Digital Forensics Reverse Engineering Lecture 2

PDF Files Structure and Analysis

Akbar S. Namin
Texas Tech University
Spring 2018

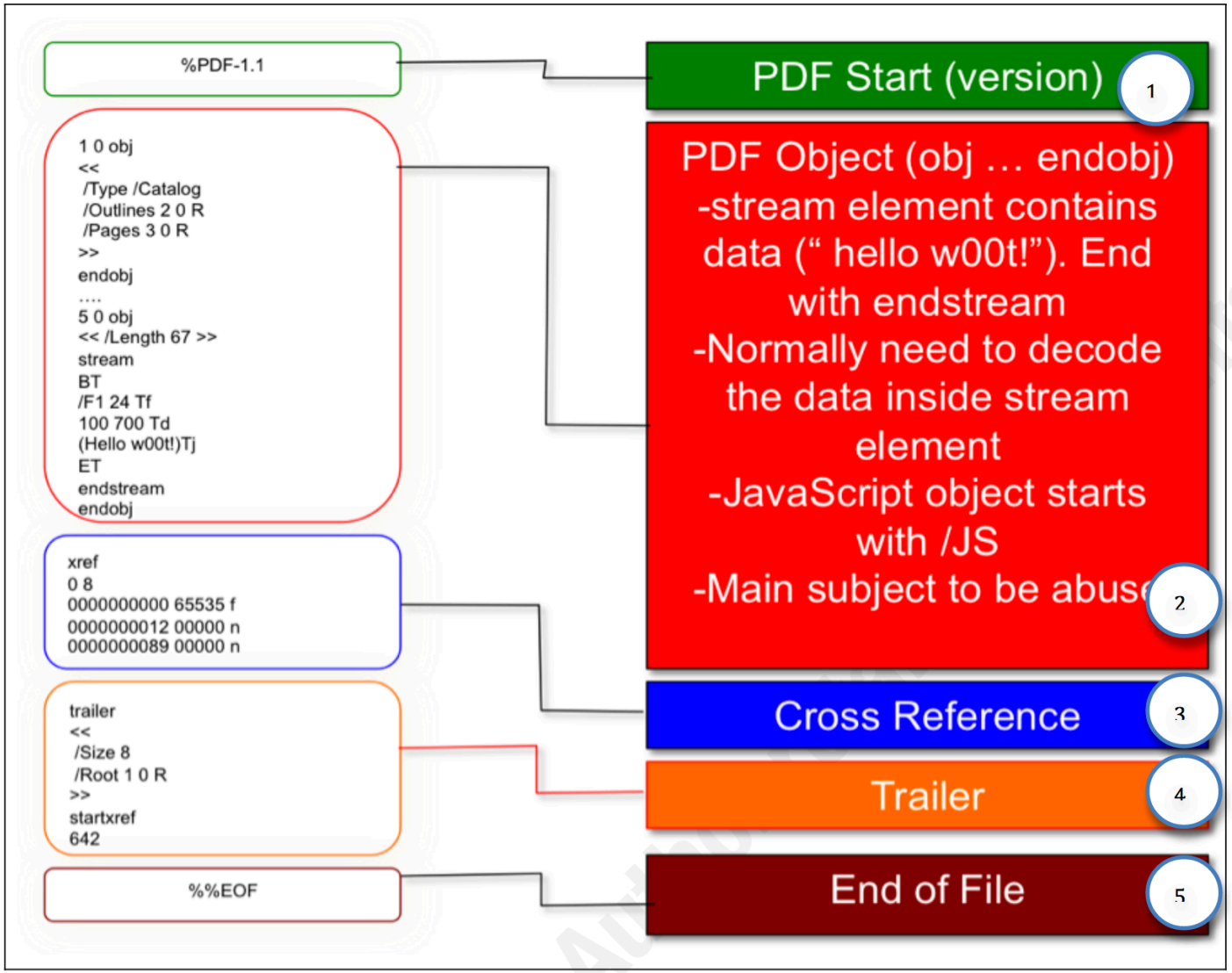
PDF Format and Structure

- Tools used
  - Text editor (e.g., vi)
  - ClamAV antivirus (http://www.clamav.net/lang/en/download/
  - Or www.virustotal.com
  - Pdftk (http://www.accesspdf.com/pdftk/
  - Patched (http://www.didierstevens.com/files/software/js-1.7.0-mod.tar.gz
  - Libemu's sctest (http://libemu.carnivore.it
  - Immunity Debugger (https://www.immunityinc.com/products-immdbg.shtml)

PDF Format and Structure

- The PDF format consists of objects, of which there are eight types:
  - Boolean values
  - Numbers
  - Strings
  - Names
  - Arrays, ordered collections of objects
  - Dictionaries, collections of objects indexed by Names
  - Streams, usually containing large amount of data
  - The Null object

