Testing libpng with Fuzzing and Concolic Testing

Roeland Krak
University of Twente
r.s.krak@student.utwente.nl

Joris Diesvelt
University of Twente
j.j.diesvelt@student.utwente.nl

***Abstract*—We constructed a practical approach to analyze software libraries with automated software testing techniques using simple wrapper programs. We have used this approach to test libpng with afl-fuzz, in doing so we were able to confirm a known vulnerability. We discuss how this approach is unfit for analysis with the current version of the angr framework. However, we elaborate on analysis techniques that could have been applied with angr had library analysis already been possible. We illustrate a pitfall in writing wrapper code, but highlight how semantical bugs may be discovered in the process of recovering from it. Furthermore, we demonstrate that fuzzing wasn't able to find other bugs in the tested library functions.**

Keywords
libpng, Fuzzing, afl-fuzz, Concolic Testing, angr

1. INTRODUCTION

Portable Network Graphics (PNG) are a widely used standard for image encoding. The standard is implemented in the official reference library *libpng*. This library has a rich history of well-documented vulnerabilities which means it is a good target to validate testing tools on that are intended to discover vulnerabilities.

Because of the widespread use of PNG as an image standard, the security of its implementations is important. As libpng is used in many projects, security issues can have a widespread effect on many different applications. Due to the proper documentation of vulnerabilities we consider the library a proper target to test using automated testing software. We tested the latest version of libpng, version 1.6.21, and version 1.6.0.

The techniques we use for automated testing are fuzzing and concolic testing, as described in [12]. We test these techniques implemented in the tools american fuzzy lop (afl-fuzz) and angr. These tools test software in two very distinct ways. We describe these techniques and tools in sections 2 and 3.

Many tools for automated software testing focus on testing executable software. Software libraries are a different form of software, which is not by itself executable, but instead offers functionality that can be called by other software. This poses a problem in applying automated testing techniques to test software libraries. We suggest wrapper software can be constructed that calls library functions, so that the wrapper programs can be tested automatically.

In this work we do not attempt to perform a broad security test of libpng, testing all possible features of the library. Instead, we only test certain libpng functions that we included in wrapper software. The contributions of this work are as follows:

- We construct an approach to analyze software libraries using automated testing techniques intended to test executable software.
- We test the security of libpng versions 1.6.0 and 1.6.21.
- We demonstrate the ease of automatically detecting vulnerabilities using these techniques, which showcases vulnerabilities can be avoided by performing automated tests during development.

The rest of this report is ordered as follows. In section 2 we describe the automated testing techniques that we use, i.e. fuzzing and concolic testing. In section 3 we describe the tools afl-fuzz and angr, which implement these techniques. In section 4 we describe the PNG format. In section 5 we describe the libpng library. In section 6 we describe our approach to testing libpng. In section 7 we expand on the importance of correct wrapper code. In section 8 we describe our testing results. In section 9 we evaluate our results. In section 10 we discuss our findings. Lastly, in section 11 we conclude our work.

2. AUTOMATED TESTING TECHNIQUES

Software testing is an important part of software development which helps discover complex bugs and vulnerabilities. There are two broad approaches to software testing. Either the code of the tested software is available and is used while testing, or it is not. These approaches are known as white-box and black-box testing respectively.

We now shortly describe fuzzing and concolic testing in the following subsections. For a more elaborate description and illustration of these techniques we refer to [12].

2.1 Fuzzing

The standard technique of automated black-box testing is known as *fuzzing*, or fuzz testing, which means many concrete inputs are provided to the software. The testing program then monitors the tested program for exceptions, such as whether the tested software crashes or assertions fail [10]. The tested inputs can be randomly constructed, or generated using specific heuristics which should be more likely to find bugs. However, fuzzing generally does not use knowledge of the inner workings of the tested software which means it can be hard to find inputs which cover rarely executed program paths.

2.2 Concolic Testing

To understand concolic testing we must first describe *symbolic* execution, which requires access to the code of the tested software and is thus a white-box testing technique. Symbolic execution interprets the code of the tested software and reasons about symbolic values instead of concrete values. Because of this, it can generate constraints on inputs which determine which execution paths are taken by the tested software [8, 15]. This allows symbolic execution to reach high program coverage as it can direct testing towards untested execution paths.

Concolic execution is the combination of concrete and symbolic execution[14, 11, 8]. This combination allows testing of software – or parts thereof – using concrete values, while at the same time reasoning about execution paths. This allows the testing program to achieve high program coverage while executing the program with concrete values.

3. TOOLS

In this section we describe the tools we used in this work. For angr we give an elaborate description of techniques that can be applied to discover different bugs.

3.1 afl-fuzz

afl-fuzz, also known as american fuzzy lop, is a tool used to fuzz test binaries. It generates inputs using genetic algorithms to automatically discover interesting test cases that are likely to trigger new program behavior [1]. The tool achieves high fuzzing speeds due to optimizations in implementation and optimizations achieved by instrumented compilation of tested programs.

Although instrumented compilation speeds up afl-fuzz significantly, the downside is that it misses many of these optimizations when executed on closed-source binaries, slowing down testing significantly.

It should be noted that absence of results from fuzzing doesn't mean no vulnerabilities or other bugs are actually present in the tested code. Because fuzzing is based on random generation of test cases, bugs will not always be found. Furthermore, our experience is that the broadness of covered paths may sometimes increase when restarting the afl-fuzz. We assume this is because afl-fuzz favors certain mutations on each execution, and the favored paths may differ between runs. This assumption is motivated by the fact that sometimes 500.000 test inputs report no crashes or hangs, whereas other times multiple crashes and hangs are reported within just thousands of tests.

