Oct 23 Lecture 14: Calling Convention

General Topic - Machine Programming: Attacks and Defenses

Discussion on PSET 2

- Graded by looking at how people compare to standard IO on sequential tests
- Discussion of Richard Cho and Richard Zou's PSET 2 - Their program finds where the main program keeps its system timer. IO System Profile attempts to profile your code by taking snapshots of the time of day and stores it into a global variable. Their code searches for something that matches the number of seconds for time of day at the start of the code. At the end of the program, they divide the timer by a multiplier to make their code seem faster.

The following code that is described will be additions of some form. The more challenging part is figuring out what exactly the types are of the items being added.

```
    .file   "f62.c"
    .text
    .globl  f
    .type   f, @function
f:
.LFB0:
    movl    8(%esp), %eax
    addl    4(%esp), %eax
    ret
.LFE0:
    .size   f, .-f
    .ident  "GCC: (Ubuntu 4.8.2-19ubuntu1) 4.8.2"
    .section    .note.GNU-stack,"",@progbits
```

f62.s

Description of f62.s: The assembly in f62.s is returning a + b.

```
    .file   "f65.c"
    .text
    .globl  f
    .type   f, @function
f:
.LFB0:
    movl    4(%esp), %edx
    movl    8(%esp), %eax
    movl    (%edx,%eax,4), %eax
    movl    12(%esp), %ecx
    addl    (%edx,%ecx,4), %eax
    ret
.LFE0:
    .size   f, .-f
    .ident  "GCC: (Ubuntu 4.8.2-19ubuntu1) 4.8.2"
    .section    .note.GNU-stack,"",@progbits
```

f65.s

Description of f65.s: There is one array where we are returning a[i] + a[j] where the type of array may be an int and the indexes may also be ints.

```
    .file   "f64.c"
    .text
    .globl  f
    .type   f, @function
f:
.LFB0:
    movl    4(%esp), %eax
    movl    (%eax), %eax
    movl    8(%esp), %edx
    addl    (%edx), %eax
    ret
.LFE0:
    .size   f, .-f
    .ident  "GCC: (Ubuntu 4.8.2-19ubuntu1) 4.8.2"
    .section    .note.GNU-stack,"",@progbits
```

f64.s

Description of f64.s: We are returning *a + *b. The parenthesis let us to know that we are dereferencing the first argument and the second argument. If we optimized this less, we would see a postamble and a preamble where the first argument is based off of 8 off of %ebp rather than 4 off of %esp. This is possible because this function is not calling any other function so the value of ebp does not need to be stored because %ebp does not change.

```
    .file   "f66.c"
    .text
    .globl  f
    .type   f, @function
f:
.LFB0:
    movl    4(%esp), %edx
    movl    8(%esp), %eax
    movl    4(%edx,%eax,8), %eax
    movl    12(%esp), %ecx
    addl    4(%edx,%ecx,8), %eax
    ret
.LFE0:
    .size   f, .-f
    .ident  "GCC: (Ubuntu 4.8.2-19ubuntu1) 4.8.2"
    .section    .note.GNU-stack,"",@progbits
```

f66.s

Description of f66.s: We are dealing with one array of structs that have size 8 because there are 8's in the size argument which defines the strides. The base address of each attempt to access the array is %edx and the address to the array seems to be the first argument. There are two indexes into the array and we're adding the second part of each array.

Indirect address syntax: off(base, index, size) = offset + base + index * size

```
    .file   "f67.c"
    .text
    .globl  f
    .type   f, @function
f:
.LFB0:
    movl    8(%esp), %eax
    movl    4(%esp), %edx
    leal    (%edx,%eax,8), %eax
    ret
.LFE0:
    .size   f, .-f
    .ident  "GCC: (Ubuntu 4.8.2-19ubuntu1) 4.8.2"
    .section    .note.GNU-stack,"",@progbits
```

f67.s

Description of f67.s: Generally, C does not allow us to add addresses together. Rather, in this file, we're adding the first argument to the second argument times 8. This represents returning the address of the ai'th element of an array where the leal instruction loads the address rather than deference the value. This could have also been an array of doubles.

