Final Report for Internship

Clifford (Kyle) Clapp

June 5th – July 28th 2017

Attached you will find an assortment of scripts and examples that I have developed or used during my Internship. All the scripts that were used and accompanying tools will be explained below. I had used an assortment of tools from the “binutils” suite, mostly “objdump” and “readelf”.

For the LIEF library, it only seems to be released for 64 bit machines. All the release packages are for 64 bit. They also released the source code and I tried to compile it source. This was unsuccessful as python was never able to find the LIEF package. I had gotten around this by moving the 32 bit binaries to a 64-bit machine with LIEF installed on it. I then made the modifications to the binary and then moved the binary back to the 32-bit machine where I would try running it. All the python scripts and commands that you will see were run on a 64-bit machine.

Binary Enumeration Script:

This works on both 64 and 32 bit binaries. The LIEF library takes care of the differences between the architectures.

The first script that I had developed is my “Binary Enumeration Script”. This script requires python3. The purpose of the program is to enumerate data from the binary path provided. To specify the binary that you would like to use, you will need to change the directory path which is the second line of the program and give an absolute path.

This will the parse through the binary provided and enumerate the data. You have the ability to choose what the program shows. The program is broken up into functions that each do a specific job. The program as it stands as some functions commented out. You can comment and uncomment the function calls at the bottom of the program to show or hide different parts of the program when the script runs.

Some of the functions require arguments, for example, the location of a function that you would like to see. These arguments can be provided at the function call. All of the functions and accompanying arguments are explained in more detail using comments within the script.

Patcher Script:

The second script that I had created was the “Patcher Script”. I had used to patch sections of the binary with modified code. An example of what I used this script for was when I need to modify function calls. When a function is called, the program jumps to another location in binary. I would use this script, entering the hex data that I would like to patch over the binary with as well as the location that I would like to patch. Given these values, the script would patch the binary and write the modified binary to a filename of the users choosing.

Inside of the script are very detailed comments that explain what each portion of the code does.

Section Insertion Script:

This script works on both 64 and 32 bit binaries. The only difference is the virtual address of the section that is to be added.

The third script I had made was designed to create and insert sections in to the binary. This script creates a binary section “object” and assigns attributes such as size and offset. The program then writes the section to the binary and then writes the changes. The reason I made this script was so that I could introduce new functions to the binary to gain info on how the binary runs or how the program flow goes. I use this script and the “Patcher Script” described above together. I would add a new function to the binary with the Section insertion script and then change the program flow by patching the jumps to point to the new function I had added. I have used this technique to look at the arguments on the stack of a certain function and to look at the size of the arguments passed on the stack.

Iterating Function Arguments Script:

The 64-bit version of this program is the only stable one as the 32-bit version has a problem of detecting duplicates when looking at the hex codes. The 32-bit function can return false data on occasion.

This script is designed to look at all the binaries and look at the function definitions. This scripts by using the tool “objdump” to put the dissembled content of the binary to the standard output and then piping the output to the script. The script takes this input and looks at the data line by line to find function arguments. To use this script, the data must be in “Intel Syntax”, this can be done by the following command.

objdump –D –M intel <binary>| python3 Iterating Function Arguments.py

The –D flag dissembles everything in the binary

The –M flag tells how the assembly output will look, by default it is AT&T syntax

<binary> This is where the name of the binary that you would like to investigate goes.

I have provided some samples that I used to test with that have different data types such as short and double long

Process of Investigation:

The first thing that I had done was look at different tools other LIEF to see which tool would be the best to investigate the binaries with. I had looked at several tools, most of which just performed a specific task. When looking at all the other tools and programs that were built, it was clear that LIEF was by far the best tool/program for the job. LIEF is able to do just about anything that you want in relation to working with binaries.

The first python script that I had built was the “Binary Enumeration Script”. This was using LIEF to apply all the functions of LIEF to a binary. The program is able to look at all the symbols/functions that were imported/exported, the dynamic symbols, the GOT (Global Offset Table) and PLT (Procedure Linkage Table) and retrieve the locations of the functions. I had used this program for different purposes throughout my internship

The first thing I had looked at was how the assembly is read in the binary. And performing some tests to see how far I could modify the assembly code without the program failing or segmentation faulting. I had performed several tests by changing taking a locally declared function “max” and changing assembly code within the binary. I had use the Patcher Script above to change the values of the variables. For example change 100 to 200 and see if the program still ran. Another thing I tested was changing the operator in the comparison statement and the binary still ran. I will include a compressed folder with the scripts that I tested with. All the tests ran fine with no segmentation faults.