3.2 angr

angr is an open-source framework that can be used for symbolic testing, concolic testing, and fuzzing. It was originally developed for Firmalice [15], a framework to detect authentication bypass vulnerabilities.

angr's main goal is to provide developers with a complete set of tools for testing and reverse engineering binaries [2]. For this reason, it combines several modules which can be used independently as well as within the angr front-end. Some notable modules are: *SimuVex*, which handles constraint generation and can simulate program states such as file inclusion and user input on graphical interfaces; *Capstone*, an open-source disassembly framework; *Claripy*, a constraint solver, similar to the commonly used Z3.

A graphical user interface (GUI) is also made available by the same developers: angr-management. With this GUI, a binary can be loaded and most of the features of the framework can be applied to this executable. Unfortunately, the current version of the GUI does not add much to the framework and suffers from frequent crashes. It is, however, able to give a decent representation of the control flow graph of the program and the branches of executed path groups, which visually aids analysis of the binary.

3.2.1 Limitations

Although the combination of these modules result in a powerful framework, it has several drawbacks. First of all, installation can be painful because it has multiple dependencies that change over time and can have compatibility issues with different versions of angr. For example, the latest version of Capstone introduced a bug in its installer and required manual interaction from the user to fix [6]. Furthermore, the framework is in an incomplete state, which means many things change each version and a large portion of the documentation is missing [3]. The biggest disadvantage in this case is the seeming impossibility to test software libraries.

Software that calls functionality from a software library is said to be *linked* to that library. Linking can be done either *dynamically* or *statically*. A dynamic link means the library needs to be installed on the operating system on which linked software is executed. The operating system then takes care of calling the library functions of the installed library at run-time. The advantage of this approach is that the library can be updated independently of the linked software.

Static linking on the other hand means the library is included in the linked software during compile time. This increases the size of the compiled software but has the advantage that a single piece of software can be executed without dependencies on libraries that need to be installed on the operating system as well.

angr does not normally evaluate function calls to dynamically linked libraries. Wrapper programs therefore need to be compiled with static links to the libraries, as this means the library code is included in the binary and angr can traverse the execution paths of the library. For our wrapper programs, as later described in section 6, we used the command `gcc [file_name.c] -static -lpng16 -lm -lz -std=c99` to compile the programs with statically linked libraries.

Unfortunately, as angr is incomplete, it does not support all possible software functionality. Static compilation of our wrapper software generates a program that triggers an error in angr while performing symbolic execution. The returned error is "`no syscall 63 for arch AMD64`", which means a system call occurs in the program which is not implemented in angr. We assume this is a system call performed by libpng, as without inclusion of this library we can run angr on our software without triggering errors. In the future, it might be possible to force angr to load static libraries alongside the specified binary, but this currently remains an open issue [5].

This means that due to the incompleteness of angr, it cannot be applied to analyze all possible software, including some software libraries. Unfortunately, this prevents us from applying concolic testing on our wrapper software. We will therefore elaborate on techniques that can be applied with angr to analyze software and detect certain bugs in software.

3.2.2 Software Analysis Techniques

Binary analysis is very complex and the angr framework attempts to make this easier by giving the developer a set of tools that are easy to use but still cover a broad range of intensive analysis techniques. We demonstrate several features of the framework by applying certain analysis techniques to sample binaries.

In the area of static analysis, angr is able to create complete control flow graphs, value flow graphs and data dependency graphs of a binary. These give insight into the flow of a program and can be used to find interesting parts of the code for further analysis. Certain locations of secrets or triggers can be found this way.

To perform dynamic binary analysis, the most important tools are the symbolic state objects. From any entry or execution point in the code, a `State` can be retrieved, which represents the current state of execution including memory mappings and register values. These values can be changed on the fly, which gives the developer the possibility to interact with the executable in any way possible.

A state comes with a set of constraints, which can be interacted with using angr's constraint solving module. Solving these constraints allows it to create a concrete path towards the state and thereby enables concolic execution. A symbolic state can also be represented as a `Path` object which allows the developer to track an execution path. Most notably, when taking a step on a `Path` it returns its successors, which means it will return all possible paths at the moment the program encounters a branch condition. To keep track of multiple paths, the more abstract object `Pathgroup` is used to bulk execute different branches.

All previously mentioned techniques are not performed automatically, but rather have to be implemented by a software tester. To make initial testing easier, the `Surveyor` engine is built to automatize some aspects. The surveyors are basically symbolic execution engines that will attempt to perform symbolic execution on a binary using the mentioned techniques while simultaneously using some tricks to increase performance or counter common problems such as path explosion.

The `Explorer` class is one such surveyor that can be told what interesting addresses should be found or which paths should be avoided. Furthermore, it attempts to detect loops so that it won't get stuck. Unfortunately, this class has some issues, as will be shown in section 6.2.2. In appendix D, a sample script is shown that uses an `Explorer` to report the input and output of all possible paths within the `normalread.c` program.

angr's internal simulation engine, *SimuVex* allows the simulation of several program related events, such as system calls, input/output, and file system access. The simulation of system calls could help with testing the libpng library. Unfortunately there is no documentation available on this topic. However, a python script was created to test the simulation of the file system with `normalread.c`. This program was modified so that it would always open the test file `test.png`. From the script, also included in appendix D, it can be seen how a simulated file system environment is set up with this test file in it. When this file is used by the executable, angr can use it as a symbolic variable. This means that it can be solved by the constraint solver by any given path, and thus files can be dynamically generated for a given program state. Of course, the content of a file can take more forms than a simple single type variable. For this reason, the `SimFile` object can be given predefined content, which the symbolic executor will respect and thereby decrease the number of possible paths.