PDF Format and Structure



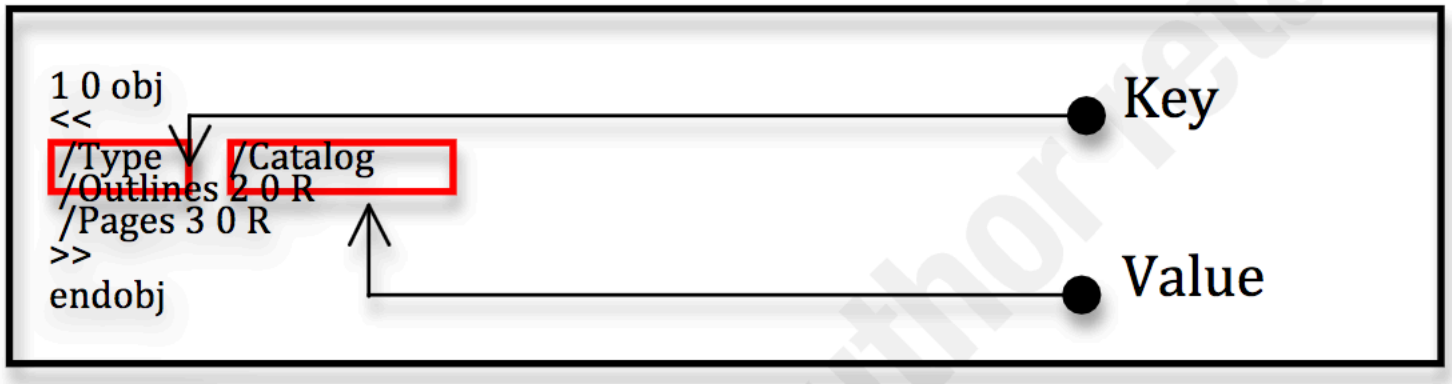
PDF Format and Structure

- Every PDF file has a header that shows the specific version of the file
- PDF files end with %%EOF
- The second element: *obj* object where malicious JavaScript code is embedded
  - The object obj starts with a reference number followed by a version number
  - Obj keyword
  - The object container
  - Endobj to indicate the end of the object

```
1 0 obj
<<
 /Type /Catalog
 /Outlines 2 0 R
 /Pages 3 0 R
>>
endobj
```

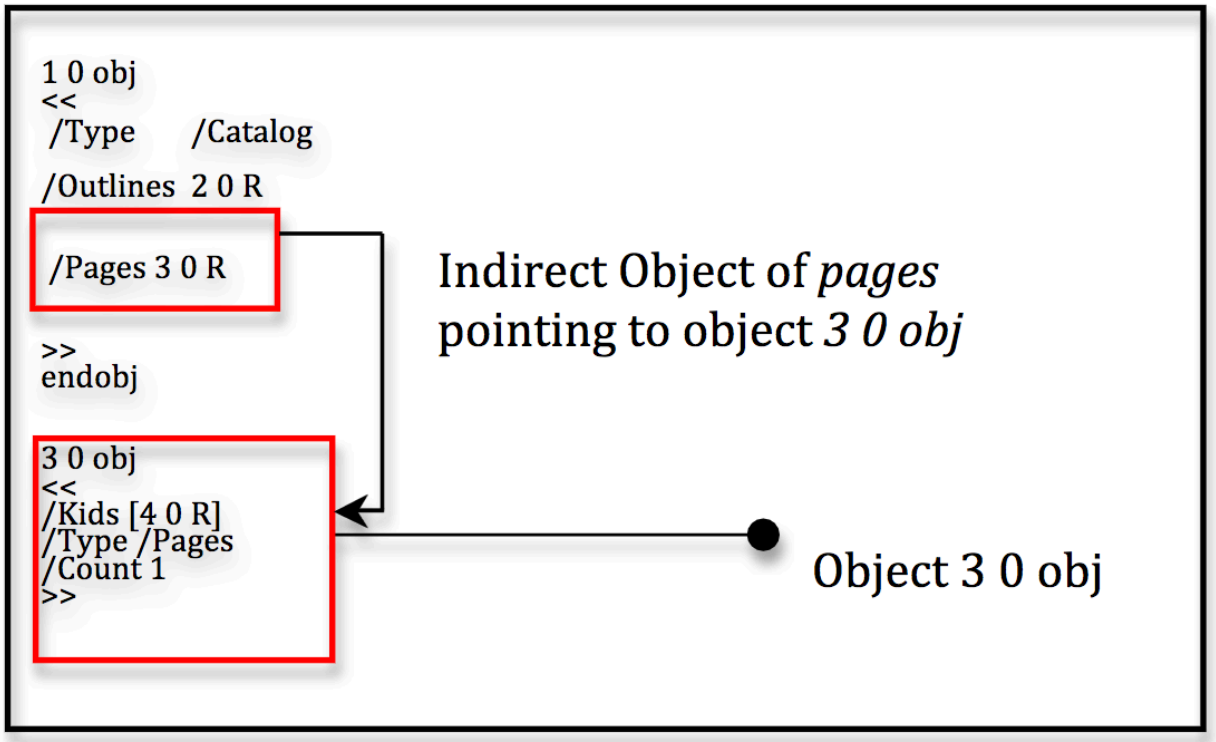
PDF Format and Structure

- The object container may consist of various objects
- The most common object is dictionary
- A dictionary is written as a sequence of key-value pairs
  - Enclosed in double angle brackets (<< … >>)
  - The first element of a dictionary is the key
  - The second element is the value



PDF Format and Structure

- Any object in a PDF file may be labeled as an indirect object
  - A unique identifier by which other objects can refer to it
  - In previous slide, the outlines key is pointing to the indirect object of 2.0



PDF Format and Structure

- Stream object
  - Like a string object, a sequence of bytes
  - Consists of a dictionary followed by zero ot more bytes bracketed between the keyword stream and endstream
  - Can be of unlimited length (string is subject to limitation)
  - Objects with large data (e.g. ,images) are represented as steams

```
5 0 obj
<< /Length 56 >>
stream
BT /F1 12 Tf 100 700 Td 15 TL (JavaScript example) Tj ET
endstream
endobj
```

