Field Level Security in Apex: WITH SECURITY_ENFORCED vs. Security.stripInaccessible

Image

Every time your users access Lightning (web) component or Visualforce page the background Apex code is being run in user context. What does it mean? If there's *with sharing* keyword on your class definition or you are inheriting *with sharing* from another class, sharing rules are enforced. In other words SOQL query will return only records, that are visible for current user. And that's great, this is one security concern less. However **object level security and field level security permissions are not respected**, therefore results of database queries will contain fields, that current user doesn't have access to. There are 3 different ways, how to ensure your user will not see, what he's not supposed to see.

- WITH SECURITY_ENFORCED clause on SOQL queries
- Security class and its method stripInaccessible
- DescribeFieldResult class and its method isAccessible

Let's inspect them one by one.

Let's get our playground ready

First, we need to get our laboratory ready for experiments. Imagine we are implementing Salesforce for soon-to-be-awesome company called "Miracle Workers". Their sales representatives are working with accounts and contacts, but they do not need access to contacts' mailing addresses. Finance guys need access only to account to issue invoices.

Therefore we have set up following **sharing defaults**:

Object	Sharing
Account	Public read
Contact	Private

And **profiles**:

Profile name	Object level security	Field level security
Sales Representative	Account: Read Contact: Read, Edit	Account.Name: Read Contact.Name: Read Contact.Phone: Read
Finance	Account: Read	Account.Name: Read

Due to internal company processes some individuals will need access to contacts addresses, so we've created also one **permission set**:

Permission set name	Object level security	Field level security
Contact Address	Contact: Read	Contact.MailingAddress: Read

One important aspect of our today's experiments is we **will need to run these experiments in context of different users. Therefore we are going to use Salesforce testing framework and its great method System.runAs(User)**.

To skip unnecessary code, I will only claim, that I've created testing class *FLSTest* with *@testSetup* method creating following data.

Users:

Name	Profile	Permission set
William Shattner	Sales Representative	
Leonard Nimoy	Sales Representative	Contact Address
DeForest Kelley	Finance	

Account: Sub Pop Records

Contacts:

Name	Phone	MailingStreet	Owner.Name
Eddie Vedder	123456789	Pearl st. 24	William Shattner
Kurt Cobain	987654321	Nirvana st. 5	Leonard Nimoy
Layne Staley	555555555	Chains st. 34	DeForest Kelley

Simple SELECT

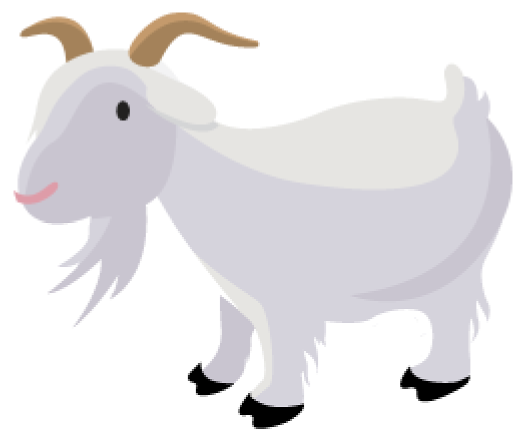
Just for the record, let me verify, that what I've been saying about *field level security*, is true with following *test method*.

```
@isTest
private static void testSelect() {
    Map<String, User> users = getUsers();
    Account wsAccount;
    Account lnAccount;
    Account dkAccount;
    System.runAs(users.get('William Shattner')) {
        wsAccount = [SELECT Name, (SELECT Name, Phone, MailingStreet FROM Contacts) FROM Account];
    }
    System.runAs(users.get('Leonard Nimoy')) {
        lnAccount = [SELECT Name, (SELECT Name, Phone, MailingStreet FROM Contacts) FROM Account];
    }
    System.runAs(users.get('DeForest Kelley')) {
        dkAccount = [SELECT Name, (SELECT Name, Phone, MailingStreet FROM Contacts) FROM Account];
    }
    System.debug(wsAccount);
    System.debug(wsAccount.Contacts);
    System.debug(lnAccount);
    System.debug(lnAccount.Contacts);
    System.debug(dkAccount);
    System.debug(dkAccount.Contacts);
}
```

