CSE 410/510 Special Topics: Software Security

Instructor: Dr. Ziming Zhao

This Class

1. ROP
2. Approaches to defeat ROP
    a. Return-less code
    b. Control-flow integrity (CFI)
    c. ...

Rop2 (32 bit)

```
FILE* fp = 0;
int a = 0;

int vulfoo(int i)
{
	char buf[200];
	fp = fopen("/tmp/exploit", "r");
	if (!fp) {perror("fopen");exit(0);}

	fread(buf, 1, 190, fp);

	// Move the first 4 bytes to RET
	*((unsigned int *)(&i) - 1) = *((unsigned int *)buf);
	a = *((unsigned int *)buf + 1);

	// Move the second 4 bytes to eax
	asm ( "movl %0, %%eax"
	:
	:"r"(a)
	);
}

int main(int argc, char *argv[])
{ vulfoo(1); return 0;}
```

Useful Gadgets

Stack pivot:

xchg rax, rsp; ret

pop rsp; ...; ret

Rop2 (32 bit)

```
FILE* fp = 0;
int a = 0;

int vulfoo(int i)
{
	char buf[200];
	fp = fopen("exploit", "r");
	if (!fp) {perror("fopen");exit(0);}

	fread(buf, 1, 190, fp);

	// Move the first 4 bytes to RET
	*((unsigned int *)(&i) - 1) = *((unsigned int *)buf);
	a = *((unsigned int *)buf + 1);

	// Move the second 4 bytes to eax
	asm ( "movl %0, %%eax"
	:
	:"r"(a)
	);
}

int main(int argc, char *argv[])
{ vulfoo(1); return 0;}
```

```
p += pack('<I', 0xf7e1a373) # 0xf7e1a373 : xchg eax, esp ; ret
p += pack('<I', 0xffffcf8c) # Move to EAX, so it will be exchanged with ESP; this is
buf+8
…
```

Generalize ROP to COP/JOP

Similarly, other indirect branch instructions, such as Call and Jump indirect can be used to launch variant attacks - called COP (call oriented programming) or JOP (jump oriented programming).

Defeating ROP/COP/JOP

How to pull off a ROP attack?

1. Subvert the control flow to the first gadget.
2. Control the content on the stack. Do not need to inject code there.
3. Enough gadgets in the address space.
4. Know the addresses of the gadgets.
5. Start execution anywhere (middle of instruction).

Ideas to defeat ROP/COP/JOP:
1. Shadow stack / control-flow integrity

Control-Flow Integrity

Principles, Implementations, and Applications

Martín Abadi
Computer Science Dept.
University of California
Santa Cruz

Mihai Budiu Úlfar Erlingsson
Microsoft Research
Silicon Valley

Jay Ligatti
Dept. of Computer Science
Princeton University

ABSTRACT

Current software attacks often build on exploits that subvert machine-code execution. The enforcement of a basic safety property, Control-Flow Integrity (CFI), can prevent such attacks from arbitrarily controlling program behavior. CFI enforcement is simple, and its guarantees can be established formally, even with respect to powerful adversaries. Moreover, CFI enforcement is practical: it is compatible with existing software and can be done efficiently using software rewriting in commodity systems. Finally, CFI provides a useful foundation for enforcing further security policies, as we demonstrate with efficient software implementations of a protected shadow call stack and of access control for memory regions.

bined effects of these attacks make them one of the most pressing challenges in computer security.

In recent years, many ingenious vulnerability mitigations have been proposed for defending against these attacks; these include stack canaries [14], runtime elimination of buffer overflows [46], randomization and artificial heterogeneity [41, 62], and tainting of suspect data [55]. Some of these mitigations are widely used, while others may be impractical, for example because they rely on hardware modifications or impose a high performance penalty. In any case, their security benefits are open to debate: mitigations are usually of limited scope, and attackers have found ways to circumvent each deployed mitigation mechanism [42, 49, 61].

The limitations of these mechanisms stem, in part, from the lack

CCS 2005, Test of Time award 2015

1. ~~Subvert the control flow to the first gadget.~~
2. Control the content on the stack. Do not need to inject code there.
3. Enough gadgets in the address space.
4. Know the addresses of the gadgets.
5. Start execution anywhere (middle of instruction).

Control Flow Integrity (CFI)

1. Control-Flow Integrity (CFI) restricts the control-flow of an program to valid execution traces.
2. CFI enforces this property by monitoring the program at runtime and comparing its state to a set of precomputed valid states. If an invalid state is detected, an alert is raised, usually terminating the application.

Any CFI mechanism consists of two abstract components: the (often static) **analysis component** that recovers the Control-Flow Graph (CFG) of the application (at different levels of precision) and the **dynamic/run-time enforcement mechanism** that restricts control flows according to the generated CFG.

Direct call/jmp vs. Indirect call/jmp

The **direct call/jmp** uses an instruction call/jmp with a **fixed address** as argument. After the compiler/linker has done its job, this address will be included in the opcode. The code text is supposed to be read/executable only and not writable. So, direct call/jmp cannot be subverted.

The **indirect call/jmp** uses an instruction call/jmp with a register as argument (**call rax, jmp rax**). Function return (**ret**) is also considered as indirect because the target is not hardcoded in the instruction.

Call or jmp is named forward-edge (at source code level map to e.g., switch statements, indirect calls, or virtual calls.). The backward-edge is used to return to a location that was used in a forward-edge earlier (return instruction).

Interrupts and interrupt returns.

CFI Enforcement Locations

```
void bar();
void baz();
void buz();
void bez(int, int);

void foo(int usr) {
  void (*func)();

  // func either points to bar or baz
  if (usr == MAGIC)
    func = bar;
  else
    func = baz;

  // forward edge CFI check
  // depending on the precision of CFI:
  // a) all functions {bar, baz, buz, bez, foo} are allowed
  // b) all functions with prototype "void (*)()" are allowed, i.e., {bar, baz, buz}
  // c) only address taken functions are allowed, i.e., {bar, baz}
  CHECK_CFI_FORWARD(func);
  func();

  // backward edge CFI check
  CHECK_CFI_BACKWARD();
}
```

https://nebelwelt.net/blog/20160913-ControlFlowIntegrity.html

Ideas to defeat ROP: 2. ASLR

1. Subvert the control flow to the first gadget.
2. Control the content on the stack. Do not need to inject code there.
3. Enough gadgets in the address space.
4. ~~Know the addresses of the gadgets.~~
5. Start execution anywhere (middle of instruction).