4. PORTABLE NETWORK GRAPHICS

PNG is a raster graphics encoding standard. It supports lossless compression of image data. The standard is documented in RFC 2083 [7].

A PNG file starts with an eight byte signature file header, followed by "chunks" containing several types of data. Decoders are required to be able to parse the IHDR, PLTE, and IDAT chunks. These chunks respectively contain header data, palette data and pixel data. Furthermore decoders are required to recognize the IEND chunk, which marks the end of the image. Other chunks may be supported, but for brevity we will not discuss them.

Chunks consist of four fields in the following order: a four byte length field, a four byte chunk type field, a dynamic length data field, and a four byte cyclic redundancy check (CRC) field. The length field describes the length of the data field, in number of bytes. The chunk type contains a textual representation of the type of chunk – for the purpose of this paper these are IHDR, PLTE, IDAT, and IEND. The data field contains the data stored in this chunk. The CRC field contains an error-detecting code which can help in detecting errors in the chunk type and chunk data.

For all other chunk types it suffices to say that there are ten standard ancillary types. Each chunk type is identified with a unique four letter code. It is also allowed to define private chunk types, so many more types may be encountered. For a detailed description of ancillary chunks we refer to RFC 2083, section 4.2 [7].

5. LIBPNG

libpng is the official reference library for PNGs [4]. It's an open-source library that contains many C functions that support almost all features of PNG. The library has a rich history of bugs and vulnerabilities documented in many Common Vulnerabilities and Exposures (CVE) identifiers, a system providing identifiers for flaws in code that result in security issues. Where applicable, we list the CVE identifier with the vulnerability. The known vulnerabilities for libpng 1.6.0 are as follows:

- A potential pointer overflow/underflow in `png_handle_sPLT()/png_handle_pCAL()`

- CVE-2015-8126: An out-of-bounds write in `png_set_PLTE()` and `png_get_PLTE()`
- CVE-2015-7981: A potential out-of-bounds read in `png_set_tIME()/png_convert_to_rfc1123()`
- Out-of-bounds memory access in `png_user_version_check()`.
- CVE-2014-0333: Hang when reading images that have zero-length IDAT chunks with the progressive (streaming) reader

In this work we focus on the PNG reading functions. There are two approaches to reading PNGs with libpng. One method opens the file normally, and parses the individual aspects of the PNG using several functions. The other method uses a progressive reader, which can be used to parse a data *stream* containing a PNG. As CVE-2014-0333 describes a vulnerability in the progressive reader, we validate the existence of this vulnerability.

6. APPROACH

As libpng is a software *library*, its code cannot be executed directly. Instead, one can interface with the functions implemented in the library by invoking them from a program.

Because afl-fuzz and angr operate on binaries, and not on source code, we cannot test the libpng code directly. Instead, we created wrapper programs that invoke the libpng functions that are to be tested. Automated testing can then be performed on these wrapper programs.

6.1 Detecting Program Crashes

Since we use wrapper programs to analyze libpng, we create code that itself is prone to errors. Not only does this mean we need to carefully analyze the code of our wrapper programs, it also means we need a definition for program crashes caused by libpng. Luckily, many libpng functions return error codes and output errors to *stderr* when faults are introduced.

These error codes should be used by any program invoking libpng, as this is the method by which libpng communicates problems in execution. This also means that errors, e.g. caused by improperly formatted PNG files or system limits, are no bugs by themselves.

Following this argumentation, we define a program crash to constitute *any improper end of a program execution, where no error code was returned.* We validate each generated test case reported as causing a crash, to ensure it does not output an error message, as this would indicate an error code could have been caught by the wrapper program.

6.2 Detecting Program Hangs

We define two forms of program hangs: *strict* hangs and *practical* hangs. Strict hangs occur when a program reaches a system state which it cannot leave, thus halting forever. Practical hangs on the other hand are states from which it takes a significant time to recover, but that not necessarily halt forever.

Formally detecting both types of hangs requires determining a program's worst-case execution time. However, detecting this is generally a hard problem, as it is equal to the halting problem [13]. Because formally detecting hangs is impracticable, we need a practical approach to detect them.

6.2.1 *Hangs Reported by afl-fuzz*

afl-fuzz detects hangs by measuring program execution time and comparing it to a configurable threshold time. If the execution time reaches the threshold time, afl-fuzz stops program execution and registers a hang for the given input. In all our tests we use the standard hang threshold time of 20 milliseconds.

For this reason, hangs registered by afl-fuzz are not necessarily strict hangs. Determining whether they are strict hangs cannot be done without code inspection. However, practical hangs can have as severe an impact as strict hangs, and we therefore think discriminating between these hangs is not necessarily required.

To determine whether a hang reported by afl-fuzz is a real hang – that is, either a strict hang or practical hang taking significant time – we execute the tested program with the input generated by afl-fuzz. If execution does not end within reasonable time we assume a hang. We consider execution time of multiple seconds convincible suggestion for a hang, as our test programs are so small that normal execution never takes more than a second on our testing machines.

6.2.2 *Hangs Reported by angr*

angr uses a different approach to report hangs. Every executable path is represented by the framework as a `Path` object. This object has the method `detect_loops` which, according to the documentation, returns the current loop iteration the path is currently on. Diving deeper in the source code, it turns out that what is actually returned, is the occurrence of most common block on the call stack. This value can then be evaluated against a predetermined threshold to determine whether this is a hang or not. This method could work because in an infinite loop, the calls within this loop will be incrementally added to the call stack. The automatic path surveyor `Explorer` chooses a default threshold of ten million repeats.

