SAPIENZA
UNIVERSITÀ DI ROMA

Security in Software Applications Proj 2

Alex Parri - 2140961 - Master's Degree in Cybersecurity

A.Y. 2024/25

Abstract

This is the report for the **second project** of the Security in Software Applications course directed by Daniele Friolo for the Academic Year 24/25 for the Master's Degree in **Cybersecurity** at Sapienza University of Rome. In this homework, the goal was to experiment with **fuzz testing**, a form of software dynamic analysis.

Specifically, it was asked to use the AFL tool to **test** the image manipulation software ImageMagick, and summarize the obtained results. It was decided to use AFL++ as it a **superior, modern and maintained** fork of the former.

The **hardware** utilized for testing is Ryzen 5800X 8-Core 16-Thread @ 4.850GHz CPU with clang v18.1.3, AFL++ v4.32c in Ubuntu 24.04.02 LTS x86_64 and 16GB of RAM.

AFL++

AFL++ (American Fuzzy Lop ++) is a modern, improved fork of the original AFL (American Fuzzy Lop) tool. It is a **powerful fuzz tester** for finding bugs and vulnerabilities automatically, relying on inputting **carefully mutated** random inputs in a target program to try and trigger unexpected behavior such as crashes and hangs, which can then be analyzed to try and find **vulnerabilities** within the program.

Setting up the tool

Setting up can be **quite overwhelming** for the faint of heart. This is also because it allows a significant degree of **customizeability** and alternatives from instrumenting, to preparing, managing the fuzzing campaigns to triaging results.

Instrumentation

First things first we **instrument** the target program, this is done to prepare it to be fuzzed efficiently and consequently improve efficacy in inputs yielding unexpected behavior. This is done by compiling it with a **special compiler** (`afl-cc`) and in this specific case in **LTO mode** (`afl-clang-lto`)

```
cd ImageMagick-6.7.7-10

# set up AFL++ environment
export CC=afl-clang-lto
export CXX=afl-clang-lto++
```

```
# sanitizer (optional)
export AFL_USE_$sanitizer=1

# build from source
./configure --disable-shared --enable-static \
    --without-magick-plus-plus --without-perl \
    --without-x --without-opencl
make -j$(nproc)
```

The tool also allows to use **sanitizers** (`AFL_USE_$sanitizer=1`) to find bugs that not necessarily result in a crash. If selected, only **one instance** at the time of the fuzzer is enough, using multiple concurrent instances would slow the execution down for without gain.

`$sanitizer`	Target
ASAN	memory corruption vulnerabilities
MSAN	read accesses to uninitialized memory
UBSAN	actions with undefined behaviour
CFISAN	instances where the control flow is illegal
TSAN	thread race conditions
LSAN	memory leaks

Alternatively it is also possible to do **black box fuzzing**, but it is a much better option to exploit the fact that the source code is **publicly available** for everyone to download and play around with.

Setting up the corpus

In order to begin its operation, the fuzzer requires a **corpus**: few small input files the application considers as valid. It was decided to randomly generate noisy ones using **ImageMagick itself** to try to maximise mutation chance

```
mkdir afl-tests/png-in afl-tests/png-out
code generate_corpus.sh
for i in {1..5}; do
  SIZE=$((RANDOM % 40 + 10))    # between 10x10 and 50x50
  ImageMagick-6.7.7-10/utilities/convert -size ${SIZE}x${SIZE}
  ↪  plasma:fractal afl-tests/png-in/noise_$i.png
done
```

Running `afl-fuzz`

Once all of the above is complete we run the fuzzer and let the **magic happen**

```
afl-fuzz -i afl-tests/png-in \
    -o afl-tests/png-out \
    -- $command
```

If **no sanitizer** is being used then it is possible to **concurrently run** multiple instances for better fuzzing efficiency. One of the possibilities is launching a **master fuzzer** (`-M`) on the first CPU core

```
# running master fuzzer on first CPU core
taskset -c 0 afl-fuzz -M masterfuzzer \
    -i afl-tests/png-in \
    -o afl-tests/png-out \
    -- $command
```

The master coordinates all the work with its **slave fuzzers** (`-S`), each ran on its own terminal handled by its own **CPU core** (`-c $i`)

```
# running $i-th slave fuzzer on CPU core $i
taskset -c $i afl-fuzz -S slavefuzzer$i \
    -i afl-tests/png-in \
    -o afl-tests/png-out \
    -- $command
```

