Introduction To Binary Exploitation

Part 1: The background

The Legal Bit

- The skills taught in these sessions allow identification and exploitation of security vulnerabilities in systems. We strive to give you a place to practice legally, and can point you to other places to practice. These skills should not be used on systems where you do not have explicit permission from the owner of the system. It is **VERY** easy to end up in breach of relevant laws, and we can accept no responsibility for anything you do with the skills learnt here.
- If we have reason to believe that you are utilising these skills against systems where you are not authorised you will be banned from our events, and if necessary the relevant authorities will be alerted.
- Remember, if you have any doubts as to if something is legal or authorised, just don’t do it until you are able to confirm you are allowed to.

Code of Conduct

- Before proceeding past this point you must read and agree our Code of Conduct, this is a requirement from the University for us to operate as a society.
- If you have any doubts or need anything clarified, please ask a member of the committee.
- Breaching the Code of Conduct = immediate ejection and further consequences.

- Code of Conduct can be found at https://wiki.shefesh.com/doku.php?id=code_conduct

Before We Begin

Before We Begin

- We will be assuming use of Linux based operating system
- We will be using the IA-32/x86 (32-bit) architecture
- Don’t worry if you struggle to get some of this
- This presentation will be an information dump, you DO NOT need to remember all of it
- I've tried to cover everything needed for next week, but I may have missed something
- So don't hesitate to ask!
- No exploits today!

Registers

Registers - Overview

- Essentially a variable for the cpu
- Fixed number exist
- Only place maths can be carried out
- Often hold pointers to other memory
- Values will often move between registers and other memory
- 2 main types
- General
- Special

Registers - General Purpose

- x86 has 8, can be broken from 32-bit to 16-bit
- eax - Used for function returns, also specially used in certain maths
- ebx - No special uses, often holds common values for optimisation
- ecx - Sometimes used as a pointer or loop counter
- edx - Often used for short term variables
- esi - Used for pointers, often source data in transfers
- edi - Same as esi, but often for destination data
- ebp - Used as a frame pointer, or general purpose in optimised code
- esp - Pointer to the top (bottom?) of the stack

Registers - Special

- Flags - Each bit has a specific meaning, Indicates something about previous operation
- eip - Stores address of next instruction to execute
- We are interested in EIP for our exploits
- As it controls the programs control flow

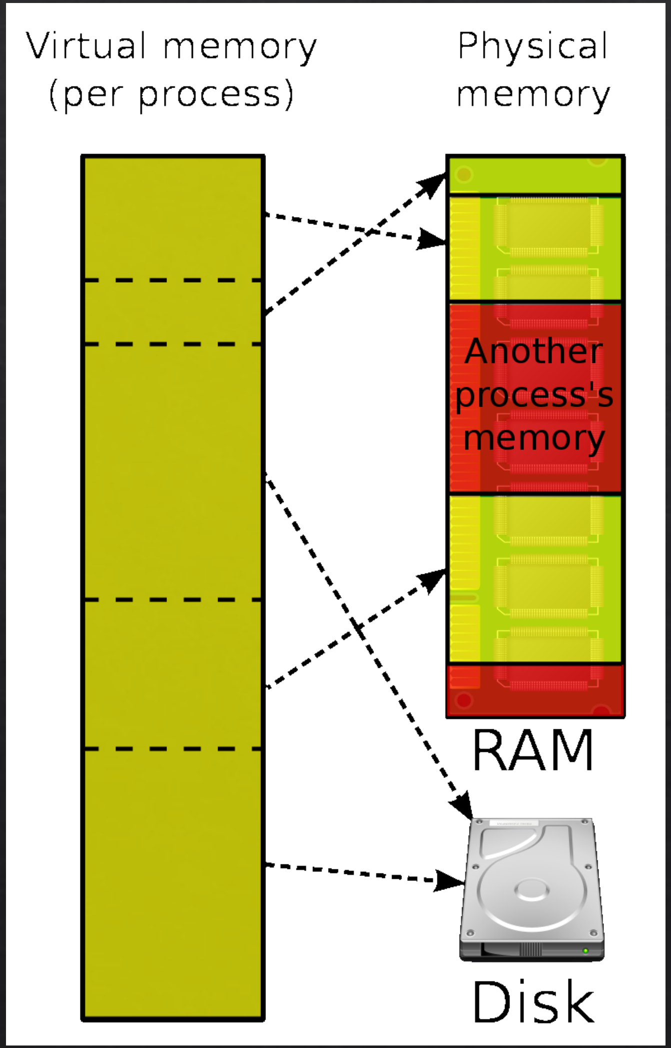
Memory

Memory - Overview

- When we talk about memory, we mean RAM not storage
- When a binary is executed it needs to be loaded into memory
- Stores instructions and data
- We address with hex, e.g 0xbfab15ce
- Operating system maps virtual memory onto physical
- Remember at the end of the day,
- Data, instructions, are all just 0s and 1s (we'll look at them in hex)
- EVERYTHING is just data (example later)

Memory - Physical vs Virtual

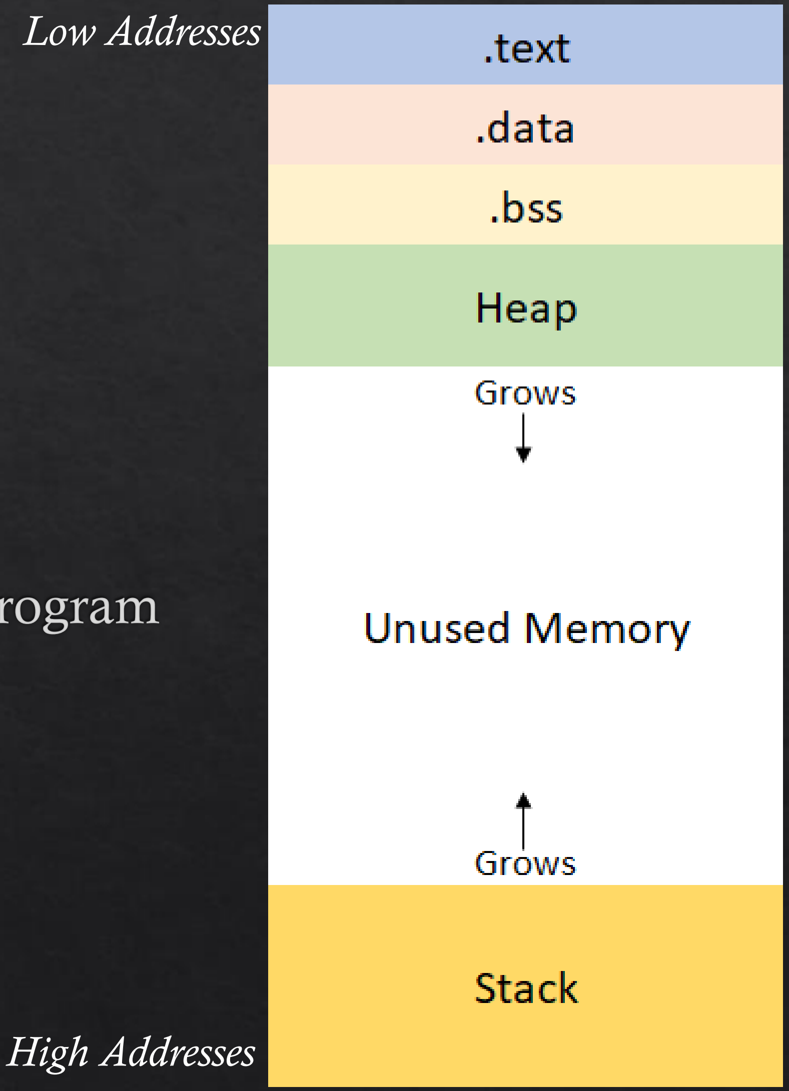
- Physical memory is the memory that actually exists
- Your RAM
- Virtual memory is an abstraction
- Every process is given a mapping of virtual memory
- Gives process illusion of full access to memory
- Often larger than then physical memory
- Uses paging to only load required data into RAM



https://en.wikipedia.org/wiki/Virtual_memory#/media/File:Virtual_memory.svg

Memory - Layout

- Process memory is segmented into various sections
- .text - Basically where a programs code is
- .data - Initialised global variables
- .bss - Uninitalised global variables
- The heap - Where dynamically allocated memory goes
- The stack - Contains details about subroutines (functions) of a program

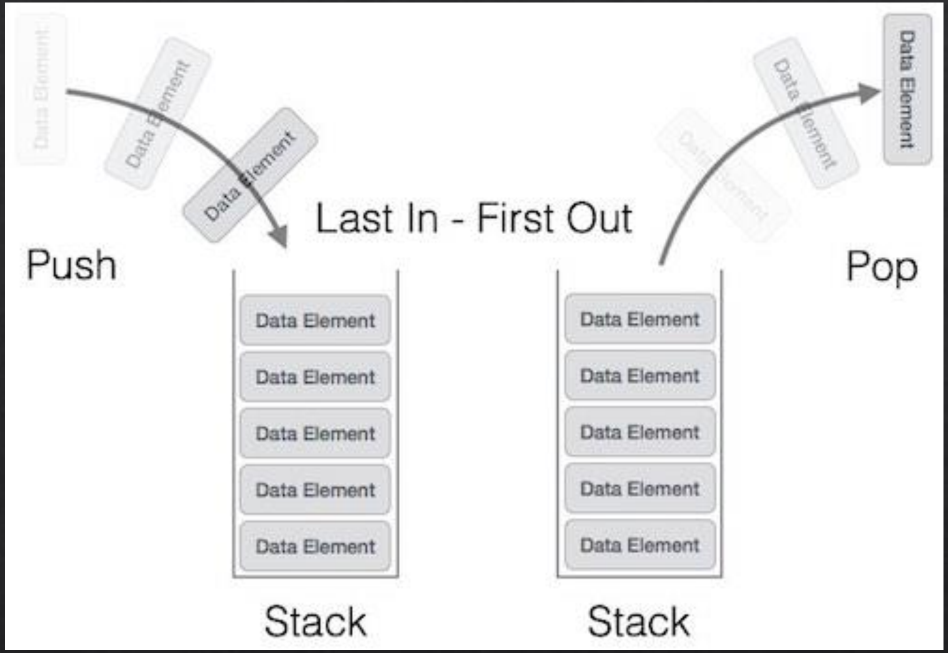


Memory - The Heap

- Used in dynamic memory allocation
- Starts at low addresses and grows to high
- Malloc/free etc (C functions)
- Persists after calling routine
- Forgetting to free() is a primary cause of a memory leak
- Various vulnerabilities utilise the heap
- But not really relevant for us yet!

