System Security
Buffer Overflow

Graded Assignment

Distribution: 24.10.2019
Hand in: 07.11.2019 13:15

This exercise is about understanding stack frames and buffer overflows. You will need to perform two return-to-libc attacks. You have the option to use different tools to exploit a vulnerability in the attached executable file.

The `bufov.zip` archive contains the binary that you must use for this exercise and a LaTeX template for your report. Your solution must be uploaded to Moodle as a pdf.

The VULNAPP program reads user input in two different ways. 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.

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`.

Tools

Note: The following tools are already installed in the virtual machine. You can use any of them. Different tools work well for different tasks.

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 classic debugger features. It is a command-line tool.

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

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.

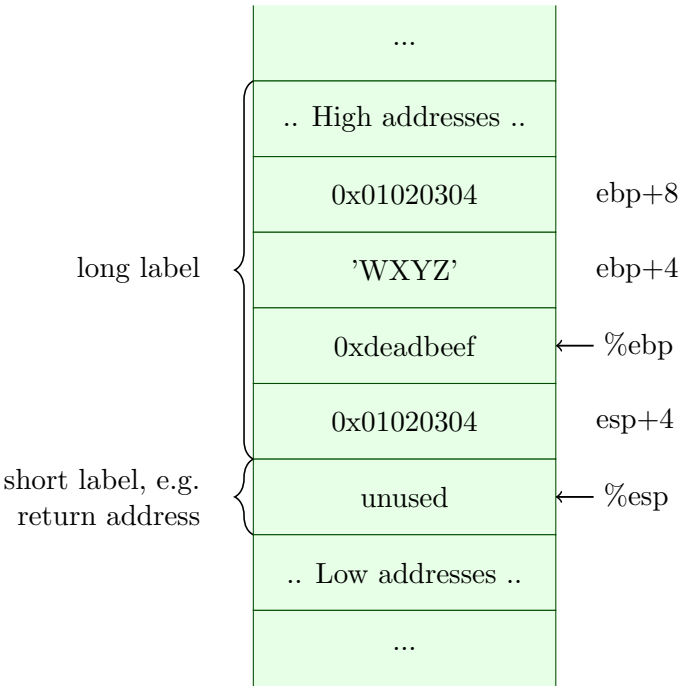
   (a) Location: before the execution of `cpybuf` begins (`0x080488a5`)
   Contents: From `esp` to `esp + 7` inclusive

   (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 a 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 the four stack frame diagrams.

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

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 folder containing `vulnapp`, 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 you keep address randomization disabled. 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 of how to call vulnapp with the ASCII characters "AAA" followed by 0xbfff5ef0 as argument:

```
> ./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

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/sh).

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 so that it looks like it would before a regular function call.

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 that you started as normal user (syssec). Therefore, during the attack, the program must be started as `./vulnapp <arg>`.

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`.

Attack 2: Execute the shell from environmental variables

Not every program might allow you to place a string in memory through an extra input as vulnapp. 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`.

Further options

Name another option where the argument for the call to system could be stored.
Name another option to terminate the program without a segmentation fault and without jumping directly to exit().

Some Advice

- In your report, explain what you did, how you found the addresses you used, and clearly specify whether your attack worked as expected or not.
- 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 with `gdb -p <pid>`.
- 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.