Chained to hit: Discovering new vectors to gain remote and root access in SAP Enterprise Software

Pablo Artuso
Onapsis
partuso@onapsis.com

Yvan Genuer
Onapsis
ygenuer@onapsis.com

Abstract

At the core of every business on the planet there will always be a mission critical application system. Overlooking its security is senseless and at the same time dangerous as it will result in putting your business at a high risk.

During 2022 multiple months-lasting research projects were kicked off as part of the Onapsis Research labs. Even though each of them had their own important results, no one was expecting that a combination of them would end up in finding chains of exploitation which could cause serious damage.

This documentation will begin with the analysis of "P4", a proprietary protocol based on Remote Method Invocation (RMI), which is uncommonly exposed to untrusted or public networks and thus making it unreachable from the Internet. Not only several critical and not-so-critical vulnerabilities will be shared, but most importantly the tactics & techniques used to unveil them.

Then, it will continue with the exploration of the Java Naming and Directory Interface (JNDI) reference injections where usual exploitation techniques did not work. As a consequence, a new and specific vector of attack was developed which included a deep dive into JNDI internals in SAP. Additionally, a reverse connectionless exploitation will be explained.

Finally, where a widely used component running as root or nt/system was targeted, which concluded the discovery of a critical flaw that may allow a local attacker to completely compromise the whole system beyond application's boundaries.

This whitepaper paper will cover in detail the analysis, processes and outcomes for each of the projects mentioned. Additionally, it will illustrate how it's possible to chain findings in order to empower the impact either in terms of criticality or exposition: Turning local network attacks to Internet-exploitable through tunneling protocols or taking root LPE to remote and anonymously exploitable.

Keywords

P4CHAINS, JNDI Reference Injection, SAP, Enterprise Software, RCE, root, P4, HTTP, httpP4tunnel, JNDI, Solution Manager, SAP Diagnostic Agent, Host Agent, Portal, RMI

1. Introduction

1.1. SAP SE

Enterprise software is one of the most important topics when discussing a company's assets. They usually manage sensitive and critical information. It is because of this reason that companies opt for experts in the field to trust one of their most critical assets. SAP is one of the largest vendors of Enterprise Software. They have been successfully developing business applications for 50 years now. With more than 450k customers and presence in more than 180 countries, it is possible to believe that almost every mid to large company today is using SAP systems for keeping its business up and running. The list of products that SAP offers is very extensive. Customers may choose which product to use based on their particular needs. However, most of these products have a common technical base: the SAP Netweaver or more recently the S4/HANA. These technical basis layers offer various network services, of which the most known are the Internet Communication Manager (ICM), Gateway service (GW), Message Server (MS) as well as end users dialog service (DIAG).

1.2. P4 protocol

P4[1] is one of the several proprietary protocols provided by SAP. Based on the well-known Java protocols RMI[2] and CORBA[3], it provides necessary features to establish communications between remote objects from different namespaces and hosts. This way P4 clients could perform different kinds of actions against objects which are inside the remote server's scope. By default, P4 listens in port 5NN04 (NN being the System Number) and it's binded to every host's interface where the system is running. As a consequence, remote access to this port is possible unless extra configuration is carried out. Due to the fact that its implementation resides inside the SAP Java NetWeaver layer (section 3.4.2), this protocol is present in several and widely-used SAP products and solutions such as: SAP Enterprise Portal, SAP PI/PO, SAP Solution Manager and more.

1.3. JNDI

The **J**ava **N**aming and **D**irectory **I**nterface (JNDI)[4], as the name suggests, is an Java standard implementation of a naming and directory service. In simple words it provides access to common resources (e.g: objects) through specifying a simple string. Its implementation is divided in two main components:

- **API**: **A**pplication **P**rogramming **I**nterface used by Java applications to perform all actions exposed by these services.
- **SPI**: **S**ervice **P**rovider **I**nterface which handles the way these services connect to external or internal resources to accomplish their task. Some of these providers are: RMI, CORBA, LDAP, DNS and more.

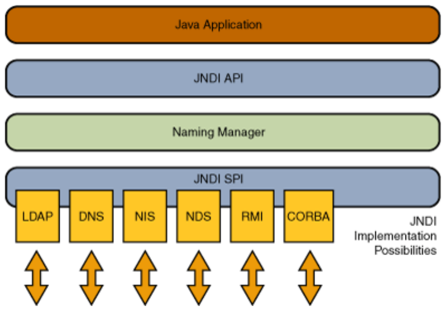

Figure 1: *JNDI Architecture*

If further knowledge related to JNDI is needed please refer to official documentation or to the "JNDI 101" section of Alvaro Muñoz and Oleksandr Mirosh research[5][6].

1.4. SAP Products and Components

1.4.1 Solution Manager

In SAP landscapes, the SAP Solution Manager (SolMan) could be compared to a domain controller system in the Microsoft world. It is a technical system that is highly connected with powerful privileges to all other SAP systems. Once an SAP system is connected to the solution manager it receives the name of "managed" or "satellite" system. As an administration solution, SolMan aims to centralize the management of all systems within the landscape by performing actions such as implementing, supporting, monitoring and maintaining the enterprise solutions. If an SAP customer wants to fully utilize the capabilities of the Solution Manager, they must install an application called Solution Manager Diagnostic Agent (SMDAgent) on each host where an SAP system is running. This Agent manages communications, instance monitoring and diagnostic feedback to the Solution Manager. From the operating system perspective, the unique user involved in all SMDAgent activities is daaadm. Administrators or end users never use P4 directly. They use HTTP or the SAP client (SAPGui) to interact with Solman. The P4 serice is used for technical and internal purposes only. Most interestingly, Solman uses it for communication between itself and all satellites systems through SMDAgent. The following image shows that Solman (SOL) manages 6 satellites (D01, Q01, P01 and D02, Q02, P02). The yellow lines, communication from SMDAgent to Solman, is done by the P4 services handle in Solman.

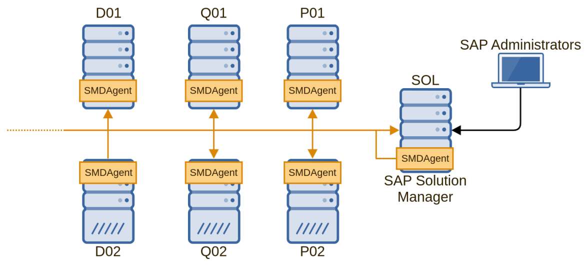

Figure 2: *Example architecture including SAP Solution Manager and SMDAgents*

1.4.2 NetWeaver Java

As mentioned above the SAP Netweaver is the basis layer, or backbone, of SAP products. This Netweaver is delivered with two different technologies, which are called "Stacks". They are the ABAP Stack and the JAVA Stack[7]. This Netweaver JAVA Stack was developed around 2000's to initially replace the ABAP Stack. But quickly SAP decided to keep both Stacks and dedicated particular products for each one. The JAVA Stack was used for products related to internet and http access. For example the Enterprise Portal runs under a JAVA Stack only. Several products, like the ERP, work on ABAP Stack and also few products, like Solution Manager or CRM, use both Stacks. This JAVA technical basis layer provides common services for all products, like Gateway, HTTP and P4 communication through the Internet Communication Manager (ICM), etc. Because the Netweaver JAVA does not provide the DIAG service, you cannot use the SAPGui (SAP Client) to connect to the system. Usually, end users use the SAP Fiori, an html5 interface based on http communications, to work on products that run under Netweaver JAVA.

1.4.3 Enterprise Portal

SAP Enterprise Portal[8] is the Web front-end component for SAP JAVA NetWeaver. It works as a "hub" to align people, information and business processes across the company for a large number of users who require access to many different applications and services (SAP or not). It usually does not contain a lot of critical data, but it is a SAP system with a lot of remote connection to other systems in the company landscape. From a security perspective it is an important component because of the number of connections as well as it is, most of the time, an internet facing component and could be an entry point for attackers.

1.4.4 Start Service

The SAP Start Service is a component installed automatically during the installation of a new SAP system. It is OS and database independent, and it can accomplish several life-cycle tasks such as: Monitoring, Start/Stop instances, and Preparing for upgrade. It is implemented as a service on Windows, and as a daemon on UNIX. These services are provided on Host Control SOAP Web Service, under different namespaces like SAPOsCol, SAPHostControl or SAPCCMS among others. The ports used are 1128 (http) and 1129 (https). By default, remotely only few SOAP services are accessible without authentication, and locally only few more are also accessible anonymously. All other SOAP services required high privileges authentication.

2. Previous work

2.1. P4 Protocol

2.1.1 [2012] Arbitrary file read in P4 service

Initially found by Juan Pablo Perez Etchegoyen and patched in 2012 with the SAP Security Note (SSN) 1682613[9]. This vulnerability, with Maximum CVSS score, allowed an unauthenticated attacker to download any type of file owned by SAP user administrator through P4. Empowered by its exploiting easiness, it could lead to a complete compromise of the system as the attacker could retrieve the SAP Secure File and further decrypt high privileged user credentials. This attack was presented as part of a bigger research project in Ekoparty 2013[10].

2.1.2 [2014] Dissecting and attacking RMI frameworks

Discovered and presented by Nahuel D. Sánchez and Sergio Abraham in the EkoParty conference of 2014[11]. They showed different attacks in some specific services exposed by the SAP Java NetWeaver layer. Their findings were patched with SAP Security Notes: [12] and [9].

2.1.3 [2017] Java deserialization attack through P4

Discovered by Kai Ullrich and patched in 2017 with the SSN 2443673[13]. This is basically the well known java deserialization attack found and adapted to the SAP JVM. If exploited, this finding could lead to a remote anonymous OS command execution. In-depth Information[14] and PoC[15] were shared by the author.

