

CS4.1真的有后门吗?

原创 先锋情报站 酒仙桥六号部队

2020-12-10原文

这是 酒仙桥六号部队 的第 125 篇文章。

全文共计2399个字，预计阅读时长7分钟。

前言

CS是什么？可能是某款历史久远的第一人称射击游戏，也可能是某个电影。由于法律原因我们这里并不指出CS是什么，但网络江湖上一直流传着他的传说。



江湖上流传最新版本4.1已经有一阵了，但是江湖上还传出各种小道消息说有后门/外联情况，导致小伙伴们都太不敢用啊。所以到底CS 4.1有没有后门，今天我们一起探索下~



皮皮虾， 我们开始吧~

一 入手CS4.1

我直接从江湖人士手中拿到了流传最火的CS4.1的破解版本。我入手的版本是号称中文版，里边的内容如下：

文件夹				2019-12-19 17:07
文件	1 KB	1 KB	文件	2018-09-07 03:05
Windows 批处理文件	1 KB	1 KB	Windows 批处理文件	2019-04-18 19:24
文件	1 KB	1 KB	文件	2018-09-07 03:05
Windows 批处理文件	1 KB	1 KB	Windows 批处理文件	2019-04-12 23:23
文件	1 KB	1 KB	文件	2018-09-06 15:05
AUTH 文件	1 KB	1 KB	AUTH 文件	2018-12-31 03:30
Windows 批处理文件	1 KB	1 KB	Windows 批处理文件	2019-06-22 19:40
Executable Jar File	2.3 KB	2.1 KB	Executable Jar File	2020-07-28 22:25
STORE 文件	2.3 KB	2.1 KB	STORE 文件	2019-06-14 19:17
Executable Jar File	767.3 KB	715.5 KB	Executable Jar File	2019-06-22 19:33
文件	1 KB	1 KB	文件	2018-09-07 03:05
Windows 批处理文件	1 KB	1 KB	Windows 批处理文件	2019-04-13 17:46
文件	1.8 KB	1 KB	文件	2017-05-23 21:23
Windows 批处理文件	2.0 KB	1 KB	Windows 批处理文件	2019-04-12 23:23

可以看到主程序 jar 包的修改日期为 2020-07-28，这个时间应该是破解重新打包的时间。我得到版本的 jar 文件 MD5 为：187a*****9613

我们知道jar包本质上就是zip包，所以我们修改文件后缀名为zip之后直接打开。查看各个文件修改的时间戳：

名称	压缩前	压缩后	类型	修改日期
..(上级目录)			文件夹	
[模糊]			文件夹	2020-06-24 19:47
[模糊]			文件夹	2020-06-24 19:47
[模糊]			文件夹	2020-06-24 19:47
[模糊]			文件夹	2020-06-24 19:47
[模糊]			文件夹	2020-06-24 19:47
[模糊]			文件夹	2020-06-24 19:47
[模糊]			文件夹	2020-06-24 19:47
[模糊]			文件夹	2020-06-24 19:46
[模糊]			文件夹	2020-06-24 19:46
[模糊]			文件夹	2020-06-24 19:46
[模糊]			文件夹	2020-06-24 19:46
[模糊]			文件夹	2020-06-24 19:46
[模糊]			文件夹	2020-06-24 19:46
[模糊]			文件夹	2020-06-24 19:46
[模糊]			文件夹	2020-06-24 19:46
[模糊]			文件夹	2020-06-24 19:46
[模糊]			文件夹	2020-06-24 19:46

可以看到大部分文件和文件夹的修改时间为2020-06-24，推测此时间为jar包生成时间，在经过一天后于2020-06-25对外发布。

经过详细排查发现有4个文件的时间戳跟其他文件明显不同：

beacon	255,336	2020/6/24 19:46:50
B [模糊].class	4,993	2020/7/27 23:29:30
common	227,496	2020/6/24 19:47:04
A [模糊].class	3,152	2020/7/27 23:24:05
S [模糊].class	1,572	2020/6/25 3:46:50
resources	5,874,223	2020/6/24 19:47:12
c [模糊].auth	16	2020/6/25 3:46:50

