ENPM691 Homework Assignment 11

Hacking of C programs and Unix Binaries-Fall 2024 lcollier

Dipam Bhattacharyya, MEng Cybersecurity dipamb2@umd.edu

***Abstract*—The assignment aims to understand the workings of dynamic linking and C libraries to manipulate their behavior. Wrapper functions will control the flow to execute the code of our choice.**

***Index Terms*—homework, hacking, C, coding**

I. Introduction

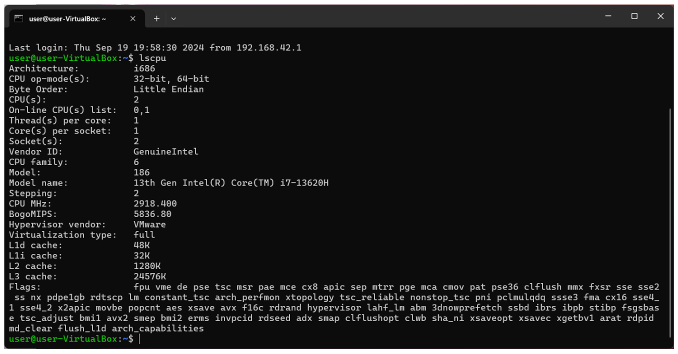
Adversaries may execute their own malicious payloads by hijacking environment variables the dynamic linker uses to load shared libraries. During the execution preparation phase of a program, the dynamic linker loads specified absolute paths of shared libraries from environment variables and files, such as LD_PRELOAD on Linux. Libraries specified in environment variables are loaded first, taking precedence over system libraries with the same function name.

In this experiment, an exploration is done on writing a ”puts” library function wrapper. The dynamic linker will resolve this instead of the actual ”libc” implementation. The wrapper will change the control flow of the program and return values of our choice.

II. Methodology

A. Host environment and architecture

[1] The experiment was done on a Ubuntu v16.04 VM with Windows x64 as the host PC. All exploit protection settings in Windows are default, including ASLR(default ON). To check the VM architecture, the ”lscpu” terminal command was given:



```
Last login: Thu Sep 19 19:58:30 2024 from 192.168.42.1
user@user-VirtualBox:~$ lscpu
Architecture:          i686
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                2
On-line CPU(s) list:   0,1
Thread(s) per core:    1
Core(s) per socket:    1
Socket(s):             2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 186
Model name:            13th Gen Intel(R) Core(TM) i7-13620H
Stepping:              2
CPU MHz:               2918.400
BogoMIPS:              5836.80
Hypervisor vendor:     VMware
Virtualization type:   full
L1d cache:             48K
L1i cache:             32K
L2 cache:              1280K
L3 cache:              24576K
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2
 ss nx pdpe1gb rdtscp lm constant_tsc arch_perfmon xtopology tsc_reliable nonstop_tsc pni pclmulqdq ssse3 fma cx16 sse4_
1 sse4_2 x2apic movbe popcnt aes xsave avx f16c rdrand hypervisor lahf_lm abm 3dnowprefetch ssbd ibrs ibpb stibp fsgsbas
e tsc_adjust bmi1 avx2 smep bmi2 erms invpcid rdseed adx smap clflushopt clwb sha_ni xsaveopt xsavec xgetbv1 arat rdpid
md_clear flush_l1d arch_capabilities
user@user-VirtualBox:~$ 
```

Fig. 1. Ubuntu CPU information

B. Dynamic linking and hijack

Generally, there are two ways of producing an executable. The first method is the static compilation. During the compilation process the executable that will be produced, will be independent and will include all it needs to function properly. The other method is the dynamically produced executable. It is dependent on shared libraries to function, and correspond to changes in the shared libraries. The gcc compiler will by default produce a dynamically linked executable, unless we specify the ’-static’ parameter to gcc.