2.1.4 [2020] Communication hijacking

Found by Yvan Genuer and patched in 2020 with SAP Security Note [16], this flaw was identified as CVE-2020-6198 and given a CVSS score of 9.8. Found as part of a bigger research affecting the SAP Solution Manager[17], exploiting this flaw could lead to arbitrary file download or unauthenticated RCE (if combined with a second vulnerability: CVE-2019-330).
Technically speaking, if a P4 communication was in progress between two systems, it was possible to remotely hijack the communication by brute forcing a part of the security token, even if P4S (SSL for P4) was enabled. Once the attacker guessed this token, it was possible to execute any P4 service without the need of providing authentication.

2.1.5 [2021] P4 Service listing and analysis

Kai Ullrich wrote a very interesting blog post [18] where he explained his journey into finding a new Java deserialization gadget specifically for SAP. Despite not being able to find a gadget, he finally found an unrelated-to-deserialization vulnerability with CVSS 9.6 identified as CVE-2021-21481 and patched with SSN 3022422[19]. The analysis he carried is strongly related to the research we present in this paper. As a matter of fact, some of the knowledge Kai shared was used to circumvent some obstacles faced during the P4 analysis phase.

2.2. Start Service

2.2.1 [2009] Missing authentication

The very first external security research about Start Service was done by Jordan Santarsieri from Onapsis late in 2009[20]. At this time the administration functions provided by the service were accessible locally and remotely. Most of them without particular authorization. Which could lead to critical information disclosure as well as OS command execution.
From these findings SAP released the SSN 1439348[21] and integrated a new parameter "service/protectedwebmethods" to enable authentication for almost all functions and let only few not dangerous functions accessible anonymously. Around 2011, Chris John Riley from (in)Security delivered talks[22] about it and also created several metasploit modules[23] related to Start Service.

2.2.2 [2020] Multiple Privileges Escalation

Pablo Artuso and Yvan genuer analyzed the Start Service as part of a bigger reserach project. After a careful analysis of each of the functions exposed, it was possible to identify several (10+) of them that were vulnerable to command injection. Although these functions required OS authentication, they were finally executing commands as root or nt authority/system. Therefore this injection led to a privilege escalation.
They all were identified as CVE-2020-6234 and patched in SSN 2902645[24]. Three of them, regarding functions ExecuteInstallationProcedure, ACOSPrepare and ExecuteOperation were highlighted during the BlackHat USA 2020 event[25].

2.3. JNDI

2.3.1 [2016] A journey from JNDI/LDAP manipulation to Remote Code Execution Dream Land

This research, carried out by Alvaro Muñoz and Oleksandr Mirosh, was presented at Black Hat USA 2016[5] and further explained in its whitepaper[6].
They present novel ways to lead to Remote Code Execution attacks abusing unprotected JNDI lookups through Reference injection. Beginning with the basics of JNDI, Alvaro and Oleksandr end up showing how through several protocols (RMI, LDAP, IIOP, etc) it was possible to make vulnerable servers fetch attacker-controlled resources and thus lead to RCE.

2.3.2 [2019] JNDI Reference injection through Local Classes

In this post[26], Michael Stepankin showed a way to achieve RCE in Apache through JNDI Reference injection even when the JVM was protected against loading external references from untrusted sources.
The idea of the attack was to leverage local classes (implemented inside the server). By carefully crafting a specific JNDI Reference and serving it in an attacker-controlled resource, Michael found the way to achieve remote code execution at the moment of the local class instantiation.
Despite the fact that it was related to specific software (Apache) and therefore not directly applicable in the context of this research, the idea of using local classes will be utilized.

3. Analysis

3.1. P4

3.1.1 Context

In order to analyze P4, several systems with different versions were used. This gave us the possibility not only to confirm that the findings were present in several components but also to find issues that were specific to certain solutions. The following table highlights the different versions used during this research.

Kernel Versions
7.50.3301.472568.20220902101413
7.50.3301.467525.20210601093523
7.50.3301.407179.20200416085516

SERVERCORE / CORE-TOOLS/ J2EE FRMW
1000.7.50.24.7.20221009183400
1000.7.50.22.0.20210804111800
1000.7.50.2.0.20160125191600

Figure 3: *JAVA Kernel and CORE-TOOLS versions.*

3.1.2 Initial connection and services listing

As mentioned in 1.2, P4 is based on RMI. In addition, systems exposed services through JNDI. As explained in SAP's docs[27] this simple JNDI connection will execute a lookup:

```java
public class P4Example {
  private static InitialContext ctx = null;
  public static void main(String[] args) {
    init("P4://", "localhost", "50004",
        "User", "Password", null);
  }

  public static void init(String schema,
      String host, String port, String user,
      String pass, String transportType) {
    Properties p = new Properties();
    if (schema == null) {schema = "P4://";}
    p.put("java.naming.factory.initial",
        "com.sap.engine.services.jndi.
InitialContextFactoryImpl");
    p.put("java.naming.provider.url", schema
        + host + ":" + port);
    p.put("java.naming.security.principal",
        user);
    p.put("java.naming.security.credentials",
        pass);
    ctx = new InitialContext(p);
}
```

Listing 1: *Example of simple JNDI connection to SAP's P4 port.*

P4 default port follows the pattern 5**XX**04, where **XX** is the SAP's instance number.
Despite the fact that we knew how to perform these lookups, at this point it was unknown which names could be used. In order to find the available JNDI names, the Telnet interface that Java systems provide was used. According to its documentation by adding the **NAMING** set of commands and later executing "LS -l -f" all the JNDI names together with its locations will be listed.
The obtained list was large (around 4500 services). Furthermore, some of these services were not "lookupable" in a remote way. With the idea of filtering out the ones that returned errors or null objects, we developed a Java script that finally led us with approximately 200 services. It was just time to roll up our sleeves.

3.1.3 Strategy and Toolset

In order to analyze such a number of services it was necessary to build a robust and systematic strategy. It is worth mentioning that this strategy was finally built in an iterative fashion while the analysis was being carried out. As part of it, several tools and resources were used:

- **Custom Java scripts:** Mainly to execute the lookups, create instances of the objects referenced, invoke its methods, etc. In this part, Kai Ullrich's script[18] was strongly leveraged to list the interfaces that proxies classes were implementing.
- **Live Debugging tools:** Such as JDB.
- **Server logs:** With appropriate levels of logging they are a key source of useful information to understand what is going on.
- **Java Code Catalog:** An internal tool developed by Onapsis' Security Research Team, more specificly by Ignacio Favro, which is able to find classes based on the method names or interface names that they implement.

Making use of these tools and resources, the following strategy (list of steps) was followed:

1. **List all services:** By using Kai Ullrich's script or/and other custom scripts, get a list of all services remotely exposed.

2. **Activate debugging in the system:** Allow the system to be debugged.

3. **Choose a service:** Select one of the JNDI names (linked to services) to be analyzed.

4. **Find implementing interfaces:** Using the Custom Java Scripts list all interfaces that the object referenced by the JNDI name implements. Furthermore, look for the interface and its implementation.

5. **Enable logs:** Turn on the most detailed level every source of logging that is related to the targeted service.

6. **Begin the analysis:** Both static and dynamic. Through the analysis of the source code of the targeted service try to discover security vulnerabilities.

7. **Track Execution flow:** By using Java Code Catalog, Live debugging or simple greps try to trace the execution flow and the calling stack when needed.

8. **Documentation:** Almost every step of the analysis is documented. The creation of a strong base of documentation was key to step on firm ground, otherwise the whole strategy becomes uncontrollable.

9. **Continue with the next service:** Go back to step 3.

Executing this strategy and using the mentioned toolset and resources, we began with the analysis of each of the exposed services.

3.1.4 Services inspection and findings

After analyzing every service that was found exposed through P4, we reported 13 vulnerabilities ranging from CVSS 5.3 to 10.0. Nonetheless, the following table gives a summary of all the security flaws that were found:

Service Name	**Description**	**CVSS**	**CVE**
Agent Simulation	Pre-auth RCE in Diagnostic Agents runnning Windows	10.0	CVE-2023-27497
Search Facade	SQL injection + DoS	9.9	CVE-2022-41272
Locking	DoS + Arbitrary OS File Read	9.9	CVE-2023-23857
Job Bean	SQL injection + DoS	9.4	CVE-2022-41271
RFC Engine	Anonymous RFC execution + password disclosure	9.4	CVE-2023-0017
OSCommand Bridge	Potential RCE in Diagnostic Agents	9.0	CVE-2023-27267
Remote Object Factory	JNDI Reference injection (pre-auth start of apps)	8.2	CVE-2023-30744
Agent Simulation	HTTP Header Injection in Solution Manager	7.2	CVE-2023-36921
Agent Simulation	Unauthenticated blind SSRF in Solution Manager	7.2	CVE-2023-36925
Cache Analyzer	Information Disclosure	5.3	CVE-2023-26460
Classload	Information Disclosure	5.3	CVE-2023-24526
Deploy	Information Disclosure	5.3	CVE-2023-24527
Object Analyzer	Information Disclosure	5.3	CVE-2023-27268

Table 1: *P4 related vulnerabilities reported.*

3.1.4.1 RFC engine

3.1.4.1.1 Analysis

- **JNDI Name:** rfcengine
- **Interface:** RFCRuntimeInterface_Stub

Within SAP's world there exists a proprietary and heavily used protocol called Remote Function Call (RFC). Usually used to establish communication between systems, this protocol provides an interface to execute functions in a remote way.

