CS 33 (Week 4)

Section 1G, Spring 2015
Professor Eggert (TA: Eric Kim)
v1.0

Announcements

- Midterm 1 was yesterday.
  - It's over! Don't stress out too much.
  - We'll go over the midterm next week
- Homework 3 due Monday
- Lab 2 is out!
  - Due on Thursday (soon!)
  - Start early

This Week

- Matrices in x86
- Structs
- Unions
- Alignment
- Pointers
- Bounds Checking/Exploits

Matrices in x86 (nested arrays)

- int A[2][3]
  - A matrix with 2 rows, 3 columns
- C style: Row-Major order
  - Row-Major: In memory, matrix is laid out row-by-row
  - Column-Major: Matrix is laid out column-by-column

1	2	3
4	5	6

```
[ 1 2 3 4 5 6 ]
```

A in **Row-Major** format
(this is what C does!)

```
[ 1 4 2 5 3 6]
```

A in **Column-Major** format

Matrix: Row-Major

- How to access element at `int A[i][j]`?
  - Recall: A is a 2x3 matrix.
- In C, using pointer arithmetic:

```
int val = *(A + 3*i + j)
```

Matrix: Row-Major

- How to access element at `int A[i][j]`?
  - Recall: A is a 2x3 matrix.
- In x86 (`%eax = &A, %ebx = i, %ecx = j`)

```
leal (%ebx,%ebx,2), %ebx   # 3*i
imull $4, %ebx             # 4*3*i (int is 4 bytes)
addl %ebx, %eax            # A + 4*3*i
leal (%eax, %ecx, 4) %eax  # A + 4*3*i + 4*j
movl (%eax) %eax
```

Challenge: Do the same, but in fewer instructions!

Recall: C Pointers

- Why does x86 multiply by 4, but C code does not?
  - C pointers remember data type, ie how large each element is!


```
int *p;
*(p+1);  // This goes 4 bytes forward!
```

- To x86, bytes are bytes. Compiler must keep track of data sizes.

Practice Problem 3.37

Consider the following source code, where *M* and *N* are constants declared with `#define`:

```
int mat1[M][N];
int mat2[N][M];

int sum_element(int i, int j) {
    return mat1[i][j] + mat2[j][i];
}
```

In compiling this program, GCC generates the following assembly code:

```
  i at %ebp+8, j at %ebp+12
    movl    8(%ebp), %ecx
    movl    12(%ebp), %edx
    leal    0(,%ecx,8), %eax
    subl    %ecx, %eax
    addl    %edx, %eax
    leal    (%edx,%edx,4), %edx
    addl    %ecx, %edx
    movl    mat1(,%eax,4), %eax
    addl    mat2(,%edx,4), %eax
```

Use your reverse engineering skills to determine the values of *M* and *N* based on this assembly code.

Solution to Problem 3.37 (page 236)

This problem requires you to work through the scaling operations to determine the address computations, and to apply Equation 3.1 for row-major indexing. The first step is to annotate the assembly code to determine how the address references are computed:

```
movl    8(%ebp), %ecx           Get i
movl    12(%ebp), %edx          Get j
leal    0(,%ecx,8), %eax        8*i
subl    %ecx, %eax              8*i-i = 7*i
addl    %edx, %eax              7*i+j
leal    (%edx,%edx,4), %edx     5*j
addl    %ecx, %edx              5*j+i
movl    mat1(,%eax,4), %eax     mat1[7*i+j]
addl    mat2(,%edx,4), %eax     mat2[5*j+i]
```

We can see that the reference to matrix `mat1` is at byte offset $4(7i + j)$, while the reference to matrix `mat2` is at byte offset $4(5j + i)$. From this, we can determine that `mat1` has 7 columns, while `mat2` has 5, giving $M = 5$ and $N = 7$.