Ideas to defeat ROP: 3. Remove gadgets

G-Free: Defeating Return-Oriented Programming through Gadget-less Binaries

Kaan Onarlioglu
Bilkent University, Ankara
onarliog@cs.bilkent.edu.tr

Leyla Bilge
Eurecom, Sophia Antipolis
bilge@eurecom.fr

Andrea Lanzi
Eurecom, Sophia Antipolis
lanzi@eurecom.fr

Davide Balzarotti
Eurecom, Sophia Antipolis
balzarotti@eurecom.fr

Engin Kirda
Eurecom, Sophia Antipolis
kirda@eurecom.fr

ABSTRACT

Despite the numerous prevention and protection mechanisms that have been introduced into modern operating systems, the exploitation of memory corruption vulnerabilities still represents a serious threat to the security of software systems and networks. A recent exploitation technique, called Return-Oriented Programming (ROP), has lately attracted a considerable attention from academia. Past research on the topic has mostly focused on refining the original attack technique, or on proposing partial solutions that target only particular variants of the attack.

In this paper, we present G-Free, a compiler-based approach that represents the first practical solution against any possible form of

to find a technique to overwrite a pointer in memory. Overflowing a buffer on the stack [5] or exploiting a format string vulnerability [26] are well-known examples of such techniques. Once the attacker is able to hijack the control flow of the application, the next step is to take control of the program execution to perform some malicious activity. This is typically done by injecting in the process memory a small payload that contains the machine code to perform the desired task.

A wide range of solutions have been proposed to defend against memory corruption attacks, and to increase the complexity of performing these two attack steps [10, 11, 12, 18, 35]. In particular, all modern operating systems support some form of memory pro-

ACSAC 2010

RET?

x86 Instruction Set Reference

RET

Return from Procedure

Opcode	Mnemonic	Description
C3	RET	Near return to calling procedure.
CB	RET	Far return to calling procedure.
C2 iw	RET imm16	Near return to calling procedure and pop imm16 bytes from stack.
CA iw	RET imm16	Far return to calling procedure and pop imm16 bytes from stack.

Description

Transfers program control to a return address located on the top of the stack. The address is usually placed on the stack by a CALL instruction, and the return is made to the instruction that follows the CALL instruction.

The optional source operand specifies the number of stack bytes to be released after the return address is popped; the default is none. This operand can be used to release parameters from the stack that were passed to the called procedure and are no longer needed. It must be used when the CALL instruction used to switch to a new procedure uses a call gate with a non-zero word count to access the new procedure. Here, the source operand for the RET instruction must specify the same number of bytes as is specified in the word count field of the call gate.

The RET instruction can be used to execute three different types of returns:

Ideas to defeat ROP: 3. Remove gadgets

Basic idea: Remove C3/C2/CA/CB from the code

Jump and call instructions may contain free-branch opcodes when using immediate values to specify their destinations. For instance, `jmp .+0xc8` is encoded as "`0xe9 0xc3 0x00 0x00 0x00`".

A free-branch opcode can appear at any of the four bytes constituting the jump/call target. If the opcode is the least significant byte, it is sufficient to append the forward jump/call with a single `nop` instruction (or prepend it if it is a backwards jump/call) in order to adjust the relative distance between the instruction and its destination:

```
jmp .+0xc8   ⇒   jmp .+0xc9
                 nop
```

Ideas to defeat ROP: 3. Remove gadgets

Basic idea: Remove C3/C2/CA/CB from the code

```
addl $0xc2, %eax   ⇒   addl $0xc1, %eax
                       inc %eax

 xorb $0xca, %al   ⇒   movb $0xc9, %bl
                       incb %bl
                       xorb %bl, %al
```

Ideas to defeat ROP: 3. Remove gadgets

Basic idea: Remove C3/C2/CA/CB from the code

Instructions that perform memory accesses can also contain free-branch instruction opcodes in the displacement values they specify (e.g., `movb %al, -0x36(%ebp)` represented as "`0x88 0x45 0xca`"). In such cases, we need to substitute the instruction with a semantically equivalent instruction sequence that uses an adjusted displacement value to avoid the undesired bytes. We achieve this by setting the displacement to a safe value and then compensating for our changes by temporarily adjusting the value in the base register. For example, we can perform a reconstruction such as:

```
movb $0xal, -0x36(%ebp)   ⇒   incl %ebp
                              movb %al, -0x37(%ebp)
                              decl %ebp
```

Ideas to defeat ROP: 3. Remove gadgets

1. Subvert the control flow to the first gadget.
2. Control the content on the stack. Do not need to inject code there.
3. ~~Enough gadgets in the address space.~~
4. Know the addresses of the gadgets.
5. Start execution anywhere (middle of instruction).

Ideas to defeat ROP: 4. Monitor CFI

Transparent ROP Exploit Mitigation using Indirect Branch Tracing

Vasilis Pappas, Michalis Polychronakis, Angelos D. Keromytis
Columbia University

Abstract

Return-oriented programming (ROP) has become the primary exploitation technique for system compromise in the presence of non-executable page protections. ROP exploits are facilitated mainly by the lack of complete address space randomization coverage or the presence of memory disclosure vulnerabilities, necessitating additional ROP-specific mitigations.

In this paper we present a practical runtime ROP ex-

bypassing the data execution prevention (DEP) and address space layout randomization (ASLR) protections of Windows [49], even on the most recent and fully updated (at the time of public notice) systems.

Data execution prevention and similar non-executable page protections [55], which prevent the execution of injected binary code (shellcode), can be circumvented by reusing code that already exists in the vulnerable process to achieve the same purpose. Return-oriented programming (ROP) [62], the latest advancement in the

USENIX Security 2013

kBouncer: Efficient and Transparent ROP Mitigation

Vasilis Pappas
Columbia University
vpappas@cs.columbia.edu

April 1, 2012

Abstract

The wide adoption of non-executable page protections in recent versions of popular operating systems has given rise to attacks that employ return-oriented programming (ROP) to achieve arbitrary code execution without the injection of any code. Existing defenses against ROP exploits either require source code or symbolic debugging information, impose a significant runtime overhead, which limits their applicability for the protection of third-party applications, or may require to make some assumptions about the executable code of the protected applications. We propose kBouncer, an efficient and fully transparent ROP mitigation technique that does not requires source code or debug symbols. kBouncer is based on runtime detection of abnormal control transfers using hardware features found on commodity processors.