In this specific case, the rfcengine service is in charge of the implementation of the RFC functions and communications that the SAP Java system will support. In order to configure and make use of this feature, the Jco RFC Provider application was built[28].

As depicted in the cited documentation, the RFC Engine allows to process both, outcoming and incoming RFC function execution.

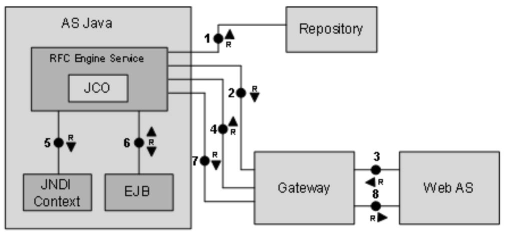

Figure 4: *rfcengine architecture.*

In order to be able to process incoming RFC function calls, the Java system must accomplish two requirements: Be connected to a repository and be registered as an External Server in the Gateway of the caller system. In order to make use of a repository, credentials (user, password and system properties) are required.
The RFC functions that are going to be able to be executed and processed by the Java system, will depend on the software installed.
All methods of interface RFCRuntimeInterface_Stub implemented by the rfcengine object did not require neither authentication nor authorization.

3.1.4.1.2 Findings

- **SAP Security Patch:** 3268093
- **CVE:** CVE-2023-0017

Unauthenticated execution of RFC functions
Through the execution of the $addBundle()$ method, it was possible to add a new connection to any arbitrary Gateway and repository. Basically meaning that without authentication an attacker could register the targeted Java system as an External Server in the Gateway of an attacker-controlled ABAP server.
Once this is carried out, the attacker could start executing the RFC calls against the Java system. As mentioned in the previous section, the impact will depend on the functions implemented, which will depend on the software installed in the system. Based on our analysis:

- **Enterprise Portal:** It is possible to create tasks with arbitrary content and assign it to any arbitrary user using the function CREATE_AWF_TASK. This could be used to impersonate users and to spread phishing in an effective way.
- **Solution Manager:** Could lead to RCE through using function FM_MAI_SIMULATION_AGENT (explained in following sections) or with very rarely using FM_GPCR_OS_COMMAND.

Unauthenticated retrieval of configured Jco plain passwords
As mentioned in the Analysis section, the RFC Engine Service is directly related to the configuration of JCo destinations. These destinations are often configured when information must be fetched from other systems. For instance, when using the portal (EP), it needs to consume information from internal systems.
Through the execution of method $getConfigurations()$, all the information related to JCo connections will be retrieved without authentication. However, for security reasons, when the object containing all the information is inspected, the password is

masqueraded. Nonetheless, due to being executed through a remote protocol (P4), analyzing the traffic it was possible to find that the actual password is being sent by the server. Therefore, as a summary, any anonymous attacker with access to the P4 port of the Java NetWeaver based system, will be able to extract all the JCo destinations configuration information (including plain passwords).

3.1.4.2 Search Facade

3.1.4.2.1 Analysis

- **JNDI Name:** com.sap.aii.af.search.api .SearchFacadeRemote
- **Interface:** SearchFacade

This SearchFacade object seemed to be used to retrieve information about very specific data related to technical stuff. Filters, extractors, components profiles are some of the keywords that seemed to be involved.
From a more technical perspective, the SearchFacade object had intensive interaction with the database. There were several functions remotely exposed that would allow somebody to insert, modify and delete some of the aforementioned keywords, and thus interact with the underlying database.

3.1.4.2.2 Findings

- **SAP Security Patch:** 3273480
- **CVE:** CVE-2022-41272

First and foremost, it was possible to remotely obtain an instance of this object through P4 without providing authentication or authorization.
However the biggest finding was related to the fact that SearchFacade was interacting with the database. Despite using prepared statements and binding the variables in a correct way, a function name delete was vulnerable to SQL injection. The flaw resided in the fact that the base query was dynamically built considering unsanitized input. Therefore, at the moment of dynamically binding the variables of the prepared statement, the injection was already present. This function didn't have output and thus the exploitation was blind.
This meant that every table of the vulnerable system was able to be read and exfiltrated, by an anonymous remote attacker.

3.1.4.3 Locking

3.1.4.3.1 Analysis

- **JNDI Name:** locking
- **Interface:** LockinRuntimeInterface_Stub

The main goal of this object seemed to be the implementation of the lock mechanism within the system. It provides functionality not only to check which locks are already in use and who has them, but also to acquire new locks. Additionally, there is functionality to read the profile (configuration file) of the Enqueue Server, which is a specific SAP component specifically in charge of managing locks.

3.1.4.3.2 Findings

- **SAP Security Patch:** 3252433
- **CVE:** CVE-2023-23857

All functionalities and methods exposed by this object were able to be called with the absence of authentication or authorization. As a consequence, any unauthenticated attacker could start acquiring locks in a non-stop way, causing the whole system to be stuck. For example, application locks may never be released by the attacker and therefore nobody could make use of them.
Additionally, a way to display the content of any arbitrary file in the OS was found. Despite the lack of authorization or authentication, the "read" execution was being carried out with <sid>adm privileges. Furthermore, there was no restriction in the type of the targeted file.
As stated by the official documentation[29], SAP systems always store credentials in an encrypted way inside the OS. These files are known as Secure Storage or SSFS. There exists

ways to decrypt these files. In fact, some of them are publicly available[30].
Leveraging the OS file read, it is possible to exfiltrate the Secure Storage of the targeted system and later decrypt it. The final results will be new plain text passwords, including the credentials to connect to the database and the well known Master Password.

3.1.4.4 Agent Simulation

3.1.4.4.1 Analysis

- **JNDI Name:** FM_MAI_SIMULATION _AGENT
- **Interface:** com.sap.sup.admin.connection. factory.AbapFactoryBean

To analyze this specific JNDI service, first of all we tried to retrieve involved jar files where this lookup is implemented. Due to that, we search, then download all the jar files from Solman (around 2000 files for 1G) and simply search for string patterns like "simulation" or "mai". This highlights the package abapconnector.jar. Analyzing the package, we found the exact JNDI service name, FM_MAI_SIMULATION_AGENT, in the class com.sap.sup.admin.connection. factory.AbapFactoryBean which confirmed that we dealt with the correct package. Dig into it a little more then we found inputs and outputs parameters name and type.

Name	Type	Direction
im_agent_name	String	input
im_collector_class	String	input
im_context_params	JCO.Table	input
im_input_params	JCO.Table	input
im_metric_params	JCO.Table	input
ex_metric_data	JCO.Table	output
exp_rc	Char1	output
exp_rc_msg	Char1024	output

Figure 5: *FM_MAI_SIMULATION_AGENT parameters.*

We understand that the IM_AGENT_NAME is the name of the satellite SAP system (see 1.4.1) where you want to execute a "collector" class, provided in IM_COLLECTOR_CLASS, inside the function RunSimulation. This is where the analysis was a bit tricky, because we didn't find these collector classes in Solman... but we found them in the SMDAgent application.
To summarize, this P4 JNDI service handled by Solman is like a wrapper to launch java class, type "collector", in remote SMDAgent.
Again we must retrieve the involved jar files for this collector class in the SMDAgent this time. Using the same technique and also searching for logical string patterns we finally spot several packages named "agelet e2emai*". In one of these 8 packages, we found what looks to be collector's classes. We extract around 90 classes that can be potentially called from the initial JNDI service on Solman.
The following is a subset of the long list of collectors found. All of them, were found under the package *com.sap.smd.mai.collector*.

- HelloWorldCollector
- SAPPingHostCollector
- SimpleFileServiceCollector
- SimpleFileServiceCollector2
- SccCollector
- SAPControlWSCollector
- LicenseCollector
- FileServiceCollector
- FileContentScanCollector
- EventLogServiceCollector

After the firsts tests we concluded 4 facts:

- These classes were the correct ones to put in parameter IM_COLLECTOR_CLASS
- Each collector class has their specific parameters name stored in table input IM_CONTEXT_PARAMS, IM_INPUT_PARAMS or IM_METRIC_PARAMS.
- Some collectors implement authentication mechanisms. Which "break" the anonymous access until this point.
- Everything is blind. We never get the output from any collector.

At this point we start to study one-by-one each collector class trying to dig up potential security vulnerabilities. Some of the classes were very simple or inputless, others quite complicated or with several configuration prerequisites and dependency.