Structs

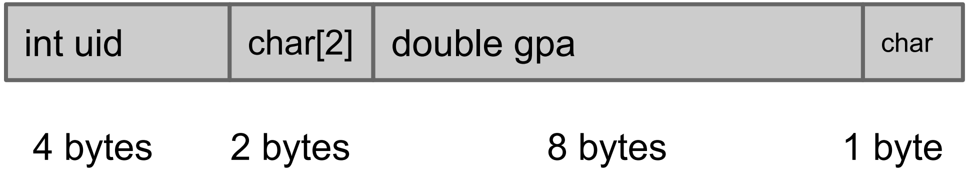
- How are structs laid out in memory?
- In x86, how to access struct fields?

```
struct student_record {
    int uid;              // 000000404
    char initials[2];     // EK
    double gpa;           // 0.42 (Ouch)
    char is_graduated;    // 0: No 1: Yes
}
```

Structs

- Fields are placed in memory contiguously
  - Always contiguous! C is not allowed to reorder struct fields.

```
struct student_record {
    int uid;
    char initials[2];
    double gpa;
    char is_graduated;
}
```



We'll talk about struct alignment in a bit

Unions

- Hacky way to save space in structs (IMO)

```
union node {
    struct {
        union node* left;
        union node* right;
    } internal;
    double data;
}
```

Suppose node represents a binary tree, with the following structure:
- If the node is a leaf node, then it stores a numerical data.
- Otherwise, it stores two pointers to its left/right children.

Note: A node can't have both data *and* pointers to children! In other words, only leaf nodes store data.

Unions

- Two equivalent-ish ways

```
union node {
    struct {
        union node* left;
        union node* right;
    } internal;
    double data;
}
```

```
struct node {
    struct {
        struct node* left;
        struct node* right;
    } internal;
    double data;
}
```

Difference: `union node` uses 8 bytes, but `struct node` uses 16 bytes!

Unions

- **Warning**: Consider the following code.

```
union node my_node = get_some_node();
```

Is `my_node` an internal node? Or is it a leaf node?

We don't know!

Need to use context to find out which "flavor" of node `my_node` is.

Unions

- **Warning**: Consider the following code.

```
union node my_node = get_some_node();
printf("Value: %f\n", my_node.data);
```

Here, `my_node` turns out to be the leaf-node variant of `union node`.

Unions

- **Warning**: Consider the following code.

```
union node my_node = get_some_node();
print_tree(my_node.internal.left);
```

Here, `my_node` turns out to be the internal-node variant of `union node`.

Unions

- **Tip**: When reverse engineering x86 code for union code, you'll need to figure out the correct union "flavor" of each variable. (HW3)
  - In HW3, pointer dereferences help disambiguate things.

Review: Little-endian vs Big-endian

```
long long bit2ll(unsigned int word0, unsigned int word1) {
  union {
    long long d;
    unsigned u[2];
  } temp;
  temp.u[0] = word0;
  temp.u[1] = word1;
  return temp.d;
}
```

```
unsigned int x = 0x0ABCDEF0
unsigned int y = 0xFACEB00F
long long val = bit2ll(x,y);
```

What is val if:

(1) The machine is little-endian?

(2) The machine is big-endian?

For both cases, how is val laid out in memory?

Review: Little-endian vs Big-endian

```
long long bit2ll(unsigned int word0, unsigned int word1) {
  union {
    long long d;
    unsigned u[2];
  } temp;
  temp.u[0] = word0;
  temp.u[1] = word1;
  return temp.d;
}
```

```
unsigned int x = 0x0ABCDEF0
unsigned int y = 0xFACEB00F
long long val = bit2ll(x,y);
```

Answer:

Little-endian: `0xFACEB00F 0ABCDEF0`

Big-endian: `0x0ABCDEF0 FACEB00F`

Review: Little-endian vs Big-endian

```
long long bit2ll(unsigned int word0, unsigned int word1) {
  union {
    long long d;
    unsigned u[2];
  } temp;
  temp.u[0] = word0;
  temp.u[1] = word1;
  return temp.d;
}
```

```
unsigned int x = 0x0ABCDEF0
unsigned int y = 0xFACEB00F
long long val = bit2ll(x,y);
```

Addresses grow left->right

How is val laid out in memory?

Little-endian:

0xf0	0xde	0xbc	0x0a	0x0f	0xb0	0xce	0xfa

Big-endian:

0x0a	0xbc	0xde	0xf0	0xfa	0xce	0xb0	0x0f

Alignment

- x86 convention: total stack space used by a function must be a multiple of 16 bytes
- Arch-dependent rules on data-alignment
  - Linux: 2-byte data types (ie short) must have an addr that is a multiple of 2.
    - Larger data types (ie int, double) must have an addr that is a multiple of 4
  - Windows: ANY data type of K bytes must have an addr that is a multiple of K.
  - Which is faster?

x86: .align directive

.align N tells compiler to make subsequent data have an addr that is a multiple of N.

```
.rodata
.string "ab\0" # starts at addr 0
.string "hi" # starts at addr 3
```

```
.rodata
.string "ab\0" # starts at addr 0
.align 4
.string "hi" # starts at addr 4
```

```
.rodata
.string "a" # starts at addr 0
.align 4
.string "hi" # starts at addr 4
```

Struct alignment

- Might need to add padding in between fields to satisfy alignment
- For struct arrays, might need to add padding at *end* of each struct to satisfy alignment
- See Chapter 3.9.3 for more details

C Pointers

- One warning: casting priority
  - Suppose p is a pointer to a char

What is the memory offset of the following expressions?

