "$roles", "admin"]
    ],
    [
        ["unique", "$roles", "$roles"],
        ["length", "$temp", "roles"],
        ["compare", $temp", ">", 0],
        ["exit", "rule_fails", "if_not_success"]
    ]

]
```

Rule explanation:

Block 0

1. Test if the assertion contains a Groups value.
2. Abort the rule if the assertion does not contain a Groups value.
3. Initialize the $roles variable to an empty array.
4. Split the colon separated list of group names into an array of individual group names

Block 1

1. Test if "student" is in the $groups array
2. Exit the block if it's not.
3. Append "unprivileged" to the $roles array

Block 2

1. Test if "helpdesk" is in the $groups array
2. Exit the block if it's not.
3. Append "admin" to the $roles array

Block 3

1. Strip any duplicate roles that might have been appended to the $roles array to assure each role is unique.
2. Count how many members are in the $roles array, assign the length to the $temp variable.
3. Test to see if the $roles array had any members.
4. Fail if no roles had been assigned.

The mapped result in JSON is:

```
{
    "roles": ["unprivileged", "admin"]
}
```

However, suppose whatever is receiving your mapped results is not expecting an array of roles. Instead it expects a comma separated list in a string. To accomplish this add the following statement as the last one in the final block:

```
["join", "$roles", "$roles", ","]
```

Then the mapped result will be:

```
{
    "roles": "unprivileged,admin"]
}
```

White list certain users and grant them specific roles

Suppose you have certain users you always want to unconditionally accept and authorize with specific roles. For example if the user is "head_of_IT" then assign her the "user" and "admin" roles. Otherwise keep processing. The list of white listed users is hard-coded into the rule.

The mapping in JSON is:

```
{
    "user": $user,
    "roles": "$roles",
}
```

The assertion in JSON is:

```
{
    "UserName": "head_of_IT"
}
```

Our rule in JSON is:

```
[
    [
        ["in", "UserName", "assertion"],
        ["exit", "rule_fails", "if_not_success"],
        ["in", "$assertion[UserName]", ["head_of_IT", "head_of_Engineering"]],
        ["continue", "if_not_success"],
        ["set", "$user", "$assertion[UserName]
        ["set", "$roles", ["user", "admin"]],
        ["exit", "rule_succeeds", "always"]
    ],
    [
        ...
    ]
]
```

Rule explanation:

Block 0

1. Test if the assertion contains a UserName value.
2. Abort the rule if the assertion does not contain a UserName value.
3. Test if the user is in the hardcoded list of white listed users.
4. If the user isn't in the white listed array then exit the block and continue execution at the next block.
5. Set the $user local variable to $assertion[UserName]
6. Set the $roles local variable to the hardcoded array containing "user" and "admin"
7. We're done, unconditionally exit and return the mapped result.

Block 1

1. Further processing

The mapped result in JSON is:

```
{
    "user": "head_of_IT",
    "roles": ["users", "admin"]
}
```

Black list certain users

Suppose you have certain users you always want to unconditionally deny access to by placing them in a black list. In this example the user "BlackHat" will try to gain access. The black list includes the users "BlackHat" and "Spook".

The mapping in JSON is:

```
{
    "user": $user,
    "roles": "$roles",
}
```

The assertion in JSON is:

```
{
    "UserName": "BlackHat"
}
```

Our rule in JSON is:

```
[
    [
        ["in", "UserName", "assertion"],
        ["exit", "rule_fails", "if_not_success"],
        ["in", "$assertion[UserName]", ["BlackHat", "Spook"]],
        ["exit", "rule_fails", "if_success"]
    ],
    [
        ...
    ]
]
```

Rule explanation:

Block 0

1. Test if the assertion contains a UserName value.
2. Abort the rule if the assertion does not contain a UserName value.
3. Test if the user is in the hard-coded list of black listed users.
4. If the test succeeds then immediately abort and return failure.

Block 1

1. Further processing

The mapped result in JSON is:

```
Null
```

Format Strings and/or Concatenate Strings

You can replace variables in a format string using the interpolate verb. String concatenation is trivially placing two variables adjacent to one another in a format string. Suppose you want to form an email address from the username and domain in an assertion.

The mapping in JSON is:

```
{
    "email": $email,
}
```

The assertion in JSON is:

```
{
    "UserName": "Bob",
    "Domain": "example.com"
}
```

Our rule in JSON is:

```
[
    [
        ["interpolate", "$email", "$assertion[UserName]@$assertion[Domain]"],
    ]
]
```

Rule explanation:

Block 0

1. Replace the variable $assertion[UserName] with it's value and replace the variable $assertion[Domain] with it's value.

The mapped result in JSON is:

```
{
    "email": "Bob@example.com",
}
```

Note, sometimes it's necessary to utilize braces to separate variables from surrounding text by using the brace notation. This can also make the format string more readable. Using braces to delimit variables the above would be:

```
[
    [
        ["interpolate", "$email", "${assertion[UserName]}@${assertion[Domain]}"],
    ]
]
```

Make associative array lookups case insensitive

Many systems treat field names as case insensitive. By default associative array indexing is case sensitive. The solution is to lower case all the keys in an associative array and then only use lower case indices. Suppose you want the assertion associative array to be case insensitive.

The mapping in JSON is:

```
{
    "user": $user,
}
```

The assertion in JSON is:

```
{
    "UserName": "Bob"
}
```

Our rule in JSON is:

```
[
    [
        ["lower", "$assertion", "$assertion"],
        ["in", "username", "assertion"],
        ["exit", "rule_fails", "if_not_success"],
        ["set", "$user", "$assertion[username]
    ]
]
```

Rule explanation:

Block 0

1. Lower case all the keys in the assertion associative array.
2. Test if the assertion contains a username value.
3. Abort the rule if the assertion does not contain a username value.
4. Assign the username value in the assertion to $user

The mapped result in JSON is:

```
{
    "user": "Bob",
}
```

Verbs

The following verbs are supported:

- set
- length
- interpolate
- append
- unique
- regexp
- regexp_replace
- split
- join
- lower
- upper
- compare

- in
- not_in
- exit
- continue

Some verbs have a side effects. A verb may set a boolean success/fail result which may then be tested with a subsequent verb. For example the `fail` verb can be used to indicate the rule fails if a prior result is either `success` or `not_success`. The `regexp` verb which performs a regular expression search on a string stores the regular expression sub-matches as a side effect in the variables `$regexp_array` and `$regexp_map`.

Verb Definitions

set

`set $variable value`

$variable

The variable being assigned (i.e. lhs)

value

The value to assign to the variable (i.e. rhs). The value may be another variable or a constant.

set assigns a value to a variable, in other words it's an assignment statement.

Examples:

Initialize a variable to an empty array.

```
["set", "$groups", []]
```

Initialize a variable to an empty associative array.

```
["set", "$groups", {}]
```

Assign a string.

```
["set", "$version", "1.2.3"]
```

Copy the `UserName` value from the assertion to a temporary variable.

```
["set", "$temp", "$assertion[UserName]"],
```

Get the 2nd item in an array (array indexing is zero based)

```
["set", "$group", "$groups[1]"]
```

Set the associative array entry "IdP" to "kdc.example.com".

```
["set", "$metadata[IdP]", "kdc.example.com""]
```

---

length

`length $variable value`

$variable

The variable which receives the length value

value

The value whose length is to be determined. May be one of array, associative array, or string.

length computes the number of items in the value. How this is done depends upon the type of value:

array

The length is the number of items in the array.

associative array

The length is the number of key/value pairs in the associative array.

string

The length is the number of *characters* (not octets) in the string.

Examples:

Count how many items are in the `$groups` array and assign that value to the `$groups_length` variable.

```
["length", "$groups_length", "$groups"]
```

Count how many key/value pairs are in the `$assertion` associative array and assign that value to the `$num_assertion_values` variable.

```
["length", "$num_assertion_values", "$assertion"]
```

Count how many characters are in the assertion's UserName and assign the value to `$username_length`.

```
["length", "$user_name_length", "$assertion[UserName]"]
```

---

interpolate

`interpolate $variable string`

$variable

This variable is assigned the result of the interpolation.

string

A string containing references to variables which will be replaced in the string.

interpolate replaces each occurrence of a variable in a string with it's value. The result is assigned to $variable.

Examples:

Form an email address given the username and domain. If the username is "jane" and the domain is "example.com" then $email will be "jane@example.com"

```
["interpolate", "$email", "${username}@${domain}"]
```

---

append

`append $variable value`

$variable

This variable **must** be an array. It is modified in place by appending `value` to the end of the array.

value

The value to append to the end of the array.

append adds a value to end of an array.

Examples:

Append the role "qa_test" to the roles list.