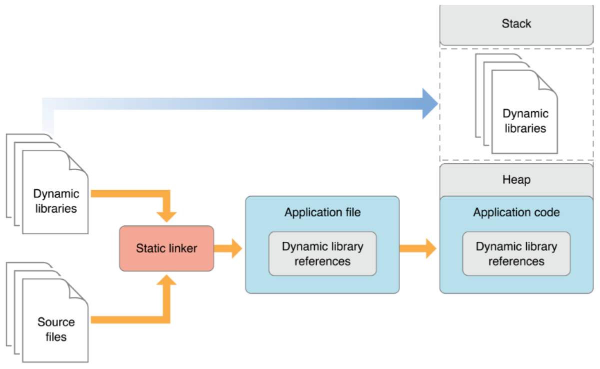


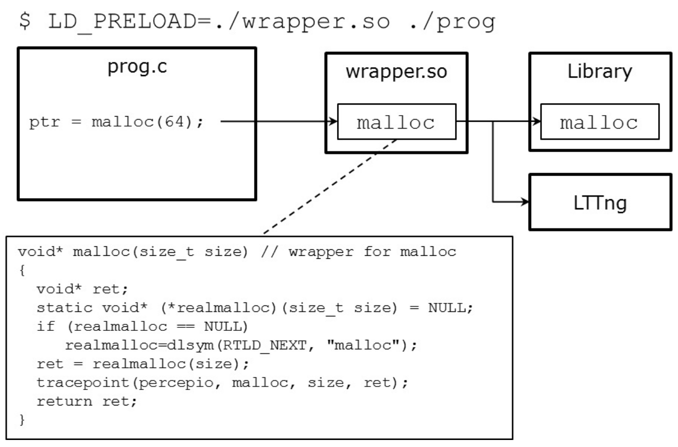
Fig. 2. Working of dynamic linking

Hijacking dynamic linker variables may grant access to the victim process’s memory, system/network resources, and possibly elevated privileges. This method may also evade detection from security products since the execution is masked under a legitimate process. Adversaries can set environment variables via the command line using the export command, setenv function, or putenv function

C. LD_PRELOAD environment variable

The runtime linker’s purpose is to bind the dynamically linked executables to their dependencies by all means. LD_PRELOAD is an environment variable that affects the runtime linker. It allows us to put a dynamic object, that will create some sort of a buffer layer, between the application references and the dependencies. It also grants you the possibility of linking with the application and relocating symbols/references.

On Linux, adversaries may set LD_PRELOAD to point to malicious libraries that match the name of legitimate libraries that are requested by a victim program, causing the operating system to load the adversary’s malicious code upon execution of the victim program. LD_PRELOAD can be set via the environment variable or /etc/ld.so.preload file. Libraries specified by LD_PRELOAD are loaded and mapped into memory by dlopen() and mmap() respectively.



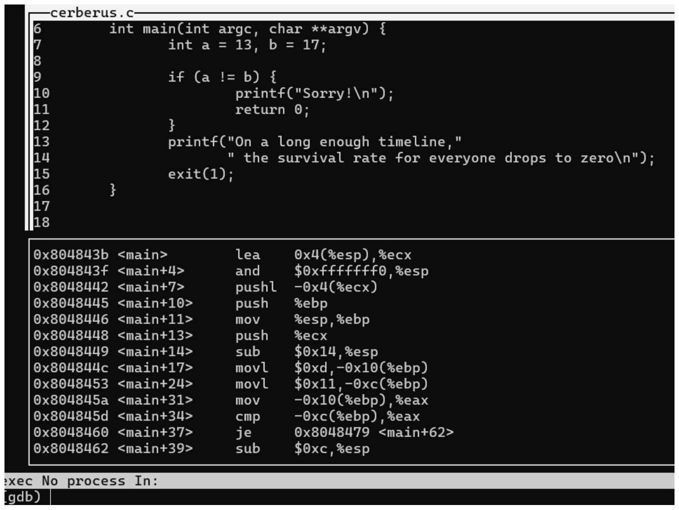
```
void* malloc(size_t size) // wrapper for malloc
{
  void* ret;
  static void* (*realmalloc)(size_t size) = NULL;
  if (realmalloc == NULL)
     realmalloc=dlsym(RTLD_NEXT, "malloc");
  ret = realmalloc(size);
  tracepoint(percepio, malloc, size, ret);
  return ret;
}
```

Fig. 3. LD_PRELOAD loading malloc library wrapper

D. GDB debugging

[2] After the generation of the executable, we are attaching gdb to it. Then, we are looking at both source and disassembled code together while also having terminal access.

$$(gdb)\ layout\ split \tag{1}$$



```
cerberus.c
        int main(int argc, char **argv) {
                int a = 13, b = 17;

                if (a != b) {
                        printf("Sorry!\n");
                        return 0;
                }
                printf("On a long enough timeline,"
                       " the survival rate for everyone drops to zero\n");
                exit(1);
        }
```

```
0x804843b <main>        lea    0x4(%esp),%ecx
0x804843f <main+4>      and    $0xfffffff0,%esp
0x8048442 <main+7>      pushl  -0x4(%ecx)
0x8048445 <main+10>     push   %ebp
0x8048446 <main+11>     mov    %esp,%ebp
0x8048448 <main+13>     push   %ecx
0x8048449 <main+14>     sub    $0x14,%esp
0x804844c <main+17>     movl   $0xd,-0x10(%ebp)
0x8048453 <main+24>     movl   $0x11,-0xc(%ebp)
0x804845a <main+31>     mov    -0x10(%ebp),%eax
0x804845d <main+34>     cmp    -0xc(%ebp),%eax
0x8048460 <main+37>     je     0x8048479 <main+62>
0x8048462 <main+39>     sub    $0xc,%esp

exec No process In:
(gdb)
```

