ELF Parsing Bugs by Example with Melkor Fuzzer

Alejandro Hernández
(@nitr0usmx)
IOActive Senior Security Consultant

Abstract

Too often the development community continues to blindly trust the metadata in Executable and Linking Format (ELF) files. In this paper, Alejandro Hernández walks you through the testing process for seven applications and reveals the bugs that he found. He performed the tests using Melkor, a file format fuzzer he wrote specifically for ELF files.

IOActive
Hardware | Software | Wetware
SECURITY SERVICES

Contents

Introduction

The ELF file format, like any other file format, is an array of bits and bytes interconnected through data structures. When interpreted by an ELF parser, an ELF file makes sense, depending upon the parsing context: runtime (execution view) or static (linking view).

In 1999, ELF was chosen as the standard binary file format for *NIX systems, and now, about 15 years later, we are still in many instances blindly trusting the (meta)data within ELF files, either as executable binaries, shared libraries, or relocation objects.

However, blind trust is not necessary. Fuzzing tools are available to run proper safety checks for every single untrusted field.

To demonstrate, I tested and found bugs in seven applications using Melkor, a file format fuzzer specifically for ELF files that I developed:

https://github.com/IOActive/Melkor_ELF_Fuzzer.

The following were tested:

- HT Editor 2.1.0
- GCC (GNU Compiler) 4.8.1
- Snowman Decompiler v0.0.5
- GDB (GNU Debugger) 7.8
- IDA Pro (Demo version) 6.6.140625
- OpenBSD 5.5 ldconfig
- OpenBSD 5.5 Kernel

Most, if not all, of these bugs were reported to the vendors or developers.

Almost all, if not all, were only crashes (invalid memory dereferences) and I did not validate whether they're exploitable security bugs. Therefore, please do not expect a working command execution exploit at the end of this white paper.

Melkor is an intuitive and, therefore, easy-to-use fuzzer. To get started, you simply identify:

- The kind of metadata you want to fuzz
- A valid ELF file to use as a template
- The number of desired test cases you want to generate (malformed ELF files that I call 'orcs,' as shown in my Black Hat Arsenal presentation, slides 51 and 52.[1]
- The likelihood of each fuzzing rule as a percentage

Options supported by Melkor:

For a quiet output, use the `-q` switch.

1. - Melkor Test of HT Editor 2.1.0

HT (http://hte.sourceforge.net) is my favorite ELF editor. It has parsers for all internal metadata.

Test Case Generation

To start, we'll fuzz only the ELF header, with a 20% chance of executing each fuzzing rule, to create 1000 test cases:

```
$./melkor -H templates/foo -l 20 -n 1000
```

You will find the test cases that are generated in the orcs_foo directory along with a detailed report explaining what was fuzzed internally.

Fuzzing the Parser

You could perform manually testing by supplying each orc (test case) as a parameter to the HT Editor. However, it would take a long time to test 1000 test cases.

For that reason, Melkor comes with two testing scripts:

- For Linux, test_fuzzed.sh
- For Windows systems, win_test_fuzzed.bat

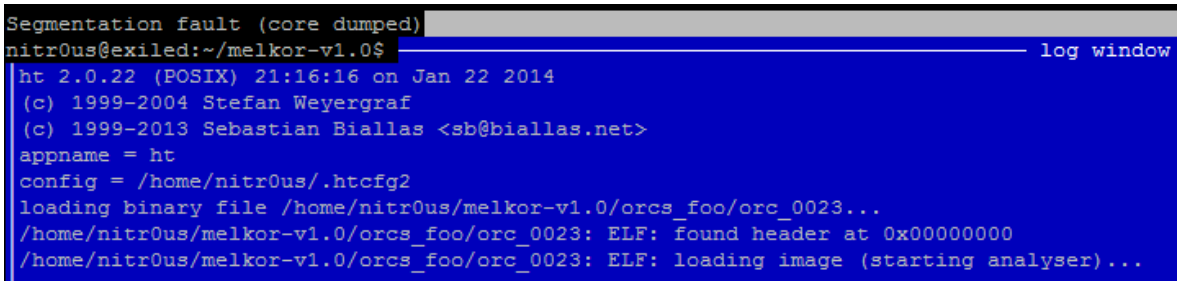
To test the scripts automatically, enter:

```
$./test_fuzzed.sh orcs_foo/ "ht"
```

