Hacking in C

Exploring Stack and Heap

Thom Wiggers

Last week

- Arrays

Last week

- Arrays
- Pointers

Last week

- Arrays
- Pointers
  - Pointers to pointers

Last week

- Arrays
- Pointers
  - Pointers to pointers
  - Pointers too (see previous point)

Last week

- Arrays
- Pointers
  - Pointers to pointers
  - Pointers too (see previous point)
- `int* a_ptr = &a;`

Last week

- Arrays
- Pointers
  - Pointers to pointers
  - Pointers too (see previous point)
- `int* a_ptr = &a;`
- Dereferencing `*a`

Last week

- Arrays
- Pointers
  - Pointers to pointers
  - Pointers too (see previous point)
- `int* a_ptr = &a;`
- Dereferencing `*a`
- Strings

Last week

- Arrays
- Pointers
  - Pointers to pointers
  - Pointers too (see previous point)
- `int* a_ptr = &a;`
- Dereferencing `*a`
- Strings
- The horrible ways strings ruin your day

Last week

- Arrays
- Pointers
  - Pointers to pointers
  - Pointers too (see previous point)
- `int* a_ptr = &a;`
- Dereferencing `*a`
- Strings
- The horrible ways strings ruin your day
- Some bit of slide-karaoke about memory that wasn't prepared

This week

The stack
  Local variables
  The stack

The heap

Special memory segments

Wrapping up memory

Reading the stack

Extra content
  Memory quizzes
  Finding memory bugs

Table of Contents

Local variables

Imagine the following program

```
#include "headers.h"

int main(int argc, char* argv[]){
    int a = 3;
    int b = 4;
    int c = some_function();
    return 0;
}

int some_function() {
    char arr[100] = {0};
    return 3;
}
```

How could we manage variables efficiently?

Properties of local variables

Local variables are:

- local to the function

Properties of local variables

Local variables are:

- local to the function
  - they can't be accessed by other functions

Properties of local variables

Local variables are:

- local to the function
  - they can't be accessed by other functions
- local to the function call

Properties of local variables

Local variables are:

- local to the function
  - they can't be accessed by other functions
- local to the function call
  - If you call the function multiple times, each has its own copy of its state

Properties of local variables

Local variables are:

- local to the function
  - they can't be accessed by other functions
- local to the function call
  - If you call the function multiple times, each has its own copy of its state
  - This holds especially when you're calling it recursively

Properties of local variables

Local variables are:

- local to the function
    - they can't be accessed by other functions
- local to the function call
    - If you call the function multiple times, each has its own copy of its state
    - This holds especially when you're calling it recursively
- Only exist during the function call

Option: pre-allocating all variables beforehand

Let's turn every local variable into a global variable

- Having a single copy per declared local variable breaks the isolation properties

Option: pre-allocating all variables beforehand

Let's turn every local variable into a global variable

- Having a single copy per declared local variable breaks the isolation properties
- To allow every function call its own local variables you'd need to fully trace the entire call graph and create copies

Option: pre-allocating all variables beforehand

Let's turn every local variable into a global variable

- Having a single copy per declared local variable breaks the isolation properties
- To allow every function call its own local variables you'd need to fully trace the entire call graph and create copies
  - Not possible if your program does stuff differently based on the input

Option: pre-allocating all variables beforehand

Let's turn every local variable into a global variable

- Having a single copy per declared local variable breaks the isolation properties
- To allow every function call its own local variables you'd need to fully trace the entire call graph and create copies
  - Not possible if your program does stuff differently based on the input
  - You'd possibly need lots and lots of space

Option: pre-allocating all variables beforehand

Let's turn every local variable into a global variable

- Having a single copy per declared local variable breaks the isolation properties
- To allow every function call its own local variables you'd need to fully trace the entire call graph and create copies
  - Not possible if your program does stuff differently based on the input
  - You'd possibly need lots and lots of space
- Clearly not an option

Active management of local variables

Assuming we don't know better, let's ask the memory manager for space each time we create a variable

- Requires setup code to be executed every function call for every variable

Active management of local variables

Assuming we don't know better, let's ask the memory manager for space each time we create a variable

- Requires setup code to be executed every function call for every variable
    - Would need to check: where have I got space? Is the memory fragmented?

Active management of local variables

Assuming we don't know better, let's ask the memory manager for space each time we create a variable

- Requires setup code to be executed every function call for every variable
  - Would need to check: where have I got space? Is the memory fragmented?
- Needs to make sure you don't have collisions between function calls

Active management of local variables

Assuming we don't know better, let's ask the memory manager for space each time we create a variable

- Requires setup code to be executed every function call for every variable
    - Would need to check: where have I got space? Is the memory fragmented?
- Needs to make sure you don't have collisions between function calls
- Execute expensive clean-up code per variable when the function returns

Active management of local variables

Assuming we don't know better, let's ask the memory manager for space each time we create a variable

- Requires setup code to be executed every function call for every variable
  - Would need to check: where have I got space? Is the memory fragmented?
- Needs to make sure you don't have collisions between function calls
- Execute expensive clean-up code per variable when the function returns

Active management of local variables

Assuming we don't know better, let's ask the memory manager for space each time we create a variable

- Requires setup code to be executed every function call for every variable
  - Would need to check: where have I got space? Is the memory fragmented?
- Needs to make sure you don't have collisions between function calls
- Execute expensive clean-up code per variable when the function returns

There's such a manager for heap variables, but those are usually somewhat long-lived! We can do better for the local variables.

Function calls

Realise that function calls are like a ladder

Function calls

Realise that function calls are like a ladder

- You can go up a step on the ladder (call a function)

Function calls

Realise that function calls are like a ladder

- You can go up a step on the ladder (call a function)
    - And go another step up, when that function calls another function

Function calls

Realise that function calls are like a ladder

- You can go up a step on the ladder (call a function)
  - And go another step up, when that function calls another function
    - And another step up when that function calls another function

Function calls

Realise that function calls are like a ladder

- You can go up a step on the ladder (call a function)
    - And go another step up, when that function calls another function
        - And another step up when that function calls another function
        - Then we step down when we return

Function calls

Realise that function calls are like a ladder

- You can go up a step on the ladder (call a function)
  - And go another step up, when that function calls another function
    - And another step up when that function calls another function
    - Then we step down when we return
  - We step down when we return

Function calls

Realise that function calls are like a ladder

- You can go up a step on the ladder (call a function)
  - And go another step up, when that function calls another function
    - And another step up when that function calls another function
    - Then we step down when we return
  - We step down when we return
- Another step down when we return

Function calls

Realise that function calls are like a ladder

- You can go up a step on the ladder (call a function)
  - And go another step up, when that function calls another function
    - And another step up when that function calls another function
    - Then we step down when we return
  - We step down when we return
- Another step down when we return
- Going sideways is not possible (without multithreading)

Function calls

Realise that function calls are like a ladder

- You can go up a step on the ladder (call a function)
  - And go another step up, when that function calls another function
    - And another step up when that function calls another function
    - Then we step down when we return
  - We step down when we return
- Another step down when we return
- Going sideways is not possible (without multithreading)
- At the bottom of the ladder is `int main()`

Stacking variables

- We use this behaviour to manage our local variables on the stack

Stacking variables

- We use this behaviour to manage our local variables on the stack
- Push your local variables on top of the stack

Stacking variables

- We use this behaviour to manage our local variables on the stack
- Push your local variables on top of the stack
- When you call a function, also push those variables on top of the stack

Stacking variables

- We use this behaviour to manage our local variables on the stack
- Push your local variables on top of the stack
- When you call a function, also push those variables on top of the stack
- When that function returns, just pop off those variables from the stack and they're gone

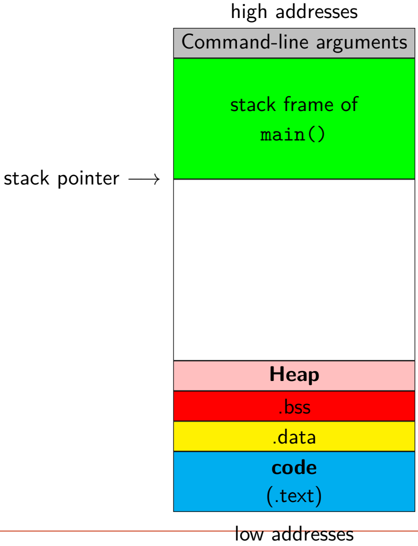
Stacking variables

- We use this behaviour to manage our local variables on the stack
- Push your local variables on top of the stack
- When you call a function, also push those variables on top of the stack
- When that function returns, just pop off those variables from the stack and they're gone
- Only thing to keep track of: where is the top of the stack