```
["append", "$roles", "qa_test"]
```

---

unique

`unique $variable value`

$variable

This variable is assigned the unique values in the `value` array.

value

An array of values. **must** be an array.

unique builds an array of unique values in `value` by stripping out duplicates and assigns the array of unique values to `$variable`. The order of items in the `value` array are preserved.

Examples:

$one_of_a_kind will be assigned ["a", "b"]

```
["unique", "$one_of_a_kind", ["a", "b", "a"]]
```

---

regexp

`regexp string pattern`

string

The string the regular expression pattern is applied to.

pattern

The regular expression pattern.

regexp performs a regular expression match against `string`. The regular expression pattern syntax is defined by the regular expression implementation of the language this API is written in.

Pattern groups are a convenient way to select sub-matches. Pattern groups may accessed by either group number or group name. After a successful regular expression match the groups are stored in the special variables `$regexp_array` and `$regexp_map`.

`$regexp_array` is used to access the groups by numerical index. Groups are numbered by counting the left parenthesis group delimiter starting at 1. Group 0 is the entire match. `$regexp_array` is valid irregardless of whether you used named groups or not.

`$regexp_map` is used to access the groups by name. `$regexp_map` is only valid if you used named groups in the pattern.

Examples:

Many user names are of the form "user@domain", to split the username from the domain and to be able to work with those values independently use a regular expression and then assign the results to a variable. In this example there are two regular expression groups, the first group is the username and the second group is the domain. In the first example we use named groups and then access the match information in the special variable `$regexp_map` via the name of the group.

```
["regexp", "$assertion[UserName]", "(?P<username>\\w+)@(?P<domain>.+)"],
["continue", "if_not_success"],
["set", "$username", "$regexp_map[username]"],
["set", "$domain", "$regexp_map[domain]"],
```

This is exactly equivalent but uses numbered groups instead of named groups. In this instance the group matches are stored in the special variable `$regexp_array` and accessed by numerical index.

```
["regexp", "$assertion[UserName]", "(\\w+)@(.+)"],
["continue", "if_not_success"],
["set", "$username", "$regexp_array[1]"],
["set", "$domain", "$regexp_array[2]"],
```

---

regexp_replace

`regexp_replace $variable string pattern replacement`

$variable

The variable which receives result of the replacement.

string

The string to perform the replacement on.

pattern

The regular expression pattern.

replacement

The replacement specification.

regexp_replace replaces each occurrence of `pattern` in `$string` with `replacement`. See regexp for details of using regular expressions.

Examples:

Convert hyphens in a name to underscores.

```
["regexp_replace", "$name", "$name", "-", "_"]
```

---

split

`split $variable string pattern`

$variable

This variable is assigned an array containing the split items.

string

The string to split into separate items.

pattern

The regular expression pattern used to split the string.

split splits `string` into separate pieces and assigns the result to `$variable` as an array of pieces. The split occurs wherever the regular expression `pattern` occurs in `string`. See regexp for details of using regular expressions.

Examples:

Split a list of groups separated by a colon (:) into an array of individual group names. If $assertion[Groups] contained the string "user:admin" then $group_list will set to ["user", "admin"].

```
["split", "$group_list", "$assertion[Groups]", ":"]
```

---

join

`join $variable array join_string`

$variable

This variable is assigned the string result of the join operation.

array

An array of string items to be joined together with `$join_string`.

join_string

The string inserted between each element in `array`.

join accepts an array of strings and produces a single string where each element in the array is separated by `join_string`.

Examples:

Convert a list of group names into a single string where each group name is separated by a colon (:). If the array `$group_list` is ["user", "admin"] and the `join_string` is ":" then the `$group_string` variable will be set to "user:admin".

```
["join", "$group_string", "$groups", ":"]
```

---

lower

`lower $variable value`

$variable

This variable is assigned the result of the lower operation.

value

The value to lower case, may be either a string, array, or associative array.

lower lower cases the input value. The input value may be one of the following types:

string

The string is lower cased.

array

Each member of the array must be a string, the result is an array with the items replaced by their lower case value.

associative array

Each key in the associative array is lower cased. The values associated with the key are **not** modified.

Examples:

Lookup `UserName` in the assertion and set the variable `$username` to it's lower case value.

```
["lower", "$username", "$assertion[UserName]"],
```

Set each member of the `$groups` array to it's lower case value. If `$groups` was ["User", "Admin"] then `$groups` will become ["user", "admin"].

