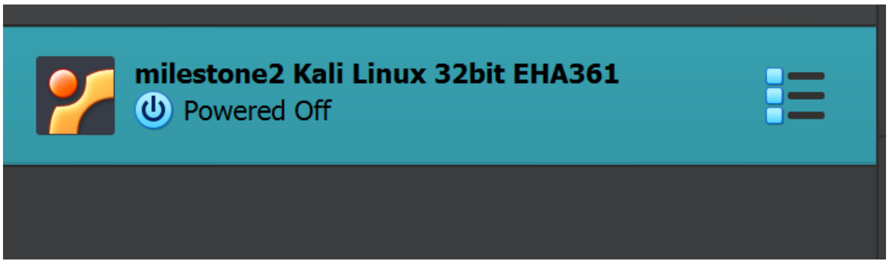
MILESTONE 2

Pieter Johannes Swart

NO. 600640

Table of Contents

Setting up Kali Linux VM



milestone2 Kali Linux 32bit EHA361

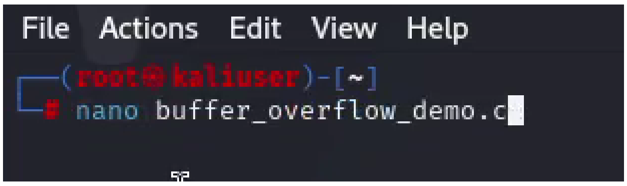
Powered Off

Step 1: Creating a buffer Overflow

The Language that I use is C, that allows low level memory access.

Open terminal => Creating a new C program

“ **nano buffer_overflow_demo.c** ” or “ **nano /home/kaliuser/Desktop/little_virus.c** ”



```
┌──(root㉿kaliuser)-[~]
└─# nano buffer_overflow_demo.c
```

The Code for buffer_overflow_demo.c

#include <stdio.h>

#include <string.h>

- “include <stdio.h>” => Allows input/output operations (e.g., printf).
- “include <string.h>” => Provides string manipulation functions (e.g., strcpy)

// This program causes a buffer overflow when executed

void vulnerable_function() {

char buffer[16]; // Fixed size buffer (16 bytes)

printf("Buffer address: %p\n", (void *)buffer);

- Declares a character array (buffer [16]) => This is a small buffer that can only store 16 bytes of data.
- Prints the memory address of buffer => Helps visualize where the buffer is stored in memory.

```
  // Intentionally overflowing the buffer with 32 'A' characters

  strcpy(buffer, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA");
```

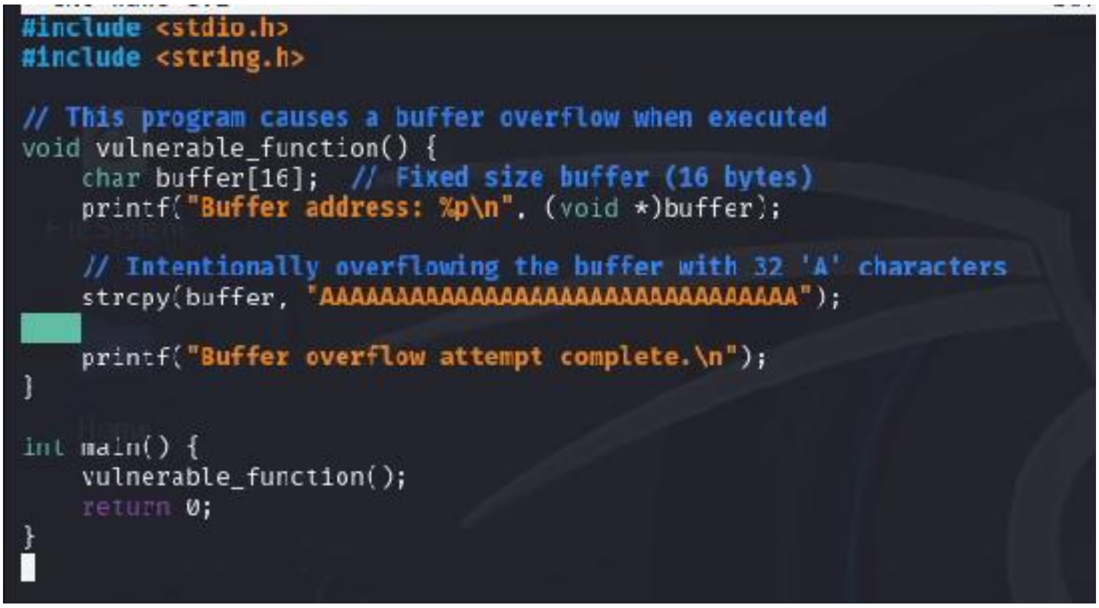
- strcpy (String Copy Function) => Copies the provided string into buffer.
- "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA" is 32 characters long.
- But buffer only has space for 16 characters. The extra characters overwrite next memory! This is what causes a buffer overflow.

```
  printf("Buffer overflow attempt complete.\n");

}
```

