GRASS - Project report

Delphine Peter, Yann Vonlanthen, Tiago Kieliger

School of Computer and Communication Sciences

May 2019

1 Exploits

1.1 Exploit 1 - A buffer overflow

The first buffer overflow vulnerability is in the function `exec` in `command.cpp`. One can exploit it to overwrite return address with the address of `hijack_flow()` and thus redirect the execution flow.

The buffer overflow vulnerability comes from a call to `strncpy` in `exec` that will copy up to `BFLNGTH = 652` characters into a buffer of size 150 bytes. The offset between `buffer` and the return address previously pushed on the stack being less or equal to `BFLNGTH`, one can simply send a command such as `ping AAAAA...` long enough to overwrite the latter return address. In order to hijack the flow, an attacker must write the address of `hijack_flow()` (easily found with gdb) in the `ping` command argument. Then, when returning from `exec`, the address of `hijack_flow()` will be popped into the register `$rip` and the execution flow will indeed be redirected to the custom function.

1.2 Exploit 2 - Another buffer overflow

The second buffer overflow vulnerability is in the function `parseCommand` in `Parser.cpp`. It is very similar to the first vulnerability. Before being parsed, the command is copied in the buffer through a call to `strncpy` that will copy at most 950 in a 750-byte buffer. As explained before, an attacker can overwrite the return address stored on the stack by sending a 950-byte-long command that contains the address of `hijack_flow()`.

Here the tough part was to make both buffer overflow vulnerabilities exploitable. Since `parseCommand` and `exec` are called sequentially, the offset to the return address in `parseCommand` must be strictly larger than the offset in `exec`. Otherwise, exploiting the bof vulnerability in `exec` would first overwrite the return address in `parseCommand` (called before `exec`), which will either trigger a Segmentation Fault or will exploit the first vulnerability a second time by redirecting to `hiijack_flow()` directly when returning from `parseCommand`. It would thus never reach `exec` so the vulnerability in `exec` would not be exploitable.

1.3 Exploit 3 - Format string

Exploiting format string vulnerabilities for an x86-64 bit architecture is much harder than for an x86-32 bit architecture. Since addresses are so long, oftentimes they start by two null-bytes, i.e stack address would have the following format: `0x00007fffffffabc0`. As our GRASS program doesn't accept null-bytes, it is impossible to directly store a valid address on the stack by simply giving it as input.
Instead, format string vulnerabilities can be exploited using two levels of indirection. I will quickly describe the high level idea of such an attack. We can start by exploring the stack and finding a stored word that points to a nearby stack location. Let's call the address where this pointer is stored *addr1* and the stored pointer *addr2*. At the stack location *addr2*, we require that a third pointer is stored, we call it *addr3*, that points to an area on the stack close to the current functions return instruction pointer. Using the format string vulnerability **repeatedly** we can now use the pointer at *addr1*, to increment the pointer at *addr2*, such that *addr3* is modified to point to *rip*. Finally using this pointer at *addr2* we can overwrite the last bytes of *rip* to the address of `hijack_flow()`. As we can only write byte by byte using `%53$hhn` this attack requires many consecutive vulnerable print calls, which is seems really doesn't fit our program architecture. After many unsuccessful

attempts we changed our vulnerability such that it is easier to exploit.
First, we use the $SIZE parameter of the **put command** to store a valid 64-bit address on the stack. Then we use this address to write to the stack location of a previously declared variable. The program will then recognize that this variable's value has changed, and invoke `hijack_flow()`. This somewhat artificial construct is necessary, as the address of `hijack_flow()` isn't close to the return instruction pointer at all, which would force the attacker to write multiple bytes, which again, can't be done in a reasonable fashion in a 64-bit system, for the reasons explained above.
Finally our input command that exploits the vulnerability looks as follows:
`put a%53$hhn 140737488342272`

1.4 Exploit 4 - Command injection

This exploit allows an attacker to run arbitrary commands on the server machine. In particular, an attacker can use it to open a calculator.
In our architecture, commands are simply executed by system calls on the server machine. For example, in the *cd* command (line 414 of file *commands.cpp*), the actual command being executed (more or less) is *" cd \" " + escape(path) + " \" "* where `path` is user supplied. The basic idea behind this is that if a user supplied string is trivially inserted in a command, an attacker could insert an arbitrary command using && which allows multiples commands to be executed at once. For example, assume that the command is *string cmd = "cd "+path*. An attacker can provide a path such as `path = "/some/path␣&&␣xcalc"` and run a calculator. Hence, to allow only one command to be executed, we insert quotation marks around the argument of the command to guard against the use of &&. Of course, the attacker could himself insert quotation marks inside `path` to close our guarding quotation marks and inject a new command, so we also escape all quotation marks from the user supplied `path`.
Back to the attack: It turns out that one of our programmers forgot to escape quotation marks from the user supplied `host` in the *ping* command (line 346 from *commands.cpp*). So an attacker can use && to run an arbitrary command.
Wrapping up, to trigger the exploit one has to send the command :
ping google.ch&&xcalc

