AO Labs

WHITE PAPER

SALESFOCRE SECURITY

Lightning Components: A Treatise on Apex Security from an External Perspective

By: Aaron Costello, Offensive Security Engineer at AppOmni

CONTENTS

INTRODUCTION

The power of custom development on the Salesforce platform using their proprietary programming language, Apex, is undeniable. At the end of 2020, there were over 3400 applications live on the AppExchange. Furthermore, many Salesforce customers also utilise Apex to code their own in-house applications. Unfortunately, unlike AppExchange-listed applications, in-house developed Apex code does not always go through an intensive security review. To make matters worse, a large portion of custom Apex code is implemented within Lightning Communities in the form of controllers, and these controllers are often exposed to the public to provide additional functionality.

Observing custom Apex controllers being called while navigating around a Lightning Community is one thing, but being able to manually craft a call to one of these controllers is another. A common scenario is adding an Apex controller on a page, but mistakenly believing that only those Apex methods directly utilised by the component will be exposed. Unfortunately, this is not the case in many configurations as a user with the necessary permissions can see the properties of any “Aura Enabled” Apex method within the controller implemented within the component. This is regardless of whether or not they are intended to be used on the page.

This article will describe the architecture of Lightning Aura components, how a call to an Apex method with parameters crafted from nothing but the provided Javascript signature, and security best practices for using these components safely.

Tools such as Burp Suite or OWASP ZAP are recommended for analyzing HTTP requests and responses when performing this testing; however the built-in browser developer tools also suffice. In the steps of this article I will be using Burp Suite and occasionally the Firefox Developer Tools.

PREREQUISITE KNOWLEDGE

This article assumes basic knowledge of Javascript and Apex, however the following explanations and links may be helpful to read prior to continuing this article.

Lightning Aura Component - In this article we specifically talk about Lightning Components using the Aura component programming model. A component is a bundle (folder) of resources such as markup (XML), controllers and helpers (JS) etc, that are wired together.

Component Markup - The component markup file (.cmp suffix) contains metadata information about a particular component.

- Controller (Server Side) - From a server-side perspective, a controller is an Apex class that contains custom logic and data manipulation that can be used by a component / page. Apex controllers are called from the client-side from within the Controller JS / Helper JS via the Aura API.
- Controller JS (Client side) - We also refer to 'Controller JS' throughout this piece, which is the corresponding client-side controller for a component. It contains methods used to handle events within the component (and in some cases call the corresponding server-side Apex via the Aura API).
- Helper JS - Helpers are often defined (but not mandatory) and implemented when there is JS that you wish to reuse across action handlers. They are associated with utility functions. Typically, calls to the Aura API are made from the helper JS.

Useful Trailhead paths:

1. Get Started with Aura Components
2. Quick Start: Aura Components
3. Aura Component Tricks & Gotchas

LAB ENVIRONMENT

AppOmni Labs has provided a Salesforce Developer instance to be used as a complimentary learning tool when going through the sections of this article. Currently there are four levels. The first of which is to assist with the Dissecting a Lightning Aura Component section. Once the reader has completed this first challenge and solidified their understanding of dissecting Lightning Aura components, they have the knowledge necessary to interact with components implemented in levels 2,3, and 4 of the lab.