- The program prints this **after writing too much data**, but the damage is already done.

```
int main() {

  vulnerable_function();

  return 0;

}
```

- Calls vulnerable_function() to trigger the buffer overflow.
- Returns 0 => Standard practice to indicate successful execution.



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

// This program causes a buffer overflow when executed
void vulnerable_function() {
    char buffer[16];  // Fixed size buffer (16 bytes)
    printf("Buffer address: %p\n", (void *)buffer);

    // Intentionally overflowing the buffer with 32 'A' characters
    strcpy(buffer, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA");

    printf("Buffer overflow attempt complete.\n");
}

int main() {
    vulnerable_function();
    return 0;
}
```

Press “Ctrl + X” **=>** Press “Y” => Press Enter

Explanation of Changes:

- The **vulnerable_function** now automatically overflows the buffer when run.
- A large string 32 “A” characters is directly written into the buffer, causing an overflow immediately when the program runs.

How to Make It More Dangerous?

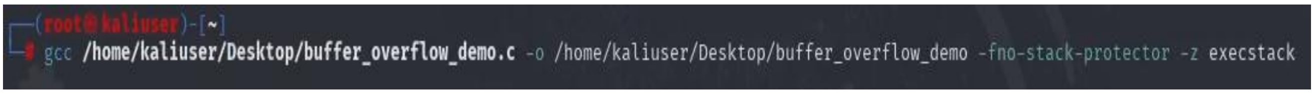
Right now, it only causes an overflow, but it may not always crash. You can:

- Increase the overflow size => Try 64 or more characters.
- Use gets() instead of strcpy().
- Overwrite a function’s return address. This is how hackers exploit programs.

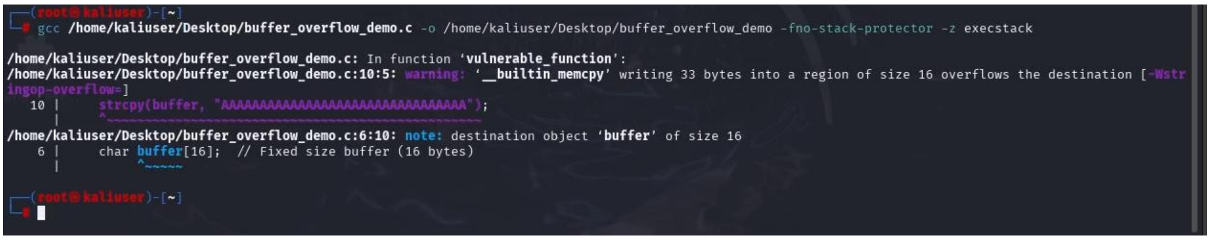
Step 2: Compile the program with no protection flags:

“ gcc /home/kaliuser/Desktop/buffer_overflow_demo.c -o /home/kaliuser/Desktop/buffer_overflow_demo -fno-stack-protector -z execstack ”

- **-fno-stack-protector**: Disables stack protection.
- **-z execstack**: Makes the stack executable.



