CS7038 - Malware Analysis - Wk08
Analysis of C Data Types

Coleman Kane
kaneca@mail.uc.edu

February 28, 2017

Data Structures Explained

Computing systems are built around data, and in particular, `structured data`. This has the feature of providing organization to the data contained within a structure, as some computer code is expected to be able to accept the contents of the structured data, and then perform operations, configure itself, or otherwise alter program execution based upon this.

In this lecture, we will take some time to analyze data structures and data representations which will be useful when reverse engineering malicious code. It will be common to have on-disk and in-memory data structures, that aren't executable code, but that need to be analyzed in order to discern what executable code is doing.

Numeric Representation - Endianness

A core component of data representation is how an architecture represents numeric and array data.

For numeric data, two common methods are used:

- Litte-Endian - least-significant digits/bytes first
- Big-Endian - most-significant digits/bytes first

These have to do with numeric data handling when multiple bytes are necessary in storing numeric data. Namely, in what order should the data be stored? In the case of written numbers using the prevalent arabic numberic system, the number “**1024**” is stored in what’s called *Big Endian* format.

In the above example, each digit is a number that can be one of 10 values, and the first digit we read represents the largest element of the power-series constructing the number $(1 \times 10^3 + 0 \times 10^2 + 2 \times 10^1 + 4 \times 10^0)$.

Endianness

For architectural reasons, in x86 including x86-64 architectures, it is common to instead use little endian. Consider adding two really large numbers togeher, typically you start with the right-hand digit of a number and work the addition operation incrementally toward the left.

The value `27452` is a decimal number that can be written in hexadecimal as `0x6b3c`. This number consists of two bytes: `6b 3c`. It is encoded using the following sequence of bytes in each ordering format:

- Little-endian: `3c 6b`
- Big-endian: `6b 3c`

Neither is arguably better than the other. These days both are commonly used, and their application is primarily driven by the underlying architecture. It can be argued, though, that little-endian is the more prevalent of the two in computers.

Signedness

Another interesting feature of numeric representation is signed-ness. This comes into play when dealing with negative numbers or really large values.

A 16-bit integer has enough unique configuration to store `65536` distinct values. Frequently, a signed 16-bit integer is one that is defined to hold any one of the values `-32768` through `32767`. An unsigned 16-bit integer is defined to hold any value from `0` through `65535` (so, as many positive integers as possible, starting with 1, and including 0).

The negative numbers in this scenario are those which have the upper-most bit set to 1, however rather than merely encoding positive numbers with this bit set, these are actually stored in what's called *Two's Complement* form:
`http://www.tfinley.net/notes/cps104/twoscomp.html`
Consider the case where you have `0` and want to subtract `1` from it. Using normal integers, and ignoring sign, you would get a value with all bits set to 1. Two's complement defines this value to be -1, rather than the more humanly-intuitive "lowest negative number" of `-32768`.

Array and Matrix Forms

In many cases, it also becomes common to find data organized in arrays and matrices in memory. In the cases of explicit arrays and matrices, the data is stored in one contiguous chunk in memory:

```
long array1[12]; /* 12*4 = 48 bytes buffer.  */
long array1[12][3]; /* 12*4*3 = 144 bytes buffer.  */
```

However, sometimes arrays and matrices are created from pointers. This may lead to a single contiguous array of pointers, but each "row" may not be adjacent to its neighbor:

```
int *matrix[3];

for(int c = 0; c < 3; c++) {
  matrix[c] = malloc(sizeof(int}*12);
}
```

In the above, we create a code-addressable 3×12 matrix to store data, but each of the 3 twelve byte rows is not guaranteed to be adjacent to one another in memory. Furthermore, the `matrix` storage really just stores indirect memory pointers rather than pointing to the first piece of data.

Struct and Class Forms

In C the `struct` and `class` keywords are used to define complex user-defined types. These are frequently collections of atomic data, and can include arrays, pointers, and even other structs and classes. Atomic data in these is stored semi-contiguously. Where some efficiency can be gained through padding, padding between elements may or may not be added.

```
struct test_struct_def {
  unsigned int ip_address;
  char modifier;
  unsigned short port;
};
```

After analyzing the above data in memory, it becomes apparent that all of the values are adjacent to one another in memory, however, there is a one-byte gap between `modifier` and `port`. In C/C++, this behavior can be controlled using (where `n` is an int constant):

`#pragma pack(n)`