Uninitialized Memory Disclosures in Web Applications

Balint Varga-Perke
Uninitialized Memory Disclosures in Web Applications

GIAC (GWAPT) Gold Certification

Author: Bálint Varga-Perke, vpbalint@silentsignal.hu
Advisor: Mohammed Haron
Accepted: January 31st 2020

Abstract

Since modern web applications are implemented in memory-safe languages, vulnerabilities arising from erroneous memory handling are often overlooked during web application testing. Recent research however shows that some memory-unsafe parsers are still popular members of the software supply chain, reanimating old bug classes. Disclosure of uninitialized memory is one of these bug classes that poses unique challenges for black- and white-box testing and vulnerability research as well. This paper will give an overview on the bug class and public cases of such vulnerabilities affecting web applications. Challenges, and possible approaches of black-box detection will be discussed in detail. Since the processing model of the affected software has a determining effect on the impact of memory disclosures, the effect of the vulnerabilities will be assessed against multiple application platforms.
1. Introduction

During web application assessments classes of memory safety issues are typically not considered (Grossman, 2006), mainly because current web application platforms, such as Java Servlet or ASP.NET are memory-safe (Open Web Application Security Project, 2010). However, for certain features it can look beneficial to sacrifice safety for higher expected performance (Neumann, 2003) or for the ease of reusing well-known libraries. Thus, web applications can be affected by the bugs of memory unsafe code in their dependencies.

This paper focuses on vulnerabilities arising from the use of uninitialized memory, a subclass of memory safety issues (Azevedo de Amorim, 2017). These problems occur when a piece of memory that is in invalid state (is neither cleared nor filled with data intended for the user) is returned to the user (Miller, Modeling the exploitation of memory safety vulnerabilities, 2012). In multi-user applications the returned contents may include sensitive data belonging to other users, or the application.

Public vulnerability reports suggest (see Appendix 4.1 and References by Chris Evans), that in case of web applications, uninitialized memory disclosures mostly occur during parsing of images. To be able to reliably identify such issues, it’s important to develop test methods, that allow automatic detection of memory disclosure. This paper describes the process of developing such methods from the theoretical background to experimental evaluation of results. A publicly available test environment is provided for others to validate and improve the presented methods. This test environment is also used to show examples of how sensitive data can be extracted from web applications of different runtimes, running on servers with different processing models.

2. Uninitialized Memory Disclosures in Web Applications

2.1. Web Application Testing and Memory Safety

A memory access is memory-safe if “it is within bounds and refers to an object that is in a valid state” (Miller, Modeling the exploitation of memory safety vulnerabilities, 2012). Memory-safety of a program can be defined as the assertion that in all possible executions of a program, all memory access is memory-safe (Hosfelt, 2019).

Bálint Varga-Perke, vpbalint@silentsignal.hu
From memory safety issues, the widely adopted OWASP Testing Guide includes steps for Buffer Overflow Testing (OTG-INVPAL-014) (Open Web Application Security Project, 2016). Tests for buffer overflow are rarely executed in practice (Grossman, 2006) because of multiple reasons. First, memory-safe application platforms can be easily identified, that outrules the targeted issues on the vast majority of the interfaces. Second, as the seemingly endless flow of memory corruption vulnerabilities (Miller, 2019) show – identifying memory safety issues is a hard problem, even with access to source code.
Third, impactful exploitation (typically aimed at code execution) of memory-corruption flaws is hard, esp. in remote settings, due to the widespread use of exploit mitigations, such as Address Space Layout Randomization (ASLR— an exploit mitigation technique based on the randomization of the virtual address space of a process). On top of this, the Testing Guide only covers buffer overflows, but not other types of memory safety flaws, such as buffer overreads or use of uninitialized memory.

This paper focuses on vulnerabilities that allow uninitialized memory disclosure (hereby referred to as UMD). To describe these issues in Miller’s model, we can work backwards from the expected result of the vulnerability to the underlying Flaws: the Vulnerabilities of interest would result in Information Disclosure. This would be caused by Read type Violations, where the Content of the affected memory parameter would be Uninitialized. The Base and Displacement properties would be likely Unknown to an attacker, while the Extent may be controlled. As we will see in sections 2.2 and 2.5, these

Bálint Varga-Perke, vpbalint@silentsignal.hu
violations most often arise from **Boundary errors** (esp. out-of-bounds reads) or **Uninitialized use** types of **Flaws**.

Memory-safe languages typically prevent uninitialized uses by automatically initializing allocated memory (Azevedo de Amorim, 2017). Although some safe languages provide unsafe allocators (such as `Buffer.allocUnsafe()` of Node.js), these are easier to check for using static code analysis, thanks to the well-defined API’s. Out-of-bounds access is typically prevented by automatically applied bounds checks by the compiler or the runtime.

Since web applications are typically implemented in languages providing such systematic protections, UMD flaws can be typically introduced in unsafe components of libraries or the runtime itself. Some web application libraries function as simple wrappers around lower level implementations. While the high-level library can be memory safe, the wrapped program may not. Today’s runtime systems and interpreters are typically implemented in C or C++, which are unsafe languages, and have a higher chance to contain memory-safety flaws. As a result, when looking for memory-safety issues in web applications, we have to look into the software supply chain: components that are outside of control of the application developer.

