

# Password Safe Platform Plugin Software Development Kit

Password Safe supports an extensive but finite list of platforms. However, with BeyondTrust's software development kit (SDK), a developer can build a new platform plugin to add support for new systems and applications on an as-needed basis.

The Password Safe plugin SDK package can be found in the Password Safe Resource Kit. To obtain the Resource Kit, download it from the Customer Portal at <https://beyondtrustcorp.service-now.com/csm>. If you do not have access to the Customer Portal, contact your account manager or submit a request to Sales at <https://www.beyondtrust.com/contact>.

The Password Safe plugin SDK requires the following prerequisites:

- BeyondInsight/Password Safe 21.3+
- Microsoft Visual Studio 2019+
- .NET Core 3.1

The plugin SDK contains detailed developer documentation, code samples, utilities, and a code generator template which quickly and easily creates all the scaffolding required for your plugin, with stub implementations of each of the features offered by the SDK. These features include:

- Managed account credential rotation
- Managed account credential rotation via a functional account
- Test managed account credential
- Test functional account credential
- Functional account credential rotation

## Getting Started with the Plugin SDK

Follow these steps to use the Password Safe Cloud Plugin SDK to build a new platform plugin.

1. Extract the **PlatformPlugin.Generator.zip** archive into the desired location.

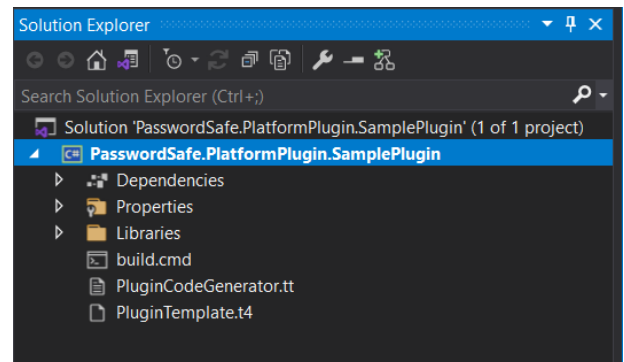


**Note:** The C# Project file (CSPROJ) should be renamed to more suitably represent the plugin, such as: **PasswordSafe.PlatformPlugin.<platform\_name>.csproj**.

2. Open the C# project using Microsoft Visual Studio 2019.



**Note:** .NET Core 3.1 framework is a prerequisite and must be installed.



3. Edit the **PluginCodeGenerator.tt** file to include your company name as well as the plugin name, version, and description.

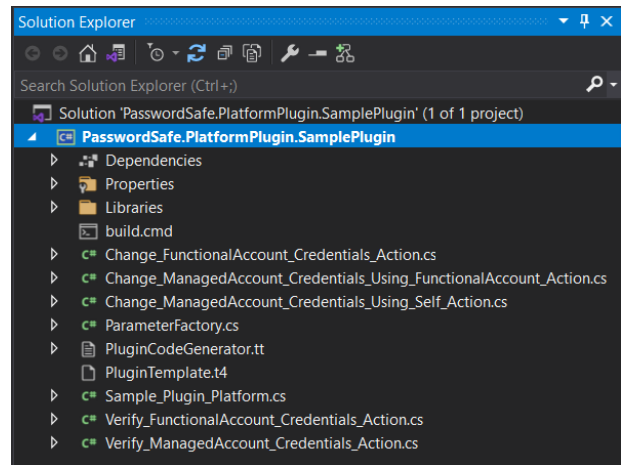


**Note:** Make sure the **RegenerateCode** variable is set to **true**. This triggers the automatic generation of the plugin code scaffolding.

```

PluginCodeGenerator.tt
1 <#@ template debug="false" hostspecific="true" language="C#" #>
2 <#@ include file="PluginTemplate.t4" #>
3 <#
4 // PluginCodeGenerator.t4
5 #>
6 <#+
7
8 bool RegenerateCode => true;
9 string PlatformName => "Sample Plugin";
10 string PluginVersion => "1.0.0.0";
11 string CompanyName => "ABC Company";
12 string PluginDescription => "Sample Plugin Description";
13 string UniqueId => Guid.NewGuid().ToString(); // or use any Guid as string value
14 #>
  
```

4. Save the **PluginCodeGenerator.tt** file. The template generator automatically creates all files for the plugin.



5. To prevent the code regeneration on every build, set the value of the **RegenerateCode** variable to **false** in the **PluginCodeGenerator.tt** file.