Since fuzzing very usually takes a while to **bear fruit**, AFL++ also supports resuming of a previous **ongoing session** with the `-i-` flag while keeping the same output and command options

```
afl-fuzz -i- \
    -o afl-tests/png-out \
    -- $command
```

The option in question is also valid for the **master-slave configuration**

```
taskset -c 0 afl-fuzz -i- -M masterfuzzer \
      -o afl-tests/png-out \
      -- $command

taskset -c $i afl-fuzz -i- -S slavefuzzer$i \
      -o afl-tests/png-out \
      -- $command
```

The variable `$command` depends on the **specific command** it was decided to fuzz.

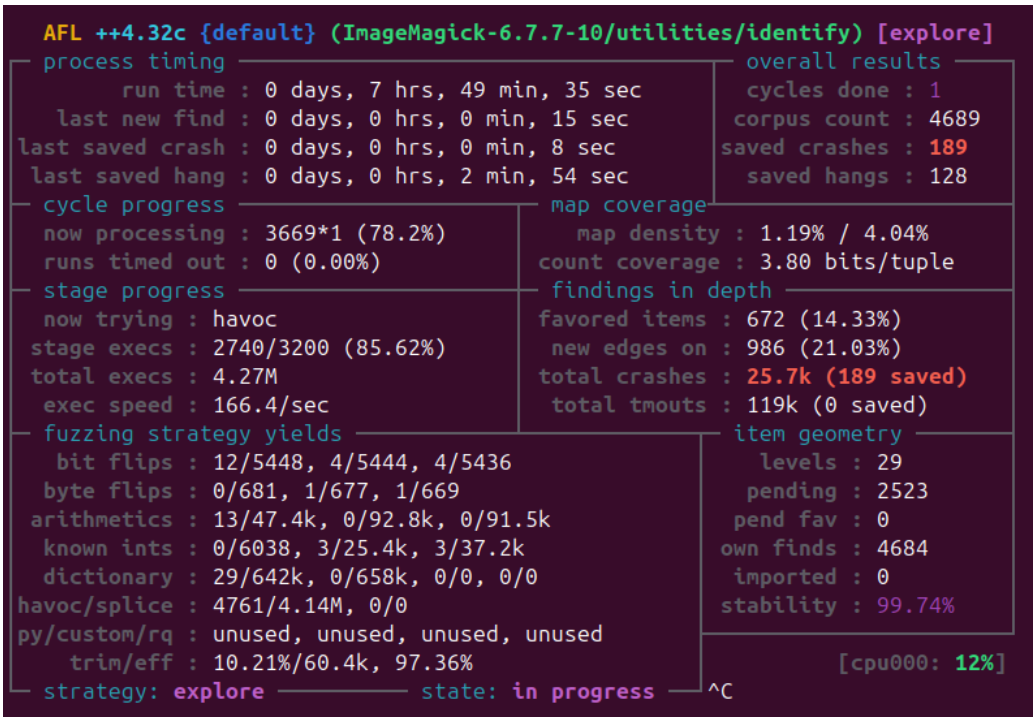
Fuzzing results

The purpose of this homework was to play around with ImageMagick. It was decided to use v6.7.7-10 to **more easily** find crashes and hangs, as testing for a more modern version would imply spending significantly more time fuzzing to obtain a **comparable** number of results.

The **chosen** command to mess around with is `identify`, whose **purpose** is reported from the official documentation as: *"The `identify` command describes the format and characteristics of one or more image files."* The `$command` variable that was previously mentioned would be assigned as follows

```
command = ImageMagick-6.7.7-10/utilities/identify @@
```

In this context it was decided to use the **address sanitizer** (`AFL_USE_ASAN=1`) and thus running a single master fuzzer, whose result of is shown in the following figure



The fuzzer ran for **almost 8 hours** (7h49m), having attempted over 4.27 million overall **mutations** with a final developed corpus of 4689 (up from the starting 5) and an overall number of detected flaws amounting to 256. These are very promising numbers, signifying that the fuzzer was **set up properly** and that it beared its fruits.

Note: for brevity reasons no hangs will be analyzed, as crashes are usually more interesting to analyze and dig deeper in.

