Project 1

Part 1 i

1.

A pointer is a variable in many programming languages (e.g. c) that contains the address of another position in memory. A simple example using an 16-bit system, a variable whose address is located at address fa5c might contain a value f160. This would mean that the pointer variable can be used to reference or change the contents of the memory location whose address is f160.

e.g.

Address	Contents
fa5c	f160
...	
f160	88

In the C language, an integer is created:

```
int j = 88;
```

A pointer could then be created:

```
int *p;
```

The pointer can then be set to look into j:

```
p = &j;
```

The various values of various outputs may be:

Address of j: f160

Value stored in *p: 88

Address in p: f160

Address of p: fa5c

2.

Memory in C is much more under the control of the programmer than in Java. They can allocate, alter and free memory from the heap. This extra control has the disadvantage that it can lead to memory leaks with uncareful programming.

Memory in Java is much more controlled by the JVM which will garbage collect any memory that is not being used any more.

3.

Static memory in C is allocated when the data structure that is to contain the data is created. The data structure will be added to the stack frame for the function that it is contained within. The memory will be released when the function returns and the stack frame is removed. e.g.

```
void main() {
	char mem1[100];
	strcpy(mem1, "Glorious peanut\n");
	printf("mem1 is %s", mem1);
}
```

would output:

```
	mem1 is Glorious peanut
```

Dynamic memory in C is allocated when a relevent system function is called which will allocate the relevant amount of memory on the heap (e.g. malloc()) and can change its since using realloc() . This should then be released using the free(). If this is not done by the process, it is the responsibility of the OS to clean up the heap (e.g. with badly written code or after an unexpected termination).

```
void main() {
	char *mem2;
	mem2 = malloc(sizeof("Glorious peanut\n")+1 * sizeof(char));
	printf("mem2 is %s", mem2);
	free(mem2);
}
```

This would output:

```
	mem2 is Glorious peanut
```

4.

Limiting the size of the integer to the smallest that will be needed might make sense. This may be the case if a data structure being used is particularly large and a large data type may take up a very large amount of memory, slowing the program down.

A problem the could occur is that specifically making an integer a particular fixed width size can be compiled to be different sizes on different machines or different compilers. For exxample a short may be 16 or 32 bits [1] on different compilers and the original intent of the programmer could be difficult to understand. This could be the case on different c compilers, for example ISO C, in which case it would be better to use

the C99 portable width integers such as int16_t, signed 16-bit integers, which gives the reader a much better understanding of intent.

One problem that could occur when using the ISO C fixed width integers mentioned above is when we are declaring particular sizes to set a side a specific size of memory. If it is assumed that, for example, an integer takes up 32-bits when in fact, it takes up 64-bits, this could lead a programmer to accidentally create an over-flow when things change (compiler, machine etc). This might not only crash a program, but might also lead to security problems if abused by outside threat.

5.

Compile-time is the period when a textual representation of a high level language is processed into machine code, a representation the can be run directly on the hardware of a computer.

Run-time is the specific period when compiled code that is taken by a computer and added to a process is run inside this process. The process itself contains a number of logical structures which are used to help with the running of the program, including such things as a stack, a heap, a location for enironment variables amongst others.

A programmer may choose to use different features of this process depending on the things they are trying to achieve, for example, local variables in a function might use the stack, whereas dynamic memory allocation might use the heap. In modern software development, it is important to take into consideration the specific security protections that may be in place in the enivornment the process is to be run (e.g. stack execution protection).

6.

```
include <stdio.h>

int glob_1;

int glob_2 = 88;

int funky(int* par_1) {
        int loc_1 = 99;
        printf("par_1 = %d at 0x%x\n", par_1, &par_1);
        printf("loc_1 = %d at 0x%x\n", loc_1, &loc_1);
        return loc_1;
}

void main() {
        int *ptr_1 = malloc(20 * sizeof(int));
        glob_1 = funky(ptr_1);

        printf("*ptr_1 = %d at 0x%d\n", *ptr_1, ptr_1);
        printf("glob_1 = %d at 0x%x\n", glob_1, &glob_1);
        printf("glob_2 = %d at 0x%x\n", glob_2, &glob_2);
        free(ptr_1);
}
```

This program outputs:

*par_1 = 77 at 0xa5817c38

loc_1 = 99 at 0xa5817c44

*ptr_1 = 77 at 0x1184264800

glob_1 = 99 at 0x4525d018

glob_2 = 88 at 0x4525d010

This will focus more on Linux based systems used here.

The two global variables, glob_1 and glob_2 demonstrate static global variables declared at compile time. glob_2 is also defined here. Any global data that is uninitialized or set to zero is placed in the .bss section of the elf. All of this is set to zero when moved into a process. Any initialized data is copied into .data section where it's value is also copied from the elf.

Here we can see from the elf file that glob_1 is in section 24 and glob_2 is in section 23 of the program file:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
55:	0000000000004010	4	OBJECT	GLOBAL	DEFAULT	23	glob_2
67:	0000000000004018	4	OBJECT	GLOBAL	DEFAULT	24	glob_1

We can see that 23 is the .data section and 24 is the .bss section. The PROGBITS here means that the initialization data is taken from the elf file, whilst NOBITS means that nothing is taken as it all just needs to be set to zero.