3.1.4.4.2 Findings

All three vulnerabilities detailed below are blind and can be exploited remotely without authentication:

SSRF SAPPingHTTPCollector

- **SAP Security Patch:** 3352058
- **CVE:** CVE-2023-36925

This collector's purpose is to perform "HTTP ping", so basically craft and send one HTTP request to a remote system using provided information from table parameters IM_METRIC_PARAMS. The entry point is the Solman through P4 but the request came from the SMDAgent as described in following flow:

$$\begin{tikzcd}
\textbf{Attacker} \arrow[r, "\text{P4}"] & \textbf{Solution Manager} \arrow[r, "\text{P4}"] & \textbf{Remote SAP system (SMDAgent)} \arrow[r, "\text{HTTP}"] & \textbf{Anywhere}
\end{tikzcd}$$

Figure 6: *SSRF SAPPingHTTPCollector flow.*

Attackers can specify target host, port, protocol type, http method, url path, payload for POST and the content-type. Despite being an issue it isself to access to access this collector anonymously, we found an arbitrary header injection that would allow an attacker to craft complex requests with credentials, cookies or custom SAP headers.

```
public String getHeaderContentType(IMetric metric) throws
    ConfigCollectorException {
String _me = "getHeaderContentType";
try {
String contentType =
    getMetricParamAsString(metric,"HEADER_CONTENT_TYPE",false);
return contentType;
}
```

Listing 2: *Metric parameter is not verified and therefore controlled by attacker*

SSRF SAPGrmgClassicCollector

- **SAP Security Patch:** 3348145
- **CVE:** CVE-2023-36921

The purpose of this collector class is similar to the SAPPingHTTPCollector (3.1.4.4.2). It crafts and sends one HTTP request to a remote system using information from a parameters table (IM_INPUT_PARAMS). The entry point is Solman but the request is performed from an SMDAgent system (see figure 6). The attacker can specify target host, port, protocol type, http method, url path and payload. Also it was possible to inject "\r\n" in the URL parameter, giving the attacker the possibility to add any arbitrary header in the request.

```
params.setValue("/aaa?
    HTTP/1.1"+"\r\n"
+ "Soapaction: " + "\r\n"
+ "User-Agent: Ona Agent"+"\r\n"
+ "Garbage: ",  "VALUE"  );
```

Listing 3: *Headers injection example.*

```
POST /aaa? HTTP/1.1
Soapaction:
User-Agent: Ona Agent
Garbage: HTTP/1.1
Host: somewhere:1234
Content-Length: 666
Content-Type: text/html
User-Agent: SAP HTTP CLIENT/6.40
```

Listing 4: *Request result received by listener.*

RCE EventLogServiceCollector

- **SAP Security Patch:** 3305369
- **CVE:** CVE-2023-27497

This collector's purpose is to gather entries stored in the Windows Event Log. To do this it uses the tool wevtutil.exe through a command line as shown in the following Listing.

```
protected static String wevtutil =
    "cmd /q /c " + windir +
    "\\system32\\wevtutil.exe qe
    AllEvents /rd:true /f:text
    /q:\"<QueryList>
```

Listing 5: *Injectable command line.*

One of the parameters stored in the IM_METRIC_PARAMS table is a variable contained in the final command line and controlled by the attacker. Even if the payload must avoid "\t\n\r\f" characters, it is possible to execute any OS command as user daaadm (owner of SMDAgent, see 1.4.1).

Attacker $\xrightarrow{\text{P4}}$ **Solution Manager** $\xrightarrow{\text{P4}}$ **Remote SAP system (SMDAgent) on Windows** $\xrightarrow{\text{cmd}}$ **OS**

Listing 6: *RCE flow.*

3.1.5 Impact

The components where most of the reported vulnerabilities were found, are shared by almost every SAP Java based solution. This includes highly critical and widely used solutions such as: Solution Manager, PI/PO, Enterprise Portal, CRM, and many more. As a consequence, it's possible to state that almost every company in the world with an SAP implementation will most likely be affected.
The following actions are possible to be carried out by an unauthenticated attacker leveraging only the vulnerabilities depicted in this section:

- Read / Exfiltration of arbitrary OS files
- Read / Exfiltration of arbitrary tables from the Database
- Registration of vulnerable system as Registered Server in arbitrary systems
- Execution of RFC Functions against vulnerable system
- Access to pre-configured (Jco) plain text passwords
- Leakage of technical information
- SSRF attacks
- Complete and Partial services disruption (DoS)
- Remote Code Execution (Windows only)

These actions could be later combined or chained with further attacks to increase the impact as shown in further sections.
The outcomes of this research project unveiled not only several critical vulnerabilities, but also strategies, processes and internal functionalities that were unknown by us. They allowed us to expand our knowledge and therefore keep pushing and improving SAP Security in a holistic way.

3.2. JNDI Reference injection in SAP Java based systems

Following sections will detail our journey from searching an unauthenticated endpoint up to finding a way to exploit JNDI reference injection in the SAP ecosystem. Even though our case of study was the SAP Enterprise Portal, the exploitation techniques found could be replicated in any other Java NetWeaver based solution.

3.2.1 Listing of unauthenticated endpoints

Automatic tools are a strong and efficient source to perform research. While analyzing hundreds of configuration files to filter out some of them based on specific properties could take weeks, automation could save valuable time and solve the same issue in a couple of seconds or minutes. The big effort is made only once: In the tool development phase. Furthermore, manual analysis may be more prompt to overlook details.
As part of previous research projects, an internal tool called Java Endpoint Analyzer (JEA)

was developed. This white-box tool will connect to a system and analyze every interesting configuration or deployment file. Once analyzed this tool will return the list of HTTP endpoints implemented inside the system, including specific properties (such as the requirement or not of authentication).
After running JEA against the latest Enterprise Portal version, the "NavigationServlet" application and "NavigationsWS" web service seemed to be exposed without authentication. Therefore, the analysis began.

3.2.2 Analysis of Navigation Service

Although being two different entry points and requiring different parameters, both the "NavigationServlet" and "NavigationWS" converged to the same place. They are part of the implementation of the concept of Navigation inside the Portal [31].
In a nutshell, the idea of this service is to manage the content that will be displayable by each user. Depending on the user and their authorizations, this service will create a specific navigation tree. Each node of this tree could be a specific content, a collection of them, a page, etc.
In order to lookup the content to be displayed, the Navigation implementation makes use of JNDI 1.3.

3.2.2.1 Connectors and Redirectors

As explained in SAP's documentation[32][33], both Connectors and Redirectors play a central role in Portal's Navigation service. Both are identified by specific prefixes. For instance, as mentioned in the quoted documentation, the PCD (Portal Content Directory) redirector and the ROLES connector are identified by "pcd:" and "ROLES:" prefixes respectively.
Technically speaking, when a specific node (object) name is being looked up, the Navigation Service will delegate the search to a specific class based on the prefix used. For example, when an object with name "pcd://xxxxx" is looked up, the class "com.sapportals.portal.pcd.pcm.roles.
RoleNavigationPcdRedirector" will handle it.
Based on our investigation, the following Connectors (C) and Redirectors(R) were found in a basic and standard implementation:

Type	Prefix	Class
R	pcd	com.sapportals.portal.pcd.pcm.roles.RoleNavigationPcdRedirector
R	pcdh	com.sapportals.portal.pcd.pcm.roles.RoleNavigationHashRedirector
R	TBN	com.sap.portal.tbn.redirector.TBNRedirector
R	OBN	com.sap.portal.obn.redirector.OBNRedirector
C	ROLES	com.sapportals.portal.pcd.pcm.roles.RoleNavigationConnector
C	gpn	com.sap.portal.ivs.global.navigation.connector.GPNavigationConnector
C	ModeledContent	com.sap.portal.modeling.preview.navigation.PreviewNavConnector
C	CollaborationConnector	com.sapportals.portal.pcd.pcm.roles.RoleNavigationConnector

Table 2: *Standard Redirectors and Connectors.*

3.2.3 JNDI arbitrary lookup injection points discovery

Our analysis started trying to find the function that was in charge of delegating the node name look up based on the provided prefix.
We found a candidate method which had the following arity:
$getNode(Hashtable\ env, String\ nodeName)$. By carefully understanding how it worked, it was possible to identify that this was what we were looking for. When nodeName has the "pcd://" prefix, $getNode()$ will delegate the search to

the redirector class RoleNavigationPcdRedirector by executing its *redirect*() method (as explained by SAP in [29]).
The actual code that *redirect*() was a standard JNDI lookup.

```
class RoleNavigationPcdRedirector{
  public redirect(String pcdURL,
      HashTable env){
    context obj =
        getPersistenceRootContext(
                        env
                    ).lookup(pcdURL);
  }
}
```

Listing 7: *redirect() pseudo code example.*

Once the JNDI context was created (in function *getPersistenceRootContext* and based on the content of env) the lookup using **nodeName** (pcdURL) was executed. Therefore, if an attacker could control the **nodeName** it will be able to perform a JNDI arbitrary lookup.
Turned out that, when a function named *getNavigationTree*() was called, some of its execution paths lead to *getNode*(). Additionally, the former method was exposed indirectly through the main two endpoints discovered using JEA 3.2.1 which meant that its execution did not require authentication. Moreover, the parameters supplied by the calling entity will end up as *getNode*()'s arguments and therefore as RoleNavigationPcdRedirector's *redirect*() arguments too.
As a summary, we finally discovered that any unauthenticated party, making use of NavigationServlet or NavigationWS, could have full control over the parameters of RoleNavigationPcdRedirector's *redirect*() and therefore execute a JNDI arbitrary lookup.
PCD's redirector class was not the only one vulnerable. Despite being the first finding, we then realized that other connectors and redirectors suffered the same consequences. In addition, it was also found that *getNavigationTree*() was not the only entrypoint but there was a second function named *getSelectedPathTree*() that could be also used to achieve the same results.

3.2.4 JNDI Reference injection Exploitation

So far we were only able to find JNDI lookup injection points. It was about time we moved to the exploitation phase.
As explained in section1.3 the JNDI architecture is based on several service providers (also known as resolvers): DNS, LDAP, RMI, CORBA, etc. Therefore, continuing with the PCD scenario, we tried to force an RMI connection to our own controlled server using as **nodeName** something similar to:

```
pcd://rmi://<host>:<port>/foo
```

The socket server configured in server <host>:<port> received the "JRMIK" magic bytes confirming that the vulnerability was present.
All techniques presented in Black Hat's 2017 JNDI injection talk[5] were a bit old (which make sense): Exploitation via Loading Classes remotely was prohibited as almost every JVM running an Enterprise Portal would have the necessary protections to block it. The technique illustrated by Michael from Veracode[26], using a reference of a class that is in the current classpath, was a good candidate to explore. Despite the fact that it was not possible to leverage it directly, as Michael's class was not present in our context, the idea of finding our own gadget inside SAP class path was promising.

