One Identity Safeguard for Privileged Sessions 8.0 LTS

Creating Custom Authentication and Authorization Plugins

SPS Creating Custom Authentication and Authorization Plugins
Updated - 21 October 2024, 09:03

For the most recent documents and product information, see Online product documentation.

Contents

1

Introduction

The following sections provide an overview on creating custom plugins for One Identity Safeguard for Privileged Sessions (SPS) to authenticate your users to external services in addition to the authentication performed on the target server. For example, such plugins can implement two-factor authentication (2FA) or multi-factor authentication (MFA) methods, or request the user to provide a valid ticket ID for the connection. For details on using an existing plugin, see *Integrating external authentication and authorization systems* in the *Administration Guide*.

This document is a general overview of plugin requirements. If you want to write your own custom plugin, make sure to use the not officially supported Plugin SDK. For details, see: https://oneidentity.github.io/safeguard-sessions-plugin-sdk/latest/

CAUTION:

Using custom plugins in SPS is recommended only if you are familiar with both Python and SPS. Product support applies only to SPS: that is, until the entry point of the Python code and passing the specified arguments to the Python code. One Identity is not responsible for the quality, resource requirements, or any bugs in the Python code, nor any crashes, service outages, or any other damage caused by the improper use of this feature, unless explicitly stated in a contract with One Identity. If you want to create a custom plugin, contact our Support Team for details and instructions.

Every SPS plugin is a Python module. SPS invokes the module to request the password of the target user. The plugin processes the request, returns the result to SPS and exits. SPS then processes the result.

The backup and restore functionality of SPS handles the uploaded plugins as part of SPS's configuration. You do not need to create separate backups of your plugins.

How the Authentication and Authorization plugin works

If a Connection Policy has an Authentication and Authorization plugin (**AA plugin**) configured, One Identity Safeguard for Privileged Sessions (SPS) runs the plugin as the last step of the connection authorization phase. SPS can request the client to perform other types of authentication before running the plugin. Using an **AA plugin** in a Connection Policy is treated as gateway authentication if:

- the plugin authenticates the user
- authentication is successful
- the plugin returns the `gateway_user` and `gateway_groups` elements, identifying the user it has authenticated

Other types of gateway authentication will come before authentication by the **AA plugin**, so information from any other type of gateway authentication (for example, the username and usergroups of this authentication) will already be available and therefore can be used by the plugin. If the Authentication and Authorization plugin does perform gateway authentication, you can use a Credential Store as well.

However, for technical reasons, the web-based gateway authentication (that is, authenticating on the SPS web interface if the **Require Gatweay Authentication on the SPS Web Interface** option is selected in the Connection Policy) is performed after the **AA plugin**, so using **AA plugin** and ticking **Require Gateway Authentication on the SPS Web Interface** at the same time is not a valid configuration.

The plugin can interactively request additional information from the client in the SSH, Telnet, and RDP protocols.

NOTE: In SPS 5.8, a user's group membership is determined by querying only the relevant groups configured for the connection from the LDAP/AD server, instead of retrieving all groups of a given user.

This may cause problems when using AD/LDAP-based gateway authentication together with an AA plugin. The AA plugin `authorize()` hook may be called with only a subset of groups as group membership lookup does not consider groups referenced in the AA plugin code.

Consider that only groups queried by SPS are affected. Gateway groups returned by the AA plugin `authenticate()` hook are passed to the `authorize()` hook unchanged.

SPS runs the `authorize` method after the `authentication` method and any inband gateway authentication or inband destination selection steps. As a result, the `authorize` method already has access to the IP address of the target server and the remote username (the username used in the server-side connection).

Optionally, the plugin can return the `gateway_user` and `gateway_groups` values. SPS will only update the **gateway username** and **gateway groups** fields in the connection database if the plugin returns the `gateway_user` and `gateway_groups` values. The returned `gateway_user` and `gateway_groups` values override any such attributes already available on SPS about the connection (that means that channel policy evaluations will be affected), so make sure that the plugin uses the original values appropriately.

If the plugin returns the `gateway_user` and `gateway_groups` values, you may have to configure an appropriate **Usermapping policy** in the **Connection Policy**. If the plugin returns a `gateway_user` that is different from the remote user, the connection will fail without a usermapping policy. For details on usermapping policies, see *Configuring usermapping policies* in the *Administration Guide*.