Memory - The Stack

- Used for data in program functions
- Last in First out (LIFO) structure
- Grows from high memory to low
- Stores local variables
- Made up of various "stack frames"
- The exploits we are looking at use the stack

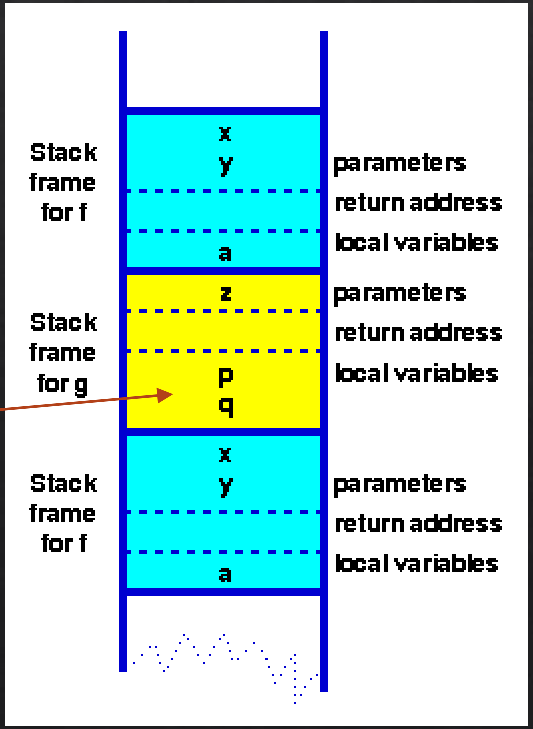


https://www.tutorialspoint.com/data_structures_algorithms/stack_algorithm.htm

Memory - Stack Frames

- Collection of data for one function call
- Includes - return address, argument variables, local variables, saved registers
- The important parts for us
- Local variables and the return address BOTH reside on a stack frame
- The return address ends up in the EIP register!
- We'll see why this is important next week

Note: LIFO means p was declared BEFORE q



https://www.cs.auckland.ac.nz/software/AlgAnim/stacks.html

An Aside - Endianness

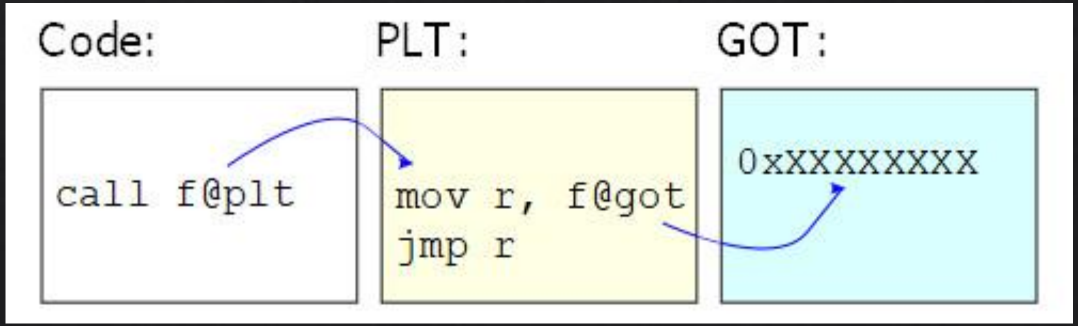
- The order in which bytes are combined into larger values
- Big-endian - Most significant byte is first
- Little-endian - Least significant bytes is first
- We will be using Little-endian
- 0x12345678 is stored as
- 0x78 0x56 0x34 0x12
- Push bytes in reverse order, but DON'T flip the bytes themselves

Note: We are using hex, 0x13 = 19 in decimal. 2 hex digits = 1 byte

Second Note: The 0x is just notation

An Aside - PLT/GOT

- Procedure Link Table / Global Offset Table
- Basically instead of loading all the code into physical memory instantly
- These tables are pointers/offsets to them
- Before code is needed its memory address will be its key in the table
- When its needed that key points it to the real memory!
- Will become more apparent later



https://nullprogram.com/blog/2018/05/27/

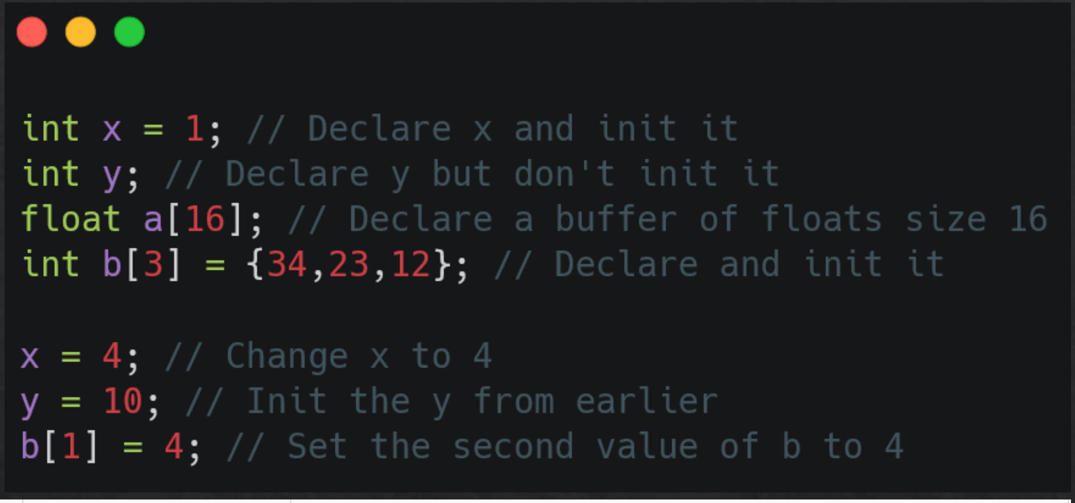
Relevant C

C - Basics

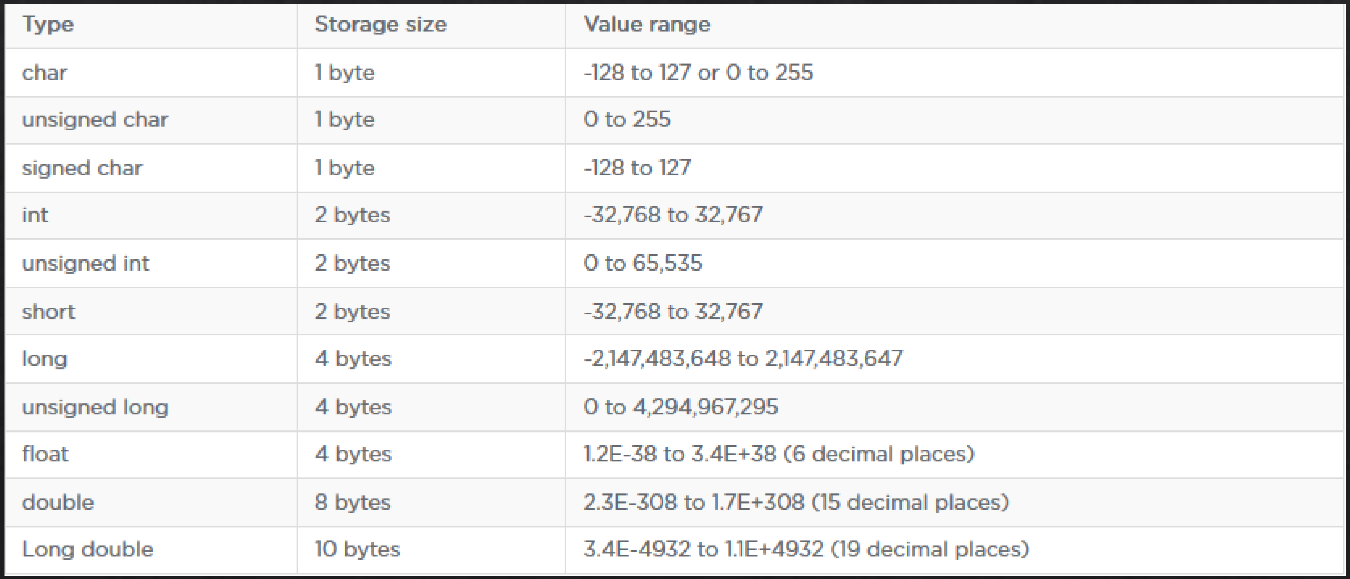
- We only need to understand some basics
- We'll assume knowledge of basics such as if, while etc
- Basically I'm going to assume basic programming knowledge
- Can seem intimidating
- But for our purposes it's fairly simple