Some test samples are built to check in which cases angr can detect a hang using this method. Appendix C shows some of the test programs written for this. The Python script `exhaust.py` takes the file name of a binary, then uses an `Explorer` to exhaust all possible paths and report the generated input and output each time a path reaches a dead end. The C programs `loop.c`, `recurfunc.c` and `explosion.c` are interesting because they show three different types of behavior. All three programs read an input and, depending on this value, they can create an infinite loop. The first one, `loop.c`, creates a never ending while-loop with a single statement in it. The looping detection method flawlessly detects these kind of statements, most likely because this statement is put incrementally on the call stack. Instead of using a while-loop structure, `recurfunc.c` uses recursive method calls to create the infinite call. Even though it was expected that angr would detect this similarly to the last program, this type of loop is not detected. `explosion.c` uses a while-loop again, however this time the expression which determines continuation of the loop is based on user input. Because each input creates a new symbolic constraint, every iteration of the loop creates a new execution path. This causes an immediate path explosion within the `Explorer`. Because the `detect_loops` function is only called on each path and because there is no similar detection on path groups, angr is not able to detect this kind of loop.

6.3 afl-fuzz test files

afl-fuzz uses smart techniques to generate input which it tests against the program under inspection. To do so, it requires at least one test file, on which it applies mutations to form new input to test. The afl-fuzz suite comes with several test files for many different file types, amongst which four PNG files.

Our first tests were performed using the supplied PNG files, as we assumed these included interesting test cases. However, binary inspection of these files proved us wrong, as the files were largely similar in occurrence of chunks and represented data.

We acquired PngSuite [16], a PNG file corpus containing many interesting edge cases of the PNG standard. Of this suite we selected the 137 files that were smaller than 1 kilobyte, as this is the limit on one of our wrapper programs. Fuzzing with these test files proved to generate crashes and hangs more quickly and consistently. Occurrence of different chunk types in this test set is displayed in figure 1. This data was gathered from the test files using the Linux shell command `grep -ro "[TYPE]" test_files/ | wc -l`, where `[TYPE]` is replaced with each of the chunk types, and all files were located in the folder `test_files`.

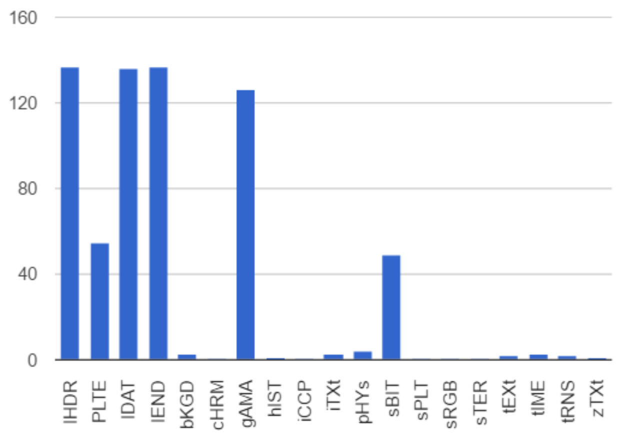

Figure 1. Occurrence of chunk types in our PNG corpus

6.4 Normal Reader Test

We created a test program that calls multiple libpng functions to load a PNG file and parse multiple of its features, such as PNG info, color type, bit depth, and image data. The code for this program can be found in appendix A. We tested this program both using libpng 1.6.0 and the latest version, which is 1.6.21. The program tests the following libpng functions:

- `png_init_io`
- `png_read_info`
- `png_get_image_width`
- `png_get_image_height`
- `png_get_color_type`
- `png_get_bit_depth`
- `png_set_strip_16`
- `png_set_palette_to_rgb`
- `png_set_expand_gray_1_2_4_to_8`
- `png_get_valid`
- `png_set_tRNS_to_alpha`
- `png_set_filler`
- `png_set_gray_to_rgb`
- `png_read_update_info`
- `png_get_rowbytes`
- `png_read_image`

The test program is largely based on a libpng example program [9]. The example program has been further simplified to perform only reading, as we consider a minimized program best to focus fuzzing on the actual library and not on wrapper code. Furthermore, the calls to `abort()` have been replaced with `return` statements, to not fool the testing programs into detection of false crashes. Lastly, a check validating a proper result of the call to `malloc` was introduced.

6.5 Progressive Reader Test

CVE-2014-0333 describes a vulnerability of libpng that can hang the library by reading an image with a zero-length IDAT chunk using the progressive streaming reader.

Firstly, we constructed a wrapper program that calls the libpng progressive streaming reader to load PNG images. The program is largely based on a progressive reader example program [17], but simplified for the same reasons the normal reader program is simplified. The code performs no other action than opening a PNG file and passing it to the progressive streaming reader function `png_process_data`. The callback functions of the streaming reader have been set to `null`, which is an allowed way of calling the reader function. Like the normal reader test, we tested this program with both libpng 1.6.0 and version 1.6.21.

Secondly, we determined what exactly is meant by a "zero-length IDAT chunk". We considered three options that could be described as such. The first option is an IDAT chunk with just zeroes in its length field, but with data in the data field. The second is an IDAT chunk with a length larger than zero in the length field, but no data in the data field. Lastly, we considered an IDAT chunk with both only zeroes in the length field, and no data in the data field.

To validate which of these cases triggers the hanging behavior as described in CVE-2014-0333, we constructed three PNG images, each with one of the distinct IDAT chunks described above. These images were then loaded by the wrapper program which passed them to the progressive streaming reader function. We concluded that only the first case causes the program to hang: a PNG image with an IDAT chunk that has a length field containing only zeroes, but with data in the data field. However, later reasoning led us to conclude that if a length field is set to zero, there is no actual data field, but the type field is immediately followed by the CRC field. This means that all data after the first four bytes following the IDAT type field, are interpreted as a new chunk.