1 Problem Description

The introduction of non-executable memory page protections led to the development of the return-to-libc exploitation technique [11]. Using this method, a memory corruption vulnerability can be exploited by transferring control to code that already exists in the address space of the vulnerable process. By jumping

Ideas to defeat ROP: 5. Indirect Branch Tracking

All indirect branch targets must start with ENDBR64/ENDBR32.

- ENDBR64/ENDBR32 is NOP on non-CET processors.

```
080493b8 <_fini>:
 80493b8:       f3 0f 1e fb             endbr32
 80493bc:       53                      push   %ebx
 80493bd:       83 ec 08                sub    $0x8,%esp
 80493c0:       e8 8b fd ff ff          call   8049150 <__x86.get_pc_thunk.bx>
 80493c5:       81 c3 3b 2c 00 00       add    $0x2c3b,%ebx
 80493cb:       83 c4 08                add    $0x8,%esp
 80493ce:       5b                      pop    %ebx
 80493cf:       c3                      ret
```

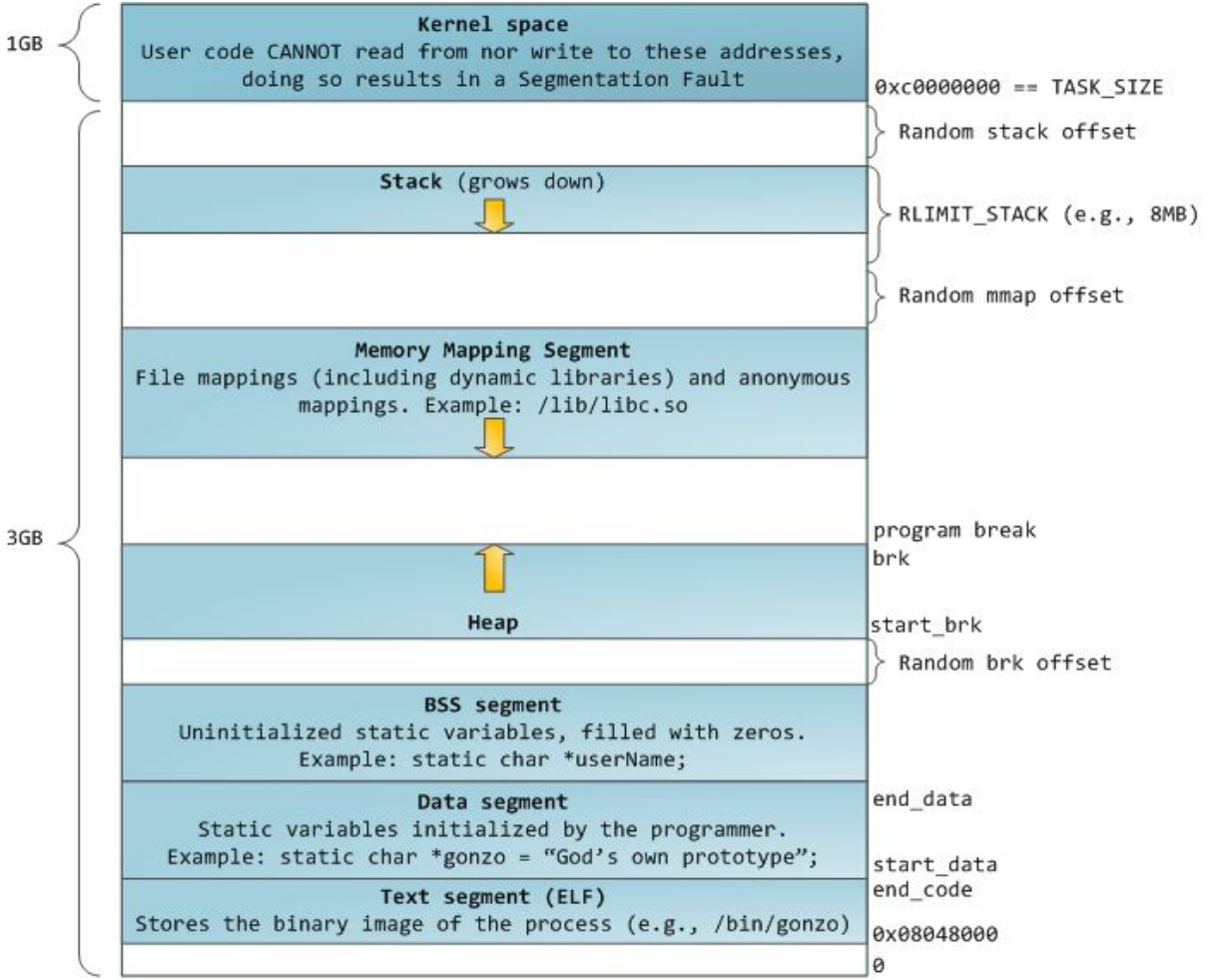
CSE 410/510 Special Topics: Software Security

Instructor: Dr. Ziming Zhao

Today

1. Heap and heap exploitation

Memory Map of Linux Process (32 bit system)



https://manybutfinite.com/post/anatomy-of-a-program-in-memory/

The Heap

The heap is pool of memory used for dynamic allocations at runtime. Heap memory is different from stack memory in that it is ***persistent between functions***.

– **malloc**() grabs memory on the heap; keyword ***new*** in C++
– **free**() releases memory on the heap; keyword ***delete*** in C++

Both are standard C library interfaces. Neither of them directly mapps to a system call.

malloc() and free()

```
void* malloc(size_t size);
```

Allocates size bytes of uninitialized storage. If allocation succeeds, returns a pointer that is suitably aligned for any object type with fundamental alignment.

```
void free(void* ptr);
```

Deallocates the space previously allocated by malloc(), etc.

calloc() and realloc()

```
void *calloc(size_t nitems, size_t size)
```

The difference in malloc and calloc is that malloc does not set the memory to zero whereas calloc sets allocated memory to zero.

```
void *realloc(void *ptr, size_t size)
```

Resize the memory block pointed to by ptr that was previously allocated with a call to malloc or calloc.