Prerequisites

- SPS supports Authentication and Authorization plugins in the RDP, SSH, and TELNET protocols.
- In RDP, using an AA plugin together with Network Level Authentication in a Connection Policy has the same limitations as using Network Level Authentication without domain membership.
- In RDP, using an AA plugin requires TLS-encrypted RDP connections. For details, see *Enabling TLS-encryption for RDP connections* in the *Administration Guide*.

3

Plugin packaging

An SPS plugin is a `.zip` file that contains a `MANIFEST` file (with no extension) and a Python module named `main.py` in its root directory. The plugin `.zip` file may also contain an optional `default.cfg` file that serves to provide an example configuration, which you can use as a basis for customization if you wish to adapt the plugin to your site's needs. The size of the `.zip` file is limited to 20 megabytes.

Including additional modules

You can invoke additional Python modules from `main.py`, provided that the total size of the `.zip` bundle does not exceed 20 megabytes and all calls are executed within the plugin timeout.

The modules must be compatible with the selected Python environment. For more information, see *The available Python environments* in the *Creating Custom Authentication and Authorization Plugins*.

The MANIFEST file

The `MANIFEST` file is a YAML file and should conform to version 1.2 of the YAML specification.

It must contain the following information about the plugin:

- `name`: The identifier of the plugin during the upload to SPS. The initial character must be an alphabetical character, while the rest may be alphabetical characters, numerals or '_'. While case sensitivity is supported, special characters (for example, '@' or '&') are not permitted.
- `description`: The description of the plugin. This description is displayed on the SPS web interface.
- `version`: The version number of the plugin. It must begin with a numeral (for example, `2.0.3`).
- `type`: The type of the plugin. It must be `credentialstore` for a Credential Store plugin and `aa` for an Authentication and Authorization plugin.
- `api`: The version number of the required SPS API. The current version number is 1.2.

It may contain the following elements:

- `entry_point: main.py`: The custom entry point of the plugin.
- `scb_min_version`: The minimum SPS product version compatible with the plugin. For example, 5.10.0 means 5F10.
- `scb_max_version`: The maximum compatible SPS product version. To allow any version below a certain value, add the ~charater. For example, 5.11.0~ means "any version up till, but not including, 5.11.0".

Example

```
name: name: SPS_RADIUS
description: RADIUS (RSA) MFA plugin plugin
version: 2.0.3
type: aa
api: 1.1
entry_point: main.py
```

API versioning

SPS supports only a single version of the plugin API.

The required version of SPS API must be in <major number>.<minor number> format.

NOTE: SPS uses semantic versioning for the API. That is, if the plugin requires API version <x>.<y>, the API version's <major number> must be equal to <x> and the <minor number> must be equal to, or greater than, <y>. Otherwise the plugin cannot be uploaded.

For example, if the API version of SPS is `1.3`, SPS can use plugins with the required API version numbers `1.0`, `1.1`, `1.2`, and `1.3`. Versions `1.4` and `2.0` will not work.

Currently the API version number is 1.2.

Plugin API versioning for Python3 plugins using the Plugin SDK module

For Python3 plugins using the Plugin SDK module the `api: version` should be the same as the <major number>.<minor number> version of the Plugin SDK. That is, if the Plugin SDK version is `1.2`, write `api: 1.2` in the `MANIFEST` file.

NOTE: The plugin does not need to be upgraded as long as the <major number> version remains the same, therefore the plugin should work with `1.3`, `1.4` or higher API versions.

NOTE: To support older SPS releases with your plugin, develop and release the plugin with an older Plugin SDK version (for more information about Plugin SDK backwards compatibility, see the History of releases).

The available Python environments

If you have entry_point: main.py in the MANIFEST file (themain.py starting with '#!/usr/bin/env pluginwrapper3')

In this case, the plugin must be Python 3.6.7 compatible. The plugin has access to these Python 3 modules:

```
oneidentity_safeguard_sessions_plugin_sdk (version == 1.7.1,
https://oneidentity.github.io/safeguard-sessions-plugin-sdk/latest/
```

NOTE: The <major> and <minor> version number of Plugin SDK is always equal to the SPS API version of the same release.

The Plugin SDK module mentioned above is a not officially supported tool that allows you to reliably access SPS features and can be downloaded from the Downloads page. In addition, the Plugin SDK module also allows you to develop or test plugins outside SPS. For more information about the Plugin SDK module, see the Developer's Guide.