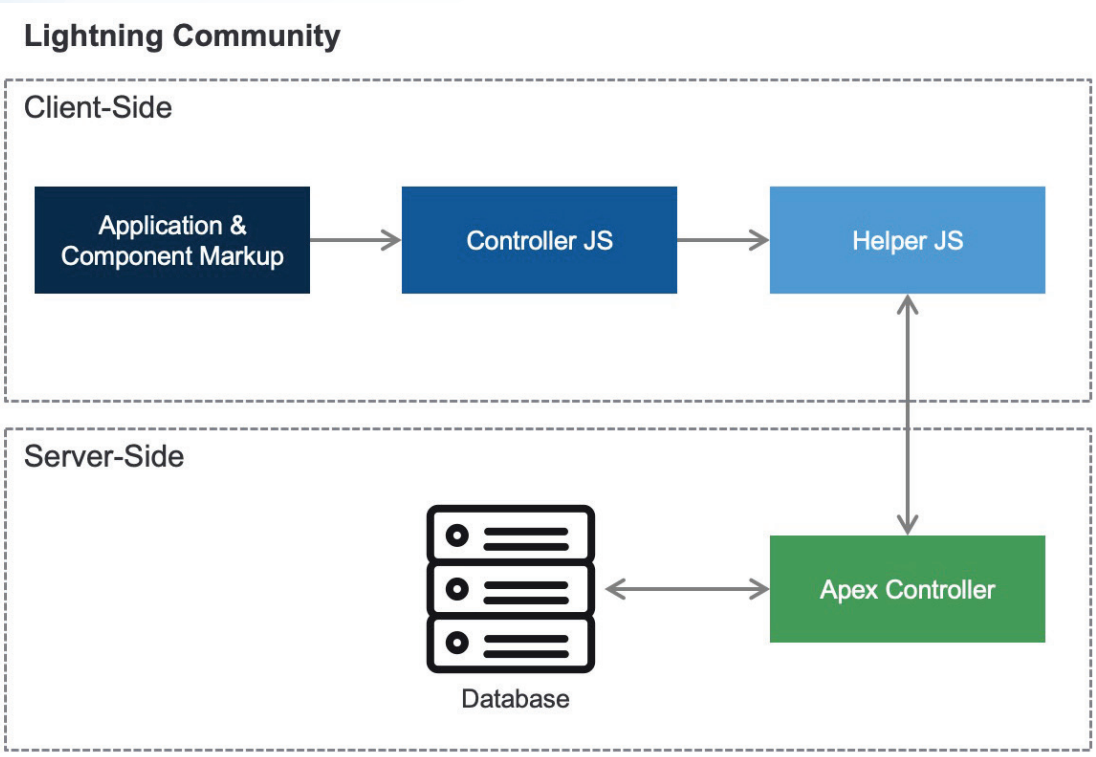
Below is a list of subsections in the article and their corresponding lab level:

Article Section	Lab Level
Dissecting a Lightning Aura Component	Level 1 - Dissecting a Component
Missing OLS / FLS security and insecure sharing	Level 2 - Missing CRUD Controls
SOQL Injection	Level 3 - SOQL Injection
SOQL Injection	Level 4 - Blind SOQL Injection

As further articles relating to Salesforce Lightning communities are released, we hope to add additional challenges to the Lab instance.

LIGHTNING (AURA) COMPONENT OVERVIEW

There are several ways in which developers can bring custom functionality to their Salesforce Lightning communities, the most



A diagram depicting communication flow for an Lightning Aura component that utilises an Apex class.

common of which are Lightning Aura components. These types of components are associated with the Lightning Framework, which is built on top of the open source Aura framework. When analyzing components throughout this article, you will notice that each has been built using the Aura component programming model as opposed to the newer LWC (Lightning Web Component) model.

When pulling apart components from an external perspective, we are most concerned about:

1. Retrieving the markup descriptor of the component
2. Analyzing the @AuraEnabled Apex methods that exist within the Apex Class implemented by a component
3. Understanding the parameters and return types of these Apex methods

In the following section I will go through this process with a sample component that was added to the AppOmni Lab instance. By following the steps outlined, you will be able to complete level 1 of the lab.

DISSECTING A LIGHTNING AURA COMPONENT

Prior to starting to read this section of the article in conjunction with attempting Lab level 1, it's necessary to understand that there will be several steps throughout the following subsections which are not technically necessary when applied to the Labs environment. There are instances where I discuss crafting a HTTP POST payload for a specific purpose, even though the payload is automatically sent for you during an action such as a link being clicked within the environment.

While it may save time to let the webserver populate these requests for you by simply loading a Lightning page, the purpose of copying values and manually crafting requests is to provide a step-by-step understanding of components and base controllers within the framework. In doing so you will establish a level of comprehension that will allow you to confidently interact with custom components on a Lightning page, without even looking at the page within your browser.

DETERMINING THE APPLICATION

First and foremost is determining in which Lightning Aura application the component / page on the site resides. Fortunately, there are definitive differences between how these Lightning applications are accessed which can be used to figure this out.