Triaging the crashes

The tool reported a number of crashes that is **too great** for any patience-armed individual to be analyzed one-by-one, therefore it was decided to **deduplicate** them based off of the top of the stack trace (#0) obtained by inputting each crash file to the `identify` command.

```
# deduplicate_crashes.sh

# create directory if not there
mkdir -p "$UNIQUE_DIR"
declare -A seen_traces

# save current ASLR setting
ASLR_SETTING=$(cat /proc/sys/kernel/randomize_va_space)

# temporarily disable ASLR
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space > /dev/null
echo "Temporarily disabled ASLR (was $ASLR_SETTING)"

# loop through all the crashes
for crash in afl-tests/png-out/default/crashes/id:*; do
    [[ -f "$crash" ]] || continue

    # execute the line and filter the top line
    trace_line=$(./ImageMagick-6.7.7-10/utilities/identify "$crash" 2>&1 |
    ↪  grep -m1 '^[[:space:]]*#0')

    # hash it for fast comparison
    hash=$(echo "$trace_line" | md5sum | awk '{print $1}')

    # check if already exists
    if [[ -z "${seen_traces[$hash]}" ]]; then
        echo "Unique crash: $crash ($trace_line)"
        cp "$crash" unique_crashes/
        seen_traces["$hash"]=1
    fi
done

# restore ASLR settings
echo "$ASLR_SETTING" | sudo tee /proc/sys/kernel/randomize_va_space >
↪  /dev/null
echo "Restored ASLR to $ASLR_SETTING"
```

The above script performs exactly what was explained. The result is a much **more contained** 19 crash files with unique top of the stack trace, this is performed given the reasonable assumption that different crashes dying in the same exact function and line are due to the **same exact reasons**.

Note: ASLR had to be temporarily disabled during its execution, otherwise every stack trace would be **unique** due to address randomization.

Only **one** crash per type will be analyzed, again for brevity reasons.

AddressSanitizer: Requested allocation size exceeds maximum

This vulnerability is already known from CVE-2016-8862 with CVSS3 8.8 HIGH

```
    #0 0x56c6e8db1803 in malloc()
    #1 0x56c6e91424f0 in AcquireMagickMemory()
    #2 0x56c6e91424f0 in AcquireQuantumMemory()
    #3 0x56c6e91424f0 in AcquireQuantumPixels()
    #4 0x56c6e91424f0 in SetQuantumDepth()
```

This error is thrown by `AcquireMagickMemory()` in the following line

```
#if !defined(MAGICKCORE_ZERO_CONFIGURATION_SUPPORT)
    memory=memory_methods.acquire_memory_handler(size == 0 ? 1UL : size);
#else
```

The `acquire_memory_handler(size)` function is a wrapper for `malloc()` trying to allocate a **very large** amount of memory (`size = 0x71446e912aaaa99`) higher than the maximum of `0x10000000000`. The `size` variable is from `AcquireQuantumMemory()` as follows

```
size=count*quantum;
```

By using **pwndbg** and setting a breakpoint on `AcquireQuantumMemory()` and letting the program crash we can see its **last instance** as follows

```
Breakpoint 1.123, AcquireQuantumMemory (count=510110349776300697,
↪  quantum=1) at magick/memory.c:461
```

Which is called by the following line in `AcquireQuantumPixels()`

```
quantum_info->pixels[i]=(unsigned char *) AcquireQuantumMemory(extent+1,
sizeof(**quantum_info->pixels));
```

Meaning that `extent = 510110349776300696`, which is passed to it from the `SetQuantumDepth()` function as **second argument** below

```
status=AcquireQuantumPixels(quantum_info,(6+quantum_info->pad)*image-
↪  >columns*((quantum_info->depth+7)/8));
```

In conclusion we have that

```
size = (6+quantum_info->pad)*image->columns*((quantum_info->depth+7)/8)
// allows for CMYKA + RLE byte + pad
```

Signifying that the program is trying to allocate `extent` number of bytes to place the image in memory through `malloc()`, but fails to do so due to **malformations** in the fuzzed file.