我们分别叫他们B文件、A文件、S文件和c文件。

这4个文件的修改代表对4.1破解流程的思路。笔者有幸从另一位江湖人士手中获取到4.0的原版，4.0到4.1版本应该只是修复了一些bug和增加了一些功能，所以我们通过对比原版和破解版之间的代码来看看破解者到底做了哪些改动。

二 修改验证

首当其冲的就是A文件，从命名上就可以看出来这个文件是进行验证授权的。

我们可以通过 Java Decompiler 来看查看源码，大部分验证逻辑都在构造函数中：

原版：

```
public Authorization() {
    String str = CommonUtils.canonicalize("c:\\...\\.auth");
    if (!new File(str).exists())
    try {
        File file = new File(getClass().getProtectionDomain().getCodeSource().getLocation().toURI());
        if (file.getName().toLowerCase().endsWith(".jar"))
            file = file.getParentFile();
        str = (new File(file, "c:\\...\\.auth")).getAbsolutePath();
    } catch (Exception exception) {
        MudgeSanity.logException("trouble locating auth file", exception, false);
    }
    byte[] arrayOfByte1 = CommonUtils.readFile(str);
    if (arrayOfByte1.length == 0) {
        this.error = "Could not read " + str;
        return;
    }
    AuthCrypto authCrypto = new AuthCrypto();
    byte[] arrayOfByte2 = authCrypto.decrypt(arrayOfByte1);
    if (arrayOfByte2.length == 0) {
        this.error = authCrypto.error();
        return;
    }
    try {
        DataParser dataParser = new DataParser(arrayOfByte2);
        dataParser.big();
        int i = dataParser.readInt();
        this.watermark = dataParser.readInt();
        byte b1 = dataParser.readByte();
        byte b2 = dataParser.readByte();
        byte[] arrayOfByte = dataParser.readBytes(b2);
    }
}
```

读取c:\...\.auth 认证文件

解密认证文件中的内容

将解密后的数据进行读取进行下一步使用

破解版：

```
public Authorization() {
    try {
        byte[] decrypt = {
            1, -55, -61, Byte.MAX_VALUE, 0, 0, 0, 0, 100, 1,
            0, 27, ..., 55,
            ..., -53, 6 };
        DataParser dataParser = new DataParser(decrypt);
        dataParser.big();
        int int1 = dataParser.readInt();
        this.watermark = dataParser.readInt();
        if (dataParser.readByte() < 41) {
            this.error = "Authorization file is not for C... S... 4.1+";
            return;
        }
        int i1 = dataParser.readByte();
        byte[] i = dataParser.readBytes(i1);
        byte[] bytes = dataParser.readBytes(dataParser.readByte());
        if (29999999 == int1) {
            this.validto = "forever";
            MudgeSanity.systemDetail("valid to", "perpetual");
        } else {
            this.validto = "20" + int1;
            MudgeSanity.systemDetail("valid to", CommonUtils.formatDateAny("##### d, YYYY", getExpirationDate()));
        }
        this.valid = true;
        MudgeSanity.systemDetail("id", this.watermark + "");
        SleevedResource.Setup(bytes);
    } catch (Exception ex2) {
        MudgeSanity.logException("auth file parsing", ex2, false);
    }
}
```

直接使用解密后的数据写在程序里

读取数据解析后的后续使用

熟悉CS破解的朋友可以看得出来，这里是CoolCat大佬的思路：

将正确的密钥文件解密后，直接将解密后的数据写到程序里进行下一步验证，并将原本读取c文件并解密的操作代码删除掉。

那既然在这里c文件已经不用读取了，那为啥在破解版本里还要有这个文件的存在呢？



这里就牵扯到CS的另一套产品控制机制，CS会将一些Windows下的操作封装成dll库，如：截屏、bypassuac、mimikatz等。并且为了控制用户需购买密钥，验证之后才可以合法使用，会将这些文件进行加密。使用时先读取文件，再用密钥文件来进行解密后使用。

