THINK INDIA JOURNAL

ISSN: 0971-1260
Vol-22-Issue-16-August-2019

Binary Exploitation: Stack Smashing

Aditya Rai, Gagandeep kaur*, Bhupinder kaur, Anshu Vashisth

Lovely Faculty of Technology and Sciences

Lovely Professional University, Phagwara, Punjab

aditya.rai2424@gmail.comgagandeep.23625@lpu.co.in,

bhupinder.23626@lpu.in, anshu.23500@lpu.co.in,

Abstract— With the increase of internet activities many hacking activities and attacks came into existence. Buffer overflow or Binary exploitation is one of the common and dangerous form of security vulnerability from the last decade. In the buffer overflow, the unauthorized person tries to get the complete or partial access of a server or host. Which further can be used for bigger attack. In this paper, I reviewed the various types of buffer overflow vulnerabilities and their mitigation techniques as well as their exploitation. The output is shown using the Linux operating system.

***Keywords*—Binary exploitation, Stack Overflow, Buffer Overflow, Format String Vulnerability, ASLR, StackCanary.**

I. INTRODUCTION

With the introduction of morris worm,many people(or hackers) started to find a way to abuse or gain control over systems or software or code snippet. Many attack methods came such as sql injection,URL tampering etc. One among them was buffer overflow or binary exploitation which was understood by 1972 [3].A big topic in its own.

Buffer flows actually means when the user inputs exceed the limit or capacity of the buffer. It consists of attacking the compiled code or a software, by overflowing the stack via input. Which inturn allowed us to get arbitrary code execution or calling the other functions via passing arguments or functions address. This way many software was cracked and were used illegally without others getting knowledge about it. With increase of such attack's developers/security researchers came with different techniques to prevent such vulnerability.[4]

But soon it also became of no use as those methods could also be exploitedhackers created such approaches that these security methods were also bypassed leading to arbitrary code execution.

ISSN: 0971-1260
Vol-22-Issue-16-August-2019

II. LITERATURESURVEY

Inside computer objects are stored in memory. Memory inside computer resembles brain inside human. Memory can actually store data and instruction. Data is processed inside computer memory.[1]

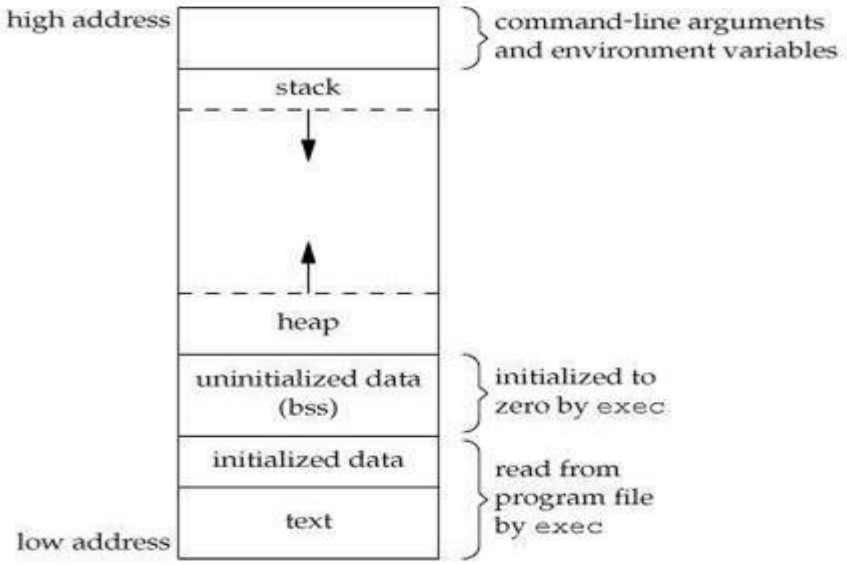


Figure 1: Memory Layout

Stack is a data structure, which is used to store elements or collection of objects , where insertion by push and deletion by pop operation. It follows LIFO order i.e. last in first out. [1][4]

