AO Labs

RESEARCH PAPER

SALESFORCE SECURITY

Understanding Salesforce Flows and Common Security Risks

By: Aaron Costello, Offensive Security Engineer at AppOmni

CONTENTS

INTRODUCTION

Organizations are seeking to simplify the development process and provide business users the ability to create applications themselves, which means the demand for Low Code tools are on the rise. There are many commonalities between most Low Code solutions: a visual interface that leverages drag and drop functionality in combination with model-driven logic, allowing for powerful process automation for both customer and employee experiences.

Salesforce's Flow Builder is no exception. Built on the familiar Lightning Platform, the Flow Builder achieves end-to-end process automation by leveraging reusable components known as Flow Actions. These action elements can be either partner-provided, out-of-the-box, or even powered by custom Apex code that allows for development teams to support administrators and no-code users.

However, that powerful functionality and customization means Flows have become another item in the registry of Salesforce features, like Lightning Components and Apex, that security teams must review. While it's easier for security teams to audit a single Flow rather than disjointed Lightning Components sprawled across various pages of a Digital Experience, swapping out Lightning components for Flows does not eliminate all risk. This article will discuss the security nuances unique to Flow development as well as permission management pitfalls, in addition to how to combat those pitfalls.

AN OVERVIEW OF SALESFORCE FLOWS

Before reading this article, it's best to have a basic understanding of Salesforce's Flow Builder and the access-granting elements within the Platform. The official documentation thoroughly covers the basic concepts and this article will focus on several concepts, including:

- **Screen Flow:** The most common type of Flow that takes the format of an interactive screen that's typically called through an action or button.
- **Flow Element:** These represent actions that are executed in the Flow. They are the "building blocks" of your Flow and provide the functionality for your processes.
  - SubFlow Element: This is a Flow which has been embedded in another Flow. Throughout this article, the parent Flow is referred to as the "Master" Flow.
  - Decision Element: This evaluates the provided conditions to determine which path in the Flow is taken next.
- **Flow Resource:** A representation of a value that can be referred to within the Flow.
  - Variable Resource: As described, a variable that can store a value and be referred to or modified throughout a Flow.
  - Formula Resource: Allows the storing of Global Variables that reference different pieces of information, including information about the user executing the Flow, organization details, and information about the current Flow session, etc.

Nothing will solidify an understanding of these concepts like practice, so it's highly recommended that readers modify, run, and debug the sample Flows provided by Salesforce within a developer instance.

FLOW PERMISSIONS

The ability for a specific user to execute Flows can be determined by a number of permission-granting elements. These include:

- System Permissions granted via a Profile or Permission Set:
  - The “Manage Flows” permission grants the ability to create, modify, execute, and delete Flows within the platform. This permission is granted sparingly as it is considered to be a sensitive admin permission.
  - The “Run Flows” permission allows for the execution of (nearly) any Flow that is active on the system. This can be assigned to anyone, including basic community portal users or even the Guest profile of a community. The possibility of a low-privileged user having this permission is one source of risk.
- The “Flow User” license can be granted to specific Users. This is equivalent to Run Flows except it occurs on the User level.
- Granular access can be granted by enabling the **“Override default behavior and restrict access to enabled profiles or permission sets”** setting on a particular Flow and allowing specific Profiles to execute it. This will override default access controls so even profiles with “Run Flows” or users with “Flow User” cannot access the Flow unless they were explicitly assigned through this method.

The “Manage Flows” permission should be considered a sensitive administrative permission as it effectively grants the ability to view and change all data and resources within the system. In addition, the unexpectedly powerful impact to access control of a feature license flag is uncommon in Salesforce, because manual management of individual user licenses is a more difficult task than management of permissions granted through a Profile or Permission set.

This brings us to the most potentially problematic configuration of the bunch: “Run Flows”. The allure of easing the pains of permission management caused by granular access is one that many Salesforce administrators cannot resist. Unfortunately, since Salesforce is a large and powerful platform, critical permission-granting elements for newer features such as Flows are often poorly understood. The caveat with “Run Flows,” in particular, is that handing over a master key to execute all Flows that do not have default behavior overridden can lead to critical security risks. Those risks include potentially granting users access to sensitive information or worse, granting access to modify information that they normally are unable to interact with. Naturally, these vulnerabilities are dependent on the functionality that a Flow seeks to perform. That being said, it is a fair assumption that if an organization is utilizing Flows to provide capabilities for their customers, such as the ability to create a case for product feedback, the organization may also be utilizing Flows to perform administrative actions. The fact that even Guest Users can be accidentally granted this permission drastically heightens the importance of strong permission management practices.

FLOW EXECUTION PROCESS

Like any visual UI element on the Lightning Platform, there is an underlying API call made that provides interaction with the backend "under the hood". Flows are no exception and Screen Flows are a prime example of this pattern. Even though an administrative Flow may not be embedded in the UI of a Digital Experience site, built-in Apex controllers can be utilized to interact with the Flow in the same manner that clicking through a Screen element does.