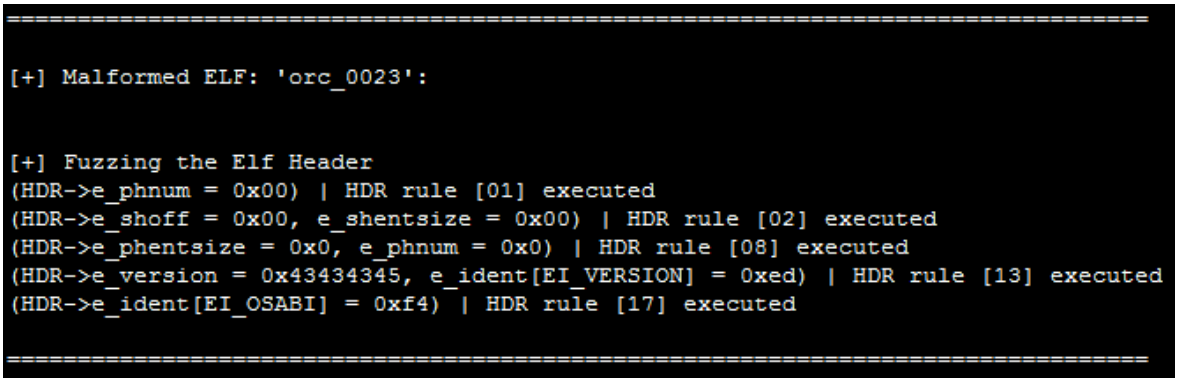
Every time HT Editor opens a valid ELF file, you must press the [F10] key to continue to the next test case.

The Bug

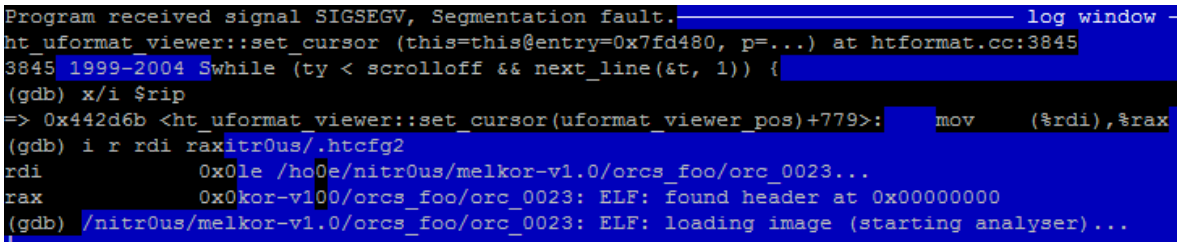
After 22 tries, the test case orc_0023 crashed the HT Editor:



The next step is to identify the cause of the crash by reading the detailed report generated by Melkor:



By debugging it with GDB, you would see:



Effectively, there is a NULL pointer dereference in the instruction `mov (%rdi),%rax`.

2. - Melkor Test of GCC (GNU Compiler) 4.8.1

I consider the GCC to be the compiler of excellence.

When you type `gcc foo.c -o foo`, you're performing all the phases (compilation, linking, etc.); however, if you want only to compile, the `-c` is necessary, as in `gcc -c foo.c`, to create the ELF relocatable object foo.o.

Normally, relocations and/or symbols tables are an important part of the .o objects. This is what we are going to fuzz.