C - Variables

- Pretty simple
- Various types
- int, char, float, double
- "Numeric" types can be signed or unsigned
- Also short or long
- Arrays are known as "Buffers"
- Are 0 indexed, last value index = (len-1)
- Various sizes and values



```c
int x = 1; // Declare x and init it
int y; // Declare y but don't init it
float a[16]; // Declare a buffer of floats size 16
int b[3] = {34,23,12}; // Declare and init it

x = 4; // Change x to 4
y = 10; // Init the y from earlier
b[1] = 4; // Set the second value of b to 4
```

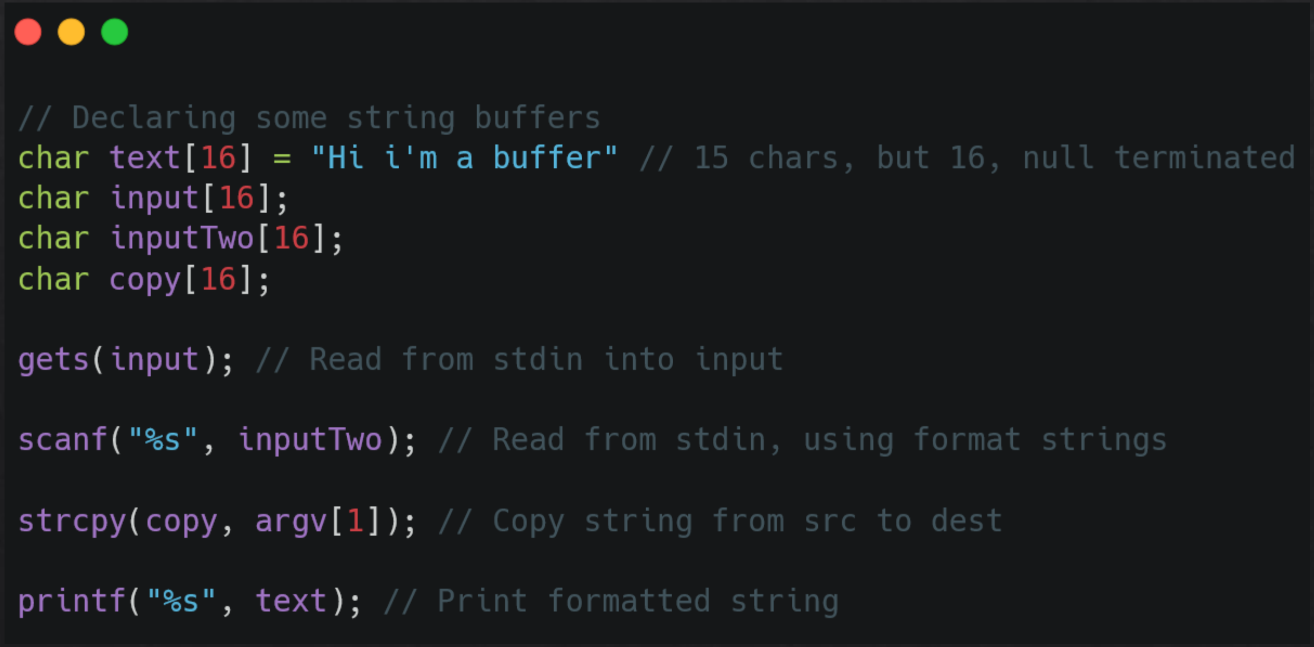


Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 bytes	-32,768 to 32,767
unsigned int	2 bytes	0 to 65,535
short	2 bytes	-32,768 to 32,767
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295
float	4 bytes	1.2E-38 to 3.4E+38 (6 decimal places)
double	8 bytes	2.3E-308 to 1.7E+308 (15 decimal places)
Long double	10 bytes	3.4E-4932 to 1.1E+4932 (19 decimal places)

Table from https://intellipaat.com/tutorial/c-tutorial/c-data-types/

C - Strings

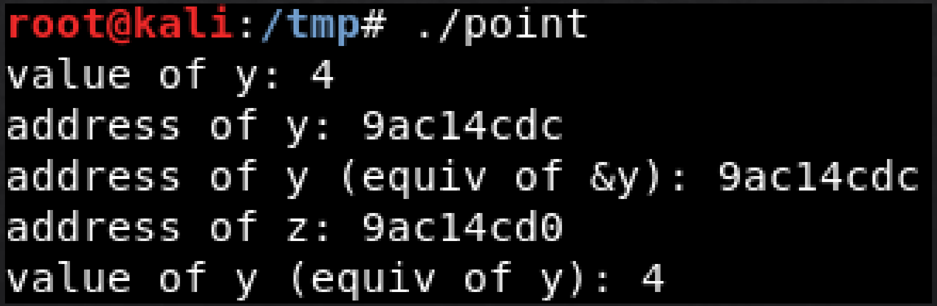
- Declared as a buffer (array) of characters
- Null terminated (\0 or 0x00 in hex)
- So max chars is [Declared size - 1]
- Only certain functions are important right now
- gets - gets(char *str)
- scanf - scanf(const char *format, …)
- strcpy - strcpt(char *dest, const char *src)
- printf - printf(const char *format, …)



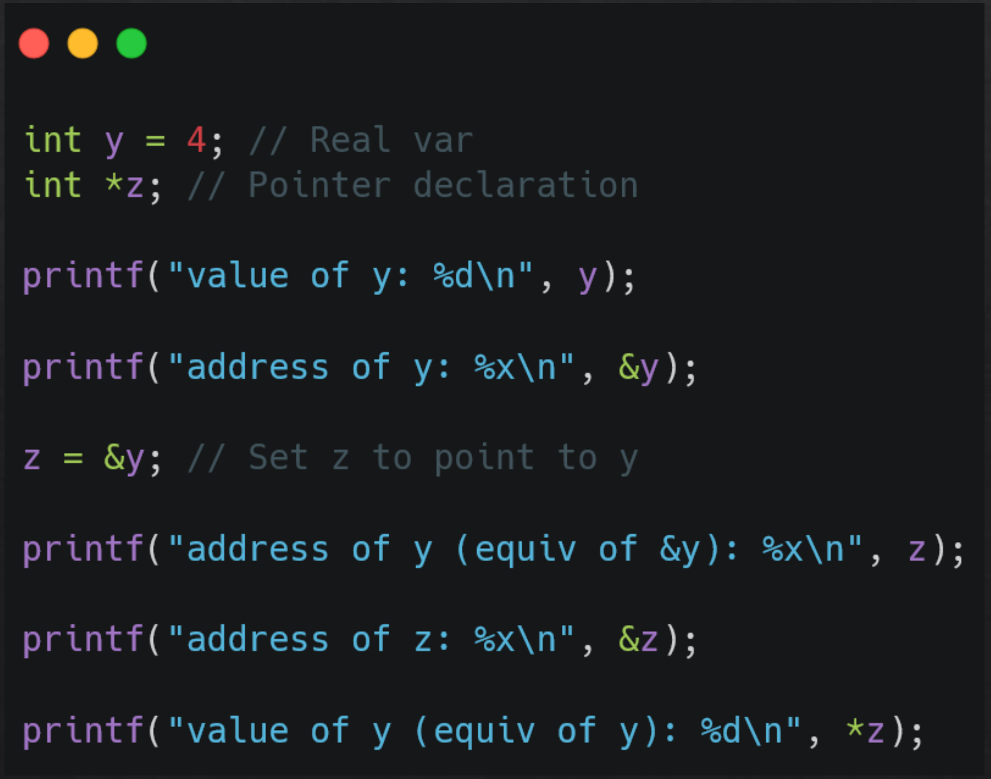
```c
// Declaring some string buffers
char text[16] = "Hi i'm a buffer" // 15 chars, but 16, null terminated
char input[16];
char inputTwo[16];
char copy[16];

gets(input); // Read from stdin into input

scanf("%s", inputTwo); // Read from stdin, using format strings

strcpy(copy, argv[1]); // Copy string from src to dest

printf("%s", text); // Print formatted string
```

