Lecture 11 Notes

Robustness

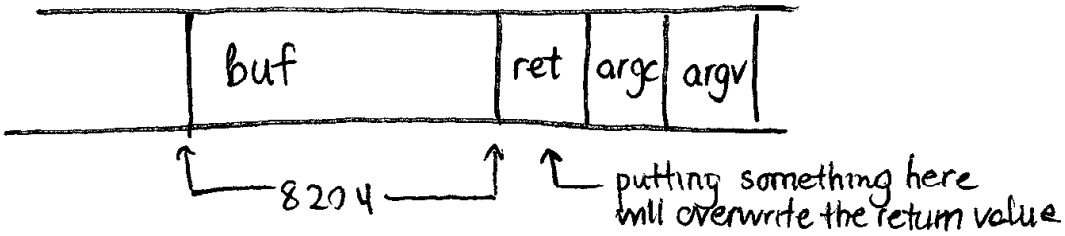
- Ability to handle or repel attacks
- Buffer overrun attack (stack smashing)

C Library

Has both safe and dangerous functions:

Dangerous	Safe
fgets(char *buf)	fgets(char *buf, int size, File *f)
This read line from stdin, terminating at \n	This reads line from f, terminates at the end of a line OR at the size limit
This gives to way for the user to specify how large the buffer is and so there is no way for the user to prevent errors	This is performed using a minimum (whichever comes first, line break or size limit), giving the user a way to prevent errors

Smash01



- Input 8204 bytes of character 'y' followed by a memory address of bomb
- Distance between buffer and return address is 8204
- If we write over 8204 we OVERWRITE our return address
- When we run this, the program prints "BOOM!"
- This type of problem is a billion dollar concern
  * Crashed the Internet in the 1980s (Morris worm) by Robert Morris

Fixing Stack Smashing

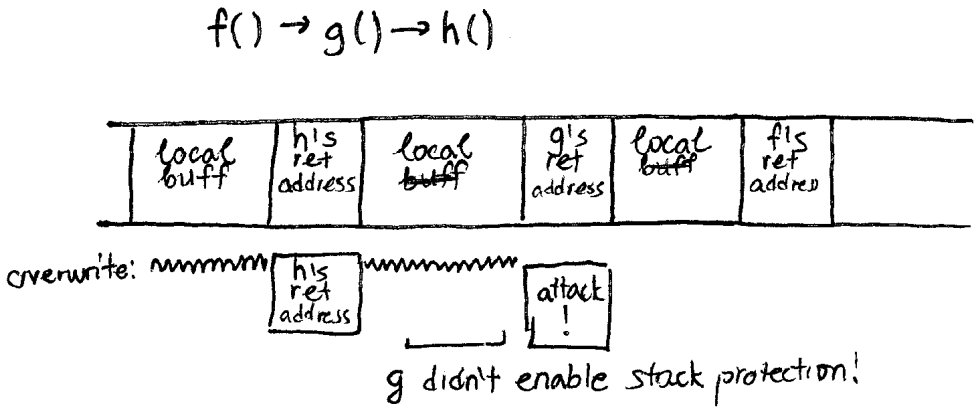
- We can limit the input size via a safe API
  - In effect, this is the only fix
  - Less than 1% of computer programs in world are likely correct
- gcc detects stack smashing already (Professor turned off module for smash01)
  - Module called -fno-stack-protector which restricts such protections

How stack smashing protection works

- On entry to a function:
  - Generate an unpredictable number, let's call it SM (Stack Magic)
    - Important that the number is unpredictable, otherwise attacks would be easy
  - We store SM on the stack (next to return address and old %ebp)
  - We also store SM in a secret other area (hidden storage)
- On exit of a function:

- Load SM from hidden storage
- Compare with the stack value
- FAIL if the values are different

- Where is the hidden storage?
  - Done using a special operator (i.e. %gs:0x14, edx), as we can see in gdb
  - Despite being a special memory area, this is best thought of as global storage duration
  - This is equivalent to placing a "canary value" onto the stack
- Downsides of such protection
  - Expensive, because stack protection must be handles in real time
  - Generating random numbers is particularly expensive
  - Fortunately, only one random number is generated per execution
  - Also, stack protection turns arbitrary execution failures into crash failures
    - (meaning you can take the program offline, but doesn't let you steal money, for example)
  - This is better than execution failures, but allows for "denial of services" attack
    - ("I will only restore the program if you pay me money")
- Possible alternative:
  - Say we use the return address as the SM number
  - Say we have f() calls g() calls h()
  - On entry we copy the return address to hidden storage
  - On exit, compare
  - The failure of this method is that f() can copy new return address into g()'s frame
  - g() thinks it is safe (stack protection is only enabled for functions that have a buffer), so it does not perform a check and is attacked



Smash02

- This program is running with stack protections
- And yet, the program IS attackable

```
class animal {
        name
        will_eat (int (*)(animal *, const char *)) // function pointer
        next (pointer)
    }
```

- Animals are arranged in a linked list
  - First defined animal is a squirrel, next is lion, third is fly
  - Function pointer is called as follows: "a->will_eat(buf)"

        - Function that gets called is determined at run time
    - In the previous attack, if we search for BOMB we find a function linked into the executable
        - In smash02.c, BOMB does not appear anywhere in the executable, so where is it?
            - Every program has its own independent memory space, and BOMB is not there
            - The computer will REFUSE to execute any instruction that is not in memory
            - Can we point it to a file including the instructions?
            - No, because files are ALSO not in memory
        - An interesting input: "echo 'BOOM!!!';sleep5^@yyyyyyy...yyy" followed by 0x0804a230
            - If we do a "nm smash02" on the executable, we find that the hex address points to a system command, __lib_c_system
                - Note: this is not stack smashing!
            - The buffer being overrun is located in heap memory, not on the stack
            - Of course, the pointer to the buffer is located on the stack
            - Because of malloc, the animal structures are located on the heap after the buffer
            - The overrun buffer is able to place the system function address into the lion's will_eat
            - The program ultimately asks the shell to do an attack itself
                - This type of attack is called a "return-to-libc" attack
                - Fundamentally, the attack is possible because gets() is used
                - The function system() runs a shell

Smash03

- If you understand smash02, you understand smash03
- What's interesting about this is that we have a size limitation on the input
- However, by taking advantage of INTEGER OVERFLOW BUGS... We can alter the size variably and create an attack
- The software designer can nonetheless prevent this by avoiding integer overflow bugs
- Note: it would be ideal to avoid artificial size limitations such as in smash03
- The user should be able to define the max size for an animal name, not the program

Preview of next lecture...

- WORST ATTACK IN THE WORLD!

```
while (1) {
}
```

- The processor only does what it is told, so it will continually jump to the same instruction
- There is no reason in the code that the processor will ever run a different instruction
- Solving the problem of infinite loops will take multiple lectures to solve
- Let's try booting our tiny operating system in a virtual machine...
    - In our tiny operating system, infinite looping in one process halts other processes

- We will investigate how this is fixed next time!