```
["lower", "$groups", "$groups"],
```

To enable case insensitive lookup's in an associative array lower case each key in the associative array. If `$assertion` was {"UserName": "JoeUser"} then `$assertion` will become {"username": "JoeUser"}

```
["lower", "$assertion", $assertion"]
```

---

upper

`upper $variable value`

$variable

This variable is assigned the result of the upper operation.

value

The value to upper case, may be either a string, array, or associative array.

upper is exactly analogous to lower except the values are upper cased, see lower for details.

---

in

`in member collection`

member

The value whose membership is being tested.

collection

A collection of members. May be string, array or associative array.

in tests to see if `member` is a member of `collection`. The membership test depends on the type of collection, the following are supported:

array

If any item in the array is equal to `member` then the result is success.

associative array

If the associative array contains a key equal to `member` then the result is success.

string

If the string contains a sub-string equal to `member` then the result is success.

Examples:

Test to see if the assertion contains a UserName value.

```
["in", "UserName", "$assertion"]
["continue", "if_not_success"]
```

Test to see if a group is one of "user" or "admin".

```
["in", "$group", ["user", "admin"]]
["continue", "if_not_success"]
```

Test to see if the sub-string "BigCorp" is in the assertion's `Provider` value.

```
["in", "BigCorp", "$assertion[Provider]"]
["continue", "if_not_success"]
```

---

not_in

`in member collection`

member

The value whose membership is being tested.

collection

A collection of members. May be string, array or associative array.

not_in is exactly analogous to in except the sense of the test is reversed. See in for details.

compare

`compare left operator right`

left

The left hand value of the binary operator.

operator

The binary operator used for comparing left to right.

right

The right hand value of the binary operator.

compare compares the left value to the right value according the operator and sets success if the comparison evaluates to True. The following relational operators are supported.

Operator	Description
==	equal
!=	not equal
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal

The left and right hand sides of the comparison operator *must* be the same type, no type conversions are performed. Not all combinations of operator and type are supported. The table below illustrates the supported combinations. Essentially you can test for equality or inequality on any type. But only strings and numbers support the magnitude relational operators.

Operator	STRING	INTEGER	REAL	BOOLEAN	MAP	LIST	NULL
==	X	X	X	X	X	X	X
!=	X	X	X	X	X	X	X
<	X	X	X				
<=	X	X	X				
>	X	X	X				
>=	X	X	X				

Examples:

Test to see if the `$groups` array has at least 2 members

```
["length", "$group_length", "$groups"],
["compare", "$group_length", ">=", 2]
```

exit

`exit status criteria`

status

The result for the rule.

criteria

The criteria upon which will cause the rule will be immediately exited with a failed status.

exit causes the rule being executed to immediately exit and a rule result if the specified criteria is met. Statement verbs such as in or compare set the result status which may be tested with the `success` and `not_success` criteria.

The exit `status` may be one of:

rule_fails

The rule has failed and no mapping will occur.

rule_succeeds

The rule succeeded and the mapping will be applied.

The `criteria` may be one of:

if_success

If current result status is success then exit with `status`.

if_not_success

If current result status is not success then exit with `status`.

always

Unconditionally exit with `status`.

never

Effectively a no-op. Useful for debugging.

Examples:

The rule requires `UserName` to be in the assertion.

```
["in", "UserName", "$assertion"]
["exit", "rule_fails", "if_not_success"]
```

---

continue

`continue criteria`

criteria

The criteria which causes the remainder of the *block* to be skipped.

continue is used to control execution for statement blocks. It mirrors in a crude way the *if* expression in a procedural language. `continue` does *not* affect the success or failure of a rule, rather it controls whether subsequent statements in a block are executed or not. Control continues at the next statement block.

Statement verbs such as in or compare set the result status which may be tested with the `success` and `not_success` criteria.

The criteria may be one of:

if_success

If current result status is success then exit the statement block and continue execution at the next statement block.

if_not_success

If current result status is not success then exit the statement block and continue execution at the next statement block.

always

Immediately exit the statement block and continue execution at the next statement block.

never

Effectively a no-op. Useful for debugging. Execution continues at the next statement.

Examples:

The following pseudo code:

```
roles = [];
if ("Groups" in assertion) {
    groups = assertion["Groups"].split(":");
    if ("qa_test" in groups) {
        roles.append("tester");
    }
}
```

could be implemented this way:

```
[
    ["set", "$roles", []],
    ["in", "Groups", "$assertion"],
    ["continue", "if_not_success"],
    ["split" "$groups", $assertion[Groups]", ":"],
    ["in", "qa_test", "$groups"],
    ["continue", "if_not_success"],
    ["append", "$roles", "tester"]
]
```