COSC 458-647
Application Software Security

Midterm Review

Topics

- x86 Memory organization
- GDB debugger
- Shellcode
- Buffer overflow

Memory organization

- Memory addressing modes
    - Real mode
    - Protected mode
    - Linear addressing
    - Conversion among them – Quiz 1
- Memory segmentation
    - 5 sections of a compile C program
        - Readable/Writable?
        - Fixed sizes?
        - Where are they in the main memory?
    - How does the stack work?
    - How does the heap work?
        - Why are they moving in different directions?

x86 registers

- EIP?
- ESP v.s. EBP?
- Stack frames
- Memory layout when
  - A function is called
  - Subfunctions are called?
- Instructions and Arithmetic calculations with registers
  - PUSH, POP
  - MOV, LEA, CALL
  - ADD, SUB, INC, DEC
  - MUL, DIV

GDB Debugger

- How to compile with gdb?
  - -g?
- How to disassemble your object file/program?
- Read registers info?
- Read variables info?
- Examine a particular memory address?

Shellcode

- What are shellcode and what can they do? Size of a Shellcode?
- What can and CAN’T it contain?
- What is the regular form of a shellcode and why?
- What is a NOP sled? Why do we need to use it?
  - Also, can we detect a BOF attack by checking NOP sleds in the instream packages?
- Writing your own helloWorld shellcode?
- If you are given a simple shellcode, can you analyze it?

Buffer Overflow

- Explain how buffer overflows work, and explain some cause for them.
- Spot a simple buffer overflow in a piece of code.
- Explain/discuss any of the countermeasures against buffer overflows
    - In lecture 8

```
#include <unistd.h>

int main(int argc, char *argv[]) {
    char buff[100];
    /*if no argument…*/
    if(argc <2) {
        printf("Syntax: %s <input string>\n", argv[0]);
        exit(0);
    }
    strcpy(buff, argv[1]);
    return 0;
}
```

```
#include <stdio.h>
void manipulate(char *buffer) {
  char newbuffer[80];
  strcpy(newbuffer,buffer);
}

int main() {
  char ch,buffer[4096];
  int i=0;

  while ((buffer[i++] = getchar()) != '\n') {};

  i=1;
  manipulate(buffer);
  i=2;
  printf("The value of i is : %d\n",i);
  return 0;
}
```

The following code demonstrates the scenario in which the code is so complex its behavior cannot be easily predicted.

This code is from the popular libPNG image decoder, which is used by a wide array of applications, including Mozilla and some versions of Internet Explorer.

```
if (!(png_ptr->mode & PNG_HAVE_PLTE)) {
     /* Should be an error, but we can cope with it */
     png_warning(png_ptr, "Missing PLTE before tRNS");
}
else if (length > (png_uint_32)png_ptr->num_palette) {
     png_warning(png_ptr, "Incorrect tRNS chunk length");
     png_crc_finish(png_ptr, length);
return;
}
...
png_crc_read(png_ptr, readbuf, (png_size_t)length);
```