PDF Format and Structure

- Stream object
  - Filters: optional entries for stream dictionary
  - Filter indicates whether the stream will be decompressed or decoded
  - The Filter's key: indicate the method of decompression or decoding for the stream
    - Usually employed by attackers to increase the difficulty and complexity of the attacks

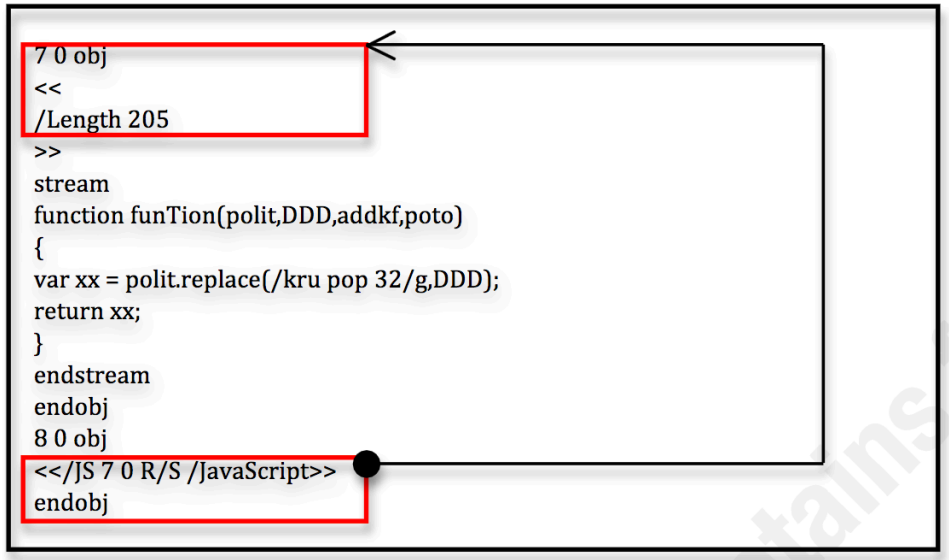
```
26 0 obj
<</Filter /FlateDecode
/Length 56
>>
stream
x<9c>ã*K,R(-N.(P°^EÓ^Z<9a>Ö\\\ ÑâäøÔ¼^T¨p<81><86>k^^JJ
P^V(X\\¬^A<91>^Fò^AÑv^Us
endstream
endobj
```

PDF Format and Structure

- Some of the filters
  - ASCII85Decode
  - ASCIIHexDecode
  - FlateDecode
  - RunLengthDecode
  - DCTDecode
  - CCITTFaxDecode
  - JBIG2Decode
  - JPXDecode

PDF Format and Structure

- JavaScript name directory
  - A common object inside PDFs
  - Starts with /JavaScript /JS java_script_code
  - Majority of malicious PDF file attacks rely on JavaScript to trigger the vulnerability
  - JavaScript used in a few forms to exploit a system
    - Used as a heap spray generator for exploitation reliability
    - Used directly or indirectly



Triggering Vulnerabilities with Heap Spray Technique

- Heap overview
- Heap exploitation
  - Heap overflows
  - Use after free
  - Heap spraying
  - Metadata corruption

The Heap

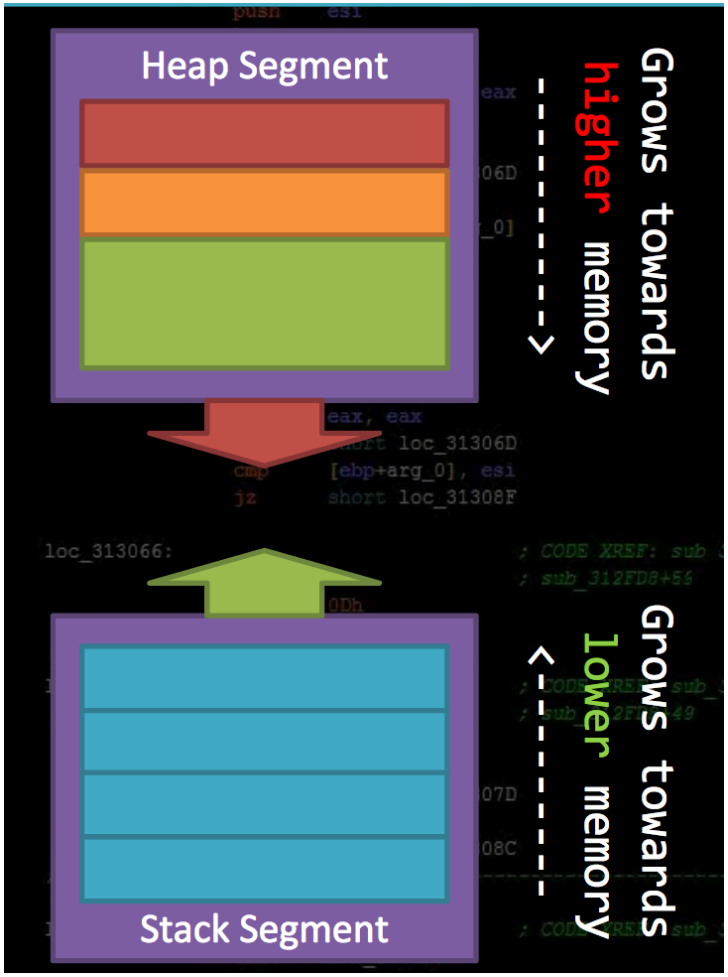
- A pool of memory used for dynamic allocations at runtime
  - malloc() grabs memory on the heap
  - free() releases memory on the heap