3.2.4.1 Finding an SAP gadget

Continuing with the RMI vector tested and by live-debugging the process, it was possible to identify the path of execution that is followed whenever an JNDI reference is loaded.
Everything starts when the NamingManager object executes the *getObjectInstance*() method. Based on our investigation and also stated by Kai Ullrich in his document**??**, the builder object is never null and therefore the *getObjectInstance*() is executed. Is worth highlighting that the first argument received in this method (**refInfo**) is the actual reference that is served through the RMI server.

```
public Object
    getObjectInstance(Object ref,
    ...){
     ObjectFactoryBuilder b =
         getObjectFactoryBuilder();
     ObjectFactory f =
         b.createObjectFactory(ref);
     return
         f.getObjectInstance(ref,..);
}
```

Listing 8: *getObjectInstance() pseudo code example.*

There are three important behaviors worth to highlight inside the factory's *getObjectInstance*():

1. A "factory class name" is extracted from the reference object.
2. A function named *findObjectFactory* is called with the "factory name" obtained from 1 as argument. As the name suggests, this function will return an instance of a factory class based on the name provided. Additionally, the class must be loadable by thread loader. At the moment of returning that instance, it is cast to the class **ObjectFactory**.
3. Finally, the *getObjectInstance* of the factory instance obtained from 2 is called.

The following pseudo code illustrates the explained behaviors:

```
private getObjectInstance(Object refInfo, ...) {
     String factoryClassName = refInfo.getFactoryClassName();
     ObjectFactory fact = findObjectFactory(factoryClassName);
     return fact.getObjectInstance();
}

public ObjectFactory findObjectFactory(String factoryClassName) {
     Class factoryClass = Class.forName(factoryClassName,
         thread.getContextClassLoader());
     return  (ObjectFactory) factoryClass.newInstance();
}
```

Listing 9: *Necessary condintions pseudocode.*

As outcome, in order to be able to successfully instantiate a class through a JNDI reference, there are some requirements that it must accomplish:

1. Be castable to ObjectFactory
2. Implement a method named *getObjectInstance*()
3. Be interesting from an exploitation perspective. In other words, if the class just returns a null, it will not be usable.
4. It must be loadable by the thread.

First, a list larger than 60 candidate classes which fulfill the first two conditions was obtained. The 4th one was harder as it required dynamic analysis. Finally, it was possible to identify a class that met all the necessary conditions: **EJBObjectFactory**.

3.2.4.2 EJBObjectFactory

As can be inferred from the name, this class is devoted to searching and returning an Enterprise Java Beans (EJB)[34] object. Its *getObjectInstance*() method, after some initial checks, calls a function named *resolveReference*() (implemented in a separate class named **DefaultRemoteObjectFactory**) providing the JNDI Reference object as argument.

Based on the typical characteristics that an EJB object could have (interfaceType, appName,

beanName, etc), *resolveReference*() will gather some of this information out of the reference object provided. As part of its execution, it will try to interact with the EJB container. That is why it calls another function named *getEnterpriseBeansContainers*() implemented inside class **DefaultContainerRepository**.
getEnterpriseBeanContainers()'s main objective is to find the EJB objects that match with the provided characteristics. As a first approach, it gets all applications that match the application name provided by calling *getOrderedTargetApps*(). In order to perform the previously mentioned action, this latter function extracts the application name from the reference and searches inside the EJB container if there is any application that matches this name. However, before doing the actual search inside the EJB container, a function named *startApp*() with the provided application name is called.
There is no need to explain what *startApp*() does but it is worth remembering that until this point neither authentication or authorization were provided. In conclusion, it is possible to turn on arbitrary applications anonymously.
In summary, the following pseudo code will illustrate the path explained:

```
Object getObjectInstance(ref
    jndiRef){
  resolveReference(jndiRef);
}

Object resolveReference(ref
    jndiRef){
  getEnterpriseBeans(jndiRef);
}

Object getEnterpriseBeans(jndiRef){
  getOrderedApps(jndiRef);
}

Object getOrderedApps(ref jndiRef){
  ejbContainer = getEJBContainer();
  appName = jndiRef.getAppName();
  appName.startApp();  <-----
  ejbContainer.getApp(appName);
}
```

Listing 10: *Path to startApp()*

3.2.4.3 Exploitation illustration

Leveraging this attack vector, the entire exploitation path could be represented with the following illustration:

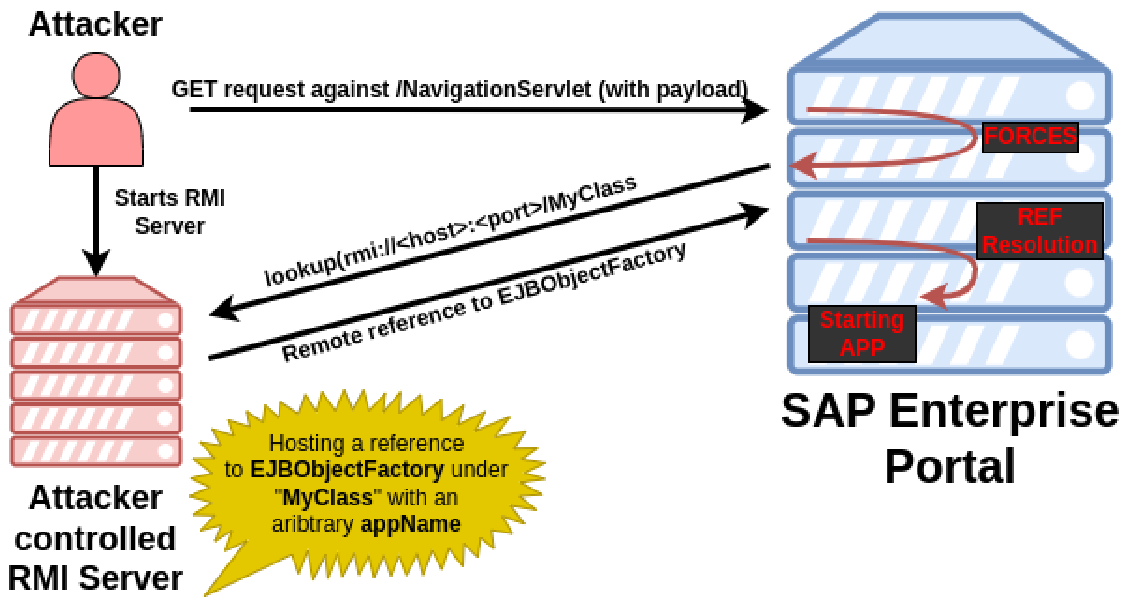
Attacker
GET request against /NavigationServlet (with payload)
Starts RMI Server
lookup(rmi://<host>:<port>/MyClass
Remote reference to EJBObjectFactory
FORCES
REF Resolution
Starting APP
SAP Enterprise Portal
Attacker controlled RMI Server
Hosting a reference to EJBObjectFactory under "MyClass" with an aribtrary appName

Figure 7: *JNDI Exploitation illustration flow.*

3.2.4.4 Findings and impact

- **SAP Security Patch:** 3289994
- **CVE:** CVE-2023-28761

This vulnerability was patched by SAP in April 2023 and despite having a medium CVSS of 6.5, it will be shown in future sections how this flaw could be leveraged to further compromise the targeted system.
The final impact of this exploitation will heavily depend on what actions could be carried out with the application that has been turned on. Nonetheless, it opens the door to considering vulnerabilities on stopped by default applications, increasing the attack surface. Furthermore, custom applications developed by SAP clients should also be considered as a point of attack.
It is known that, unfortunately, sometimes security by obscurity is a fact. Many times the decision of turning off vulnerable applications instead of patching them occurred. In that case, this new vector of attacks could be catastrophic. Based on our research, one application which was stopped by default could allow unauthenticated attackers to further compromise the targeted system. This chained attack will be introduced in further sections.

3.2.4.5 Exploitation variants

There are many ways to actually exploit the findings presented. We will present them divided in four groups:

1. **Initial entrypoint:** As explained in section 3.2.2 two different types of entry points were found: NavigationServlet (a servlet) and NavigationWS (a SOAP service). Both exposed through HTTP. One key difference, while the former could be exploited using GET requests, the latter will require POST ones.

2. **Main function:** As explained in section 3.2.3, despite the entrypoint used there are two different functions that could converge in the same piece of vulnerable code. Using either $getNavigationTree()$ or getSelectedPathTree() could allow a successful exploitation.

3. **Connectors/Redirectors:** As depicted in section 3.2.2.1 there are multiple JNDI Reference injection points depending on the class used. Several of them could lead to a successful exploitation.

4. **Resolvers:** As explained briefly at the beginning of section 3.2.4 the Java standard Naming package provides multiple Service Providers (LDAP, RMI, IIOP, etc). Each SP implements its own resolver. During our investigation we only focus on RMI exploitation, but we strongly believe that, at least, IIOP could also be used. LDAP exploitation seemed not possible in this context because of how SAP manages the JNDI NavigationPrincipal object.

3.2.4.6 JNDI reverseless exploitation

Usually when dealing with JNDI arbitrary lookup exploitation there is a need to create a reverse connection. The reason resides in the nature of how JNDI Service Providers (and its resolvers) work. In fact, in our own exploitation an RMI server exposing a JNDI Reference object is needed.
This inner characteristic helps firewalls or other security products to detect when attacks like this occur. Clearly depending on the context, most of the times a reverse connection thought RMI could sound suspicious. Specifically in the case of SAP systems, it will be hard to find a scenario where an RMI is actually benign.
While performing the analysis for finding our own gadget and how to exploit it, we came across the different kinds of resolvers that were possible to be used. By default, as explained in section 3.3, RMI, DNS, IIOP were candidates. However, SAP had specific classes that were implementing their own JNDIResolvers. One of these resolvers was the "EJB" JNDIResolver. As any other JNDI Resolver, first it will create a specific Context object. In this case its name was **JNDIejbResolverContext**. Once the Context is already defined, the next step will be

the execution of the *lookup*() function. After analyzing this latter method, it was discovered that it was performing two important and huge steps:

1. Creating the JNDI reference from the lookup string (by calling a function named *createReferenceFromSchemeLookupString*())
2. Calling *EJBObjectFactory.getObjectInstance*() using the created reference as argument.