C - Pointers

- Variables have memory locations
- Can get the memory address of a variable with &
- Pointers are variables whose value is the address of another variable
- Declared using *



```
root@kali:/tmp# ./point
value of y: 4
address of y: 9ac14cdc
address of y (equiv of &y): 9ac14cdc
address of z: 9ac14cd0
value of y (equiv of y): 4
```



```c
int y = 4; // Real var
int *z; // Pointer declaration

printf("value of y: %d\n", y);

printf("address of y: %x\n", &y);

z = &y; // Set z to point to y

printf("address of y (equiv of &y): %x\n", z);

printf("address of z: %x\n", &z);

printf("value of y (equiv of y): %d\n", *z);
```

x86 Assembly

x86 - Overview

- Assembly is a low level programming language
- Generally architecture specific
- We will be covering some basic assembly calls
- You don’t need to know them by heart, we’re not expecting you to write assembly!
- 2 Major syntaxes
- AT&T vs Intel
- We will use intel
- Because I prefer it

x86 - Intel Syntax

- Destination comes before source, e.g move <dest> <source>
- Parameter size can be derived from name of register
- Effective address arithmetic in square brackets e.g [eax + 4]
- Assembler automatically detects symbol types, e.g registers, constants
- Use suffixes to declare types if cant be derived

x86 - Intel Syntax - Data Formats

- If you need to declare a suffix they are
- byte prt - 1 byte
- word ptr - 2 bytes
- dword ptr - 4 bytes

x86 - Data movement

mov <op1> <op2>

- Copies data from second operand (register/memory contents, or constant) into location represented by first operand (register/memory)
- Cannot copy straight from memory to memory

push <op1>

- Places operand onto top of the stack

pop <op1>

- Removes top item from stack and places it in location given by operand

x86 - Data Movement ctd

lea <op1> <op2>

- Takes address from second operand and places it in register specified by first operand
- Effectively generates a pointer to a memory region

xchg <op1> <op2>

- Swap the contents of destinations specified by operands 1 and 2

x86 - Arithmetic

add <op1> <op2>

- Adds together values given by 2 operands
- Storing result in first operand
- Only one operand may be a memory location

sub <op1> <op2>

- Subtracts value of second operand from first
- Stores result in first
- Only one operand may be a memory location

x86 - Arithmetic ctd

inc/dec <op1>

- Increments/Decrements contents of operand by 1

imul <op1> <op2> <op3>

- First operand must be a register, 3rd operand is optional
- With 2: multiplies op1 and op2, stores in op1
- With 3: multiplies op2 and op3, stores in op1

idiv <op1>

- Treats edx:eax as 64-bit int
- Divides by operand and stores in edx

x86 - Logic

and/or/xor <op1> <op2>

- Carries out the respective logical bitwise action
- Stores result in locations given by op1
- XOR is particularly useful, as xoring a register with itself clears it

not <op1>

- Flips all bit values in operand (1->0, 0->1)

neg <op1>

- Twos complement negation of operand contents

x86 - Logic ctd

shl/shr <op1> <op2>

- Shifts first operands bits left or right
- Amount of bits to shift by specified by second operand
- Pads resulting empty bit positions with 0

x86 - Control Flow

nop

- Does nothing, just move onto the next instruction
- Often used to pad to word boundaries

jmp <op1>

- Sets EIP to address given in operand
- Effectively changes next instruction

j[condition] <op1>

- Based on result of last arithmetic operation
- Conditions include, equal, not equal, was 0, greater than, less than etc
- If condition is met, acts like normal jmp
- Else acts like nop

x86 - Control Flow ctd

cmp <op1> <op2>

- Subtracts values given by op2 from op1 but doesn’t store it
- Effectively just sub without the storage phase
- Commonly used before a j[condition]

call <op1>

- Pushes current location onto stack
- Then jmp to the location given in op1

ret

- Pops the stack and jumps to the location, call/ret are used together to implement subroutines

x86 - Control Flow ctd

int <op1>

- Generates a software interrupt of type dictated by op1
- Important ones are
- int 3 - Used by debuggers to create breakpoints
- int 80 - Triggers a syscall

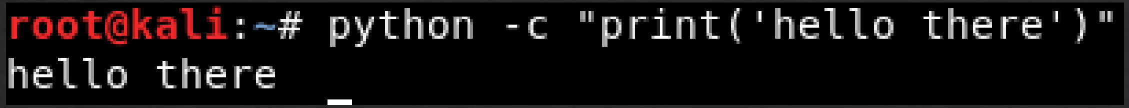
sysenter

- A better (slightly faster) way to trigger a syscall

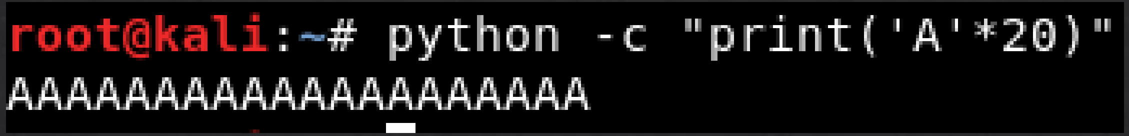
Useful Python

Useful Python

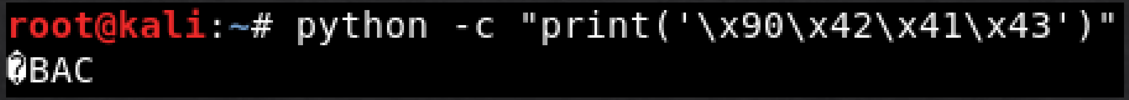
- Run bits of python from the command line -
- **python -c "print('hello there')"**
- Repeat strings
- **print('A'*100)**
- Use hex bytes
- **print('\x90\x42\x41\x43')**
- Useful modules - pwntools, requests, sockets, struct
- Run as args
- **./binary $(python -c "print('arg')")**



```
root@kali:~# python -c "print('hello there')"
hello there
```



```
root@kali:~# python -c "print('A'*20)"
AAAAAAAAAAAAAAAAAAAA
```



```
root@kali:~# python -c "print('\x90\x42\x41\x43')"
�BAC
```

Debugging With GDB

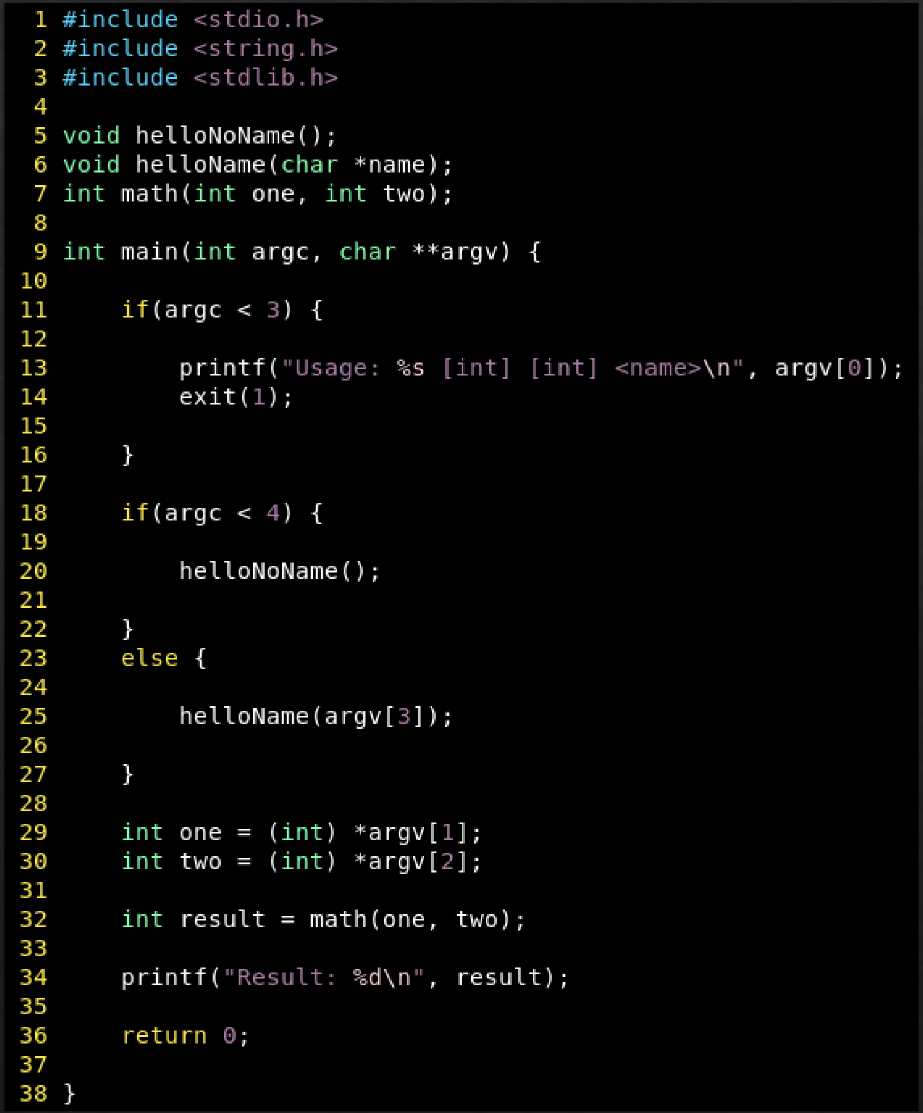
GDB - What is GDB?