After validating that the vulnerability is present in libpng 1.6.0, we loaded the wrapper program into afl-fuzz. afl-fuzz almost immediately presented us with one unique generated hang. After running the program for over 20 minutes, no inputs causing new unique hangs or crashes were generated, and we decided to stop fuzzing. The input generating this hang is described in section 8.2.

7. WRAPPER CODE CORRECTION

In this section we want to emphasize the importance of correct wrapper code, as faults in this code can lead to false results and a distorted view of the targeted library. We illustrate this by our initial findings for the normal reader test.

In the initial version of this wrapper program, the lines shown in listing 1 were missing. We fuzzed the program using libpng 1.6.0 and this resulted in six crash reports and eight hang reports. We validated that all six test cases reported to crash caused a segmentation fault without returning an error message. Upon inspection of the test cases reported to hang, they either were duplicates of the crashing test cases or were other cases which did not really hang.

```
png_set_crc_action(png_ptr, PNG_CRC_QUIET_USE,
↪  PNG_CRC_QUIET_USE);
```

```
if (row_pointers == NULL) return;
```

Listing 1: Missing lines of wrapper code

We inspected the binary constitution of the generated test cases, and found two similarities between all six files. Firstly, the height of the IHDR chunk – encoded in data bytes 5 to 8 – is set to the value 64. Secondly, all images are missing the entire IEND chunk. Other than that, the test cases only differ in ancillary chunks.

7.1 Validating the Test Results

To determine which of the common features actually causes the crash we manually mutated a properly formatted PNG file. We created one version with the first byte of the height set to 64, one version without an IEND chunk, and one version with both these features.

Testing the program with these files revealed PNG files with the first byte of the height field set high cause segmentation faults. The file missing the IEND chunk did not cause a program crash. The fact that all generated test files were missing the IEND chunk must have been a side effect of afl-fuzz's mutation.

Since no bug related to the height field is documented on the libpng homepage, we evaluated whether it is still present in the latest version of libpng. It turned out we could not trigger the bug for the latest version of libpng, nor for any other version after 1.6.9.

Because the normal reader program calls many functions, the crash alone does not reveal in which function the bug can be found. We therefore used the GNU Debugger[1] to figure out what function caused the crash. To our surprise, we discovered the allocation in listing 2 caused the segmentation fault.

```
row_pointers[y] = (png_byte*)
↪  malloc(png_get_rowbytes(png,info));
```

Listing 2: Allocation causing segmentation fault

Initially this did not seem to make any sense, as we could not explain why our code worked for libpng versions 1.6.10 and later, but it crashed on earlier versions. Upon further inspection, we discovered that in version 1.6.10 a bug was fixed in libpng that related to wrong CRC fields. Before version 1.6.10, wrong CRCs caused libpng to trigger a warning, and continue program execution. Since version 1.6.10, wrong CRCs trigger an error, and libpng subsequently calls `longjmp()`. This function is used to pass errors to the program calling the library, and in our wrapper program is caught by the code in listing 3, which subsequently ends program execution by calling `return`.

```
if(setjmp(png_jmpbuf(png))) {
    return;
}
```

Listing 3: libpng error catching in wrapper code

Because afl-fuzz generates many test files with wrong CRCs, we were already used to seeing warnings about this, and paid no further attention to it. However, it turns out that in version 1.6.10 and later, this error prevented us from attempting to allocate an enormous amount of memory, based on the height field of the PNG. The code for this allocation can be found in listing 4.

```
row_pointers = (png_bytep*)
↪  malloc(sizeof(png_bytep) * height);
```

Listing 4: Attempt to allocate memory according to PNG height

This thus meant that the CRC warning before version 1.6.10 allowed our program execution to continue, so our own wrapper code eventually caused a segmentation fault whenever memory allocation failed. In versions 1.6.10 and later, the CRC error prevented us from ever reaching this code. Indeed, our code would have also failed in later versions of libpng had we used test files with proper CRCs.

Ironically, the bug in our own code has revealed a change in behavior between library versions. Also, the fact that faulty CRCs caused only a warning, and not an error, prior to version 1.6.10 may be considered a bug in itself, as these faults should have triggered errors. This is however a semantical bug, and not something that can normally be discovered by fuzz testing alone. It was the process of researching the unexpected behavior that revealed this

[1]https://www.gnu.org/software/gdb/

bug, and without this research and interpretation, the bug could have not been discovered by afl-fuzz.

This example illustrates how a simple mistake in wrapper code can easily lead to false testing results. Furthermore, it demonstrates how unveiling the actual mistake can take some significant effort but may result in further insights. We corrected our code with the lines from listing 1. The first of these lines disables CRC checks, as these are irrelevant for fuzzing and actually cause many test cases to fail needlessly. This line was also added to the wrapper code for the progressive streaming reader. The second line corrects an error in our own normal reader wrapper code.

8. RESULTS

This section describes the test cases generated by afl-fuzz for libpng 1.6.0 and 1.6.21. These tests used the final version of the wrapper code as can be found in appendices A and B.

8.1 Normal Reader Results

For both libpng 1.6.0 and 1.6.21, afl-fuzz covered 141 paths in total when fuzzing the normal reader wrapper program. In both cases, it discovered four of these paths itself, and 137 related directly to the supplied test files. Seven unique hangs were reported for the test of libpng 1.6.0, and eight for version 1.6.21. We manually tested these generated test files and found that none of them caused a program hang. Instead, they all generated errors which caused program execution to stop. This means afl-fuzz has found no bugs in this wrapper program for either of the tested versions.