### 2.2. History

Exploitation of uninitialized variables has been explored since at least 2005 (Flake, 2006), while effects of such errors have been likely observed since the dawn of computing.

As of today, the most notorious case of UMD vulnerabilities has been CVE-2014-0160, aka. Heartbleed. Heartbleed was the result of a bug in the OpenSSL library, that caused a buffer overread when processing special SSL/TLS “Heartbeat” packets. Since the flaw existed at the lower-level protocol parsers of web servers, the vulnerability allowed to extract application data, and even private keys from the memory of the affected servers (Sullivan, 2014).

Uninitialized memory disclosure in the context of web applications was popularized by Chris Evans with a series of disclosures during 2017. Evans first proved
the absence of ASLR at dropbox.com and box.com (Evans, 2017). He exploited a known vulnerability of the ImageMagick library to leak server memory and find pointers in them. Later, Evans used a previously unknown memory disclosure vulnerability to prove that ASLR was enabled on the targeted program modules (Evans, Proving Box.com fixed ASLR via ImageMagick uninitialized zlib stream buffer, 2017). Evans also managed to leak private data of Yahoo! users by using two previously unknown uninitialized memory disclosures in ImageMagick. The first vulnerability allowed him to access images sent by other users via Yahoo! Mail (Evans, *bleed continues: 18 byte file, $14k bounty, for leaking private Yahoo! Mail images, 2017). The second vulnerability allowed him to access authentication secrets and filesystem paths from the server (Evans, *bleed, more powerful: dumping Yahoo! authentication secrets with an out-of-bounds read, 2017).

Following these disclosures several vulnerabilities were identified on popular websites and published through their bug bounty programs. All of the six cases found published on the HackerOne platform are related to ImageMagick GIF parsing. In five cases the reports were associated with CVE-2017-15277, a vulnerability found by Emil Lerner, who also wrote an exploit generator and reported one of the vulnerabilities (#251732) (Lerner, 2017).

The following table summarizes the discussed vulnerabilities, and the cause:

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>Uninitialized Memory Use</th>
<th>Out-of-Bounds Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>CESA-2017-0002 / CVE-2017-9098 / YB1</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>CESA-2017-0003</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>CVE-2015-8958 / YB2</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>CVE-2017-15277</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

Although all above examples involved ImageMagick, assuming that similar problems only affect a single software component would be naive. The size of the

Bálint Varga-Perke, vpbalint@silentsignal.hu
discussed sample is too small to draw such conclusions, and another exploitable library will be shown in later sections too. More generally, image processing can be considered a good candidate feature to target in order to find UMD’s in web applications. The first reason for this is that processing images from untrusted sources is a common web application feature. The second reason is that due to the complexity of image (or any digital signal) processing, it makes sense to reuse already available components, which are typically implemented in unsafe languages. The complexity of the task also means, that these old components will likely have many flaws. Aside of the complexity, safe languages can make the implementation harder (e.g.: lack of unsigned integer types in Java) and can introduce performance penalty. Although lately considerable development effort has been put into performance improvements (e.g. JIT compilation, GC optimizations), historical experience still affects design decisions of signal processing implementations

It should be noted that other kinds of features may also be affected by similar vulnerabilities, although no such case was found during the writing of this paper.

2.3. Exploitability

This section discusses the factors that can motivate attackers (not) to exploit uninitialized memory disclosures in the context of web applications. Chris Evans discussed such factors before (Evans, *bleed continues: 18 byte file, $14k bounty, for leaking private Yahoo! Mail images, 2017) – in this section only general considerations are described while providing more background information.

2.3.1. Processing models

Modern operating systems implement virtual memory management, that prevents processes from directly accessing each others data. This means, that if the process in which the vulnerable component runs is only accessible by a single application user, then data of other users can’t be improperly disclosed. For example, if image conversion is performed by spawning fresh converter processes (with clean address spaces), then any user can only access its own image data. In contrast, if the main request handler process only invokes a library function for conversion, previously processed data may be accessible.

Bálint Varga-Perke, vpbalint@silentsignal.hu
While using fresh processes can be an effective mitigation against user data leaks, other sensitive data can still be disclosed this way. Such data includes pointers (for ASLR bypass), full filesystem paths, exact version information, etc. Process creation also bears the risk of introducing command injection vulnerabilities.

2.3.2. Heap implementation

The user-mode memory-allocator usually behaves differently depending on the size of the requested allocation. Smaller allocations are often served by reusing memory already requested from the operating system, but freed by the application. User-space memory allocators typically request a large piece of virtual memory, and use its smaller pieces on demand, sparing syscalls. Note that this is the reason that accessing previously freed memory doesn’t result in access violation – the accessed virtual address is valid, only the user-space allocator treats it as free.

Larger allocations may be served by requesting brand new memory pages from the kernel. Since allocating new virtual memory for a process would risk cross-process memory leaks, the operating system erases memory before giving it to a user process. Thus, accessing contents of uninitialized memory contents is only possible if the target memory buffer is reused instead of freshly allocated, i.e. when the allocation is small enough.

2.3.3. Accuracy

The accuracy of the disclosed data can be severely reduced by “noise” or compression.