As pseudo code, this could be expressed like this:

```
class JNDIejbResolverContext {
  Object lookup(jndiName){
    ref = createReferenceFromScheme
LookupString(jndiName);
    EJBObjectFactory.
        getObjectInstance(ref);
  }
}
```

Listing 11: *lookup pseudo code example.*

As the name suggests, *createReferenceFromSchemeLookupString* will grab every possible EJB characteristics (appName, interfaceType, etc) out of the **jndiName** and create a reference with that information.
So far, in order to have a successful exploitation, it was necessary to build a reference linked to the EJBObjectFactory class, containing specific EJB characteristics (**appName**) and serve it through the RMI server. This way, when the system performs the *lookup*() function it will load this class from the server and call the *getObjectInstance*() with the built reference. It is exactly what this EJB Resolver was doing.
In other words, by using the EJB resolver and crafting the EJB characteristics carefully inside the looked up name, it is possible to get rid of the reverse connection. As a consequence, applications may be turned on exploiting JNDI reference injections without the need of a reverse connection.
Finally, the new exploitation illustration could be:

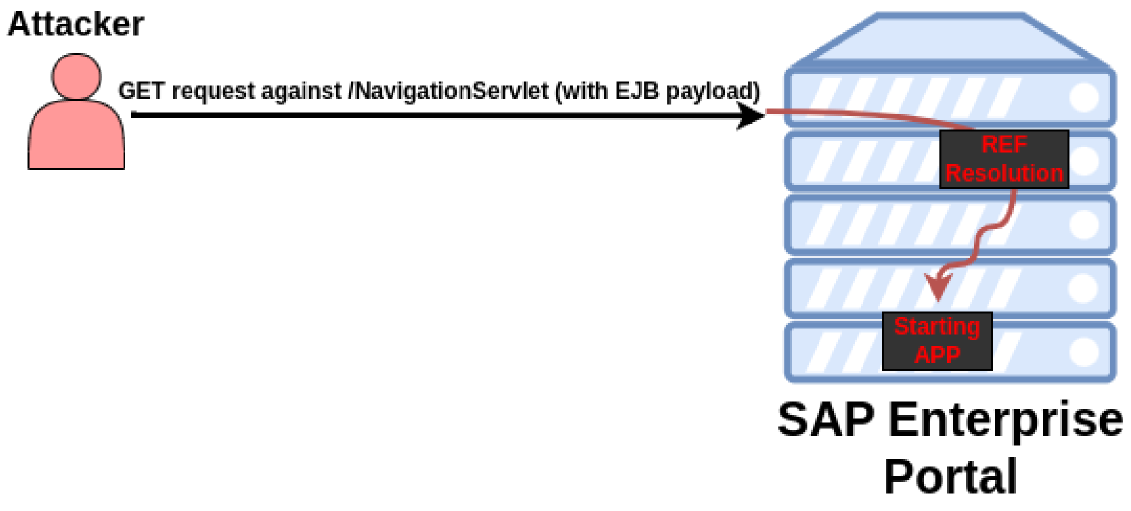

Figure 8: *Reverseless exploitation illustration.*

3.3. Start Service

3.3.1 Definition and tasks

This component used several binaries from the SAP Kernel as tools for administrators to perform a task directly as well as a service to receive administration request tasks from Web Service.
Two of these binaries are saposcol and saphostexec, which runs as a service under root or nt

authority/system. It is possible to interact with these binaries locally or remotely on port 1128 (http) and 1129 (https) through SOAP Services under the namespace SAPOsCol and SAPHostControl.

3.3.2 Findings

3.3.2.1 Buffer Overflow

- **SAP Security Patch:** 3275727
- **CVE:** CVE-2023-27498

Two of the SOAP Web methods under the namespace SAPOsCol, SendRequestAsync and SendRequest can be accessed locally without authentication or authorization. They don not correctly handle one parameter which leads to a memory corruption vulnerability, Stack Based Buffer Overflow, on saposcol Unix binary through these methods.
The exploitability of this vulnerability is quite easy due to the following 4 facts:

1. The crash is reliable and appears during a ret mnemonic with controlled RSP registry.
2. It is possible to leak the libc version using another SOAP method, *GetHwConfText*, in the same namespace also without authentication locally.
3. Using the same method *GetHwConfText*, it is also possible to bypass ASLR because the output of this method leaks the current RIP address of the currently running saposcol.
4. Only NX security feature is enabled on saposcol binary.

```
[*] '/usr/sap/hostctrl/exe/saposcol'
Arch:     amd64-64-little
RELRO:    No RELRO
Stack:    No canary found
NX:       NX enabled
PIE:      No PIE (0x400000)
```

Listing 12: *checksec output of Saposcol binary.*

Indeed this method, *GetHwConfText*, executes the sapsysinfo.sh[20] on OS side, as root, then sends back the whole output to the requester. One of these information is the result of the command line "cat /proc/[1-9]*/stat" (line 2654 of script) where it is possible to filter on "saposcol" and get the process status file content on the running saposcol. The following output is an example of a Process status file of the saposcol service.

```
2998 (saposcol) S 1 2998 2998 0 -1
1077944640 9008285 117854845 1 78 15505
37835 30224 54924 20 0 1 0 4916206973
27897856 946 18446744073709551615 4194304
6643541 140721156071872 140721156059176
140567129181712 0 65536 162533383 17920
18446744071680322219 0 0 17 0 0 0 111 0 0
7692288 7766592 24748032 140721156079070
140721156079147 140721156079147
140721156079577 0
```

The 28th number is the current address of the RIP register, 140721156071872 (0x7ffc328521c0) in the previous example. Having this information, knowing the libc version and also that most of the time the service waits on "__nanosleep_nocancel+7", it is possible to calculate offset for the libc base then perform a ret2libc type exploitation.

```
$ python3 libc_leaks.py -t saphost -p 1129
[+] Opened conn to saphost on port 1129: Done
[+] Receiving all data: Done (385.01KB)
[*] Closed conn to saphost port 1129
Target information : Linux
Libc version found : glibc 2.17
Saposcol leak : 0x7f8d7e9c6e10
Libc base     : 0x7f8d7e902000
Libc system   : 0x7f8d7ecde4c0
Libc /bin/sh  : 0x7f8d7ea88f89
Libc gadget   : 0x7f8d7e944fd8
Libc pop rdi  : 0x7f8d7e924ac8
Payload       : b'AAAAAAAAAAAAA<redacted>'
[+] Opened conn to saphost port 1129: Done
[+] Receiving all data: Done (871B)
[*] Closed conn to saphost port 1129
```

Listing 13: *Libc base leak example.*

3.3.2.2 OS command injection

- **SAP Security Patch:** 3285757
- **CVE:** CVE-2023-24523

The buffer overflow explained in section 3.3.2.1 was an important finding but only exploitable on Unix type systems and also could require payload adaptation depending on the libc version and target. We also found a more reliable and OS independent vulnerability which is an OS command injection in a method, *ConfigureOutsideDiscovery*, under namespace SAPHostControl. One of the parameters is used by the application in command "move/mv" to transfer a configuration file from temporary directory to the final directory. It was possible to inject arbitrary OS commands in this parameter. The command is executed as OS administrator user : root or nt authority/system.

```
[Thr 140226482845568] received CommandData:
     status=1,pid=4294967295,timeout=0,
cancellation_time=0,options=0,envhandling=0
[Thr 140226482845568]
     CommandManager::StartOSCommand: start
     /bin/sh
[Thr 140226482845568] Current environment
     will be used
[Thr 140226482845568] Environment:
[Thr 140226482845568]   XDG_SESSION_ID=1719
[Thr 140226482845568]   HOSTNAME=saphost
[Thr 140226482845568]   SHELL=/bin/bash
[Thr 140226482845568]   USER=root
...
[Thr 140226482845568]
     LD_LIBRARY_PATH=/usr/sap/hostctrl/exe
[Thr 140226482845568] PID 89259: root:
     Executing command "/bin/sh -c mv -f
     /usr/sap/hostctrl/work/tmpslddest.cfg
     /usr/sap/hostctrl/exe/config.d/
slddest_INJECTION.cfg"
```

Listing 14: *"dev_saphostexec" trace highlight the injectable command as root.*

4. Vulnerability Chaining

4.1. The importance of vulnerabilities sequences

It is rare to find one critical vulnerability to gain high privilege access remotely without authentication. When it happens, it's easy to recognize the danger behind it as usually the CVSS score is around 10. Under these circumstances, companies can react quickly to it.
However, there are more tricky-to-spot ways to achieve the same impact by linking several less critical vulnerabilities. This situation demands much more knowledge for companies in order to understand which flaws could be chained and how to prevent them. Furthermore, weaknesses exposed only to internal networks could also be underestimated when compared against those ones affecting Internet facing systems.
The following section will show how chaining attacks could be impactful in the SAP environment. Additionally, it will pursue raising awareness about why it is crucial to also take into account this "less" critical type of vulnerabilities.