Test Case Generation

Inside the templates/ folder, a foo.o file is compiled with the same Makefile to create Melkor, which in turn will be used as a template to create 5000 (default -n option) malformed relocatable files. We instruct Melkor to fuzz the relocations within the file (`-R`) and the symbol tables (`-s`) as well:

```
$./melkor -Rs templates/foo.o
```

During the fuzzing process, you may see verbose output:

```
================================================================================
[+] Malformed ELF 'orc_4995.o':

[+] Fuzzing the relocations section .rela.text with 14 SHT_RELA entries

[+] Fuzzing the relocations section .rela.debug_info with 59 SHT_RELA entries
. SHT[7] REL[2] rule [02] executed
. SHT[7] REL[6] rule [02] executed
. SHT[7] REL[17] rule [02] executed
. SHT[7] REL[42] rule [02] executed

[+] Fuzzing the relocations section .rela.debug_aranges with 2 SHT_RELA entries

[+] Fuzzing the relocations section .rela.debug_line with 1 SHT_RELA entries

[+] Fuzzing the relocations section .rela.eh_frame with 1 SHT_RELA entries

[+] Fuzzing the Symbol Table .symtab with 25 entries
. SHT[19] SYM[0] rule [01] executed
. SHT[19] SYM[2] rule [15] executed
. SHT[19] SYM[3] rule [04] executed
```

Fuzzing the Parser

In order to test GCC with every malformed .o object, a command like `gcc -o output malformed.o` must be executed. To do so automatically, the following arguments are supplied to the testing script:

```
$./test_fuzzed.sh orcs_foo.o/ "gcc -o output"
```

You can observe how mature GCC is and how it properly handles every malformed struct, field, size, etc.:

```
Press any key to start the testing...
--------------------------------------------------
Testing program: gcc -o output orcs_foo.o/orc_0001.o
/usr/bin/ld: orcs_foo.o/orc_0001.o: access beyond end of merged section (1073804207)
orcs_foo.o/orc_0001.o: In function `main':
foo.c:(.text+0x8a): undefined reference to `help'
foo.c:(.text+0x91): relocation truncated to fit: R_X86_64_PC32 against symbol `stdout@@GLIBC_2.2.5' defined in .dynbss
./../x86_64-linux-gnu/crt1.o
/usr/bin/ld: orcs_foo.o/orc_0001.o(.debug_info+0xbeef): reloc against `.debug_str': error 2
/usr/bin/ld: final link failed: Nonrepresentable section on output
collect2: error: ld returned 1 exit status
--------------------------------------------------
Testing program: gcc -o output orcs_foo.o/orc_0002.o
/usr/bin/ld: orcs_foo.o/orc_0002.o: bad reloc symbol index (0xb16b00b5 >= 0x19) for offset 0x4a in section `.text'
orcs_foo.o/orc_0002.o: error adding symbols: Bad value
collect2: error: ld returned 1 exit status
--------------------------------------------------
Testing program: gcc -o output orcs_foo.o/orc_0003.o
/usr/bin/ld: orcs_foo.o/orc_0003.o: bad reloc symbol index (0x1e >= 0x19) for offset 0x71 in section `.debug_info'
orcs_foo.o/orc_0003.o: error adding symbols: Bad value
```