When a Salesforce Lightning Community is built using 'Experience Builder', the application in use is 'communityApp' within the 'siteforce' namespace. Assuming a custom domain was not used, they are accessed at a URL like the following:

`https://<domain_prefix>.<instance_location>.force.com/<directory_prefix>/s/`

In contrast, if a custom Lightning Aura application was created and made publicly available, it is accessed at a URL that takes the following format:

`https://<domain_prefix>.<instance_location>.force.com/<namespace>/<appname>.app`

The first 'Experience Builder' approach is more common and less code-heavy, but both can be utilised to take advantage of the features that the Lightning Aura framework offers. If using the above path examples isn't satisfactory for you to differentiate between the two, the source of a Lightning page on the site will reveal the Lightning application upon which that the page was built:


```
},"host":"","auraCmpDefBaseURI":"/auraCmp
  "mode":"PROD","app":"c:SampleApp","path
    "APPLICATION@markup://c:SampleApp":"M
  },"globalValueProviders":[{
    "type":"$SObjectType","values":{
      "CurrentUser":{
        "isChatterEnabled":false
```


This page exists on the SampleApp Lightning application in the default 'c' namespace.


```
},"host":"/s/sfsites","auraCmpDefBaseURI":"/auraCmp
  "mode":"PROD","app":"siteforce:communityApp","fwu
    "APPLICATION@markup://siteforce:communityApp":"
  },"apce":1,"apck":"FwCzpplvbdTZBKUbwy8ozg","mlr":
    "lightning":"interop"
  },"lv":"51","csp":1,"uad":0
```


This page exists on the communityApp application in the 'siteforce' namespace, it was likely created using Experience Builder.

We can see that the suffix of the URL on the Lab environment is '/s/', and each lab page is located within this directory. This way we know that it exists in the siteforce:communityApp Lightning application.

Retrieving the markup descriptor of custom Components

Navigating to the Lab homepage `/s/` with Burp Suite logging requests in the background, an endpoint of the following format is loaded:

```
/s/sfsites/l/...<long_string>../bootstrap.js?aura.attributes=...
```

This file is loaded by default within Lightning communities, and contains information about the pages that are contained within them. In relation to communities, this endpoint and its parameters are always included within the last script tag on the last line of the homepage `/s/`. In the event that this page is not loaded for some reason, you can manually take the endpoint from there and navigate to it yourself.

To find custom pages that were created and added to the community, search for the string '__c' within the response of the above loaded endpoint. In doing so in the lab, we can see the API name for the Lab level one page:


```
},
"/lightning-components-level-1":{
  "dev_name":"Secret_Component__c","cache_minutes":"30","themeLayoutType":"Inner","route_uddid":"0I30900000cyVRI","
},
```


Copy the following attribute values within the "/lightning-components-level-1" key, they will be useful soon:

- id
- event
- themeLayoutType
- view_uuid

In addition, search for the strings 'publishedChangelistNum' and 'brandingSetId' within the entire response body, and copy them also.

Open the Level 1 page (/lightning-components-level-1) in your browser. Open Burp Suite's proxy history and find any POST request to the `/s/sfsites/aura` endpoint. Remove the `message` POST parameter value and we'll replace it with our own payload below. Replace the placeholder values with the values copied in the above step.

———

```
{"actions":[{"descriptor":"serviceComponent://ui.comm.runtime.components.aura.
components.siteforce.controller.PubliclyCacheableComponentLoaderController/
ACTION$getPageComponent","callingDescriptor":"UNKNOWN","params":{"attributes":{"viewId":"<id_
value>","routeType":"<event_value>","themeLayoutType":"<themeLayoutType_
value>","params":{"viewid":"<view_uuid_value>","view_uddid":"","entity_name":"","audience_
name":"","picasso_id":"","routeId":""},"hasAttrVaringCmps":false,"pageLoadType":"STANDARD_
PAGE_CONTENT","includeLayout":true},"publishedChangelistNum":<publishedChangelistNum_
value>,"brandingSetId":"<brandingSetId_value>"}}]}
```

———

The purpose of the `getPageComponent` method of the `PubliclyCacheableComponentLoaderController` is to retrieve components that exist on the page, both built-in and custom.

