Reverse Engineering Executive Document for Lwan, a high-performance and scalable web server

(revised September 2018)

Jay Chow, Kevin Hamilton, and Yuqing Wang, MSSI graduate students, JHUISI

Abstract – Our goal of this project was to perform reverse engineering techniques that are fundamental to security assessment in any organization. For an external security assessment, these techniques help users achieve design comprehension to improve the underlying security of the software application. The open-source software application for reverse engineering is Lwan, a high-performance and scalable web server, that was compiled and ran in Linux using GCC. Our first activity included running command-line commands such as objdump, ldd, nm, strace and cat /proc/<pid>/maps to gain a better understanding of the executable generated from building this open-source web server in C. Then, we performed static analysis that utilized Understand to help us better understand the structure of code and relationships among subcomponents in many source files. We ran reverse engineering tools on the binary file for Lwan such as Binvis.io for entropy and visual analysis of byte patterns in the binary. Lastly, we created a python script that parses ELF executables and prints the relevant headers after extracting the contents. We have attached key screenshots to support our reverse engineering of Lwan.

1 Project Overview

Our goal as a group was to perform a sequence of reverse engineering tasks on an open-source application compiled and ran in Linux using GCC. After compiling Lwan in C, we performed reverse engineering on the binary file. Although we did not completely reverse engineer the application, we dumped the headers and virtual memory(when the server process was running), printed the shared library dependencies, listed the symbols, and ran strace on the server process. Then, through static analysis, we imported the project folder into SciTools Understand to capture summary metrics, a cluster call butterfly graph and control flow graph. Lastly, we ran binvis.io on the binary file to plot the entropy of a file and developed a program that parses the binary to print out the headers for a file. We identified a particular file format, reverse engineered a portion of the contents and wrote a python script to extract these contents and output them to describe the contents.

1.1 Lwan Overview

Lwan is a performance-driven and easy-to-scale web server. To be used in embedded devices and servers, Lwan is built in C with low disk and memory footprints[1]. Lwan can serve both static and dynamic content to web users and it can also be extended as a software library. Lwan handles connections individually by coroutines, which can be seen as a per-CP scheduler. The coroutines simplify the management of resources. For example, once a connection from a client is closed, memory is reclaimed back, files get closed and references are accounted for. Unlike other web servers, Lwan ensures low latency, and increased predictability compared to other garbage collectors, making the software easier to use as a user[2].

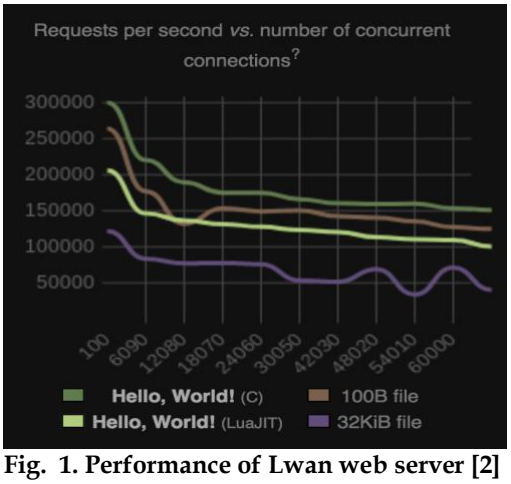

Fig. 1. Performance of Lwan web server [2]

Web services can be built on top of Lwan as a

software library. As such, it has functionalities exceeding a static file server. Lwan can be used as a library to build web services on top of it. As shown on the next page, we used ArchStudio to generate a high-level system architecture diagram using a top-down tree approach.