Below is an example of how the Flow execution process works behind the scenes via the Aura API:

1. When a Flow commences on a Lightning Community, the following Apex method is executed by the Flow engine:

serviceComponent://ui.interaction.runtime.components.controllers.FlowRuntimeController/ACTION$runInterview

This method takes the developer-defined API name of the Flow, along with additional arguments that may be required by the first element. This API name is a field with the same name returned by multiple standard Apex controllers when either of these steps are performed:

1.1 Directly querying the FlowDefinitionView sObject using known Apex controllers that have the ability to read record data will immediately return the "ApiName". This is dependent on the calling user, as even Guest Users with the "Run Flows" permission do not have visibility into this field.

1.2 Retrieve List View ID for "All_Flows" through either of the below methods with their required arguments. Note that the platform may return a different List View for (a) if a new default List View was configured.

a. java://ui.processhome.components.Flow.listview.FlowListViewController/ACTION$getDefaultListView

b. serviceComponent://ui.force.components.controllers.lists.listViewManager.ListViewManagerController ACTION$getMetadataInitialLoad

The returned List View ID can be passed to this method and the "ApiName" field contains the Flow API name:

serviceComponent://ui.force.components.controllers.lists.listViewDataManager.ListViewDataManagerController/ ACTION$getItems

As shown, the APIName field is not a secret as only authorized users may read the records of the FlowDefinitionView sObject.

The HTTP response will also provide the input parameters for the current Flow element. User input fields can be seen in the fields array and have a "fieldType" value of "INPUT". The name field of the input is the parameter name to be passed when executing the Flow. The "dataType" field dictates whether it is a string, record ID, integer, or other variable.

2. To proceed to the next stage of the Flow, the engine executes the method below. It automatically populates the serializedEncodedState value and any input variables that were provided during interaction.

serviceComponent://ui.interaction.runtime.components.controllers.FlowRuntimeController/ACTION$executeAction

The above steps demonstrate the possibility of interacting with a Flow purely through the API, and offers insight into what goes on "under the hood" of a Flow. I've referred to the term "administrative Flow" several times. The next section will outline what makes a Flow "administrative" in nature and, as a result, far more powerful.

FLOW EXECUTION CONTEXTS

Similar to Apex code, Flows have their own individual settings that dictate the level of permission with which the Flow actions execute. When a Flow is accessed via Flow Builder by a System Administrator or someone with the "Manage Flows" permission, the context in which the Flow runs can be modified. The options are:

1. User or System Context – Depending how the Flow is launched
2. System context (with sharing) to enforce Record Level Access
3. System context without sharing

Options (2) and (3) are the most concerning, as they could potentially allow a user to perform actions above their own privilege level. If none of the above options are selected, these are the default contexts for different execution sources as seen in the official documentation:

Launch Method	Default Context
Apex	Depends on Code
Experience Cloud Site	User
Embedded as a visual component inside a custom Aura component	User
Embedded as a visual component inside a Visualforce page	User
Custom Button	User
Custom Link	User
Direct Link	User
Flow Action	User
Lightning Page	User
Process Builder	System
REST API	User
Run from an Apex method of a custom Aura component controller	Depends on Code
Run from an Apex method of a Visualforce controller	Depends on Code
Scheduled Flow	System
Web Tab	User

If the context depends on code, Apex uses the "with sharing" and "without sharing" keywords to specify whether to enforce organization-wide default settings, role hierarchies, sharing rules, manual sharing, teams, and territories. A Flow called by Apex always ignores object and field-level access permissions. If Process Builder is the distribution method, the user who triggered the process sometimes requires other permissions. For example, if a process launches a Flow that attempts to save permission-set license assignments, and the running user doesn't have the "Assign Permission Sets" permission, an error occurs.

In the case of a Master Flow and a SubFlow, the execution context will depend on how the Flow is triggered:

- A master Flow running in system context will cause actions run in the SubFlow to be run in system context as well, regardless of whether the SubFlow was originally created and configured to run in user context.
- A master Flow running in user context that has a SubFlow running in system context will proceed to run the actions in the SubFlow in system context.

Keep in mind that a SubFlow is also technically a standalone Flow. As a result, it can be triggered directly using "$runInterview" separate from the main/master Flow. In these cases, execution context depends on how this SubFlow is configured to run on its own. So, whilst Sub Flows are a perfect candidate for following the DRY principle, they introduce additional security overhead which will be discussed in detail within "The SubFlow Conundrum" section.

The table of defaults clearly demonstrates that Salesforce has taken a secure-by-default approach for the most common implementations of Flows, such as those on Experience Cloud Sites and Lightning Pages, by enforcing User context by default. This can, however, lead to a false sense of security by convincing an Administrator that by not manually implementing a Flow on a publicly-facing interface such as a Site, it cannot be reached. That increases the likelihood that the Administrator would opt in for the Flow to run with administrative privileges (without sharing). Nonetheless, the previous section demonstrated that with the right permissions, these "hidden" Flows are just as reachable through Aura.

FLOW SECURITY CONSIDERATIONS AND HOW TO ADDRESS THEM

Lock Down Powerful Flows

Ideally, Flows should always run in the context of the executing user. That being said, there may be cases where it is desirable to execute a Flow without enforcing Sharing Rules. In this case, the Flow in question should have the "Override default behavior and restrict access to enabled profiles or permission sets" option enabled, and subsequently only whitelisting existing high-privileged profiles such as the System Administrator.

