Ltrace

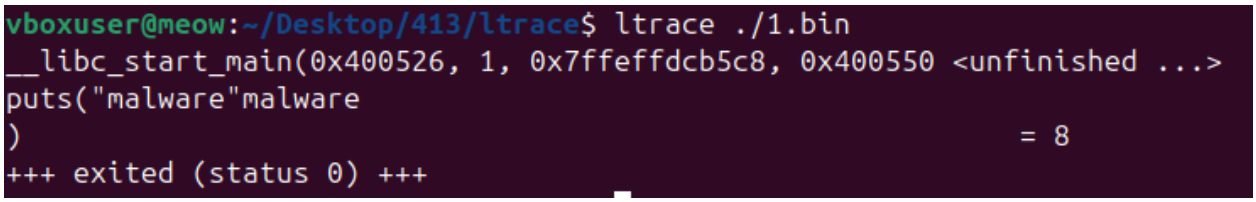
Code can be found here: https://github.com/c2003-tamu/413

Environment Setup

- Clone git repository
- Navigate to 413/ltrace
- Ensure ltrace is installed
    - Sudo apt install ltrace

1.bin

Ltrace



From the output above, we can see that essentially the only dynamically linked function calls are __libc_start_main and puts. __libc_start_main is essentially what is starting up our program, and puts is simply putting a string to stdout. Not super helpful.

Strace

```
vboxuser@meow:~/Desktop/413/ltrace$ strace ./1.bin
execve("./1.bin", ["./1.bin"], 0x7ffe049e5200 /* 49 vars */) = 0
brk(NULL)                               = 0x26aa9000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x747fdbe90000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=59839, ...}) = 0
mmap(NULL, 59839, PROT_READ, MAP_PRIVATE, 3, 0) = 0x747fdbe81000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\220\243\2\0\0\0\0\0"..., 832) = 832
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
fstat(3, {st_mode=S_IFREG|0755, st_size=2125328, ...}) = 0
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
mmap(NULL, 2170256, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x747fdbc00000
mmap(0x747fdbc28000, 1605632, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) = 0x747fdbc28000
mmap(0x747fdbdb0000, 323584, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1b0000) = 0x747fdbdb0000
mmap(0x747fdbdff000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1fe000) = 0x747fdbdff000
mmap(0x747fdbe05000, 52624, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x747fdbe05000
close(3)                                = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x747fdbe7e000
arch_prctl(ARCH_SET_FS, 0x747fdbe7e740) = 0
set_tid_address(0x747fdbe7ea10)         = 3823
set_robust_list(0x747fdbe7ea20, 24)     = 0
rseq(0x747fdbe7f060, 0x20, 0, 0x53053053) = 0
mprotect(0x747fdbdff000, 16384, PROT_READ) = 0
mprotect(0x600000, 4096, PROT_READ)     = 0
mprotect(0x747fdbece000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0
munmap(0x747fdbe81000, 59839)           = 0
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}) = 0
getrandom("\x90\xb6\xda\xe7\x34\x65\x16\xc7", 8, GRND_NONBLOCK) = 8
brk(NULL)                               = 0x26aa9000
brk(0x26aca000)                         = 0x26aca000
write(1, "malware\n", 8malware
)               = 8
exit_group(0)                           = ?
+++ exited with 0 +++
```

The strace output again is not super useful. Essentially all this tells us is that write() is called in order to put “malware\n” to stdout.

Analysis

Using gdb, we can see that this code has a statically linked function called malicious() that it calls in its main() function.

```
(gdb) disassemble
Dump of assembler code for function main:
   0x0000000000400526 <+0>:     push   %rbp
   0x0000000000400527 <+1>:     mov    %rsp,%rbp
=> 0x000000000040052a <+4>:     call   0x400544 <malicious>
   0x000000000040052f <+9>:     test   %eax,%eax
   0x0000000000400531 <+11>:    jle    0x40053d <main+23>
   0x0000000000400533 <+13>:    mov    $0x4005d4,%edi
   0x0000000000400538 <+18>:    call   0x400400 <puts@plt>
   0x000000000040053d <+23>:    mov    $0x0,%eax
   0x0000000000400542 <+28>:    pop    %rbp
   0x0000000000400543 <+29>:    ret
End of assembler dump.
```

This malicious() function has the following instructions:

```
Dump of assembler code for function malicious:
   0x0000000000400544 <+0>:     push   %rbp
   0x0000000000400545 <+1>:     mov    %rsp,%rbp
=> 0x0000000000400548 <+4>:     mov    $0x1,%eax
   0x000000000040054d <+9>:     pop    %rbp
   0x000000000040054e <+10>:    ret
End of assembler dump.
```

All this function does is essentially return 1.