[Nr] Name	Type	Address	Offset	Size
[23] .data	PROGBITS 0	000000000004000	00003000	0000000000000014
[24] .bss	NOBITS	0000000000004014	00003014	000000000000000c

Here is some of the information taken from the stack frame for the funky() function from above using gdb:

(gdb) info frame

Stack level 0, frame at 0x7fffffffd9a0:

The following shows the argument and also local variable information taken using gdb. This shows that these variables are stored on the stack:

(gdb) info locals

loc_1 = 99

(gdb) info args

par_1 = 0x555555559260

In main() the following line shows an example of the allocation of memory during runtime.:

int *ptr_1 = malloc(20 * sizeof(int));

7.

An assembler takes assembly language instructions and "assembles" or converts them into machine code instructions. Machine code is a low-level lanaguage the runs directly on a CPU. Assembly language is simply a symbolic representation of machine code which is easier for humans to understand.

A compiler takes a particular high-level language (e.g. C) and converts a program into machine code. The high level language is a programming language that is independent of the hardware inside of the computer. The compiler goes through a variety of steps which create an executable file which can be run indepdently of the compiler.

An interpreter is similar to a compiler in that it takes a high-level lanaguage and converts it into machine code. The difference is that the interpreter will convert only small parts (e.g. one line) of the high-level language to machine code before running that small section and then move onto the next line(s). This means that no executable file is created and the interpreter needs to be supplied with the high-level language file to be able to run it. The interpreter may also translate the source code into an intermediate language and then directly execute this language or supply the intermediate language to be run at a different time and possibly even on a different computer with a different types of processor.

8.

C is a compiled language. Python is not a compiled language. Python is an interpreted language.

A python .py file is run through an interpreter, which will take it through a set of steps as follows:

1. Lexcial analysis - Creates a set of tokens from the original source code
2. Parse - Create a structure for the tokens
3. Compile - Create bytecode from the parsed code. This is an intermediate representation of the .py file.

The bytecode is stored in a .pyc file. This .pyc file can then be immediately run or else it can be passed on to be run at a different time and possibly even on a different computer with a different types of processor.

To run the bytecode it must be passed to a virtual machine. This virtual machine is simply an emulation of another CPU and related systems that can run the bytecode. The VM will convert the bytecode instructions to the machine code of the processor on the machine the virtual machine is running on so that it can be run as a program native to that machine.

9.

The GNU compiler is one of a set of compilers for creating C which was built by Free Software Foundation. It was called this because, at the time, there was only one compiler.

It is only available for *nix computers.

It is operated (usually) through a command line interface. In its most basic form, compiling can be achieved by entering the following:

```
gcc program_name
```

There are many extra options that can be added to expand to this instruction to expand its capabilities e.g.:

```
gcc -o object_name program_name
```

which will name the object file.

10.

GDB is a command-line debugger used to find bugs and test compiled programs. It allows a program to be paused at any point and the contents of the program to be examined and also altered. A program can also be stepped-through, which means that it can be run one line at a time and any function can be moved into and out of. The debugger has many extra advanced features, e.g. examination and editing of the stack, cpu registers and memory locations.

Some gdb commands [2]

/

a. To view all cpu registers - i r

To view specific registers - p $reg e.g. p $rdx

b. To set breakpoint at function - break func_name

To set a breakpoint at line - break line_num or break filename: line_num

c. Set a breakpoint at a call to the specific function and then step into the function and examine the address that the debugger displays for the current instruction or the $eip.

d. to examine memory, the x/nfu command can be used. Here n is the number of units needed, f is the format (like printf - e.g. d for integer) and u is unit (b - byte, h - half-word 2 bytes , word 4 bytes, g - giant word 8 bytes)

x/3db 0x555555559260

outputs:

0x555555559260: 0 0 0

Project 1 Part ii

1.

Address space layout randomization is a technique used to help fight against exploitations via memory alteration (e.g. stack overflows). It does this by randomly positioning import structures in a processes memory so that an an attacker is unable to jump into those sections of memory easily.

There are at multiple ways that this could be overcome including: [3].

i. The first is to brute force the attack so that the program is run as many times as is needed so that the correct area for the memory exploit is guessed and and the attack code can be run. This is not always a choice. Even running this kind of attack once and failing can be enough to make a system admin suspicious of an attack. Also, it could take a long time for this to work (maybe many days), depending on the amount of randomization.

ii. The second method is to use an information leak. Here we a leak the address for something such as and then we can work out the difference between the leaked address and a static address. This difference can then be used to work out the address of anything else in that process.

2.

Stack canaries work by adding a random number(s) at certain positions on the stack (e.g. after buffers) and then checking that those values have not been altered before returning. Linux will also set the first byte of the canary value to x00 so as to terminate any string used to cause the overflow.