Also, external users and profiles such as the Community Portal profile and Guest User profile should not be granted "Run Flows" or "Flow User" permissions but rather individual/granular access to the Flows they need. The reasoning behind this is that even if all powerful Flows have Profile whitelisting enabled, it's possible that a new Flow that will run in system context will be created in the future. Perhaps the creator of this Flow forgets to enable whitelisting, leaving it exposed to external actors with these skeleton key permissions.

Avoid Client-Side Validation

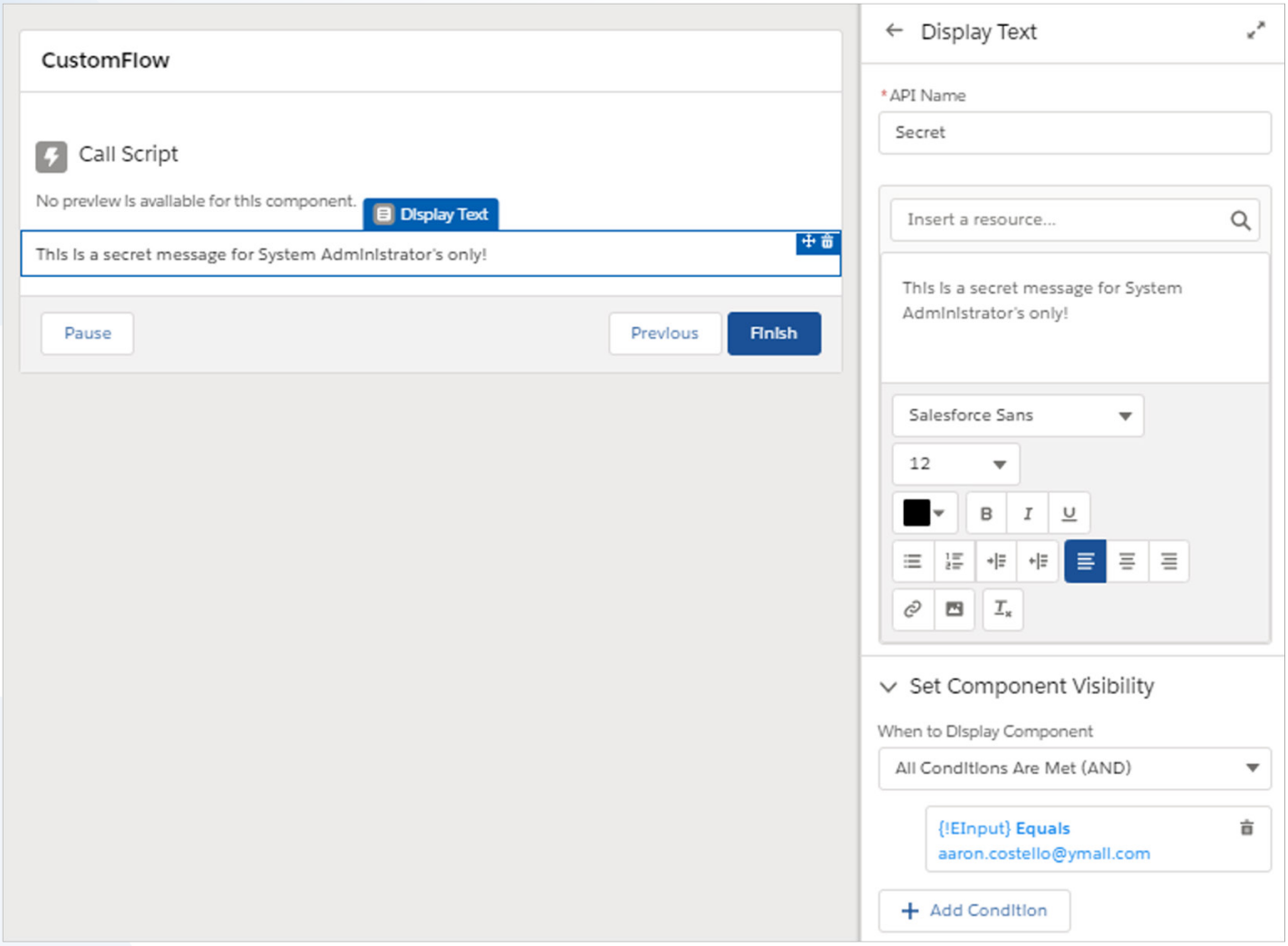
When using Screen elements, avoid performing client-side validation (seen in Flows as "Input Validation") to check if user input is equal to a stored variable in the Flow. This is due to the fact that a malicious user can see the value of the stored variable, and can then immediately insert the correct answer that the element is looking for. Here is an example seen in the response when interacting with a screen item:


```
"contextMap":[
   {
      "flow.Birthdate_Variable":"1956-08-15"
   }
],
```


The expected answer is leaked in the response.

Do Not Rely on Component Visibility

Screen elements have the ability to show / hide inputs or text depending on the outcome of any conditions within their "Set Component Visibility" option. For example, a Screen element might show extra fields if an email entered was that of an administrator, as seen below:



A secret message configured to be displayed when the value of the user supplied "EInput" variable is equal to a specific email address.