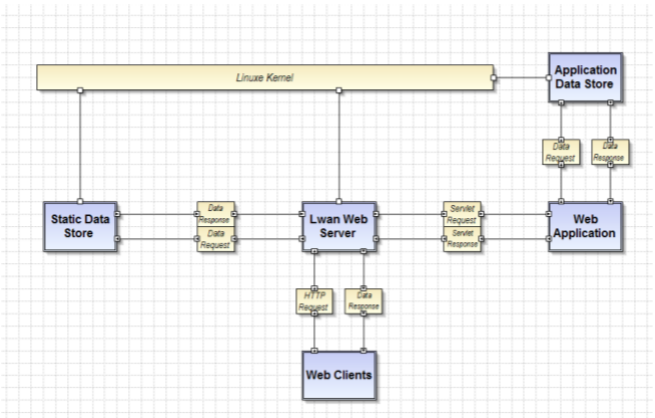

Fig. 2. Top-level system architecture diagram for Lwan using ArchStudio

1.2 Lwan Architecture

Adapted from Lwan's official webpage[7], we learned a great deal about the internals of the software implementation of Lwan. Lwan uses a low memory footprint, roughly 500KiB(Kibibyte) for 10,000 idle connections. It occupies a low disk footprint: x86-64 executable has 110KiB and a small RAM footprint with a low number of system calls to the OS. For Lwan, the HTTP request parser is hand-written and the static file server uses an efficient way according to the file size. To be more specific, the copies between the kernel and the programs are limited. And Lwan sends all these small size files through vectored I/O, the process of gathering data from or scattering data into a given set of buffers. If the client asks for a gzip compression, pre-compressed files will be sent. Lwan's architecture encompasses a wait-free multithreaded design that supports Linux, FreeBSD and macOS. It use few threads for the connection: one thread for accepting connections, another thread per logical CPU for handling the connection. Such a mode make asynchronous I/O easier with a purpose-built loop when developers use C.

Lwan's architecture includes an efficient cache that is used to list directories, file information, compressed file contents and compiled Lua scripts. There are APIs that Lwan provides to create web applications and extensions in Lua or C. The HTTP/1.0 and HTTP/1.1 in the architecture is used for supporting keep-alive connections and pipelined requests. Within Lwan's architecture, there exist PROXY protocol support, systemd socket activation, IPv6 ready support, buildbots, and a test suite in Python to test the server. [2]

PROJECT EXECUTION

First, we read the documentation for Lwan on github(https://github.com/lpereira/lwan). Then we git cloned the code across many source files from the remote repository. Then, we used CMake, a cross-platform and open-source software application, to manage the build process of Lwan using a compiler-independent method. We documented the key steps to build the entire software project and identified the source language, compiler and operating system. Then, after obtaining the binary, we dumped the headers using $ objdump -h Lwan to display section headers of the object file. Next, we printed the shared library dependencies required by Lwan by typing $ ldd <binary>. Following that, we listed the symbols from the lwan executable by $ nm <binary>. Also, when we ran the Lwan web server in the background, we dumped the virtual memory that corresponds to the process ID for Lwan typing $ cat /proc/<pid>/maps. Lastly, we ran $ sudo strace -p <pid> traces system calls and signals.

Second, we documented the key functionalities of Lwan web server and explained the data interaction and listed Lwan's main system-level functions in a table for a better understanding of Lwan. Analyzing the source code, we generated a top-level system architecture using ArchStudio to better understand the software structure.

Third, we imported the software project into SciTools Understand to plot the metrics summary, construct a cluster call butterfly graph, capture a control flow graph. These tools gave us a pictorial and visual depiction of the relationship between the components and the logic flow of Lwan. We identified a specific function that processes user input and analyzed it for vulnerabilities

Lastly, we used binvis.io to plot the entropy of the Lwan executable to check if it was encrypted or compressed. Following that, we reversed engineered a portion of the file format to identify a header structure and name the fields.

Our python program parses the executable file format to print out the headers for the file and prints the results. This script could be used for any ELF executable on the Linux OS compiled using GCC. The script extracts contents and outputs them to describe the contents. We also looked at other software such as Katai Struct and Hexinator to parse the binary but decided to write our own python script to learned the offsets and structure within an ELF executable used in the Linux OS. On the next page, we have attached a snippet of the python script that we wrote to parse any arbitrary ELF executable file format on Linux compiled using GCC.