Stack Canaries are exploitable in at least three ways [4] [5]:

i) Check function hijacking

When the canary check fails, a function is often called to deal with the situation (stack_chk_fail in glibc in linux). Here the function can be hijack to point to the attackers own code by modifying the Global Offset table or similar so that the reference to stack_chk_fail points to the attack shellcode.

ii) Override thread local storage (TLS)

Here we could use the fact that the TLS in linux is used to canary value which will be compared to the value stored on the stack. If a massive overflow can be created such that the canary value on the stack as well as the value on the TLS are the same, the exploit code can be run.

iii) Brute force guess the canary value by running the program over and over until the correct one is guessed.

3.

A buffer overflow happens when too much information is copied into a buffer causing the data to 'leak' into the memory at the end of the buffer.

The stack is located in memory between the stack base pointer at the bottom of the stack (which is in high memory in a Linux process) and (in older attacks) the stack pointer which points to the next memory location that data will be pushed on to the stack.

The stack is used to hold stack frames. Stack frames contain information about each and every call to a function that is made in a process.

For the most basic type of stack overflow, which would simply crash the process being attacked, a user could find some kind of input function for the current process that has not been sufficiently protected for overflows and then enter or alter the input so that the values entered is too big for the buffer, which can then leak into and through any other local variables and then over the functions return address. The returning function would then try to jump to the address contained in the corrupted memory location causing some kind of fatal segmentation error.

A further advancement to this is to add some shell code to the end of the corrupting input as well as corrupting the function return address to the start of this shell code.

4.

EBP - The Extended Base Pointer points to the bottom of the stack fram. In an x86 CPU this is in high memory. This is a constant value as the stack never moves while the function is running.

ESP - The Extended Stack pointer points the the very top of the stack, which is low in memory compared witht the EBP. i.e. the stack is basically up-side-down in memory. Everytime something is popped into the stack the stack pointer will have a relevant number subtracted from it so it is always pointing to the next memory location to position items on the stack.

EIP - The Extended Instruction Pointer is only used with the stack on some occasions. When a function returns to the previous position it was called from, the EIP is set to the return address stored in the current stack frame before that frame is removed.

ESI and EDI - The Extended Source Index and the Extended Destination Index are also only used sometimes by the stack. They are used when transferring data from one buffer to another in an efficient manner. In this case the source and/or destination buffer could be located on the stack.

Part 1 iii

-O0

The -O option allows the change of the optimization level of the compiler from -O0 which has no optimization to -O3 as welll as -0fast) which can have significant improvement although the can lead to a large image size.

For -O0, the gcc documentation states that it is used to[6]:

'Reduce compilation time and make debugging produce the expected results. This is the default.'

This means that firstly the amount of time for compiling is reduced, making the development process easier and secondly that the lack of optimization means that the resulting code comes out without any alterations for optimization (e.g. ordering of machine code instructions) which will make debugging and understanding code easier.

The -O0 is the default, but using other optimization levels may cause unexpected behaviour which could exploits such as stack overflows to fail. One example of this would be inline functions which are present on level 3 optimizations.

-g

This option adds debugging information to the binary in a format that is native to the operating system (from one of stabs, COFF, XCOFF or DWARF [7]). The default format can be changed by using -gstabs+, -gstabs, -gxcoff+, -gxcoff, or -gvms

For Ubuntu, the default system is DWARF which uses a tree format, where each node is represents elements from the onject file, such as types, variables, or functions [8]

For the DWARF format, extra sections are added to the executable that are not loaded up when it is run meaning that there is no performance hit, although the size of the file is larger:

```
[26] .debug_aranges    PROGBITS         0000000000000000  00003037
     0000000000000030  0000000000000000           0     0     1
[27] .debug_info       PROGBITS         0000000000000000  00003067
     000000000000038e  0000000000000000           0     0     1
[28] .debug_abbrev     PROGBITS         0000000000000000  000033f5
     000000000000011d  0000000000000000           0     0     1
[29] .debug_line       PROGBITS         0000000000000000  00003512
     0000000000000135  0000000000000000           0     0     1
[30] .debug_str        PROGBITS         0000000000000000  00003647
     00000000000002b5  0000000000000001  MS       0     0     1
```

This basically means that gdb will show lots of extra infomation about the c program and not just information about the assembly language, which is what is displayed when -g is not used.

-fno-stack-protector

The -fno-stack-protector option will disable the stack protector from the executable created by GCC.

The stack protector for fstack-protectoradds "...a guard variable to functions with vulnerable objects. This includes functions that call alloca, and functions with buffers larger than 8 bytes".. [9]. If protoection is needed for all functions, then the fstack-protector-all is needed.

These guard variables are set up with random numbers and when a function is entered and they are then examined when the funtion is returned from. If the contents of the guard variable are not the same as the value stored in the GOT, the program will halt.