Results:

```
Account:{Name=Sub Pop Records, Id=0013X00002UYISaQAP}
(Contact:{AccountId=0013X00002UYIRIQA5, Id=0033X00002tiiBMQAY, Name=Eddie Vedder, Phone=123456789, MailingStreet=Pearl st. 24})
Account:{Name=Sub Pop Records, Id=0013X00002UYISaQAP}
(Contact:{AccountId=0013X00002UYIRIQA5, Id=0033X00002tiiBNQAY, Name=Kurt Cobain, Phone=987654321, MailingStreet=Nirvana st. 5})
Account:{Name=Sub Pop Records, Id=0013X00002UYISaQAP}
(Contact:{AccountId=0013X00002UYIRIQA5, Id=0033X00002tiiBOQAY, Name=Layne Staley, Phone=555555555, MailingStreet=Chains st. 34})
```

As you can see, **sharing rules are respected** – Account *Sub Pop Records* is visible for all users, but each user sees just Contact he owns. On the other hand, object level security permissions were skipped as user DeForest Kelley sees Contact and field level security permissions were skipped as users William Shattner and Deforest Kelley can see Contact.MailingStreet.



WITH SECURITY_ENFORCED

First way to ensure object and field security permissions is **WITH SECURITY_ENFORCED** clause on SOQL query. This **will raise exception in case SOQL query tries to access something, that's not visible for the user**.

```
@isTest
private static void testSelectWithSecurityEnforced() {
    Map<String, User> users = getUsers();
    System.runAs(users.get('William Shattner')) {
        try {
            Account wsAccount = [SELECT Name, (SELECT Name, Phone, MailingStreet FROM Contacts) FROM Account WITH SECURITY_ENFORCED];
            System.debug(wsAccount);
            System.debug(wsAccount.Contacts);
        } catch (Exception ex) {
            System.debug(ex.getMessage());
        }
    }
    System.runAs(users.get('Leonard Nimoy')) {
        try {
            Account lnAccount = [SELECT Name, (SELECT Name, Phone, MailingStreet FROM Contacts) FROM Account WITH SECURITY_ENFORCED];
            System.debug(lnAccount);
            System.debug(lnAccount.Contacts);
        } catch (Exception ex) {
            System.debug(ex.getMessage());
        }
    }
    System.runAs(users.get('DeForest Kelley')) {
        try {
            Account dkAccount = [SELECT Name, (SELECT Name, Phone, MailingStreet FROM Contacts) FROM Account WITH SECURITY_ENFORCED];
            System.debug(dkAccount);
            System.debug(dkAccount.Contacts);
        } catch (Exception ex) {
            System.debug(ex.getMessage());
        }
    }
}
```

Results:

```
Insufficient permissions: secure query included inaccessible field
Account:{Name=Sub Pop Records, Id=0013X00002UYIkxQAH}
(Contact:{AccountId=0013X00002UYIkxQAH, Id=0033X00002tiiVNQAY, Name=Kurt Cobain, Phone=987654321, MailingStreet=Nirvana st. 5})
Insufficient permissions: secure query included inaccessible field
```

Well, it works, but is it really useful? In my opinion in most cases **you need to know, that you've tried to query fields without sufficient permissions, but you also need your results! Ideally with unpopulated inaccessible fields.**

Imagine we have Lightning (web) component used by sales representatives. Important part of that component is to display contacts. Using **WITH SECURITY_ENFORCED** would render this component useless for most of the sales representatives, even though you've only wanted to hide MailingAddress from them!

The only possible use case I see now is to cut off user from component completely, which should be handled in profiles and permission sets.