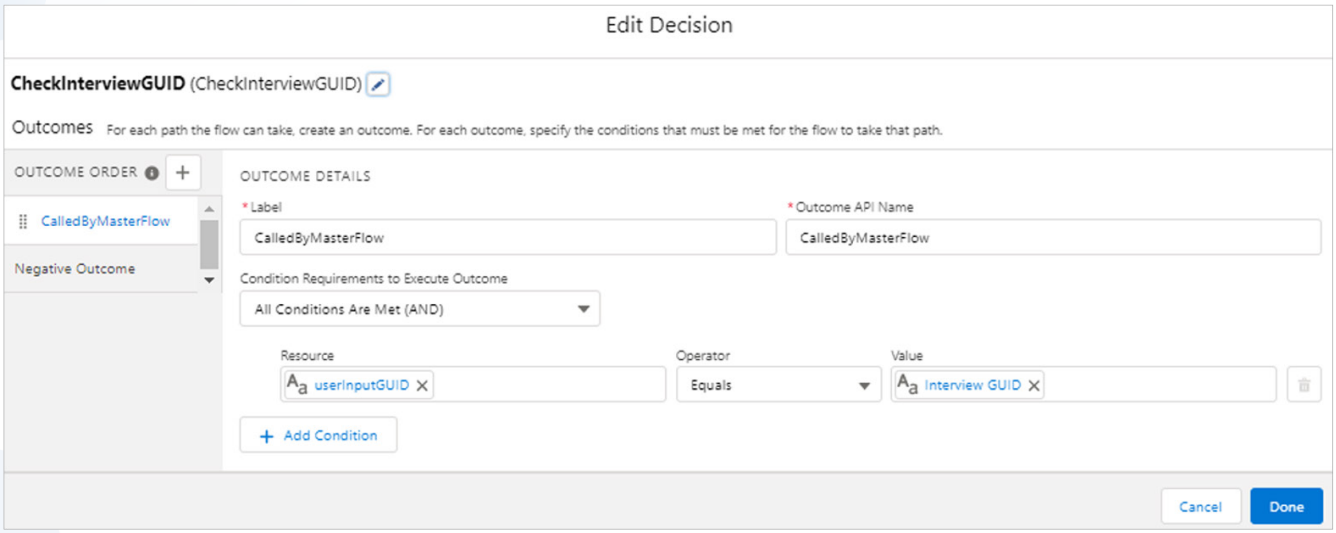
The caveat of "Set Component Visibility" is that the value within the "Display Text" component is only hidden from the UI. The secret message is exposed in the HTTP response regardless of the email entered:


```
{
  "outputs":null,
  "isRequired":false,
  "helpText":null,
  "metadataValues":null,
  "inputs":null,
  "dataType":"STRING",
  "errorMessage":null,
  "label":"<p>This is a secret message for System Administrator&#39;s only!<\/p>",
  "name":"Secret",
  "triggersUpdate":false,
  "choices":null,
  "contextMap":[
    {
      "flow.EInput":"ss@ymail.com"
    }
  ],
```


The secret message is visible within the HTTP response, even though the value of "EInput" did not satisfy the condition.

It's always recommended to separate administrative and user functionality into separate Flows. However, it is possible for both to exist side by side within the same Flow. To avoid the above issue, follow these steps:

1. Create a new Resource within the Manager section of the Toolbox.
2. Set the "Resource Type" to Formula, and insert "{!$Profile.Name}" within the Formula text box.
3. Insert a Decision element into the Flow, just before you'd like the functionality of the Flow to change depending on the running user (System Administrator vs any other profile).
4. Ensure that the Decision element is configured as follows. Note that the default outcome (failure) of the condition is for non-System Administrators:



If the running User's profile is "System Administrator," then the "isAdmin" path is followed to the Administrator Screen. Additional conditions and outcomes can be configured for more than one single profile if required.

Utilising this configuration allows for a different route to be automatically followed by high-privileged users, without the need to create an additional separate Flow.

Implement Access Control Checks within Apex Actions

In the event that Flow Builder does not have a built-in element or action that supports your business needs, Salesforce provides the ability to define Apex classes that can be called from an action within a Flow. These are known as "Apex Actions," an action element that "invokes" a single Apex method defined with the "Invocable Method" annotation, allowing for dynamic inputs and outputs to and from the Flow. Notably when an Apex Action is executed from a Flow, it occurs on the server-side. As a result an individual is unable to see the input values, name, or even point of execution for an Apex Action during the typical Flow execution process.

Apex actions do not adhere to the same rules as standard Flow Actions such as "Get Records," which are run in the context as defined by the individual Flow's settings. Instead, they operate in the context of how they were defined via the sharing keywords, like regular Apex. In addition to considering whether sharing rules should / should not be enforced, the developer of the Invocable Method must make certain that both object and field-level permission checks are implemented for any SOSL/SOQL/DML operations being performed.