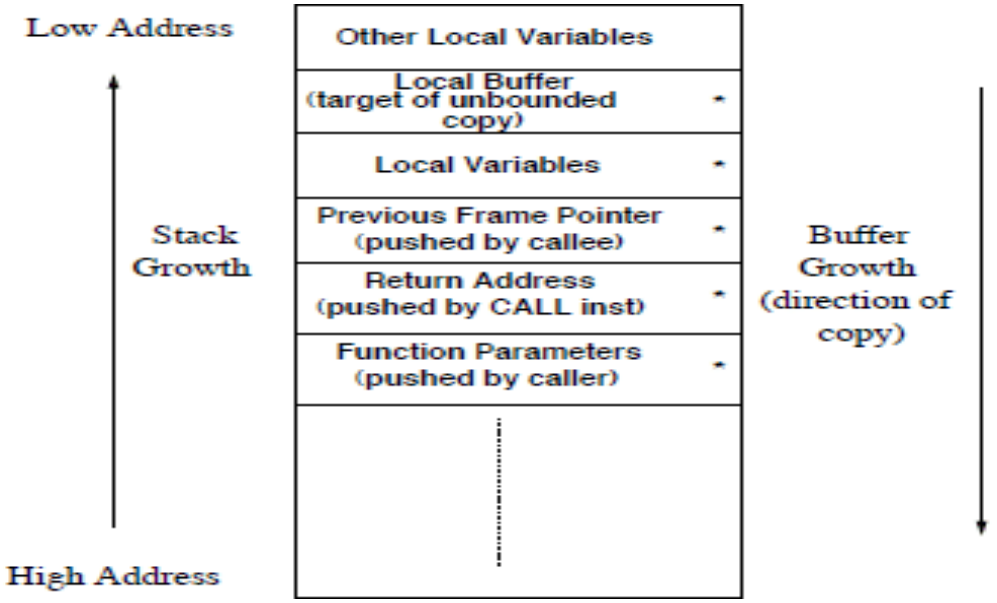


Figure 2:Stack Representation

ISSN: 0971-1260
Vol-22-Issue-16-August-2019

Terminologies About Stack-

- Stack Pointer Or Stack Base Pointer- This pointer always points at the stack's base address (ESP for 32 bit; RBP for 64 bit).
- Instruction Pointer- This points to the instruction that is in execution (EIP for 32 bit; RIP for 64 bit).
- Return Address- This consists the address of next function, which will be pushed in stack, after previous function is executed
- Arguments- These are the address which holds the arguments which is passed in function(32 Bit Direct Passing of Arguments, 64 Bits Passing Through Register Like RDI,RSI,R14,R15etc)

(Naming of these pointers depends on the type of system architecture For 64 Bit- R ;For 32 Bit -E)

- Buffer- It is the location inside memory which is used to store the data which is taken as input from user while running a program.[11]

Assembly Language

Low Level Language Which is converted by assembler into machine language which inturn will be understood by system. Assembly language consist of instructions that are executed usingstacks.Eachinstructionhassomeoperationstodowith registers and stack. When the instruction is called using function, the address of function is pushed into stack and address is stored in eip, arguments are also pushed into stack.

Some instructions used in x86 processors.

- MOV –this instruction is used to move the data from one register or buffer toanother
- ADD –this instruction is used to add the twovalues.
- SUB –this instruction is used to subtraction of values.
- PUSH –this instruction is used to push/insert the data onto astack.
- POP –this instruction is used to pop/remove data from astack.
- JMP - jump to another location.
- INT - interrupt instruction used to interrupt the runningactivity.

Assembly language Instructions to add numbers 6 and 4:

ISSN: 0971-1260
Vol-22-Issue-16-August-2019

mov eax, 6 - loads 3 in register "eax"

mov ebx, 4 - loads 4 inregister "ebx"

add eax, ebx, ecx - adds "eax" and "ebx" and stores output(10) in "ecx"

BUFFER OVERFLOW

This vulnerability arises when any program does not have a check on the length of user input and user can crash the software or can provide some malicious input in order to gain access.

Tools used to analyze and exploit buffer overflow-

- checksec - used to check protection of program
- gdb(gnome debugger) - excellent tool to analyze, disassemble and debug theprogram.
- readelf - used to see the functions and objects used in program
- python language - used to create exploit and interact with program.

