Week 10. Data Structures, Integral Data Types, Integer Overflow, Endianness

Data Structures

Arrays

- Arrays a kind of data structure that can store a fixed-size sequential collection of elements of the same type.
  - AKA "homogeneous container"
- Consider this code chunk:

```
#include <stdio.h>
int main() {
    int a[20];
    int i;
    for (i=0; i<20; i++)
        a[i]=i*2;
    for (i=0; i<20; i++)
        printf ("a[%d]=%d\n", i, a[i]);
return 0; };
```

  - Review with godbolt.org or radare2, walk through the population and printing of the array. Should be straightforward this far into the semester.
  - The variable `a` is type `int*` (pointer to an int). When passing an array to another function, you can technically pass a pointer to that array, but typically you are passing the pointer to the first element in that array, the rest to be calculated in an obvious way.
    - This can be demonstrated by stepping through a program which passes an array with radare2 debug mode (and visual mode to view the stack).
  - If you index this pointer as `a[2]`, 2 is just to be added to the pointer and the element placed there (to which calculated pointer is pointing) is to be returned.
  - In C/C++, a string is actually a `const char[]` (char array) and can be indexed the same way any other array can. `"string"[2]` would return `'r'`
    - This can also easily be demonstrated
  - Buffer Overflows revisited:
    - Less fun / malicious overflows

- AKA. **Segmentation Fault**
- Consider and compile the following code block:

```
#include <stdio.h>
int main() {
        int a[20];
        int i;
        for (i=0; i<30; i++)
                a[i]=i;
return 0;
};
```

- Executing this code will result in a seg fault
- GDB and/or Radare2 will help identify when and where segfault occurs. Debugging can be useful!

- With all of this in mind….
  - Why is this impossible?

```
void f(int size)
{
    int a[size];
...
};
```

  - That's just because the compiler must know the exact array size to allocate space for it in the local stack layout on at the compiling stage.
  - If you need an array of arbitrary size, allocate it by using malloc(), then access the allocated memory block as an array of variables of the type you need.
    - Thats what you get for not having garbage collectors
  - **Conclusion**
    - An array is a pack of values in memory located adjacently.
    - It's true for any element type, including structures.
    - Access to a specific array element is just a calculation of its address.
    - A pointer to an array and address of a first element—is the same thing. This is why ptr[0] and *ptr expressions are equivalent in C/C++.

Structures

G Leaden - Week 10 **Reverse Engineering**

- User defined data type available in C that allows the user to combine data items of different type.
  - AKA “heterogeneous container”
  - Consider the tm struct in UNIX:

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

void main() {
    struct tm t;
    time_t unix_time;
    unix_time=time(NULL);
    localtime_r (&unix_time, &t);
    printf ("Year: %d\n", t.tm_year+1900);
    printf ("Month: %d\n", t.tm_mon);
    printf ("Day: %d\n", t.tm_mday);
    printf ("Hour: %d\n", t.tm_hour);
    printf ("Minutes: %d\n", t.tm_min);
    printf ("Seconds: %d\n", t.tm_sec);
};
```

  - Once this is compiled
    - Run through debug mode in radare2
      - Set breakpoints at sys.printf
      - Continue until breakpoint, then view stack in visual mode
      - You can see the struct in the stack, but we dont know how the tm struct is defined in time.h
        - LETS FIND OUT

```c
struct tm {
  int    tm_sec;
  int    tm_min;
  int    tm_hour;
  int    tm_mday;
  int    tm_mon;
  int    tm_year;
  int    tm_wday;
  int    tm_yday;
```

```
  int   tm_isdst;
};
```

- Int is 32 bytes
- Stack should look something like this (translated)

Hexadecimal number	decimal number	field name
0x00000025	37	tm_sec
0x0000000a	10	tm_min
0x00000012	18	tm_hour
0x00000002	2	tm_mday
0x00000005	5	tm_mon
0x00000072	114	tm_year
0x00000001	1	tm_wday
0x00000098	152	tm_yday
0x00000001	1	tm_isdst

- Although this struct contains ONLY int values, it follows generally that the only difference would be the byte size in the stack depending on type.
- Fields Packing
  - Consider this code block:

```
#include <stdio.h>
struct s {
     char a;
     int b;
     char c;
     int d;
};
void f(struct s s)
{
     printf ("a=%d; b=%d; c=%d; d=%d\n", s.a, s.b, s.c,
s.d);
};
int main() {
     struct s tmp;
     tmp.a=1;
     tmp.b=2;
     tmp.c=3;
     tmp.d=4;
     f(tmp);
};
```

- We have 2 char fields (1 byte each) and 2 int fields (4 bytes each)
- When compiled, we see that space is reserved for the struct is 16 bytes. Why?
  - It's easier for the CPU to access memory at aligned addresses and to cache data from it.
- Other thing to note, the struct is copied and placed somewhere else, because the compiler does not know if the f function will modify the struct or not.

Integral Data Types

- Bit
  - Atomic
  - Obvious usage for bits are boolean values: 0 for false and 1 for true.
  - In C/C++ environment, 0 is for false and any non-zero value is for true.
- Nibble
  - 4 bits
  - Binary-coded decimal
    - Almost never seen in practice, except for magic numbers (easy to unpack and pack a BCD)
- Byte
  - Byte is primarily used for character storage. 8-bit bytes were not common as today. Punched tapes for teletypes had 5 and 6 possible holes, this is 5 or 6 bits for byte.
  - char one byte character (PDP-11, IBM360: 8 bits; H6070: 9 bits)
    - C language manual
- Word
  - Data type for general purpose registers.
    - Bytes are practical for characters, but impractical for other arithmetical calculations.

    - Hence, many CPUs have GPRs with width of 16, 32 or 64 bits.
    - There was a time when hard disks and RAM modules were marketed as having n kilo-words instead of b kilobytes/megabytes.
  - Int is typically stored as a Word
  - 16-bit C/C++ environment on PDP-11 and MS-DOS has long data type with width of 32 bits, perhaps, they meant long word or long int?
  - 32-bit C/C++ environment has long long data type with width of 64 bits.
  - GDB has the following terminology: halfword for 16-bit, word for 32-bit and giant word for 64-bit.
  - The *word* Word is very ambiguous.
- Signed and Unsigned Numbers
  - Some may argue, why unsigned data types exist at first place, since any unsigned number can be rep- resented as signed. Yes, but absence of sign bit in a value extends its range twice. Hence, signed byte has range of -128..127, and unsigned one: 0..255.
    - Another benefit of using unsigned data types is self- documenting: you define a variable which can't be assigned to negative values.
    - Unsigned data types are absent in Java, for which it's criticized. It's hard to implement cryptographical algorithms using boolean operations over signed data types.
    - Values like 0xFFFFFFFF (-1) are used often, mostly as error codes.
  - Computers typically utilize two's compliment to represent signed numbers.
  - Numbers can be signed or unsigned.

  - C/C++ signed types:
    - int64_t (-9,223,372,036,854,775,808 .. 9,223,372,036,854,775,807) (- 9.2.. 9.2 quintillions) or 0x8000000000000000..0x7FFFFFFFFFFFFFFF),
    - int (-2,147,483,648..2,147,483,647 (- 2.15.. 2.15Gb) or 0x80000000..0x7FFFFFFF),
    - char (-128..127 or 0x80..0x7F),

- ssize_t. Unsigned:
- uint64_t (0..18,446,744,073,709,551,615 (18 quintillions) or 0..0xFFFFFFFFFFFFFFFF), – unsigned int (0..4,294,967,295 (4.3Gb) or 0..0xFFFFFFFF),
- unsigned char (0..255 or 0..0xFF),
- size_t.

- Signed types have the sign in the most significant bit: 1 means “minus”, 0 means “plus”.
- Promoting to a larger data types is simple: 1.28.5 on page 405.
- Negation is simple: just invert all bits and add 1.
  We can keep in mind that a number of inverse sign is located on the opposite side at the same proximity from zero. The addition of one is needed because zero is present in the middle.
- The addition and subtraction operations work well for both signed and unsigned values. But for multiplication and division operations, x86 has different instructions: IDIV/IMUL for signed and DIV/MUL for unsigned.

Integer Overflow

- First, take a look at this implementation of itoa() function from [Brian W. Kernighan, Dennis M. Ritchie, The C Programming Language, 2ed, (1988)]:
  - It has a subtle bug. Try and identify it.

```
void itoa(int n, char s[])
{
        int i, sign;
        if ((sign = n) < 0) /* record sign */
                n = -n; /* make n positive */
        i = 0;
        do { /* generate digits in reverse order */
                s[i++] = n % 10 + '0'; /* get next digit */
        } while ((n /= 10) > 0); /* delete it */
        if (sign < 0)
                s[i++] = '-';
        s[i] = '\0';
strrev(s);
}
```

- In a two's complement number representation, our version of itoa does not handle the largest negative number, that is, the value of n equal to $-(2^{wordsize-1})$. Explain why not. Modify it to print that value correctly, regardless of the machine on which it runs.
  - The answer is: the function cannot process largest negative number (INT_MIN or 0x80000000 or -2147483648) correctly.
  - How to change sign? Invert all bits and add 1. If to invert all bits in INT_MIN value (0x80000000), this is 0x7fffffff. Add 1 and this is 0x80000000 again. So changing sign has no effect. This is an important artifact of two's complement system.

Endianness

- Endianness refers to the sequential order in which bytes are arranged into larger numerical values when stored in memory or when transmitted over digital links.
- Two conflicting and incompatible formats are in common use:
- **Big-endian**
  - Big end (most significant bit)
  - The 0x12345678 value is represented in memory as:

address in memory	byte value
+0	0x12
+1	0x34
+2	0x56
+3	0x78

    - Some current big-endian architectures include the IBM z/Architecture, Freescale ColdFire (which is Motorola 68000 series-based), Xilinx MicroBlaze, Atmel AVR32.
- **Little-endian**
  - Little end (least significant bit)
  - The 0x12345678 value is represented in memory as:

address in memory	byte value
+0	0x78
+1	0x56
+2	0x34
+3	0x12

- The Intel x86 and also AMD64 / x86-64 series of processors use the little-endian format, and for this reason, it is also known in the industry as the "Intel convention".

- **Bi-endian**
  - Capable of computing or passing data in either endian format.
  - Some architectures (including ARM versions 3 and above, PowerPC, Alpha, SPARC V9, MIPS, PA-RISC, SuperH SH-4 and IA-64) feature a setting which allows for switchable endianness in data fetches and stores, instruction fetches, or both.
  - This feature can improve performance or simplify the logic of networking devices and software.