The Bug

Normally, in a Linux system, when a program fails due to memory corruption or an invalid memory dereference, it writes to STDERR the message: "*Segmentation fault.*" As a quick way to identify if we found bugs in the linker, we can simply look for that message in the output of the testing script (the script already redirected the STDERR of each test case to STDOUT).

```
$./test_fuzzed.sh orcs_foo.o/ "gcc -o output" | egrep "Testing program|Segmentation fault"
```

```
Testing program: gcc -o output orcs_foo.o/orc_0280.o
Testing program: gcc -o output orcs_foo.o/orc_0281.o
collect2: error: ld terminated with signal 11 [Segmentation fault], core dumped
Testing program: gcc -o output orcs_foo.o/orc_0282.o
Testing program: gcc -o output orcs_foo.o/orc_0283.o
collect2: error: ld terminated with signal 11 [Segmentation fault], core dumped
Testing program: gcc -o output orcs_foo.o/orc_0284.o
Testing program: gcc -o output orcs_foo.o/orc_0285.o
Testing program: gcc -o output orcs_foo.o/orc_0286.o
Testing program: gcc -o output orcs_foo.o/orc_0287.o
Testing program: gcc -o output orcs_foo.o/orc_0288.o
Testing program: gcc -o output orcs_foo.o/orc_0289.o
Testing program: gcc -o output orcs_foo.o/orc_0290.o
Testing program: gcc -o output orcs_foo.o/orc_0291.o
collect2: error: ld terminated with signal 11 [Segmentation fault], core dumped
Testing program: gcc -o output orcs_foo.o/orc_0292.o
Testing program: gcc -o output orcs_foo.o/orc_0293.o
Testing program: gcc -o output orcs_foo.o/orc_0294.o
Testing program: gcc -o output orcs_foo.o/orc_0295.o
Testing program: gcc -o output orcs_foo.o/orc_0296.o
```

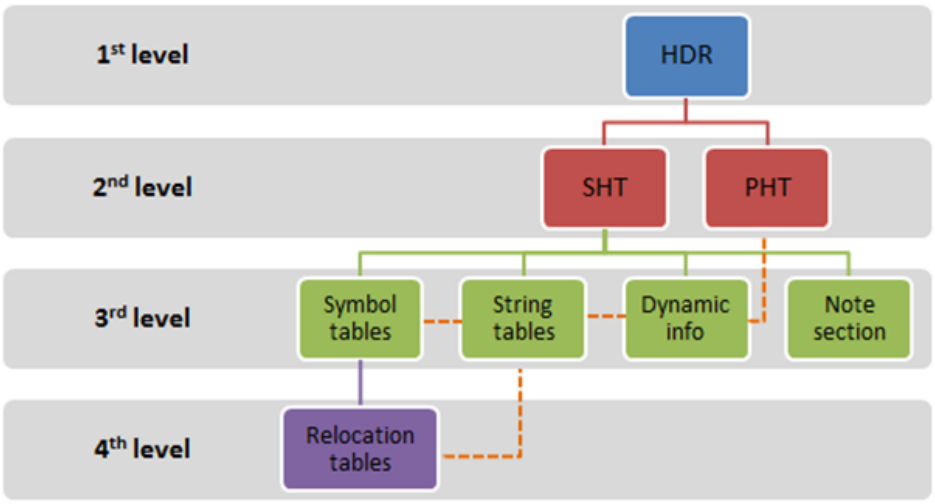
Filtering for only those that ended with a "*Segmentation fault*," I saw that 197 of 5000 test cases triggered a bug.

3. - Melkor Test of the Snowman Decompiler v0.0.5

Snowman (http://derevenets.com) is a great native code to C/C++ decompiler for Windows. It's free and supports PE and ELF formats in x86 and x86-64 architectures.

Test Case Generation

In the previous example, I could have mentioned that after a period of testing, I noticed that some applications properly validated all fields in the initial header and handled the errors. So, in order to fuzz more internal levels, I implemented the following metadata dependencies in Melkor, which shouldn't be broken:



With these dependencies, it's possible to corrupt deeper metadata without corrupting structures in higher levels. In the previous GCC example, it's evident that these dependencies were in place transparently to reach the third and fourth levels of metadata, symbol tables, and relocation tables respectively. For more about dependencies in Melkor, see Melkor Documentation: ELF Metadata Dependencies[2].