- `pyOpenSSL (version >= 17.5.0, https://pyopenssl.org/en/17.5.0/index.html)`
- `python-ldap (version >= 3.0.0, https://www.python-ldap.org/en/python-ldap-3.0.0/)`
- `requests (version >= 2.18.4, http://docs.python-requests.org/en/master/)`
- `urllib3 (version >= 1.22, https://urllib3.readthedocs.io/en/latest/)`
- `pyyaml (version >= 3.12, https://pyyaml.org/)`

If you have no entry_point in the MANIFEST file

The plugins must be compatible with Python version 2.6.5, and have access to the following Python modules:

- `dns`
- `httplib`
- `json`
- `lxml`
- `openssl`
- `urllib`
- `urllib2`
- `xml`
- `xmllib`
- `xmlrpclib`

The main.py module

The `main.py` file is a Python module that the framework attempts to run. The following restrictions apply:

- The `main.py` module must contain the `Plugin` class. SPS searches for the plugin hook implementations under the `Plugin` class. SPS instantiates this class and invokes the hooks on the resulting instance.
- The `Plugin` class must have an `__init__(self, configuration="")` method. This is how the **Configuration** (for example, at **Policies** > **AA Plugin Configuration** > **Configuration** or **Policies** > **Credential Stores** > **Configuration**) is passed to the `Plugin` instance as string.
- The `Plugin` class must have member methods for all defined hooks.

The plugin is executed when a predefined entry point (hook method) is invoked. After returning the result, the plugin exits immediately.

NOTE: Plugins have a global timeout limit. The plugin timeout is half of the timeout value of the protocol proxy that uses the plugin (configured on the **Traffic Controls** > **Protocol name** > **Settings** page of the SPS web interface). By default, the proxy timeout is `600` seconds,therefore the default plugin timeout is `300` seconds.

Hooks can be defined with zero or more arguments and can usually return `None` or a dict with the appropriate keys. The order of the hook arguments is not defined. Instead, all arguments are passed by name.

All arguments are optional. Only the arguments actually used in the hook need to be specified.

No global state is preserved inbetween calls. Therefore, you have to use the `cookie` key in the returned dictionary to persist data between subsequent calls of the same plugin or between the different methods of a plugin. The cookie should be a dictionary containing simple data items. It has to be serializable to JSON. To persist data between two different plugins used in the same session, use the `session_cookie` key.

You can use (`**kwargs`) to get all possible call arguments in a hook, including the `cookie` argument.

The following hooks must all be implemented:

- authenticate on page 12: Called to identify the user connecting through SPS.
- authorize on page 18: Called when the remote username and the address of the target server are available (after the authentication hook and any inband gateway authentication or inband destination selection are completed).
- session_ended on page 24: Called when the session is closed. It is called exactly once for the same session. For example, you can use this hook to send a log message related to the entire session, or close the ticket related to the session if the plugin interacts with a ticketing system.

authenticate

The `authenticate` method performs the authentication of the session and returns a verdict that determines if SPS permits the connection to continue to the target server.

Example

```
def authenticate(self,
                         session_id,
                         protocol,
                         connection_name,
                         client_ip,
                         client_port,
                         key_value_pairs):
      return {
            'verdict': 'ACCEPT',
            'additional_metadata': 'my_metadata',
            'my_key': 'my_value',
      }
```

You must implement the `authenticate` method in the plugin.

TIP: If you do not want to do anything in this method, include an empty method that returns the `ACCEPT` verdict.

Example

```
def authenticate (self):
      return {
            'verdict': 'ACCEPT',
      }
```

In addition, no gateway authentication has been performed by the plugin if the `authenticate` method returns:

- `None`.
- The dict `{'verdict': 'NONE'}`.

Input arguments

The order of the arguments does not make a difference, only their names do. Every argument is optional.

- `session_id`

  Type: string

  Description: The unique identifier of the session.

  `cookie`

  Type: dictionary

  Description: The cookie returned by the previous hook in the session. If this is the first call for that session, it is initialized as an empty dictionary, otherwise it has the value returned by one of the previous calls in this particular AA plugin. You can use the cookie to maintain the state for each particular connection or to transfer information between the different methods of the plugin. For an example that transfers information in the cookie between two methods, see Examples on page 25.

- `session_cookie`

  Type: dictionary

  Description: You can use the session cookie to maintain global state between plugins for each particular connection. If this is the first call for that session, it is initialized as an empty dictionary, otherwise it has the value returned by a previous plugin hook in the session.