```
┌──(root㉿kaliuser)-[~]
└─# gcc /home/kaliuser/Desktop/buffer_overflow_demo.c -o /home/kaliuser/Desktop/buffer_overflow_demo -fno-stack-protector -z execstack
```



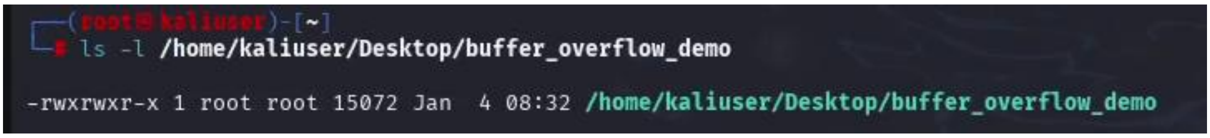
```
┌──(root㉿kaliuser)-[~]
└─# gcc /home/kaliuser/Desktop/buffer_overflow_demo.c -o /home/kaliuser/Desktop/buffer_overflow_demo -fno-stack-protector -z execstack

/home/kaliuser/Desktop/buffer_overflow_demo.c: In function ‘vulnerable_function’:
/home/kaliuser/Desktop/buffer_overflow_demo.c:10:5: warning: ‘__builtin_memcpy’ writing 33 bytes into a region of size 16 overflows the destination [-Wstr
ingop-overflow=]
   10 |     strcpy(buffer, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA");
      |     ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/home/kaliuser/Desktop/buffer_overflow_demo.c:6:10: note: destination object ‘buffer’ of size 16
    6 |     char buffer[16];  // Fixed size buffer (16 bytes)
      |          ^~~~~~

┌──(root㉿kaliuser)-[~]
└─# 
```

That warning you’re seeing is exactly what we expect when causing a buffer overflow.

Step 3: Trigger the Buffer Overflow

Confirm the Compiled File Exists

“ ls -l /home/kaliuser/Desktop/buffer_overflow_demo ”



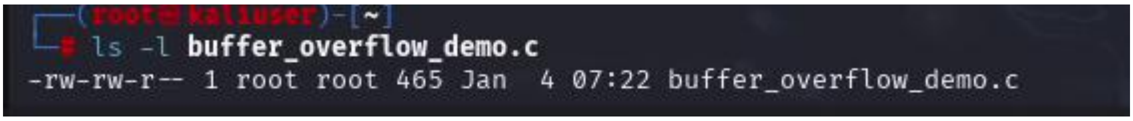
```
┌──(root㉿kaliuser)-[~]
└─# ls -l /home/kaliuser/Desktop/buffer_overflow_demo

-rwxrwxr-x 1 root root 15072 Jan  4 08:32 /home/kaliuser/Desktop/buffer_overflow_demo
```

The “ **x** ” means the file is executable.

If You don't see "**x**" this is how you fix it

Confirm the Compiled File Exists in the Right Location

" **ls -l buffer_overflow_demo.c** "

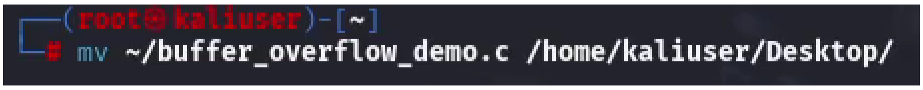