The Heap

- Heap
  - Dynamic memory allocations at runtime
  - Suitable for objects, big buffers, structs, persistence, and larger things
  - Slower, and manual
    - Done by the programmer
    - malloc/calloc/realloc/free
- Stack
  - Fixed memory allocations known at compile time
  - Local variables, return addresses, and function args
  - Fast and automatic
    - Done by the compiler
    - Abstract away any concept of allocation/de-allocation

The Heap

- Heap vs. stack
  - Heap grows down towards higher memory
  - Stack grows up towards lower memory



The Heap

- The syntax and semantic of malloc(0x100)
  - Allocate a 0x100 byte buffer
- How many byte are allocated for the following cases?
  - malloc(32)
  - malloc(4)
  - malloc(20)
  - malloc(0)

The Heap

- The syntax and semantic of malloca(0x100)
  - Allocate a 0x100 byte buffer
- How many byte are allocated for the following cases?
  - malloc(32)
    - 40 bytes
  - malloc(4)
    - 16 bytes
  - malloc(20)
    - 24 bytes
  - malloc(0)
    - 16 bytes

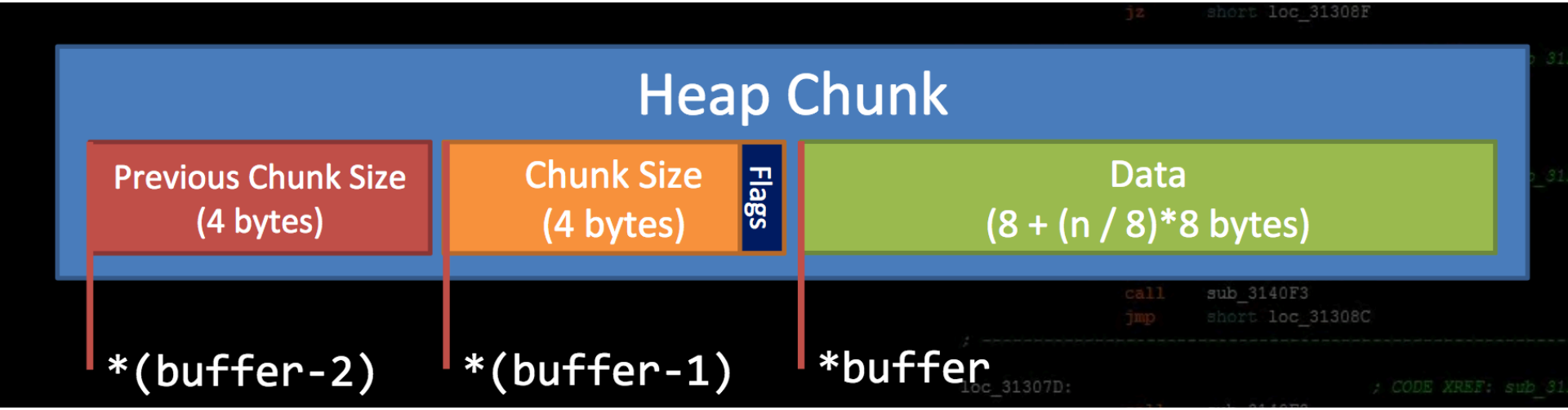
The Heap

prints distance between mallocs (size of chunk)

```
lecture@warzone:/levels/lecture/heap$ ./sizes
malloc(32) is at 0x086a6008,  40 bytes to the next pointer
malloc( 4) is at 0x086a6030,  16 bytes to the next pointer
malloc(20) is at 0x086a6040,  24 bytes to the next pointer
malloc( 0) is at 0x086a6058,  16 bytes to the next pointer
malloc(64) is at 0x086a6068,  72 bytes to the next pointer
malloc(32) is at 0x086a60b0,  40 bytes to the next pointer
malloc(32) is at 0x086a60d8,  40 bytes to the next pointer
malloc(32) is at 0x086a6100,  40 bytes to the next pointer
malloc(32) is at 0x086a6128,  40 bytes to the next pointer
lecture@warzone:/levels/lecture/heap$
```

The Heap

- Heap chunks
  - Size of previous chunk (if previous chunk is free)
  - Chunk size
    - Size of entire chunk including overhead
  - Data
    - The newly allocated memory
    - Pointer returned by malloc