How to use malloc() and free()

```
int main()
{
        char * buffer = NULL;

        /* allocate a 0x100 byte buffer */
        buffer = malloc(0x100);

        /* read input and print it */
        fgets(stdin, buffer, 0x100);
        printf("Hello %s!\n", buffer);

        /* destroy our dynamically allocated buffer */
        free(buffer);
        return 0;
}
```

Heap vs. Stack

Heap

- Dynamic memory allocations at runtime
- Objects, big buffers, structs, persistence, larger things

Slower, Manual
– Done by the programmer
– malloc/calloc/recalloc/free
– new/delete

Stack

- Fixed memory allocations known at compile time
- Local variables, return addresses, function args

Fast, Automatic; Done by the compiler
– Abstracts away any concept of allocating/de-allocating

Heap Implementations

Doug Lea malloc or **dlmalloc**. Default native version of malloc in some old distributions of Linux (http://gee.cs.oswego.edu/dl/html/malloc.html)

ptmalloc. ptmalloc is based on dlmalloc and was extended for use with multiple threads. On Linux systems, ptmalloc has been put to work for years as part of the GNU C library.

tcmalloc. Google's customized implementation of C's malloc() and C++'s operator new (https://github.com/google/tcmalloc)

jemalloc. jemalloc is a general purpose malloc(3) implementation that emphasizes fragmentation avoidance and scalable concurrency support.

The **Hoard** memory allocator. UMass Amherst CS Professor Emery Berger

Which implementation on my laptop?

ldd --version

GLIBC 2.31

Ptmalloc2

https://elixir.bootlin.com/glibc/glibc-2.31/source/malloc/malloc.c

```
➜ heapfrees ldd --version
ldd (Ubuntu GLIBC 2.31-0ubuntu9.2) 2.31
Copyright (C) 2020 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
Written by Roland McGrath and Ulrich Drepper.
```

Overview of dlmalloc

The Linux version of the dynamic memory allocator. Even though it has been updated, from the point of view of software infused bugs and exploits, new versions are still more or less similar to the original one.

Design goals:

Maximizing Portability To rely on as few system-dependent features as possible, system calls in particular.

Minimizing Space The allocator should not waste memory. It should obtain the least amount of memory from the system it requires, and should maintain memory in ways that minimize fragmentation—that is, it should try to avoid creating a large number of contiguous chunks of memory that are not used by the program.

Minimizing Time The malloc(), free(), and realloc calls should be fast on average.

Overview of dlmalloc

The Linux version of the dynamic memory allocator. Even though it has been updated, from the point of view of software infused bugs and exploits, new versions are still more or less similar to the original one.

Design goals:

Maximizing Locality Allocate chunks of memory that are typically requested or used together near each other.This will help minimize CPU page and cache misses.

Maximizing Error Detection Should provide some means for detecting corruption due to overwriting memory, multiple frees, and so on. It is not supposed to work as a general memory leak detection tool at the cost of slowing down.

Minimizing Anomalies It should have reasonably similar performance characteristics across a wide range of possible applications whether they are GUI or server programs, string processing applications, or network tools.

Malloc_chunk (ptmalloc2 in glibc2.31)

```
struct malloc_chunk {
  INTERNAL_SIZE_T      mchunk_prev_size;  /* Size of previous chunk (if free).  */
  INTERNAL_SIZE_T      mchunk_size;       /* Size in bytes, including overhead. */

  struct malloc_chunk* fd;         /* double links -- used only if free. */
  struct malloc_chunk* bk;

  /* Only used for large blocks: pointer to next larger size.  */
  struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
  struct malloc_chunk* bk_nextsize;
};
```

Both in-use and freed

Only for freed

INTERNAL_SIZE_T is the same as size_t. 8 bytes in 64 bit;
4 bytes in 32 bits machine.
Pointer is 8/4 bytes on a 64/32 bit machine, respectively.

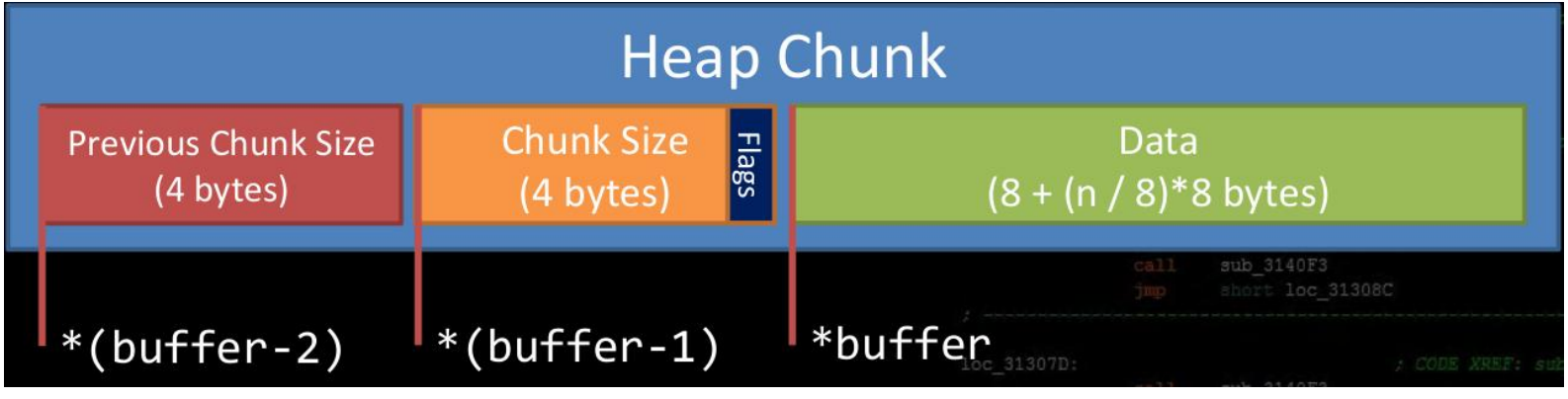
https://elixir.bootlin.com/glibc/glibc-2.31/source/malloc/malloc.c

Heap Chunks (figures in 32 bit)

buffer = malloc(0x100);

//Out comes a heap chunk

Previous Chunk Size: Size of previous chunk (if prev chunk is free)
Chunk Size: Size of entire chunk including overhead
Data: Your newly allocated memory / ptr returned by malloc
Flags: Because of 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 – set when previous chunk is in use
0x02 IS_MMAPPED – set if chunk was obtained with mmap()
0x04 NON_MAIN_ARENA – set if chunk belongs to a thread arena



Malloc Trivia

How many bytes on the heap are your ***malloc chunks*** really taking up?