Fig. 4. gdb split layout output

To check the stack frames(in reverse order of being invoked) used in the program:

$$(gdb)\ bt \tag{2}$$

And to inspect a particular frame:

$$(gdb)\ info\ frame\ 1 \tag{3}$$

E. Code Analysis and Verification

For the experiment, the "cerberus.c" code (See Appendix A) is used. The main declares two variables, a and b, and assigns them different values. After the comparison between the two values, it prints the "Sorry" message and exits. The code is designed in such a way that the second print message will never be displayed in a normal execution flow.

Another C code, the "megatron.c" (See Appendix B) is used to write the "puts" wrapper function. The main motive for writing this is to redirect the control to the second print statement without executing the first. It manipulates the value of the ESP register and also clears the valid stack created.

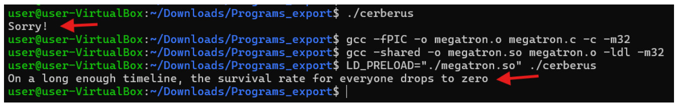
Through a series of commands (See Appendix C), the second code will be used to create a library file that will be loaded during the runtime of the executable produced using the first code. On running, the puts function defined by "megatron.c" is used.

III. Results

Figure5 shows the output of the "cerberus.c", normal execution as well as using the hijacked library.

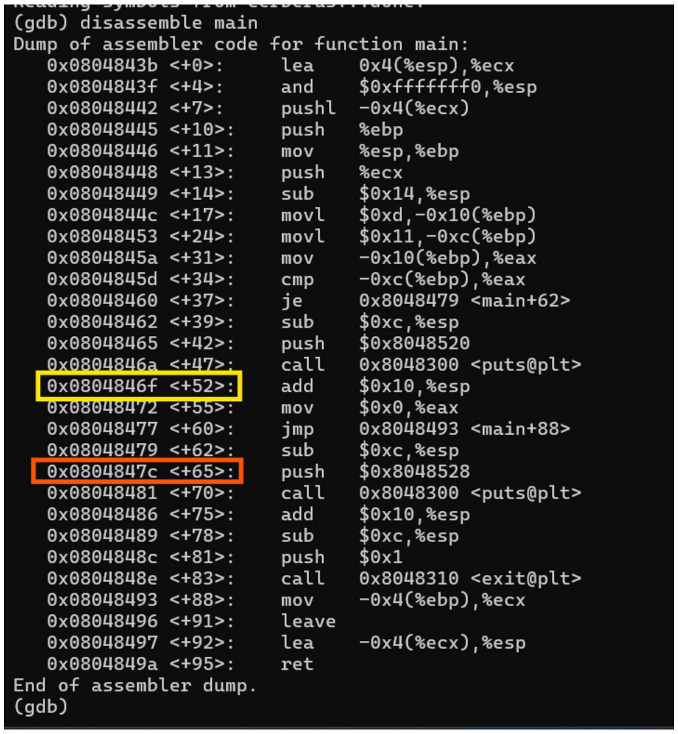
Figure6 indicates the addresses in the disassembly of "cerberus.c", yellow being the one that the compiler would return to. Orange is the address we want the compiler to jump to after hijack.

Figure7 displays the disassembly of "puts" in "megatron.c". The red box shows the stack allocation for that function.