AddressSanitizer: heap-use-after-free

This vulnerability is already known from CVE-2016-10051 with CVSS3 7.8 HIGH

```
    #0 0x56c399c76563 in EOFBlob()
    #1 0x56c39a2c3914 in ReadPWPImage()
    #2 0x56c399c8f050 in ReadImage()
    #3 0x56c399fdfbf8 in ReadStream()
    #4 0x56c399c8db73 in PingImage()
    ...
```

The use-after-free is done with `image` at the following line in `EOFBlob()`

```
assert(image->signature == MagickSignature);
```

Which was allocated by `AcquireMagickMemory()` this way

```
#if !defined(MAGICKCORE_ZERO_CONFIGURATION_SUPPORT)
    memory=memory_methods.acquire_memory_handler(size == 0 ? 1UL : size);
#else
```

Previously freed in `ReadPWPImage()` at this line

```
for(;;) {
    ...
}
pwp_image=DestroyImage(pwp_image);
```

What happened is that **some condition** caused the above infinite `for` loop to `break` and mistakenly free the image, likely due to **malformations** in the fuzzed input.

AddressSanitizer: heap-buffer-overflow

This vulnerability is already known from CVE-2008-1097 with CVSS3 9.8 CRITICAL

```
    #0 0x5dba909dc427 in ReadPCXImage()
    #1 0x5dba9043d050 in ReadImage()
    #2 0x5dba9078dbf8 in ReadStream()
    #3 0x5dba9043bb73 in PingImage()
    #4 0x5dba9043c385 in PingImages()
    ...
```

This occurs as the program recognizes the crash image as PCX format and tries to convert it to **pixel packets**, but instead overflows the `scanline` buffer in `ReadPCXImage()` while incrementing it

```
// Uncompress image data
p=pcx_pixels+(y*pcx_info.bytes_per_line*pcx_info.planes);
r=scanline;
...
for (x=0; x < ((ssize_t) image->columns-3); x+=4) {
    *r++=(*p >> 6) & 0x3;
    ...
}
```

Both `scanline` and `pcx_pixels` are buffers obtained by calling `AcquireQuantumMemory()` with arguments the inner details of the crash file, which may be **malformed** due to it being fuzzed

```
pcx_packets=(size_t)image->rows*pcx_info.bytes_per_line*pcx_info.planes;

pcx_pixels=(unsigned char*)AcquireQuantumMemory(pcx_packets,
↪  sizeof(*pcx_pixels));

scanline=(unsigned char*)AcquireQuantumMemory(MagickMax(image->columns,
↪  pcx_info.bytes_per_line),MagickMax(8,pcx_info.planes)*sizeof(*scanline));
```

The vulnerability was addressed in newer versions by an `ImproperImageHeader` exception

```
if ((image->columns==0) || (image->rows==0) ||
↪  (pcx_info.bits_per_pixel==0))
    ThrowReaderException(CorruptImageError,"ImproperImageHeader");
```

Which prevents the execution from **reaching** the overflow.

AddressSanitizer: strcpy-param-overlap

This vulnerability doesn't have a CVE associated to it and doesn't occur in newer versions

```
    #0 0x618db072f05d in strcpy()
    #1 0x618db0d3dbf7 in convertHTMLcodes()
    #2 0x618db0d39c4e in parse8BIM()
    #3 0x618db0d329c8 in ReadMETAImage()
    #4 0x618db083d050 in ReadImage()
    ...
```

This error is thrown inside `convertHTMLcodes()` in the following `strcpy()`

```
for (i=0; i < codes; i++) {
if (html_codes[i].len <= len)
    if (stringnicmp(s,html_codes[i].code,(size_t)html_codes[i].len) == 0)
    {
        (void)strcpy(s+1,s+html_codes[i].len);
        *s = html_codes[i].val;
        return html_codes[i].len-1;
    }
}
```

This occurs because ImageMagick recognizes the file as inside an **embedded profile** of type `8BIM`, the signature for Photoshop Image Resource Block (a data structure to store information inside images), and is trying to parse it through the `parse8BIM()` function

```
char *s = &token[next-1];
len -= (ssize_t)convertHTMLcodes(s,(int)strlen(s));
```

Specifically it recognizes it of type `8BIMTEXT`, which is the ASCII representation of the `8BIM` format

```
if (LocaleCompare(image_info->magick,"8BIMTEXT") == 0) {
    length=(size_t) parse8BIM(image, buff);
}
```

The overlap then happens due to **malformations** in the `token` array.