Continuing with Snowman, I created only 200 test cases with fuzzed sections in the Section Header Table (SHT), without touching the ELF header, using the default likelihood of fuzzing rules execution, which is 10%:

```
$./melkor -S templates/foo -n 200
```

Fuzzing the Parser

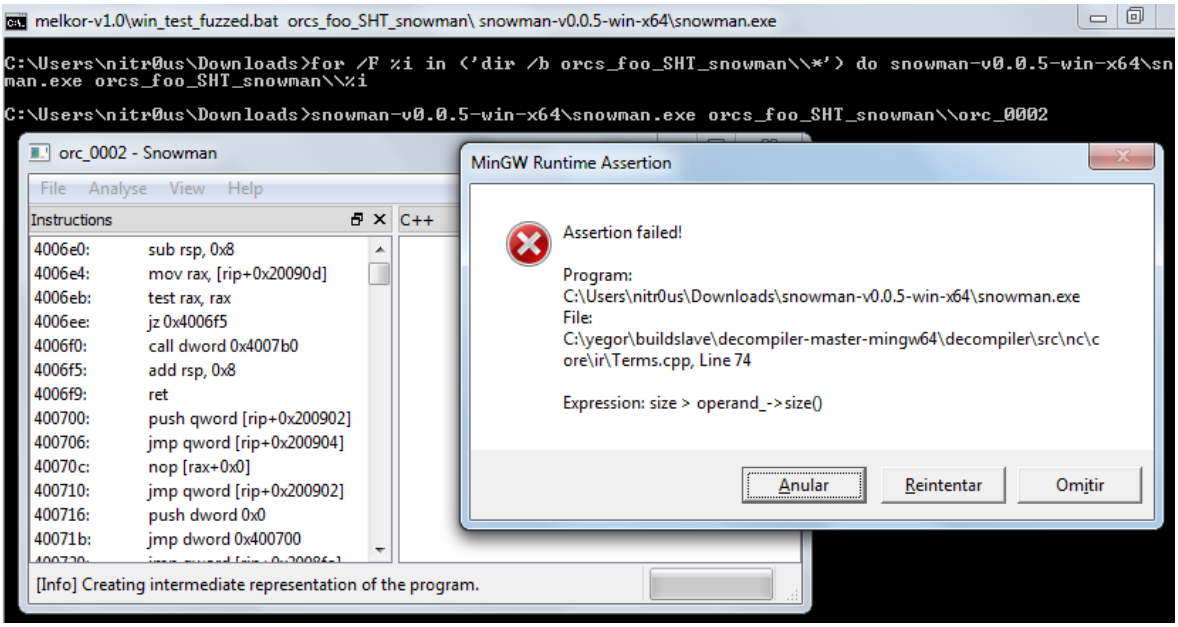
Since snowman.exe runs on Windows machines, I then copied the created test cases to the Windows box where Snowman was loaded and tested each case using win_test_fuzzed.bat as follows:

```
C:\Users\nitr0us\Downloads>melkor-v1.0\win_test_fuzzed.bat
orcs_foo_SHT_snowman\ snowman-v0.0.5-win-x64\snowman.exe
```

For every opened snowman.exe for which there is no exception, it's necessary to close the window with the [Alt] + [F4] keyboard combination. Sorry for the inconvenience but I kept the testing scripts as simple as possible.

The Bug

I was lucky on testing day. The second orc triggered an unhandled exception that made Snowman fail:



4. - Melkor Test of GDB (GNU Debugger) 7.8

GDB, the most used debugger in *NIX systems, is another great piece of code.

When you type `gdb foo`, the necessary ELF data structures and other metadata is parsed and prepared before the debugging process; however, when you execute a program within GDB, some other metadata is parsed.

Test Case Generation

Most applications rely on the SHT to reach more internal metadata; the data and the code itself, etc. As you likely noticed in the previous example and as you'll see now with GDB, malformed SHTs might crash many applications. So, I created 2000 orcs with fuzzed SHTs:

```
$./melkor -S templates/foo -n 2000
```

Fuzzing the Parser

If GDB doesn't find anything wrong with the ELF to be debugged, it will leave you at the well-known `(gdb)` prompt waiting for your input. You must type `quit` in order to return to the OS shell. Hence, to automate the testing with our script, it's necessary to comment line 79 and uncomment line 80, as shown:

```
# $2 $1$file 2>&1
echo quit | $2 $1$file 2>&1 # Example: "echo quit | gdb -q orcs/x"
```

That will automatically feed GDB with *"quit"* and continue to the other files.