Unlike regular Apex controllers, the most notable feature of Invocable Methods is their availability to be listed and executed through the REST API, entirely separate from the Flow. To put this into context, it may be helpful to summarize how Apex actions have evolved with the Salesforce Platform over time. Prior to the Summer '19 release, any user with the ability to run a Flow could inherently execute Apex methods that may be embedded within, without needing to be assigned access to the Apex class via Profile or Permission Sets. The release changed this behavior, making it so that a user must explicitly be granted access to an Apex class if it has Invocable Methods used within the Flow they wish to run.

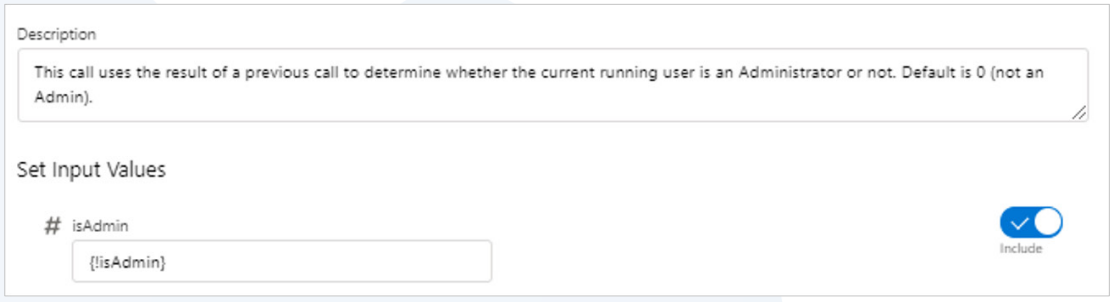
This modification has resulted in potential security implications, including:

- Since Apex access is granted by class and not individual method, accidental exposure of sensitive AuraEnabled methods to unauthorized users can occur if the Invocable Method resides within the same class.
- Any "API Enabled" user may send a GET request to specific REST API endpoints to list all Apex Actions that are available to them, in conjunction with each Action's required parameters and expected types. Using this information, a POST request with arbitrary data may be crafted and sent to the individual Action over the API, with the output returned in the HTTP response.


```
{
  "allowsTransactionControl":true,
  "category":"Account",
  "configurationEditor":null,
  "description":"Returns the list of account names corresponding to the specified account IDs.",
  "fillColor":null,
  "genericTypes":[
  ],
  "hasCallout":false,
  "iconId":null,
  "iconName":null,
  "inputs":[
    {
      "apexClass":null,
      "byteLength":0,
      "configuration":false,
      "description":null,
      "implicitType":null,
      "label":"ids",
      "maxOccurs":1,
      "name":"ids",
      "picklistValues":null,
      "required":false,
      "sobjectType":null,
      "toolingType":null,
      "type":"ID"
    }
  ],
  "label":"Get Account Names",
  "name":"AccountQueryAction",
  "outputs":[
    {
      "apexClass":null,
      "description":null,
      "label":"output",
      "maxOccurs":1,
      "name":"output",
      "picklistValues":null,
      "sobjectType":null,
      "type":"STRING"
    }
```


An example of the details returned for a specific Apex Action via a GET request to the Apex Action REST API endpoint.

This second point may be particularly concerning depending on your reliance on Apex Actions within Flows. There are plenty of cases in which hardcoded inputs such as flags, or Formula variables, are passed as input to these Actions and will affect either the behavior of or data returned from the Action. This is in contrast to the aforementioned process of when a Flow is the caller of an Action, not allowing for the tampering with or exposure of Apex Action inputs that are not taken directly from user input.



Example Apex Action utilizing a static value.

If a non-Administrator executes this Flow, the {!isAdmin} variable will be 0 and the value returned will be “User is not an Admin”. However, a malicious user could execute the Apex action via the REST API and tamper with the input:

```
{
  "inputs":[
    {
      "isAdmin":1
    }
  ]
}
```

```
16 [
     {
       "actionName":"AccountQueryAction",
       "errors":null,
       "isSuccess":true,
       "outputValues":{
         "output":"You are an Admin"
       }
     }
]
```

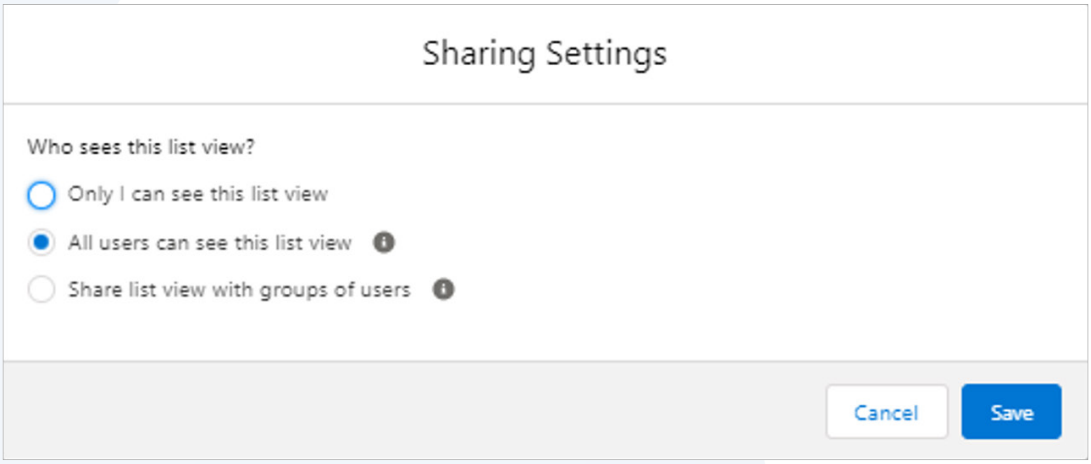
Querying the Action directly allowed for an arbitrary value to be passed to “isAdmin”.