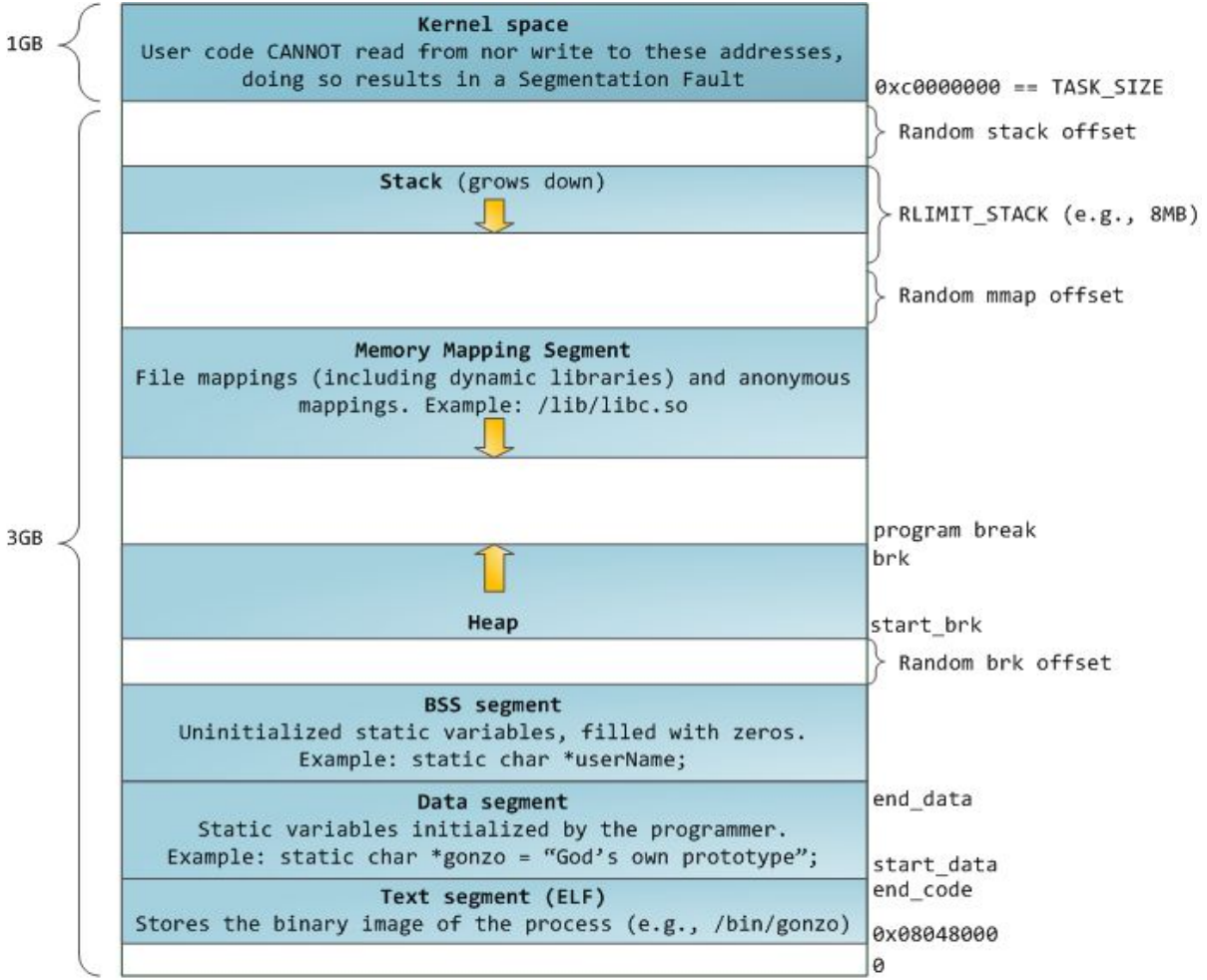
- malloc(32);
- malloc(4);
- malloc(20);
- malloc(0);

code/heapsizes

```
int main()
{
   unsigned int lengths[] = {32, 4, 20, 0, 64, 32, 32, 32, 32, 32};
   unsigned int * ptr[10];
   int i;

   for(i = 0; i < 10; i++)
      ptr[i] = malloc(lengths[i]);

   for(i = 0; i < 9; i++)
      printf("malloc(%2d) is at 0x%08x, %3d bytes to the next pointer\n",
            lengths[i],
            (unsigned int)ptr[i],
            (ptr[i+1]-ptr[i])*sizeof(unsigned int));

  return 0;}
```

https://github.com/RPISEC/MBE/blob/master/src/lecture/heap/sizes.c

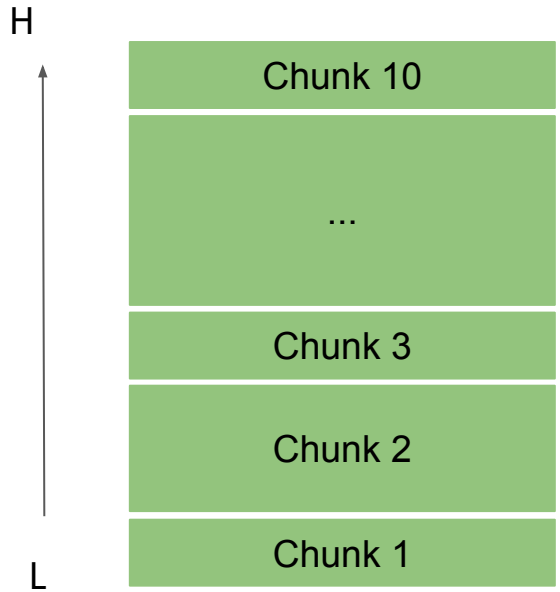
Heap goes from low address to high address