Once updated, it's time to fuzz it:

```
$./test_fuzzed.sh orcs_foo/ "gdb -q"
```

As in GCC, you'll see that GDB and BFD handle many errors:

```
Testing program: gdb -q orcs_foo/orc_0212
BFD: /home/nitr0us/melkor-v1.0/orcs_foo/orc_0212: invalid string offset 32767 >= 328 for section `.bss'
BFD: /home/nitr0us/melkor-v1.0/orcs_foo/orc_0212: invalid string offset 32767 >= 328 for section `.bss'
"/home/nitr0us/melkor-v1.0/orcs_foo/orc_0212": not in executable format: Bad value
(gdb) ------------------------------------------------
Testing program: gdb -q orcs_foo/orc_0213
BFD: /home/nitr0us/melkor-v1.0/orcs_foo/orc_0213: invalid string offset 65535 >= 328 for section `.shstrtab'
BFD: /home/nitr0us/melkor-v1.0/orcs_foo/orc_0213: invalid string offset 65535 >= 328 for section `.shstrtab'
"/home/nitr0us/melkor-v1.0/orcs_foo/orc_0213": not in executable format: Bad value
(gdb) ------------------------------------------------
Testing program: gdb -q orcs_foo/orc_0214
BFD: /home/nitr0us/melkor-v1.0/orcs_foo/orc_0214: don't know how to handle section `.note.gnu.build-id' [0x
"/home/nitr0us/melkor-v1.0/orcs_foo/orc_0214": not in executable format: Bad value
(gdb) ------------------------------------------------
Testing program: gdb -q orcs_foo/orc_0215
"/home/nitr0us/melkor-v1.0/orcs_foo/orc_0215": not in executable format: File format not recognized
(gdb) ------------------------------------------------
Testing program: gdb -q orcs_foo/orc_0216
"/home/nitr0us/melkor-v1.0/orcs_foo/orc_0216": not in executable format: Memory exhausted
```

The Bug

While none of the 2000 orcs raised a segmentation fault, I manually tested some of the orcs and found that many were unable to be debugged due to "memory exhaustion":

```
nitr0us@exiled:~/orcs$ gdb -q ./gdb_mem_exhaustion
Reading symbols from ./gdb_mem_exhaustion...done.
(gdb) r
Starting program: /home/nitr0us/orcs/gdb_mem_exhaustion
utils.c:1062: internal-error: virtual memory exhausted.
A problem internal to GDB has been detected,
further debugging may prove unreliable.
Quit this debugging session? (y or n) n

This is a bug, please report it.  For instructions, see:
<http://www.gnu.org/software/gdb/bugs/>.
```

5. - Melkor Test of IDA Pro (Demo) 6.6.140625

IDA Pro is the beautifully visual debugger that runs in different OSs and supports the ELF file format.

I used the demonstration version, downloadable from the official web site: https://www.hex-rays.com/products/ida/support/download_demo.shtml.

Test Case Generation

IDA Pro validates most ELF header fields to identify whether it is possible to analyze the binary and, if not, it bypasses those validations. Only the SHT of each test case was fuzzed:

```
$./melkor -S templates/foo -n 500
```