- `connection_name`

  Type: string

  Description: The name of the Connection policy that handles the client's connection request.

- `client_hostname`

  Type: string

  Description: A string containing the hostname of the client, if DNS lookup has been successful. If not, the value of this parameter is `None`.

- `client_ip`

Type: string

Description: A string containing the IP address of the client.

- `client_port`

  Type: int

  Description: The port number of the client.

- `gateway_user`

  Type: string

  Description: Contains the gateway username of the client if already available (for example, if the user performed inband gateway authentication), otherwise its value is `None`.

- `key_value_pairs`

  Type: dictionary

  Description: A dictionary containing plugin-specific information (for example, it may include a token ID). This dictionary also contains any key-value pairs that the user specified. In the plugin, such fields are already parsed into separate key-value pairs. For details on how the user can provide such data during a connection, see *Integrating external authentication and authorization systems* in the *Administration Guide*.

- `protocol`

  Type: string

  Description: The protocol used in the connection that the plugin is currently processing. Enter one of the following values: **`rdp`**, **`ssh`**, **`telnet`**.

- `target_username`

  Type: string or `None`

  Description: Contains information about the target username if already available (for example, if the user performed inband gateway authentication), otherwise its value is `None`.

Returned values

The method must return a dictionary with the following (required or optional) elements.

The required elements are:

- `verdict`, which must contain one of the following returned values:
  - `ACCEPT`, which returns `gateway_user` and `gateway_groups` together.
  - `NEEDINFO`, which returns `question`.
  - `DENY`
  - `NONE`

The optional elements are:

- `cookie`
- `session_cookie`
- `additional metadata`
- `gateway_user`
- `gateway_groups`
- `question`

The elements in more detail:

- `verdict`

Type: string
Required: yes

  Description: Must contain one of the following values:

  - `ACCEPT`:The authentication was successful, the client can continue the connection

    If the plugin returns both `gateway_users` and `gateway_groups` elements, it means that gateway authentication has been performed.
  - `DENY`: Reject the connection.
  - `NEEDINFO`: The authentication requires more information to be completed.
  - `NONE`: No gateway authentication was performed by the plugin.

  For example, the following sample code rejects the connection.

  Example

  ```
  return {'verdict': 'DENY'}
  ```

- `cookie`

  Type: dictionary

  Required: no

  Description: The cookie returned by the previous hook in the session. If this is the first call for that session, it is initialized as an empty dictionary, otherwise it has the value returned by one of the previous calls in this particular AA plugin. You can use the cookie to maintain the state for each particular connection or to transfer information between the different methods of the plugin. For an example that transfers information in the cookie between two methods, see Examples on page 25.

- `session_cookie`

  Type: dictionary

  Required: no

  Description: You can use the session cookie to maintain global state between plugins for each particular connection. If this is the first call for that session, it is initialized as an empty dictionary, otherwise it has the value returned by a previous plugin hook in the session.

- `additional_metadata`

  Description: The value of this string will be stored in the **Additional metadata** column of the SPS connection database, and will be available on the SPS search interface.

- `gateway_user`

  Type: string

  Required: no

- `gateway_groups`

  Type: list

  Required: no

  Description: Optionally, the plugin can return the `gateway_user` and `gateway_groups` values. SPS will only update the gateway username and gateway groups fields in the connection database if the plugin returns the `gateway_user` and `gateway_groups` values. The returned `gateway_user` and `gateway_groups` values override any such attributes already available on SPS about the

connection (which means that channel policy evaluations will be affected), so make sure that the plugin uses the original values appropriately.

> NOTE: If the plugin returns the `gateway_user` and `gateway_groups` values, you may have to configure an appropriate **Usermapping Policy** in the Connection Policy. If the plugin returns a `gateway_user` that is different from the remote user, the connection will fail without a Usermapping Policy. For details on Usermapping Policies, see *Configuring usermapping policies* in the *Administration Guide*.

For example, the following sample code accepts the connection and sets the `gateway_user` and `gateway_groups` fields. (Naturally, you should write the plugin code that actually retrieves these data from the third-party system.) For details, see Examples on page 25.