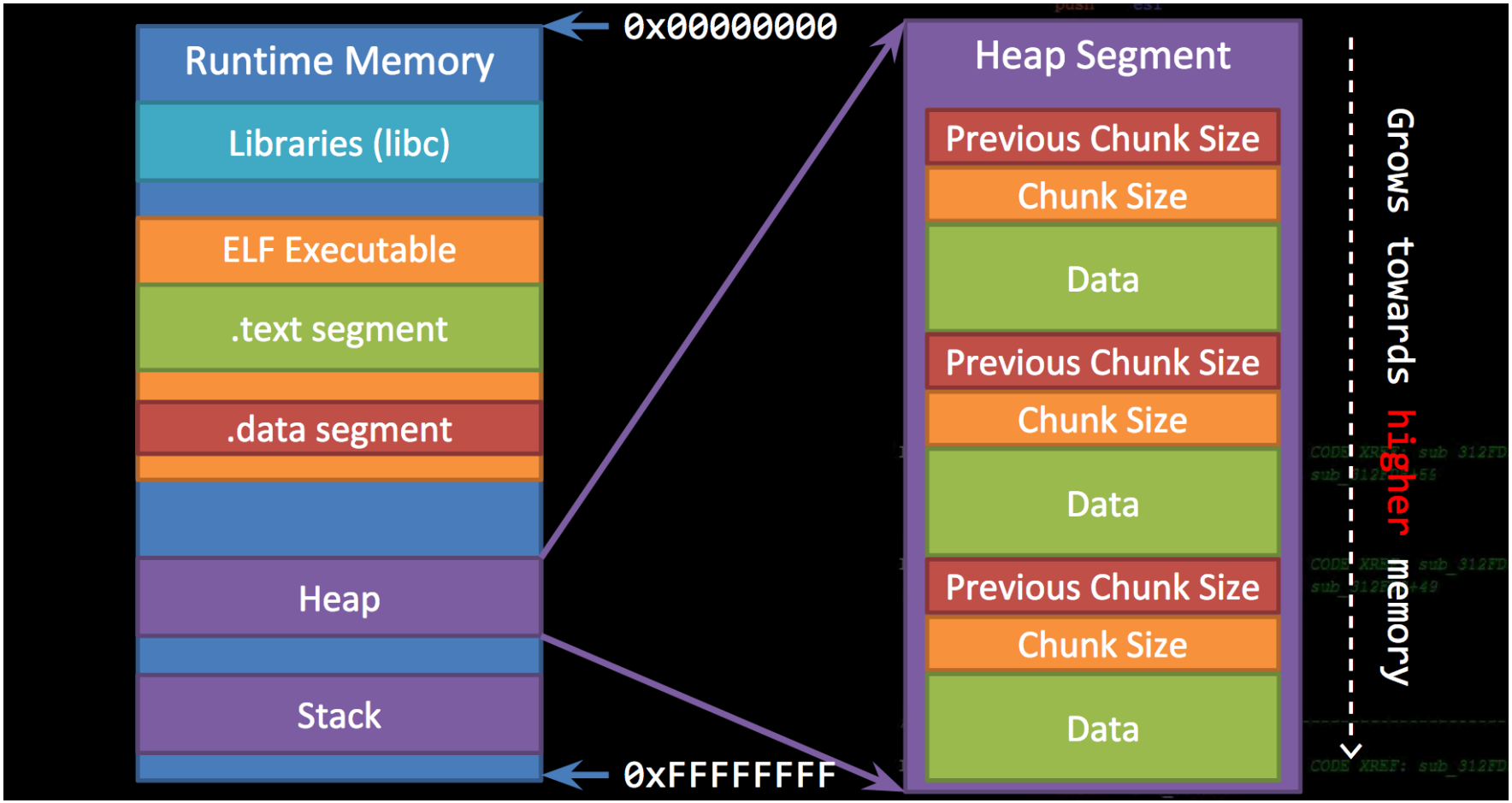
The Heap

- Heap chunks
  - Flags
    - Due to byte alignment , the lower 3 bits of the chunk size field would always be zero.
    - Instead they are used for flag bits.
      - 0x01 PREV-INUSE when previous chunk is in use
      - 0x02 IS_MMAPPED the chunk obtained by mmap()
      - 0x04 NON_MAIN_ARENA the chunk belongs to a thread arena
  - Heap chunks exist in two stages
    - In use (malloc'ed)
    - free'ed

```
lecture@warzone:/levels/lecture/heap$
lecture@warzone:/levels/lecture/heap$ ./heap_chunks
mallocing...
[ prev - 0x00000000 ][ size - 0x00000011 ][ data buffer (0x08276008) -------> ... ] - from malloc(0)
[ prev - 0x00000000 ][ size - 0x00000011 ][ data buffer (0x08276018) -------> ... ] - from malloc(4)
[ prev - 0x00000000 ][ size - 0x00000011 ][ data buffer (0x08276028) -------> ... ] - from malloc(8)
[ prev - 0x00000000 ][ size - 0x00000019 ][ data buffer (0x08276038) -------> ... ] - from malloc(16)
```