Brand new Security class and its stripInaccessible method

In *Winter '20 release* Salesforce has introduced Security class with powerful method *stripInaccessible(accessCheckType, sourceRecords)*. This method strip inaccessible fields from records, that have already been retrieved or have been deserialized from other source. Salesforce itself even instructs to use this method to verify accessibility before inserting any record obtained from untrusted source. This slightly implies, that *stripInaccessible* also checks for *sharing rules*, but it is not!

```
@isTest
private static void testStripInaccessible() {
    Map<String, User> users = getUsers();
    SObjectAccessDecision wsStrippedRecords;
    SObjectAccessDecision lnStrippedRecords;
    SObjectAccessDecision dkStrippedRecords;
    System.runAs(users.get('William Shattner')) {
        wsStrippedRecords = Security.stripInaccessible(AccessType.READABLE, [SELECT Name, (SELECT Name, Phone, MailingStreet FROM Contacts) FROM Account]);
    }
    System.runAs(users.get('Leonard Nimoy')) {
        lnStrippedRecords = Security.stripInaccessible(AccessType.READABLE, [SELECT Name, (SELECT Name, Phone, MailingStreet FROM Contacts) FROM Account]);
    }
    System.runAs(users.get('DeForest Kelley')) {
        dkStrippedRecords = Security.stripInaccessible(AccessType.READABLE, [SELECT Name, (SELECT Name, Phone, MailingStreet FROM Contacts) FROM Account]);
    }
   System.debug(((List<Account>) wsStrippedRecords.getRecords())[0].Contacts);
    System.debug(((List<Account>) lnStrippedRecords.getRecords())[0].Contacts);
    System.debug(((List<Account>) dkStrippedRecords.getRecords())[0].Contacts);
}
```

Results:

```
(Contact:{AccountId=0013X00002UYJh3QAH, Id=0033X00002tijeUQAQ, Name=Eddie Vedder, Phone=123456789})
(Contact:{AccountId=0013X00002UYJh3QAH, Id=0033X00002tijeVQAQ, Name=Kurt Cobain, Phone=987654321, MailingStreet=Nirvana st. 5})
()
```

The only draw back of this approach is, you will get ***"FATAL_ERROR System.SObjectException: SObject row was retrieved via SOQL without querying the requested field"*** **exception, if you try to access stripped properties**. Fortunately, there is a beautiful way, how to over come this. Following example is not really sophisticated, but clear enough to explain the concept.

```
@isTest
private static void testUnpopulateInaccessible() {
    Map<String, User> users = getUsers();
    SObjectAccessDecision wsStrippedRecords;
    System.runAs(users.get('William Shattner')) {
        wsStrippedRecords = Security.stripInaccessible(AccessType.READABLE, [SELECT Name, (SELECT Name, Phone, MailingStreet FROM Contacts) FROM Account]);
    }
    List<Contact>>contacts = ((List) wsStrippedRecords.getRecords())[0].Contacts;
    for (Contact contact : contacts) {
        for(String fieldName : wsStrippedRecords.getRemovedFields().get('Contact')) {
            contact.put(fieldName, null);
        }
    }
    System.debug(contacts);
}
```

Result:

```
(Contact:{AccountId=0013X00002UYJu5QAH, Id=0033X00002tijqBQAQ, Name=Eddie Vedder, Phone=123456789, MailingStreet=null})
```

If you are clever - and I think we've been clever so far - **you will find this method way more flexible than *WITH SECURITY_ENFORCED***. You can also use it in the completely same way as *WITH SECURITY_ENFORCED*, if you raise your own exception in case *SObjectAccessDecision.getRemovedFields()* returns non empty map.

As said before, **Security.stripInaccessible doesn't help with sharing**. This means, when you are working with records, that were provided to you by the source you have no visibility over, you have to perform another check yourself.

DescribeFieldResult class

This is the oldest way to manage field level security access. It still works, but it's not as easy to use as *Security.stripInaccessible*. The thing is you have to check for each field yourself.

In following example we will perform semi-generic check of queried Contact records. We will pretend, that we won't have any information about what fields will be queried. On the other hand we will assume, there will be no subquery on related records.