The first couple tests that I had performed all dealt with arithmetic functions. I had then moved on to string data types. There was quite a distinction with between whether the string was declared with an array or with a pointer. In my tests I had found that a string that was declared in an array you were able to see in the hex code being declared on the stack. A string that was declared by a pointer, is just reference by a pointer in the function stack space. In terms of stack space that is needed for both types of declaration, the array needs as much stack space as is required to fit the entire string. Dissassembling the binary, you are able to see the entire string declared as hex in the stack space. A string that was declared by a pointer only needs 8 bytes of data as the pointer just points to a different location in the program.

Another aspect of the binary that I looked at was the characteristics of jumps. I had first experimented with this by defining and creating 2 arithmetic functions but only calling one in the program. The 2 functions that I had used was the “max” function mentioned earlier that takes 2 integers, compares them and returns the larger one. The second function that I had used was a function called “add”. This function took 2 integers, added them and returned the sum. In the compiled binary, I had called the “max” function but not the add function. Looking at the assembly I had found that the “add” function still existed as the compiler did not remove it. I had used the patcher script to edit the call of the “max” function so it now pointed to the “add” function. Through these experiments I learned that the jump operations use 2’s complement to jump to address locations less than the jump instruction. This also proved that I was able to make basic changes to the program flow without making the binary segmentation fault at runtime.

Results of Preliminary Investigation:

The conclusion of the primary investigation shows me that the binary is able to be modified given that the code being patched in is no longer than the assembly code that already exists at that location. For example, you can patch 4 hex bytes over an existing 4 hex bytes. If you were to try and patch the same location with 5 hex bytes, it would insert the 5 hex bytes but write over the instruction that started at the 5th byte location. I found at this point it was easier to insert a new section into the binary and just intercept the jumps to point to the newly added data if you needed to write more data than what was available.

Looking at the Stack:

Once it was figured out that the binary assembly could be modified without segmentation faults. I had proceeded to look at the stack of the functions. As the end goal was to interact with a specific function inside of the binary, it was important to look at what arguments and/or data that was passed to the function on the stack as well as the calling convention that the function used.

I was able to look at the stack by inserting a new section of assembly code that would look at all the variable being passed on the stack. Once the new section of data was added that iterated through the values that were passed on the stack, I intercepted the jumps in the function call that then moved the program control to the new section that I added.

The way I looked at the parameters on the stack were very primitive. I did not use a loop but just change the value of the base pointer to the value that I knew existed a parameter, I pushed the value to the EBX register and then called the print function to print the value that I changed the base pointer to.

This script and program is included in a compressed folder called “Looking at the Stack”

I had tried to make a loop looking at the variables that were passed on the stack in assembly code but I was not able to get a working implementation working. I was not able to get the logic to have the binary iterate through the stack and print the value/size of the data. This is where I had developed the “Iterating Function Arguments Script”. This was a python implementation of what I had tried to do in assembly. I had used Regular Expressions to look for the size of the arguments being passed to the stack.

Results of looking at the Stack:

During this investigation I had learned a lot of how binaries are assembled. Because I had to add code to the binary, I had to learn the roles of sections and segments and their relation to each other. It was necessary for me to learn what the roles were of each section. When adding the code that would look at the variables on the stack I learned that the flag for the LIEF function “add\_section” must be set to true because this would make sure that the added section gets added to the first LOAD segment which is loaded at run time. This allows my code to be run by the binary. I also found that when a section was added for to the binary, it was added at the very beginning of the binary. As big as the section was, LIEF seemed to automatically patch the rest of the sections because the binary never segmentation faulted.

Linking against the function:

Once the value/size of the arguments being passed on the stack were determined. I had tried to link against the executable function, in this case “Max”, with another C test program. This other C program in theory could be a fuzzer that could test what breaks and does not break the function. I had then developed a python program that would be able to change header type value from “executable” to any other type of binary. For linking the tests I had changed the header type to either “relocatable” or “dynamic”. This allowed me to experiment by attempting to link against it using several methods.