The class was broken up into groups to tackle the files f75.s, f76.s, f77.s, f78.s, f79.s, f80.s, f81.s, f82.s.

```
    .file   "f77.c"
    .text
    .globl  f
    .type   f, @function
f:
.LFB0:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    movl    $0, -8(%ebp)
    movl    $0, -4(%ebp)
    jmp .L2
.L3:
    movl    -4(%ebp), %eax
    leal    0(,%eax,4), %edx
    movl    8(%ebp), %eax
    addl    %edx, %eax
    movl    (%eax), %eax
    addl    %eax, -8(%ebp)
    addl    $1, -4(%ebp)
.L2:
    movl    -4(%ebp), %eax
    cmpl    12(%ebp), %eax
    jl  .L3
    movl    -8(%ebp), %eax
    leave
    ret
.LFE0:
    .size   f, .-f
    .ident  "GCC: (Ubuntu 4.8.2-19ubuntu1) 4.8.2"
    .section    .note.GNU-stack,"",@progbits
```

f77.s

Description of f77.s: We are looping a number of times defined by the second argument, index into the array each time based off of what loop we are on, and take the sum of the array elements that we pass through. The file f76.s does the exact same thing, but is optimized differently so that the assembly appears different.

For the bomb, we need to treat the input like an experiment where we run experiments against the code. By changing the input, you can puzzle out what the assembly does by figuring out what input passes. We now moved onto a prior practice midterm question where we need to match f1, f2, f3, and f4 with different data structures: array, array of pointers to arrays, linked list, binary tree.

You can approach this question by looking at the different functions and seeing how complex they are to each other. Arrays only require 1 dereference, array of pointers require 2 dereferences, linked lists require loops, and binary trees require decisions and loops. Our initial hypothesis is that arrays should be the shortest assembly and binary trees should be the longest assembly.

We then look at the assembly to confirm this hypothesis. f4 only requires one dereference. f2 has two dereferences and then returns, so it is the array of pointers. f1 has loops in it, so it is the linked list. f3 has a loop and a test to move to the right or left, so it is a binary tree.

```
1  f1:
2      movl    4(%esp), %eax
3      movl    8(%esp), %ecx
4      testl   %ecx, %ecx
5      jle .L2
6      xorl    %edx, %edx
7  .L3:
8      movl    4(%eax), %eax
9      incl    %edx
10     cmpl    %ecx, %edx
11     jne .L3
12 .L2:
13     movl    (%edx), %eax
14     ret
15 f2:
16     movl    8(%esp), %edx
17     leal    0(,%edx,4), %ecx
18     movl    4(%esp), %eax
19     movl    (%eax,%ecx), %eax
20     addl    %ecx, %eax
21     movl    (%eax), %eax
22     ret
23 f3:
24     pushl   %esi
25     pushl   %ebx
26     movl    12(%esp), %ecx
27     movl    16(%esp), %esi
28     movl    20(%esp), %eax
29     testl   %esi, %esi
30     jle .L9
31     xorl    %edx, %edx
32 .L10:
33     movl    %eax, %ebx
34     andl    $1, %ebx
35     movl    4(%ecx,%ebx,4), %ecx
```

```
36      incl    %edx
37      sarl    %eax
38      cmpl    %esi, %edx
39      jne .L10
40  .L9:
41      movl    (%ecx), %eax
42      popl    %ebx
43      popl    %esi
44      ret
45  f4:
46      movl    8(%esp), %edx
47      movl    4(%esp), %eax
48      movl    (%eax,%edx,4), %eax
49      ret
```