For example we have a code snippet which have two functions, win() and main().'win' function cannot be access normally while execution as the function is not called. Only 'main' function is accessed byuser.

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

void win()
{
  printf("code flow successfully changed\n");
}

int main(int argc, char **argv)
{
  volatile int (*fp)();
  char buffer[64];

  fp = 0;

  gets(buffer);

  if(fp) {
      printf("calling function pointer, jumping to 0x%08x\n", fp);
      fp();
  }
}
```

A hacker could easily exploit this code and execute the win() function which should not be executed anyhow as per code flow.[5]We compile this code. And try to exploit it.We cansee directly inside code that gets() function is used. Which will have a vulnerability, which professional developers doesn`t use.

ISSN: 0971-1260
Vol-22-Issue-16-August-2019

Man Page Of gets()

"From the documentation of gets(), we can see it says-**Never use gets().** Because it is impossible to tell without knowing the data in advance how many characters gets() will read, and because gets() will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security. Use fgets()instead."

In order to exploit it,firstly we have to find the length of input after which we overwrite the stack. We call it as finding padding or EIP Control.

Via Hit and Trial Method-

Passing a random long string to the program.[10] Linux Terminal Output--

user@protostar:/opt/protostar/bin$ ./stack3

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaaaaaaaaaaaaaa

calling function pointer, jumping to 0x61616161 Segmentation fault

We see we get a segmentation fault, so trying with lesser length than previous.(0x61 is hex value of 'a')

Linux Terminal Output

user@protostar:/opt/protostar/bin$ ./stack3

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaa

calling function pointer, jumping to0x00006161 Segmentationfault

Now we can see that our 2 'a' are been written on the stack. Same thing can be seen via command-> dmesg | tail

Linux Terminal Output

user@protostar:/opt/protostar/bin$ dmesg | tail

[661.005905] stack3[1948]: segfault at 6161 ip 00006161 sp bffff77c error 4 instack3[8048000+1000]

We can see the same here.

So, we came to know that input of 64 characters fills our buffer.To be precise this 64 characters have filled our buffer and EBP(Stack base pointer).Since Base Pointer doesn`t creates issue we do not face error on overwriting of it. On passing random input after 64 characters, we start over writing our return address. Which can be understood by our

ISSN: 0971-1260
Vol-22-Issue-16-August-2019

stack memory representation.[4][7]

Now, we are done with EIP control, we can provide address of function to return or to execute, in our case 'win' function. Whose address can be found using->

#readelf -Ws [FILENAME] Linux Terminal Output-

user@protostar:/opt/protostar/bin$ readelf -Ws stack3 Symbol table '.dynsym' contains 7 entries:

Num: Value Size Type Bind Vis NdxName

0: 00000000 0 NOTYPE LOCAL DEFAULT UND

1: 00000000 0 NOTYPE WEAK DEFAULT UNDgmon_start

.....

.....

56: 08048424 20 FUNC GLOBAL DEFAULT 14 win

.....

68: 08048438 65 FUNC GLOBAL DEFAULT 14 main