- A debugging tool for programs written in many languages
- Can disassemble assembled binaries
- Monitor execution
- Change stuff - modify memory, etc
- Either run the binary with gdb, or attach to already running
- We will run with gdb

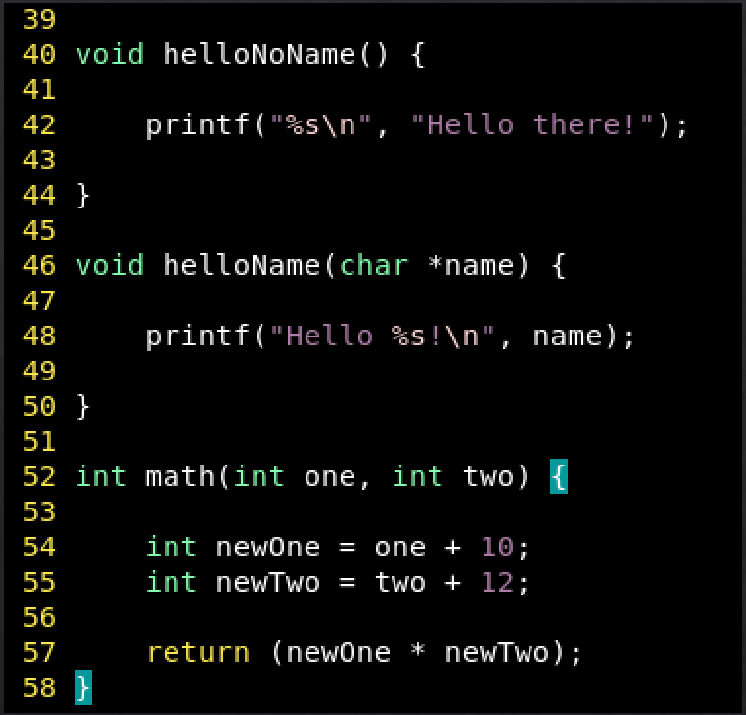
Note: string.h wasn't needed in the final version of this, but I'd already taken screenshots with memory addresses and didn't want them to change

Note on 64-bit kali you need to install "libc6-dev-i386" to compile

GDB - Our Binary



```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void helloNoName();
void helloName(char *name);
int math(int one, int two);

int main(int argc, char **argv) {

    if(argc < 3) {

        printf("Usage: %s [int] [int] <name>\n", argv[0]);
        exit(1);

    }

    if(argc < 4) {

        helloNoName();

    }
    else {

        helloName(argv[3]);

    }

    int one = (int) *argv[1];
    int two = (int) *argv[2];

    int result = math(one, two);

    printf("Result: %d\n", result);

    return 0;

}
```



```c

void helloNoName() {

    printf("%s\n", "Hello there!");

}

void helloName(char *name) {

    printf("Hello %s!\n", name);

}

int math(int one, int two) {

    int newOne = one + 10;
    int newTwo = two + 12;

    return (newOne * newTwo);
}
```

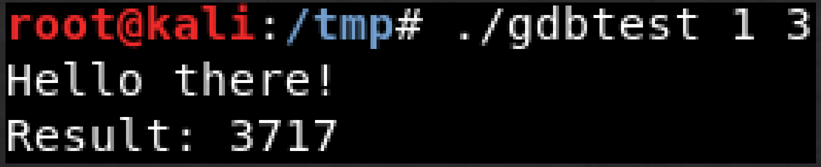
Available at: https://pastebin.com/BGw48aE2

```
gcc gdbtest.c -o gdbtest -fno-stack-protector -z execstack -m32
```

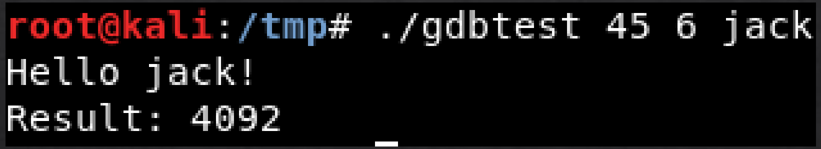
Note: ASLR must also be turned off (this is default in kali)

GDB - Our Binary ctd

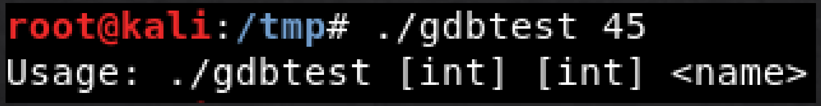
- Lets look at what happens when we run it
- How about with a name this time
- This time lets run it wrong
- What if I don't use an int?
- Remember earlier?
- The example of everything just being data?
- In memory there is no distinction between types!



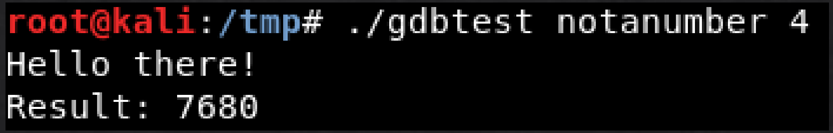
```
root@kali:/tmp# ./gdbtest 1 3
Hello there!
Result: 3717
```



```
root@kali:/tmp# ./gdbtest 45 6 jack
Hello jack!
Result: 4092
```



```
root@kali:/tmp# ./gdbtest 45
Usage: ./gdbtest [int] [int] <name>
```



```
root@kali:/tmp# ./gdbtest notanumber 4
Hello there!
Result: 7680
```

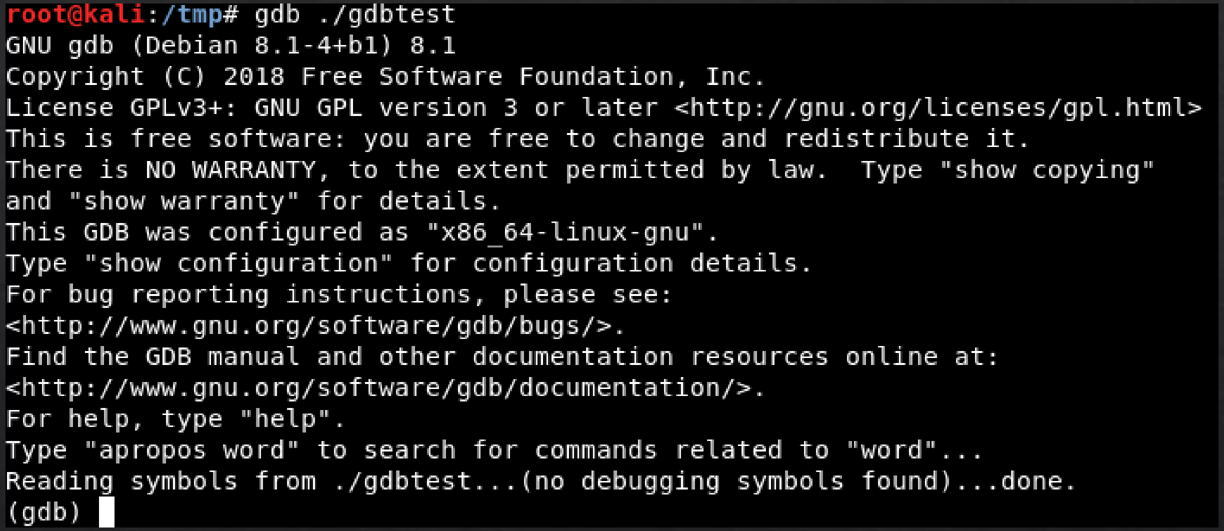
GDB - Opening

- Simply call gdb with the binary as a parameter
- Can also attach to running process
- Easy to run process

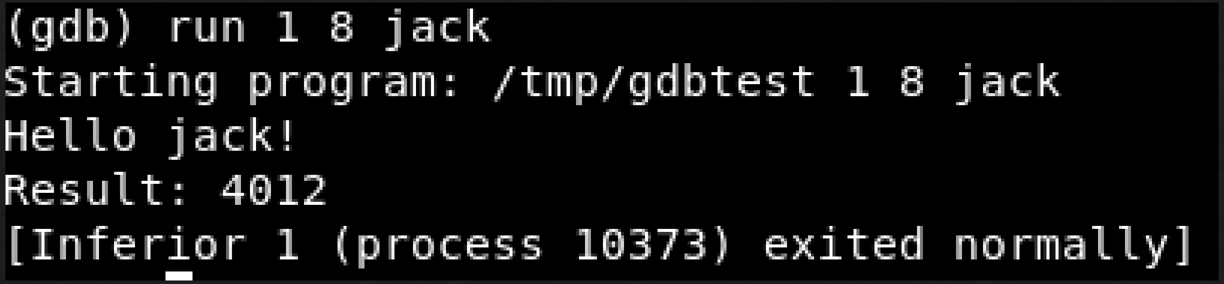
Note: The default gdb option is to use AT&T assembly syntax, you can change this with

set disassembly-flavor intel

I have this in ~/.gdbinit so I wont be doing it live



```
root@kali:/tmp# gdb ./gdbtest
GNU gdb (Debian 8.1-4+b1) 8.1
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./gdbtest...(no debugging symbols found)...done.
(gdb) 
```

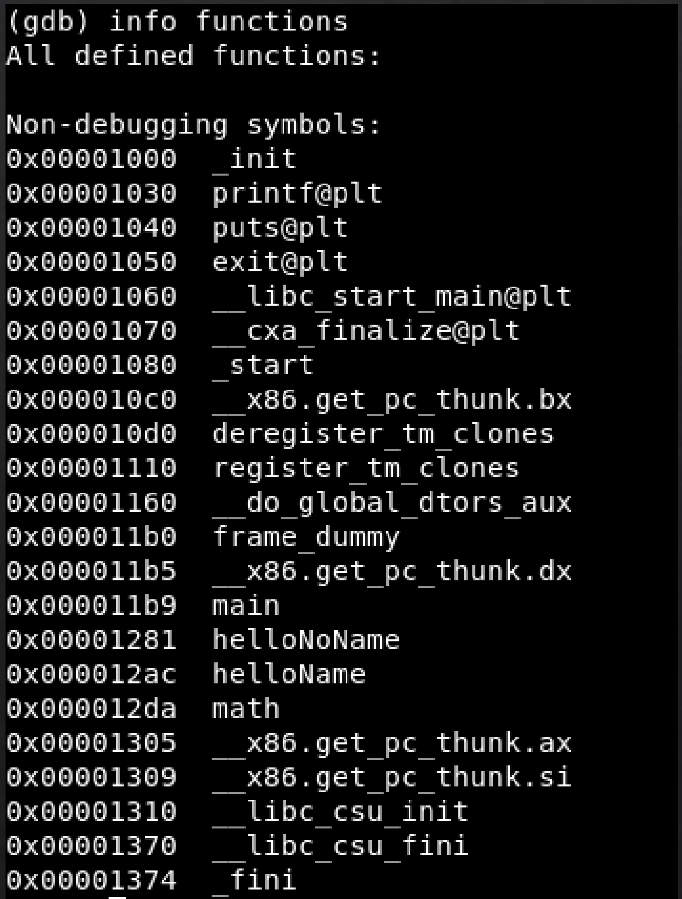


```
(gdb) run 1 8 jack
Starting program: /tmp/gdbtest 1 8 jack
Hello jack!
Result: 4012
[Inferior 1 (process 10373) exited normally]
```