f80.s

We are going to use the things we've learned about the stack and the way the stack is arranged to attempt to break the program. We will also focus on how the compiler stops behavior designed to break the program. We will look at stacksmash.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main() {
6      char buffer[100];
7      if (gets(buffer))
8          printf("Read %d character(s)\n", strlen(buffer));
9      else
10         printf("Read nothing\n");
11 }
```

stacksmash.c

This function reads an initialized buffer. The buffer is initialized by the function gets(), which is part of the library. The function gets() reads a line from standard in until it reaches EOF. If we run stacksmash, the program executes, saying read 0 characters. If the null file is the input, the program will output "'read nothing"'. If we run stacksmash, Hello, World, will output "Read 12 characters". However, the gets() function is very dangerous, the manual specifies that it should not be used because it does not check buffer overflow. It is impossible to know in advance how many characters gets() will read.

From UNIX man page for gets: "gets() reads a line from stdin into the buffer pointed to by s until either a terminating newline or EOF, which it replaces with '\0'. No check for buffer overrun is performed. Never use gets(). Because it is impossible to tell without knowing the data in advance how many characters gets() will read, and because gets() will continue to store characters past the end of the buffer, it is extremely dangerous to use. Use fgets() instead."

How can we cause interesting behavior? We can change the return address of the function. A stack smashing attack or buffer overrun attack is when we convince the program to read more characters than the buffer contains. In this way, we can overwrite the return address.

There are multiple ways to figure out how to overwrite the return address. We can try to dissect

assembly code to do this. Or we can look at the memory starting at where the buffer begins. When main begins, esp points to the stack address. Let's look at the disassembly of main, and set breakpoint in gdb just before return. We can see that hte buffer actually has 112 characters.

We can create psol.txt, which has 112 y's and 4 \0 characters. When running program with gdb, there is no interesting behavior. We expected a segmentation fault on return, where the program reads 112 characters and then attempt to return, but compiler terminated the program without running. The program terminates before the breakpoint (just before the return) is reached.

This is different than when compiler knows the code will always result in undefined behavior, and replaces it. In stacksmash, undefined behavior is dynamic, and depends on the input, so it should be harder for the compiler to deal with. How is the compiler finding the error?

Run backtrace: gets_chk gets called, which checks the initialized buffer size for buffer overflow. The compiler terminates the program because this function catches the error. How is the compiler finding this buffer size, given that main does not have an argument for it?

GCC creates the size. The actual function call to gets() calls _bos(_str) which checks for buffer overflow. This type of innovation has made the language C much safer and more secure than people originally thought it would be. Ultimately, when coding, we should always check for size arguments to prevent buffer overflows.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int read_line(char* buffer) {
    if (gets(buffer))
        return 1;
    else
        return 0;
}

int main() {
    char buffer[100];
    if (read_line(buffer))
        printf("Read %d character(s)\n", strlen(buffer));
    else
        printf("Read nothing\n");
}
```

stacksmashf.c

stacksmashf.c: We can hide the size from GCC by giving the argument as a pointer to the buffer. We can run gdb stacksmashf, and we can see that the assembly calls the actual gets() function. However, running this with psol.txt (112 y's and 4 \0 characters) again results in "stacksmash detected"

When we examine the assembly, we can see that after pushing ebp and edi and dealing with correct alignment, a canary is added to the stack. At the end of the program, it checks the value pointed to by the stack pointer (just before possibly returning) and compares it to the value with the original value held in the canary. If the values are different, the program calls stack _chk _fail, otherwise returns normally.

This is getting into the next unit of the course, operating system programming. The gs is actually not on the stack, but the thread local storage area. But the program behind the scenes places the canary on the stack. This is very similar to our first problem set, where we check for invalid writes by checking past the allocated memory. The canary has an unexpected value and is pretty random, so checking if the value is changed is a good way to check for buffer overflow.

The compiler inserts a canary when it thinks that there might be a buffer overflow problem, which is usually when a character array is created.

We can break the program by adding unsafe flags, which turns off the protection features of the compiler. This breaks the program, and you can insert more instructions in the buffer and create any behavior wanted.