When the above HTTP request is sent, you can begin searching for customizations. But prior to doing so, it's good to make note that each component is part of a namespace. An example of default namespaces being `aura`, `ui` and `force`. Custom components that are created and added to a community will either use the namespace chosen by the Salesforce administrator, the namespace of a package, or simply use the `c` namespace which refers to the default namespace.

To make the search for custom Apex easier, I recommend looking for the string `"descriptor":"apex://`, as built-in controllers will typically use the 'serviceComponent://' prefix instead.

Here is a snippet from the response body in relation to the Level 1 page:


```
],
"componentDefs":[
  {
    "xs":"P",
    "descriptor":"markup://c:basicComponent",
    "cd":{
      "descriptor":"compound://c.basicComponent",
      "ac":[
        {
          "n":"getFlag",
          "descriptor":"apex://level1Controller/ACTION$getFlag",
          "at":"SERVER",
          "rt":"apex://String",
          "pa":[
            {
              "name":"myMap",
              "type":"apex://Map<String,Boolean>"
            }
          ]
        }
      ]
    },
```


In the above JSON block, we can see that the markup reference string for this component is "markup://c:basicComponent". Notice how the `c` namespace is used, which would indicate that this Salesforce environment does not have a custom namespace set, and also that this component is not part of a managed package.

Technically, you could stop at this step and retrieve all of the information you need from this response. But it's incredibly messy as not only are there a large number of built-in components added to the page but, in larger communities, there is also the likelihood of other custom components added to the page. It can get confusing quickly and be a pain to work with, so I'll describe what I have found to be the cleanest method that can be consistently used to work from this point.

Cleanly working with component definitions

Once the markup reference string ("descriptor attribute") has been retrieved, there are a number of ways in which the attributes of a component can be analyzed, such as via the `/auraCmpDef` endpoint. However, Salesforce has also supplied a built-in controller as part of the Aura framework that removes the need to hunt for the values that the `/auraCmpDef` endpoint requires.

The 'ComponentController' is a Java class which has been defined for use as a serviceComponent, and it provides a 'getComponentDef' action which is @AuraEnabled, so we can call it through the Aura API.

Underneath the hood, here's how it works:

1. It takes a "namespace:component" string via the "name" parameter
2. It retrieves the component descriptor by passing the supplied input to definitionService.getDefDescriptor(), with an additional argument of "ComponentDef.class", the purpose of which is to detail the interface of the type of definition we want to describe.
3. Passes the component descriptor to definitionService.getDefinition() which then gets the named definition. The attributes of the component are returned in the output as serialized JSON.

Modify the `message` POST parameter with the below value, to analyze 'basicComponent', and send it:


```
{"actions":[{"id":"76;a","descriptor":"aura://ComponentController/
ACTION$getComponentDef","callingDescriptor":"UNKNOWN","params":{"name":"c:basicComponent"}}]}
```

We now have a response containing only the information relevant to this one specific component:


```
{
  "actions":[
    {
      "id":"76;a",
      "state":"SUCCESS",
      "returnValue":{
        "xs":"P",
        "descriptor":"markup://c:basicComponent",
        "cd":{
          "descriptor":"compound://c.basicComponent",
          "ac":[
            {
              "n":"getFlag",
              "descriptor":"apex://level1Controller/ACTION$getFlag",
              "at":"SERVER",
              "rt":"apex://String",
              "pa":[
                {
                  "name":"myMap",
                  "type":"apex://Map<String,Boolean>"
                }
              ]
            }
          ]
        },
```


We can break down what's happening here by looking at the ApplicationKey enum defined in the source:

"ac" : The actions accessible within this component.

- "n" : The name of the action
- "descriptor" : This is the string used to call the action (server-side Apex method) via the Aura API. The Apex class here is 'UserHandlerController' and has a method 'getDetails'.
- "rt" : The return type from the method, which in this case is a list of records from the User SObject (it could also return one single record).
- "pa" : These are the parameters taken by the Apex method. There is a single parameter called getDetails which takes a Map, where String is the data type of the keys and their values can be any variable type.

Another important attribute but not pictured is "cc". This is the lockerized component class, which contains the descriptor at the beginning and everything after "return $A.lockerService.createForDef(\n" is the controller and (optional) helper JS.