https://manybutfinite.com/post/anatomy-of-a-program-in-memory/

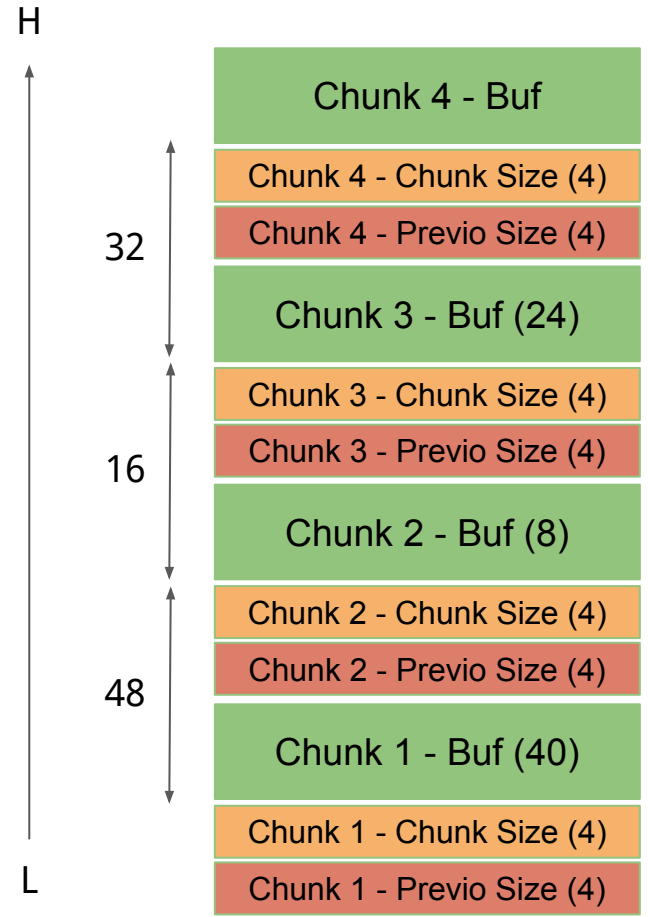
code/heapsizes

```
int main()
{
   unsigned int lengths[] = {32, 4, 20, 0, 64, 32, 32, 32, 32, 32};
   unsigned int * ptr[10];
   int i;

   for(i = 0; i < 10; i++)
      ptr[i] = malloc(lengths[i]);

   for(i = 0; i < 9; i++)
      printf("malloc(%2d) is at 0x%08x, %3d bytes to the next pointer\n",
            lengths[i],
            (unsigned int)ptr[i],
            (ptr[i+1]-ptr[i])*sizeof(unsigned int));

  return 0;}
```

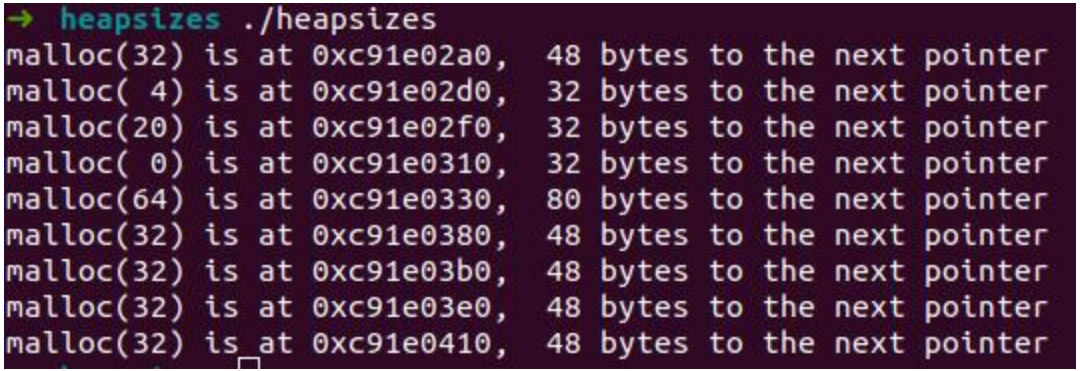


code/heapsizes 32bit

```
➜ heapsizes ./heapsizes32
malloc(32) is at 0x5695b1a0,  48 bytes to the next pointer
malloc( 4) is at 0x5695b1d0,  16 bytes to the next pointer
malloc(20) is at 0x5695b1e0,  32 bytes to the next pointer
malloc( 0) is at 0x5695b200,  16 bytes to the next pointer
malloc(64) is at 0x5695b210,  80 bytes to the next pointer
malloc(32) is at 0x5695b260,  48 bytes to the next pointer
malloc(32) is at 0x5695b290,  48 bytes to the next pointer
malloc(32) is at 0x5695b2c0,  48 bytes to the next pointer
malloc(32) is at 0x5695b2f0,  48 bytes to the next pointer
```