Taking the above behavior into account, Salesforce administrators should take the following precautions when implementing Apex actions:

1. Restrict “API Enabled” access for external users such as those with the Community Portal profile. As mentioned, Flows that are organically run as intended do not require the running user to have this permission, as the Apex Action is executed on the server-side. If a Digital Experience leverages Lightning Components to interact with custom Apex, there is no need for REST API access (unless there is a need to access mobile apps).
2. Where possible, ensure that Apex classes that contain Invocable Methods do not contain other sensitive methods that are exposed via other avenues such as the Aura API.
3. The upcoming “Disable Rules for Enforcing Explicit Access to Apex Classes” update in the Spring ’22 release will revert to requiring a user to have explicit Apex class access in order to execute an Apex Action during a Flow. Once this update is applied, access that was granted to users for the sole purpose of being able to run a Flow successfully should be immediately revoked.

Restrict List View Visibility from External Users

If a user has the “Run Flows” or “Flow User” permission, they have the ability to view (and execute) all non-publicly facing Flows that have not been restricted to a whitelist of Profiles by leveraging built-in Apex controllers. This is due to the default visibility setting of the “All_Flows” List View within Salesforce, which returns the required data to execute any returned Flows.



The default “All_Flows” List View Sharing Settings as seen from the UI.

There is, however, a mitigating factor in the event that the “Run Flows” permission is assigned to Guest Users specifically. The Apex controllers that are commonly used by actors to enumerate record details will not reveal the “ApiName” field value (the name of the Flow) within the FlowDefinitionView object either when directly queried or via ListView-related controllers. So, while a Guest user may be able to execute administrative Flows, Salesforce has done an outstanding job to ensure there is no easy approach to enumerate them.

On the other hand, a Community Portal user with the same permission will be able to retrieve the names straight from the FlowDefinitionView object, or alternatively retrieve the names from the default “All_Flows” List View which, by default, is readable by all users.

To restrict visibility to active but non publicly-facing Flows, this setting should be further restricted to either “Only I can see this list view” or selecting “Share list view with groups of users”. By selecting the latter option, a custom group of users or the pre-existing “Public Group - All Internal Users” can be chosen to prevent Community Portal and Partner Portal users from having read access to this List View.

This setting may be changed via the following steps:

1. From Setup, search for “Flows” and select it.
2. Click the cogwheel icon on the upper-right hand corner of the table and select “Sharing Settings”.
3. Choose one of the aforementioned secure options and click “Save”.