8.2 Progressive Reader Results

The result of the fuzz test for the progressive reader using libpng 1.6.0 was one image, which was a test case that caused a hang. During the test, afl-fuzz has covered 137 paths total. Firstly, we confirmed that the test case indeed causes a hang when loaded by our wrapper program. We did so by calling the wrapper program from the command line and validating that it did not finish, as it would have done when calling it with a valid image.

Secondly, we inspected the binary data of the image with a hex editor. This showed that the IDAT chunk had a length field containing only zeroes, and that the data field was not empty. The format of this image is thus exactly in line with our expectations as described in section 6.5.

Testing version 1.6.21, afl-fuzz covered 137 paths, which all related directly to the 137 input test files. One hang was reported by afl-fuzz, but upon inspection this did not turn out to be a real hang, as program execution ended quickly when manually calling the test program with this input.

9. EVALUATION

We shortly explain our evaluation of the results we got from fuzz-testing libpng for both wrapper programs. All hangs reported by afl-fuzz for the normal reader test turned out to not actually hang. The test cases all generated libpng errors which require some time to process. This processing time passed the hanging threshold time which caused them to generate hang reports by afl-fuzz.

As the bug in the progressive reader function has already been documented, we can assume it has been well-tested. However, for completeness we validated this bug. We took a properly formatted PNG file and manually mutated its IDAT chunk length field to contain only zeroes. Calling the wrapper program with this mutated image gives the expected result as it causes a program hang. This leads us to conclude that this one malformed feature is indeed the cause of the hang.

10. DISCUSSION

Fuzzing can take a long time when testing large programs with complex execution paths. On the other hand, our approach shows that wrapper programs on a couple of library functions can remain simple enough to quickly deliver relevant fuzzing results. However, this approach has also shown fuzzing can deliver many false results which require manual validation. As we demonstrated in section 7, the cause of bugs is not always clear and it may be laborious to discover.

So far we have referred to the reported problems as bugs. Vulnerabilities are a class of bugs that allow the bug to be exploited in a malicious context, thus causing a security issue. To classify bugs as a vulnerabilities we need to discuss whether the bugs can be exploited.

libpng is used in client software such as web browsers. If a web browser consistently hangs upon receiving a malicious image, an attacker can exploit this to effectively deny users with a vulnerable browser access to a certain website. The existence of a CVE identifier for the bug causing a system hang suggests this bug can be considered a vulnerability. Indeed, the description of CVE-2014-0333 states this hang can be used for a denial of service attack.

The semantical bug we discovered in libpng prior to version 1.6.10 can however not be exploited by an attacker. Supplying an image with a wrong CRC to libpng is not something that by itself is exploitable, as either the code continues and the image is not displayed, or all other PNG fields are parsed properly. This bug is thus not a vulnerability.

These results demonstrate that fuzzing can be used to discover real bugs in programs, that may otherwise remain undetected for a prolonged period. Furthermore, using small test programs and a good set of test files allows fuzz testing to deliver results quickly and consistently.

All hangs reported by afl-fuzz when testing libpng 1.6.21 turned out to be false reports, and testing generated no crash reports. This does not necessarily mean that this version contains no vulnerabilities or other bugs. Firstly, it should be noted that fuzz-testing can only discover limited classes of bugs, amongst which are no semantical bugs. Secondly, fuzz-testing uses random input and may therefore miss edge cases that trigger certain bugs. Because of these reasons, it would not be correct to assume that a lack of bug reports guarantees that no bugs are present in the tested software. We can however safely say that bugs causing software crashes and hangs are *unlikely* to exist in the tested functions.

11. CONCLUSION

In this work, we have demonstrated how wrapper programs for software libraries can be used to fuzz test these libraries. We have shown how it is important to validate fuzz testing results as false bug reports are likely, either because of bugs in wrapper code or because detecting hangs is hard. We have illustrated the importance of properly written wrapper programs, and shown how semantical bugs can be discovered upon analysis of unexpected program behavior.

Furthermore, we have demonstrated that given a strong testing corpus and proper wrapper programs, fuzz testing is a simple and quick process. Using fuzz testing, we have confirmed that bug CVE-2014-0333 could have easily been discovered before deployment. With this case, we have demonstrated that fuzz testing delivers actual results.

We have also demonstrated that it is hard to use this approach with the angr testing framework, as it is incomplete and may run into problems due to unimplemented functionality. However, we have also shown the possibility of functional tests using angr, in cases where program analysis does not run into unimplemented functionality.

12. REFERENCES

[1] american fuzzy lop. `http://lcamtuf.coredump.cx/afl/`. Accessed: April 4 2016.

[2] angr, a binary analysis framework. `http://angr.io`. Accessed: April 9 2016.

[3] Introduction | angr documentation. `http://docs.angr.io/`. Accessed: April 9 2016.

[4] libpng. `http://libpng.org/pub/png/libpng.html`. Accessed: April 4 2016.

[5] Loading static library for analysis #45. `https://github.com/angr/angr/issues/45`. Accessed: April 9 2016.

[6] pip install in virtualenv installs lib to wrong directory #445. `https://github.com/aquynh/capstone/issues/445`. Accessed: April 10 2016.

[7] T. Boutell et al. PNG (portable network graphics) specification version 1.0. RFC 2083, RFC Editor, March 1997.

[8] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[9] G. Cottenceau and Y. Niwa. A simple libpng example program. `https://gist.github.com/niw/5963798`. Accessed: April 6 2016.

[10] C. Holler, K. Herzig, and A. Zeller. Fuzzing with code fragments. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 445–458, 2012.

[11] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun. jfuzz: A concolic whitebox fuzzer for java. In *NASA Formal Methods*, pages 121–125, 2009.

[12] R. Krak and J. Diesvelt. A survey of automatic software testing techniques. `https://github.com/TUDelft-CS4110/2016-los-piratas-informaticos/blob/master/los-piratas-informaticos-summary.pdf`.