“Noise” occurs when parts of sensitive data is overwritten by other unimportant or incomprehensible data during the normal use of process memory. For example, some bytes of a session cookie can be overwritten by the Content-Type information of another HTTP request buffer. Noise completely destroys parts of the leaked data, but this is not necessarily catastrophic for an attacker. For example, shorter parts of authentication secrets can be practically brute-forced, or partial file contents can be extracted from a damaged archive file.

Bálint Varga-Perke, vpbalint@silentsignal.hu
Lossy compression can discard information from the uninitialized buffer. This factor is especially common in case of vulnerabilities of image processing software, when uninitialized memory is treated as the raw image data. While lossy encoding is irreversible, many times inputs can be constructed in a way that doesn’t result in a compression operation. For example, developers may spare the thumbnailing step, if an uploaded image is small enough. If compression can’t be avoided, we can treat the output as some noise was applied to it. In this case the parts of data affected by the noise are not completely destroyed, as compression algorithms are designed so that they only distort information below some acceptable level. In this case we have ways aside of brute-force to increase the chances of accurate recovery. In his 2017. publication, Evans crafted his input so that it was encoded with a very limited set of symbols: a 1 bit-per-pixel image. Recovery of distorted symbols in the output became easier, because distorted symbols from the output could be more easily reclassified into two groups than into several. Further methods of information recovery are algorithm-dependent and are the subject of rate-distortion theory, that is out of the scope of this paper.

2.3.4. Reliability

The layout of process memory is randomized due to inherent factors (e.g. software version, heap state) and intentional measures, such as ASLR. This makes reliable exploitation of memory corruptions hard, especially in remote settings. Failed exploitation attempts may crash the target service by either causing memory access violation or moving the process into another unrecoverable state. Crashes can render the target system unavailable or alert administrators. Even in possession of precise knowledge about the target process state, exploitation can fail due to external factors such as service load (e.g. when grooming the heap by performing appropriately sized allocations and deallocations in order to create a desired sequence of properly sized allocated and free chunks) or OS-wide hardenings.

In case of UMD vulnerabilities read type violations occur, that don’t affect the further computation performed by the program. This prevents corrupting the program state in an unintended way and also spares the trouble of transforming the initial violation into a more useful (e.g. execution type) one. While out-of-bounds reads can result in
access violations, since user-space memory allocators generally operate on a relatively large preallocated space, the risk of this is manageable. For example, reading 64 kB after the end of an allocated buffer has about 1.24% chance of going over the end of a 5 MB preallocated heap area. Uninitialized memory access is by definition in-bounds, so it can’t result in access violation.

Even if exploitation would pose challenges in terms of targeting sensitive data, this stability allows increasing the success rate of the simplest methods by repeated execution. The power of these “untargeted”, but persistent methods was demonstrated by CloudFlare’s Heartbleed Challenge. Heartbleed was a vulnerability in the OpenSSL library that resulted in memory disclosure via an out-of-bounds read operation. The aim of the challenge was to recover the RSA private keys used by an Nginx webserver by leaking memory using Heartbleed. The winners of the challenge sent requests in order of $10^5 - 10^6$ magnitude to recover remnants of the private key.

2.4. Impact

Vulnerability scoring systems such as the Common vulnerability Scoring System (MITRE, 2019) or the OWASP Risk Rating Methodology (OWASP, 2019) measure the technical impact based on the classic Confidentiality-Integrity-Availability triplet. From this aspect UMD vulnerabilities always result in loss of confidentiality, but in practice the consequence of a breach would depend on the nature of the accessed data too. As described in the Exploitability section, the virtual address space of the web server process(es) is the outermost boundary of the data that can be accessed. In worst case, security critical application data, such as TLS private keys or database credentials can be accessed. Accessing such data would jeopardize the security of the entire target system and its users. Non-critical application data, such as file paths or module base addresses can also be accessed. While the disclosure of such data is undesirable, an attacker would require at least another vulnerability (such as a file include or memory corruption issue) to use them. Third, data of other users (cross-session data), including (partial) HTTP bodies and headers, from requests and responses can contain a wide range of sensitive information. Because of the high level of variance, the impact of UMD issues must be

Bálint Varga-Perke, vpbalint@silentsignal.hu
measured on a case-by-case basis, factoring in non-technical factors, such as the Business Impact Factors described in OWASP RRM.

2.5. Vulnerability Case Study

To better understand UMD type vulnerabilities and to develop practical detection methods, the vulnerability CVE-2019-6976 will be used as a case study.

CVE-2019-6976 was an uninitialized memory leak in the libvips (Cupitt, libvips, n.d.) imaging library discovered on 18th January 2019, and publicly documented on 18th April the same year (Varga-Perke, Drop by Drop - Bleeding Through Libvips, 2019). Libvips relies on other popular image parser libraries (such as ImageMagick or libjpeg) to provide a versatile, high-performance implementation with a uniform interface for image transformation.

The underlying libraries (or libvips itself) can’t always handle input data, but libvips tries to handle these errors gracefully, producing valid output images even in case of invalid input. The vulnerability occurred, because libvips failed to initialize its buffers holding pixel data. If the input image processor failed to fill the pixel data, libvips proceeded with image generation, using the uninitialized buffer as input pixel map. This resulted in raw memory contents visibly appearing on the output images.