I had first tried to link dynamically against the program, first I changed the header type to “dynamic” and changed the entrypoint to the value of the function “max” that I wanted to link against. The executable was then classed as a shared object. I had attempted to link against this executable turned shared object directly as well as using gcc to put this executable into a shared library and then linking against it. Both of these options had returned a variety of errors. The best case that I had gotten was that the executable was able to compile but the program segmentation faulted. I had also left the entrypoint of the program the same without modifying and that did not fix any of the errors.

I had then tried to link the program statically. I had changed the file type header to “relocatable” and using the archiver tool “ar” I was able to put the executable into an archive. I had tried linking the test program to this archive but it returned the same result as the dynamic linking test. It had produced several errors with the best case that it compiled with warnings and the program segmentation faulted. I had also tried linking with the entrypoint of the binary the non modified entrypoint value and that did not help

I had attempted to link against the executable function several ways that were not successfully. I had found out after several different attempts that in order for a C program to link against a shared object or an archive of object files, the (shared) object files need to be compiled with PIC (Position Independent Code). PIC allows the object to be loaded into any memory location. This is necessary for linking objects as there could be multiple program instances trying to access a single shared object at different locations within the shared object. What PIC does is that all the jump instructions are created as relative within the shared object. This allows the shared object to be loaded into any memory address and all jump instructions will still work regardless of memory location. Programs without PIC will have absolute jumps that need to be loaded at the same address every time that they are called. This created a problem when I linked against non PIC code as I was able to successfully link and able to run the binary, but with absolute addressing, this caused the program to segmentation fault.

Another error that found when trying to link the programs with the original executable that contained the function that was needed, the executable needed to be a PIE (Position Independent Executable). This is because the Executable Binary was then turned into a shared object.

Without PIC/PIE in the original executable there were several errors that the linker found that had confused me at first. When creating the original function, I did not export any of the static symbols to the dynamic symbol table. When I had tried to link the test binary with the original binary, it said there was duplicate definitions of several functions such as “\_\_data\_start”, “\_fini” , “main”, and “\_\_dso\_handle”. I think what is happening is that because all these function symbols are still stored statically in the original binary, when the test binary is linked against it, the test binary tries to import all the static symbols which conflict with its own. If these symbols were exported to the dynamic symbol table when the original binary was compiled, the symbols would not interfere with the test binary and would not cause any errors.

An example of the command that I would run to get this error:

gcc -o linked\_test link\_test.c -L. –ltest

**linked\_test:** The output from the linking process

**link\_test.c:** The test binary that is linked with the function from the original executable turned shared object

**-L.:** Telling the linker where to find the library (In this case it was the same directory)

**-ltest:** This is the library (the original executable) that the test binary is trying to link against

**The “test” shared object has not been compiled with PIC or had its symbols exported during compilation**

**kyle@kyle-VirtualBox:~/Desktop/64\_bit\_test$ gcc -o linked\_test link\_test.c -L. -ltest**

**./libtest.so: In function `\_fini':**

**(.fini+0x0): multiple definition of `\_fini'**

**/usr/lib/gcc/x86\_64-linux-gnu/5/../../../x86\_64-linux-gnu/crti.o:(.fini+0x0): first defined here**

**./libtest.so: In function `data\_start':**

**(.data+0x0): multiple definition of `\_\_data\_start'**

**/usr/lib/gcc/x86\_64-linux-gnu/5/../../../x86\_64-linux-gnu/crt1.o:(.data+0x0): first defined here**

**./libtest.so: In function `data\_start':**

**(.data+0x8): multiple definition of `\_\_dso\_handle'**

**/usr/lib/gcc/x86\_64-linux-gnu/5/crtbegin.o:(.data+0x0): first defined here**

**./libtest.so:(.rodata+0x0): multiple definition of `\_IO\_stdin\_used'**

**/usr/lib/gcc/x86\_64-linux-gnu/5/../../../x86\_64-linux-gnu/crt1.o:(.rodata.cst4+0x0): first defined here**

**./libtest.so: In function `\_start':**

**(.text+0x0): multiple definition of `\_start'**

**/usr/lib/gcc/x86\_64-linux-gnu/5/../../../x86\_64-linux-gnu/crt1.o:(.text+0x0): first defined here**

**./libtest.so: In function `main':**

**(.text+0xf6): multiple definition of `main'**

**/tmp/ccQCYSQn.o:link\_test.c:(.text+0x0): first defined here**

**./libtest.so: In function `\_init':**

**(.init+0x0): multiple definition of `\_init'**

**/usr/lib/gcc/x86\_64-linux-gnu/5/../../../x86\_64-linux-gnu/crti.o:(.init+0x0): first defined here**

**/usr/lib/gcc/x86\_64-linux-gnu/5/crtend.o:(.tm\_clone\_table+0x0): multiple definition of `\_\_TMC\_END\_\_'**

**./libtest.so:(.data+0x10): first defined here**

**/usr/bin/ld: error in ./libtest.so(.eh\_frame); no .eh\_frame\_hdr table will be created.**

When I tried to link the program statically with the original executable as a relocatable file, I also received this error of multiple definitions of the same symbol.

As for error:

/usr/bin/ld: error in ./libtest.so(.eh\_frame); no .eh\_frame\_hdr table will be created.

I have not found an exact cause to this error. I have researched what the “.eh\_frame\_hdr” section is and it relates to unwinding the stack in case of an error. The related section in the original executable (shared object libtest.so in this case) “.eh\_frame” contains an error and this is where I am uncertain. I am not sure how the “eh\_frame” section is supposed to look like.

Attached will be a text file containing commentary on what I felt was wrong or causing the error. This file contains some of the error files that I had ran into during the compilation process.

Fixes for the errors above:

When getting the multiple symbol error, I had then manually removed all the conflicting symbols from the shared library or archive made from the original executable. This was done with the following commands.

strip –strip-symbol=<symbol name> <shared library or archive that is conflicting>

Another solution I found was to compile the object files and link them separately. While linking I had use the “ld” command flag <-zmuldefs>. This supressed the multiple definition warning, although because of this I am not sure which symbol the program had used, I was not able find which symbol took precedence through the documentation.

For the “no .eh\_frame\_hdr table will be created” error, I was never able to get rid of this warning. I feel that this was because I was trying to link against non-PIC compiled binary. Every time I compiled a binary and this warning came up the binary segmentation faulted.

The code example that did work:

This example is a custom made executable and shared object. Showing how PIC makes it much easier to link against an executable.

gcc -fPIC -pie -o libtest.so test.c -Wl,-E

In order to link against an executable, this is how it must be compiled. As you can see that “libtest.so” is compiled with both the “-pie” and “-fpic” flags which indicate Position Independent Code. The “–WL,-E” switch tells the compiler to export all the symbols in the dynamic symbols table. The result of this command is a single file called “libtest.so”. It is labelled as a shared object but does not have to because you can run “libtest.so” as an executable assuming the code inside of the C file is designed to be run.

Gcc –o linked\_test program.c -L. -ltest

As the file created above was called “libtest.so”, the compiler recognizes this as a library, as it was compiled with PIC, it can be linked against. The file called “program.c” is a file that will need a function defined by the file “test.c” in the first command line. This will produce a runnable “linked\_test” file that does not segmentation fault.

Mixing PIC with non-PIC:

kyle@kyle-VirtualBox:~/Desktop/64\_bit\_test$ ld -shared -fPIC -o linked\_test link\_test.o modded -zmuldefs

ld: error in modded(.eh\_frame); no .eh\_frame\_hdr table will be created.

ld: linked\_test: No symbol version section for versioned symbol `\_\_libc\_start\_main@@GLIBC\_2.2.5'

ld: final link failed: Nonrepresentable section on output

kyle@kyle-VirtualBox:~/Desktop/64\_bit\_test$

**linked\_test:** This is the name of the final binary.

**Link\_test.o:** This is the object file of the C program that I need the symbol/function to complete the linking process.

**modded:** This is the original compiled executable that we are trying to get the function from

This is with the header value of the original compiled executable staying the same and with the header value being changed to the location of the max function. I had noticed that once I changed the header value of the original binary, the hex location and name of the symbol changed once I modified the entry point. I then had to run python script to modify the numbers back to their original value and name.