Stack frames and the stack pointer

Example:

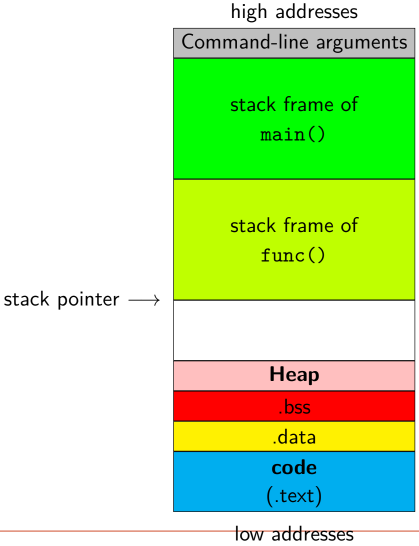
```
int func(int a, int b)
{
  ...
  return 10001;
}

int main(void)
{
  ...
  int x = func(42, 23)
  ...
}
```



Stack frames and the stack pointer

Example:

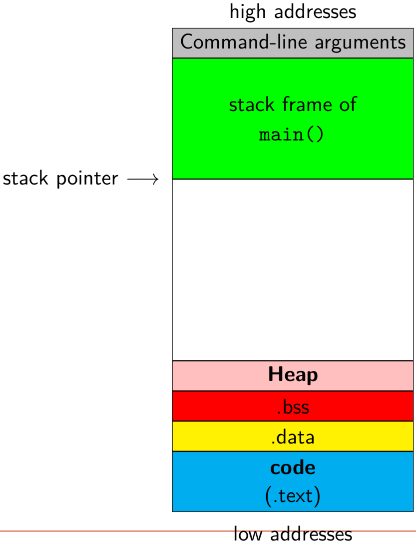
```
int func(int a, int b)
{
  ...
  return 10001;
}

int main(void)
{
  ...
  int x = func(42, 23)
  ...
}
```



Stack frames and the stack pointer

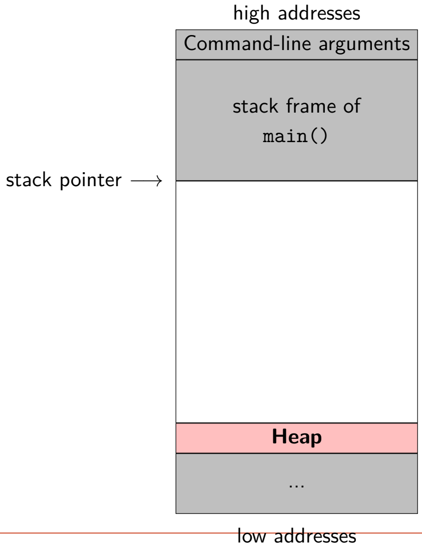
Example:

```
int func(int a, int b)
{
  ...
  return 10001;
}

int main(void)
{
  ...
  int x = func(42, 23)
  ...
}
```



A zoom into the stack frame

- Stack before the function call



high addresses

Command-line arguments

stack frame of `main()`

stack pointer ⟶

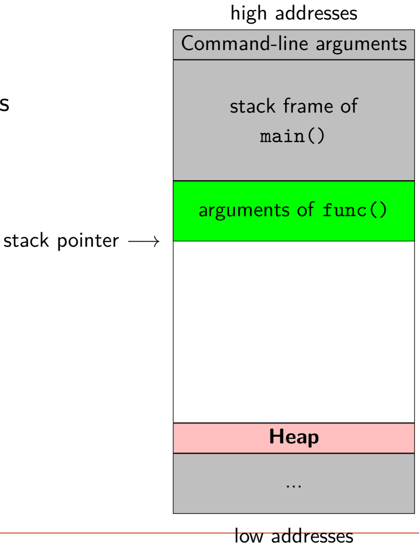
Heap

...

low addresses

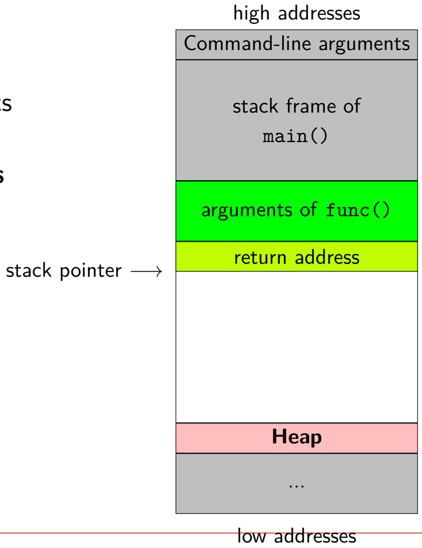
A zoom into the stack frame

- Stack before the function call
- Caller (`main`) first puts arguments for `func` on the stack



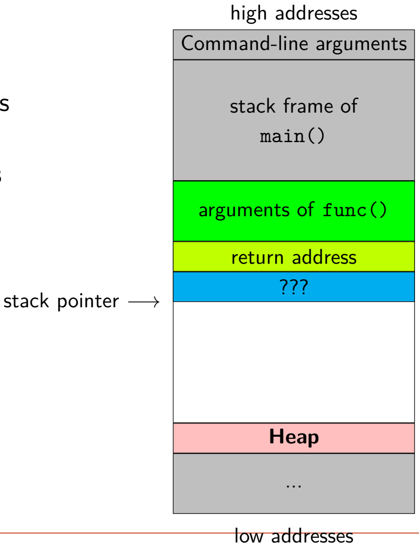
A zoom into the stack frame

- Stack before the function call
- Caller (`main`) first puts arguments for `func` on the stack
- Caller pushes the **return address** onto the stack



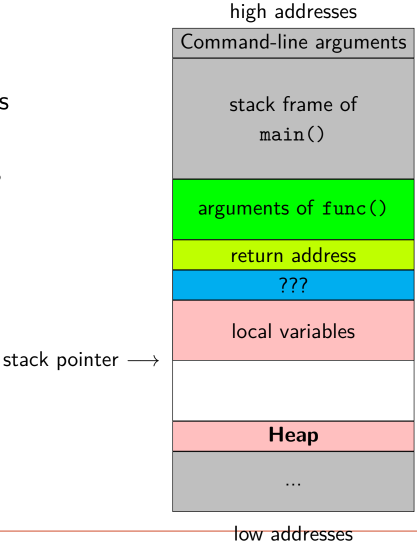
A zoom into the stack frame

- Stack before the function call
- Caller (`main`) first puts arguments for `func` on the stack
- Caller pushes the **return address** onto the stack
- ???



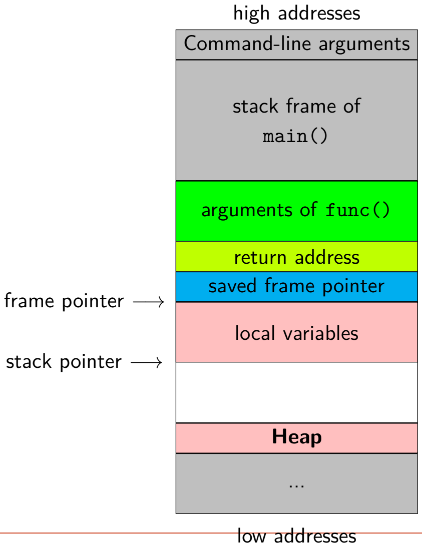
A zoom into the stack frame

- Stack before the function call
- Caller (`main`) first puts arguments for `func` on the stack
- Caller pushes the **return address** onto the stack
- ???
- Callee pushes local variables onto the stack



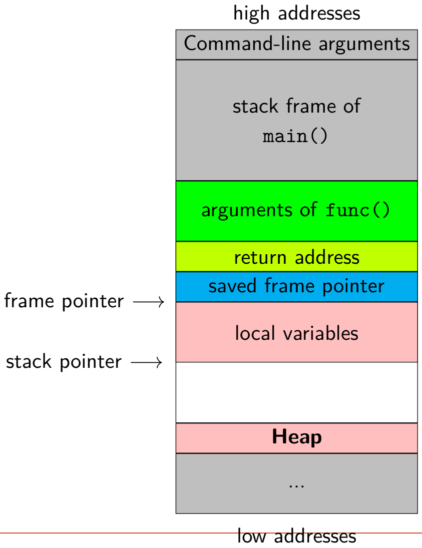
The frame pointer

- So what's with the ???...?