2.5.1. Discovery

CVE-2019-6976 was discovered during a black-box web application penetration test. The vulnerability could be triggered by uploading a corrupt profile picture to the application. The Upload Scanner Burp Suite extension was used to automatically upload special images and detect potential vulnerabilities based on application behavior. Some of the uploaded images were from the fingerping (Bongard, n.d.) library, that is a collection of corrupt images that allows identifying image parser libraries based on parsing errors.

The affected interface of the target web application was the one responsible for profile image uploads. Since the application used libvips to resize the uploaded images, some of the corrupt images showed visual patterns similar to the ones described by Chris Evans. The problem could be recognized because some corrupt images were uploaded last by the Upload Scanner. This way the resulting thumbnails (containing uninitialized

Bálint Varga-Perke, vpbalint@silentsignal.hu
memory contents) remained on the application interface until a new profile picture was uploaded. Another factor of discovery was that the human tester had sufficient background knowledge to recognize the problem. This means that the original discovery was a matter of luck rather than the result of some methodological approach. Section 2.6 will discuss ways to reliably detect similar issues.

2.5.2. Modeling Vulnerable Applications

To help understanding UMD vulnerabilities in context of web applications and to assist tool development, a Docker image was developed (Silent Signal, 2019). Docker allows running consistent user-spaces on different machines, using a lightweight configuration file. The original Docker image was developed by Imre Taba with the guidance of the author. The original image included a Python web application that resized uploaded images with a vulnerable version of libvips. For the purposes of this paper, further developments to the image was made by the author.

The Docker container can be started with the following command:

```
docker run --privileged \
-p 8888:80  -p 8085:8085 -p 8099:8099 \
-t leakbuild
```

The `-p` flags forward the ports of the host system to the services running in the container. The `--privileged` flag allows accessing the `/proc` filesystem from inside the container, so memory mappings can be observed. The following command can be used to get interactive shell access to the container:

```
docker run --privileged \
-p 8888:80  -p 8085:8085 -p 8099:8099 \
-it leakbuild bash
```

When running in interactive mode, the services must be manually started by executing the following command in the `/app` directory of the container:

```
./restart.sh &
```

The image contains three web applications: one written in PHP, running on Apache HTTPd, one written using the Python Flask microframework, and one implemented with Node.js (TCP/8888 – Apache/PHP; TCP/8099 – Node.js; TCP/8085 –

Bálint Varga-Perke, vpbalint@silentsignal.hu
Python/Flask). The latter two web applications use the development servers of their frameworks. The docker image and the application dependencies are set up with a vulnerable version of the libvips library.

The Apache HTTP server is configured with the default Prefork Multi-Processing Module (MPM) (Apache Software Foundation, n.d.). According to the documentation (Apache Software Foundation, n.d.) the Prefork MPM module “implements a non-threaded, pre-forking web server” and “each server process may answer incoming requests, and a parent process manages the size of the server pool”. In contrast to the Worker and Event MPM’s, the Prefork module uses one thread per child process. Intuitively this should reduce the likelihood of cross-session data disclosure from a single process. Single-threaded execution also makes loading of not thread-safe modules (such as the one provided by the libapache2-mod-php Debian package) possible.

2.6. Developing Detections

In the last section we concluded, that the discovery of CVE-2019-6976 was a question of luck: a capable human observer was needed to observe the target system at the right time, when unusual application output could be recognized.

After discovering CVE-2019-6976 a new detection was developed for the Upload Scanner Burp Suite extension (modzero AG, 2018). The additions were developed by Imre Taba with the guidance of the author (Silent Signal, 2019). This addition compresses the output images and calculates the compression ratios of the inputs and outputs. Requests resulting below median ratios are finally reported as potential vulnerabilities, effectively marking half of the output for review. This section refines the implemented technique and provides measures of its effectiveness.

2.6.1. Recognizing Randomness

A reliable (black-box) detection method would automatically a) inspect all input-output pairs for a potentially large number of test cases and b) recognize “unusual” patterns in the output. Inspecting application requests and responses is a basic functionality of virtually all web application test or development tool. On the other hand, pattern recognition is generally a task more suitable for the human brain than for a

Bálint Varga-Perke, vpbalint@silentsignal.hu
program. Thus, for black-box detection the main challenge is to create an algorithm that can recognize if a block of data contains uninitialized memory contents.

After experimenting with the test target, corrupt output images could be manually separated into two groups:

i. **Noisy images:** Objects in images can be mathematically described as areas of continuous brightness (Lindeberg). Frequent, sharp changes in brightness (or color) appear as noise to the human eye. Although non-image data in memory can follow structures, smooth (continuous) transitions between neighboring byte values are rare. A majority of observed corrupt images thus appear noisy, as if they were in part generated randomly.

ii. **Dotty images:** Some observed images appear almost entirely black, containing only a few lighter dots. Such output can be the result of disclosing rarely used memory, most of which remained zero initialized between allocation and disclosure. Although these images don’t usually contain useful information, their presence is a good indicator of application malfunction.

The following images show examples of the two categories:

![Figure 2 - Noisy output image](image-url)

Bálint Varga-Perke, vpbalint@silentsignal.hu
Based on this observation, two approaches were taken simultaneously to improve the existing detection algorithm. First, the “dottyness” of images were measured by calculating the ratio of black to non-black pixels. Second, other statistical approaches were tried to detect outliers of the observed compression ratios.