Going back to the main() function, we can see that test %eax, %eax is the next instruction. This is essentially checking if eax is zero or not and then if it is, flipping the ZF flag. Then, if this flag is set to 1, the program skips over putting “malware\n” to stdout and just returns. This behavior can be seen below (note that nothing is put to stdout after I set the ZF flag to 1):

```
(gdb) disassemble
Dump of assembler code for function main:
   0x0000000000400526 <+0>:     push   %rbp
   0x0000000000400527 <+1>:     mov    %rsp,%rbp
   0x000000000040052a <+4>:     call   0x400544 <malicious>
   0x000000000040052f <+9>:     test   %eax,%eax
=> 0x0000000000400531 <+11>:    jle    0x40053d <main+23>
   0x0000000000400533 <+13>:    mov    $0x4005d4,%edi
   0x0000000000400538 <+18>:    call   0x400400 <puts@plt>
   0x000000000040053d <+23>:    mov    $0x0,%eax
   0x0000000000400542 <+28>:    pop    %rbp
   0x0000000000400543 <+29>:    ret
End of assembler dump.
(gdb) x/10x $eax
0x1:    Cannot access memory at address 0x1
(gdb) set $eflags |= (1 << 6)
(gdb) i r eflags
eflags         0x242              [ ZF IF ]
(gdb) c
Continuing.
[Inferior 1 (process 5723) exited normally]
```

Normal Run

```
vboxuser@meow:~/Desktop/413/ltrace$ ./1.bin
malware
```

Seeing as the malicious() function is simply returning 1, it makes complete sense that “malware\n” would be put to stdout. It should only not put it to stdout if malicious() returns 0.

2.bin

Ltrace

```
vboxuser@meow:~/Desktop/413/ltrace$ ltrace ./2.bin
Couldn't find .dynsym or .dynstr in "/proc/3973/exe"
malware
```

This ltrace is even less useful than the ltrace for 1.bin. It doesn't even indicate the puts() or printf() or any other function was used to interact with stdout. This seems to insinuate that these functions were statically linked in the file, rather than dynamically linked.

Strace

```
vboxuser@meow:~/Desktop/413/ltrace$ strace ./2.bin
execve("./2.bin", ["./2.bin"], 0x7ffcd2676950 /* 49 vars */) = 0
uname({sysname="Linux", nodename="meow", ...}) = 0
brk(NULL)                               = 0x17f87000
brk(0x17f881c0)                         = 0x17f881c0
arch_prctl(ARCH_SET_FS, 0x17f87880)     = 0
readlink("/proc/self/exe", "/home/vboxuser/Desktop/413/ltrac"..., 4096) = 39
brk(0x17fa91c0)                         = 0x17fa91c0
brk(0x17faa000)                         = 0x17faa000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}) = 0
write(1, "malware\n", 8malware
)                 = 8
exit_group(0)                           = ?
+++ exited with 0 +++
```

From this output, we can see that a lot of the setup that was previously done in 1.bin is skipped this time (weird, probably has something to do with the fact that this attacker has statically linked things in their code rather than using dynamic linking). We also see that the write() syscall is used to put "malware\n" to stdout. When paired with the information seen above, we can deduce that the crafter of this malware statically linked their puts() (or printf() or any other function used to put something to stdout) to their binary and manually called the write() syscall from there or their compiler optimized their syscall into write().

Analysis

When we get into gdb, we can immediately see that puts() is called, but is not called from plt (procedure linkage table). This means that puts() was manually put into their file, rather than dynamically linking (compare the puts on line +18 to the puts@plt on line +18 in 1.bin).

The rest of the executable is exactly the same as 1.bin, just with statically linked functions instead of dynamically linked. This can be seen in depth when inspecting the objdump -d of the file, where we get a massive response, seen in 413/ltrace/objdump2.txt. When we compare this to 413/ltrace/objdump1.txt, we can see that there is a large amount of stuff in 2.bin than there is in 1.bin, presumably because the attacker statically linked libraries used in 2.bin and dynamically linked them in 1.bin.

Normal Run

Given that this is essentially exactly the same as 1.bin, the behavior being the same is expected, as the malicious() function still always returns 1, and the program will only not print “malware\n” if it returns 0.