I had then changed the entry point to the new value and I still got the above errors. From my experiments I have found that changing the header values do not make a difference. At the time of writing this I have not found a definitive cause of errors or how to fix them.

kyle@kyle-VirtualBox:~/Desktop/64\_bit\_test$ gcc -shared -fPIC modded link\_test.o -zmuldefs

/usr/bin/ld: warning: Cannot create .eh\_frame\_hdr section, --eh-frame-hdr ignored.

/usr/bin/ld: error in modded(.eh\_frame); no .eh\_frame\_hdr table will be created.

Another example that was able to compile was this command. The compiler did not error out and was able to make an output file. This program segmentation faulted immediately but the interesting part is that the header of file broke. The output file icon was a text file as Ubuntu did not recognize it as an executable. When trying to look at the file through objdump I had received this error.

kyle@kyle-VirtualBox:~/Desktop/64\_bit\_test$ objdump -D a.out

a.out: file format elf64-x86-64

objdump: BFD (GNU Binutils for Ubuntu) 2.26.1 internal error, aborting at ../../bfd/elf64-x86-64.c:6089 in elf\_x86\_64\_get\_plt\_sym\_val

objdump: Please report this bug.

Because of this I was not able to investigate what went wrong in the code or where the program segmentation faulted.

My initial look into building my own “\_\_libc\_main\_start” was that it was slightly more complicated than it looked. The program control starts at the “\_main” which sets up some preliminary requirements. It then calls the “\_\_libc\_main\_start” which is called through the PLT table. Because this function is from the GNU C library it is linked at runtime and I will have to look into finding out exactly what it does. Once past the “\_\_libc\_main\_start” function, I am not aware of how the “main” function is called.

When looking at the control flow of an ELF binary, I had noticed there was a function called “start” at the very beginning of the binary. This “start” function can be located with the tool “objdump”. Inside of the “start” function is a call to the “\_\_libc\_start\_main@plt” function which setups the stack and readies the kernel for the “main” function. As the “\_\_libc\_start\_main@plt” function is from the dynamic library, this is why it is called from the PLT table as the linker performs the linking at run time.

From what I have read indicates that the “\_\_libc\_start\_main@plt” function calls the main function. The “main” function executes and when it finishes it goes to the destructor function. The destructor function clears the stack and the registers for the next function.

Before trying to add my own start function to the code I wanted to see how original start function worked. This is the original “\_start” function in the program “64\_bit\_test”. Notice how the main function calls the \_\_libc\_start\_main function.

**0000000000400430 <\_start>:**

**400430: 31 ed xor %ebp,%ebp**

**400432: 49 89 d1 mov %rdx,%r9**

**400435: 5e pop %rsi**

**400436: 48 89 e2 mov %rsp,%rdx**

**400439: 48 83 e4 f0 and $0xfffffffffffffff0,%rsp**

**40043d: 50 push %rax**

**40043e: 54 push %rsp**

**40043f: 49 c7 c0 10 06 40 00 mov $0x400610,%r8**

**400446: 48 c7 c1 a0 05 40 00 mov $0x4005a0,%rcx**

**40044d: 48 c7 c7 26 05 40 00 mov $0x400526,%rdi**

**400454: e8 b7 ff ff ff callq 400410 <\_\_libc\_start\_main@plt>**

**400459: f4 hlt**

**40045a: 66 0f 1f 44 00 00 nopw 0x0(%rax,%rax,1)**

I had then modified this system call to jump to the main function.

**400454: e8 cd 00 00 00 callq 400526 <main>**

I had ran the program and the main function and the max function worked perfectly even though the stack and parameters were not set up for the function call. The program did not even segmentation fault. I had experimented with both a function call (as seen above in AT&T syntax) as well as just a jump to the “main” function and both instances ran fine.

I had found this very interesting as the code was able to run without the stack being properly setup as well as none of the destructors ran.

Another technique that I had tried was compiling a program using Position Independent Code. Inside this program was just a blank main function.

**#include <stdio.h>**

**int main () {}**

This was all that I had compiled. I had then taken the “max” function and added it to the end of the file above. I had then changed the name and location of the symbol “main” from pointing at “main” to point at “max”.