The Heap

- Heap allocations

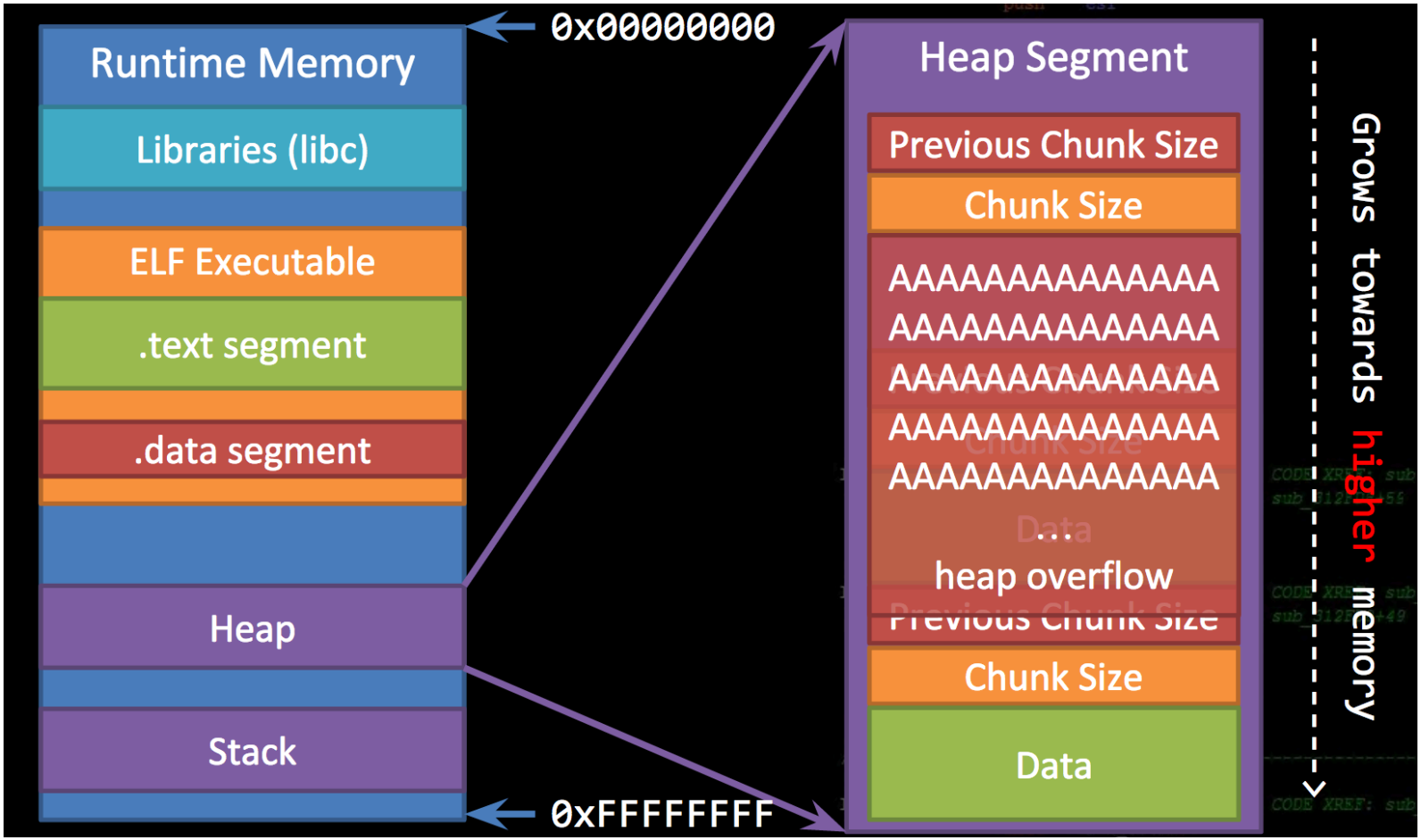


The Heap

```
lecture@warzone:/levels/lecture/heap$ ./print_frees
mallocing...
[ prev - 0x00000000 ][ size - 0x00000011 ][ data buffer (0x08265008) ----> ... ] - Chunk 0x08265000 - In use
[ prev - 0x00000000 ][ size - 0x00000011 ][ data buffer (0x08265018) ----> ... ] - Chunk 0x08265010 - In use
[ prev - 0x00000000 ][ size - 0x00000011 ][ data buffer (0x08265028) ----> ... ] - Chunk 0x08265020 - In use
[ prev - 0x00000000 ][ size - 0x00000019 ][ data buffer (0x08265038) ----> ... ] - Chunk 0x08265030 - In use
[ prev - 0x00000000 ][ size - 0x00000021 ][ data buffer (0x08265050) ----> ... ] - Chunk 0x08265048 - In use
[ prev - 0x00000000 ][ size - 0x00000029 ][ data buffer (0x08265070) ----> ... ] - Chunk 0x08265068 - In use
[ prev - 0x00000000 ][ size - 0x00000049 ][ data buffer (0x08265098) ----> ... ] - Chunk 0x08265090 - In use
[ prev - 0x00000000 ][ size - 0x00000089 ][ data buffer (0x082650e0) ----> ... ] - Chunk 0x082650d8 - In use
[ prev - 0x00000000 ][ size - 0x00000109 ][ data buffer (0x08265168) ----> ... ] - Chunk 0x08265160 - In use
[ prev - 0x00000000 ][ size - 0x00000209 ][ data buffer (0x08265270) ----> ... ] - Chunk 0x08265268 - In use
[ prev - 0x00000000 ][ size - 0x00000409 ][ data buffer (0x08265478) ----> ... ] - Chunk 0x08265470 - In use
[ prev - 0x00000000 ][ size - 0x00000809 ][ data buffer (0x08265880) ----> ... ] - Chunk 0x08265878 - In use
[ prev - 0x00000000 ][ size - 0x00001009 ][ data buffer (0x08266088) ----> ... ] - Chunk 0x08266080 - In use
[ prev - 0x00000000 ][ size - 0x00002009 ][ data buffer (0x08267090) ----> ... ] - Chunk 0x08267088 - In use
[ prev - 0x00000000 ][ size - 0x00004009 ][ data buffer (0x08269098) ----> ... ] - Chunk 0x08269090 - In use

freeing every other chunk...
[ prev - 0x00000000 ][ size - 0x00000011 ][ fd - 0x08265020 ][ bk - 0x08265048 ] - Chunk 0x08265000 - Freed
[ prev - 0x00000010 ][ size - 0x00000010 ][ data buffer (0x08265018) ----> ... ] - Chunk 0x08265010 - In use
[ prev - 0x00000000 ][ size - 0x00000011 ][ fd - 0x08266080 ][ bk - 0x08265000 ] - Chunk 0x08265020 - Freed
[ prev - 0x00000010 ][ size - 0x00000018 ][ data buffer (0x08265038) ----> ... ] - Chunk 0x08265030 - In use
[ prev - 0x00000000 ][ size - 0x00000021 ][ fd - 0x08265000 ][ bk - 0xb7730450 ] - Chunk 0x08265048 - Freed
[ prev - 0x00000020 ][ size - 0x00000028 ][ data buffer (0x08265070) ----> ... ] - Chunk 0x08265068 - In use
[ prev - 0x00000000 ][ size - 0x00000049 ][ fd - 0xb7730450 ][ bk - 0x08265160 ] - Chunk 0x08265090 - Freed
[ prev - 0x00000048 ][ size - 0x00000088 ][ data buffer (0x082650e0) ----> ... ] - Chunk 0x082650d8 - In use
[ prev - 0x00000000 ][ size - 0x00000109 ][ fd - 0x08265090 ][ bk - 0x08265470 ] - Chunk 0x08265160 - Freed
[ prev - 0x00000108 ][ size - 0x00000208 ][ data buffer (0x08265270) ----> ... ] - Chunk 0x08265268 - In use
[ prev - 0x00000000 ][ size - 0x00000409 ][ fd - 0x08265160 ][ bk - 0x08266080 ] - Chunk 0x08265470 - Freed
[ prev - 0x00000408 ][ size - 0x00000808 ][ data buffer (0x08265880) ----> ... ] - Chunk 0x08265878 - In use
[ prev - 0x00000000 ][ size - 0x00001009 ][ fd - 0x08265470 ][ bk - 0x08265020 ] - Chunk 0x08266080 - Freed
[ prev - 0x00001008 ][ size - 0x00002008 ][ data buffer (0x08267090) ----> ... ] - Chunk 0x08267088 - In use
lecture@warzone:/levels/lecture/heap$
```