At this point it would appear we have absolutely everything we need to construct a valid request! But there's one detail that's missing, which is the fact that the 'getFlag' action takes a map as input but didn't specify the key name(s) in the parameter attribute. This is the reason for the explanation of the "cc" attribute above, as the controller or helper JS will contain this information.

Without a doubt, the Javascript provided in the attribute is awfully messy, and would require some cleaning to be completely legible. There are newline literals, and the escapeForJavascriptString() Java method that the Aura framework employs has also escaped single quotes, double quotes, and backslashes before sending the Javascript to the client.

One option is to clean up this JS, but an alternative is to view the JS in a beautified format that is referenced in the page source. In order to do this, follow these steps:

1. Open the page for the Level 1 challenge in your browser and open the Developer Console's debugger
2. Under 'Sources' open 's' -> 'components/c', and there will be a JS file with the same name as the component, in this case it's

'basicComponent.js'. Keep in mind that the 'c' parent directory is only used since this component is in the 'c' namespace.

Opening this file, we see the following:

basicComponent.js × aura_prod.js

```
return {
"meta":{
  "name":"c$basicComponent",
  "extends":"markup://aura:component"
},
"controller":{
  "myAction":function(component, event, helper) {

  var action = component.get('c.getFlag');
      const userInput = component.get('v.userInput')

      if (param) {

        action.setParams({
          myMap: {
              returnflag: userInput,
        }});

        const actionPromise = new Promise(function (resolve, reject) {
          action.setCallback(this, function (response) {
```

In the above, we can see that `component.get('c.getFlag)` is used to create a single use instance of the `getFlag` method. Further down, the `setParams` function is used to set data for the action. We now know that the Apex method is expecting a parameter called 'returnflag' within the 'myMap' map.

Calling the Apex method via Aura

The below payload is a skeleton that can be reused to call any custom apex.

```
{"actions":[{"id":"123;a","descriptor":"apex://<namespace>.<controller>/
ACTION$<action>","callingDescriptor":"UNKNOWN","params":{<parameters>}}]}
```

Changing the placeholders with the information we gained from the previous section, we arrive at the following final string:

```
{"actions":[{"id":"123;a","descriptor":"apex://level1Controller/
ACTION$getFlag","callingDescriptor":"UNKNOWN","params":{"myMap":{"returnflag":true}}}]}
```

You may have noticed that there is no namespace defined in the above payload, this is due to the fact that the default 'c' namespace is not required to prefix the controller class name. This is a blackbox test, but since the 'returnFlag' value is of type Boolean, we only need to request 'true' and 'false' to cover all possible outcomes.

Place the payload string into the `message` POST parameter of the request to `/s/sfsites/aura` we looked at from the `getComponentDef` action, and send it:

```
{"actions":[{"id":"123;a","descriptor":"apex://c.level1Controller/
ACTION$getFlag","callingDescriptor":"UNKNOWN","params":{"myMap":{"returnFlag":true}}}]}
```

It's worth mentioning that at this point you will have noticed other parameters in the POST body of the requests, such as `aura.context` and `aura.token` being the most important. The first contains information such as the Aura framework UID value, and the application name & namespace etc. The second most important piece of information here is the `aura.token` value. A value of `undefined` indicates that the request is sent from the privileged context of the `Guest User` profile since you are unauthenticated. If you were authenticated, this would be a JWT token instead of `undefined`.

Sending the POST request with the value ‘true’ will return the flag, completing the first level and cementing your understanding of dissecting custom components/apex!

```
16
17 message=
{"actions":[{"id":"81;a","descriptor":"apex://level1Controller/ACTI
ON$getFlag","callingDescriptor":"UNKNOWN","params":{"myMap":{"retur
nFlag":"true"}}}]}&aura.context=
%7B%22mode%22%3A%22PROD%22%2C%22fwuid%22%3A%22AE8981CB2KpCUerBipCwX
g%22%2C%22app%22%3A%22siteforce%3AcommunityApp%22%2C%22loaded%22%3A
%7B%22APPLICATION%40markup%3A%2F%2Fsiteforce%3AcommunityApp%22%3A%2
2sVZ6cnpPX79_SIZb4Z-5KQ%22%7D%2C%22dn%22%3A%5B%5D%2C%22globals%22%3
A%7B%7D%2C%22uad%22%3Afalse%7D&aura.pageURI=%2Fs%2F&aura.token=
undefined
```

```
17 {
     "actions":[
       {
         "id":"81;a",
         "state":"SUCCESS",
         "returnValue":"Congratulations! The flag is: {!Secret9823324}",
         "error":[
         ]
       }
     ],
     "context":{
```