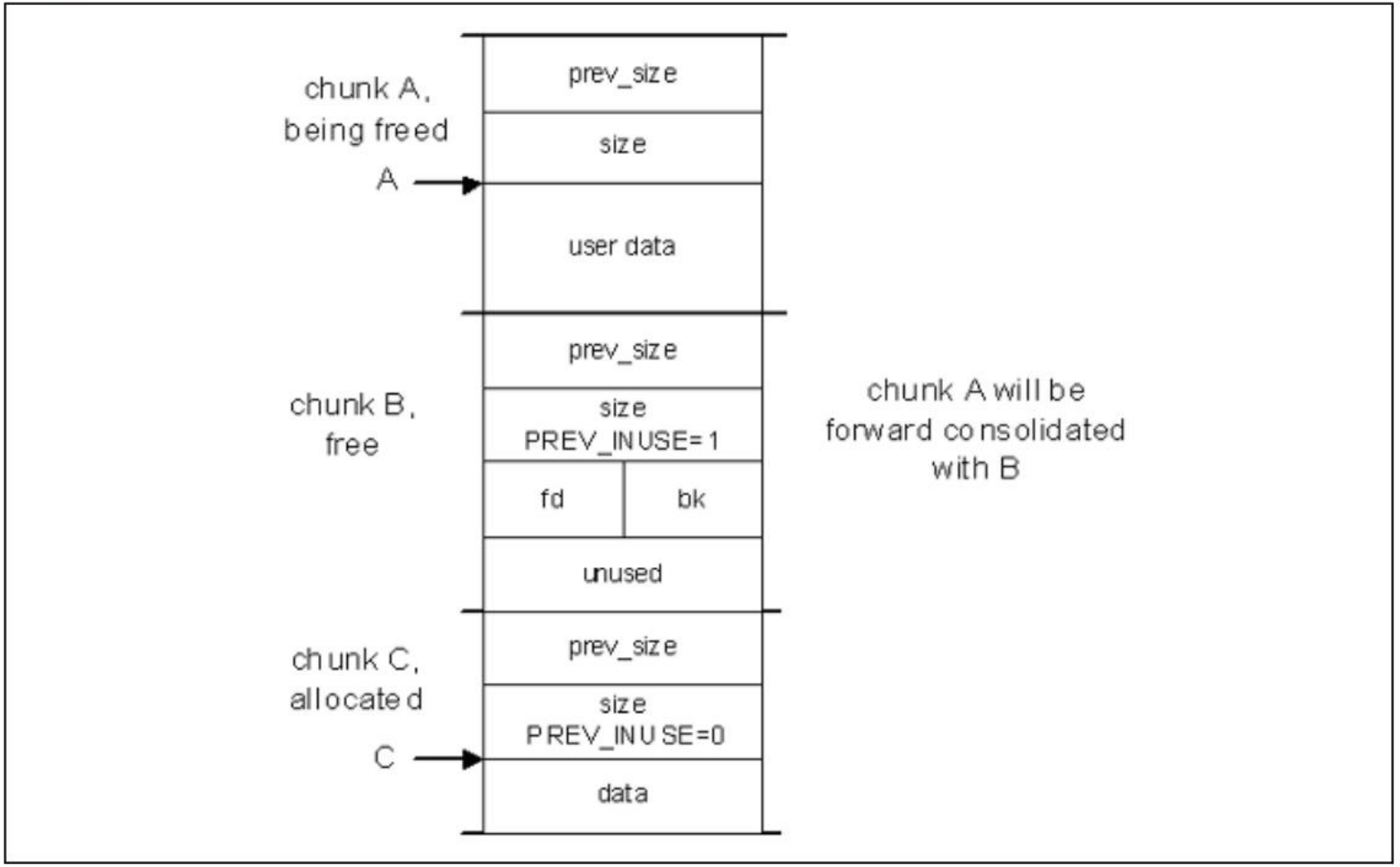
code/heapsizes 64bit



Malloc Trivia

How many bytes on the heap are your ***malloc chunks*** really taking up?

- malloc(32); 48 bytes (32bit/64bit)
- malloc(4); 16 bytes (32bit) / 32 bytes (64bit)
- malloc(20); 32 bytes (32bit/64bit)
- malloc(0); 16 bytes (32bit) / 32 bytes (64bit)


chunk A,
being freed
A
prev_size
size
user data
chunk B,
free
prev_size
size
PREV_INUSE=1
fd
bk
unused
chunk A will be
forward consolidated
with B
chunk C,
allocated
C
prev_size
size
PREV_INUSE=0
data

code/heapchunks

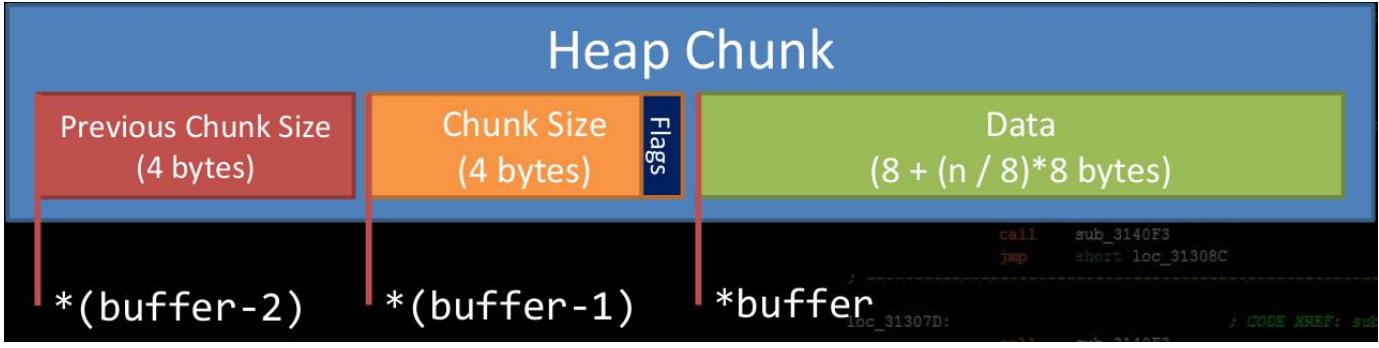
```
void print_chunk(size_t * ptr, unsigned int len)
{
  printf("[ prev - 0x%08x ][ size - 0x%08x ][ data buffer (0x%08x) -------> ... ] - from
malloc(%d)\n", *(ptr-2), *(ptr-1), (unsigned int)ptr, len);}

int main()
{
   void * ptr[LEN];
   unsigned int lengths[] = {0, 4, 8, 16, 24, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384};
   int i;

   printf("mallocing...\n");

   for(i = 0; i < LEN; i++)
      ptr[i] = malloc(lengths[i]);

   for(i = 0; i < LEN; i++)
      print_chunk(ptr[i], lengths[i]);
   return 0;}
```

Extended from https://github.com/RPISEC/MBE/blob/master/src/lecture/heap/heap_chunks.c