The Heap

- Heap overflows



The Heap

- Heap overflows
  - Many objects and structures live on the heap
    - Any handles to corrupt data on the heap is now a point of vulnerability
  - A common practice to put function pointers in structs which generally are malloc'd on the heap
    - Overwrite a function pointer on the heap, and force a codepath to call that object's function

The Heap

- Heap overflow (Example)

```
struct toystr {
    void (* message)(char *);
    char buffer[20];
};
```

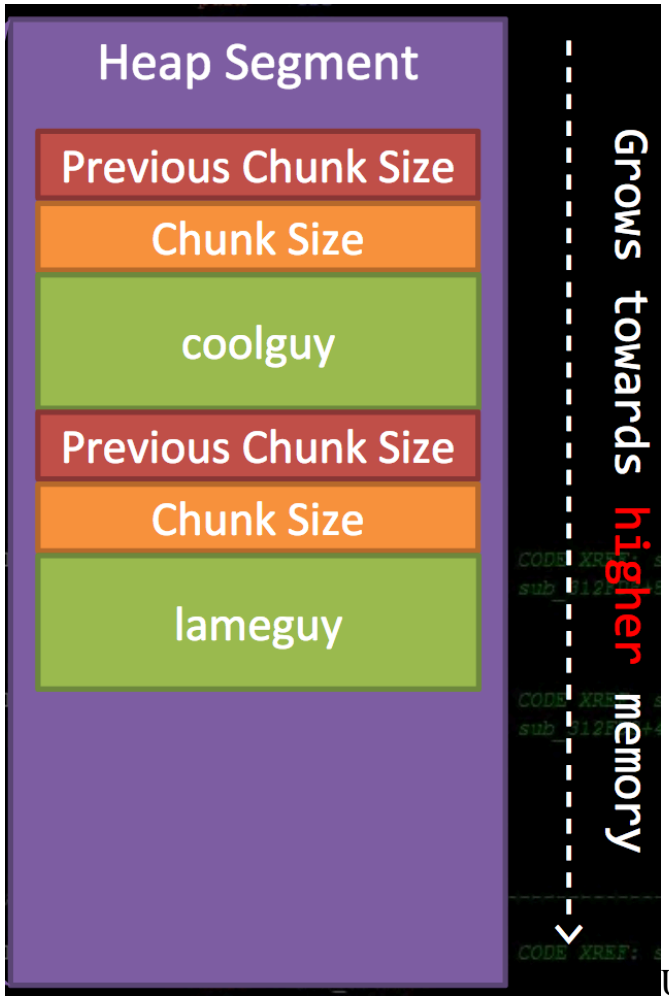
```
coolguy = malloc(sizeof(struct toystr));
lameguy = malloc(sizeof(struct toystr));

coolguy->message = &print_cool;
lameguy->message = &print_meh;

printf("Input coolguy's name: ");
fgets(coolguy->buffer, 200, stdin);  // oopz...
coolguy->buffer[strcspn(coolguy->buffer, "\n")] = 0;

printf("Input lameguy's name: ");
fgets(lameguy->buffer, 20, stdin);
lameguy->buffer[strcspn(lameguy->buffer, "\n")] = 0;

coolguy->message(coolguy->buffer);
lameguy->message(lameguy->buffer);
```