COMMON APEX VULNERABILITIES

MISSING OBJECT / FIELD / RECORD LEVEL PERMISSION CHECKS

This class of vulnerability can be slotted comfortably in the same category of an IDOR for regular web applications, and if Salesforce had an OWASP Top 10 equivalent, then these missing security controls would be number 1. The issue arises when custom Apex is returning information to the calling user, typically as a result of a SOQL or SOSL query utilising user controlled input, but does not check if the calling user has permissions to do so. Permissions can be checked at multiple levels:

- Object Level - Can the user access this type of object and perform the requested action (create, read, edit, delete)?
- Field Level - Can the user access the requested fields on the object with the requested action (read, edit)?
- Record Level (Often referred to as Sharing Settings) - Which records within the object can the user access with the requested action (read, edit)?

These settings are then checked against the permissions defined within the current user's Profile and Permission set(s). If the user satisfies the access control check, the information will be returned. Alternatively if the user fails the check, then functionality will be implemented to either return `null` or an error message.

Example Vulnerable Apex

Take a scenario where the following Apex class was added to a component to a Lightning community for authenticated users.

```
public without sharing class CaseController {
        @AuraEnabled
        public static String getCurrentUser {
                String currentUser = [select Name from User where Id =:UserInfo.getUserId()].Name;
                return currentUser;
}
        @AuraEnabled
        public static List<Case> getCaseInfo(String name){
                return [SELECT FIELDS(ALL) FROM Case WHERE SuppliedName = :name LIMIT 20] ;
        }
}
```

The Helper JS file (caseHelper.js) looks like the following:

```
getCurrentUser : function(cmp) {
        var action = cmp.get('c.getCurrentUser');
        const userInput = cmp.get('v.name');
...
        action.setParams({email: userInput});
        action.setCallback(this,function(response){
          var state = response.getState();
          if (state === "SUCCESS"){
                const returnedUsername = response.getReturnValue();
                self.getCaseInfo(cmp,returnedUsername);
...
        getCaseInfo : function(cmp,userName) {
          var action = cmp.get('c.getCaseInfo');
...
```

The CaseController Apex class has some interesting methods:

- The getUser method returns the Name of the user calling it via the Aura API
- The getCaseInfo method returns a list of Case SObject records belonging to those with a supplied name via input, also invocable via the Aura API.
- The Apex class is defined as 'without sharing' so it runs in system context (similar to having the 'Modify All Data' permission)

The Helper JS chains these two methods together to be performed in a synchronous fashion, feeding the result of `getCurrentUser` to the `getCaseInfo` method.

Since the component and its namespace are leaked in the response of multiple endpoints, such as `bootstrap.js`, an actor could dissect the component and the callable actions defined in it using the method described in this article. And by doing so, easily discern what the application is trying to do here based on the Apex method names and parameter names. They could craft a call to the Aura API with the following `message` parameter payload to retrieve information on another user's case(s) by merely knowing their full name (assume that the namespace is 'c'):

```
{"actions":[{"id":"123;a","descriptor":"apex://CaseController /
ACTION$getCaseInfo","callingDescriptor":"UNKNOWN","params":{"name":"John Doe"}}]}
```

To test a sample component that's configured intentionally to have a lack of CRUD controls, we recommend attempting Lab level 2.

IMPLEMENTING ACCESS CONTROL CHECKS

There are multiple ways in which this class of vulnerability could be solved, each of which are extensively documented in the Apex developer guide. It's also fortunate that over past releases, Salesforce has committed themselves to a secure-by-default approach when it comes to Apex development. One such example includes implicit 'with sharing' for Apex classes.