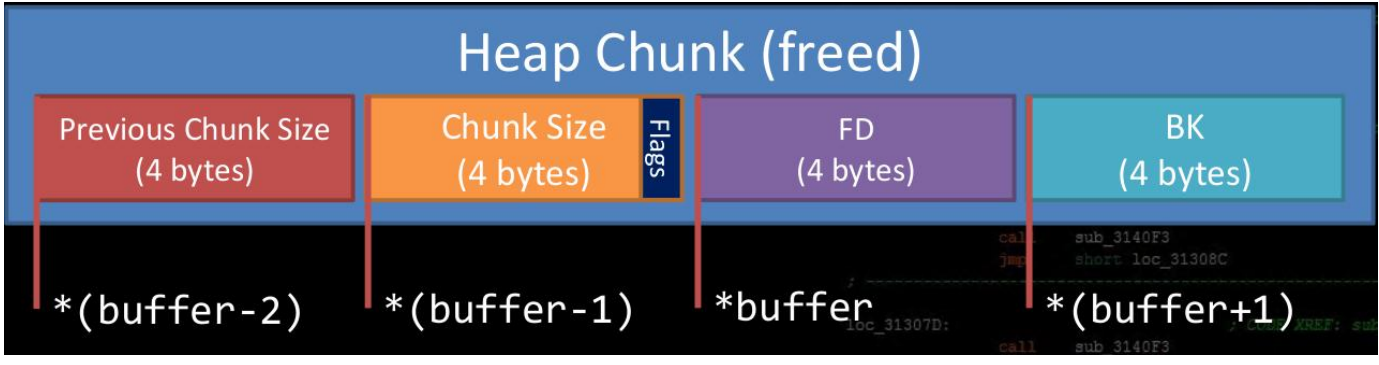
```
➜ heapchunks ./heapchunks32
mallocing...
[ prev - 0x00000000 ][ size - 0x00000011 ][ data buffer (0x57b665b0) -------> ... ] - from malloc(0)
[ prev - 0x00000000 ][ size - 0x00000011 ][ data buffer (0x57b665c0) -------> ... ] - from malloc(4)
[ prev - 0x00000000 ][ size - 0x00000011 ][ data buffer (0x57b665d0) -------> ... ] - from malloc(8)
[ prev - 0x00000000 ][ size - 0x00000021 ][ data buffer (0x57b665e0) -------> ... ] - from malloc(16)
[ prev - 0x00000000 ][ size - 0x00000021 ][ data buffer (0x57b66600) -------> ... ] - from malloc(24)
[ prev - 0x00000000 ][ size - 0x00000031 ][ data buffer (0x57b66620) -------> ... ] - from malloc(32)
[ prev - 0x00000000 ][ size - 0x00000051 ][ data buffer (0x57b66650) -------> ... ] - from malloc(64)
[ prev - 0x00000000 ][ size - 0x00000091 ][ data buffer (0x57b666a0) -------> ... ] - from malloc(128)
[ prev - 0x00000000 ][ size - 0x00000111 ][ data buffer (0x57b66730) -------> ... ] - from malloc(256)
[ prev - 0x00000000 ][ size - 0x00000211 ][ data buffer (0x57b66840) -------> ... ] - from malloc(512)
[ prev - 0x00000000 ][ size - 0x00000411 ][ data buffer (0x57b66a50) -------> ... ] - from malloc(1024)
[ prev - 0x00000000 ][ size - 0x00000811 ][ data buffer (0x57b66e60) -------> ... ] - from malloc(2048)
[ prev - 0x00000000 ][ size - 0x00001011 ][ data buffer (0x57b67670) -------> ... ] - from malloc(4096)
[ prev - 0x00000000 ][ size - 0x00002011 ][ data buffer (0x57b68680) -------> ... ] - from malloc(8192)
[ prev - 0x00000000 ][ size - 0x00004011 ][ data buffer (0x57b6a690) -------> ... ] - from malloc(16384)
➜ heapchunks ./heapchunks
mallocing...
[ prev - 0x00000000 ][ size - 0x00000021 ][ data buffer (0x665046b0) -------> ... ] - from malloc(0)
[ prev - 0x00000000 ][ size - 0x00000021 ][ data buffer (0x665046d0) -------> ... ] - from malloc(4)
[ prev - 0x00000000 ][ size - 0x00000021 ][ data buffer (0x665046f0) -------> ... ] - from malloc(8)
[ prev - 0x00000000 ][ size - 0x00000021 ][ data buffer (0x66504710) -------> ... ] - from malloc(16)
[ prev - 0x00000000 ][ size - 0x00000021 ][ data buffer (0x66504730) -------> ... ] - from malloc(24)
[ prev - 0x00000000 ][ size - 0x00000031 ][ data buffer (0x66504750) -------> ... ] - from malloc(32)
[ prev - 0x00000000 ][ size - 0x00000051 ][ data buffer (0x66504780) -------> ... ] - from malloc(64)
[ prev - 0x00000000 ][ size - 0x00000091 ][ data buffer (0x665047d0) -------> ... ] - from malloc(128)
[ prev - 0x00000000 ][ size - 0x00000111 ][ data buffer (0x66504860) -------> ... ] - from malloc(256)
[ prev - 0x00000000 ][ size - 0x00000211 ][ data buffer (0x66504970) -------> ... ] - from malloc(512)
[ prev - 0x00000000 ][ size - 0x00000411 ][ data buffer (0x66504b80) -------> ... ] - from malloc(1024)
[ prev - 0x00000000 ][ size - 0x00000811 ][ data buffer (0x66504f90) -------> ... ] - from malloc(2048)
[ prev - 0x00000000 ][ size - 0x00001011 ][ data buffer (0x665057a0) -------> ... ] - from malloc(4096)
[ prev - 0x00000000 ][ size - 0x00002011 ][ data buffer (0x665067b0) -------> ... ] - from malloc(8192)
[ prev - 0x00000000 ][ size - 0x00004011 ][ data buffer (0x665087c0) -------> ... ] - from malloc(16384)
```

Heap Chunks – Two states (figures in 32 bit)

Heap chunks exist in two states
– in use (malloc'd)



– free'd.
Forward Pointer: A pointer to the next freed chunk
Backwards Pointer: A pointer to the previous freed chunk
Implementation-defined.



code/heapfrees

```
void print_inuse_chunk(unsigned int * ptr)
{
	printf("[ prev - 0x%08x ][ size - 0x%08x ][ data buffer
(0x%08x) ----> ... ] - Chunk 0x%08x - In use\n", \
		*(ptr-2),
		*(ptr-1),
		(unsigned int)ptr,
		(unsigned int)(ptr-2));
}

void print_freed_chunk(unsigned int * ptr)
{
	printf("[ prev - 0x%08x ][ size - 0x%08x ][ fd - 0x%08x ][ bk -
0x%08x ] - Chunk 0x%08x - Freed\n", \
		*(ptr-2),
		*(ptr-1),
		*ptr,
		*(ptr+1),
		(unsigned int)(ptr-2));
}
```

```
int main()
{
   unsigned int * ptr[LEN];
   unsigned int lengths[] = {32, 32, 32, 32, 32}; int i;

   printf("mallocing...\n");
   for(i = 0; i < LEN; i++)
      ptr[i] = malloc(lengths[i]);

   for(i = 0; i < LEN; i++)
      print_inuse_chunk(ptr[i]);

   printf("\nfreeing all chunks...\n");
   for(i = 0; i < LEN; i++)
      free(ptr[i]);

   for(i = 0; i < LEN; i++)
      print_freed_chunk(ptr[i]);

   return 0;}
```

Take away

The strength and popularity of heap overflow exploits comes from the way specific memory allocation functions are implemented within the individual programming languages and underlying operating platforms.

Many common implementations store control data in-line together with the actual allocated memory.