GDB - Inspecting Binaries

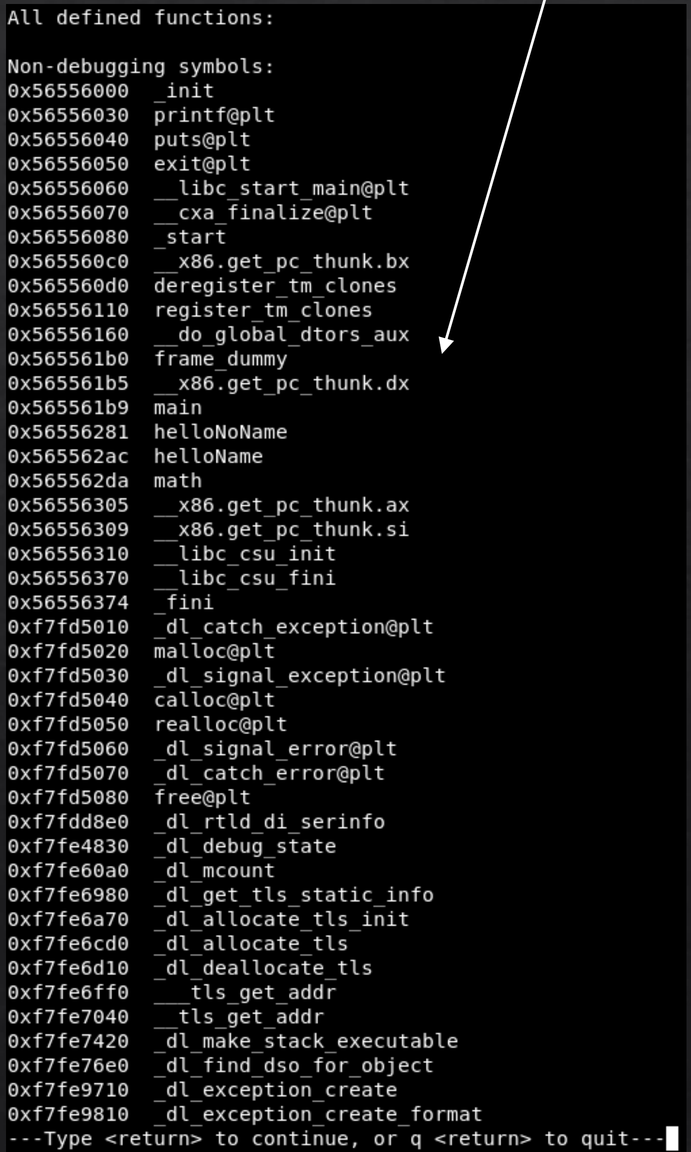
- Look at functions
- But remember, before the binary has been run
- They aren't the true addresses
- After a run
- Note address changes!
- And all the new ones
- Don't worry, most don’t matter

Before a run



```
(gdb) info functions
All defined functions:

Non-debugging symbols:
0x00001000  _init
0x00001030  printf@plt
0x00001040  puts@plt
0x00001050  exit@plt
0x00001060  __libc_start_main@plt
0x00001070  __cxa_finalize@plt
0x00001080  _start
0x000010c0  __x86.get_pc_thunk.bx
0x000010d0  deregister_tm_clones
0x00001110  register_tm_clones
0x00001160  __do_global_dtors_aux
0x000011b0  frame_dummy
0x000011b5  __x86.get_pc_thunk.dx
0x000011b9  main
0x00001281  helloNoName
0x000012ac  helloName
0x000012da  math
0x00001305  __x86.get_pc_thunk.ax
0x00001309  __x86.get_pc_thunk.si
0x00001310  __libc_csu_init
0x00001370  __libc_csu_fini
0x00001374  _fini
```

After a run

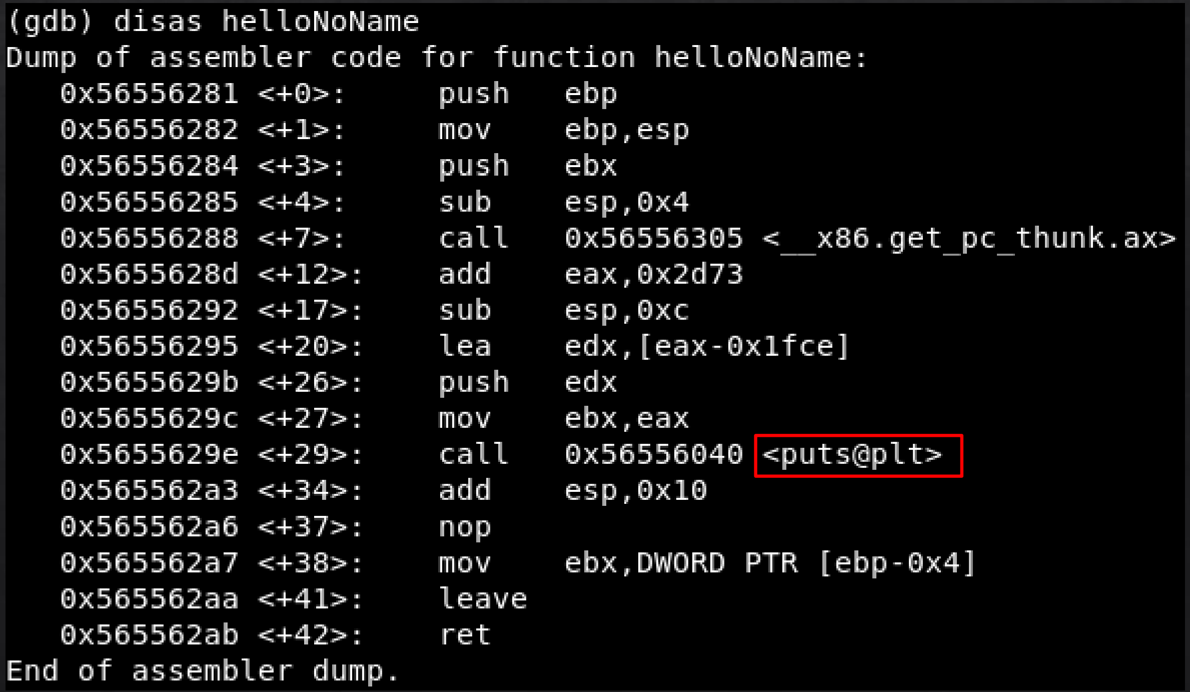


```
All defined functions:

Non-debugging symbols:
0x56556000  _init
0x56556030  printf@plt
0x56556040  puts@plt
0x56556050  exit@plt
0x56556060  __libc_start_main@plt
0x56556070  __cxa_finalize@plt
0x56556080  _start
0x565560c0  __x86.get_pc_thunk.bx
0x565560d0  deregister_tm_clones
0x56556110  register_tm_clones
0x56556160  __do_global_dtors_aux
0x565561b0  frame_dummy
0x565561b5  __x86.get_pc_thunk.dx
0x565561b9  main
0x56556281  helloNoName
0x565562ac  helloName
0x565562da  math
0x56556305  __x86.get_pc_thunk.ax
0x56556309  __x86.get_pc_thunk.si
0x56556310  __libc_csu_init
0x56556370  __libc_csu_fini
0x56556374  _fini
0xf7fd5010  _dl_catch_exception@plt
0xf7fd5020  malloc@plt
0xf7fd5030  _dl_signal_exception@plt
0xf7fd5040  calloc@plt
0xf7fd5050  realloc@plt
0xf7fd5060  _dl_signal_error@plt
0xf7fd5070  _dl_catch_error@plt
0xf7fd5080  free@plt
0xf7fdd8e0  _dl_rtld_di_serinfo
0xf7fe4830  _dl_debug_state
0xf7fe60a0  _dl_mcount
0xf7fe6980  _dl_get_tls_static_info
0xf7fe6a70  _dl_allocate_tls_init
0xf7fe6cd0  _dl_allocate_tls
0xf7fe6d10  _dl_deallocate_tls
0xf7fe6ff0  ___tls_get_addr
0xf7fe7040  __tls_get_addr
0xf7fe7420  _dl_make_stack_executable
0xf7fe76e0  _dl_find_dso_for_object
0xf7fe9710  _dl_exception_create
0xf7fe9810  _dl_exception_create_format
---Type <return> to continue, or q <return> to quit---
```