In the case of our example class above, there is really no need for the `getCaseInfo` method to be AuraEnabled, as all user input could be removed by simply having the first class call the second with the current user's name. However, if the developer wanted to keep this style of functions interacting with each other, they could remove the 'without sharing' statement within the class declaration to ensure the Apex class is running in user context, which prevents users accessing records to which access has not been granted.

This is assuming that Guest Users do not have permissions to access these Apex methods due to Profile Permissions, with the lightning page that contains their accompanying components not visible due to Criteria Based Audience settings. In the event that they could mistakenly be given access, it would be wise to also perform Object and Field level checks by performing the SOQL query using 'WITH SECURITY_ENFORCED' for read operations and using the 'isCreatable', 'isDeletable', and 'isUpdateable' describe methods on DML operations. This would prevent the issue on older orgs (created pre-Summer'20) of Guest User's having the ability to be owners of the records they created, and thus able to access them if Case read permissions were removed but access checks at the object level were not performed in the case of custom Apex such as this.

Below is an example of how this Apex class could be redefined in a secure fashion:

```
...
public with sharing class CaseController {
        @AuraEnabled
        public static List<Case> getUserCaseInfo {
                String currentUser = [SELECT Name From User WHERE Id = UserInfo.getUserId()].Name;
        try {
                List<Case> caseRecords = [SELECT FIELDS(ALL) FROM Case WHERE SuppliedName = :currentUser LIMIT 20
                WITH SECURITY_ENFORCED ] ;
                        return caseRecords;
        } catch( System.QueryException ee) {
                        return null;
        }
}
...
```

SOQL INJECTION

If you're a security practitioner, you are likely no stranger to injection attacks on DBMS such as MySQL, MSSQL etc. The concept of SOQL injection is ultimately the same, an actor being able to access information they should not have access to, due to user input being supplied to an SOQL query. Even though this issue can have a critical impact depending on the point of injection within a query, the fact that SOQL queries are limited to 'read' as opposed to its SQL counterpart can lead developers into a false sense of security. This can cause them to opt for more flexible but inherently less secure options when building their queries, such as the use of dynamic queries as opposed to static queries.

Sample Vulnerable Apex

The singular method within this Apex class is utilised to show a user the content of any public document which has a keyword that matches that which the user supplies. While it does ensure that the calling user has access to the Document object's 'Body' field via a CRUD check prior to performing the query, it directly inserts user input into a dynamic SOQL query.

Apex Class

```
...
public without sharing class DocumentController {
        @AuraEnabled
        public static List<Document> getDocument(String searchString) {
                if (!Schema.sObjectType.Document.fields.Body.isAccessible()){
                return null;
                }
                String queryString = 'SELECT Body FROM Document WHERE (IsPublic = true AND Keywords =
\'%'+searchString+'%\' ';
                List<Document> documentContent = database.query(queryString);
                return documentContent;
        }
}
...
```

The corresponding controller JS for this Apex can be straightforward since it is only calling a single method:

```
...
{
        getData: function(component, event, helper) {
            var action = component.get('c.getDocument);
            action.setParams({searchString:component.get('v.userInput')});
            action.setCallback(this, function(response){
        var state = response.getState();
            if (state === "SUCCESS"){
                    const returnValue = response.getReturnValue();
              component.set('v.answer',returnValue);
              }
        });
        $A.enqueueAction(action);
          }
})
...
```

An actor could insert a crafted string that would be interpreted as a continuation of this SOQL query. The below payload is one such example of what they may use in their call to the Aura API, however it has been URL decoded for readability purposes, as the Aura API will throw an error upon seeing the '%' character in a fully decoded message.

```
...
{"actions":[{"id":"123;a","descriptor":"apex://DocumentController /
ACTION$getDocument","callingDescriptor":"UNKNOWN","params":{"searchString":"anything%')  OR (Name LIKE '"}}]}
...
```

On the back-end, the SOQL statement that would be executed is:

```
...
SELECT Body FROM Document WHERE (IsPublic = true AND Keywords LIKE '%anything%') OR (Name Like '%')
...
```

Since the new addition to the query will match every Document record, it will return the Body of all documents regardless of the Document being public or not.

For a hands-on example of a component that interacts with an SOQL injection vulnerable Apex method, see Lab Level 3.