```
┌──(root㉿kaliuser)-[~]
└─# ls -l buffer_overflow_demo.c
-rw-rw-r-- 1 root root 465 Jan  4 07:22 buffer_overflow_demo.c
```

Move the File to the Desktop (If Needed)

If the file exists in your home directory (/root/), move it to the Desktop for easier access:

" **mv ~/buffer_overflow_demo.c /home/kaliuser/Desktop/** "



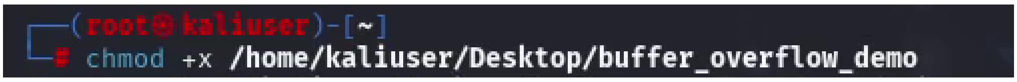
```
┌──(root㉿kaliuser)-[~]
└─# mv ~/buffer_overflow_demo.c /home/kaliuser/Desktop/
```

Compile the program with no protection flags to make it vulnerable:

" **gcc /home/kaliuser/Desktop/buffer_overflow_demo.c -o /home/kaliuser/Desktop/buffer_overflow_demo -fno-stack-protector -z execstack** "

Make the Program Executable:

" **chmod +x /home/kaliuser/Desktop/buffer_overflow_demo** "

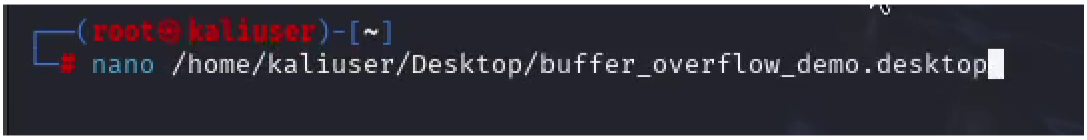


```
┌──(root㉿kaliuser)-[~]
└─# chmod +x /home/kaliuser/Desktop/buffer_overflow_demo
```

- chmod => Stands for Change Mode. It is used to modify file permissions in Linux.
- "+x" => Adds execute permission to the file.

Create a Desktop Launcher for One-Click Execution:

" **nano /home/kaliuser/Desktop/buffer_overflow_demo.desktop** "