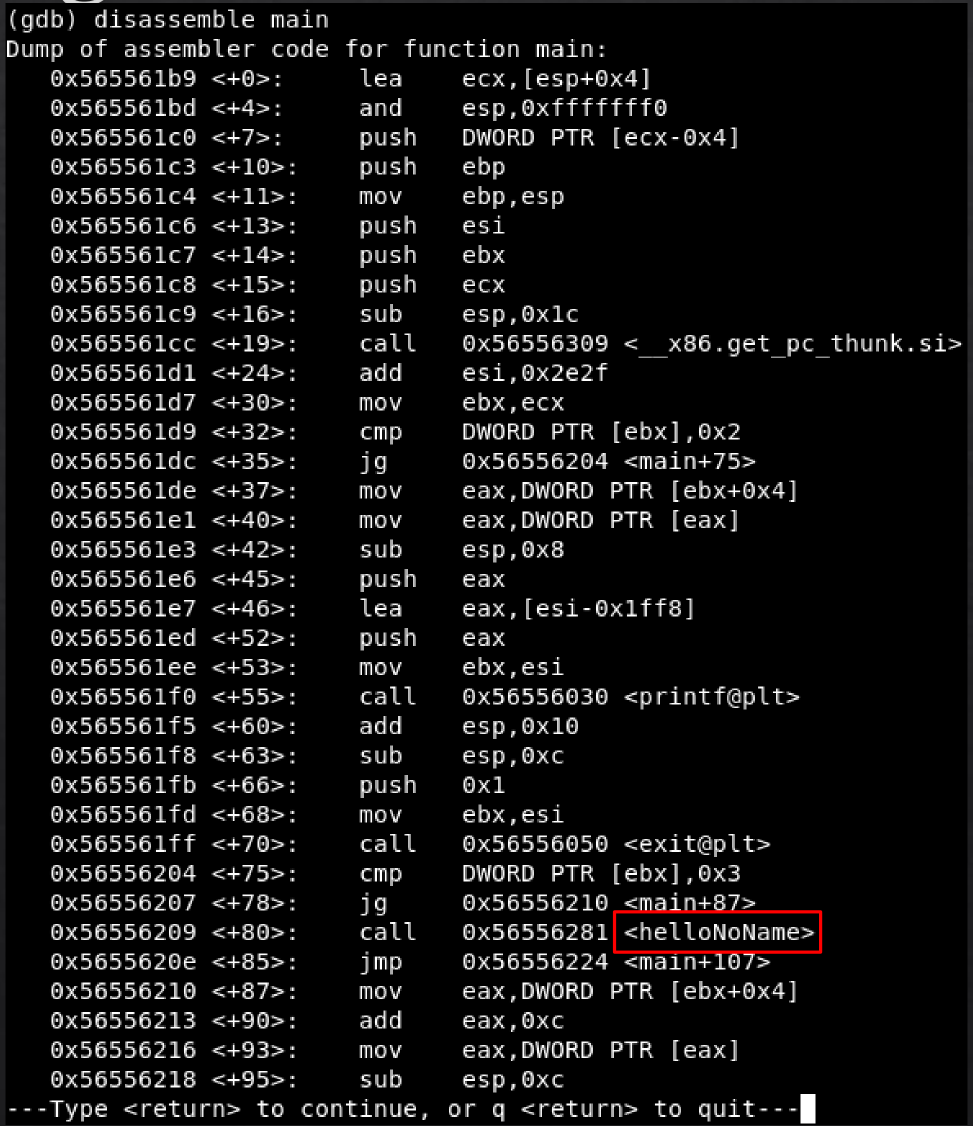
Note: I'll be doing this AFTER a run

GDB - Disassembling Binaries

- Lets look at some
- This shows the assembly
- gdb will try and be helpful
- E.g resolving calls to there functions



```
(gdb) disas helloNoName
Dump of assembler code for function helloNoName:
   0x56556281 <+0>:     push   ebp
   0x56556282 <+1>:     mov    ebp,esp
   0x56556284 <+3>:     push   ebx
   0x56556285 <+4>:     sub    esp,0x4
   0x56556288 <+7>:     call   0x56556305 <__x86.get_pc_thunk.ax>
   0x5655628d <+12>:    add    eax,0x2d73
   0x56556292 <+17>:    sub    esp,0xc
   0x56556295 <+20>:    lea    edx,[eax-0x1fce]
   0x5655629b <+26>:    push   edx
   0x5655629c <+27>:    mov    ebx,eax
   0x5655629e <+29>:    call   0x56556040 <puts@plt>
   0x565562a3 <+34>:    add    esp,0x10
   0x565562a6 <+37>:    nop
   0x565562a7 <+38>:    mov    ebx,DWORD PTR [ebp-0x4]
   0x565562aa <+41>:    leave
   0x565562ab <+42>:    ret
End of assembler dump.
```

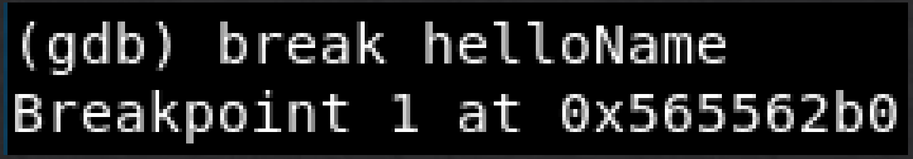


```
(gdb) disassemble main
Dump of assembler code for function main:
   0x565561b9 <+0>:     lea    ecx,[esp+0x4]
   0x565561bd <+4>:     and    esp,0xfffffff0
   0x565561c0 <+7>:     push   DWORD PTR [ecx-0x4]
   0x565561c3 <+10>:    push   ebp
   0x565561c4 <+11>:    mov    ebp,esp
   0x565561c6 <+13>:    push   esi
   0x565561c7 <+14>:    push   ebx
   0x565561c8 <+15>:    push   ecx
   0x565561c9 <+16>:    sub    esp,0x1c
   0x565561cc <+19>:    call   0x56556309 <__x86.get_pc_thunk.si>
   0x565561d1 <+24>:    add    esi,0x2e2f
   0x565561d7 <+30>:    mov    ebx,ecx
   0x565561d9 <+32>:    cmp    DWORD PTR [ebx],0x2
   0x565561dc <+35>:    jg     0x56556204 <main+75>
   0x565561de <+37>:    mov    eax,DWORD PTR [ebx+0x4]
   0x565561e1 <+40>:    mov    eax,DWORD PTR [eax]
   0x565561e3 <+42>:    sub    esp,0x8
   0x565561e6 <+45>:    push   eax
   0x565561e7 <+46>:    lea    eax,[esi-0x1ff8]
   0x565561ed <+52>:    push   eax
   0x565561ee <+53>:    mov    ebx,esi
   0x565561f0 <+55>:    call   0x56556030 <printf@plt>
   0x565561f5 <+60>:    add    esp,0x10
   0x565561f8 <+63>:    sub    esp,0xc
   0x565561fb <+66>:    push   0x1
   0x565561fd <+68>:    mov    ebx,esi
   0x565561ff <+70>:    call   0x56556050 <exit@plt>
   0x56556204 <+75>:    cmp    DWORD PTR [ebx],0x3
   0x56556207 <+78>:    jg     0x56556210 <main+87>
   0x56556209 <+80>:    call   0x56556281 <helloNoName>
   0x5655620e <+85>:    jmp    0x56556224 <main+107>
   0x56556210 <+87>:    mov    eax,DWORD PTR [ebx+0x4]
   0x56556213 <+90>:    add    eax,0xc
   0x56556216 <+93>:    mov    eax,DWORD PTR [eax]
   0x56556218 <+95>:    sub    esp,0xc
---Type <return> to continue, or q <return> to quit---
```

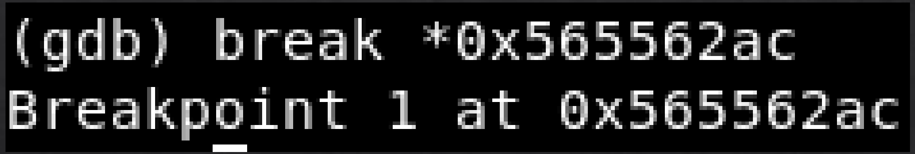
Note: I'll be doing this AFTER a run

GDB - Breakpoints

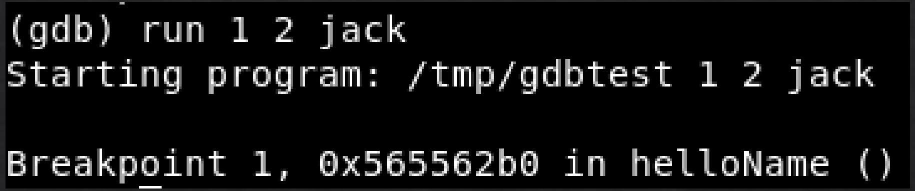
- We saw earlier, helloName() is at 0x565562ac
- Lets pause execution at it
- 2 ways to pause (that we will look at now anyway)
- give gdb a * before 0x to indicate it is an address
- But they have different addresses!?
- Lets run an see what happens on each
- They both break at the start of helloName()
- Because pointers
- GDB AUTOMATICALLY BREAKS ON SEG FAULT!!!