Fig. 5. A snippet of our code in elf_parse.py

3 ACCOMPLISHMENTS

We read the manuals provided in Linux kernel for commands such as objdump, ldd, nm, cat/proc/<pid>/maps[9] and strace. The manuals and online references gave us a better understanding of those commands. It was an accomplishment for us to understand the specific purposes of the commands in the context of reverse engineering. For example, an attacker who has no access to the source files but only the binary could execute commands such as dumping the virtual memory to view the addresses of the stack, heap and text section of the code. An attacker could also run a strace to see the systems calls and signals that the software application interacts with the OS. From these tools, an attacker could better comprehend the software application even without looking at the source code. Our group collaboratively analyzed the source code, looked at Lwan's main system-level functions and created a top-level system architecture diagram. From Understand, we learned the sheer size, complexity, control flow and call-graph structure of Lwan that was built using CMake.

On Github, we git cloned, understood and ran Lwan on a virtual machine. Since Lwan is a big software project, it utilizes CMake to compile the many software modules. We utilized CMake to generate the makefile, which contains a set of directives used by make tool[8].

We tested Lwan on our Ubuntu platform and used Firefox to check the web page created by Lwan. By default, the Lwan web server listens on port 8080 and use 2 threads with maximum 2048 sockets per thread. We tested it on our local machine and it works well, as shown by the screenshot below.

```
xyalbino@xyalbino-virtual-machine:/lwan/build2/src/bin/lwan$ sudo ./lwan
Loading configuration file: lwan.conf.
Could not open configuration file: lwan.conf: No such file or directory (error
number 2).
Using 2 threads, maximum 2048 sockets per thread.
Listening on http://127.0.0.1:8080.
Ready to serve.
```

Fig. 2. Running Lwan on Ubuntu after compiling using GCC

To dive deep into Lwan, we tested a sample application called freegeoip.net which is provided by the Lwan developers on the github. The main function of freegeoip is get an IP address from the users in the web page(the user's own IP address by default) and return the details of the address. This application is a good example to show the basic functions of Lwan since it includes the processing of handling the request and interacting with a light-level database. We also used Understand tools to analyze the relationship between the application freegeoip file and other source files in Lwan. We generated a cluster call butterfly graph to analyze the directory structure and a control flow to review the execution flow of all the functions in freegeoip source file.

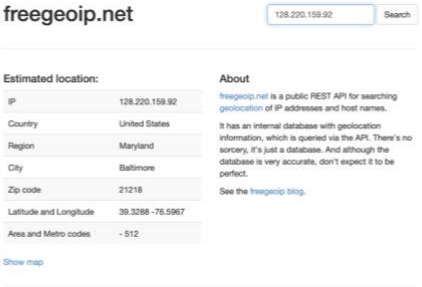

Fig. 3. Visiting FreeGeoIP web page on Firefox

Although Katai Struct was not used to parse the binary file format, we wrote our own python script to take in an arbitrary ELF executable as input and printed out the relevant results as output. We first needed to understand the header structure in order to think about our algorithm to walk through the data in order to parse the relevant information. We read the file in byte by byte into a list. We then checked the contents of the list by index to pull out the information for output. An example output can be viewed in the figure below.

```
ELF Parser:$ python3 elf_parse.py lwan
Filename: lwan
Signature: ELF
Built for a 64 bit system
Built for a big endian system
ELF Version 1
Operating system derivative of System V
Object File Type: ET_DYN
Architecture: x86_64
Process memory entry point: \x9084000000000000
Program headers start at \x40 byte offset
Process memory entry point: \x9072030000000000
ELF header size: 64 Bytes
Program Header Table Size : 56 Bytes
Program Header Entries : 116
Section Header Size : 64 Bytes
Section Entries : 31
Index of Section Header Table : 30
```

Fig. 4. Running elf_parse.py to parse the headers

4 LESSONS LEARNED

Initially, we were confused about the commands when we first used tools like CMake and Make. In our opinion, we thought that we have to use command like GCC to generate binary files. After searching for some documents, we found that CMake and Make had already done the job for us. CMake recognizes which compilers to use for a given kind of source(which is GCC in our program) and Make automatically builds and installs the packages for us[3].