```
┌──(root㉿kaliuser)-[~]
└─# nano /home/kaliuser/Desktop/buffer_overflow_demo.desktop
```

Add the Following Code to the “**.desktop**” File:

[Desktop Entry]

Name=Buffer Overflow Demo

Comment=Triggers a buffer overflow vulnerability

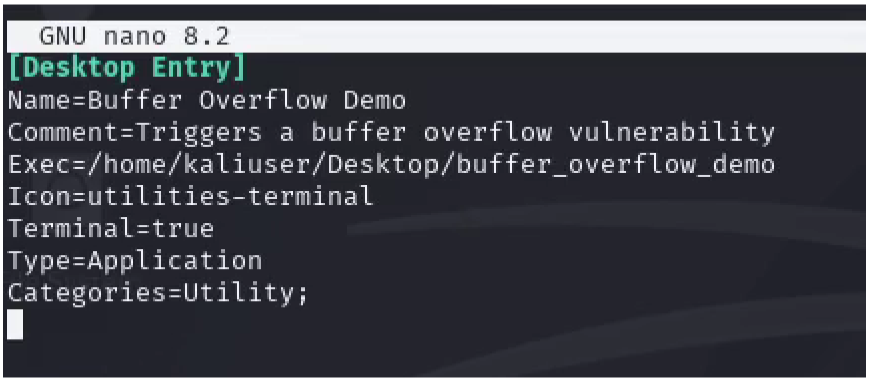
Exec=/home/kaliuser/Desktop/buffer_overflow_demo

Icon=utilities-terminal

Terminal=true

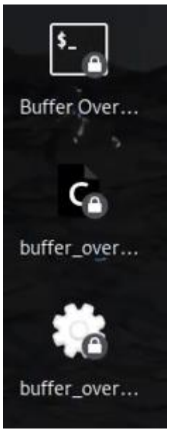
Type=Application

Categories=Utility;



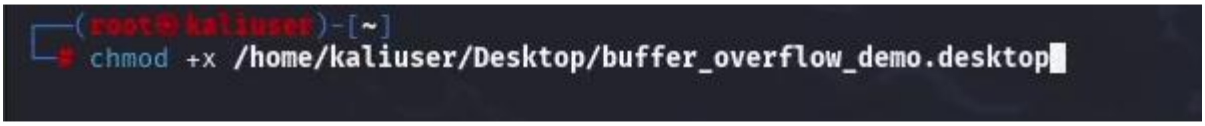
```
  GNU nano 8.2
[Desktop Entry]
Name=Buffer Overflow Demo
Comment=Triggers a buffer overflow vulnerability
Exec=/home/kaliuser/Desktop/buffer_overflow_demo
Icon=utilities-terminal
Terminal=true
Type=Application
Categories=Utility;
```

Press “Ctrl + X” **=>** Press “Y” => Press Enter



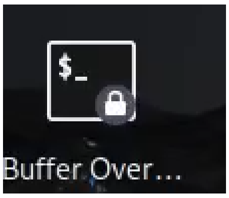
Save and Set Permissions:

“ **chmod +x /home/kaliuser/Desktop/buffer_overflow_demo.desktop** ”

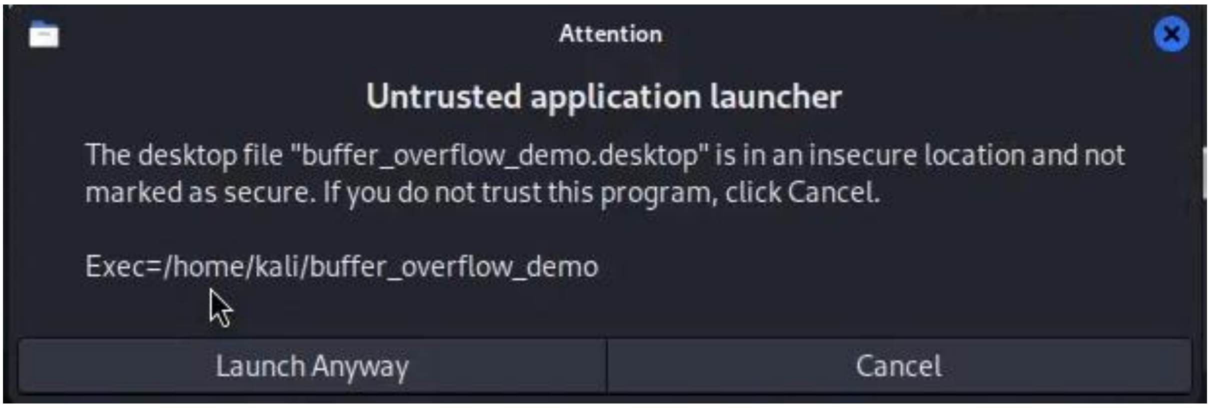