When originally compiling the program, I had explicitly told the gcc to export all the symbols and use Position Independent Code in hopes that it would export the “max” function when linking the file to another executable.

I had compiled the original Position Independent Code file as follows:

**gcc -fPIC -o blank\_main -rdynamic -ldl -Wl,-E blank\_main.c**

The flags in the command are as follows:

-**fPIC:** Forcing the resulting compilation to have Position Independent Code

**-o blank\_main:** The name of the output file

**-rdynamic:** This tells the linker to add all sybols, not just the ones that are used to the dynamic symbols table

**-ldl:** This enables the usage of commands in the program such as “dlopen()”

**-Wl,-E:** The tells the linker to export all the symbols in the binary

**blank\_main.c:** This is the C file that contain the blank main code

I had also to add the original function as a section to an executable that was compiled with the flags –fPIC and –pie, the Boolean flag in the LIEF function “add\_section” has to be set to “False”. With the Boolean value set to “True”, it will generate the following error. The difference between the 2 flags is that it will give the code that is added a different virtual address.

**Traceback (most recent call last):**

**File "insert\_end\_function.py", line 52, in <module>**

**binary.write("modded")**

**lief.conversion\_error: Invalid virtual address**

After adding the section to the binary with the Boolean flag set to “False”, I had patched the main function to jump to the newly added code. I was able to patch the code but then the program segmentation faulted. I think this is because the code was added with the flag set to “False” as this sets the added code to non-executable. When the program tries to run the code, it segmentation faults.

I had then tried the same tactic as before but I only compiled the binary with the –fPIC flag. This allowed me to add the original function to the binary as a section with the Boolean flag on the “add\_section” as “True”. I had then tried to link the files together but it encountered an error stating multiple definitions of the same symbol were found. I had fixed with using the “-zmuldefs” flag as mentioned above. When running the program, it segmentation faulted.

In the process of experimenting, I have noticed that the hex offsets of the symbols change when the file header changes. Through research I had found that when the header states the file is an executable, the hex offset does not include the load address. When I had changed the file header to “Relocatable”, the hex offsets had increased by “4003e0” hex values as this is the load value. The hex offsets were taking into account the value of the start address. At the point of writing this I am not sure if this effects how the symbols are treated when linking.

The last thing that I looked at were the sections of the executable to see how little the binary needed to run. To do this I just copied “nop codes” over the section assembly code. The results are as follows.

The binary needs a proper start function otherwise it will error with:

**bash: ./modded: cannot execute binary file: Exec format error**

“.note.ABI-tag” “.note.gnu.build-id”

I was able to completely write over these sections with “nop codes” and the binary ran without segmentation faulting.

“.rodata” “.eh\_frame” “.eh\_frame\_hdr”

When attempting to overwrite these sections I encountered this error:

**Unable to find the symbol associated with the relocation (idx: 3)**

I was able to write over this section of the binary. I was able to put nopcodes (x90) in the first five hex addresses of the hex section.

When running the executable, I still received this error.

**./modded: error while loading shared libraries: ./modded: unsupported version 0 of Verneed record**

.got.plt .dynamic .fini\_array .init\_array .fini .plt.got .plt .init .rela.plt .rela.dyn .gnu.version\_r .gnu.version .dynsym .gnu.hash .dynstr

When overwriting these sections I encountered this error:

**Unable to find the symbol associated with the relocation (idx: 3)**

I was able to write over this section of the binary. I was able to put nopcodes (x90) in the first five hex addresses of the hex section.

When running the executable, I still received this error.

**./modded: error while loading shared libraries: ./modded: unsupported version 0 of Verneed record**

The nopcodes (x90) that I had written to the file were not visible when I used “objdump -D”, this leads me to believe that I was not able to write over the file, but somehow still corrupted the file.

In an attempt to get around this, I had tried the command:

“objcopy --set-section-flags .gnu.hash=load test”

This command sets the flags of the section specified “.gnu.hash” with a comma separated list. In this example I had just tried the flag “load” so that the section will be loaded at run time, effectively changing the permissions. This is the same flag that the “.text” sections have and I have been able to overwrite those sections without any issues.