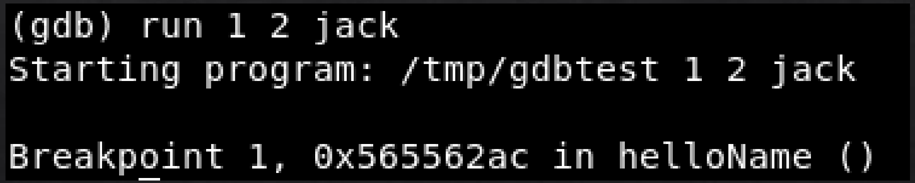
```
(gdb) break helloName
Breakpoint 1 at 0x565562b0
```



```
(gdb) break *0x565562ac
Breakpoint 1 at 0x565562ac
```



```
(gdb) run 1 2 jack
Starting program: /tmp/gdbtest 1 2 jack

Breakpoint 1, 0x565562b0 in helloName ()
```



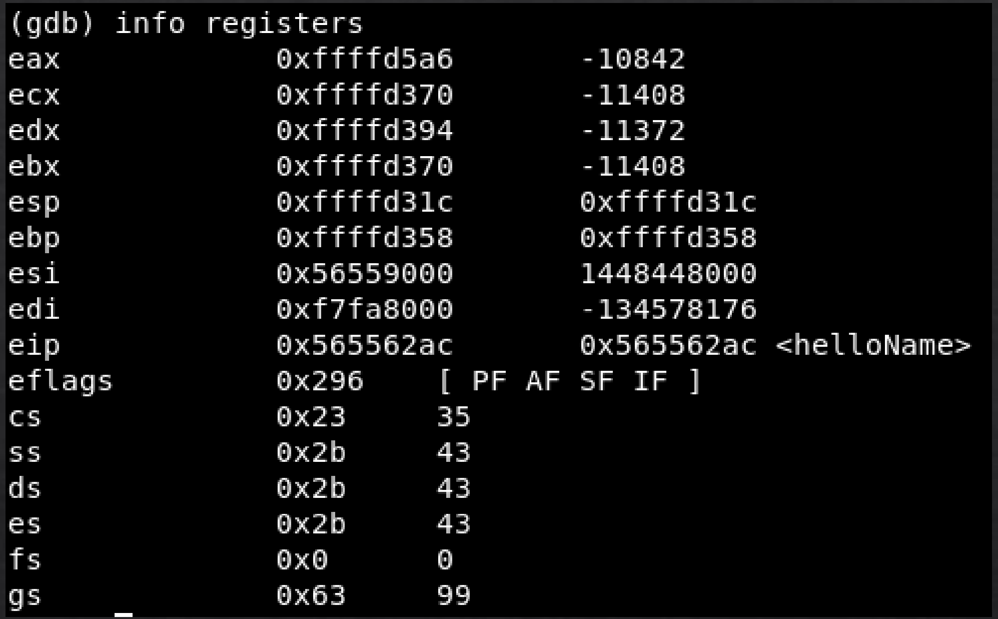
```
(gdb) run 1 2 jack
Starting program: /tmp/gdbtest 1 2 jack

Breakpoint 1, 0x565562ac in helloName ()
```

Breakpoints can be cleared, disabled and enabled. See later cheat sheet

Note: I'll be doing this AFTER a run

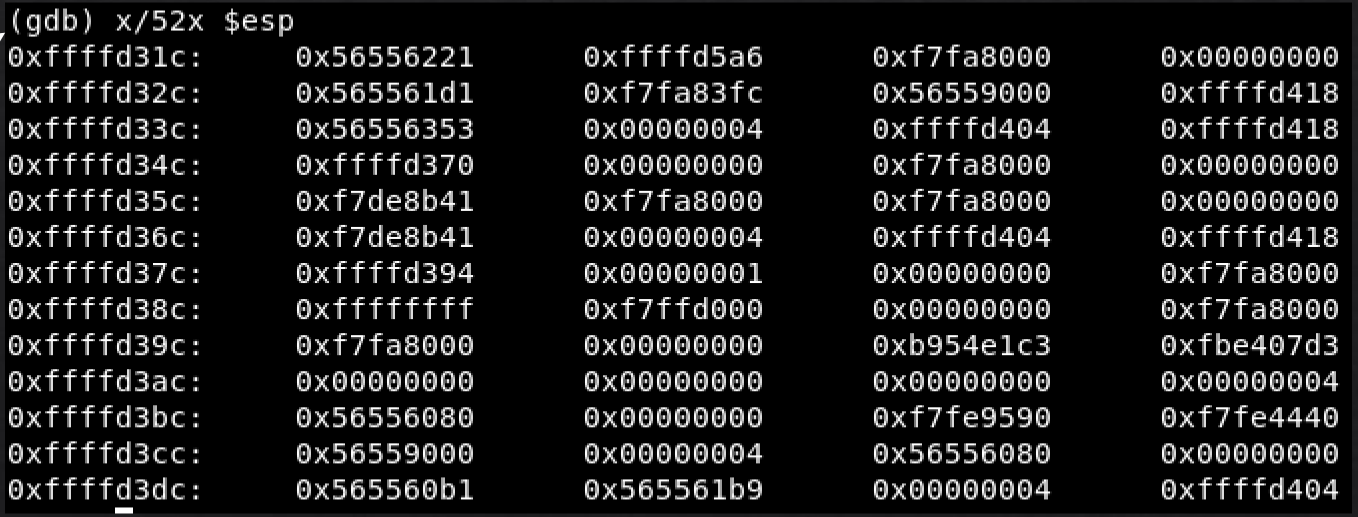
GDB - Look at what happened

- So we're in a break point
- Lets inspect the current state of the process
- The registers?
- How about the stack?
- This will become very useful next week!



```
(gdb) info registers
eax            0xffffd5a6          -10842
ecx            0xffffd370          -11408
edx            0xffffd394          -11372
ebx            0xffffd370          -11408
esp            0xffffd31c          0xffffd31c
ebp            0xffffd358          0xffffd358
esi            0x56559000          1448448000
edi            0xf7fa8000          -134578176
eip            0x565562ac          0x565562ac <helloName>
eflags         0x296      [ PF AF SF IF ]
cs             0x23       35
ss             0x2b       43
ds             0x2b       43
es             0x2b       43
fs             0x0        0
gs             0x63       99
```

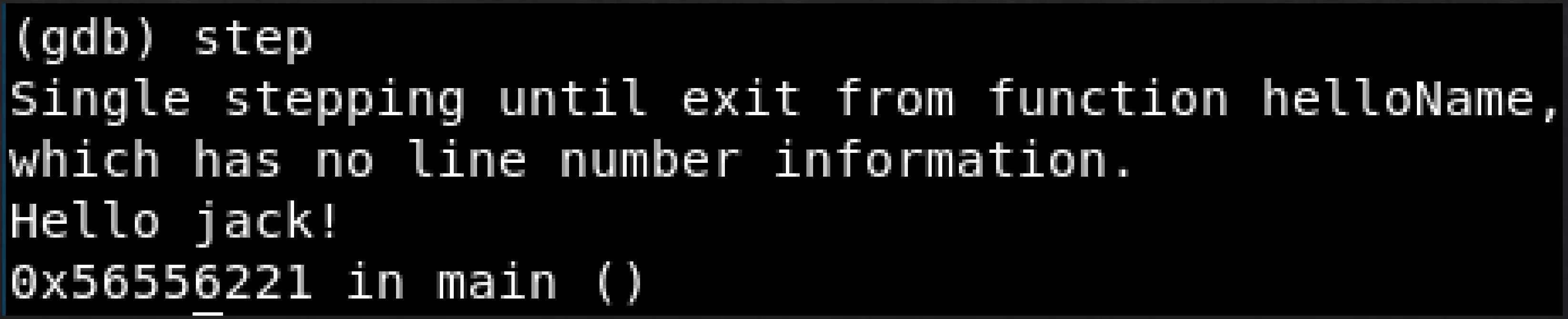
Can reference registers with $



```
(gdb) x/52x $esp
0xffffd31c:     0x56556221      0xffffd5a6      0xf7fa8000      0x00000000
0xffffd32c:     0x565561d1      0xf7fa83fc      0x56559000      0xffffd418
0xffffd33c:     0x56556353      0x00000004      0xffffd404      0xffffd418
0xffffd34c:     0xffffd370      0x00000000      0xf7fa8000      0x00000000
0xffffd35c:     0xf7de8b41      0xf7fa8000      0xf7fa8000      0x00000000
0xffffd36c:     0xf7de8b41      0x00000004      0xffffd404      0xffffd418
0xffffd37c:     0xffffd394      0x00000001      0x00000000      0xf7fa8000
0xffffd38c:     0xffffffff      0xf7ffd000      0x00000000      0xf7fa8000
0xffffd39c:     0xf7fa8000      0x00000000      0xb954e1c3      0xfbe407d3
0xffffd3ac:     0x00000000      0x00000000      0x00000000      0x00000004
0xffffd3bc:     0x56556080      0x00000000      0xf7fe9590      0xf7fe4440
0xffffd3cc:     0x56559000      0x00000004      0x56556080      0x00000000
0xffffd3dc:     0x565560b1      0x565561b9      0x00000004      0xffffd404
```

GDB - Carrying On

- Once you're done with the break
- You want to carry on execution
- Until the next break (or just run)



```
(gdb) step
Single stepping until exit from function helloName,
which has no line number information.
Hello jack!
0x56556221 in main ()
```

GDB - Useful Commands

- A good cheat sheet https://tinyurl.com/y8ndpbya (for pure gdb)
- Set gdb to follow child process on fork()
- **set follow-fork-mode child**
- Set disassembly mode to intel
- **set disassembly-flavor intel**
- Backtrace execution
- **backtrace full**
- Conditional break!
- **break <where> if <condition>**

GDB - Plugins

- Some plugins provide extra functions for gdb
- 2 big ones
- PEDA & GEF
- I personally prefer PEDA for normal use (although I'm considering swapping)
- But GEF has more options, e.g supports more architectures
- We wont be using these yet, but keep them in mind
- (Especially for HackBack!)

Conclusions

Conclusions

- That was a lot of information
- Don’t worry if you don’t get it all
- Or if you forget some!
- Hopefully I've covered everything
- Next week we'll be breaking binaries