而修改的文件S文件就是读取文件并进行解密的关键类，我们来看下原版的相关调用代码：

```
byte[] arrayOfByte2 = authCrypto.decrypt(arrayOfByte1);
if (arrayOfByte2.length == 0) {
    this.error = authCrypto.error();
    return;
}
try {
    DataParser dataParser = new DataParser(arrayOfByte2);
    dataParser.big();
    int i = dataParser.readInt();
    this.watermark = dataParser.readInt();
    byte b1 = dataParser.readByte();
    byte b2 = dataParser.readByte();
    byte[] arrayOfByte = dataParser.readBytes(b2);
    if (b1 < 40) {
        this.error = "Authorization file is not for [redacted] 4.0+";
        return;
    }
    if (29999999 == 1) {
        this.validto = "forever";
        MidgeSanity.systemDetail("valid to", "perpetual");
    } else {
        this.validto = "20" + i;
        CommonUtils.print_stat("Valid to is: " + this.validto + "");
        MidgeSanity.systemDetail("valid to", CommonUtils.formatDateAny("#### d, YYYY", getExpirationDate()));
    }
    this.valid = true;
    MidgeSanity.systemDetail("id", this.watermark + "");
    SleevdResource.Setup(arrayOfByte);
} catch (Exception exception) {
    MidgeSanity.logException("auth file parsing", exception, false);
}
}

public class S[...] source {
    private static SleevdResource singleton;
    private S[...] data = new S[...]();
    public static void Setup(byte[] paramArrayOfbyte) {
        singleton = new S[...]();
    }
    public static byte[] readResource(String paramString) {
        return singleton._readResource(paramString);
    }
    private S[...]e(byte[] paramArrayOfbyte) {
        this.data.registerKey(paramArrayOfbyte);
    }
    private byte[] _readResource(String paramString) {
        String str = CommonUtils.strrep(paramString, "resources/", "s[...]");
        byte[] arrayOfByte1 = CommonUtils.readResource(str);
        if (arrayOfByte1.length > 0) {
            long l = System.currentTimeMillis();
            return this.data.decrypt(arrayOfByte1);
        }
        byte[] arrayOfByte2 = CommonUtils.readResource(paramString);
        if (arrayOfByte2.length == 0) {
            CommonUtils.print_error("Could not find r[...] n[...]e: " + paramString + " [ERROR]");
        } else {
            CommonUtils.print_stat("Used internal resource: " + paramString);
        }
        return arrayOfByte2;
    }
}
```

解密密钥文件中的内容

读取其中的字节用于解密资源

将字节用于解密

将字节进行注册

读取资源时进行解密

可以很清晰的看出从A文件中调用S文件的过程。

我们再来看下破解版的代码：

```
public class S1 {
    private static S1 singleton;

    private S1 data;

    public static void Setup(byte[] array) {
        singleton = new S1(CommonUtils.readResource("resources/c/...auth"));
    }

    public static byte[] readResource(String s) {
        return singleton._readResource(s);
    }

    private S1(byte[] array) {
        (this.data = new S1()).registerKey(array);
    }

    private byte[] _readResource(String s) {
        byte[] resource = CommonUtils.readResource(CommonUtils.strrep(s, "resources/", "sleeve/"));
        if (resource.length > 0) {
            System.currentTimeMillis();
            return this.data.decrypt(resource);
        }
        byte[] resource2 = CommonUtils.readResource(s);
        if (resource2.length == 0) {
            CommonUtils.print_error("Could not find : " + s + " [ERROR]");
        } else {
            CommonUtils.print_stat("Used internal resource: " + s);
        }
        return resource2;
    }
}
```


直接读取c...auth中的内容进行注册

读取资源时解密，跟原版一样


我们看到破解版中直接从c文件中读取数据直接作为字节进行注册，而原版读取了c文件中的内容还需要解密之后读取字节才可以注册。所以虽然原版和破解版都包含该文件，但是文件内容和意义大不相同，我们从文件大小也可以看出差别：

文件夹中已包含一个相同名称的文件
C:\tools\CS\C...S...4.1\c...auth

是否用解压出来的文件：

 16 字节 ← **破解版16字节**
修改时间: 2020年6月25日, 3:46:50 (较新)