Example

```
return {
       'verdict': 'ACCEPT',
       'gateway_user': 'username-received-from-third-party',
       'gateway_groups': [
             'usergroup1-received-from-third-party',
             'usergroup2-received-from-third-party'
       ]
}
```

- `question`

Type: tuple
Required: no

Description: A tuple that contains key-question pairs and optionally a third element to disable echoing. You can use it to request additional information from the client when using the `NEEDINFO` verdict in RDP, Telnet, and SSH connections. For example, the following sample code displays a prompt (in this case, `Enter your token number`) to the user. For details, see Examples on page 25.

Example

```
return {
       'verdict': 'NEEDINFO',
       'question': ('token', 'Enter your token number: ')
}
```

If the optional third element is `True`, the answer will not be echoed to the client.

TIP: Set the third element to **`True`** if the answer to the question is sensitive information (for example, a password).

Example

```
return {
       'verdict': 'NEEDINFO',
       'question': ('token', 'Enter your token number: ',
True)
}
```

Note that in SPS version 4.3.0 and 4.3.1, `question` was a dictionary. Starting with version 4.3.2, it is a tuple.

Requesting more information from the client

To request additional information from the client (for example, a one-time password from a token, or a ticket ID), the `authenticate` method may return the `NEEDINFO` verdict and the `question` tuple containing key-question pairs. The questions are asked from the user in a protocol-specific way and the `authenticate` method is called again with a `key_value_pairs` argument containing the answers in key-answer pairs, where the key belongs to the corresponding question. Alternatively, you can also use the `cookie` to supply additional information to the plugin.

authorize

The `authorize` method performs the authorization of the session and returns a verdict that determines if SPS permits the connection to continue to the target server. This method is executed only once.

SPS executes the `authorize` method after the `authentication` method, and any inband gateway authentication or inband destination selection steps. As a result, the `authorize`

method already has access to the IP address of the target server and the remote username (the username used in the server-side connection).

You must implement the `authorize` method in the plugin.

TIP: If you do not want to do anything in this method, include an empty method that returns the `ACCEPT` verdict. Otherwise, the connection will fail with the following log message: `Calling Authorize hook of AA plugin failed..`

Example

```
def authorize (self):
    return {'verdict': 'ACCEPT'}
```

Input arguments

The order of the arguments does not make a difference, only their names do. Every argument is optional.

- `session_id`

  Type: string

  Description: The unique identifier of the session.

- `cookie`

  Type: dictionary

  Description: The cookie returned by the previous hook in the session. If this is the first call for that session, it is initialized as an empty dictionary, otherwise it has the value returned by one of the previous calls in this particular AA plugin. You can use the cookie to maintain the state for each particular connection or to transfer information between the different methods of the plugin. For an example that transfers information in the cookie between two methods, see Examples on page 25.

- `session_cookie`

  Type: dictionary

  Description: You can use the session cookie to maintain global state between plugins for each particular connection. If this is the first call for that session, it is initialized as an empty dictionary, otherwise it has the value returned by a previous plugin hook in the session.

- `connection_name`

Type: string

Description: The name of the Connection Policy that handles the client's connection request.

- `client_hostname`

  Type: string

  Description: A string containing the hostname of the client, if DNS lookup has been successful. If not, the value of this parameter is `None`.

- `client_ip`

  Type: string

  Description: A string containing the IP address of the client.

- `client_port`

  Type: int

  Description: The port number of the client.

- `gateway_groups`

  Type: string list

  Description: The final gateway groups of the gateway user.

- `key_value_pairs`

  Type: dictionary

  Description:

  A dictionary containing plugin-specific information (for example, it may include the username).

  This dictionary also contains any key-value pairs that the user specified when establishing the connection. In the plugin, such fields are already parsed into separate key-value pairs. For details on how the user can provide such data during a connection, see *Integrating external authentication and authorization systems* in the *Administration Guide*.

- `protocol`

Type: string

Description: The protocol used in the connection that the plugin is currently processing. Enter one of the following values: **`rdp`**, **`ssh`**, **`telnet`**.

- `client_port`

  Type: int

  Description: The port number of the client.

- `target_server` - DEPRECATED

  Type: string or `None`

  Description: This parameter is DEPRECATED. Use `server_ip` instead. Contains information about the target server if already available (for example, if the user performed inband gateway authentication), otherwise its value is `None`.

- `server_ip`

  Type: string or `None`

  Description: Contains information about the target server if already available (for example, if the user performed inband gateway authentication), otherwise its value is `None`.