```
┌──(root㉿kaliuser)-[~]
└─# chmod +x /home/kaliuser/Desktop/buffer_overflow_demo.desktop
```

Click on Buffer Overflow Demo



To run the Buffer Overflow double click on “Buffer Overflow Demo => click “Launch Anyway”

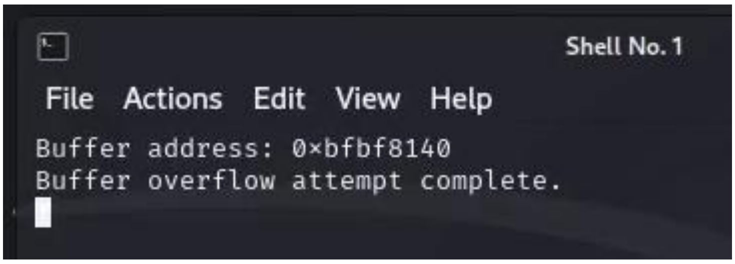


Untrusted application launcher

The desktop file "buffer_overflow_demo.desktop" is in an insecure location and not marked as secure. If you do not trust this program, click Cancel.

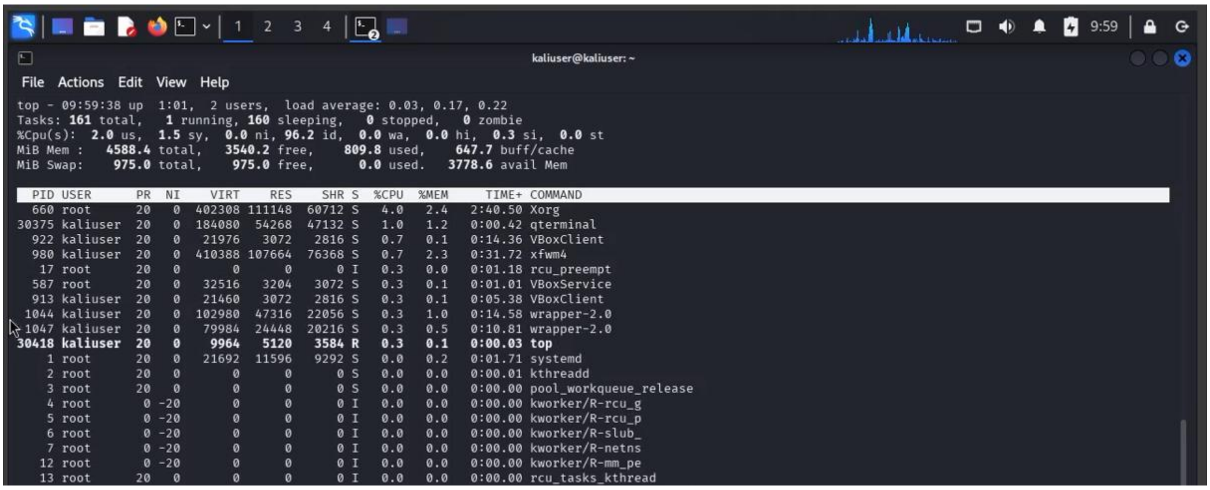
Exec=/home/kali/buffer_overflow_demo

The Output:



```
Buffer address: 0×bfbf8140
Buffer overflow attempt complete.
```

CPU Issues command => “**top**”