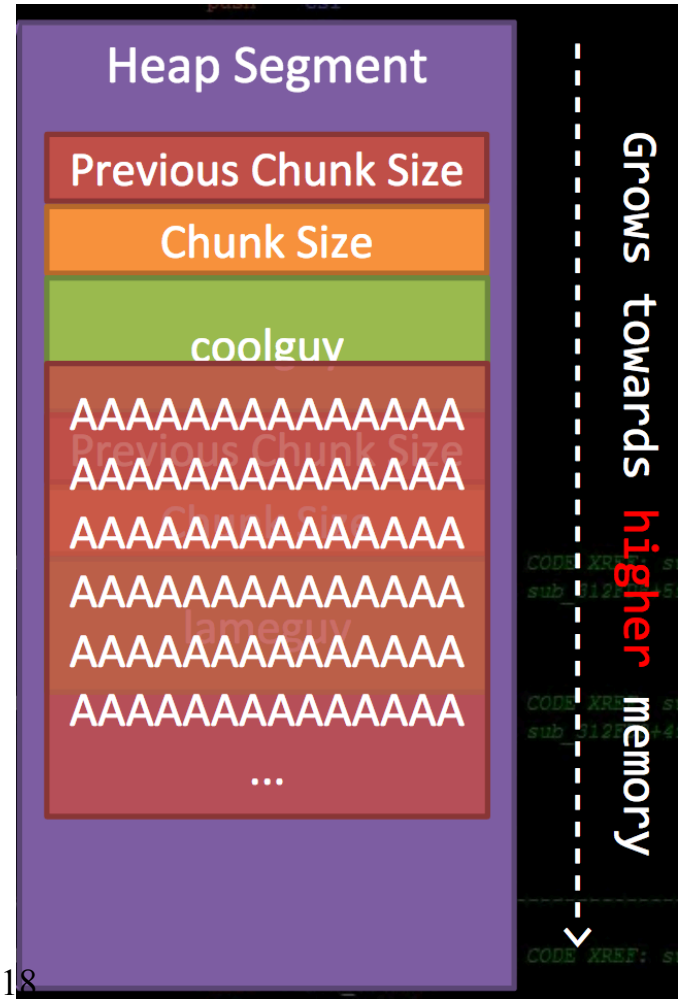
Silly heap overflow

Overwritten function pointer!

The Heap

- Heap overflow

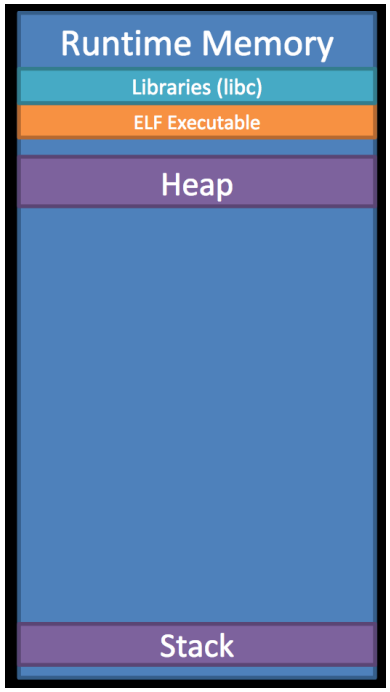


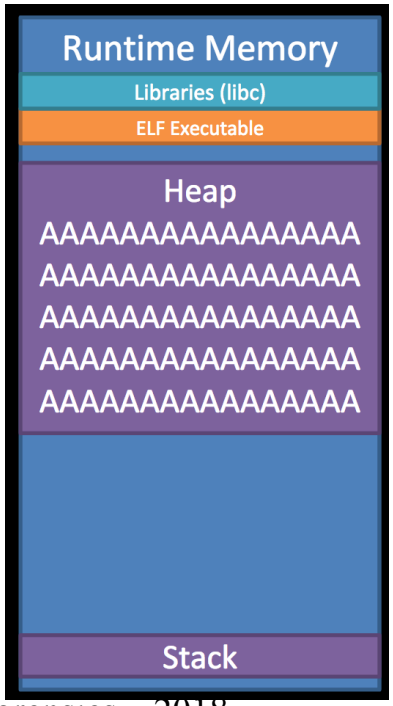


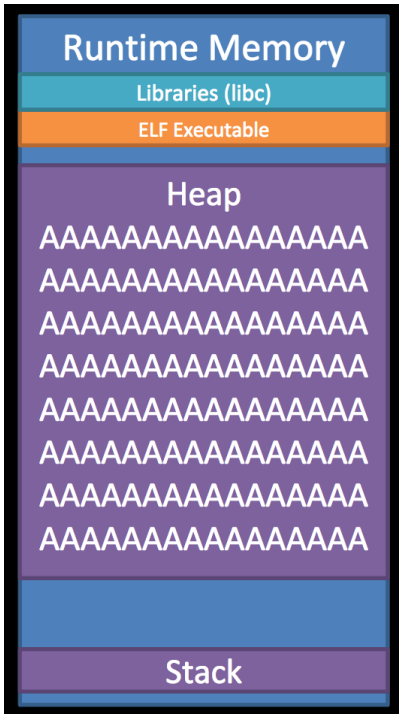
The Heap

- Heap spraying
  - A technique used to increase exploit reliability, by filling the heap with large chinks of data relevant to the exploit you are trying to land
  - Not a vulnerability or security flaw

```
filler = "AAAAAAAAAAAAA...";
for(i = 0; i < 3000; i++)
{
    temp = malloc(1000000);
    memcpy(temp, filler, 1000000);
}
```







The Heap

- Heap spraying
  - An exploitation technique that increases the exploitability of memory corruption vulnerabilities
  - Allocation of many objects containing malicious code (+NOP shellcode) in the heap
  - Increasing the attacker's chance to jump to a location within the heap, successfully malicious code
  - It can assist bypassing ASLR
    - Address space layout randomization (**ASLR**) is a memory-protection process for operating systems (OSs) that guards against buffer-overflow attacks by randomizing the location where system executable are loaded into memory.
  - Generally found in browser exploits
  - Usually is done in something like JavaScript placed on a malicious html page

```
memory = new Array();
for(i = 0; i < 0x100; i++)
    memory[i] = ROPNOP + ROP;
```

The Heap

- Heap spraying on 32bit
  - The possible address space is max 4GB (2^32 bytes)
  - It is unlikely to spray 3GB of anything as heap locations can be somehow predictable, even with ASLR
- Heap spraying on 64bit
  - Cannot be really used to bypass ASLR
- Heap spray payloads
  - Very common to spray some critical value for the exploit, fake objects, or ROP chains