- `target_port` - DEPRECATED

  Type: integer or `None`

  Description: This parameter is DEPRECATED. Use `server_port` instead. Contains information about the target port if already available (for example, if the user performed inband gateway authentication), otherwise its value is `None`.

- `server_port`

  Type: integer or `None`

  Description: Contains information about the target port if already available (for example, if the user performed inband gateway authentication), otherwise its value is `None`.

- `target_username` - DEPRECATED

Type: string

Description: This parameter is DEPRECATED. Use `server_username` instead. The username SPS uses to authenticate on the target server.

- `server_username`

  Type: string

  Description: The username SPS uses to authenticate on the target server.

- `server_hostname`

  Type: string

  Description: A string containing the hostname of the server, if DNS lookup has been successful. If not, the value of this parameter is `None`.

Returned values

The method must return a dictionary with the following (required or optional) elements.

The required elements are:

- `verdict`, which must contain one of the following returned values:
  - `ACCEPT`, which indicates that the authentication was successful and the client can continue the connection.
  - `DENY`, which rejects the connection.

The optional elements are:

- `cookie`
- `session_cookie`
- `additional metadata`

The elements in more detail:

- `verdict`

  Type: string

  Must contain one of the following values:

  - `ACCEPT`: The authentication was successful, the client can continue the connection.
  - `DENY`: Reject the connection.

  For example, the following sample code rejects the connection.

Example

```
return  {
       'verdict':  'DENY'
 }
```

- `cookie`

  Type: dictionary

  Required: no

  Description: The cookie returned by the previous hook in the session. If this is the first call for that session, it is initialized as an empty dictionary, otherwise it has the value returned by one of the previous calls in this particular AA plugin. You can use the cookie to maintain the state for each particular connection or to transfer information between the different methods of the plugin. For an example that transfers information in the cookie between two methods, see Examples on page 25.

- `session_cookie`

  Type: dictionary

  Required: no

  Description: You can use the session cookie to maintain global state between plugins for each particular connection. If this is the first call for that session, it is initialized as an empty dictionary, otherwise it has the value returned by a previous plugin hook in the session.

- `additional_metadata`

  Type: string

  Required: no

  Description: The value of this string will be stored in the **Additional metadata** column of the SPS connection database, and will be available on the SPS search interface.

session_ended

A session is the logical unit of user connections: it starts with logging in to the target, and ends when the connection ends. SPS executes the `session_id` hook when the session is closed. It is called exactly once for the same session.

> TIP: You can use this hook to send a log message related to the entire session or close the ticket related to the session if the plugin interacts with a ticketing system.

You must implement the `session_ended` method in the plugin.

Input arguments

- `session_id`

  Type: string

  Description: The unique identifier of the session.

- `cookie`

  Type: dictionary

  Description: The cookie returned by the previous hook in the session. If this is the first call for that session, it is initialized as an empty dictionary, otherwise it has the value returned by one of the previous calls in this particular AA plugin. You can use the cookie to maintain the state for each particular connection or to transfer information between the different methods of the plugin. For an example that transfers information in the cookie between two methods, see Examples on page 25.

- `session_cookie`

  Type: dictionary

  Description: You can use the session cookie to maintain global state between plugins for each particular connection. If this is the first call for that session, it is initialized as an empty dictionary, otherwise it has the value returned by a previous plugin hook in the session.

Returned values

This hook does not return values.

session_ended example

The following example formats every information received in the cookie into key-value pairs and prints a log message including this information into the log file.

Example

```
def session_ended(self, session_id, session_cookie, cookie):
    session_details = ','.join(['{0}={1}'.format(
        key, cookie[key]) for key in sorted cookie.keys()
    ])
    print("Session ended; session_id='{0}', session_details='{1}'".
        format(session_id, session_details))
```

Examples

The following example checks if the user has entered the string **good** as the token number. If the value of the token number is anything other than `good`, the plugin displays a prompt to the user up to three times. After three unsuccessful attempts, the plugin terminates the connection.

Example

```
def authenticate(self, key_value_pairs, cookie):
      if key_value_pairs.get('token') == "good":
            return {'verdict': 'ACCEPT'}

      cookie['cnt'] = cookie.get('cnt', 0) + 1
      if cookie['cnt'] > 3:
            return {'verdict': 'DENY'}

      return {'verdict': 'NEEDINFO',
                  'question': ('token', 'Enter token number: '),
                  'cookie': cookie
      }
```

The following example shows how to use the cookie to transfer data from the `authenticate` method to the `session_ended` method.

