Milestone Exploits

Jack Eadie, jeadie, 20865679

Exploit 1

The first exploit utilises vunerabilities in the string formatting used in the usage function. The usage function outputs a usage message to the user if they have an incorrect amount of inputs. It dynamically takes the name of the program before the other arguments in the form:

```
"Syntax: %.60s backup|restore pathname password%c"
```

The program formats this into a buffer variable before copying it over to an output before printing. Of the two printf function only the first protects against strings that would overflow the buffer, by cutting the parameter off at 60. This, however does not account for the second string formatting. A malicious actor could insert `"%60d"` which would be accepted by the first printf but then expanded in the latter, unprotected printf, resulting in the output variable being overflowed.

The exploit does exactly that. The parameter passed in is of the form.:

```
<SHELLCODE>%51d<ADDRESS OF SHELLCODE>
```

It is less than 60 characters and the formatter `"%51d"` is not translated until the second printf. This padding coincides with the start of the return address which allows for the return address to be overwritten. When the usage function is finished, instead of returning to main, it jumps to the start of the shellcode and opens a root shell.

An obvious mitigation to this would be to not have both printf formatters and to always use fixed sized string methods such as snprintf.

Exploit 2

The second exploit is a buffer overflow when loading a file into memory during the backup process. This exploit requires bypassing the password which is achieved by utilising an integer overflow to mitigate password length checking so that an overflow could override the `accept` flag.

checkPassword Bypass

The first check is at backup.c:177, `strlen(argv[3]) > 128`. So, setting the password to exactly 128 characters will still work. The next check is before copying the user's password to memory, `len <8`. The problem with this is the variable len is instantiated as a char, which when compared against an int, is converted in the range [-127, 127]. A 128 byte password (which clears the first check) would be converted to -127 passing the second check. This leads the `strcpy(pw, user_pw)` on backup.c:154 to copy 128 bytes into `pw`, which is an 8 bytes buffer. A buffer overflow attack can then overwrite the variable `accept` to be non-zero. This variable does not get set to 0 if the user's password is incorrect, only set to one if they are equal. This will allow the accept variable (overwritten in the buffer overflow) to be returned not equal to zero: tricking the program into thinking the checkPassword has validated the password to be successful.

copyFile Exploit

There is another buffer overflow vunerability in copyFile. There is no checking that the file will overflow the 2560 character buffer. Hence, filling a file with more than 2560 characters and running the `backup` command will load the file into the buffer, but force an overflow. The relative memory (from the start of the buffer) is laid out in the form:

Relative Memory Address	0	46	2560	2564	2568
Variable Name	buffer	buffer	len	i	Return addr
File Content	shellcode	Non-zero char	Non-zero char	rjump_addr-1	shellcode_addr

The file was first filled with shellcode, then spaced to the end of the buffer (to 2560 characters) and then 4 more bytes to overwrite the the variable len. The next character in the file is 0x0b. This overwrites the first byte of the variable i; changing it from 0xA04 (2564) to 0xA0B (2568. The next line (backup.c:39) then increments i to 0xA0C, pointing the buffer to overwrite the return address of the function. The file then has the address of the shellcode (or equivalently the start of the buffer) which will get written on top of the previous return address. When the function completes, it jumps to the shellcode (instead of back to `main.c`) and opens a root privilege shell.

Mitigation

Both the checkPassword integer overflow and the copyFile overflow could be mitigated. For the checkPassword, if the len variable was an uint then the subsequent strlen would have correctly identified it as more than 8 characters. A better stategy would be to replace `strcpy(pw, user_pw)` with `strncpy(pw, user_pw, 8).` This would only copy the first 8 characters regardless of the user_pw and still allow for valid password checking. The copyFile vulnerability could be avoided by using copying the file in segements. When `i>2560` the buffer is then copied into the destination file, `i` reset to 0 and more characters from the src file read over again.