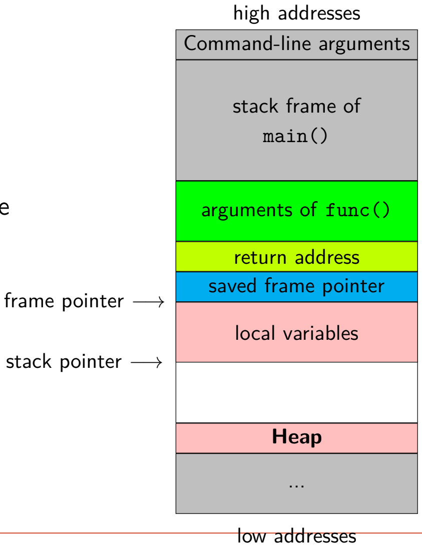
The frame pointer

- So what's with the ???. . . ?
- Traditionally also have an *frame pointer*



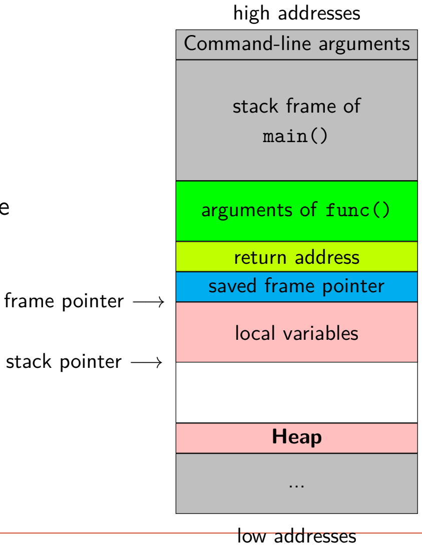
The frame pointer

- So what's with the ???. . . ?
- Traditionally also have an *frame pointer*
- Pointing to the end (high address) of the active stack frame



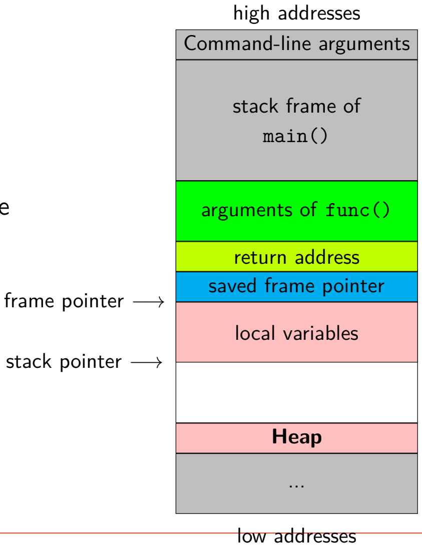
The frame pointer

- So what's with the ???. . . ?
- Traditionally also have an *frame pointer*
- Pointing to the end (high address) of the active stack frame
- On x86 in `ebp` register (AMD64: `rbp`)



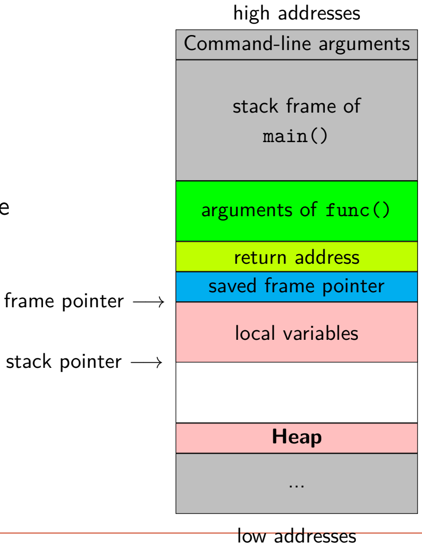
The frame pointer

- So what's with the ???...?
- Traditionally also have an *frame pointer*
- Pointing to the end (high address) of the active stack frame
- On x86 in `ebp` register (AMD64: `rbp`)
- Function call also saves previous frame pointer on the stack



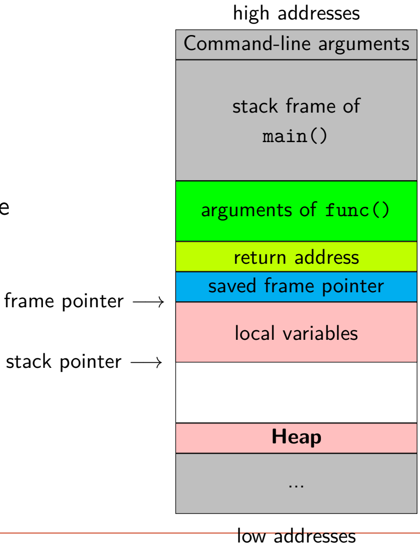
The frame pointer

- So what's with the ???. . . ?
- Traditionally also have an *frame pointer*
- Pointing to the end (high address) of the active stack frame
- On x86 in `ebp` register (AMD64: `rbp`)
- Function call also saves previous frame pointer on the stack
- On AMD64 commonly omitted:



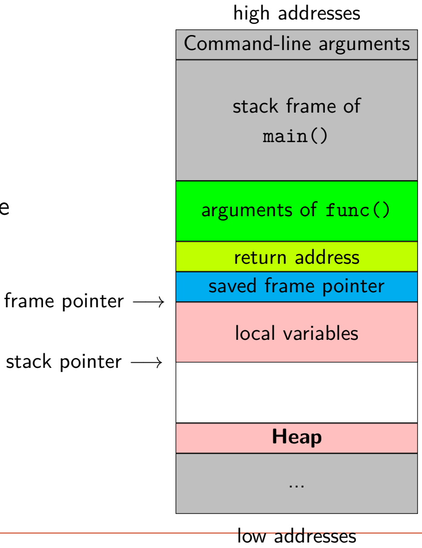
The frame pointer

- So what's with the ???. . . ?
- Traditionally also have an *frame pointer*
- Pointing to the end (high address) of the active stack frame
- On x86 in `ebp` register (AMD64: `rbp`)
- Function call also saves previous frame pointer on the stack
- On AMD64 commonly omitted:
  - Faster function calls



The frame pointer

- So what's with the ???. . . ?
- Traditionally also have an *frame pointer*
- Pointing to the end (high address) of the active stack frame
- On x86 in `ebp` register (AMD64: `rbp`)
- Function call also saves previous frame pointer on the stack
- On AMD64 commonly omitted:
  - Faster function calls
  - One additional register available



Other stuff on the stack

So the other helpful uses of the stack:

- Passing function arguments†

Other stuff on the stack

So the other helpful uses of the stack:

- Passing function arguments†
  - Push them on the stack before jumping to the function

Other stuff on the stack

So the other helpful uses of the stack:

- Passing function arguments†
  - Push them on the stack before jumping to the function
- Keep track of the return address

Other stuff on the stack

So the other helpful uses of the stack:

- Passing function arguments†
  - Push them on the stack before jumping to the function
- Keep track of the return address
  - Push it on the stack

Other stuff on the stack

So the other helpful uses of the stack:

- Passing function arguments†
  - Push them on the stack before jumping to the function
- Keep track of the return address
  - Push it on the stack
  - Return from function: pop local vars, pop arguments, get return address, jump back

Other stuff on the stack

So the other helpful uses of the stack:

- Passing function arguments†
  - Push them on the stack before jumping to the function
- Keep track of the return address
  - Push it on the stack
  - Return from function: pop local vars, pop arguments, get return address, jump back
- Store the return value‡

Other stuff on the stack

So the other helpful uses of the stack:

- Passing function arguments†
  - Push them on the stack before jumping to the function
- Keep track of the return address
  - Push it on the stack
  - Return from function: pop local vars, pop arguments, get return address, jump back
- Store the return value‡
  - Returning from function: pop vars, args, return address, push result, jump back

Other stuff on the stack

So the other helpful uses of the stack:

- Passing function arguments†
  - Push them on the stack before jumping to the function
- Keep track of the return address
  - Push it on the stack
  - Return from function: pop local vars, pop arguments, get return address, jump back
- Store the return value‡
  - Returning from function: pop vars, args, return address, push result, jump back
- Managing the frame pointer

Other stuff on the stack

So the other helpful uses of the stack:

- Passing function arguments†
  - Push them on the stack before jumping to the function
- Keep track of the return address
  - Push it on the stack
  - Return from function: pop local vars, pop arguments, get return address, jump back
- Store the return value‡
  - Returning from function: pop vars, args, return address, push result, jump back
- Managing the frame pointer

† The first 4 (Windows) / 6 (rest) arguments are passed via registers on AMD64 for speed reasons

Other stuff on the stack

So the other helpful uses of the stack:

- Passing function arguments†
  - Push them on the stack before jumping to the function
- Keep track of the return address
  - Push it on the stack
  - Return from function: pop local vars, pop arguments, get return address, jump back
- Store the return value‡
  - Returning from function: pop vars, args, return address, push result, jump back
- Managing the frame pointer

† The first 4 (Windows) / 6 (rest) arguments are passed via registers on AMD64 for speed reasons

‡ Returned via register on x64, ARM, ARMv8 and probably other platforms

Stack overflow

- You're probably aware of `https://stackoverflow.com`
- Named for running out of stack space: a *stack overflow*
- Limits set by:
  - Hardware
  - Operating system
- Get (set) limit on Linux via
  - `ulimit -s` (`ulimit -s kb`) on the shell (sets for the current shell)
  - `getrlimit()` (`setrlimit()`) in C

Common stack bugs

- Stack overflow caused by

Common stack bugs

- Stack overflow caused by
  - (infinite) recursion

Common stack bugs

- Stack overflow caused by
  - (infinite) recursion
  - Creating too-large local variables

Common stack bugs

- Stack overflow caused by
  - (infinite) recursion
  - Creating too-large local variables
- Stack variables are not auto-initialised

Common stack bugs

- Stack overflow caused by
  - (infinite) recursion
  - Creating too-large local variables
- Stack variables are not auto-initialised
  - If you read them, you'll find what previous function call put there

Common stack bugs

- Stack overflow caused by
  - (infinite) recursion
  - Creating too-large local variables
- Stack variables are not auto-initialised
  - If you read them, you'll find what previous function call put there
- The stack mixes program and control data

Common stack bugs

- Stack overflow caused by
  - (infinite) recursion
  - Creating too-large local variables
- Stack variables are not auto-initialised
  - If you read them, you'll find what previous function call put there
- The stack mixes program and control data
- Writing beyond buffers may overwrite frame pointers or return addresses

Common stack bugs

- Stack overflow caused by
  - (infinite) recursion
  - Creating too-large local variables
- Stack variables are not auto-initialised
  - If you read them, you'll find what previous function call put there
- The stack mixes program and control data
- Writing beyond buffers may overwrite frame pointers or return addresses
  - Segmentation fault, if you overwrote with garbage

Common stack bugs

- Stack overflow caused by
  - (infinite) recursion
  - Creating too-large local variables
- Stack variables are not auto-initialised
  - If you read them, you'll find what previous function call put there
- The stack mixes program and control data
- Writing beyond buffers may overwrite frame pointers or return addresses
  - Segmentation fault, if you overwrote with garbage
  - A hacked system, if you overwrote with the address of your attack code. . .

. . . how bad is “wrong” exactly?

Home > Automotive Design Center > How To Article

Toyota's killer firmware: Bad design and its consequences

Michael Dunn -October 28, 2013

EDN MOMENT
1st US rocket to reach outer space launches, February 24, 1949

. . . how bad is “wrong” exactly?

Home > Automotive Design Center > How To Article

Toyota's killer firmware: Bad design and its consequences

Michael Dunn -October 28, 2013

EDN MOMENT
1st US rocket to reach outer space launches, February 24, 1949

“On Thursday October 24, 2013, an Oklahoma court ruled against Toyota in a case of unintended acceleration that lead to the death of one the occupants. Central to the trial was the Engine Control Module’s (ECM) firmware.”

What went wrong?

- Critical variables were not mirrored (stored twice)
- Most importantly, result value `TargetThrottleAngle` wasn't mirrored
- Also critical data structes of the real-time OS weren't mirrored

What went wrong?

- Critical variables were not mirrored (stored twice)
- Most importantly, result value `TargetThrottleAngle` wasn't mirrored
- Also critical data structes of the real-time OS weren't mirrored
- Stack overflow
  - Toyota claimed stack upper bound of 41% of total memory
  - Stack was actually using 94% of total memory
  - Analysis ignored stack used by some 350 assembly functions

What went wrong?

- Critical variables were not mirrored (stored twice)
- Most importantly, result value `TargetThrottleAngle` wasn't mirrored
- Also critical data structes of the real-time OS weren't mirrored
- Stack overflow
  - Toyota claimed stack upper bound of 41% of total memory
  - Stack was actually using 94% of total memory
  - Analysis ignored stack used by some 350 assembly functions
- Code used recursion (forbidden by MISRA-C guidelines)
- MISRA-C: guidelines by the Motor Industry Software Reliability Association

What went wrong?

- Critical variables were not mirrored (stored twice)
- Most importantly, result value `TargetThrottleAngle` wasn't mirrored
- Also critical data structes of the real-time OS weren't mirrored
- Stack overflow
  - Toyota claimed stack upper bound of 41% of total memory
  - Stack was actually using 94% of total memory
  - Analysis ignored stack used by some 350 assembly functions
- Code used recursion (forbidden by MISRA-C guidelines)
- MISRA-C: guidelines by the Motor Industry Software Reliability Association

"A litany of other faults were found in the code, including buffer overflow, unsafe casting, and race conditions between tasks."

Limitations of the stack

```
int* table_of(int num, int len) {
  int table[len];
  for (int i=0; i <= len; i++) {
    table[i] = i * num;
  }
  return table; /* an int[] can be used as an int* */
}
```

What happens if we call this function as follows?:

```
int *table3 = table_of(3,10);
printf("5 times 3 is %d \n", table3[5]);
```

Limitations of the stack

```
int* table_of(int num, int len) {
  int table[len];
  for (int i=0; i <= len; i++) {
    table[i] = i * num;
  }
  return table; /* an int[] can be used as an int* */
}
```

What happens if we call this function as follows?:

```
int *table3 = table_of(3,10);
printf("5 times 3 is %d \n", table3[5]);
```

- The stack cannot preserve data beyond `return` of a function.
- Except of course of returned *data* (not pointers!)

Limitations of the stack

```
int* table_of(int num, int len) {
  int table[len];
  for (int i=0; i <= len; i++) {
    table[i] = i * num;
  }
  return table; /* an int[] can be used as an int* */
}
```

What happens if we call this function as follows?:

```
int *table3 = table_of(3,10);
printf("5 times 3 is %d \n", table3[5]);
```

- The stack cannot preserve data beyond `return` of a function.
- Except of course of returned *data* (not pointers!)
- Obvious other limitation: size!

Table of Contents

The heap

- Think about the heap as a large piece of scrap paper
- We can request (large) continuous space on the piece of paper
- Note that "continuous" is easily ensured by virtual memory

The heap

- Think about the heap as a large piece of scrap paper
- We can request (large) continuous space on the piece of paper
- Note that “continuous” is easily ensured by virtual memory
- This space is accessible through a pointer
- Space remains valid across function calls
- Every function that “knows” a pointer to the space can use it

`malloc`

- Function to request space: `void *malloc(size_t nbytes)`
- Need to `#include <stdlib.h>` to use `malloc`
- `size_t` is an unsigned integer type

`malloc`

- Function to request space: `void *malloc(size_t nbytes)`
- Need to `#include <stdlib.h>` to use `malloc`
- `size_t` is an unsigned integer type
- Returns a `void` pointer to `nbytes` of memory
- Can also fail, in that case, it returns `NULL`

`malloc`

- Function to request space: `void *malloc(size_t nbytes)`
- Need to `#include <stdlib.h>` to use `malloc`
- `size_t` is an unsigned integer type
- Returns a `void` pointer to `nbytes` of memory
- Can also fail, in that case, it returns `NULL`
- Usually pointers in C are typed, `void *x` is an “untyped” pointer
- A `void *` implicitly casts to and from any other pointer type
- Remember that this is *not* the case in C++!

`malloc`

- Function to request space: `void *malloc(size_t nbytes)`
- Need to `#include <stdlib.h>` to use `malloc`
- `size_t` is an unsigned integer type
- Returns a `void` pointer to `nbytes` of memory
- Can also fail, in that case, it returns `NULL`
- Usually pointers in C are typed, `void *x` is an “untyped” pointer
- A `void *` implicitly casts to and from any other pointer type
- Remember that this is *not* the case in C++!
- Example of `malloc` usage:
  ```
  int *x = malloc(10000 * sizeof(int));
  ```
- Request for space for 10 000 integers on the heap

NULL

- The value `NULL` is guaranteed to not point to a valid address
- The following code produces **undefined behavior**:

```
int *x = NULL;
int i = *x;
```

`NULL`

- The value `NULL` is guaranteed to not point to a valid address
- The following code produces **undefined behavior**:

```
int *x = NULL;
int i = *x;
```

- Important to note: `NULL` is not the same as `0`

NULL

- The value `NULL` is guaranteed to not point to a valid address
- The following code produces **undefined behavior**:

```
int *x = NULL;
int i = *x;
```

- Important to note: `NULL` is not the same as `0`
- In boolean expressions, `NULL` evaluates to false
- These two lines have the same semantics:

```
if(x == NULL) { printf("NULL\n"); }
if(!x) { printf("NULL\n"); }
```

ALWAYS check for malloc failure!