Blind SOQL Injection

Basic SOQL injection scenarios have been extensively covered in the security space and typically re-use the example shown within Salesforce's official documentation. However very few have touched on the issue of Blind SOQL Injection, which is similar to Blind SQL injection techniques that leverage information in the HTTP response to discern if specific values exist in the database.

Moving from dynamic queries to static queries or escaping single quotes is often not enough, as an actor could take advantage of SOQL's wildcard characters (% and _) to enumerate values within records over a number of HTTP requests.

Sample Vulnerable Apex

Take the following example of Apex with the following logic, which may serve for checking the validity of your API key:

```
...
...
public without sharing class KeyHandler{
@AuraEnabled
public static Boolean checkKeyValidity(String userInput){
Integer validKey = [SELECT COUNT() FROM api_keys__c WHERE keyvalue__c LIKE :userInput WITH SECURITY_ENFORCED];
if(validKey == 1){
        return true;
} else {
        Return false;
}
...
...
```

Even if API keys are high entropy and relatively long, they are still at risk of being enumerated accurately through the '_' wildcard character in this specific scenario.

Initially, the length of the API key would need to be determined (assuming each key is generated with a different length), and then the individual characters. A sequence of malicious values for 'userInput' in the above example could take the form of the following:

```
"userInput":"__________" -> Check if there is another user's key of length 10 -> returned false
"userInput":"___________" -> Check if there is another user's key of length 11 -> returned true
"userInput":"a__________" -> Check if there is another user's key starts with a -> returned false
"userInput":"b__________" -> Check if there is another user's key starts with b -> returned true
"userInput":"b1_________" -> Check if there is another user's key starts with b1 -> returned true
..many attempts later..
"userInput":"b1gVS6pQ9x2" -> Check if there is another user's key is b1gVS6pQ9x2 -> returned true
```

Lab 4, the last level of the Aura component lab, demonstrates this vulnerability scenario.

SECURELY PERFORMING SOQL STATEMENTS

At this stage it must be understood that SOQL injection vulnerabilities can be largely context dependent. While the recommended use of static queries would have fixed the first example, they did not sufficiently protect against the Blind vector.

A combination of the following mechanisms should always be considered when passing user input to an SOQL query:

- Whenever possible, opt for static queries over dynamic queries
- If you are relying on the flexibility of dynamic queries, utilising string.escapeSingleQuotes() will prevent SOQL injection scenarios that take a similar form to the SOQL injection example in this article.
- Appropriately assign user input to variables of the appropriate type, allowing you to take advantage of typecasting prior to inserting them in your SOQL query.
- Avoid using the LIKE operator in conditional WHERE expressions when possible, and opt for more explicit operators such as '='. This removes the possibility of an actor taking advantage of wildcard characters.
- In extreme cases, consider building a Pattern and removing non-alphanumeric characters if it fits your use case.

Salesforce's Trailhead has an excellent resource which explains how different protection mechanisms each have their own place.

CONCLUSION

Salesforce is continuously pushing a secure-by-default model for custom development, but we understand that custom development is prone to human error. A recent trend study by Salesforce Ben on the Salesforce ecosystem has detailed that releases are going to be pushed out both quicker and more frequently, as development teams adopt a more Agile approach. Therefore, we recommend an aggressive approach to securing your Apex code by shifting your security efforts left. For one, purpose-built Static Code Analysis (SCA) tools that were specifically built to understand Apex, like Clayton, should be introduced into your DevOps pipeline to help identify software security issues before they go live. As there are always privileged code components intended to perform administrative or other sensitive functionality, the next step in securing your Salesforce Apex and other "cloud code" components is reducing the security surface area by gaining ongoing visibility into which user groups and personas have been provisioned access to these components. This can be a challenging task as there are multiple mechanisms through which access can be provisioned, with some not being visible or apparent in the standard UI. . The AppOmni for Salesforce platform can give you immediate visibility into these exposures and how they were created, as well as allow you to create continuous monitoring policies to ensure that your Salesforce environments remain secure and continuously managed.

Combining these two approaches will provide you with the best fighting chance at preventing insecure and inappropriately assigned Apex code from finding a home within your production instances.