1.5 Exploit 5 - You can't break what's broken

This exploit is based on two different bugs and allows an attacker to run the `hijack_flow()` function.
Firstly, one has to notice that in the file *parser.cpp* at line 236 there is the switch case for the *exit* command. One of our programmers forgot to check the number of arguments. Moreover the clumsy programmer also forgot to write a `break;` statement which leads the `default` body to be executed. (For any other command except *exit*, the `default` body is never executed).
Secondly, inside the `default` body the string `BACKDOOR_SECRET` is initialized with 128 completely random characters. It is then compared to the argument given with the *exit* command before triggering `hijack_flow()`. At first sight, it seems impossible to guess the correct `BACKDOOR_SECRET`, however the trick is that in the comparison `if (tokens[1] == BACKDOOR_SECRET)` the two 0's of secret are actually O's. So we are not comparing with the real `BACKDOOR_SECRET` variable which is totally random, instead we are comparing it with another `BACKDOOR_SECRET` variable defined on line 19

of parser.cpp which is initialized as : `string BACKDOOR_SECRET = "none"`. This type of attack is called homograph attack. A more sneaky way to hide this vulnerability, would have been to replace the E characters with the Cyrillic Epsilon, which is displayed as the same character in most editors. However, sadly Unicode characters in variable names are not supported by GCC yet. (Note that it would work with CLANG)
Wrapping up, to trigger the exploit one has to send the command `exit none`

2 Backdoors

2.1 Backdoor 1 - Hold the backdoor !!!

This first backdoor can only be exploited through a first preimage attack. That is, on a `login` command, a call to `checkBackdoor()` (command.cpp, line 294) will check if the hash of the given username is equal to some constant stored in command.cpp and, if the test succeeds, the function `hijack_flow()` is called. The latter constant precisely corresponds to the hash digest of the string "H0D0R". Therefore, sending the command `login H0D0R` exploits the backdoor and makes the server run `hijack_flow()`.

Thanks to his beloved mentor S. Vaudenay, the crypto apprentice knew that it is very hard to find a so-called preimage, i.e. a string that hashes to the same value as our secretly chosen phrase. However, he did not take into account that since the string is only 5 chars long, it is easy to bruteforce it !

Note that due to SHA functions being part of the `openssl` library which is not pre-installed on Kali Linux, we simply used the default `hasher` object in C++.

2.2 Backdoor 2 - Sanitize or sanitizn't ?

This second backdoor is well hidden inside the helper function `sanitizePath()` in the file `commands.cpp`. This function is used multiple times to sanitize user input, for example to avoid tricks such as *mkdir abc/../../def*, which could allow an attacker to reach outside the base directory. However, on line 153 we have a check `else if(strcmp(token, "NULL") == 0)` which is triggered when part of the path is equal to NULL. Then the following call will start a calculator : `exec(new char[6]{120, 99, 97, 108, 99, 32}, out);`. Instead of passing "xcalc" to the exec function (which would be too obvious) we pass the integers that correspond to {x, c, a, l, c, }.
To wrap up, one can trigger this exploit by calling for example : *mkdir NULL*

2.3 Backdoor 3 - Come and Get the calculator !

The third backdoor is well hidden in the already pretty convoluted `get_cmd()` function in the file `commands.cpp`. On line 533, we define `uint64_t maxFileSize = 9042810646913912;` which may appear to be a somewhat legitimmate maximum file size. however, it is in fact the decimal representation of a char array of 8 characters containing the word xcalc! When the file size is 0 and the first character of the filename is a dot, the following command is executed : `exec((char*)&maxFileSize, out)` which starts a calculator! So to trigger the exploit, one should simply put an empty file with a name starting with a dot an then get that specific file.
To wrap up, the exploit can be triggered by calling for example *put .exploit.txt 0* and then calling *get .exploit.txt* where .exploit.txt is an empty file that has to be created in the same folder as the client executable.