Fuzzing the Parser

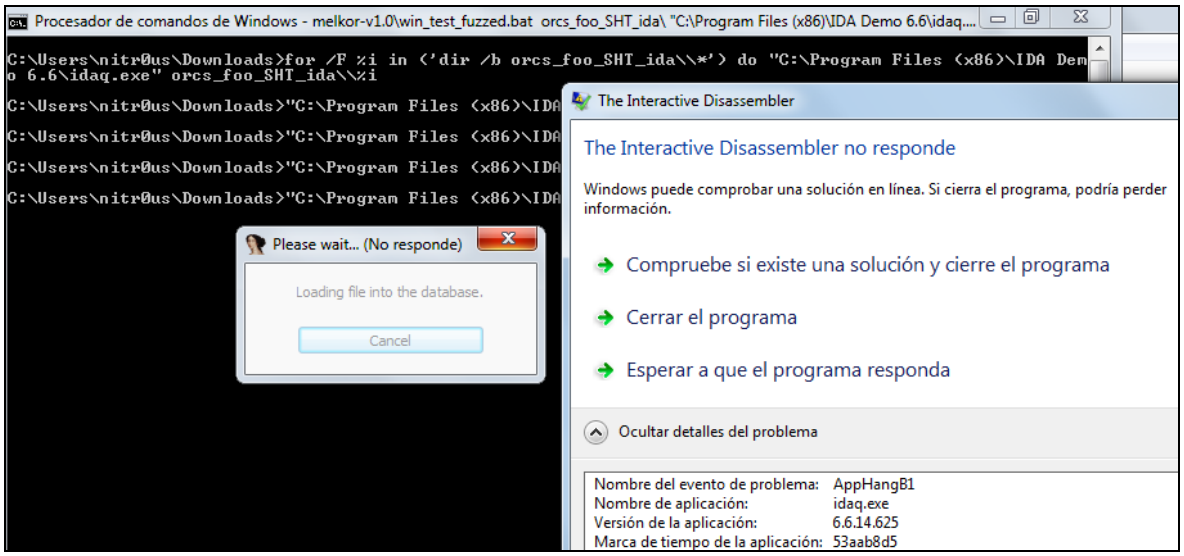
Because it was tested in a Windows environment, the win_fuzz_tested.bat was used:

```
C:\Users\nitr0us\Downloads>melkor-v1.0\win_test_fuzzed.bat
orcs_foo_SHT_ida\ "C:\Program Files (x86)\IDA Demo 6.6\idaq.exe"
```

For every opened idaq.exe for which there is no exception, it's necessary to close the window with the [Alt] + [F4] keyboard combination.

The Bug

I found a few IDA Pro bugs. In some cases, the orcs raise *"Unhandled C++ Errors"* and in others, the application just hung and created large id1 files (about 7 GB each):



6. - Melkor Test of OpenBSD ldconfig

OpenBSD, a favorite OS of mine, has a utility called ldconfig, which I tested. (Go to `$man ldconfig` for more information.)

Test Case Generation

Some test cases were created by corrupting the Program Header Table (PHT), SHT, and notes section.

```
$./melkor -PSN obsd_5.2 -n 2000
```

Fuzzing the Parser

In this case, you do not need to use test_fuzzed.sh because to run `ldconfig -P` you need only to pass the name of the directory where the libraries are located:

```
#ldconfig -Pv orcs_obsd_5.2/
```

```
root@calypso:/home/nitr0us# ldconfig -Pv orcs_obsd_5.2
loading dir orcs_obsd_5.2
processing orcs_obsd_5.2/orc_0868
processing orcs_obsd_5.2/orc_1423
processing orcs_obsd_5.2/orc_0427
processing orcs_obsd_5.2/orc_1184
processing orcs_obsd_5.2/orc_1914
processing orcs_obsd_5.2/orc_1623
processing orcs_obsd_5.2/orc_1545
processing orcs_obsd_5.2/orc_0604
processing orcs_obsd_5.2/orc_0841
processing orcs_obsd_5.2/orc_0893
processing orcs_obsd_5.2/orc_1544
processing orcs_obsd_5.2/orc_1987
Segmentation fault
root@calypso:/home/nitr0us#
```

The Bug

Most of the bugs are invalid pointer dereferences in libexec/ld.so/ldconfig/prebind.c. They occur when `elf_check_note()` is called, if the current `p_type` is `PT_NOTE` (note information).