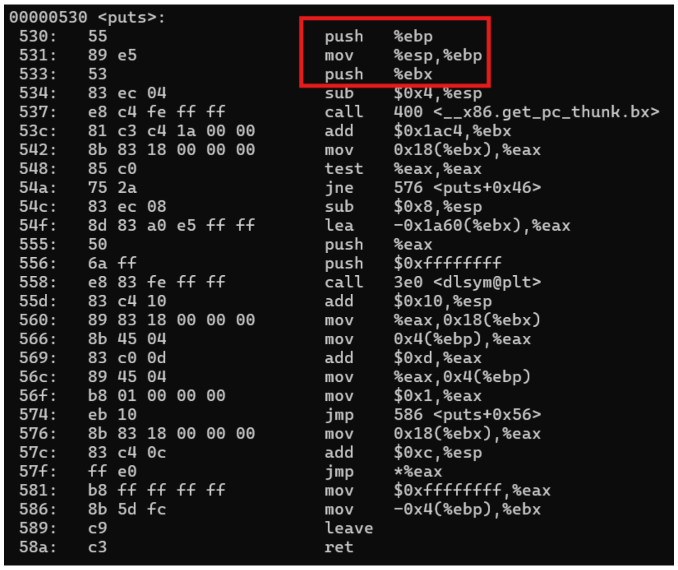
```
user@user-VirtualBox:~/Downloads/Programs_export$ ./cerberus
Sorry!
user@user-VirtualBox:~/Downloads/Programs_export$ gcc -fPIC -o megatron.o megatron.c -c -m32
user@user-VirtualBox:~/Downloads/Programs_export$ gcc -shared -o megatron.so megatron.o -ldl -m32
user@user-VirtualBox:~/Downloads/Programs_export$ LD_PRELOAD="./megatron.so" ./cerberus
On a long enough timeline, the survival rate for everyone drops to zero
user@user-VirtualBox:~/Downloads/Programs_export$
```

Fig. 5. Hijacked code using LD_PRELOAD



```
(gdb) disassemble main
Dump of assembler code for function main:
   0x0804843b <+0>:     lea    0x4(%esp),%ecx
   0x0804843f <+4>:     and    $0xfffffff0,%esp
   0x08048442 <+7>:     pushl  -0x4(%ecx)
   0x08048445 <+10>:    push   %ebp
   0x08048446 <+11>:    mov    %esp,%ebp
   0x08048448 <+13>:    push   %ecx
   0x08048449 <+14>:    sub    $0x14,%esp
   0x0804844c <+17>:    movl   $0xd,-0x10(%ebp)
   0x08048453 <+24>:    movl   $0x11,-0xc(%ebp)
   0x0804845a <+31>:    mov    -0x10(%ebp),%eax
   0x0804845d <+34>:    cmp    -0xc(%ebp),%eax
   0x08048460 <+37>:    je     0x8048479 <main+62>
   0x08048462 <+39>:    sub    $0xc,%esp
   0x08048465 <+42>:    push   $0x8048520
   0x0804846a <+47>:    call   0x8048300 <puts@plt>
   0x0804846f <+52>:    add    $0x10,%esp
   0x08048472 <+55>:    mov    $0x0,%eax
   0x08048477 <+60>:    jmp    0x8048493 <main+88>
   0x08048479 <+62>:    sub    $0xc,%esp
   0x0804847c <+65>:    push   $0x8048528
   0x08048481 <+70>:    call   0x8048300 <puts@plt>
   0x08048486 <+75>:    add    $0x10,%esp
   0x08048489 <+78>:    sub    $0xc,%esp
   0x0804848c <+81>:    push   $0x1
   0x0804848e <+83>:    call   0x8048310 <exit@plt>
   0x08048493 <+88>:    mov    -0x4(%ebp),%ecx
   0x08048496 <+91>:    leave
   0x08048497 <+92>:    lea    -0x4(%ecx),%esp
   0x0804849a <+95>:    ret
End of assembler dump.
(gdb)
```

Fig. 6. Disassmebly of cerberus.c main()



```
00000530 <puts>:
 530:   55                      push   %ebp
 531:   89 e5                   mov    %esp,%ebp
 533:   53                      push   %ebx
 534:   83 ec 04                sub    $0x4,%esp
 537:   e8 c4 fe ff ff          call   400 <__x86.get_pc_thunk.bx>
 53c:   81 c3 c4 1a 00 00       add    $0x1ac4,%ebx
 542:   8b 83 18 00 00 00       mov    0x18(%ebx),%eax
 548:   85 c0                   test   %eax,%eax
 54a:   75 2a                   jne    576 <puts+0x46>
 54c:   83 ec 08                sub    $0x8,%esp
 54f:   8d 83 a0 e5 ff ff       lea    -0x1a60(%ebx),%eax
 555:   50                      push   %eax
 556:   6a ff                   push   $0xffffffff
 558:   e8 83 fe ff ff          call   3e0 <dlsym@plt>
 55d:   83 c4 10                add    $0x10,%esp
 560:   89 83 18 00 00 00       mov    %eax,0x18(%ebx)
 566:   8b 45 04                mov    0x4(%ebp),%eax
 569:   83 c0 0d                add    $0xd,%eax
 56c:   89 45 04                mov    %eax,0x4(%ebp)
 56f:   b8 01 00 00 00          mov    $0x1,%eax
 574:   eb 10                   jmp    586 <puts+0x56>
 576:   8b 83 18 00 00 00       mov    0x18(%ebx),%eax
 57c:   83 c4 0c                add    $0xc,%esp
 57f:   ff e0                   jmp    *%eax
 581:   b8 ff ff ff ff          mov    $0xffffffff,%eax
 586:   8b 5d fc                mov    -0x4(%ebp),%ebx
 589:   c9                      leave
 58a:   c3                      ret
```