替换已有文件：

 256 字节 (较大) ← **原版有256字节**
修改时间: 2018年12月31日, 3:30:40

所以这些文件的修改都是针对验证进行了破解，并没有加入后门。

三 exit暗桩

最后一个修改文件为B文件，我们来看下B文件，这个文件是TeamServer的核心文件：

```
public B (Resources paramResources) {
    this.resources = paramResources;
    this.socks = new B (this);
    this.data = new B (this);
    this.appid = getClass().getProtectionDomain().getCodeSource().getLocation().getPath();
    this.data.shouldPad(isPaddingRequired());
    this.parsers.add(new MimikatzCredentials(paramResources));
    this.parsers.add(new DcSyncCredentials(paramResources));
    this.parsers.add(new MimikatzDcSyncCSV(paramResources));
    this.parsers.add(new ScanResults(paramResources));
    this.parsers.add(new NetViewResults(paramResources));
}

protected boolean isPaddingRequired() {
    boolean bool = false;
    try {
        ZipFile zipFile = new ZipFile(this.appid);
        Enumeration< extends ZipEntry> enumeration = zipFile.entries();
        while (enumeration.hasMoreElements()) {
            ZipEntry zipEntry = enumeration.nextElement();
            long l1 = CommonUtils.checksum(zipEntry.getName());
            long l2 = zipEntry.getName().length();
            if (l1 == 75L && l2 == 21L) {
                if (zipEntry.getCrc() != 1661186542L && zipEntry.getCrc() != 1309838793L)
                    bool = true;
                continue;
            }
            if (l1 == 144L && l2 == 20L) {
                if (zipEntry.getCrc() != 1701567278L && zipEntry.getCrc() != 3030496099L && zipEntry.getCrc() != 1514902380L)
                    bool = true;
                continue;
            }
            if (l1 == 62L && l2 == 26L) {
                if (zipEntry.getCrc() != 4015977862L && zipEntry.getCrc() != 2741377737L)
                    bool = true;
                continue;
            }
            if (l1 == 224L && l2 == 23L && zipEntry.getCrc() != 1056789379L && zipEntry.getCrc() != 2460238802L)
                bool = true;
        }
        zipFile.close();
    } catch (Throwable throwable) {}
    return bool;
}
```

获取当前jar包路径

判断是否需要做exit暗桩

将当前jar包用zip解析

遍历jar包中的文件

关键文件进行CRC校验，如果有修改返回true，则需要填充暗桩

this.data.shouldPad调用的函数就在我们的修改类B文件里，我们来看下原版和破解版的区别：

原版：


```
public void shouldPad(boolean paramBoolean) {
    this.shouldPad = paramBoolean;
    this.when = System.currentTimeMillis() + 180000L;
}

public void task(String paramString, byte[] paramArrayOfbyte) {
    synchronized (this) {
        List<byte[]> list = getQueue(paramString);
        if (this.shouldPad && System.currentTimeMillis() > this.when) {
            CommandBuilder commandBuilder = new CommandBuilder();
            commandBuilder.setCommand(3);
            commandBuilder.addString(paramArrayOfbyte);
            list.add(commandBuilder.build());
        } else {
            list.add(paramArrayOfbyte);
        }
        this.tasked.add(paramString);
    }
}
```

如果判断需要增加暗桩就会将when参数置为当前时间+30分钟

30分钟后会exit，此处即为exit暗桩

破解版：

```
public void shouldPad(boolean shouldPad) {
    this.shouldPad = false;
    this.when = System.currentTimeMillis() + 180000L;
}

public void task(String s, byte[] array) {
    synchronized (this) {
        List<byte[]> queue = getQueue(s);
        if (this.shouldPad && System.currentTimeMillis() > this.when) {
            CommandBuilder commandBuilder = new CommandBuilder();
            commandBuilder.setCommand(3);
            commandBuilder.addString(array);
            queue.add(commandBuilder.build());
        } else {
            queue.add(array);
        }
        this.tasked.add(s);
    }
}
```