6. Implement the desired methods in the plugin. Each method is automatically created with a stub implementation, which returns the **NotImplementedException** for each action. Remove the exception and replace with the proper corresponding code and business logic.

```


Verify_ManagedAccount_Credentials_Action.cs
101 hosts = paramFactory.Get<HostByOptions_Value>();
102
103 (int) portNumber = paramFactory.Get<Port_Parameter>(paramFactory.DefaultPort);
104 int timeout = paramFactory.Get<Timeout_Parameter>(paramFactory.DefaultConnectionTimeout);
105
106 foreach (string host in hosts)
107 {
108     if (result.IsFinished || result.Result == EPluginActionResult.Success) break;
109     else if (result.IsFinished)
110     {
111         result = new PluginActionResult(result);
112     }
113
114     result.Dump = $"Verify_ManagedAccount_Credentials action on Custom Plugin system: Server={host} {portNumber}";
115
116     // Add your code here and remove NotImplementedException
117     throw new NotImplementedException("Missing implementation in the class Verify_ManagedAccount_Credentials_Action");
118
119 }
120
121 catch (ArgumentException ex)
122 {
123     result.Dump = $"Parameter error: {ex.Message}";
124     result.IsFinished = true;
125     EPluginActionErrorCode wrongParameters =
126     EPluginActionErrorCode.WrongParameters,
127     GlobalConstants.ActionFailedBeing;
128 }
129
130 }
  
```

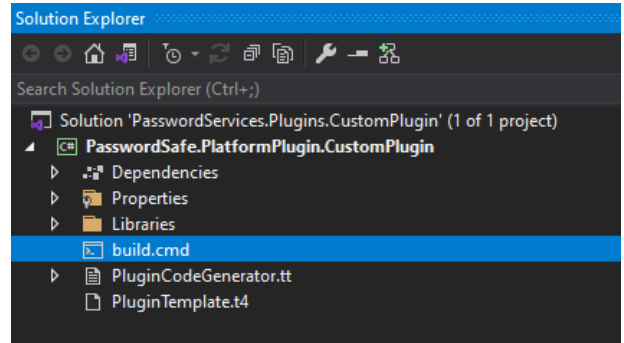
7. Each plugin action has access to several available parameters and is provided in each action as a code comment.

```

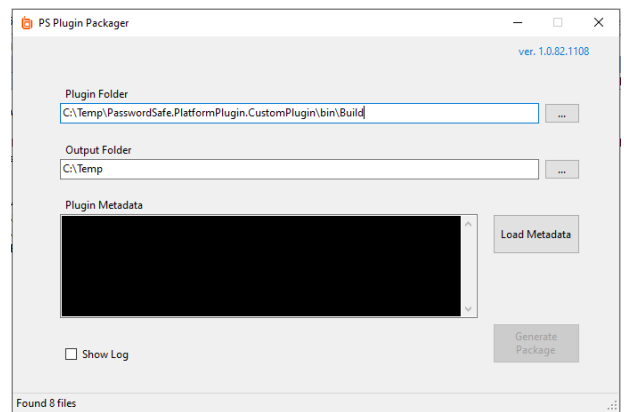
Verify_ManagedAccount_Credentials_Action.cs
PluginCodeGenerator.tt
63 public override PluginActionResult ExecuteAction(ActionParameters args)
64 {
65     var result = new PluginActionResult(args);
66
67     try
68     {
69         if (args.ActionType != SupportedActionType)
70         {
71             result.IsFinished = true;
72             EPluginActionErrorCode notSupported =
73             EPluginActionErrorCode.WrongParameters,
74             $"Action '{SupportedActionType}' does not process '{args.ActionType}'";
75             return result;
76         }
77     }
78     var paramFactory = new ParameterFactory(args);
79
80     // Document required parameters below
81     // --- Functional Account parameters
82     // string functionalAccountName = paramFactory.GetStringValue(ActionParameter.FunctionalAccount_AccountName_Only, true);
83     // string currentPassword = paramFactory.GetStringValue(ActionParameter.FunctionalAccount_CredentialsCurrent_Password);
84     // string newPassword = paramFactory.GetStringValue(ActionParameter.FunctionalAccount_CredentialsNew_Password, true);
85     // --- Managed Account parameters
86     // string managedAccountName = paramFactory.GetStringValue(ActionParameter.ManagedAccount_AccountName_Only, true);
87     // string currentPassword = paramFactory.GetStringValue(ActionParameter.ManagedAccount_CredentialsCurrent_Password);
88     // string newPassword = paramFactory.GetStringValue(ActionParameter.ManagedAccount_CredentialsNew_Password, true);
  
```

- Build the project and ensure there are no errors. (The **build.cmd** file provides an easy-to-use command line interface.)