`(int*) p+7`	4*7 = 28 bytes (p is first cast as int*, then incremented)
`(int*) (p+7)`	7 bytes (p is still treated as a char* ptr)

C Function Pointers

```
int (*f)(int,char)
```

Means: f is a pointer to a function that takes two arguments (int, char), and returns an int.

```
int (*f)(int, char) = &my_fn;
```

What gets printed out?

47

```
int add_two(x) {
  return x + 2;
}
int add_three(x) {
  return x + 3;
}
int compose(int val, int (*f)(int), int(*g)(int)) {
  return f(g(val));
}
int main(int argc, char** argv) {
  int (*fnptr1)(int) = &add_two;
  int (*fnptr2)(int) = &add_three;
  int s = compose(42, fnptr1,fnptr2);
  printf("s is: %d\n", s);
  return 1;
}
```

Warning: Function Ptr vs Prototype

```
(int*) f(char*,int)
```

This is a **function prototype**, declaring a function f that takes 2 args (char*,int), and returns an int*.

```
int* (*f)(char*,int)
```

f is a **pointer** to a function that takes two args (char*,int), and returns an int*.

gdb - Debugger

For Lab 2, you may find these lines useful:

```
$ gdb --args emacs -batch -eval '(print (* 37
-26))'
(gdb) set disassemble-next-line on
(gdb) break main
(gdb) run
(gdb) stepi
(gdb) info registers
```

gdb - Debugger

Also:

```
(gdb) disassemble \m main
```

Bounds Checking

Scenario: A function declares a local char buffer with a **fixed** size, and allows user to input characters from the keyboard into the buffer.

BUT! The function doesn't check to see if the user typed past the end of the buffer.

Bounds Checking

"Best" case: Program crashes

  What scenario could result in a crash?

Worst case: Attacker gains control of your machine!

Bounds Checking

```c
/* Sample implementation of library function gets() */
char *gets(char *s)
{
    int c;
    char *dest = s;
    int gotchar = 0; /* Has at least one character been read? */
    while ((c = getchar()) != '\n' && c != EOF) {
        *dest++ = c; /* No bounds checking! */
        gotchar = 1;
    }
    *dest++ = '\0';  /* Terminate string */
    if (c == EOF && !gotchar)
        return NULL; /* End of file or error */
    return s;
}
```

Stack Smashing

When a function writes past the end of a buffer (ie array), this is called a **buffer overflow**.

In the Computer Security community, this is also known as **Stack Smashing**, especially when a buffer overflow is used for malicious purposes.

Stack Smashing (Reading)

If you're curious, Google "Stack Smashing for Fun and Profit"

Purely optional, ie if you're bored and somehow have free time :P

Stack Smashing

How to exploit a buffer overflow?

Recall: Goals of attacker are typically:

1. Read sensitive data (passwords, etc)
2. Disrupt service (ie DDoS)
3. Execute code on machine

Stack Smashing

Trick 1: Overwrite caller's saved eip on stack, and write the address of code that *we* want to execute!

Super Neat Trick: Write our malicious code into the array we are overflowing, then set caller's saved eip to start of our code!

Stack Smash Defenses

- Only allow OS to execute code from read-only section of memory
  - Called "Data Execution Prevention" (DEP)
  - Known workarounds
    - store code on heap
    - Call syscalls to disable DEP
    - make a series of legit function/library calls to achieve hack ("return-to-libc")

Address Space Layout Randomization (ASLR)

- Several exploits require knowing the precise address of locations on the stack (ie the address of the caller's saved eip).
- Defense: randomize the stack
  - Start stack at some random offset
  - Defeats attacks that assume a specific memory layout

NOP-Sleds

- Scenario: we are injecting malicious code into a buffer.
    - Goal: Need to put the address of the first malicious instruction into the caller's saved eip
    - With ASLR, this is much more difficult. We could do a brute-force search, but search space is large.