During this process, a new test set (Varga-Perke, Image Memory Leak Test Suite, 2019) was generated from the Fingerping test inputs, containing only blank images. The original motivation for this was to increase the difference of compression ratios between uncorrupted and noisy outputs – see Appendix 4.2. The new test suite consists of 56 PNG images at the time of writing. The PNG format is simple, lossless, and widely accepted, making it an ideal test input format. The new test set was generated by parsing the IDAT chunks of the original PNG images and replacing their (uncompressed) contents with zeroes (Varga-Perke, PNG IDAT Filler Script, 2019). In PNG images IDAT chunks contain the effective color values to be displayed. To improve compression, IDAT data can be filtered, meaning that individual values can not only represent individual channel brightness, but relative brightness to neighboring pixel channels. By providing zero values, the filter method is always set to None (no optimization), and the pixel values will have the same color (usually black, but in case of palette-based encoding, palette index zero can correspond to arbitrary color. This doesn’t affect the maximum compression ratio of the generated data). Test images with filter values other than zero can provide

Bálint Varga-Perke, vpbalint@silentsignal.hu
interesting results too. Such test images can be manually created to improve test coverage. The transformer script performs minimal parsing on the input file (detects IDAT markers at arbitrary positions), so even invalid PNG files can be transformed.

Even with this liberal transformer, some files couldn’t be processed. Additional manual work was needed in cases when the input files didn’t contain IDAT chunks or when improperly formatted zlib data was present inside the IDAT chunks. Single color JNG and MNG files were generated from scratch, using ImageMagick. Files with invalid zlib headers (bad zlib method, checkbits and window values) were created from the blank control image by editing the appropriate values in a hexeditor. The image with nonconsecutive IDAT chunks was generated by splitting the IDAT chunk of the control image using a hexeditor, then rearranging the chunks using Python. No modification was required in case of the test file with an empty zlib object, as this file didn’t contain any image data. The information available regarding CVE-2014-0333 was inconsistent with the corresponding test file, so it was excluded from further tests.

While the new, blank test suite provided higher contrast compression ratios between noisy and blank images, it also provided a trivial way to detect memory disclosures: if an output image contained more than a single color, it likely was the result of corrupt processing.

### 2.6.2. Measuring Detection Quality

In order to measure the effectiveness of this technique a high number of generated images had to be collected. A simple, yet highly configurable tool (Varga-Perke, Image Collector Script, 2019) was developed to collect transformed images in bulk. Although this tool is primarily aimed to support the test applications available in the test container, it’s simple enough to be easily extended to support more complex targets (such as ones requiring authentication).

Another tool (Varga-Perke, Image Memory Leak Detector Script, 2019) was implemented different measurements on the collected images. The tool relies on the Pillow (PIL) library (Alex Clark and contributors, n.d.) to parse images, and extract each channel of each raw pixel into a byte array. Note that at this stage flaws of the parser should not affect the results as output images are expected to be well-formatted.

Bálint Varga-Perke, vpbalint@silentsignal.hu
To assess the effectiveness of a detection method an oracle could be created based on the observations of errors produced by libvips. The vulnerable version of libvips reliably creates corrupt images from a specific subset of the test suit. A result can be judged based on the output filename, because `collect.py` includes the name of the input file in the name of collected output. The oracle was implemented as a Python script that consumes the CSV output produced by `detect.py`. The oracle compares each detection result with the predicted value, and reports any false positive or false negative detections.

When running the described trivial detection on 10337 collected images, the oracle reported 81 false negative results and no false positives. All false negatives were outputs generated from a PNG with invalid zlib checksum. This particular test case always results in almost entirely black output images, and was only recognized as a trigger after early dotyness measurements reported some extreme values – visual observation didn’t recognize these anomalies. Other results were in fact all black, so there was no way to distinguish them from valid outputs.

The tests were performed in a setup where the target service created JPEG output with lossy compression. The output data shows that in case of blank input compression doesn’t introduce artifacts, which could result in false positive results.

These results show that the new test suite allows precise, automatic detection of corrupt output images.

2.7. Information Extraction

After suspicious output was detected using the appropriate test suite it’s time to extract any meaningful information from the generated images. For this it is important to identify the flaw of the parser and the transformations applied by the target application. With this information a target specific trigger image can be created, that will always result in corrupt output, with minimal information loss.

One of the flaws of the target libvips version is the incorrect handling of invalid zlib window size. The Node.js model application listening at port TCP/8099 of the test container crops images to 200x200 pixels. A trigger input can be generated from a 200x200 blank PNG file, setting the first byte after its “IDAT” marker to 0x88 with a

Bálint Varga-Perke, vpbalint@silentsignal.hu
hexeditor. The PHP application converts its inputs to a given format without resizing. The Python application resizes images to 400 pixels width. Trigger images to these later applications can be created in the same fashion.

Information extraction was assessed using the `collect.py` script that was developed so that it can perform repeated uploads with given delays. Also, the script is capable to extract raw pixels from the resulting images and detect patterns in the resulting byte array.

### 2.7.1. Cross-Session Data

Cross-session can be easily identified when using the target application interface to upload a recognizable file in one session while repeatedly uploading the trigger file from another. In case of cross-session data disclosure (different uploads handled by the same process) parts of the control image will become visually recognizable.