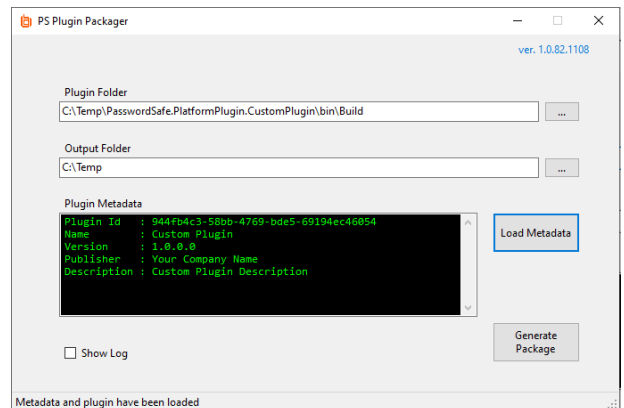
 **Note:** Refer to the *Plugin SDK* for a test harness utility, which allows the ability to test and debug your plugin prior to adding it to Password Safe.



- Once the plugin is ready to be used, it needs to be packaged into a Password Safe Plugin (PSPLUGIN). Extract the **Password Safe Plugin Packager.zip** from the Platform SDK and run the **Password SafePasswordSafe.Plugins.Packager.exe** application.
- Select the **Plugin Folder** (the build output location from Step 8), and specify the desired **Output** location.



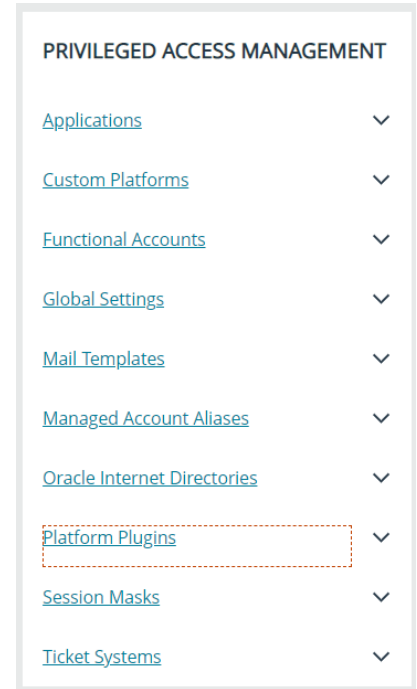
- To validate the plugin was created correctly, click the **Load Metadata** button. The plugin metadata values which were entered in the **PluginCodeGenerator.tt** file in Step 3 are displayed.



- Click the **Generate Package** button to generate the **plugin.psplugin** file.
- Add the new plugin into Password Safe via the **Configuration > Privileged Access Management > Platform Plugins** menu.



**Note:** Access to this configuration entry requires a *BeyondInsight Administrator's permission*.



PRIVILEGED ACCESS MANAGEMENT

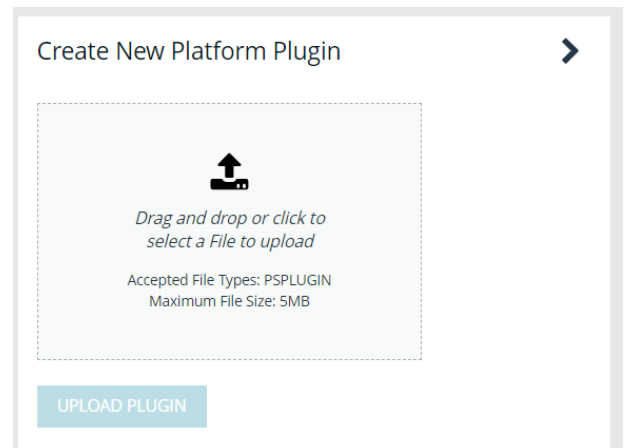
- [Applications](#) ✓
- [Custom Platforms](#) ✓
- [Functional Accounts](#) ✓
- [Global Settings](#) ✓
- [Mail Templates](#) ✓
- [Managed Account Aliases](#) ✓
- [Oracle Internet Directories](#) ✓
- [Platform Plugins](#) ✓
- [Session Masks](#) ✓
- [Ticket Systems](#) ✓

- Click **Create New Platform Plugin**. Either drag and drop the file, or click the **Upload** tile to browse to and select a file to upload.




**Note:** The maximum supported file size for a plugin package is 5 MB.

- Click **Upload Plugin**. The plugin package is added to Password Safe with the ability to create managed systems, managed accounts, and functional accounts for this new platform.



Create New Platform Plugin >



Drag and drop or click to select a File to upload

Accepted File Types: PSPLUGIN  
Maximum File Size: 5MB

UPLOAD PLUGIN