- The following code is terribly unsafe:

```
int *table = malloc(TABLESIZE * sizeof(int));
for(size_t i=0;i<TABLESIZE;i++){
  table[i] = 42;
}
```

ALWAYS check for malloc failure!

- The following code is terribly unsafe:

```
int *table = malloc(TABLESIZE * sizeof(int));
for(size_t i=0;i<TABLESIZE;i++){
  table[i] = 42;
}
```

ALWAYS check for malloc failure!

- The following code is terribly unsafe:

  ```
  int *table = malloc(TABLESIZE * sizeof(int));
  for(size_t i=0;i<TABLESIZE;i++){
    table[i] = 42;
  }
  ```
- `malloc` might return `NULL`
- `table[i]` dereferences the pointer `table`
- Consequence: **undefined behavior!**

ALWAYS check for malloc failure!

- The following code is terribly unsafe:

  ```
  int *table = malloc(TABLESIZE * sizeof(int));
  for(size_t i=0;i<TABLESIZE;i++){
    table[i] = 42;
  }
  ```
- `malloc` might return `NULL`
- `table[i]` dereferences the pointer `table`
- Consequence: **undefined behavior!**
- Correct version:

  ```
  int *table = malloc(TABLESIZE * sizeof(int));
  if(table == NULL) exit(255);
  for(size_t i=0;i<TABLESIZE;i++)
  table[i] = 42;
  ```

ALWAYS check for malloc failure!

- The following code is terribly unsafe:

  ```
  int *table = malloc(TABLESIZE * sizeof(int));
  for(size_t i=0;i<TABLESIZE;i++){
    table[i] = 42;
  }
  ```

- `malloc` might return `NULL`
- `table[i]` dereferences the pointer `table`
- Consequence: **undefined behavior!**
- Correct version:

  ```
  int *table = malloc(TABLESIZE * sizeof(int));
  if(table == NULL) exit(255);
  for(size_t i=0;i<TABLESIZE;i++)
  table[i] = 42;
  ```

- Could alternatively use boolean behavior of `NULL`:

  ```
  if(!table) exit(255);
  ```

`free`

- You, the programmer, are in charge of *releasing* memory!
- When you don't need some allocated memory anymore, use `free(x);`
- Here, `x` is a pointer to previously `malloc`'ed memory

`free`

- You, the programmer, are in charge of *releasing* memory!
- When you don't need some allocated memory anymore, use `free(x);`
- Here, `x` is a pointer to previously `malloc`'ed memory
- Typical usage patters:

```
int *x = malloc(NUMX * sizeof(int));
if(x == NULL) { exit(-1); }
...
free(x);
```

- The calls to `malloc` and `free` can be in different functions

`free`

- You, the programmer, are in charge of *releasing* memory!
- When you don't need some allocated memory anymore, use `free(x);`
- Here, `x` is a pointer to previously `malloc`'ed memory
- Typical usage patters:

```
int *x = malloc(NUMX * sizeof(int));
if(x == NULL) { exit(-1); }
...
free(x);
```

- The calls to `malloc` and `free` can be in different functions
- Not freeing `malloc`'ed memory is known as a *memory leak*

`free`

- You, the programmer, are in charge of *releasing* memory!
- When you don't need some allocated memory anymore, use `free(x);`
- Here, `x` is a pointer to previously `malloc`'ed memory
- Typical usage patters:

```
int *x = malloc(NUMX * sizeof(int));
if(x == NULL) { exit(-1); }
...
free(x);
```

- The calls to `malloc` and `free` can be in different functions
- Not freeing `malloc`'ed memory is known as a *memory leak*
- Can be super annoying to debug

`realloc`

- Sometimes want to *expand* or *shrink* `malloc`'ed space
- Do this by using
  `void *realloc(void *ptr, size_t new_size);`
- Content in the allocated area is preserved
- New space is created (or cut away) “at the end”

`realloc`

- Sometimes want to *expand* or *shrink* `malloc`'ed space
- Do this by using
  `void *realloc(void *ptr, size_t new_size);`
- Content in the allocated area is preserved
- New space is created (or cut away) "at the end"
- This call may also return `NULL`
- If return value is `NULL`, previously allocated memory is not freed!

`realloc`

- Sometimes want to *expand* or *shrink* `malloc`'ed space
- Do this by using
  `void *realloc(void *ptr, size_t new_size);`
- Content in the allocated area is preserved
- New space is created (or cut away) "at the end"
- This call may also return `NULL`
- If return value is `NULL`, previously allocated memory is not freed!
- Usage pattern:

```
xnew = realloc(x, NEWSIZE);
if(xnew == NULL) {
  free(x);
  exit(-1); // or continue with old size of x
} else {
  x = xnew;
}
```

`calloc`

- Remember that data on the stack is not initialized
- Global variables are initialized
- Memory space allocated with `malloc` is *not* initialized

`calloc`

- Remember that data on the stack is not initialized
- Global variables are initialized
- Memory space allocated with `malloc` is *not* initialized
- Alternative: use `calloc`:

  ```
  void *calloc(size_t nitems, size_t size)
  ```
- Request space for `nitems` elements of size `size` each
- Memory space is initialized to zero

`calloc`

- Remember that data on the stack is not initialized
- Global variables are initialized
- Memory space allocated with `malloc` is *not* initialized
- Alternative: use `calloc`:

  ```
  void *calloc(size_t nitems, size_t size)
  ```
- Request space for `nitems` elements of size `size` each
- Memory space is initialized to zero
- Example usage:

  ```
  int *p = calloc(1000, sizeof(int));
  if(p == NULL) { exit(-1); }
  ```
- Request space for 1000 integers, all initialized to zero

`malloc` vs. `calloc`

- Aside from initialization, any difference between
    - `int *p = malloc(nelems*sizeof(int));` and
    - `int *p = calloc(nelems,sizeof(int));`?

`malloc` vs. `calloc`

- Aside from initialization, any difference between
  - `int *p = malloc(nelems*sizeof(int));` and
  - `int *p = calloc(nelems,sizeof(int));`?
- Multiplication `nelems*sizeof(int)` can overflow!
- Result: successful allocation, but of *much less* memory!

`malloc` vs. `calloc`

- Aside from initialization, any difference between
  - `int *p = malloc(nelems*sizeof(int));` and
  - `int *p = calloc(nelems,sizeof(int));`?
- Multiplication `nelems*sizeof(int)` can overflow!
- Result: successful allocation, but of *much less* memory!
- Another difference:
  - `malloc` doesn't guarantee you that you can *use* the memory you requested
  - Linux optimistically grants you the memory
  - Later *access* to this memory may still fail
  - `calloc` gives you memory that is actually "backed" by the OS
  - But if you don't actually use it, it'll slow you down

Heap management

- Remember `free`?:

```
int *p = malloc(1000*sizeof(int));
if(p == NULL) exit(-1);
...
free(p);
```

Heap management

- Remember `free`?:

  ```
  int *p = malloc(1000*sizeof(int));
  if(p == NULL) exit(-1);
  ...
  free(p);
  ```

- Question: How does `free` know, how much memory belongs to a pointer?

Heap management

- Remember `free`?:

```
int *p = malloc(1000*sizeof(int));
if(p == NULL) exit(-1);
...
free(p);
```

- Question: How does `free` know, how much memory belongs to a pointer?
- Answer: `malloc` needs to write this information somewhere
- Obvious location: the heap

Heap management

- Remember `free`?:

```
int *p = malloc(1000*sizeof(int));
if(p == NULL) exit(-1);
...
free(p);
```

- Question: How does `free` know, how much memory belongs to a pointer?
- Answer: `malloc` needs to write this information somewhere
- Obvious location: the heap
- One solution: maintain a table of all malloc'ed addresses and space

Heap management

- Remember `free`?:

  ```
  int *p = malloc(1000*sizeof(int));
  if(p == NULL) exit(-1);
  ...
  free(p);
  ```

- Question: How does `free` know, how much memory belongs to a pointer?
- Answer: `malloc` needs to write this information somewhere
- Obvious location: the heap
- One solution: maintain a table of all malloc'ed addresses and space
- Other solution: write information just before the pointer

Dangling pointers, double-free, ...