To use Scitools Understand to analyze our source codes, we have to get the udb file for the Lwan. At first, we followed the official instruction for Understand and tried to use buildspy to get the project since "Buildspy is a tool that allows gcc/g++ users to create an Understand project during a build"[4]. We modified the default compiler in our CMakeFile from "gcc" to "gccwrapper" and used the CMake command to get the makefile[5]. Then we use commands buildspy and Make to construct our binary file and the udb database file. However, when we tried to open this udb file in the Understand, the Understand tool could only read the files under the root directory of our program. The reason might be that we missed some options when we used the buildspy command with the command Make. We didn't figure this out, so we just used Understand tool to make the udb file directly and fortunately, it worked for us. We generated some graphs by Understand and it amazed us as it showed the structure of the source files correctly and clearly.

In the process of analyzing the sample application freegeoip, we found some vulnerabilities in the part of how it interacts with the database. Freegeoip takes the user's input and uses it to compose the sql command directly. There is no sanitization for the input and it might be easy for attackers to exploit this vulnerability. The developer may have only considered about the functional implementation without security. Leandro A.F Pereira, the primary developer for Lwan, admitted that security was not a main consideration during development. A malicious user could execute a typical format string attack. This kind of vulnerability is very common in many kinds of web applications, and the discoverers may identify such vulnerability by tracking the data flow from user input to database, or just use static analysis to check the user input.

We tried to use Katai Struct to parse the binary. However, the process of learning how to use Katai Struct from the ground up took us too much time. As such, we decided to write a python script to parse the binary executable. Lastly, some of the output from the commands were complex and difficult to relate to. For example, we could not fully understand some of the headers, symbols and systems calls that were made by Lwan. However, we were able to confirm address space layout randomization when we dumped the virtual memory of the running server. The stack and the heap were in random sections of the virtual memory address space.

One of the major challenges that we had when developing our script was how python reads in bytes as byte strings. If the hex was a valid ASCII character it would turn it into the alphanumeric symbol. While we initially avoided this it became a issue which the header information we wanted to extract was in this form. To negate this we created an if statement that took this malformed bytes and converted them to the form that worked for the rest of our program.

From this project, we learned how challenging developing a huge software project like Lwan can be. We had to read the source code across many different source files to understand the multi-threaded low-level code in C. In addition, some portions of the official documentation for Lwan were technical. Another key takeaway is that although we did not fully reverse engineer the binary back to the original source code, we performed important steps to reverse engineer an open source web server. This gave us an invaluable exposure to how an attacker might start his or her reverse engineering activities with open source applications. As a result, we came to a conclusion that open source projects have a higher likelihood of being reverse-engineered due to the easy access of source code via Github or BitBucket, as examples.

5 REFERENCES

[1] L. Pereira, "Experimental, scalable, high performance HTTP server", https://github.com/lpereira/lwan. 2018.

[2] L. Pereira, "Lwan Project," https://lwan.ws. 2018.

[3] P. Joshi, "CMake vs Make," Perpetual Enigma, https://prateekvjoshi.com/2014/02/01/cmake-vs-make, Feb, 2014.

[4] Scientific Toolworks Inc., "Scitools Understand User Guide and Reference Manual," available at: https://scitools.com/documents/manuals/pdf/understand.pdf. Feb, 2018.

[5] C. Yarns, "How to Set C or C++ Compiler for Cmake," https://codeyarns.com/2013/12/24/how-to-set-c-or-c-compiler-for-cmake. Dec, 2013.

[6] Software Architecture and Implementation Lab, "Tutorial of ArchStudio," available at: https://info.umkc.edu/sail/wp-content/uploads/2016/02/TutorialAS.pdf. Feb, 2016.

[7] "Vectored I/O," Wikipedia, available at: https://en.wikipedia.org/wiki/Vectored_I/O

[8] IEEE Std., "Makefile Syntax," The Open Group Base Specifications, available at: http://pubs.opengroup.org/onlinepubs/9699919799/utilities/make.html#tag_20_76_13_04. 2017.

[9] "Understanding Linux /proc/id/maps", Stack Overflow, https://stackoverflow.com/questions/1401359/understanding-linux-proc-id-maps. Apr, 2009.

[10] Kaitai Project, "Kaitai Struct: A new way to develop parsers for binary structures," https://kaitai.io. 2015.

[11] "ELF Header," The Santa Cruz Operation, http://www.sco.com/developers/gabi/2000-07-17/ch4.eheader.html. July, 2000.