Project Members: **Ryan Fontaine**

Project Option: **Automatic Exploit Generation**

Project Title: **ChatGDB**

Description of Problem

Binary exploitation CTF problems can often be very complicated and difficult to understand quickly. Because of their difficulty, solving Binary CTFs can consume a lengthy amount of time. To maximize points during CTF competitions it is vital to solve problems quickly.

This project aims to hasten the solving of Binary CTFs by creating an Automatic Exploitation Generation (AEG) tool called 'ChatGDB'. This tool has various functionalities useful for solving simple buffer overflow Binary CTFs, notably it can quickly perform dynamic brute forcing, create a custom payload, build a simple shellcode payload, and print function addresses from executables.

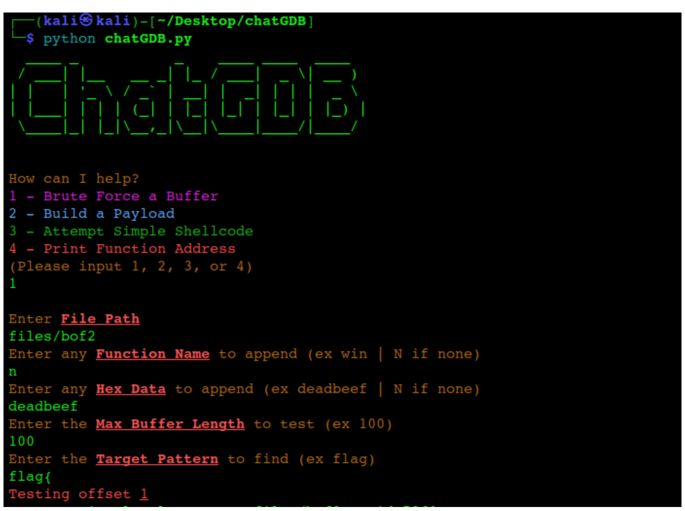
Currently this tool assumes correct user input, input files being ELF executables, and the input files being 32bit programs. It also assumes there is no stack guard or randomization. In its current state, only simple buffer overflow problems can be solved. However, it can also be a proof of concept for future additions.

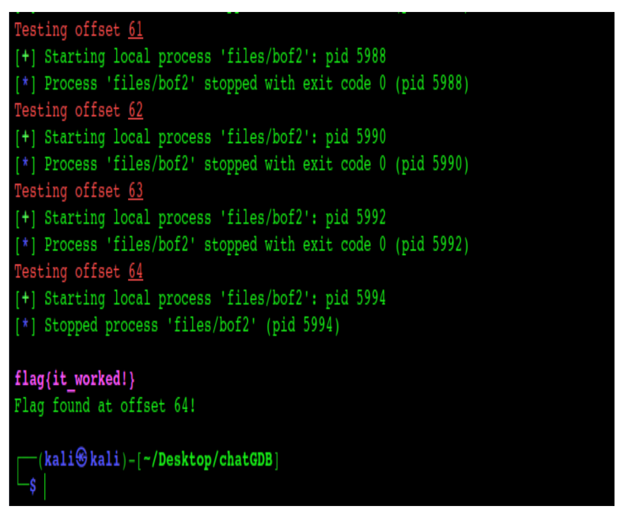
The tool can be launched from the main folder using 'python3 chatGDB.py'. The testing files used can be found in the subfolder 'files'. There are also temporary 'flag.txt' files that are used by the testing files.

Description of the exploits and techniques

This program is a Command Line Interface tool that is built around four main functionalities.

1. "Brute Force a Buffer". Here a user inputs dynamic data including: 'File Path', 'Any Function Names to Append', 'Any Hex Data to Append', 'Max Buffer (Payload) Length', and a 'Target Pattern' the program can use to filter output from testing files. Here the function will test payloads of increasing length up to the specified Max Buffer Length and will add any addresses of given functions or hex data to the end of the payload. If the response from the testing file includes the specified Target Pattern, then the brute forcing will stop, and the flag will be output.





2. “Build a Payload”. In this function, a user can create dynamic payload for various needs across different CTFs. The user inputs the amount of filler bytes first, used to overflow a buffer. Then the user can add any function address or hex data to the end of the payload. This is useful because depending on the users’ choices, many different payloads can be made. In this screenshot, a payload is made where the EBP register is overwritten with the address of the win function and the parameter 0xdeadbeef is passed alongside it. However, the return address was overwritten with @@@@ so it will lead to a segmentation fault. With this dynamic input, a user can also create payloads with no parameters, or simple Return Oriented Programming payloads as well. After completion, ChatGDB will output the users’ created payload which they can then use to complete CTF challenges.

3. “Attempt Simple Shellcode”. The third functionality of ChatGDB can be used to create a simple payload that includes shellcode. This shellcode is built for 32bit ELF executable files, and was sourced from the [Buffer Overflow Seed Lab] completed for Project 2. The function assumes the shellcode is placed inside the buffer and the program should jump to the start of user input. It prompts the user for the size of the buffer, then the offset of the shellcode from the start of the buffer, and lastly the address of the buffer storing user input. It then creates a NOP sled followed by the shellcode. If given correct buffer size, then the return address will be overwritten with the given address of the user input buffer.

4. “Print Function Address”. This final functionality can be used to quickly print a function address from a given ELF executable. I had some difficulties with this as I was unable to solve the picker-IV executable file with ChatGDB. The file asks for the address of the win function, however, when using ChatGDB to find the win address it did not accept the input. When using the actual gdb function, I found that the win address was different than the ChatGDB output. If I had more time, I would continue to investigate this and see what was causing this bug.

Files:

The main file for user input is found in ‘ChatGDB.py’. This script calls the remaining other files depending on which functionality the user requests. The sub-scripts ‘InputMenu.py’ and ‘SendOffset.py’ are called inside ‘bruteForceBuffer.py’. The remaining files: ‘bruteForceBuffer.py’, ‘buildAPayload.py’, ‘createShellcode.py’, and ‘printFunction.py’ contain the main functions offered by ChatGDB.

There are six testing files found in the ‘files’ subfolder. Five of the six were able to be solved using ChatGDB; picker-IV was not able to be completed due to the input format needed. However, this could be added in future additions. Another future addition would be to allow the user to choose what character they want to fill the buffer with.

Results/demonstration examples

I have recorded a video demonstration of the results:

https://www.youtube.com/watch?v=Q7hjyswH-pU