[13] N. Nakka, G. P. Saggese, Z. Kalbarczyk, and R. K. Iyer. An architectural framework for detecting process hangs/crashes. In *Dependable Computing-EDCC 5*, pages 103–121. Springer, 2005.

[14] K. Sen and G. Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *Computer Aided Verification*, pages 419–423. Springer, 2006.

[15] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS*, 2015.

[16] W. van Schaik. PngSuite. `http://www.schaik.com/pngsuite/`. Accessed: April 6 2016.

[17] N. Whiteford. Progressive PNG rendering using libpng, quick example. `http://41j.com/blog/2012/12/`. Accessed: April 6 2016.

APPENDIX

A. WRAPPER CODE FOR NORMAL PNG READING

normalread.c

```c
/* A simple libpng example program
 * based on http://zarb.org/~gc/html/libpng.html
 *
 * Modified by Yoshimasa Niwa to make it much simpler
 * and support all defined color_type.
 *
 * Further modified to serve as a wrapper program for testing.
 *
 * Compile using the following instruction
 * gcc normalread.c -lpng16 -std=c99
 * call the function with a PNG file as argument
 * proper execution ends with no output
 *
 * Copyright 2002-2010 Guillaume Cottenceau.
 *
 * This software may be freely redistributed under the terms
 * of the X11 license.
 */

#include <stdlib.h>
#include <stdio.h>
#include <png.h>

int width, height;
png_byte color_type;
png_byte bit_depth;
png_bytep *row_pointers;

void test_png_file(char *filename) {
  FILE *fp = fopen(filename, "rb");

  png_structp png = png_create_read_struct(PNG_LIBPNG_VER_STRING, NULL, NULL, NULL);
  png_set_crc_action(png, PNG_CRC_QUIET_USE, PNG_CRC_QUIET_USE);

  if(!png) {
    return;
  } else {
  png_infop info = png_create_info_struct(png);
  if(!info) {
    return;
  } else {
  if(setjmp(png_jmpbuf(png))) {
    return;
  } else {

    png_init_io(png, fp);

    png_read_info(png, info);

    width      = png_get_image_width(png, info);
    height     = png_get_image_height(png, info);
    color_type = png_get_color_type(png, info);
    bit_depth  = png_get_bit_depth(png, info);

    // Read any color_type into 8bit depth, RGBA format.
    // See http://www.libpng.org/pub/png/libpng-manual.txt

    if(bit_depth == 16)
    png_set_strip_16(png);
    if(color_type == PNG_COLOR_TYPE_PALETTE)
    png_set_palette_to_rgb(png);

    // PNG_COLOR_TYPE_GRAY_ALPHA is always 8 or 16bit depth.
    if(color_type == PNG_COLOR_TYPE_GRAY && bit_depth < 8)
    png_set_expand_gray_1_2_4_to_8(png);

    if(png_get_valid(png, info, PNG_INFO_tRNS))
    png_set_tRNS_to_alpha(png);

    // These color_type don't have an alpha channel then fill it with 0xff.
    if(color_type == PNG_COLOR_TYPE_RGB ||
     color_type == PNG_COLOR_TYPE_GRAY ||
     color_type == PNG_COLOR_TYPE_PALETTE)
    png_set_filler(png, 0xFF, PNG_FILLER_AFTER);

    if(color_type == PNG_COLOR_TYPE_GRAY ||
     color_type == PNG_COLOR_TYPE_GRAY_ALPHA)
    png_set_gray_to_rgb(png);

    png_read_update_info(png, info);

    row_pointers = (png_bytep*)malloc(sizeof(png_bytep) * height);
    if (row_pointers == NULL) return;
    for(int y = 0; y < height; y++) {
    row_pointers[y] = (png_byte*)malloc(png_get_rowbytes(png,info));
    }

    png_read_image(png, row_pointers);

    fclose(fp);
  }}}
}

int main(int argc, char *argv[]) {
  if(argc != 2) abort();

  test_png_file(argv[1]);

  return 0;
}
```

B. WRAPPER CODE FOR PROGRESSIVE PNG READING

progressiveread.c

```
/* Progressive PNG rendering using libpng, quick example
 * based on http://41j.com/blog/2012/12/
 *
 * Compile using the following instruction
 * gcc progressiveread.c -lpng16 -std=c99
 * call the function with a PNG file as argument
 * proper execution ends with no output
 */

#include <stdlib.h>
#include <png.h>

png_structp png_ptr;
png_infop   info_ptr;

int initialize_png_reader() {
  png_ptr = png_create_read_struct (PNG_LIBPNG_VER_STRING, (png_voidp)NULL,NULL,NULL);
```

```c
  if(!png_ptr) return 1;

  png_set_crc_action(png_ptr, PNG_CRC_QUIET_USE, PNG_CRC_QUIET_USE);
  info_ptr = png_create_info_struct(png_ptr);

  if(!info_ptr) {
    png_destroy_read_struct(&png_ptr, (png_infopp)NULL, (png_infopp)NULL);
    return 1;
  }
  if(setjmp(png_jmpbuf(png_ptr))) {
    png_destroy_read_struct(&png_ptr, &info_ptr, (png_infopp)NULL);
    return 1;
  }
  png_set_progressive_read_fn(
      png_ptr, (void *)NULL, (void *)NULL, (void *)NULL, (void *)NULL);
  return 0;
}

int process_data(png_bytep buffer, png_uint_32 length) {

  if (setjmp(png_jmpbuf(png_ptr))) {
    png_destroy_read_struct(&png_ptr, &info_ptr, (png_infopp)NULL);
    return 1;
  }

  png_process_data(png_ptr, info_ptr, buffer, length);

  return 0;
}

int main(int argc, char **argv) {
  if(argc != 2) abort();
  initialize_png_reader();

  FILE *fp = fopen(argv[1], "rb");

  char buffer[1025];
  int length = fread(buffer,1,1024,fp);
  process_data(buffer,length);
  fclose(fp);

  return 0;
}
```

