CMSC389R

Binaries II

COMPUTER SCIENCE
UNIVERSITY OF MARYLAND

recap

HW6 and HW7

--

Questions?

Itinerary

- Review
- Reverse Engineering
  - Static analysis
  - Dynamic analysis
- Tools
- Exercises

Review

- x86 Assembly
  - Registers, instructions, conventions
- Tools
  - gdb

Essence of Analysis

- Jump into completely unknown code
- Get high-level idea of operation
  - *Which* parts do computation?
  - *Which* parts do checking?
- Dig into “interesting” parts
  - *What* does this compute?
  - *Why* is there a check?
- Build complete mental map

Static Analysis

- “lacking in movement, action, or change”
- Analyzing a binary without running it
- Useful for certain circumventions
  - Malware
  - Network access
  - System modifications

Dynamic Analysis

- “stimulates change or progress”
- Analyzing a binary by running it
  - May be too complex to comprehend statically
  - May exhibit unique behavior based on environment in which it executes
- Behavioral Analysis

Tools

- *strings* - print strings of ASCII in file
  - *-e* to change encoding
  - Only prints strings >4 in length
  - Useful for hardcoded values in binaries
  - Quickly search files for known ASCII values
    - *strings memory_dump | grep “FLAG-{“*

Tools

```
[j@b0x:~][130]$ strings /bin/ls
/lib64/ld-linux-x86-64.so.2
libselinux.so.1
_ITM_deregisterTMCloneTable
__gmon_start__
_Jv_RegisterClasses
_ITM_registerTMCloneTable
_init
fgetfilecon
freecon
lgetfilecon
_fini
libc.so.6
fflush
strcpy
gmtime_r
__printf_chk
fnmatch
readdir
```

Tools

- *readelf* - information on ELF files
  - “Executable and Linkable Format”
  - Extracts metadata from binary based on ELF format
  - see ELF.png in git repo

Tools

- *file* - determines the type of a file
  - helpful to determine type of binary
    - Windows? Linux? macOS? ARM?
  - If magic bytes are corrupted this may not work

Tools

```
[j@b0x:~]$ file /bin/ls                                                    (04-27 13:59)
/bin/ls: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Lin
ux 2.6.32, BuildID[sha1]=3c233e12c466a83aa9b2094b07dbfaa5bd10eccd, stripped
[j@b0x:~]$ file Pictures/2017-08-19-220516_656x673_scrot.png               (04-27 13:59)
Pictures/2017-08-19-220516_656x673_scrot.png: PNG image data, 656 x 673, 8-bit/color RGB, non-interlaced
[j@b0x:~]$ file /etc/shadow                                                (04-27 13:59)
/etc/shadow: regular file, no read permission
[j@b0x:~]$                                                                 (04-27 14:00)
```

Tools

- *gdb* - good ol’ GNU debugger
  - *si, ni* for stepping in/over *instructions*
    - Binaries compiled w/o -d flag won’t have source code available
  - *p* to print values
    - *p $eax* to print registers
    - *p *($ebx)* to print value pointed by register
      - May need to cast pointers like in C
  - *b* to set breakpoints
    - Useful to skip computations and go to results

Tools

- *gdb*
  - Lots of useful plugins to aid in debugging
  - pwndbg - https://github.com/pwndbg/pwndbg
  - PEDA - https://github.com/longld/peda
  - GEF - https://github.com/hugsy/gef
  - gdbinit - https://github.com/gdbinit/gdbinit
  - Most add stack/register/instruction viewing windows, syntactic sugar, or architecture compatibility

Buffer Overflow

- Dynamic analysis technique
- When user input is not handled properly
- Accomplish things from variable modifications to arbitrary code execution

overflow.c

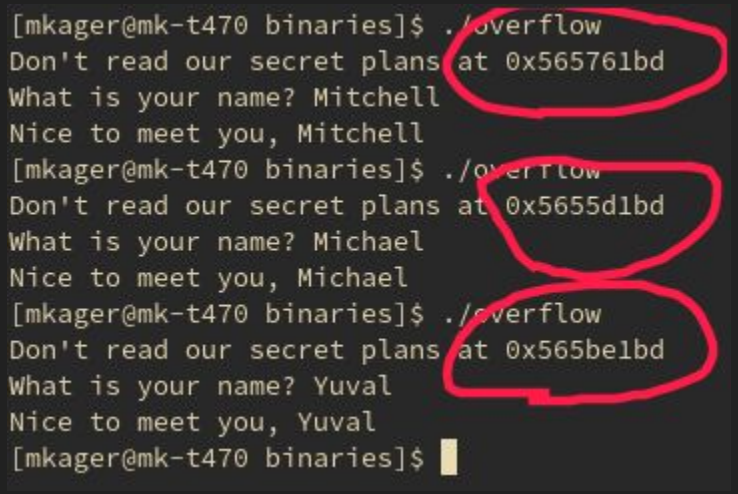
```c
#include <stdio.h>

void top_secret_function(void) {
    printf("----- ALERT: PLAN TO TAKE DOWN WATTSAMP WEBSITE IS NOW IN ACTION -----\n");
}

void greet(void) {
    char name[10];
    /* prompt user for name */
    printf("What is your name? ");
    /* read name from user via... gets? `man gets` */
    gets(name);
    /* greet the user! */
    printf("Nice to meet you, %s\n", name);
}

int main(void) {
    printf("Don't read our secret plans at %p\n", &top_secret_function);
    greet();
}
```

overflow.c

```
[mkager@mk-t470 binaries]$ make overflow
gcc -Wall -m32 -z execstack -fno-stack-protector -Og -g    overflow.c   -o overflow
overflow.c: In function ‘greet’:
overflow.c:12:5: warning: implicit declaration of function ‘gets’; did you mean ‘fgets’? [-Wimplicit-function-declaration]
   12 |     gets(name);
      |     ^~~~
      |     fgets
/usr/bin/ld: /tmp/ccAsVXmx.o: in function `greet':
/home/mkager/bin/binaries/overflow.c:12: warning: the `gets' function is dangerous and should not be used.
[mkager@mk-t470 binaries]$
```

