1. Stack Overflow Challenges
1 - Stack Overflow Challenges

LEVEL 0

Level 0 is a basic stack overflow which doesn’t require us to obtain control over code-execution.

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

#define BUF_SIZE 120

void vuln_func(char* arg)
{
    char filename[30];
    char buffer[BUF_SIZE];
    FILE* fp;
    strcpy((char*)filename,"/var/levels/level0/level0.c");
    strcpy(buffer,arg);
    printf("Welcome %s\n",buffer);
    fp = fopen(filename,"rb");
    if(fp == 0)
    {
        printf("[-]Error opening file %s\n",filename);
        return;
    }
    printf("Contents of file %s\n",filename);
    while(!feof(fp))
    {
        memset((char*)buffer,0,BUF_SIZE);
        fread(buffer,1,BUF_SIZE-1,fp);
        printf("%s",buffer);
    }
    printf("]\"");
    fclose(fp);
    return;
}
```

As we can see, strcpy on line 19 copies the content at the arg pointer to the buffer up until a null-byte. Located after this buffer is the filename buffer which is 30 bytes long. As the application source-code shows in lines 23 and 33-38 it reads the contents of filename and prints them to the user.
**LEVEL 1**

Level 1 is a stack overflow where we have to obtain control over codeflow in order to execute our own shellcode and obtain shell-access to the target machine.

```c
void disp(char* str)
{
    printf("%s\r\n", str);
    return;
}

void vuln_func(char* arg)
{
    void (*ptr)();
    char buf[128];

    printf("buf 0 %p\r\n", buf);
    printf("ptr 0 %p\r\n", ptr);

    ptr = (void (*)(()))disp;

    strcpy(buf, arg);
    ptr(buf);

    return;
}
```

As we can see from line 20, again the data at the pointer with user-supplied content is copied to buffer ‘buf’ of 128 bytes. This time, however, there is no content to be read. Instead, there is a pointer ‘ptr’ which gets assigned the address of function ‘disp’ in line 18. On line 21, the pointer ‘ptr’ is treated like a function and is called with buf as an argument. What this means is that on line 21, the value of the ‘ptr’ variables is moved into register eax and a CALL eax instruction is executed, thus transferring codeflow to whatever the address contained in ‘ptr’ is.
Level 2 is a stack overflow where we have to obtain control over codeflow in order to execute our own shellcode and obtain shell-access to the target machine.

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

void vuln_func(char* arg)
{
    char buf[128];
    printf("buf \0 x\n", buf);
    strcpy(buf, arg);
    return;
}

int main(int argc, char* argv[])
{
    char tmp_buf[4096];
    fgets(tmp_buf, 4095, stdin);
    vuln_func(tmp_buf);
    return 0;
}
```
**LEVEL 3**

Level 3 is a stack overflow where we have to obtain control over codeflow in order to execute our own shellcode and obtain shell-access to the target machine. This time there is a static canary on line 9 which when overwritten with the improper value exits the application and thus doesn’t perform a RET instruction and hence doesn’t transfer control flow back to the return-address.

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

void vuln_func(char* arg)
{
    int canary;
    char buf[128];

    canary = 0x0BADCODE;

    strcpy(buf, arg);

    printf("buf %p\r\n", buf);
    printf("0x%04x\r\n", canary);

    if(canary != 0x0BADCODE)
    {
        exit(1);
    }
    else
    {
        return;
    }
}
**LEVEL 4**

Level 4 is a stack overflow where we have to obtain control over codeflow in order to execute our own shellcode and obtain shell-access to the target machine. It is similar to level3 but differs on lines 7 and 10-12 in the sense that there is now a shadow canary and the value is randomly generated. Hence we cannot predict its static value.

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

void vuln_func(char* arg)
{
  int canary;
  int shadow_canary;
  char buf[128];

  srand(time(0));
  canary = (rand() & 0xFFFFFFFF);
  shadow_canary = canary;

  strcpy(buf, arg);

  printf("buf 0x%p\r\n", buf);
  printf("c: 0x%04x s: 0x%04x\r\n", canary, shadow_canary);

  if (canary != shadow_canary)
  {
    printf("[-] Stack smashing detected\r\n");
    exit(1);
  }
  else
  {
    return;
  }
}

int main(int argc, char* argv[])
{
  char tmp_buf[1024];
  fgets(tmp_buf, 1024, stdin);
  vuln_func(tmp_buf);
  return 0;
}
```