The following is a memory dump of the stack for this c program

```
include <stdlib.h>
static int myGlobal = 0;
int main(int argc, char** argv) {
        int *x = malloc(10*sizeof(int));
        return 0;
}
```

With -fno-stack-protector goes from:

rsp 0x7fffffffd9a0 0x7fffffffd9a0

to

Stack level 0, frame at 0x7fffffffd9d0

```
-rwxrwxr-x 1 liveo liveo 17808 Jan 22 14:25 bss
```

```
(gdb) x/100db 0x7fffffffd9a0
0x7fffffffd9a0: -88     -38     -1      -1      -1      127     0       0
0x7fffffffd9a8: 96      80      85      85      1       0       0       0
0x7fffffffd9b0: -96     -38     -1      -1      -1      127     0       0
0x7fffffffd9b8: 0       -120    6       -89     61      -106    -32     -55
0x7fffffffd9c0: -112    81      85      85      85      85      0       0
0x7fffffffd9c8: 107     -21     -37     -9      -1      127     0       0
0x7fffffffd9d0: 0       0       0       0       0       0       0       0
```

```
(gdb) x/100dw 0x7fffffffd9a0
0x7fffffffd9a0: -9560   32767   1431654496      1
0x7fffffffd9b0: -9568   32767   -1492744192     -908028355
0x7fffffffd9c0: 1431654800      21845   -136582293      32767
0x7fffffffd9d0: 0       0       -9560   32767
```

```
0x7fffffffd9a0: 0x00007fffffffdaa8      0x0000000155555060
0x7fffffffd9b0: 0x00007fffffffdaa0      0xc9e0963da7068800
0x7fffffffd9c0: 0x0000555555555190      0x00007ffff7dbeb6b
0x7fffffffd9d0: 0x0000000000000000      0x00007fffffffdaa8
```

With -fno-stack-protector goes from:

rsp 0x7fffffffd9a0 0x7fffffffd9a0

to

Stack level 0, frame at 0x7fffffffd9d0

```
-rwxrwxr-x 1 liveo liveo 17760 Jan 22 14:32 bss
```

```
0x7fffffffd9a0: -88     -38     -1      -1      -1      127     0       0
0x7fffffffd9a8: 80      80      85      85      1       0       0       0
0x7fffffffd9b0: -96     -38     -1      -1      -1      127     0       0
0x7fffffffd9b8: 96      -110    85      85      85      85      0       0
0x7fffffffd9c0: 96      81      85      85      85      85      0       0
0x7fffffffd9c8: 107     -21     -37     -9      -1      127     0       0
0x7fffffffd9d0: 0       0       0       0       0       0       0       0
```

```
0x7fffffffd9a0: -9560   32767   1431654480      1
0x7fffffffd9b0: -9568   32767   1431671392      21845
0x7fffffffd9c0: 1431654752      21845   -136582293      32767
0x7fffffffd9d0: 0       0       -9560   32767
```

```
0x7fffffffd9a0: 140737488345768 5726621776
0x7fffffffd9b0: 140737488345760 93824992252512
0x7fffffffd9c0: 93824992235872  140737351773035
0x7fffffffd9d0: 0       140737488345768
```

As can be seen from above, although the stack does not change size, but it has changed when the canary value was added. It should also be noted that the size of the executable has grown from 17760 to 17808 bytes, which is 48 bytes.

-x execstack

This option makes the stack executable. By default gcc stops code executing on the stack to stop an attacker performing a stack overflow and running code there. This is partly to keep the ability to compile older code that was written when gcc allowed code to run on the stack.

On the PT_GNU_STACK entry in the elf file ("contains the access rights (read, write, execute) of the stack" [10]).

This is the the program header for GNU_STACK when the code from question 3 is compiled without the '-z execstack' option':

```
-[:)] % readelf -l bss | grep -A2 STACK
 GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                0x0000000000000000 0x0000000000000000  RW     0x10
```

This is the same with the '-z execstack' option':

```
[:)] % readelf -l bss | grep -A2 STACK
GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
               0x0000000000000000 0x0000000000000000  RWE    0x10
```

Notice the flags in the second entry now has an E for execute.

Part 1 iv

1. The stack on 32-bit linux address space can be found close to the top of its contents (which is in low memory). Here we can see the bottom of the process for a running c program (The lowest part of its memory using the pmap utility):

```
00005649752f0000    788K r---- gdb
00005649753b5000   4248K r-x-- gdb
00005649757db000   2068K r---- gdb
00005649759e0000    876K r---- gdb
0000564975abb000     84K rw--- gdb
0000564975ad0000    136K rw---   [ anon ]
00005649774b7000   3464K rw---   [ anon ]
00007f86900a9000   3384K rw---   [ anon ]
00007f86903f7000     28K r--s- gconv-modules.cache
00007f86903fe000  15320K r---- locale-archive
00007f86912f4000     28K rw---   [ anon ]
00007f86912fb000      8K r---- libbz2.so.1.0.4
00007f86912fd000     52K r-x-- libbz2.so.1.0.4
```

Here is the top (highest memory) of the same process:

```
00007f8692092000      4K r---- ld-2.29.so
00007f8692093000      4K rw--- ld-2.29.so
00007f8692094000      4K rw---   [ anon ]
00007ffd3209b000    136K rw---   [ stack ]
00007ffd321f4000     12K r----   [ anon ]
00007ffd321f7000      4K r-x--   [ anon ]
ffffffffff600000      4K --x--   [ anon ]
 total             44496K
```

The default size being (the hard limit only being changeable by root):

```
Max stack size            8388608              unlimited            bytes
```

Here is the stack point from the same running program when it is in the main() function:

```
esp                 0xffffcbc0           0xffffcbc0
```

The same programs stack pointer when we are in a function called from main():

```
esp                 0xffffcb80           0xffffcb80
```

This is showing that when the top of the stack is increased that the stack pointer moves down in memory from 0xffffcbc0 to 0xffffcb80.