- **Never** use a pointer after it has been freed, e.g.,

```
int *x = malloc(SIZEX * sizeof(int));
...
free(x);
...
printf("Let's see what the value of x is now: %p\n", x);
```

- This is **undefined behaviour**

Dangling pointers, double-free, ...

- **Never** use a pointer after it has been freed, e.g.,

```
int *x = malloc(SIZEX * sizeof(int));
...
free(x);
...
printf("Let's see what the value of x is now: %p\n", x);
```

- This is **undefined behaviour**
- Also, never double-free a pointer, e.g.,

```
int *x = malloc(SIZEX * sizeof(int));
...
free(x);
free(x);
```

Dangling pointers, double-free, ...

- **Never** use a pointer after it has been freed, e.g.,

  ```
  int *x = malloc(SIZEX * sizeof(int));
  ...
  free(x);
  ...
  printf("Let's see what the value of x is now: %p\n", x);
  ```
- This is **undefined behaviour**
- Also, never double-free a pointer, e.g.,

  ```
  int *x = malloc(SIZEX * sizeof(int));
  ...
  free(x);
  free(x);
  ```
- Not always that obvious, you may have *pointer aliases*
- Pointer alias: multiple pointers to the same location

Dangling pointers, double-free, ...

- **Never** use a pointer after it has been freed, e.g.,

```
int *x = malloc(SIZEX * sizeof(int));
...
free(x);
...
printf("Let's see what the value of x is now: %p\n", x);
```

- This is **undefined behaviour**
- Also, never double-free a pointer, e.g.,

```
int *x = malloc(SIZEX * sizeof(int));
...
free(x);
free(x);
```

- Not always that obvious, you may have *pointer aliases*
- Pointer alias: multiple pointers to the same location
- Never "lose" the last pointer to a location
- This inevitable creates a memory leak: you *cannot* `free` anymore!

Table of Contents

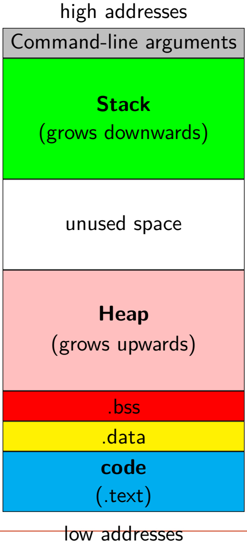
Memory segments

We now covered the stack and the heap, the most important segments, but there are more

Memory segments

We now covered the stack and the heap, the most important segments, but there are more

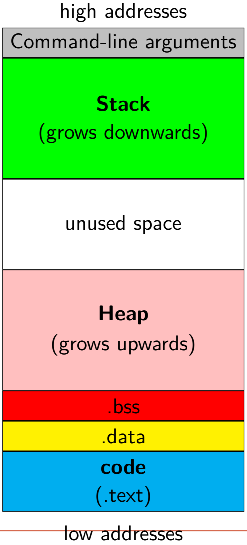
- stack: for local variables (including command-line arguments)



Memory segments

We now covered the stack and the heap, the most important segments, but there are more

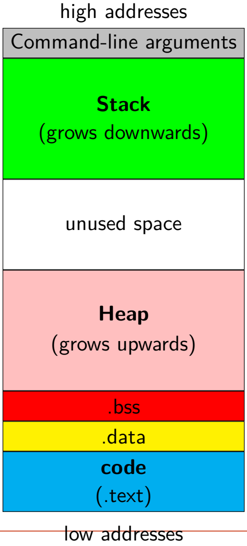
- stack: for local variables (including command-line arguments)
- heap: For *dynamic* memory



Memory segments

We now covered the stack and the heap, the most important segments, but there are more

- stack: for local variables (including command-line arguments)
- heap: For *dynamic* memory
- data segment:


high addresses

Command-line arguments

Stack
(grows downwards)

unused space

Heap
(grows upwards)

.bss

.data

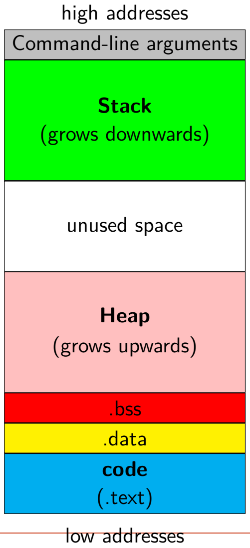
code
(.text)

low addresses

Memory segments

We now covered the stack and the heap, the most important segments, but there are more

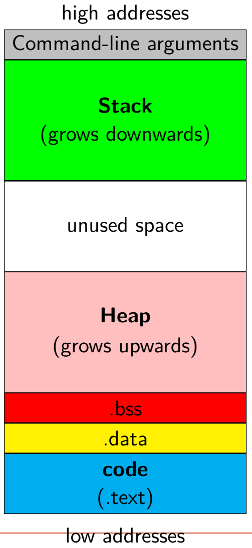
- stack: for local variables (including command-line arguments)
- heap: For *dynamic* memory
- data segment:
    - global and static uninitialized variables (`.bss`)



Memory segments

We now covered the stack and the heap, the most important segments, but there are more

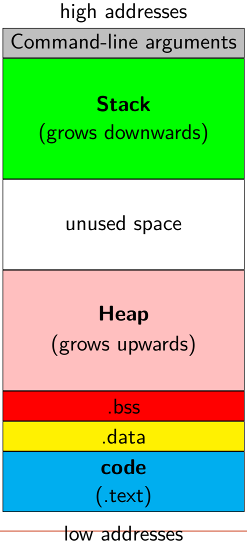
- stack: for local variables (including command-line arguments)
- heap: For *dynamic* memory
- data segment:
  - global and static uninitialized variables (`.bss`)
  - global and static initialized variables (`.data`)



Memory segments

We now covered the stack and the heap, the most important segments, but there are more

- stack: for local variables (including command-line arguments)
- heap: For *dynamic* memory
- data segment:
  - global and static uninitialized variables (`.bss`)
  - global and static initialized variables (`.data`)
- code segment: code (and possibly constants)



Global variables

- Global variables are declared outside of all functions
- Example:

```
#include <stdio.h>
long n = 12345678;
char *s = "hello world!\n";
int a[256];
...
```

- The initialized variables `n` and `s` will be in `.data`
- The uninialized variable `a` will be in `.bss`

Global variables

- Global variables are declared outside of all functions
- Example:

```
#include <stdio.h>
long n = 12345678;
char *s = "hello world!\n";
int a[256];
...
```

- The initialized variables `n` and `s` will be in `.data`
- The uninialized variable `a` will be in `.bss`
- The `.bss` section is typically initialized to zero
  - Otherwise you'd be able to read what was left there by another process

Global variables

- Global variables are declared outside of all functions
- Example:

```
#include <stdio.h>
long n = 12345678;
char *s = "hello world!\n";
int a[256];
...
```

- The initialized variables `n` and `s` will be in `.data`
- The uninialized variable `a` will be in `.bss`
- The `.bss` section is typically initialized to zero
  - Otherwise you'd be able to read what was left there by another process
- Some platforms have a special non-initialized `.bss` subsection
- Example: AVR microcontrollers with a `.noinit` section

Global variables

- Global variables are declared outside of all functions
- Example:

```
#include <stdio.h>
long n = 12345678;
char *s = "hello world!\n";
int a[256];
...
```

- The initialized variables `n` and `s` will be in `.data`
- The uninialized variable `a` will be in `.bss`
- The `.bss` section is typically initialized to zero
  - Otherwise you'd be able to read what was left there by another process
- Some platforms have a special non-initialized `.bss` subsection
- Example: AVR microcontrollers with a `.noinit` section
  - Typically your processes on such a device don't have secrets from each other because you wrote all of them.

Static variables

- A static variable is local, but keeps its value across calls
- Example:

```
void f()
{
  static int x = 0;
  printf("%d\n", x++);
}
```

- If `x` was not declared static, this function would always print `0`

Static variables

- A static variable is local, but keeps its value across calls
- Example:

```
void f()
{
  static int x = 0;
  printf("%d\n", x++);
}
```

- If `x` was not declared static, this function would always print `0`

Static variables

- A static variable is local, but keeps its value across calls
- Example:

```
void f()
{
  static int x = 0;
  printf("%d\n", x++);
}
```

- If `x` was not declared static, this function would always print `0`
- Different for static `x`; output increases by one for every call

Static variables

- A static variable is local, but keeps its value across calls
- Example:

```
void f()
{
  static int x = 0;
  printf("%d\n", x++);
}
```

- If `x` was not declared static, this function would always print 0
- Different for static `x`; output increases by one for every call
- Would get the same behavior if `x` was global
- ... but a global `x` could be modified also by other functions

Table of Contents

Stack vs. heap vs. data segment

Data segment

- Data in the data segment exists throughout the whole execution of the program
  - global variables accessible to every function
  - static local variables only accessible to the respective function

Stack vs. heap vs. data segment

Data segment

- Data in the data segment exists throughout the whole execution of the program
  - global variables accessible to every function
  - static local variables only accessible to the respective function

Stack

- Space on the stack *allocated automatically*
- Stack space automatically removed when returning from a function
- Certain risk of overflowing the stack

Stack vs. heap vs. data segment

Data segment

- Data in the data segment exists throughout the whole execution of the program
  - global variables accessible to every function
  - static local variables only accessible to the respective function

Stack

- Space on the stack *allocated automatically*
- Stack space automatically removed when returning from a function
- Certain risk of overflowing the stack

Heap

- Space on the heap needs to be *requested manually* (`malloc`)
- Request may be denied (`NULL` return) and this must be handled
- Space on the heap needs to be *freed manually* (`free`)
- Risk of memory leaks, double frees, etc.