...	...	0x08	0x02	0xf3	0xff	0x33	0x74	0x11	0x00

Start of my malicious code.

How to guess this address?

NOP-Sleds

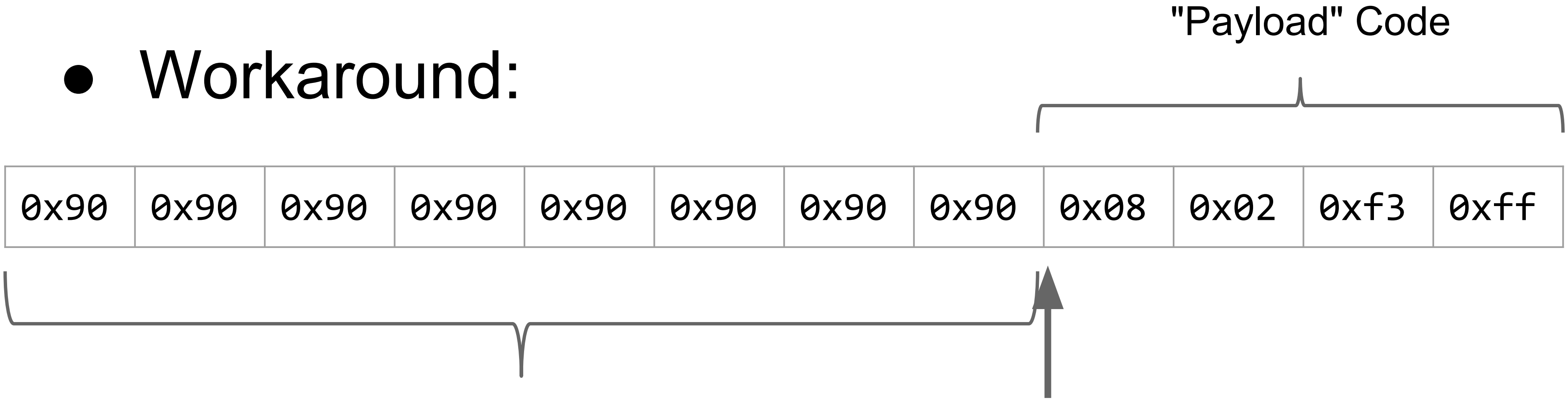
- Workaround:

...	...	0x08	0x02	0xf3	0xff	0x33	0x74	0x11	0x00

Instead of having to guess this address (hard)

NOP-Sleds

- Workaround:



Instead, guess *any* of these addresses!

If we hit any of the NO-OP instructions, then the processor will "slide" from left to right until it reaches our malicious code.

Canaries

- <u>Idea</u>: Instead of trying to **stop** buffer overflows, instead try to **detect** them.
  - If detect, then halt the program.

Canaries

- Compiler adds special value (canary) to stack at the end of a local buffer.
- When function is returning, check canary value.
  - If the canary value changed, then a buffer overflow must have happened.
    - Issue a "Stack Overflow Exception"
  - Else, return to caller as normal

Canaries

- By halting before returning, we prevent the eip being set to an address of the attacker's choosing.
- Can you think of ways to bypass a canary?
  - Assume that it isn't feasible to try to guess the canary value.

Computer Security

- Studying ways to attack vulnerable systems (and defend against malicious attackers)
  - Web security is ***hugely*** important these days
    - Banks, customer data, SSN's, etc.
- Very active field of research
  - Web security, mobile security, network security, ...

Computer Security (cont.)

- Cryptography
  - Using math to design robust, secure cryptosystems
  - Ie "one-way" functions: functions that are simple to evaluate in one direction, but computationally infeasible to invert.
  - Involves a crazy amount of number theory
    - Ie properties of prime numbers

Computer Security (cont.)

- If this stuff excites you, consider taking a few courses in security
  - CS 136: Introduction to Computer Security

Looking Forward...

Today: 4/24

Lab 2: Due 4/30

HW 4: Due 5/08

Lab 3: Due 5/13

Midterm 2: 5/14 ← **~3 weeks from now**

Tips

- Study/prepare early
- Take advantage of resources
  - Prof/TA office hours, Piazza, UPE/ACM tutoring
  - Can even e-mail me/other TA's to go over things if office hours isn't enough (depending on our schedules, we can help)
- Read textbook! ***Very*** helpful.
  - We follow the textbook pretty closely
  - Doing the practice exercises helps consolidate things
- You can do it!