Inside this function, ldconfig tries to access the content of a pointer plus the `p_offset` element of the current program header. If `p_offset` holds a large value, it will fail:

```
/*
 * check if the given executable header on an ELF executable
 * has the proper OpenBSD note on the file if it is not present
 * binaries will be skipped.
 */
int
elf_check_note(void *buf, Elf_Phdr *phdr)
```

```
{
        u_long address;
        u_int *pint;
        char *osname;

        address = phdr->p_offset;
        pint = (u_int *)((char *)buf + address);
        osname = (char *)buf + address + sizeof(*pint) * 3;

        if (pint[0] == 8 /* OpenBSD\0 */ &&
            pint[1] == 4 /* ??? */ &&
            pint[2] == 1 /* type_osversion */ &&
            strcmp("OpenBSD", osname) == 0)
                return 1;

        return 0;
}
```

7. - Melkor Test of OpenBSD 5.5 Kernel

OpenBSD 5.5 Kernel is the great masterpiece we all know.

Test Case Generation

To test an operating system's ELF loader, you could corrupt the PHT of an executable and then try to execute it:

```
$./melkor -P obsd_5.5 -n 3000
```

Fuzzing the Parser

Note that a second parameter in test_fuzzed.sh, which is the application to test, is not used because every file within the supplied directory will be directly executed:

```
$./test_fuzzed.sh orcs_obsd_5.5/
```

The Bug

Fuzzing the parser produced a local kernel panic. I've written about that in a separate advisory[3] which includes a proof of concept code as well.

This bug demonstrates why fuzzing, in addition to manual testing, is important; a very specific circumstance triggered the kernel panic.

Conclusion

Clearly, we would be in error if we assumed that ELF files, due to the age of the format, are free from parsing mistakes; common parsing mistakes are still found.

It would also be a mistake to assume that parsers are just in the OS kernels, readelf or objdump. Many new programs support 32 and 64-bit ELF files, and antivirus engines, debuggers, OS kernels, reverse engineering tools, and even malware may contain ELF parsers.

I hope you have seen from these examples that fuzzing is a very helpful method to identify functional (and security) bugs in your parsers in an automated fashion. An attacker could convert a single crash into an exploitable security bug in certain circumstances or those small crashes could be employed as anti-reversing or anti-infection techniques.

Feel free to fuzz, crash, fix, and/or report the bugs you find to make better software.

Happy fuzzing.

Alejandro Hernández

Acknowledgements

1. IOActive, Inc.

References

[1] Alejandro Hernández. “In the lands of corrupted elves: Breaking ELF software with Melkor fuzzer.” <https://www.blackhat.com/docs/us-14/materials/arsenal/us-14-Hernandez-Melkor-Slides.pdf>

[2] Melkor Documentation: ELF Metadata Dependencies and Fuzzing Rules. <https://github.com/IOActive/Melkor_ELF_Fuzzer/tree/master/docs>

[3] IOActive Security Advisory: OpenBSD ≤ 5.5 Local Kernel Panic. <http://www.ioactive.com/pdfs/IOActive_Advisory_OpenBSD_5_5_Local_Kernel_Panic.pdf>

About the Writer

Alejandro Hernández is a senior security consultant at IOActive, Inc., who has more than 10 years of experience in the security space. He provides security services to Fortune 500 companies and other organizations around the world. In addition to authoring Melkor, he co-authored DotDotPwn, a directory traversal fuzzer. He holds technical certifications and is a speaker at security conferences in South America and the United States. Follow Alejandro on Twitter: @nitr0usmx.

About IOActive

IOActive is a comprehensive, high-end information security services firm with a long and established pedigree in delivering elite security services to its customers. Our world-renowned consulting and research teams deliver a portfolio of specialist security services ranging from penetration testing and application code assessment through to semiconductor reverse engineering. Global 500 companies across every industry continue to trust IOActive with their most critical and sensitive security issues. Founded in 1998, IOActive is headquartered in Seattle, USA, with global operations through the Americas, EMEA and Asia Pac regions. Visit www.ioactive.com for more information. Read the IOActive Labs Research Blog: http://blog.ioactive.com/. Follow IOActive on Twitter: http://twitter.com/ioactive.