Table of Contents

Remember `printf`?

> `int printf(const char *format, ...);`
>
> *[printf] writes the output under the control of a* ***format string*** *that specifies how subsequent arguments are converted for output.* *src: man 3 printf*

Having fun with `printf`

What does the following program do?

```
// program.c
int main(int argc, char* argv[]) {

    printf(argv[1]);
}
```

```
~$ gcc -Wall -Wextra -Wpedantic -o program program.c
(gcc8 complains **only** about unused variable argc)
~$ ./program hi
hi
```

Having fun with `printf`

What does the following program do *wrongly*?

```
// program.c
int main(int argc, char* argv[]) {

    printf(argv[1]);
}
```

```
~$ gcc -Wall -Wextra -Wpedantic -o program program.c
(gcc8 complains **only** about unused variable argc)
~$ ./program hi
hi
```

Having fun with `printf`

What does the following program do *wrongly*?

```
// program.c
int main(int argc, char* argv[]) {
    // should have been printf("%s", argv[1]);
    printf(argv[1]);
}
```

How do we make this program misbehave?

Having fun with `printf`

What does the following program do *wrongly*?

```
// program.c
int main(int argc, char* argv[]) {
    // should have been printf("%s", argv[1]);
    printf(argv[1]);
}
```

What happens if we run `./program %x`?

Having fun with `printf`

What does the following program do *wrongly*?

```
// program.c
int main(int argc, char* argv[]) {
    // should have been printf("%s", argv[1]);
    printf(argv[1]);
}
```

What happens if we run `./program %x`?
It will print the second argument of `printf`, even if it's not there!

Remember `printf`?

```
int printf(const char *format, ...);
```

*[printf] writes the output under the control of a **format string** that specifies how subsequent arguments are converted for output.* src: `man 3 printf`

Format string attacks

- Reading data known since 1989
- First attack that broke something in 1999
- Remember, C is from 1972!
- Allows to read data from the stack and heap.
- Easy to spot: if there is no `"` after `printf(`, it's suspicious
- If we want compiler warnings from `gcc`, we need to use `-Wformat=2`, because of course why switch this on by default.
- The clang compiler *does* report these by default.

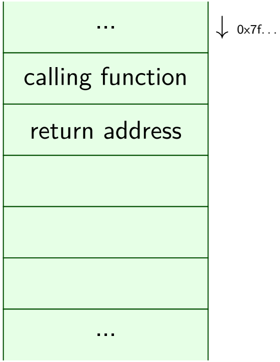
Side-step: calling a function on `x86_64`

If we want to call a function `func(a, b, c, d, e, f, g, h)`, your computer does the following:

Side-step: calling a function on `x86_64`

If we want to call a function `func(a, b, c, d, e, f, g, h)`, your computer does the following:

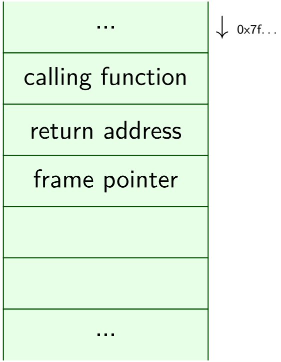
1. Put return address on the stack



Side-step: calling a function on `x86_64`

If we want to call a function `func(a, b, c, d, e, f, g, h)`, your computer does the following:

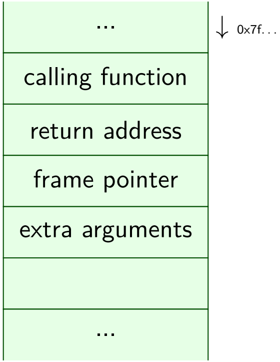
1. Put return address on the stack
2. Store the frame pointer



Side-step: calling a function on `x86_64`

If we want to call a function `func(a, b, c, d, e, f, g, h)`, your computer does the following:

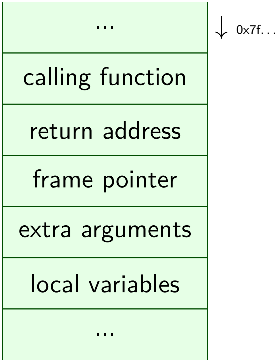
1. Put return address on the stack
2. Store the frame pointer
3. Put the first six arguments (a. . . f) in registers



Side-step: calling a function on `x86_64`

If we want to call a function `func(a, b, c, d, e, f, g, h)`, your computer does the following:

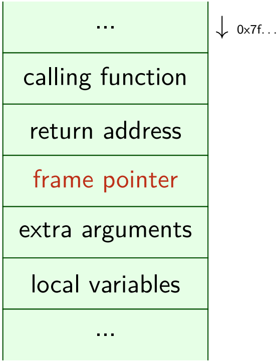
1. Put return address on the stack
2. Store the frame pointer
3. Put the first six arguments (a...f) in registers
4. Put the remaining arguments (g, h) on the stack.



Side-step: calling a function on `x86_64`

If we want to call a function `func(a, b, c, d, e, f, g, h)`, your computer does the following:

1. Put return address on the stack
2. Store the frame pointer
3. Put the first six arguments (a. . . f) in registers
4. Put the remaining arguments (g, h) on the stack.
5. Jump to the address of `func`

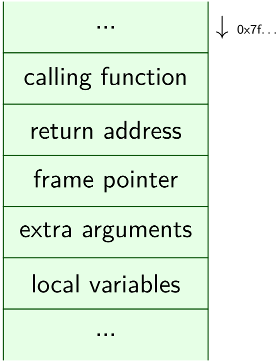


Side-step: calling a function on `x86_64`

If we want to call a function `func(a, b, c, d, e, f, g, h)`, your computer does the following:

1. Put return address on the stack
2. Store the frame pointer
3. Put the first six arguments (a...f) in registers
4. Put the remaining arguments (g, h) on the stack.
5. Jump to the address of `func`

Why do we put arguments into registers?

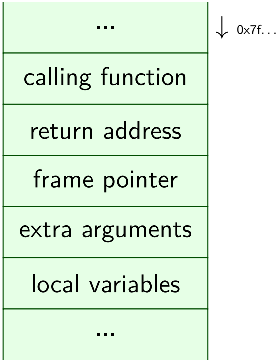


Side-step: calling a function on `x86_64`

If we want to call a function `func(a, b, c, d, e, f, g, h)`, your computer does the following:

1. Put return address on the stack
2. Store the frame pointer
3. Put the first six arguments (a. . . f) in registers
4. Put the remaining arguments (g, h) on the stack.
5. Jump to the address of `func`

Why do we put arguments into registers?



Putting the first few arguments in registers saves a lot of time waiting for memory.

So what do we see?

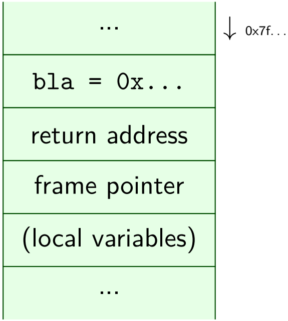
- So if we run `./printf %p`, we will print the value of the second register that would contain an argument.
- If we print `./printf '%7$p'`, we will print the first 8 bytes on the stack.