直接将shouldPad写死为false

判断为false不会增加exit命令，即去掉exit暗桩

我们可以看到破解版直接将shouldPad置为了false，这样无论校验结果如何都不会增加exit命令，从而达到去掉exit暗桩的效果。

所以B文件的修改是为了去除作者进行产品控制的exit暗桩。

四 外联问题

笔者用的方法比较笨：将服务器端和被控端跑在纯净的操作系统中，操作系统中将更新关闭并且只安装了WireShark来进行长期抓包。

经过测试，在运行了24小时之后查看WireShark报文中并未发现可疑外联。

当然即便是使用这种笨办法测试，也不能完全100% 确信没有外联。

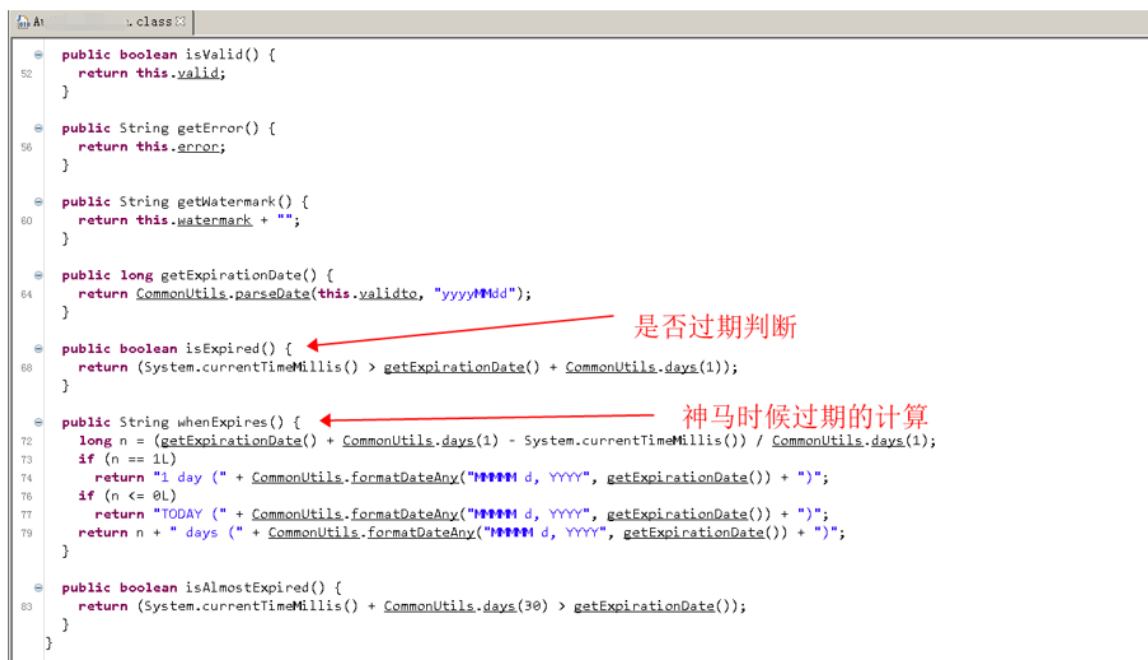
五 其他问题

目前看来CS4.1已经可以列入我们的武器库中了，但是其实还有其他问题需要我们来自己动手修改。

过期问题

虽然我们在破解验证的时候通过大佬的分享使用了正版密钥的数据，但这些数据中包含了授权时间信息，网上有资料说需要将过期的判断改一下。

我们来看下破解版中的此部分代码，此部分代码在A文件中：



```
public boolean isValid() {
    return this.valid;
}

public String getError() {
    return this.error;
}

public String getWatermark() {
    return this.watermark + "";
}

public long getExpirationDate() {
    return CommonUtils.parseDate(this.validto, "yyyyMMdd");
}

public boolean isExpired() {
    return (System.currentTimeMillis() > getExpirationDate() + CommonUtils.days(1));
}

public String whenExpires() {
    long n = (getExpirationDate() + CommonUtils.days(1) - System.currentTimeMillis()) / CommonUtils.days(1);
    if (n == 1L)
        return "1 day (" + CommonUtils.formatDateAny("MMMM d, YYYY", getExpirationDate()) + ")";
    if (n <= 0L)
        return "TODAY (" + CommonUtils.formatDateAny("MMMM d, YYYY", getExpirationDate()) + ")";
    return n + " days (" + CommonUtils.formatDateAny("MMMM d, YYYY", getExpirationDate()) + ")";
}

public boolean isAlmostExpired() {
    return (System.currentTimeMillis() + CommonUtils.days(30) > getExpirationDate());
}
```