C. SAMPLE CODE FOR TESTING ANGR FUNCTIONALITIES

branch.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

void test(char *input)
{
    if(strcmp(input, "42") == 0) {
        abort(); // Simulate a crash
    } else {
        printf("Normal execution continues\n");
    }
}

int main(int argc, char** argv) {
    char in[20];
    read(0, in, 8);
    test(in);
}
```

loop.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

void infloop()
{
    while(1) {
        printf("loop");
    }
}

void test(char *input)
{
    char* backdoor = "42";

    if(strcmp(input,backdoor) == 0) {
        infloop(); // Simulate an infinite loop
    } else {
        printf("Normal execution continues\n");
    }
}

int main(int argc, char** argv) {
    char in[20];
    read(0, in, 8);
    test(in);
}
```

recurfunc.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

void infloop()
{
    printf("loop");
    infloop();
}

void test(char *input)
{
    char* backdoor = "42";

    if(strcmp(input,backdoor) == 0) {
        infloop(); // Simulate an infinite loop
    } else {
        printf("Normal execution continues\n");
    }
}

int main(int argc, char** argv) {
    char in[20];
    read(0, in, 8);
    test(in);
}
```

explosion.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

int main(int argc, char** argv) {
    char in[20];
    read(0, in, 8);

    char* secret = "42";

    while(strcmp(in,secret) != 0) {
        read(0, in, 8);
    }
}
```

exhaust.py

```python
#!/usr/bin/env python

import angr
import time
import sys

def main():
    if len(sys.argv) < 2:
        print 'argument missing. give a binary to test.'
        return

    p = angr.Project(sys.argv[1])

    state = p.factory.entry_state() # address of entry point
    path = p.factory.path(state) # initial path from entry point
    pg = p.factory.path_group(path) # pathgroup from entry point

    e = p.surveyors.Explorer()

    while e.done == False:
        e.run(100)
        print time.time(),' ',e
    print 'done'

    for i in range(0,len(e.deadended)):
        print 'deadend ',i,': ',e.deadended[i]
        try:
            print 'in -> ', e.deadended[i].state.posix.dumps(0)
        except:
            print 'no input generated'
        try:
            print 'out -> ',e.deadended[i].state.posix.dumps(1)
        except:
            print 'no output generated'

def test(): pass

if __name__ == '__main__':
    main()
```

D. ANGR SCRIPT FOR TESTING NORMALREAD.C

explorer.py

```python
#!/usr/bin/env python

import angr
import simuvex
import time

def explorer():
    load_options = {}
    load_options['auto_load_libs'] = True
    load_options['force_load_libs'] = ['libpng16.so','libpng.so','libpng16.so.0']

    p = angr.Project('normalread', load_options=load_options)

    state = p.factory.entry_state() # address of entry point
    path = p.factory.path()

    e = p.surveyors.Explorer()

    maxRuns = 100000
    step = 10
    run = 0

    deCount = 0
    erCount = 0

    while e.done == False and run < maxRuns:
        e.run(step)
        run += step
        print time.time(),' ',e

        if len(e.deadended) > deCount:
            for i in range(deCount,len(e.deadended)):
            print 'deadend ',i,': ',e.deadended[i]
                try:
                    print 'in -> ', e.deadended[i].state.posix.dumps(0)
                except: pass
                try:
                    print 'out -> ',e.deadended[i].state.posix.dumps(1)
                except: pass
                try:
                    print 'err -> ',e.deadended[i].state.posix.dumps(2)
                except: pass
            deCount = len(e.deadended)

        if len(e.errored) > erCount:
            for i in range(erCount,len(e.errored)):
            print 'error ',i,': ',e.errored[i]
                try:
                    print 'in -> ', e.errored[i].state.posix.dumps(0)
                except: pass
                try:
                    print 'out -> ',e.errored[i].state.posix.dumps(1)
                except: pass
                try:
                    print 'err -> ',e.errored[i].state.posix.dumps(2)
                except: pass
            erCount = len(e.errored)

    print 'done'

```

```
def test(): pass

if __name__ == '__main__':
    print explorer()
```

simufile.py

```
#!/usr/bin/env python

import angr
import simuvex

def main():
    p = angr.Project('normalread') # adjusted version of normalread

    state = p.factory.blank_state(remove_options={simuvex.s_options.LAZY_SOLVES})

    image_name = "test.png"

    content = simuvex.SimSymbolicMemory(memory_id="file_%s" % image_name)
    content.set_state(state)

    #storing content in the simulated file can decrease number of paths and constraint solving time
    #f = open('test.png', 'r')
    #content.store(0, f.read())

    image_file = simuvex.SimFile(image_name, 'rw', size=255, content=content)

    state.posix.fs = {image_name: image_file} # set state filesystem

    path = p.factory.path(state=state)
    ex = p.surveyors.Explorer(start=path)

    deCount = 0

    while len(ex.active) > 0:
    ex.run(10)
        print ex
    if len(ex.deadended) > deCount:
            for x in range(deCount,len(ex.deadended)):
                try:
            print ex.deadended[x].state.posix.dumps(1) # dump the std output
                except: pass
                try:
                    print ex.deadended[x].state.posix.dumps(2) # dump the std err
                except: pass
            deCount = len(ex.deadended)

def test(): pass
if __name__ == '__main__':
    main()
```