To find out, which fields were actually queried, we are going to use method *getPopulatedFieldsAsMap*. Because this method doesn't consider field with null value to be populated field, we need to assess each record separately.

```
@isTest
private static void testIsAccessible() {
    Map<String, User> users = getUsers();
    List<Contact> wsContacts;
    System.runAs(users.get('William Shattner')) {
        wsContacts = [SELECT Name, Phone, MailingStreet FROM Contact];
        Map<String, Schema.SObjectField> fieldMap = Schema.SObjectType.Contact.fields.getMap();
        Map<String, Boolean> fieldToAccessibility = new Map<String, Boolean>();
        for (Contact contact : wsContacts) {
            Set<String> populatedFields = contact.getPopulatedFieldsAsMap().keySet();
            for (String fieldName : populatedFields) {
                Boolean isAccessible = fieldToAccessibility.get(fieldName);
                if (isAccessible == null) {
                    isAccessible = fieldMap.get(fieldName).getDescribe().isAccessible();
                    fieldToAccessibility.put(fieldName,isAccessible);
                }
                if (!isAccessible) {
                    contact.put(fieldName, null);
                }
            }
        }
     }
     System.debug(wsContacts);
}
```

Results:

```
(Contact:{Name=Eddie Vedder, Phone=123456789, MailingStreet=null, Id=0033X00002tjA7oQAE})
```

Benchmark

There is one last thing I am really interested in and that's performance. I've benchmarked all discussed approaches, even though each of them is doing something little different. Code below can be hugely improved for specific use case, therefore results are only indicative.

```
@isTest
private static void testBenchmark() {
    Map<String, User> users = getUsers();
    List<Contact> contactsToInsert = new List<Contact>();
    for(Integer i = 0; i < 1000; i++) {
        contactsToInsert.add(new Contact(
            FirstName = 'Chris',
            LastName = 'Cornell',
            Phone = '111111111',
            MailingStreet = 'Garden st. 63',
            OwnerId = users.get('William Shattner').Id
        ));
    }
    insert contactsToInsert;

    //TEST WITH SECURITY_ENFORCED
    System.runAs(users.get('William Shattner')) {
        try {
            List<Contact> wsContacts = [SELECT Name, Phone, MailingStreet FROM Contact WITH SECURITY_ENFORCED];
        } catch(Exception ex) {
        }
    }
    //TEST Security class
    System.runAs(users.get('William Shattner')) {
        SObjectAccessDecision wsAccessDecision = Security.stripInaccessible(AccessType.READABLE, [SELECT Name, Phone, MailingStreet FROM Contact]);
        List<Contact> contacts = wsAccessDecision.getRecords();
        Set<String> removedFields = wsAccessDecision.getRemovedFields().get('Contact');
        for (Contact contact : contacts) {
            for(String fieldName : removedFields) {
                contact.put(fieldName, null);
             }
       }
    }

    //TEST DescribeFieldResult class
    System.runAs(users.get('William Shattner')) {
        List<Contact> wsContacts = [SELECT Name, Phone, MailingStreet FROM Contact];
        Map<String, Schema.SObjectField> fieldMap = Schema.SObjectType.Contact.fields.getMap();
        Map<String, Boolean> fieldToAccessibility = new Map<String, Boolean>();
        for(Contact contact : wsContacts) {
            Set<String> populatedFields = contact.getPopulatedFieldsAsMap().keySet();
            for (String fieldName : populatedFields) {
                Boolean isAccessible = fieldToAccessibility.get(fieldName);
                if (isAccessible == null) {
                    isAccessible = fieldMap.get(fieldName).getDescribe().isAccessible();
                    fieldToAccessibility.put(fieldName,isAccessible);
                }
                if (!isAccessible) {
                    contact.put(fieldName, null);
                }
            }
        }
    }
}
```

Results:

```
WITH SECURITY_ENFORCED:          6 ms
Security.stripInaccessible:      250 ms
DescribeFieldResult.isAccessible: 590 ms
```

I guess, we have all expected these result. Not only ***Security.stripInaccessible*** **is easier to work with, but it is also way faster than *DescribeFieldResult.isAccessible***. *WITH SECURITY_ENFORCED* doesn't cost anything, but I struggle to find good use case for it.