4.2. Root RCE on SAP system through Solman P4

Because the LPE to root or nt authority/system vulnerability3.3.2.2 can be exploited locally without authentication through a SOAP Web Service, it is possible to trigger it from the SSRF and header injection vulnerability found in SAPPingHTTPCollector 3.1.4.4.2 on all SMDAgent managed by Solman. The unauthenticated access to the service FM_MAI_SIMULATION_AGENT3.1.4.4, through the P4 service, will be the only entry point required to compromise all SAP systems connected to the targeted Solman. The following illustration depicts the attack chain.

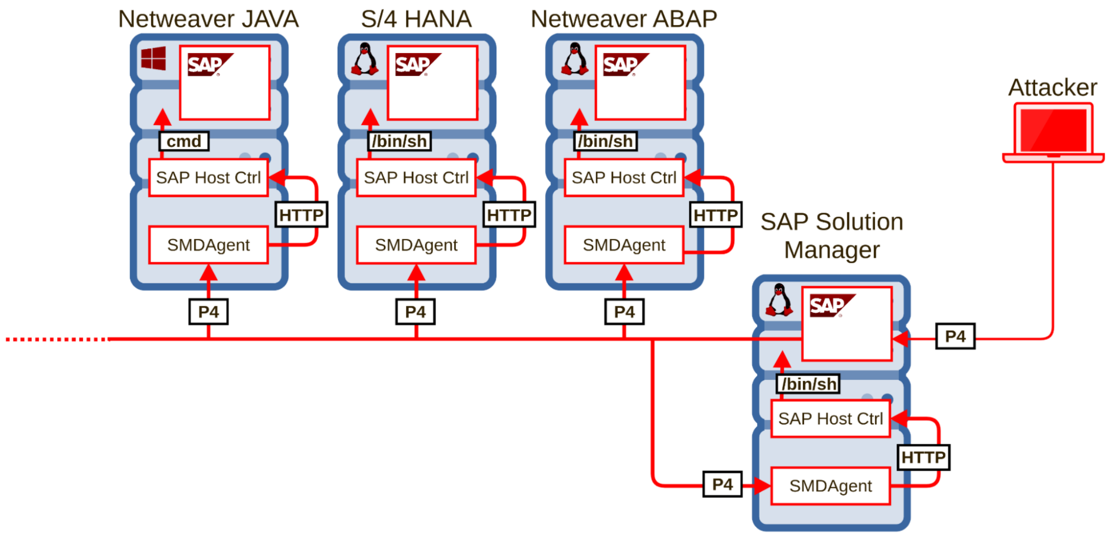

Figure 9: *Global view of P4 attack from Solman to all satellites systems.*

$\textbf{Attacker} \xrightarrow{\text{P4}} \textbf{Solution Manager} \xrightarrow{\text{P4}} \textbf{Remote SAP system (SMDAgent)} \xrightarrow{\text{HTTP}} \textbf{localhost:1128 (SAP Host Control)} \xrightarrow{\text{cmd}} \textbf{OS as root}$

4.3. P4 exploitation through HTTP

As already mentioned, although being a remote protocol, P4 is not usually exposed to external networks. This does not translate into more security, but it helps understanding that customers' exposure to Internet bad actors is usually low. Nonetheless, this is not totally true.
The P4 port is not the only way to interact using P4 with a Java-based system. Java NetWeaver comes with a stopped-by-default application called **tc je p4tunelling app**. As the name suggests, this application allows encapsulation of P4 traffic inside HTTP requests. It will be extremely rare to find a real scenario where this app is turned on. Nevertheless, because of our findings explained in section 3.2 it is possible to turn it on anonymously. Once turned on, unauthenticated bad actors could exploit P4 vulnerabilities shown in section 3.1.4 and perform tasks such as: Read all database tables, exfiltrate OS files including the Secure Store, leak of pre-configured plain passwords, execute RFC functions against the system, perform denial of service attacks, technical information disclosure, etc. As a consequence, the whole attack presented in section 3.1.4.3.2 could be finally abused through HTTP.
Enterprise Portal, the type of systems affected by this chain of attacks, are usually Internet facing. In other words, they are heavily threatened to be compromised as the level of exposition is wide.
The following illustrations depicts in 4 steps how this chain of vulnerabilities could give an anonymous bad actor the power to harm the system using P4 attacks:

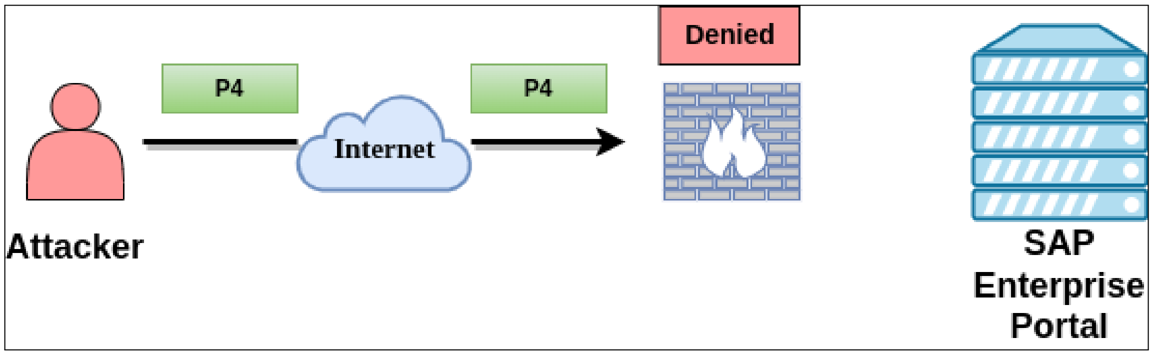

Figure 10: *Full HTTP + P4 illustration part 1. Attacker cannot reach P4 port of Enterprise Portal.*

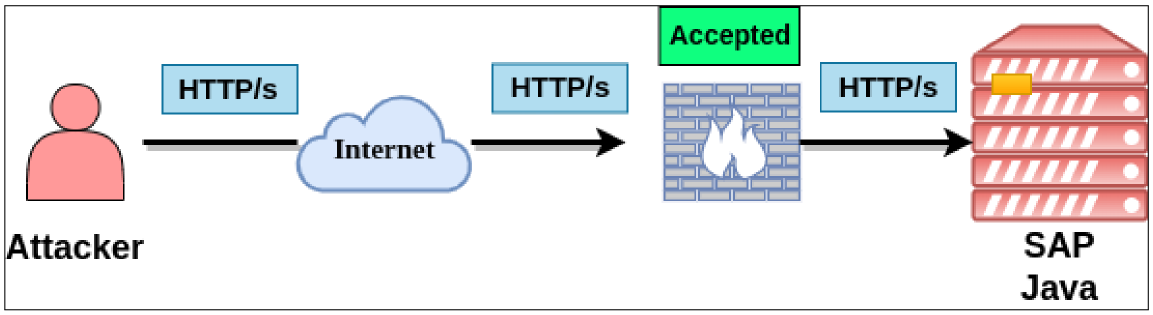

Figure 11: *Full HTTP + P4 illustration part 1. Attacker cannot reach P4 port of Enterprise Portal.*

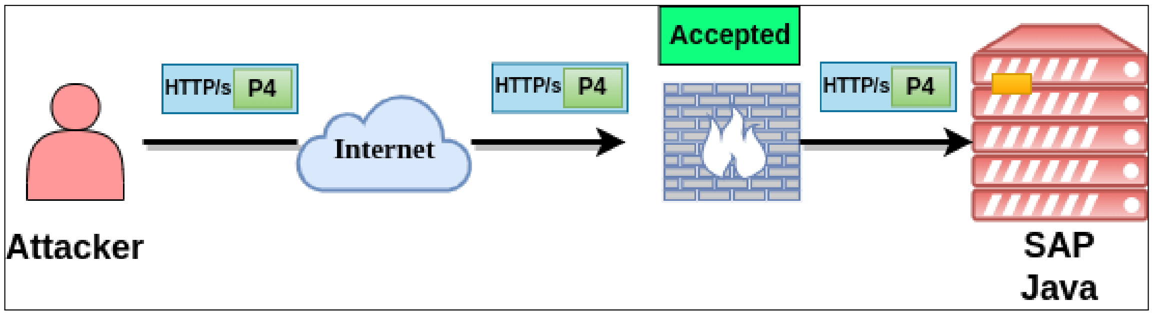

Figure 12: *Full HTTP + P4 illustration part 1. Attacker cannot reach P4 port of Enterprise Portal.*

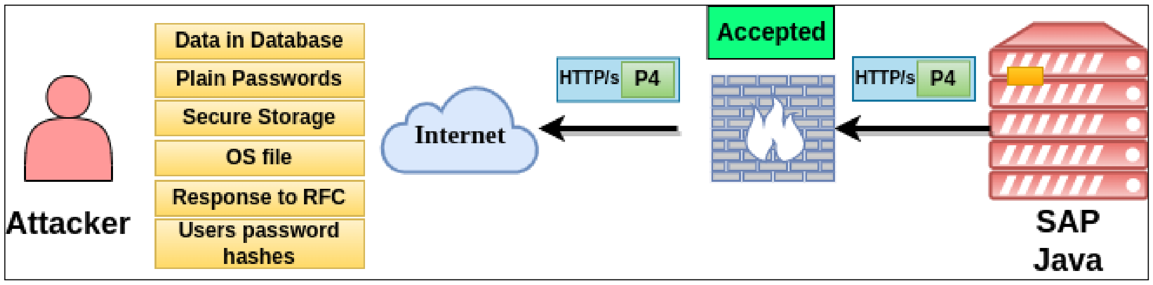

Figure 13: *Full HTTP + P4 illustration part 1. Attacker cannot reach P4 port of Enterprise Portal.*

4.4. From unauthenticated HTTP access to root: Combining 'em all!

It is possible to actually chain all the attacks and techniques developed in this whitepaper plus already known techniques, in order to get root privileges starting from HTTP access.
Step by step the full real world chain exploitation could work like this:

1. As shown in section 4.3, an unauthenticated attacker turns on the P4 tunneling app through HTTP using CVE-2023-28761.

2. As explained in 3.1.4.3.2 the attacker retrieves the Secure Store files using CVE-2023-23857.

3. Locally and using public exploit[35], the attacker decrypts Secure Store files which could contain several credentials like, but not limited to, the Master Key and database user and password.