The following picture was used as a control picture in some tests:

![Control Image](image1.png)

**Figure 4 - Control image**

The following picture was generated from a trigger image. The text from the control image is clearly recognizable, some of it remained intact and readable:

![Trigger Image Transformation](image2.png)

**Figure 5 - Result of trigger image transformation**

Bálint Varga-Perke, vpbalint@silentsignal.hu
All three test applications showed similar results. This kind of information leak seems to be the easiest to observe.

2.7.2. Text-Based Data

HTTP/1.1 is a text-based protocol, and web applications are usually configured using text-based formats (such as XML, YAML, connection strings, etc.). Therefore, the most straightforward way to find useful information in random memory contents of the target is to look for ASCII character strings. HTTP/1.1 protocol is based on US-ASCII, and other popular encodings, like UTF-8 and Latin-1 also include ASCII as their subset.

When using the --decompress argument of the collect.py script raw color values are stored with .dat extension along with the downloaded images. The retrieved strings can be easily inspected using the strings command line utility. The following command output shows, that non-critical application data (absolute application paths) were retrieved in high numbers during 1000 trigger file uploads:

```
$ strings -nl2 paper_measurements/8099_trigger_1000_2/*.dat | fgrep app | sort | uniq -c
 2 /app/leaks/node
 8 /app/leaks/node_modules/sharp/l
203 /app/leaks/node_modules/sharp/lib/icc/cmyk.icm
239 /app/leaks/node_modules/sharp/lib/icc/sRGB.icc
```

Inspecting string data also proved useful during tool development, as the presence of comprehensible strings is an indicator of precise data extraction.

In order to demonstrate the cross-session extraction of string data collect.py was executed in a control process, uploading a 1x1 pixel image along with a “password” passed in a dummy command line parameter. In another process, collect.py was used to trigger UMD and detect the presence of the password in the resulting images.

The following control command was used:

```
python collect.py --url http://localhost:8085/ \
   --extract 'href="(\"\")+"' --prefix "http://localhost:8085" \
   --params 'ext=png&password=Password1' --iparam file --sleep 0.5 \
   --indir /path/to/1x1/ \
   --nokeep --repeat 10000
```

The following trigger command was used:

```
Bálint Varga-Perke, vpbalint@silentsignal.hu
```
python collect.py --url http://localhost:8085/ \
--extract 'href="([^\"]+)"' --prefix "http://localhost:8085" \
--params 'ext=png' --iparam file --sleep 1.0 \
--indir /path/to/trigger/ --outdir /tmp/collect/ --repeat 1000 \
--decompress --pattern '50617373776f726431' | fgrep FOUND

The experiment yielded multiple results. An example of resulting image data is shown below in a hexeditor:

![Hexedit](https://example.com/hexedit.png)

This is despite the fact that the application doesn’t use the dummy parameter. In real applications, many copies of HTTP parameters and headers may be created during parsing, esp. if the web application relies on immutable string objects.

This result could not be reproduced on the Python or PHP applications, even after reducing the MaxRequestWorkers parameter of the Apache Prefork module to 1. This result doesn’t rule out the possibility of cross-session disclosures though, since several factors affect the range of accessible data.

### 2.7.3. Pointer Disclosure

Uninitialized memory buffers can contain pointers to previously allocated addresses in the virtual address space of the process. Identifying valid pointers can help defeat ASLR and enable code-reuse, such as ROP (Erik Buchanan, 2008) or return-to-libc (Designer, 1997). Code reuse techniques are essential techniques for memory corruption exploitation when the W^X policy (Wikipedia, n.d.) is in effect. This way

Bálint Varga-Perke, vpbalint@silentsignal.hu
memory disclosure can help the execution of more impactful attacks, such as ones leading to remote code execution.

Knowing the specifics of the target architecture and operating system can help with the identification of pointers. User-space memory occupies the lower part of the address space, so the most significant bits of any user pointer must be zeroes. In practice, on Linux, the highest 3 bytes of shared library base addresses are well-known (0x00007F or 0x00007E) (Hector Marco-Gisbert, 2014). Since program sections remain continuous in memory, pointers to the same section start with the same nibbles. Series of pointers are typically aligned, and show the above (or similar) patterns in repetition. These patterns can even be visually recognizable (Evans, What does a pointer look like, anyway?, 2014).

A script (Varga-Perke, Pointer Identification Script, 2019) was developed to identify possible pointers in image data based on the frequency of their occurrence. The script uses a sliding window to create statistics of 64-bit values present in byte streams. After decoding values as 64-bit big-endian integers the script masks some of the least significant bits. Next, values between 0xFFFFFFFF and 0x0100000000000000 are counted in one map (“high”), while values between 0xFFFFFFFF and 0xBFFFFFFF are counted in another (“low”). To help assessing the results, the script is also capable of parsing the /proc/PID/maps file of Linux processes, and assign observed, common values to virtual memory mappings.

Experiments were made against the Node.js and PHP test application to assess the effectiveness of this approach. The following images show the results of counting the potential pointers:
Figure 6 - Most common pointers collected from 1000 runs against two instances of the Apache/PHP test application

Figure 7 - Most common pointers collected from 1000 runs against two instances of the Node.js test application

The first columns show the most common (masked) address ranges. The second columns show the number of their occurrences. If a map file was provided and a match was found, the third columns (in brackets) show the corresponding mapping.