Example

```
import sys


class Plugin(object):

    def authenticate(self, session_id, cookie, protocol,
    connection_name, client_ip, client_port, key_value_pairs):
        token = key_value_pairs.pop('token', None)

    # Accept the connection if the user provides a token number
    if token:
        # Write code here that validates the token number and
        retrieves the username and usergroups of the user
        # We add the client_ip to the 'cookie' so it will be
        available in the session_ended method as well
        return {
            'verdict': 'ACCEPT',
            'gateway_user': 'username-received-from-third-party',
            'gateway_groups': [
                'usergroup1-received-from-third-party',
                'usergroup2-received-from-third-party'],
            'additional_metadata': token,
            'cookie': {'client_ip': client_ip}
        }

    # Display a prompt to the user to request a token number
    else:
        return {
            'verdict': 'NEEDINFO',
            'question': ('token', 'Enter your token number: ')
        }

    def session_ended(self, session_id, cookie):
        session_details = ','.join([
            '{0}={1}'.format(key, cookie[key]) for key in
            sorted(cookie.keys())
        ])

    # Send a log message when the session ends, including the
    # client_ip address received in the cookie
    print("Session ended; session_id='{0}', session_details='{1}'".
        format(session_id, session_details))
```

Tips and tricks

If you need the public hostname of SPS in the plugin, the plugin can read it from the `/etc/hostnickname` file.

The sample configuration file (default.cfg)

Your plugin `.zip` file may contain an optional `default.cfg` sample configuration file. This file serves to provide an example configuration that you can use as a basis for customization if you wish to adapt the plugin to your site's needs.

The only prerequisites for this file are as follows:

- It must be a UTF-8 encoded text file.
- The size of the file must not exceed 10 KiB.

Other than these prerequisites, the contents of the file are not restricted in any way.

Plugin troubleshooting

On the default log level, One Identity Safeguard for Privileged Sessions (SPS) logs everything that the plugin writes to `stdout` and `stderr`. Log message lines are prefixed with the session ID of the proxy, which makes it easier to find correlating messages.

To transfer information between the methods of a plugin (for example, to include data in a log message when the session is closed), you can use a cookie.

If an error occurs while executing the plugin, SPS automatically terminates the session.

NOTE: This error is not visible in the verdict of the session. To find out why the session was terminated, you have to check the logs.

5

Integrating SPS to ticketing systems

From SPS 5 LTS and later, this functionality is available using the Authentication and Authorization (AA) plugin.SPS executes the `authorize` method after the `authentication` method, and any inband gateway authentication or inband destination selection selection steps. As a result, the `authorize` method already has access the IP address of the target server, and the remote username (that is, the username used in the server-side connection).

To use an AA plugin to integrate SPS to a ticketing system, note the following points.

- You can only request the ticket ID or other information from the user in the authentication hook (authenticate on page 12). For details on how the user can provide such data during a connection, see *Integrating external authentication and authorization systems* in the *Administration Guide*.
- You must implement the actual authorization (for example, connecting and querying the ticketing system) in authorize on page 18. As a side effect, if the user submits an invalid ticket ID (or other invalid information) in the authentication hook, this error will not be recognized until the authorization hook. The user cannot correct this error and SPS will reject the connection. In this case, the user must initiate a new connection to provide the correct information.
- Only the Remote Desktop (RDP), Secure Shell (SSH), and Telnet protocols are supported.

About us

One Identity solutions eliminate the complexities and time-consuming processes often required to govern identities, manage privileged accounts and control access. Our solutions enhance business agility while addressing your IAM challenges with on-premises, cloud and hybrid environments.

Contacting us

For sales and other inquiries, such as licensing, support, and renewals, visit https://www.oneidentity.com/company/contact-us.aspx.

Technical support resources

Technical support is available to One Identity customers with a valid maintenance contract and customers who have trial versions. You can access the Support Portal at https://support.oneidentity.com/.

The Support Portal provides self-help tools you can use to solve problems quickly and independently, 24 hours a day, 365 days a year. The Support Portal enables you to:

- Submit and manage a Service Request
- View Knowledge Base articles
- Sign up for product notifications
- Download software and technical documentation
- View how-to videos at www.YouTube.com/OneIdentity
- Engage in community discussions
- Chat with support engineers online
- View services to assist you with your product