是否过期判断

神马时候过期的计算

代码中并没有处理过期判断，我们可以将判断直接返回false即可。

但是我们其实也可以不做任何修改，也不会有影响：

```
try {
    byte[] decrypt = {
        1, -55, -61, Byte.MAX_VALUE, 0, 0, 0, 0, 100, 1,
        0, ..., ..., -53, 6 };
    DataParser dataParser = new DataParser(decrypt);
    dataParser.big();
    int int1 = dataParser.readInt();
    this.watermark = dataParser.readInt();
    if (dataParser.readByte() < 41) {
        this.error = "Authorization file is not for C... = 4.1*";
        return;
    }
    int i1 = dataParser.readByte();
    byte[] i = dataParser.readBytes(i1);
    byte[] bytes = dataParser.readBytes(dataParser.readByte());
    if (29999999 == int1) {
        this.validto = "forever";
        MudgeSanity.systemDetail("valid to", "perpetual");
    } else {
        this.validto = "20" + int1;
        MudgeSanity.systemDetail("valid to", CommonUtils.formatDateAny("#### d, YYYY", getExpirationDate()));
    }
    this.valid = true;
    MudgeSanity.systemDetail("id", this.watermark + "");
    SleevedResource.Setup(bytes);
} catch (Exception ex2) {
    MudgeSanity.logException("auth file parsing", ex2, false);
}

public boolean isPerpetual() {
    return "forever".equals(this.validto);
}
```

通过判断是否为29999999来确定是否为永久版

破解版中的密钥字节经过测试计算为29999999，即为永久版。

```
public static void main(String[] args) {
    // TODO Auto-generated method stub
    final byte[] decrypt = { 1, -55, -61, 127, 0, 0, 0, 0, 100, 1, 0, 27, -27, -66, 82, -58, 37, 92, 51, 85, -114,
        -118, 28, -74, 103, -53, 6 };
    DataInputStream content = new DataInputStream(new ByteArrayInputStream(decrypt));

    byte[] bdata = new byte[8];
    ByteBuffer buffer = ByteBuffer.wrap(bdata).order(ByteOrder.LITTLE_ENDIAN);
    buffer.order(ByteOrder.BIG_ENDIAN);

    buffer.clear();
    Arrays.fill(buffer.array(), (byte) 0);

    try {
        content.read(bdata, 0, 4);
    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
    int int1 = buffer.getInt(0);

    System.out.print(int1);
}
```

密钥字节读取后输出结果为29999999即为永久版

而在所有调用中只有一个文件有用到这些判断：

```
public class License {
    public static void checkLicenseGUI(Authorization paramAuthorization) {
        if (!paramAuthorization.isValid()) {
            CommonsUtils.print_error("Your authorization file is not valid: " + paramAuthorization.getError());
            JOptionPane.showMessageDialog(null, "Your authorization file is not valid.\n" + paramAuthorization.getError(), null, 0);
            System.exit(0);
        }
        if (paramAuthorization.isPerpetual()) ← 首先判断是否为永久版，是永久版则跳出检查
            return;
        if (paramAuthorization.isExpired()) { ← 非永久版才会进行是否过期的验证判断
            CommonsUtils.print_error("Your license is expired. Please contact : ");
            JOptionPane.showMessageDialog(null, "Your license is expired.\nPlease contact ");
            System.exit(0);
        }
        if (paramAuthorization.isAlmostExpired()) {
            CommonsUtils.print_warn("Your license expires in " + paramAuthorization.whenExpires() + ". Email ");
            JOptionPane.showMessageDialog(null, "Your license expires in " + paramAuthorization.whenExpires() + "\nEmail : ");
        }
    }

    public static boolean isTrial() {
        return false;
    }

    public static void checkLicenseConsole(Authorization paramAuthorization) {
        if (!paramAuthorization.isValid()) {
            CommonsUtils.print_error("Your authorization file is not valid: " + paramAuthorization.getError());
            System.exit(0);
        }
        if (paramAuthorization.isPerpetual())
            return;
        if (paramAuthorization.isExpired()) {
            CommonsUtils.print_error("Your license is expired. Please contact : ");
            System.exit(0);
        }
    }
}
```

非永久版才会进行过期的验证判断，我们既然已经是永久版，就可以不用管过期问题。

WaterMark

watermark即为水印，此参数中在A文件会进行赋值。

```
protected boolean valid = false;

public A n() {
    try {
        byte[] decrypt = {
            1, -55, -61, Byte.MAX_VALUE, 0, 0, 0, 0, 100, 1,
            0, ..., -53, 6 };
        DataParser dataParser = new DataParser(decrypt);
        dataParser.big();
        int int1 = dataParser.readInt();
        this.watermark = dataParser.readInt(); ← 对watermark进行赋值
    }
}
```