These results show that high frequency values that “look like” pointers likely correspond to mapped memory ranges. Likely heap locations can also suggest the state of ASLR on the target (Evans, Proving missing ASLR on dropbox.com and box.com over the web for a $343 bounty, 2017). In this case, the node binary is not compiled as

Bálint Varga-Perke, vpbalint@silentsignal.hu
position independent, while Apache is. If the main executable is statically positioned, code-reuse becomes trivial. It should be noted though, that this information may be obtained through simple OS and/or HTTP server fingerprinting (Open Web Application Security Project, n.d.), assuming that the target uses default binary packages.

In case of position independent executables, the value of the pointer leaks is not straightforward because of multiple reasons. First, common values don’t always correspond to the same kind of mapping. Even assuming that heap addresses are always of the highest frequency, the offsets between the heap and the base images (or libraries, from where executable code could be borrowed) are not static. Second, leaks from known libraries is relatively rare. In the previous examples of Apache, among libraries, addresses from libvips were the most common, with 22nd and 38th highest frequencies overall.

Identifying application-specific structures with code pointers (such as virtual function tables) may be a viable way to assist code-reuse. While knowing the randomization level and partial addresses of the target may help, this task is expected to require targeted expertise and tooling.

3. Conclusion

This paper demonstrated that image parsing errors resulting in uninitialized memory disclosures can be reliably and automatically detected. This allows more thorough testing of web applications, where unsafe image parsing is a relevant, but often overlooked problem. Test results showed that instead of complex pattern recognition or statistical analysis, a well-chosen test suite can provide a simple, yet precise detection method. The exploitability of memory disclosures was demonstrated in different environments.

During this research several tools were developed which are to be open-sourced to encourage more research in the area. Developing new test cases will require long-term effort to cover the widest set of formats and their edge-cases. In case of lossy image formats, rate-distortion theory may provide improved methods for information recovery.

Bálint Varga-Perke, vpbalint@silentsignal.hu
As ever increasing parts of our analogue lifes are being digitally processed, similar problems are expected to show at new and exciting places. Maybe at the same time – similarly to the discussed, visible leaks – we will soon learn to recognize memory safety issues with our other senses too?

Bálint Varga-Perke, vpbalint@silentsignal.hu
References

Alex Clark and Contributors. (n.d.). Pillow. Retrieved from PyPi: https://pypi.org/project/Pillow/


Bálint Varga-Perke, vpbalint@silentsignal.hu


Bálint Varga-Perke, vpbalint@silentsignal.hu


Bálint Varga-Perke, vpbalint@silentsignal.hu
https://github.com/v-p-b/image-memleak/blob/master/collect.py

https://github.com/v-p-b/image-memleak/blob/master/detect.py

https://github.com/v-p-b/image-memleak-testsuite/

https://github.com/v-p-b/image-memleak/blob/master/fill_idat.py

https://github.com/v-p-b/image-memleak/blob/master/pointers2.py

https://en.wikipedia.org/wiki/W%5EX
4. Appendix

4.1. Cases of Uninitialized Memory Disclosure on HackerOne

Unupdated ImageMagic leads to uninitialized server memory disclosure - https://hackerone.com/reports/274594
ImageMagick GIF coder vulnerability leading to memory disclosure - https://hackerone.com/reports/302885
Uninitialized server memory disclosure via ImageMagick in my.mail.ru and cloud.mail.ru - https://hackerone.com/reports/251732
ImageMagick GIF coder vulnerability leading to memory disclosure - https://hackerone.com/reports/315256
Uninitialized server memory disclosure via ImageMagick - https://hackerone.com/reports/294548

4.2. Initial experimental results with other detection methods

Due to the nature of the vulnerabilities to detect, we can expect the contents of noisy images to be based on multiple (independent) pseudo-random sources: pointer values randomized by ASLR, data sent by other user of the target, heap metadata shaped through the lifetime of the process, etc. We can therefore expect that some parts of the output data will be incompressable (Peter Sunehag, 2015), and consequentially, that noisy outputs will be less compressible than others.

To detect dotty images, we can simply calculate the ratio of zero to non-zero values in the output bitmap (for each pixel component). This ratio is expected to be significantly higher in case of dotty images than for correctly generated ones.

The previous techniques assume that the source images are neither noisy nor dotty, and that the transformation applied by the target application don’t introduce such features. Choosing good test inputs can also have a high impact on the quality of statistical detections. For the compression based detection to work best, inputs containing minimal information should be provided. Let us treat pixels of the input image as output sequence of an entropy source. If all pixels are of a constant value, the entropy associated with this source is 0. According to Shannon’s Source Coding Theorem¹, minimal theoretical compression rate is achieved, when entropy is minimal. Assuming that

¹ https://en.wikipedia.org/wiki/Shannon%27s_source_coding_theorem

Bálint Varga-Perke, vpbalint@silentsignal.hu
practical compression algorithms approach the theoretical limit, we can expect that compressed image sizes will be minimal in case of blank images.

This way properly processed images will show extremely high compression ratios which should be easily distinguishable from noisy images. In case of image inputs, completely black pictures serve best, and also match the input criteria for dotty detection (no dots).