Fig. 7. Disassmebly of megatron.c puts()

IV. Discussions

A. Changing control flow

The ”if” condition in the ”cerberus.c” code will always be true and the print function will be called. Since there are no arguments, the compiler calls the ”puts” function. The wrapper ”puts” in ”megatron.c” first initializes the pointer to NULL. This will be done only for the first call to the ”puts”. Then dlsym(RTLD_NEXT, ”puts”) loads the next symbol in the table, essentially calling the actual library function. The ”__asm__ __volatile__” forces the compiler to execute the code as-is without optimization.

When the ”puts” is called, a stack is allocated where the output string will be stored. Along with that, the return address of the next line of code will be stored at an offset of 4 bytes from EBP, at a higher address. Here, it will be ”0x0804846f”, the next line after the first ”puts” call (in yellow), as in Fig.6. As we do not want to print this string, we add ”13” to the return address and restore the value on the stack. The addition of ”13” will result in ”0x0804847c” which is our intended print string, in the orange box.

With this code, we successfully transferred execution control to the second ”puts”.

B. Stack reset and ESP

Once the second ”puts” is called, the function stack will have the new string, the instruction in the orange box, fig.6. This time the wrapper function will not have ”puts” as NULL and the second ”__asm__ __volatile__” section will execute.

As in the code (Appendix B), we add ”12” to the stack pointer so that it overwrites the old initialized stack, or clears the stack. As seen in Fig.7, the disassembly of ”puts” shows the pushing of 8 bytes of values in the stack(marked in red). Beyond this, 4 bytes of return address will be present, so 12 bytes are added here. After this, we call the ”glibc puts” with ”jmp *%0”. This time the second string is printed out by the library function.

Using both these techniques, we got the ‘impossible‘ output as seen in Fig.5.

C. Unexpected Results

One of the interesting things is the stack allocation seen in Fig.7. An extra register of 4 bytes, EBX is pushed, which is not required since we are not doing any calculation/ evaluation.

D. Limitations and Considerations

The experiment was done on an Ubuntu VM and these address mapping changes if performed on another OS. Also here the hack works because we are able to see the addresses in memory and would be difficult in real life. Moreover, different compilers may address memory differently than our GCC compiler.

Appendix A
The ”cerberus.c” code:

```
/*
  * cerberus.c, Impossible statement
*/

 #include <stdio.h>

 int main(int argc, char **argv) {
         int a = 13, b = 17;
```

```
        if (a != b) {
                printf("Sorry!\n");
                return 0;
        }

        printf("On a long enough\
        timeline the survival rate\
        for everyone drops to zero\n");

        exit(1);
 }
```

Appendix A
The ”megatron.c” code:

```
#define _GNU_SOURCE
#include <stdio.h>
#include <dlfcn.h>
#include <stdarg.h>

/* Pointer to the original
*  puts call */

static int (*_puts)\
(const char *str) = NULL;

int puts(const char *str)
{
    if (_puts == NULL)    {
        _puts = (int (*)(const char *str))\
        dlsym(RTLD_NEXT, "puts");

        // Hijack the RET address and\
        modify it to <main+xx>

        __asm__ __volatile__ (
              "movl 0x4(%ebp), %eax \n"
              "addl $13, %eax \n"
              "movl %eax, 0x4(%ebp)"
        );

        return 1;
    }
    __asm__ __volatile__ (
          "addl $12, %%esp \n"
            "jmp *%0 \n"
                   :
                   : "g" (_puts)
                   : "%esp"
             );
            return -1;
}
```

Appendix C
Commands loading the wrapper library

```
$gcc -fPIC -o megatron.o megatron.c -c -m32

$gcc -shared -o megatron.so megatron.o -ldl -m32

$LD_PRELOAD="./megatron.so" ./cerberus
```

References

[1] Bhattacharyya, D.: In: ENPM691 Homework Assignment 3. pp. 1–3 (2024)

[2] Bhattacharyya, D.: In: ENPM691 Homework Assignment 2. pp. 1–4 (2024)