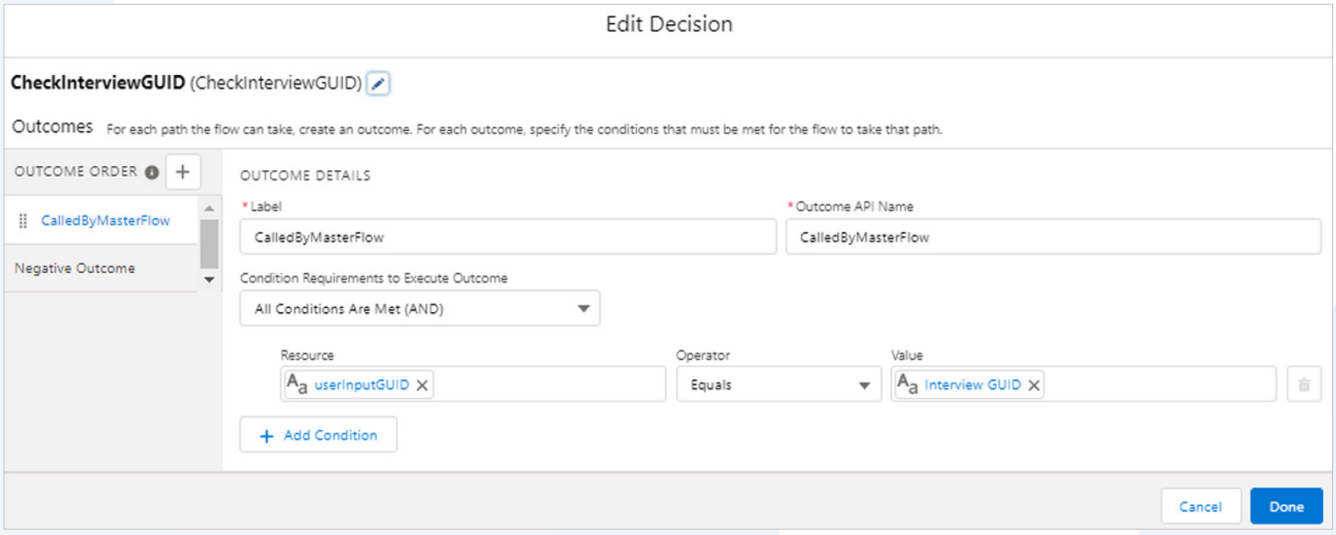
THE SUBFLOW CONUNDRUM

Organizations may wish to minimize element re-use by creating many Flows that cater to different needs, and these Flows are intended to be used as SubFlows only. So instead of a complex Master Flow containing fifty elements, these elements may split across multiple other Flows, grouped by functionality. This allows for commonly used actions to be reused across multiple Flows by embedding them as SubFlows. A common example is a “Return Records” Flow, which may perform a “GetRecords” action and display the resulting values to a Screen element. It is an attractive approach, but these implementations come with a few important security implications:

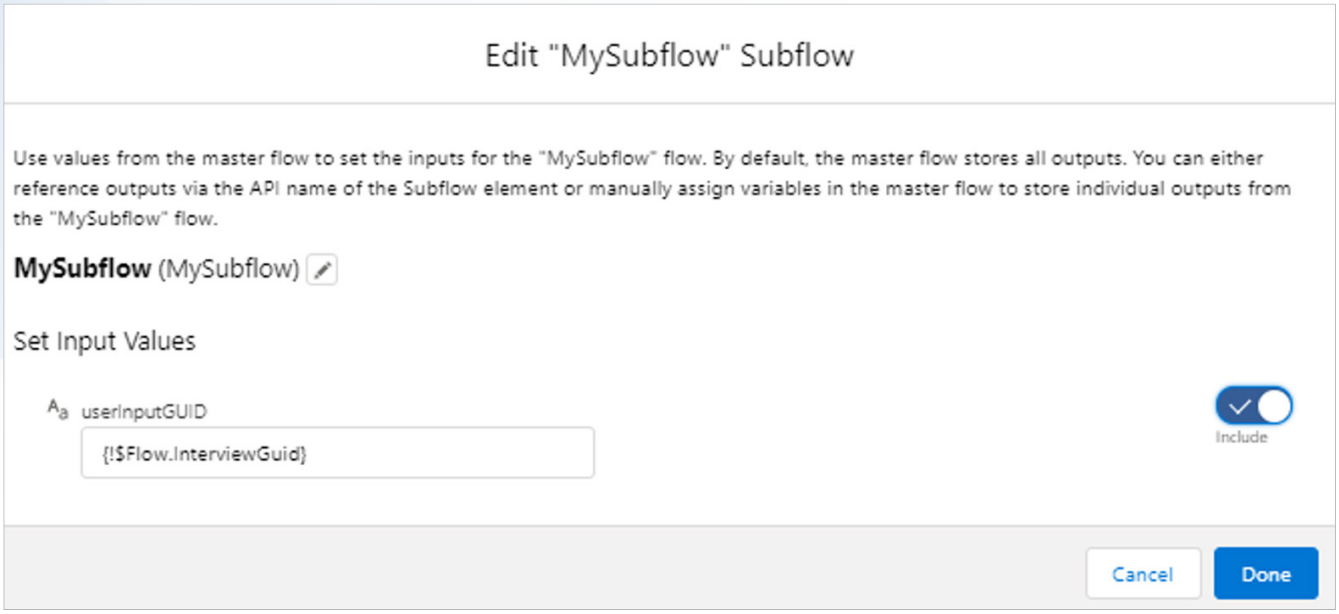
1. It is often forgotten that a Flow intended to only be executed as a SubFlow within a Master Flow is still a singular and standalone Flow itself. Thus, it can be invoked and executed separately, as demonstrated within the “Flow Discovery and Execution” section.
2. Administrators/Flow developers are now responsible for auditing the security of both the Master Flow and its SubFlows, instead of one large Flow. The primary check should be of the privileges Flows are configured to run with.

So how can teams ensure that a Flow can exist solely as a SubFlow and never be executed standalone? Fortunately, there is a way to implement an access check using nothing but native elements and Global variables. Here is a sample implementation guide, where there is a Master Flow with one SubFlow embedded:

1. Open the SubFlow in Flow Builder, and create a new Variable of type Text. The “Available for input” option should be selected.
2. Create a Decision element, comparing the previously created variable value to “{!$Flow.InterviewGuid}”. When the values are not equal, the user should be routed to an error Screen (and the Flow interview ended). Note that in the below screenshot, “{!$Flow.InterviewGuid}” was immediately re-labeled to “Interview GUID” once entered:

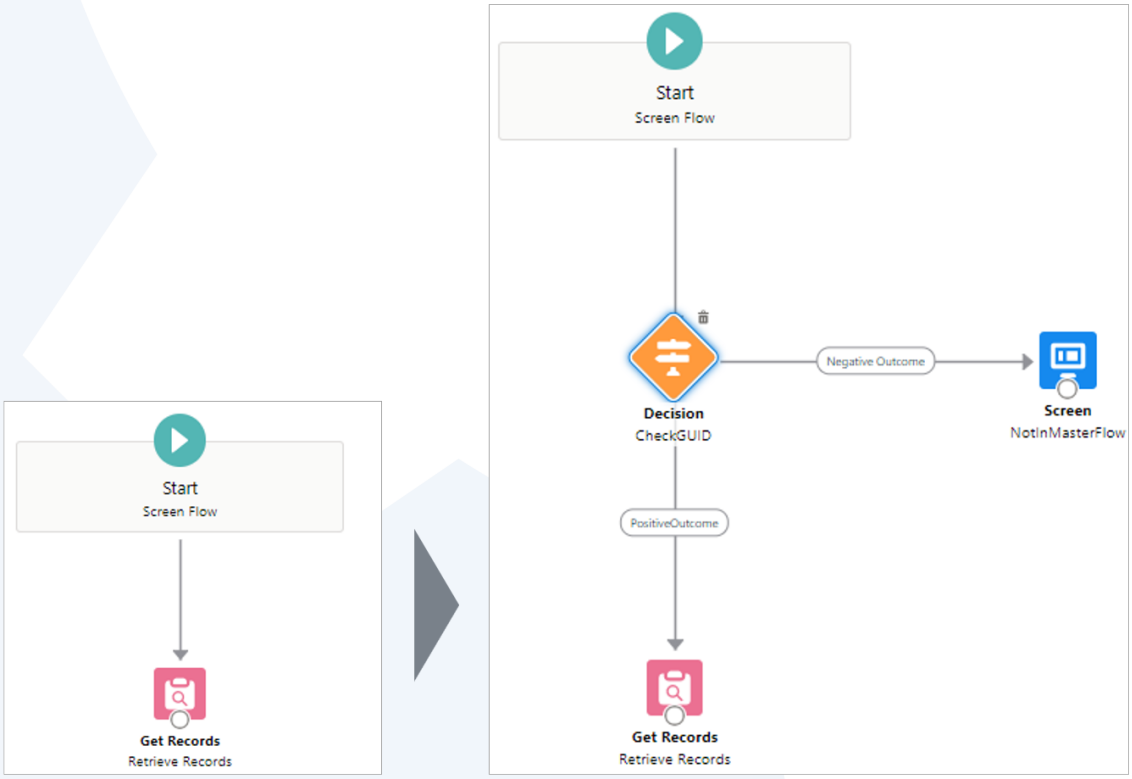


3. Save these changes, and re-activate the Flow if it was previously active.
4. Open the Master Flow in Flow Builder and select the SubFlow element. Under “Set Input Values,” the variable created in step (1) should appear. Click the slider, and enter “{!$Flow.InterviewGuid}” into the field:



5. Save these changes, and re-activate the Flow if it was previously active.

This configuration works because every individual running instance of a Flow (also known as Flow interview) has a unique GUID that is easily predictable. When a Flow is executed as a SubFlow, it inherits the Interview GUID from the parent due to the fact that they are seen as a single instance. In the above, the Master Flow passes its own GUID to the SubFlow. The SubFlow subsequently checks whether the GUID value received from the input variable is the same as its own, ensuring that it will only continue execution if executed by a parent Flow. A malicious user can attempt to directly invoke the SubFlow, but they are required to predict what the GUID of the SubFlow will be. This is not reliably predictable, and as a result serves as sufficient protection from direct invocation.



Subflow Before

Subflow After Implementing the GUID Check

Subflow before (left) and after (right) implementing the GUID check.

CONCLUSION

Low Code solutions such as Salesforce Flows are undoubtedly an attractive alternative to both business users and seasoned Apex developers alike. But developers must be aware that just because these pre-built Flow components and elements are presented in an enticing drag-and-drop format, that does not mean that Flows are exempt from the same types of security misconfigurations as their traditional development counterparts. Like any new feature, there are unique caveats that must be taken into account prior to deployment in a production environment.

If your team is looking to integrate Flows into your development process, here are some security questions to consider:

- Do any users have the “Manage Flows” or “Run Flows” permissions? How do you track and approve those assignments?
- Do any users have the Flow User license on their user account? How do you track and approve those assignments?
- How do you track which Flows are available to which user groups?
- Are all Flows set to override default access behaviors? How do you track and approve Flows that do not have default access overridden?
- Do you use SubFlows? Are all SubFlows safe for direct invocation? If not, how are those Flows protected against direct invocation?
- Do you use client-side validation for any Flows?
- Is Component Visibility used to hide any Flow stages? Are their hidden statuses intended to be a security control?
- How do you track Apex class assignment to groups of users?
- How do you monitor that the “All_Flows” list view is not exposed to all users?

At AppOmni, we thoroughly research and test these new features so that our customers don’t have to. As part of AppOmni Insights, our engine can surface risky Flow permission assignments and identify privileged Flows within your organization, allowing you to enjoy the Low Code alternative without the concern of security vulnerabilities.

To learn more, email us at **info@appomni.com** or visit **appomni.com**.

AppOmni is a leading provider of SaaS Security Management software. Its patented technology scans APIs, security controls, and configuration settings to compare the current state of enterprise SaaS deployments against best practices and business intent. AppOmni makes it easy for security and IT teams to protect and monitor their entire SaaS environment from each vendor to every end-user.

AppOmni