A stand-alone detection tool was developed to measure the effectiveness of different detection methods. The tool was configured with two “detector” classes:

- CompressorDetector compresses the bitmap data of images using the DEFLATE algorithm as implemented in the zlib library. It returns the ratio of original bitmap size to the compressed size.
- RareDotDetector counts the non-zero bytes in the bitmap of the image. It returns the ratio of zero to non-zero bytes, or -1 if there were no non-zero bytes.

For improved detection accuracy a new test set was generated from the Fingerping test inputs – see 2.6.

To automatically identify noisy or dotty images, outliers in the data must be detected. The implementation in UploadScanner marks half of the output set as suspicious, regardless of the actual distribution of compression ratios (dotty images are currently not detected).

With the improved test suite we can expect that properly processed images would produce about the same compression ratios and dottyness. Blank images would show extreme compression ratios, and dottyness of 3 at most (single non-black component in every 4 channels). Corrupt images would show modest compression ratios – the compression ration of the standard Lena test image results in 10.39x ratio with JPEG and 1.65 with PNG\(^2\). In case of dotty images our initial assumption is that ratios above 3 should be treated as abnormal.

\(^2\) http://links.uwaterloo.ca/Repository/TIF/lena3.tif

Bálint Varga-Perke, vpbalint@silentsignal.hu
The following charts show the compression and dottyness ratios (Y axis) of 400-400 output files (X axis) generated with vulnerable libvips, using the fingerping and new test suites:

Figure 8 - Compression ratios of output images generated from the fingerping test suite

SHA-256: ecabbfc5c57cac78f90cf150b1491ae1a8b96a9dec5083a3a5cd3ba500edd605
Using convert utility from ImageMagick 6.8.9-9 Q16 x86_64 2019-06-15 without extra parameters.

Bálint Varga-Perke, vpbalint@silentsignal.hu
Figure 9 - Compression ratios of output images generated from the new test suite

Figure 10 - Dottyness ratios of output images generated from the fingerping test suite, invalid ratios (only zeroes) excluded (logarithmic scale)

Bálint Varga-Perke, vpbalint@silentsignal.hu
Figure 11 - Dottyness ratios of output images generated from the new test suite, invalid ratios (only zeroes) excluded (logarithmic scale)

The above charts show that using the purpose-built test suite, the outliers in compression rate data became more easily distinguishable. Although such differences are harder to visually spot in case of RareDotDetector results, the improved test set provided more extreme outliers, which could be trivially detected.

For automatic detection of extremes in series of data of varying range standard deviation from the mean can be used. Since we expect the majority of the test cases to produce similar (blank) images, we can expect the mean of the measurements to be close these “normal” values, and the standard deviation to be small. Meanwhile, corrupt images would result measurements far from the mean, allowing a clean-cut separation of these results.

To assess the effectiveness of this approach an oracle can be created based on the observations of errors produced by libvips – see 2.6.2.

The oracle was first configured to report any samples with dottyness or compression rates outside a single standard deviation from the mean of the corresponding values. During this time the oracle wasn’t set up to recognize images generated from ones with invalid zlib checksums. This was because human observation never identified such images as corrupt – they were “all black”. However, measurements shows extreme dottyness rates (several thousand) in case of these images. After careful reevaluation dots
were identified on these images too, so their identifier ("bad zlib checksum") was added to the oracle.

![Uninitialized Memory](image)

Figure 12 - Bottom right corner of an image with dottyness rate of approx. 1200, with 550% zoom

After the oracle was corrected, the initial outlier detection provided results similar to the following based on results of 100 runs with the improved test suite (5600 images) against the implemented test applications:

```
> python oracle.py paper_measurements/8888_testsuite_100x/detect_noCVE.csv
compressorDetector - STDevDetector with 2.000000 multiplier
RareDotDetector - STDevDetector with 1.000000 multiplier

STATISTICS:
compressorDetector: Mean: 804.022600 Stdev: 329.818605
RareDotDetector: Mean: 326.003320 Stdev: 458.913320
```

The picture shows that images with bad zlib checksums still produce some false negatives, but the majority of these are in fact blank images (-1.0 dottyness, high compression rate), that can’t be detected anyway. To improve these results the allowed difference from mean was reduced to half standard deviation in case of RareDotDetector. The new results are shown on the following screenshot:

The new results are shown on the following screenshot:

```
> python oracle.py paper_measurements/8888_testsuite_100x/detect_noCVE.csv
compressorDetector - STDevDetector with 2.000000 multiplier
RareDotDetector - STDevDetector with 1.000000 multiplier

STATISTICS:
compressorDetector: Mean: 804.022600 Stdev: 329.818605
RareDotDetector: Mean: 326.003320 Stdev: 458.913320
```

Bálint Varga-Perke, vpbalint@silentsignal.hu
By adjusting the threshold, the detectable false negatives disappeared. It should be noted, that due to the extreme values produced by RareDotDetector, high number of false negative results occur when using this method alone. By using both methods, only undetectable false negatives occur, improving the results of a single, compression based detector too.
<table>
<thead>
<tr>
<th>SANS OnDemand</th>
<th>OnlineUS</th>
<th>Anytime</th>
<th>Self Paced</th>
</tr>
</thead>
<tbody>
<tr>
<td>SANS SelfStudy</td>
<td>Books &amp; MP3s OnlyUS</td>
<td>Anytime</td>
<td>Self Paced</td>
</tr>
</tbody>
</table>