```
top - 09:59:38 up  1:01,  2 users,  load average: 0.03, 0.17, 0.22
Tasks: 161 total,   1 running, 160 sleeping,   0 stopped,   0 zombie
%Cpu(s):  2.0 us,  1.5 sy,  0.0 ni, 96.2 id,  0.0 wa,  0.0 hi,  0.3 si,  0.0 st
MiB Mem :   4588.4 total,   3540.2 free,    809.8 used,    647.7 buff/cache
MiB Swap:    975.0 total,    975.0 free,      0.0 used.   3778.6 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
  660 root      20   0  402308 111148  60712 S   4.0   2.4   2:40.50 Xorg
30375 kaliuser  20   0  184080  54268  47132 S   1.0   1.2   0:00.42 qterminal
  922 kaliuser  20   0   21976   3072   2816 S   0.7   0.1   0:14.36 VBoxClient
  980 kaliuser  20   0  410388 107664  76368 S   0.7   2.3   0:31.72 xfwm4
   17 root      20   0       0      0      0 I   0.3   0.0   0:01.18 rcu_preempt
  587 root      20   0   32516   3204   3072 S   0.3   0.1   0:01.01 VBoxService
  913 kaliuser  20   0   21460   3072   2816 S   0.3   0.1   0:05.38 VBoxClient
 1044 kaliuser  20   0  102980  47316  22056 S   0.3   1.0   0:14.58 wrapper-2.0
 1047 kaliuser  20   0   79984  24448  20216 S   0.3   0.5   0:10.81 wrapper-2.0
30418 kaliuser  20   0    9964   5120   3584 R   0.3   0.1   0:00.03 top
    1 root      20   0   21692  11596   9292 S   0.0   0.2   0:01.71 systemd
    2 root      20   0       0      0      0 S   0.0   0.0   0:00.01 kthreadd
    3 root      20   0       0      0      0 S   0.0   0.0   0:00.00 pool_workqueue_release
    4 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/R-rcu_g
    5 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/R-rcu_p
    6 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/R-slub_
    7 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/R-netns
   12 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/R-mm_pe
   13 root      20   0       0      0      0 I   0.0   0.0   0:00.00 rcu_tasks_kthread
```

us (User CPU time):

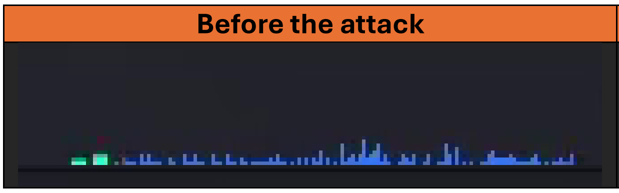
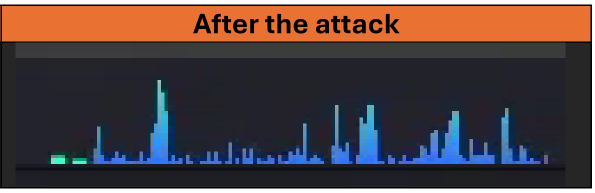
This shows the percentage of CPU utilization that occurred while executing at the user applications. High user CPU time mean that te applications or processes are busy.

sy (System CPU time):

Represents the percentage of CPU utilization that occurred in the kernel or system level. A high system time can indicate that the OS is busy with system level tasks

%CPU:

For individual processes, high %CPU values indicate processes that are currently using a lot of the CPU's processing power.

Before the attack	After the attack
	

The overflow did happen, but the overflow wasn't serious enough to overwrite critical memory like the **return address**. The buffer was filled, but it didn't corrupt important memory areas that would cause a crash.

Increased the overflow string from **32** characters to **64** characters.

What Happened?

- The buffer "char buffer[16]" was declared to hold only 16 bytes.
- The "strcpy()" function copies the input directly into the buffer without checking its size.
- If you *input more than 16 bytes*, the data overflows into next memory, potentially corrupting it or causing a crash.

Diagram Explained

Before Buffer Overflow (Normal Memory Layout)		After Buffer Overflow (Memory Curruption)	
Stack Memory	The Reason	Stack Memory	The Reason
Return Address	Safe, function returns normal	Overwritten Data	Corrupt return address
Save Registers		Save Registers	
Buffer: 16 Bytes	The word “Hello” fits in allocated space	Buffer: 16 Bytes	“AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA” (OverFlow)

The input overwrites the return address, causing undefined behaviour.

References

alesanchezr, 2024. 12 Sep, pp. https://github.com/4GeeksAcademy/cybersecurity-syllabus/blob/main/07-pentesting-red-team/buffer-overflow.md.

Mathpati, N., 2023. Exploiting Buffer Overflow Vulnerabilities: A Step-by-Step Guide (Part 2). 24 APR, pp. https://www.cobalt.io/blog/pentester-guide-to-exploiting-buffer-overflow-vulnerabilities.

Unknown, 2018. Proj 3: Linux Buffer Overflow With Shellcode (20 pts.). 2 APR, pp. https://samsclass.info/127/proj/p3-lbuf1.htm.

Unknown, 2025. Buffer Overflow Attack with Example. Jan 29, pp. https://www.geeksforgeeks.org/buffer-overflow-attack-with-example/.

Unknown, n.d. Proj 3: Linux Buffer Overflow With Shellcode (20 pts.).

Zachariah, B., 2023. Understanding top Command CPU Usage. 18 Sept, pp. https://linuxopsys.com/top-command-cpu-usage.

Zhang, F., 2021. Lab 2: Buffer Overflows. 11 Sep, pp. https://fengweiz.github.io/20fa-cs315/labs/lab2-instruction.pdf.