`printf` is a powerful debugger

```
#include <stdio.h>
void do_print(char* string)
  { printf(string); }

int main(int argc, char** argv) {
  long bla = 0xDEADC0DECAFEF00D;
  do_print(argv[1]);
}
```

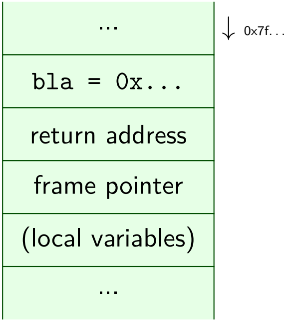
`printf` is a powerful debugger

```
#include <stdio.h>
void do_print(char* string)
  { printf(string); }

int main(int argc, char** argv) {
  long bla = 0xDEADC0DECAFEF00D;
  do_print(argv[1]);
}

./printf "$(perl -e ’print "%p "x14’)"
```



printf is a powerful debugger

```
#include <stdio.h>
void do_print(char* string)
  { printf(string); }

int main(int argc, char** argv) {
  long bla = 0xDEADC0DECAFEF00D;
  do_print(argv[1]);
}
```



…

bla = 0x...

return address

frame pointer

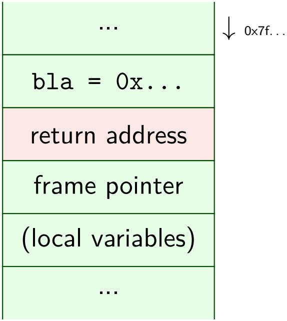
(local variables)

…

```
./printf "$(perl -e ’print "%p "x14’)"
  0x7fffffffe4e8 0x7fffffffe500 0x7ffff7f82578 0x7ffff7f83be0
  0x7ffff7f83be0 (nil) 0x7fffffffe810 0x7fffffffe400 0x555555555199
  0x7fffffffe4e8 0x255555050 0x7fffffffe4e0 0xdeadc0decafef00d
  0x5555555551d0
```

`printf` is a powerful debugger

```
#include <stdio.h>
void do_print(char* string)
  { printf(string); }

int main(int argc, char** argv) {
  long bla = 0xDEADC0DECAFEF00D;
  do_print(argv[1]);
}
```



```
./printf "$(perl -e 'print "%p "x14')"
  0x7fffffffe4e8 0x7fffffffe500 0x7ffff7f82578 0x7ffff7f83be0
  0x7ffff7f83be0 (nil) 0x7fffffffe810 0x7fffffffe400 0x555555555199
  0x7fffffffe4e8 0x255555050 0x7fffffffe4e0 0xdeadc0decafef00d
  0x5555555551d0
```

Table of Contents

What’s wrong with this code (part 1)?

```
int f()
{
  int *a = malloc(100 * sizeof(int));
  if(a == NULL) return -1;
  char *x = (char *)a;
  ...
  free(x);
  free(a);
}
```

What’s wrong with this code (part 1)?

```
int f()
{
  int *a = malloc(100 * sizeof(int));
  if(a == NULL) return -1;
  char *x = (char *)a;
  ...
  free(x);
  free(a);
}
```

What’s wrong with this code (part 1)?

```
int f()
{
  int *a = malloc(100 * sizeof(int));
  if(a == NULL) return -1;
  char *x = (char *)a;
  ...
  free(x);
  free(a);
}
```

- Fairly simple: double-free.

What’s wrong with this code (part 2)?

```
int* f()
{
  int a[100];
  for(i=0;i<100;i++)
    a[i] = i;
  return a;
}
```

What’s wrong with this code (part 2)?

```
int* f()
{
  int a[100];
  for(i=0;i<100;i++)
    a[i] = i;
  return a;
}
```

What's wrong with this code (part 2)?

```
int* f()
{
  int a[100];
  for(i=0;i<100;i++)
    a[i] = i;
  return a;
}
```

- Return type is `int *`, returning a is not a *type* problem
- Remember that an array can "decay" to a pointer to its first element

What’s wrong with this code (part 2)?

```
int* f()
{
  int a[100];
  for(i=0;i<100;i++)
    a[i] = i;
  return a;
}
```

- Return type is `int *`, returning `a` is not a *type* problem
- Remember that an array can “decay” to a pointer to its first element
- Code is syntactically completely correct C
- Returning pointer to a local variable is **undefined behavior**
- Never do this, not even for debugging purposes

What’s wrong with this code (part 2)?

```
int* f()
{
  int a[100];
  for(i=0;i<100;i++)
    a[i] = i;
  return a;
}
```

- Return type is `int *`, returning a is not a *type* problem
- Remember that an array can “decay” to a pointer to its first element
- Code is syntactically completely correct C
- Returning pointer to a local variable is **undefined behavior**
- Never do this, not even for debugging purposes
- Any decent compiler will put out warnings

What’s wrong with this code (part 3)?

```
int f()
{
  int *a = malloc(100 * sizeof(int));
  int x = 5;
  int *y = a;
  a = &x;
  free(a);
  return x;
}
```

What’s wrong with this code (part 3)?

```
int f()
{
  int *a = malloc(100 * sizeof(int));
  int x = 5;
  int *y = a;
  a = &x;
  free(a);
  return x;
}
```

What’s wrong with this code (part 3)?

```
int f()
{
  int *a = malloc(100 * sizeof(int));
  int x = 5;
  int *y = a;
  a = &x;
  free(a);
  return x;
}
```

- No check whether `malloc` returned `NULL`
- The function is *so* wrong, that this isn’t even really a problem

What’s wrong with this code (part 3)?

```
int f()
{
  int *a = malloc(100 * sizeof(int));
  int x = 5;
  int *y = a;
  a = &x;
  free(a);
  return x;
}
```

- No check whether `malloc` returned `NULL`
- The function is *so* wrong, that this isn’t even really a problem
- The `free` is used on a *stack* address

What’s wrong with this code (part 3)?

```
int f()
{
  int *a = malloc(100 * sizeof(int));
  int x = 5;
  int *y = a;
  a = &x;
  free(a);
  return x;
}
```

- No check whether `malloc` returned `NULL`
- The function is *so* wrong, that this isn’t even really a problem
- The `free` is used on a *stack* address
- The value of `y` is lost after `return`
- Cannot `free` the allocated memory anymore

`valgrind`

- Memory bugs are hard to find manually
- They are one of the biggest problems in C code
- Luckily there is tool assistance: `valgrind`

`valgrind`

- Memory bugs are hard to find manually
- They are one of the biggest problems in C code
- Luckily there is tool assistance: `valgrind`
- Run code is a sort of virtual machine, include memory checks
- Muuuuuuch slower than actually running the code, but:
  - Find memory leaks (`malloc` without `free`)
  - Find access to freed memory
  - Find double-free
  - Find branches and memory access depending on uninitialized data

`valgrind`

- Memory bugs are hard to find manually
- They are one of the biggest problems in C code
- Luckily there is tool assistance: `valgrind`
- Run code is a sort of virtual machine, include memory checks
- Muuuuuuch slower than actually running the code, but:
  - Find memory leaks (`malloc` without `free`)
  - Find access to freed memory
  - Find double-free
  - Find branches and memory access depending on uninitialized data
- Many more tools beyond the memory checker in valgrind, e.g.,
  - `cachegrind`, a cache profiler
  - `callgrind`, generating call graphs

`valgrind`

- Memory bugs are hard to find manually
- They are one of the biggest problems in C code
- Luckily there is tool assistance: `valgrind`
- Run code is a sort of virtual machine, include memory checks
- Muuuuuuch slower than actually running the code, but:
  - Find memory leaks (`malloc` without `free`)
  - Find access to freed memory
  - Find double-free
  - Find branches and memory access depending on uninitialized data
- Many more tools beyond the memory checker in valgrind, e.g.,
  - `cachegrind`, a cache profiler
  - `callgrind`, generating call graphs
- `valgrind` is a dynamic analyzer, not static
- For example, no guarantees of branch coverage

`valgrind`

- Memory bugs are hard to find manually
- They are one of the biggest problems in C code
- Luckily there is tool assistance: `valgrind`
- Run code is a sort of virtual machine, include memory checks
- Muuuuuuch slower than actually running the code, but:
  - Find memory leaks (`malloc` without `free`)
  - Find access to freed memory
  - Find double-free
  - Find branches and memory access depending on uninitialized data
- Many more tools beyond the memory checker in valgrind, e.g.,
  - `cachegrind`, a cache profiler
  - `callgrind`, generating call graphs
- `valgrind` is a dynamic analyzer, not static
- For example, no guarantees of branch coverage
- Generally good practice:
  - run your code in valgrind before submitting/publishing
  - make sure that valgrind reports no errors

Sanitizers

- Another way to do these sorts of checks is using `libasan`
- Compile your code with `-fsanitize=address`
- Will slow down your code because it's doing checks all the time
- Will terminate when it finds bad behaviour
- Other sanitizers available
  - `-fsanitize=undefined`
  - `-fsanitize=memory`
  - `-fsanitize=thread`
  - `-fsanitize=leak`
- Not all of them can be used together, some are not useful by themselves.