Compilation warning -- interesting.

overflow.c

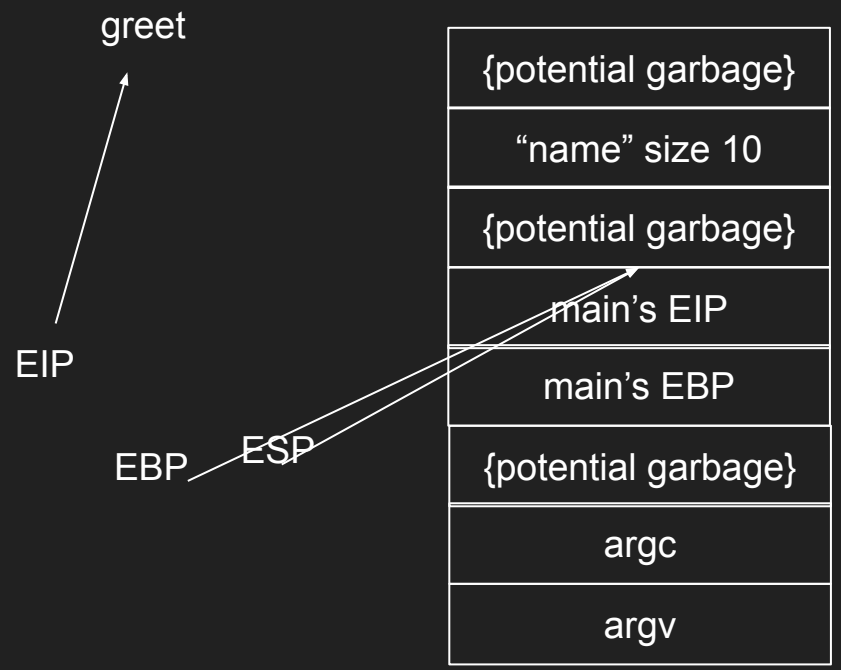


Why is the address of top_secret_function changing every execution?
How can we disable this?

overflow.c

- What is vulnerable?
- How can we exploit this?

Stack in greet



What do we control? What are the terms of the data we can put in it? Size of data, values of bytes, etc. What can we do?

overflow.c

- Very contrived example
- Consider how you might go about leaking the desired address for your EIP
  - If I can control data on the stack, I can add an execution payload to the stack (with a nop sled) and any stack address in this range will suffice
  - Else, format strings work. Might realize that known function (i.e. ‘system’) always exists t a certain offset of some other variable

format.c

```c
#include <stdio.h>

int main(int argc, char **argv) {
    /* x, a 4-byte sequence encoded as an integer, exists on the stack */
    int x = 0xdeadbeef;
    /* now a_x, a pointer to x, exists on the stack AND points to a location near it on the stack */
    int *a_x = &x;

    /* simple argc check */
    if (argc < 2) {
        fprintf(stderr, "Please provide a string to print.\n");
        return -1;
    }

    /* print the location of x this execution, might help to ensure offset$ chosen correctly */
//    printf("x location: %p\n", &x);

    /* print the value of x */
    printf("before x value: %x\n", x);

    /* print user-provided string */
    printf(argv[1]);

    /* print the value of x */
    printf("\nafter x value: %x\n", x);

    return 0;
}

```

format.c

- What is vulnerable?
- How can we exploit this?

format.c

- printf(“%d”, 1);
  - What happens?
- printf(“%d”);
- printf(“%2$d, %1$d”, 1, 2);
  - Be careful passing $ via bash -- what does this mean?
- Format strings
  - %s, %d, %u, %x, %p, ***%n***

format.c

- What is the purpose of the a_x variable?
- Try to change the value of x
- What ramifications can you imagine?
  - GOT
  - Arbitrary pointer write

format.c

- Writing 0xffffffff to an address at a known offset (i.e. %11$n), let’s say address is 0xdeadbeef
- 0xffffffff = 4294967295, can’t print that many chars (DOS)
- Can print 0xffff characters, write to 0xdeadbeec…
- Then 0xffff more and write to 0xdeadbeef!

byteorder.c

```c
#include <stdio.h>
#include <stdint.h>

int main(void) {
    int x = 0;

    *((uint16_t*)(((char*)&x))) = 0xdead;
    *((uint16_t*)(((char*)&x)+2)) = 0xbeef;

    printf("%x\n", x);
}

```

What will this output? Think hard -- what does the integer “1” look like in memory?

byteorder.c

```c
#include <stdio.h>
#include <stdint.h>

int main(void) {
    int x = 0;
    int *y;

    *((uint16_t*)(((char*)&x))) = 0xdead;
    *((uint16_t*)(((char*)&x)+2)) = 0xbeef;

    y = "\xef\xbe\xad\xde";

    printf("%x\n", x);
    printf("%x\n", *y);
}
```

What will this output?

Exercises

- Download files from git
  - Week 8, under *exercises/0x0**
- Mess around with them, get them to work!

homework #8

will be posted soon.

Let us know if you have any questions!

This assignment has 2 parts.

It is due by 5/5 at 11:59PM.

Next week’s class is our final meeting!