To summarise, the stack is near the top of the process memory, just below the environment variables and when it gets bigger it grows downwards in memory to a maximum size.

2.

The following program will be used:

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

int glob_1;
int glob_2 = 88;

int funky(int* par_1) {

    int loc_1 = 99;

    printf("par_1 = %d at 0x%x\n", par_1, &par_1);
    printf("loc_1 = %d at 0x%x\n", loc_1, &loc_1);

    return loc_1;
}

void main() {
    int *ptr_1 = malloc(20 * sizeof(int));

    ptr_1 = 77;

    glob_1 = funky(ptr_1);

    printf("*ptr_1 = %d at 0x%d\n", *ptr_1, ptr_1);
    printf("glob_1 = %d at 0x%x\n", glob_1, &glob_1);
    printf("glob_2 = %d at 0x%x\n", glob_2, &glob_2);

    free(ptr_1);
}
```

Here we have stepped through to the funky() function at line 11:

```
(gdb) info frame
Stack level 0, frame at 0xffffcbb0:
 eip = 0x565561f3 in funky (compile.c:11); saved eip = 0x5655627f
 called by frame at 0xffffcbf0
 source language c.
 Arglist at 0xffffcba8, args: par_1=0x4d
 Locals at 0xffffcba8, Previous frame's sp is 0xffffcbb0
 Saved registers:
  ebx at 0xffffcba4, ebp at 0xffffcba8, eip at 0xffffcbac
```

...and the stack pointer and base pointer...

```
esp            0xffffcb80          0xffffcb80
ebp            0xffffcba8          0xffffcba8
```

... and finally a memory dump of the stack frame area for funky():

hexadecimal bytes:

```
0xffffcb80:     0x00    0x00    0x00    0x00    0x84    0xcc    0xff    0xff
0xffffcb88:     0x00    0x10    0xf8    0xf7    0x4d    0x00    0x00    0x00
0xffffcb90:     0x00    0x00    0x00    0x00    0x00    0x10    0xf8    0xf7
0xffffcb98:     0x63    0x00    0x00    0x00    0x00    0x39    0x71    0x72
0xffffcba0:     0x00    0x10    0xf8    0xf7    0xcc    0x8f    0x55    0x56
0xffffcba8:     0xd8    0xcb    0xff    0xff    0x7f    0x62    0x55    0x56
0xffffcbb0:     0x4d    0x00    0x00    0x00    0xcc    0x8f    0x55    0x56
```

hexadecimal words:

```
0xffffcbc0:     0x00000001      0xffffcc84      0xffffcc8c      0x0000004d
0xffffcbd0:     0xffffcbf0      0x00000000      0x00000000      0xf7dc3751
0xffffcbe0:     0xf7f81000      0xf7f81000      0x00000000      0xf7dc3751
0xffffcbf0:     0x00000001      0xffffcc84      0xffffcc8c      0xffffcc14
```

A stack might consist of the following:

Top of stack <- Lowest memory address in stack

Function 2 frame <- Called from function 1

Function 1 frame <- Called from main()

main activation frame <- Contains main automatic variables and ebp = 0

main return address <- _libc_start_main_

argc <- Argument count for main

argv <- Array of arguments for main (highest memory)

Each of the function frames then contains:

Functions automatic variables <- Lowest memory address in frame

ebp for frame

return address <- Points to instruction after called from

param1 <- If included

param2 <- If included

etc.

3.

The program starts by setting up main() information

Top of stack

main activation frame

main return address

argc
argv

When main calls a function a new stack frame is added to the top of the stack. Going from the bottom of the stack frame we have:

i. The parameters for the function (first parameter at the highest position, last parameter at the lowest).
ii. After that there is the return address in main which is the next instruction to run after the one that called this function.

iii. The base pointer for the current stack frame so that it can be returned to if another function is called.

iv. The functions automatic variables.

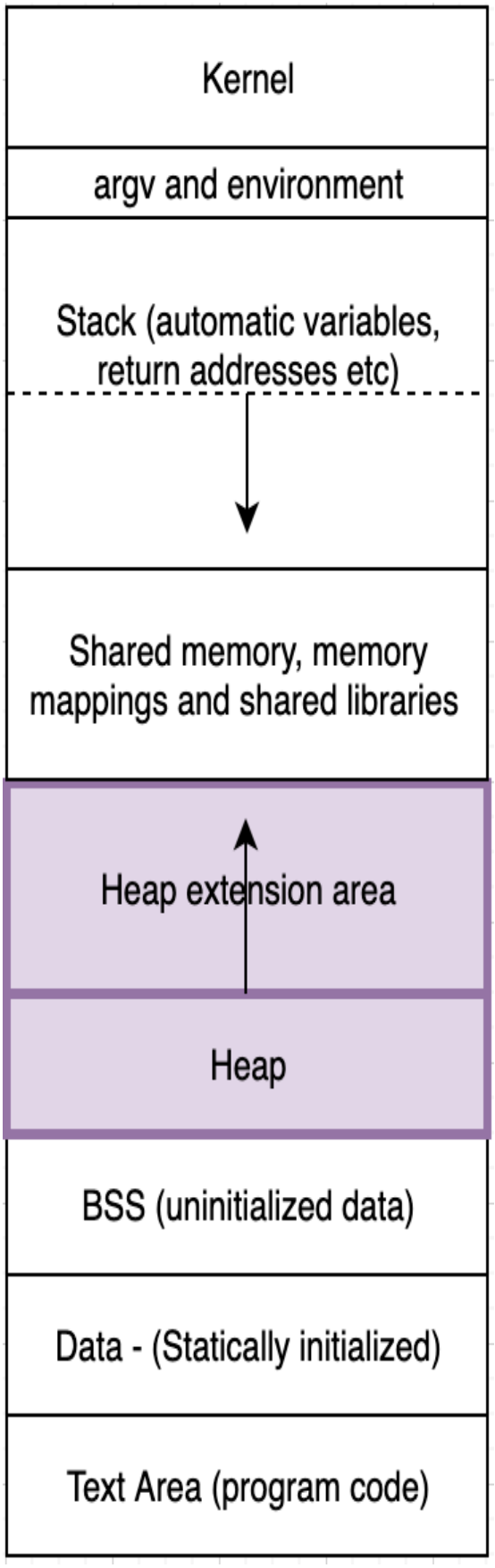
Once this stack frame has been built the instruction pointer will be set to the start of its code and will run from there. Function calls inside this will build further frames.

When exiting from the function, the frame pointer is copied over the stack pointer which will clear that frame from the stack. The return pointer is also popped from the stack and the instruction pointer is set to this so that the code in the previous function after the call is run.

Upon finding main(), normally control will (usually) return to _libc_start_main_.

4.

The heap is located in the lowest end of memory (top of process), above the text (program) and data areas and directly above the bss (uninitialized data). The heap grows from lower memory, upwards towards memory at higher addresses.


Kernel
argv and environment
Stack (automatic variables, return addresses etc)
Shared memory, memory mappings and shared libraries
Heap extension area
Heap
BSS (uninitialized data)
Data - (Statically initialized)
Text Area (program code)

5.

Data on the heap remains there for the remainder of the programs running time. For this reason, it is a more stable area of memory. It is a larger area of memory than is not managed in c, so needs to be managed by the programmer. It needs to be created using functions such as malloc() and then freed again with free(). Areas can also be resized using realloc().

The memory itself is slower to use than other memory and pointer are needed to access it. It is not limited in size (except by limits imposed by the hardware).

The heap memory may become fragmented as it is used because of the way it is allocated and then deallocated. Also, memory leaks can occur if memory is not freed after it has been finished with.

6.

Only data that is dynamically allocated is stored on the heap. This type of dynamic data may be accessed from different functions. This does not include data such as character arrays.

Globals are stored in two different locations. Constants that are global are stored along with the code in the code segment. The data segment is used for non-constant globals.

Project 1 Part 2 i

1. Line 10 is a vulnerable line of code. The reason for this is that the call scanf() does not check for buffer overflows. This means that an attacker could enter a specially created string that is larger than the space allocated for the buffer in the call. In this case the str1 character array which is limited to only 8 chars where a char in c99 is set to 'CHAR_BIT's where CHAR_BIT is at least 8 bits.

The attackers could fill these memory locations and the locations in the stack after these with special values that will enable the attacker to trick the program into thinking that canary value is valid and then send the current function return to a value stored in the buffer itself which would overcome other restrictions.

2.

This program can output "Valid input! Access Granted". One way this can be achieved is by a brute force exploit. To do this an attacker would need to overcome a number of mechanisms including:

i. The Canary Cookie: An exploit would run the victim program many times attempting to guess 3 of the 4 byes in the word (the last is set to 0 to defeat other exploits)

ii. The ESP stored return value. At the start, the stack pointer is stored on the stack.

```
0x0804851b <+0>:     lea    ecx,[esp+0x4]
0x0804851f <+4>:     and    esp,0xfffffff0
0x08048522 <+7>:     push   DWORD PTR [ecx-0x4]
```

At the end, this is returned to ESP.

