

*CS7038 - Malware Analysis - Wk02.2*  
**Container Model of Data Files**

Coleman Kane  
kaneca@mail.uc.edu

January 19, 2017

# Data Organization Models

Data organization in a computing system is frequently standardized, so that applications can share data interchangeably.

You're probably familiar with some end-user functional standards, such as:

- Filesystems
- Files
- Folders
- Programs
- Archives (ZIP files, etc.)

## Files as Containers

You're already somewhat familiar with the idea of files as containers.

One easy example of this is ZIP files.

- The ZIP *container* is a single file
- The *container* can be conceived as being a bucket that has other files stored within it
- The *container* also contains another data object that describes how all of these files are organized within the ZIP file

In many cases, I like to call these *streams* in an effort to separate the *container* concept from the OS-tied concepts of files, filesystems, folders, etc.

A more generic way to represent the concept of a raw data stream might be using the C array. For instance a 32kB stream might be:

```
byte stream[32768];
```

## Streams and Data Structures

So, a stream can point to a raw byte array. If you're familiar with the concept of *typecasting*, the following operation will look very familiar:

```
struct zip_stream *zip_data =  
    (struct zip_stream *)&stream;
```

This would then provide the user with a pointer that can decode the raw data in `stream` using the protocol described in `struct zip_stream`.

A **very generalized** data structure definition for `struct zip_stream`

might be:

```
struct zip_stream {  
    byte **streams;  
    uint stream_len;  
    char **pathnames;  
}
```

## Streams Within Streams

So, say you have decided to ZIP up a ZIP file. For instance, the first one in the archive might have the filename “inside.zip”.

Then you may have:

```
zip_data->pathnames[0] = "inside.zip";
```

Additionally, you could then provide access to the inner ZIP file:

```
struct zip_stream *inner_zip_stream =  
    (struct zip_stream*) (zip_data->streams[0]);
```

Furthermore, if you decided to store a PDF file and a DOC file in the ZIP file, you could get a structural definition of those types of files:

```
struct pdf_stream;  
struct doc_stream;
```

I like to think of this as the **container model** of streams (or “artifacts”).

## Adding Parsing to the Mix

An additional step, which had been left out, is that frequently the data is not organized in a format that maps 1:1 to a programming language data structure (we will describe *why* this is later on).

Parsing functions are intended to take encoded data and fill in a language-native data structure with the necessary contents to reconstruct and navigate the file from the application/tool.

So, a more complete example might be to define a new function:

```
struct zip_stream *parse_zip(byte *input_stream);
```

And then, you would get a pointer to the decoded data structure:

```
struct zip_stream *inner_zip_stream(  
    zip_data->streams[0]);
```

## Applying This to Malware Analysis

A significant portion of malware analysis requires performing **black box reverse engineering** of suspected malware artifacts (which you can more generally represent as *streams*).

One goal that you will frequently have to perform is attempting to decipher the *protocol*, at least in part, of a suspect data stream, so that you can explore the structure of the artifact to extract characteristics such as: look & behavior.

In short, malware analysis frequently involves building up your own *parsing function* to at least pull out **contextually significant** contents from the containing stream, to make them available for inspection and analysis.