4. The attacker uses these credentials to login into the SAP Portal. Our experience highlights that the combination of Administrator/<Master Key> often works.

5. With this high privilege access, the attacker may use the WS Navigation application, a SOAP client embedded inside the Netweaver Administration dashboard, to initiate communication with the internal SOAP service of the HostControl.

6. The attacker exploits CVE-2023-24523 through this access to and executes OS commands as root.

7. By nature the SAP Portal is largely connected to internal SAP Systems. This connection information is stored in table J2EE_CONFIGENTRY. The attacker could query this table, in order to gather information about the SAP Solution Manager system of the company.

8. From this SAP Portal root access attacker starts the same attack shown in section 4.2 to “root” all SAP systems inside the whole SAP implementation.

5. Staying protected

5.1. SAP Security Patches

From December 2022 to April 2023, SAP released 12 patches involving JNDI, P4 and Start Service. These patches provided protection against all vulnerabilities covered in this document.

CVE	CVSS	Patch	Description
CVE-2023-27497	10	3305369	Multiple vulnerabilities in SAP Diagnostics Agent (OSCommand Bridge and EventLogServiceCollector)
CVE-2023-23857	9.9	3252433	Improper Access Control in SAP NetWeaver AS for Java
CVE-2022-41272	9.9	3273480	Improper access control in SAP NetWeaver AS Java (User Defined Search)
CVE-2022-41271	9.4	3267780	Improper access control in SAP NetWeaver AS Java (Messaging System)
CVE-2023-0017	9.4	3268093	Improper access control in SAP NetWeaver AS for Java
CVE-2023-24523	8.8	3285757	Privilege Escalation vulnerability in SAP Host Agent (Start Service)
CVE-2023-36921	7.2	3348145	Header Injection in SAP Solution Manager
CVE-2023-36925	7.2	3352058	Unauthenticated blind SSRF in SAP Solution Manager
CVE-2023-27498	7.2	3275727	Memory Corruption vulnerability in SAPOSCOL
CVE-2023-28761	6.5	3289994	Missing Authentication check in SAP NetWeaver Enterprise Portal
CVE-2023-26460	5.3	3288096	Improper Access Control in SAP NetWeaver AS Java (Cache Management Service)
CVE-2023-24526	5.3	3288394	Improper Access Control in SAP NetWeaver AS Java (classload)
CVE-2023-27268	5.3	3288480	Improper Access Control in SAP NetWeaver AS Java (Object Analyzing Service)
CVE-2023-24527	5.3	3287784	Improper Access Control in SAP NetWeaver AS Java for Deploy Service

Table 3: *All findings related to these paper and their patches.*

5.2. P4 Protection

SAP devoted big effort into securing their systems against these vulnerabilities. As a consequence to help their customers, besides the patches for each vulnerability, they also published two extra notes: 3273729 and 3299806. These notes add extra information about P4 vulnerabilities and also share general recommendations in regards to how to be protected. The general recommendation is always to restrict and monitor P4 access as much as possible. Avoid exposing it to untrusted networks.
Furthermore, the possibility of using other security measures like monitoring or prevention systems (IPS, IDS, Firewalls, etc) are always encouraged. Restriction of RMI-like traffic could block and stop an attacker from performing their exploitation. Note 3299806 details the process on how to add specific rules to the ICM component in order to block access to the HTTP P4 tunneling application.
If the system's NetWeaver version is lower than 7.5 (which means that is no longer officially supported) most probably it will be vulnerable and it will not have any patch available. The only possibility in this case, will be to update

to NetWeaver 7.5

6. Conclusions

Throughout this document it was demonstrated how multiple research subjects affecting SAP systems could be combined in order to empower the impact of exploitation. As a side effect, it also highlighted the heterogeneity of the findings due to being present in areas which, at first glance, seemed completely unrelated: A proprietary network protocol, a local service and a standard and well known Java API called JNDI.

P4 seems to be a protocol that still needs to be analyzed. It was identified that several services were exposed through this protocol without enforcing any type of authentication mechanism. We believe that with our own efforts and previous ones done by other researchers, P4 will keep moving towards a more secure state. However, it seems to be still a long road to drive and research to perform.

In regards to JNDI we do believe that interesting results were achieved. Being able not only to find a new way of exploitation but also unveiling the internals of JNDI in SAP, could help future researchers with new ideas get into these topics in an easier manner.

Start Service is maybe one of the most important and critical components of SAP systems. As such, it keeps demonstrating that when vulnerabilities affecting it are found, the impact is automatically high. Additionally, due to being a component present almost in every implementation, its surface of attack is large. We suspect that research projects around this component will continue to appear and therefore customers should carefully monitor who interacts with it.

Protecting SAP systems is complex, and thus it requires time, effort and careful attention. Even though its security has been improving towards a more secure state during the last years, its nature makes it harder. Their highly hyperconnected landscape, where an SAP system must work with several other systems internally or over the internet, the numbers of involved applications, software or protocols are just a few reasons behind its complexity. Additionally, due to being related to the most core and critical company's business processes, they require several steps in the workflow each time an update or change is to be introduced. Therefore, the patching process becomes tough.

References

[1] https://help.sap.com/saphelp_ewm900/helpdata/en/48/295738a14558d8e10000000a421937/content.htm?no_cache=true

[2] https://docs.oracle.com/javase/7/docs/technotes/guides/rmi/

[3] https://docs.oracle.com/cd/E12531_01/tuxedo100/CORBA_ref/index.html

[4] https://docs.oracle.com/javase/8/docs/technotes/guides/jndi/index.html

[5] https://www.blackhat.com/docs/us-16/materials/us-16-Munoz-A-Journey-From-JNDI-LDAP-Manipulation pdf

[6] https://www.blackhat.com/docs/us-16/materials/us-16-Munoz-A-Journey-From-JNDI-LDAP-Manipulation pdf

[7] https://help.sap.com/docs/SAP_NETWEAVER_750/ff18034f08af4d7bb33894c2047c3b71/c6040065b1d34e75bdb21d2771e144f6.html?version=7.5.21

[8] https://community.sap.com/topics/portal/enterprise-portal

[9] https://me.sap.com/notes/1682613

[10] https://www.slideshare.net/sproctor05/onapsis-ekopartyerp-securityhowhackerscanopenthesafeandtal

[11] https://www.slideshare.net/sproctor05/dissecting-and-attacking-rmi-frameworksekoparty

[12] https://me.sap.com/notes/1819822

[13] https://me.sap.com/notes/2443673

[14] https://codewhitesec.blogspot.com/2017/05/sap-customers-make-sure-your-sapjvm-is.html

[15] https://github.com/codewhitesec/sap-p4-java-deserialization-exploit/blob/master/sapwn-disarmed.py

[16] https://me.sap.com/notes/2845377

[17] https://conference.hitb.org/hitblockdown002/materials/D2T1%20-%20SAP%20RCE%20-%20The%20Agent%20Who%20Spoke%20Too%20Much%20-%20Yvan%20Genuer.pdf

[18] https://codewhitesec.blogspot.com/2021/06/about-unsuccessful-quest-for.html

[19] https://me.sap.com/notes/3022422

[20] https://vulners.com/securityvulns/securityvulns:doc:25500

[21] https://me.sap.com/notes/1439348

[22] https://blog.c22.cc/2011/12/11/seczone-2011-sap-insecurity-slides/

[23] https://github.com/rapid7/metasploit-framework/tree/master/modules/auxiliary/scanner/sap

[24] https://me.sap.com/notes/2902645

[25] https://i.blackhat.com/USA-20/Wednesday/us-20-Artuso-An-Unauthenticated-Journey-To-Root-Pwning-
pdf

[26] https://www.veracode.com/blog/research/exploiting-jndi-injections-java

[27] https://help.sap.com/doc/saphelp_nw73ehp1/7.31.19/en-US/48/
2d9ba88aef4bb9e10000000a42189b/content.htm?no_cache=true

[28] https://help.sap.com/saphelp_nwce10/helpdata/en/44/3bd73865524903e10000000a1553f7/
content.htm?no_cache=true

[29] https://help.sap.com/saphelp_SNC700_ehp01/helpdata/en/cd/
14c93ec2f7df6ae10000000a114084/content.htm?no_cache=true

[30] https://github.com/erpscanteam/SecStoreDec

[31] https://help.sap.com/docs/SAP_NETWEAVER_750/f2f3f4b4543a4803b9023e8c31f1e72a/
4a1c7aa139e11b42e10000000a42189c.html

[32] https://help.sap.com/docs/SAP_NETWEAVER_750/f2f3f4b4543a4803b9023e8c31f1e72a/
4a2b31ce2c4d1d0fe10000000a42189c.html

[33] https://help.sap.com/docs/SAP_NETWEAVER_750/f2f3f4b4543a4803b9023e8c31f1e72a/
4a26f7aa8d6f0455e10000000a421937.html

[34] https://docs.oracle.com/cd/E24329_01/web.1211/e24446/ejbs.htm

[35] https://github.com/erpscanteam/SecStoreDec