```
0x080485b7 <+156>:   mov    ecx,DWORD PTR [ebp-0x4]
0x080485ba <+159>:   leave
0x080485bb <+160>:   lea    esp,[ecx-0x4]
```

This value is then used to return to the previous calling function. The attack could change this value so that it may hit the buffer which would have been set up to contain the location of line 15 in the code. This would need to be run multiple times for each canary, because ASLR (if it is used and not switch off another way), shifts the stack poisiton. This should find the buffer quite quickly, finding one of the buffer values.

Step ii is simulated in the following by first stepping up breakpoints and a debugger directive the ignore the cookie value:

```
   0x080485a6 <+139>:   mov    edx,DWORD PTR [ebp-0xc]
   0x080485a9 <+142>:   xor    edx,DWORD PTR gs:0x14
   0x080485b0 <+149>:   je     0x80485b7 <main+156>
   0x080485b2 <+151>:   call   0x80483c0 <__stack_chk_fail@plt>
   0x080485b7 <+156>:   mov    ecx,DWORD PTR [ebp-0x4]
   0x080485ba <+159>:   leave
   0x080485bb <+160>:   lea    esp,[ecx-0x4]
   0x080485be <+163>:   ret
End of assembler dump.
gdb-peda$ break *0x080485a6
Breakpoint 1 at 0x80485a6
gdb-peda$ commands
Type commands for breakpoint(s) 1, one per line.
End with a line saying just "end".
>set $eip=0x080485b7
>c
>end
gdb-peda$ break *0x080485be
Breakpoint 2 at 0x80485be
```

The program is then run (assuming the cookie is correct):

```
[----------------------------------registers-----------------------------------]
EAX: 0x0
EBX: 0x0
ECX: 0xbffff000 --> 0xb7fbb000 --> 0x1b1db0
EDX: 0x6a511043
ESI: 0xb7fbb000 --> 0x1b1db0
EDI: 0xb7fbb000 --> 0x1b1db0
EBP: 0x0
ESP: 0xbfffeffc --> 0xb7e21637 (<__libc_start_main+247>:        add    esp,0x10)
EIP: 0x80485be (<main+163>:     ret)
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80485b7 <main+156>:        mov    ecx,DWORD PTR [ebp-0x4]
   0x80485ba <main+159>:        leave
   0x80485bb <main+160>:        lea    esp,[ecx-0x4]
=> 0x80485be <main+163>:        ret
   0x80485bf:   nop
   0x80485c0 <__libc_csu_init>: push   ebp
   0x80485c1 <__libc_csu_init+1>:       push   edi
   0x80485c2 <__libc_csu_init+2>:       push   esi
[------------------------------------stack-------------------------------------]
0000| 0xbfffeffc --> 0xb7e21637 (<__libc_start_main+247>:       add    esp,0x10)
0004| 0xbffff000 --> 0xb7fbb000 --> 0x1b1db0
0008| 0xbffff004 --> 0xb7fbb000 --> 0x1b1db0
0012| 0xbffff008 --> 0x0
0016| 0xbffff00c --> 0xb7e21637 (<__libc_start_main+247>:       add    esp,0x10)
0020| 0xbffff010 --> 0x1
0024| 0xbffff014 --> 0xbffff0a4 --> 0xbffff28a ("/home/project1/project/proj")
0028| 0xbffff018 --> 0xbffff0ac --> 0xbffff2a6 ("XDG_VTNR=7")
[------------------------------------------------------------------------------]
```

Here ESPs final bye is fc (ff - 4) from the program. AAAABBBBCCCCDDDD is input (but should be replaced with repeated address of line 15.)

This should be run as above until the buffer is hit:

```
[----------------------------------registers-----------------------------------]
EAX: 0x0
EBX: 0x0
ECX: 0xbfaf5200 ("DDDD")
EDX: 0x5cd9ef43
ESI: 0xb7f84000 --> 0x1b1db0
EDI: 0xb7f84000 --> 0x1b1db0
EBP: 0x0
ESP: 0xbfaf51fc ("CCCCDDDD")
EIP: 0x80485be (<main+163>:     ret)
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80485b7 <main+156>:        mov    ecx,DWORD PTR [ebp-0x4]
   0x80485ba <main+159>:        leave
   0x80485bb <main+160>:        lea    esp,[ecx-0x4]
=> 0x80485be <main+163>:        ret
   0x80485bf:   nop
   0x80485c0 <__libc_csu_init>: push   ebp
   0x80485c1 <__libc_csu_init+1>:       push   edi
   0x80485c2 <__libc_csu_init+2>:       push   esi
[-------------------------------------stack-------------------------------------]
0000| 0xbfaf51fc ("CCCCDDDD")
0004| 0xbfaf5200 ("DDDD")
0008| 0xbfaf5204 --> 0xbfaf5200 ("DDDD")
0012| 0xbfaf5208 --> 0x0
0016| 0xbfaf520c --> 0xb7dea637 (<__libc_start_main+247>:       add    esp,0x10)
0020| 0xbfaf5210 --> 0xb7f84000 --> 0x1b1db0
0024| 0xbfaf5214 --> 0xb7f84000 --> 0x1b1db0
0028| 0xbfaf5218 --> 0x0
[------------------------------------------------------------------------------]
```

3.

The code would need to be changed so that either:

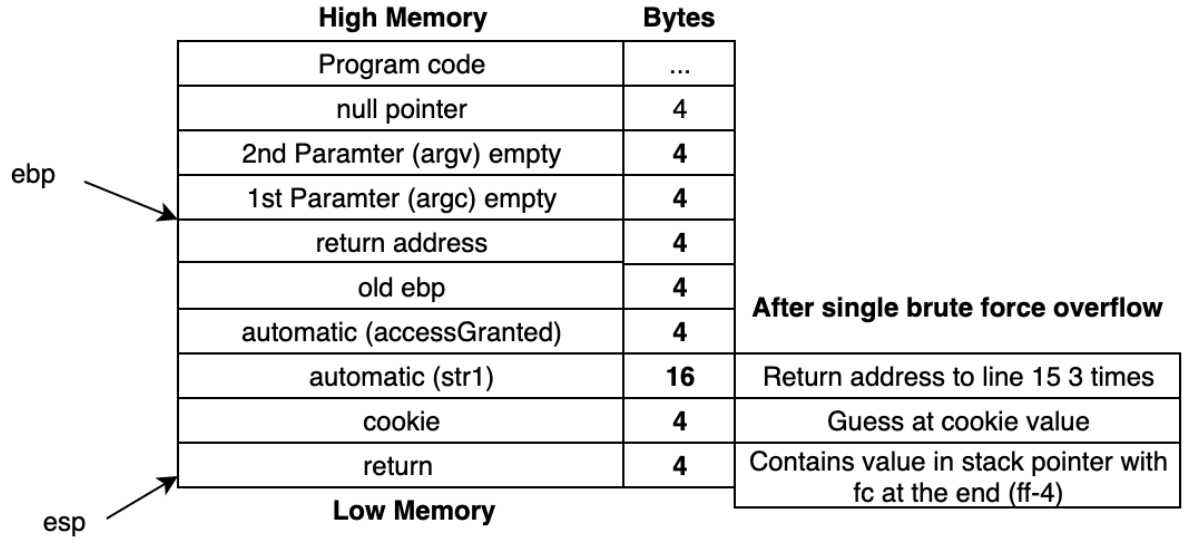
i) The string that is copied it in limited to one less that size of the buffer and then the null terminator is added to the end by hand [13 - https://www.sudo.ws/todd/papers/strlcpy.html]

ii) Use a function such as strlcpy() which have been to be safer to use. This particular function is guaranteed to "to NUL-terminate the destination string for all strings where the given size is non-zero" [13]. It will also take the size of the destination string into the function as a parameter. Lastly, the function will fill the rest of the string with zeroes.

4.

Strongly-types languages can be useful in helping stop buffer overflows, although they are not a complete solution. One way they can do this is by only allowing execution of code on data of the correct type. Another way would be in guaranteeing certain safety features with any data types such as limit checking on buffers.

Part 2 ii


High Memory
Bytes
Program code
...
null pointer
4
2nd Paramter (argv) empty
4
ebp
1st Paramter (argc) empty
4
return address
4
old ebp
4
automatic (accessGranted)
4
automatic (str1)
16
cookie
4
return
4
esp
Low Memory
After single brute force overflow
Return address to line 15 3 times
Guess at cookie value
Contains value in stack pointer with fc at the end (ff-4)

Citations

1 - “Don't Follow These 5 Dangerous Coding Standard Rules " Barr Code,” *Barr Code*. [Online]. Available: https://embeddedgurus.com/barr-code/2011/08/dont-follow-these-5-dangerous-coding-standard-rules/. [Accessed: 27-Jan-2020].

2 - “GDB Cheat Sheet.pdf,” *Google Drive*. [Online]. Available: https://drive.google.com/file/d/12R1ReHSqngx-nL5pnsQ_YuEjiWj6IhPA/view. [Accessed: 27-Jan-2020].

3 - *YouTube*. [Online]. Available: https://www.youtube.com/watch?v=Pht6y4p63SE. [Accessed: 27-Jan-2020].

4 - O. S. I. R. I. S. L. & C. T. F. LLC, “Stack Canaries¶,” *Stack Canaries - CTF 101*. [Online]. Available: https://ctf101.org/binary-exploitation/stack-canaries/. [Accessed: 27-Jan-2020].

5 - C. T. F. W. Team, “Canary¶,” *Canary - CTF Wiki*. [Online]. Available: https://ctf-wiki.github.io/ctf-wiki/pwn/linux/mitigation/canary/. [Accessed: 27-Jan-2020].

6 - *Optimize Options - Using the GNU Compiler Collection (GCC)*. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc-4.9.4/gcc/Optimize-Options.html#Optimize-Options. [Accessed: 27-Jan-2020].

7 - *Using the GNU Compiler Collection (GCC): Debugging Options*. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Debugging-Options.html. [Accessed: 27-Jan-2020].

8 - “Exploring the DWARF debug format information,” *IBM Developer*. [Online]. Available: https://developer.ibm.com/technologies/systems/articles/au-dwarf-debug-format/. [Accessed: 27-Jan-2020].

9 - *GCC Stack Protector options*, 24-May-2016. [Online]. Available: https://mudongliang.github.io/2016/05/24/stack-protector.html. [Accessed: 27-Jan-2020].

10 - “Lair Of The Multimedia Guru,” *Lair Of The Multimedia Guru " PT_GNU_STACK*. [Online]. Available: https://guru.multimedia.cx/pt_gnu_stack/. [Accessed: 27-Jan-2020].