经过测试发现此参数赋值为0：

```
14 final byte[] decrypt = { 1, -55, -61, 127, 0, 0, 0, 0, 100, 1, 0, 27, -27, -66, 82, -58, 37, 92, 51, 85,
15 -114, -118, 28, -74, 103, -53, 6 };
16
17 DataInputStream content = new DataInputStream(new ByteArrayInputStream(decrypt));
18
19 byte[] bdata = new byte[8];
20 ByteBuffer buffer = ByteBuffer.wrap(bdata).order(ByteOrder.LITTLE_ENDIAN);
21 buffer.order(ByteOrder.BIG_ENDIAN);
22
23 buffer.clear();
24 Arrays.fill(buffer.array(), (byte) 0);
25
26 content.read(bdata, 0, 4);
27 int int1 = buffer.getInt(0);
28
29 buffer.clear();
30 Arrays.fill(buffer.array(), (byte) 0);
31
32 content.read(bdata, 0, 4);
33 int1 = buffer.getInt(0);
34
35 System.out.print(int1);
36 } catch (IOException e) {
37 // TODO Auto-generated catch block
38 e.printStackTrace();

```

经过计算watermark数值为0

为0就会有一个很不好的后果，从变量命名上我们也可以获知一二，就是会增加水印。但是CS增加的水印并不会显示在界面UI上，而是会在生成的shellcode中：

```
44
45 public String pad(final String s, final int n) {
46     final StringBuffer sb = new StringBuffer();
47     sb.append(s);
48     while (sb.length() < n) {
49         if (this.watermark == 0) {
50             sb.append("50!P%QAP[4\\PZX54(P^7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*\u0000");
51         }
52         else {
53             sb.append((char)CommonUtils.rand(255));
54         }
55     }
56     return sb.toString().substring(0, n);
57 }
58
59 public String getWatermark() {
60     final Packer packer = new Packer();
61     packer.addInt(this.watermark);
62     return CommonUtils.bString(packer.getBytes());
63 }
64

```

watermark为0时会增加一段神秘代码

有人会觉得哪有啥的，不就在shellcode里增加了一段谁都看不懂的字符串，搞不好还进行混淆了呢，这有什么的~



但是非常抱歉，这段神秘代码并不是一段普通的字符串！

这段字符串是EICAR（欧洲计算机防病毒研究所）开发的一种测试代码，本意是用于测试设备的防病毒能力。它本质上不是病毒，也并不包含任何程序代码。

最简单的测试方式是将这段字符串复制到一个空白的“新建文本文档.txt”中并保存，用杀软对此文件进行扫描会立即告警。



共发现风险项目1个，建议立即处理

全部忽略

立即处理

扫描已完成

风险项目	状态
<input checked="" type="checkbox"/>  新建文本文档.txt 引擎测试程序 TEST/AVEngTestFile!EICAR	待处理 详情

风险详情

病毒类型: 引擎测试程序 (TEST/AVEngTestFile!EICAR)

病毒描述: 此程序不包含恶意代码，仅用来检验反病毒引擎是否正常工作。

风险路径:  新建文本文档.txt

处理建议: 立即处理

[打开文件路径](#) [信任文件](#)

所以一旦某个文件包含或者流量中包含这段代码，就会被大部分杀软和流量检测设备检测到并发出告警。

我们的解决方案就是将watermark在验证阶段赋值为非0，并且将这段神秘代码在CS中删除。

经过反编译修改并重新打包后，我们终于可以愉快的玩耍啦~

终于可以愉快玩耍了



六 总结

所有4个修改的地方我们都一一看过了，外联测试我们用笨方法也测试过了，这个版本还是比较干净的，但人心险恶江湖叵测，随着流传版本越来越多，很有可能在传播过程中有人放进自己的代码。当然也不排除有官方作者藏得更深的暗桩，所以我们可能需要更多的细心和耐心还需要更深的代码功底，才可能挖掘出更深层的东西。





知其黑 守其白

分享知识盛宴，闲聊大院趣事，备好酒肉等你



长按二维码关注 酒仙桥六号部队

精选留言

用户设置不下载评论