69: 080482e0 0 FUNC GLOBAL DEFAULT 12 _init

Here, we can see that 0x08048424 is the address of win function.But we cannot directly pass the address. We have to follow Endian format.

Endian Format-

Endian format refers to the ordering of bytes (or sometimes bits) within a binary representation of a number.

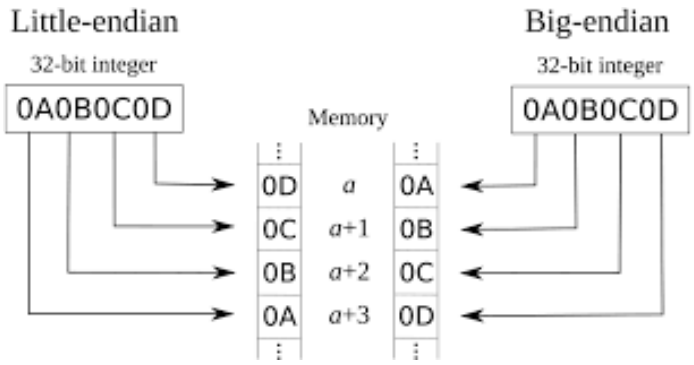


Figure 3: Endian Format

Little Endian Format: In this address is passed in reverse format.

Big Endian Format- In this same addressis passed.

Little endian format is used here to pass the address i.e. in reverse order.So, 0x08048424

ISSN: 0971-1260
Vol-22-Issue-16-August-2019

in little endian becomes \x24\ x84\x04\x08.Hence, final payload becomes -> 64*'a'+\ x24\x84\x04\x08.

Linux Terminal Output-

user@protostar:/opt/protostar/bin$ python -c "print 64*'a'+'\x24\x84\x04\x08'" |./stack3

calling function pointer, jumping to 0x08048424

code flow successfully changed

And successfully able to change the code execution to win function which was not intented to executenormally.

Final memory representation is as follows-

Our 'a's have filled the buffer and EBP and just after it win address is overwritten on return address.

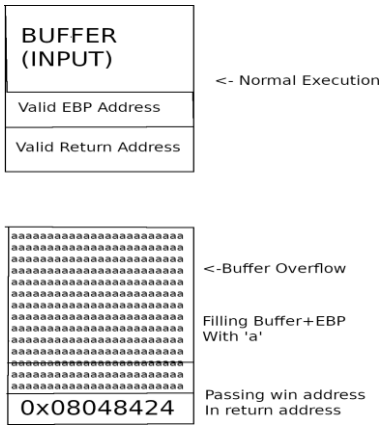


Figure 4: Buffer Input

By using this vulnerability, it was able to change the code execution. As when the EIP(Instruction pointer) reaches return address it returns to win() function and executesit.[8][11]

Another approach could be used here. Instruction can be passed and modify it in a way that it is executed by the program.This methods is generally used to get a shell/remote access of the service(if it is running on a server or any other person system) by passing shellcode in out input.[9] 'Shell-storm.org' this website have a lot of pre made shellcodes using which we can get shell tosystem.

COUNTERMEASURES

Later this attack became famous and some countermeasures were developed. They are-

ISSN: 0971-1260
Vol-22-Issue-16-August-2019

- **NXBit**

This technique marks writable part of memory as non executable. Buffer is a writeable part of memory, if it is marked non executable our passed shellcode wont be executed.When the non executable part of memory is overwritten by any other value, it will not execute, the program will crash.[8][11]

These files are compiled as-

'gcc -m32 -fno-stack-protector -static -znoexecstack -o FILE' This adds 'noexecstack' protection on stack i.e. non executable stack.On using 'checksec' command on compiled file we can see the protection-

user@ubuntu-xenial:/$ checksec 1_staticnx

[*] '/vagrant/lessons/6_bypass_nx_rop/build/1_staticnx' Arch: i386-32-little

RELRO: Partial RELRO Stack: No canary found **NX:**

NXenabled

PIE: NoPIE

- **Canary**

This type of protection technique involves having a randomised guard value after stack frame local variables and before the return address. When the program is executed and when it return instruction address is executed ,the guard value is compared if it is same the program continues otherwise the program crashes.[1]

On checking protection usingchecksec-

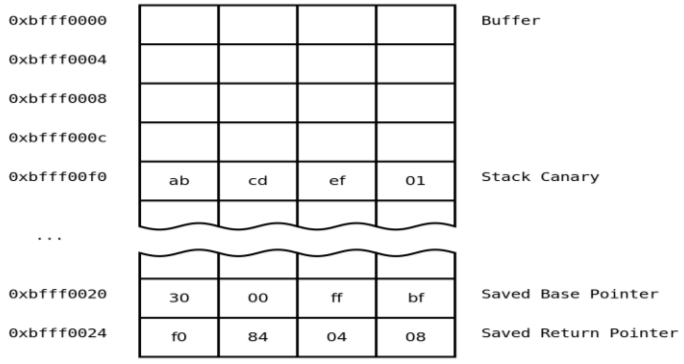
Arch:i386-32-little

RELRO:Partial RELRO

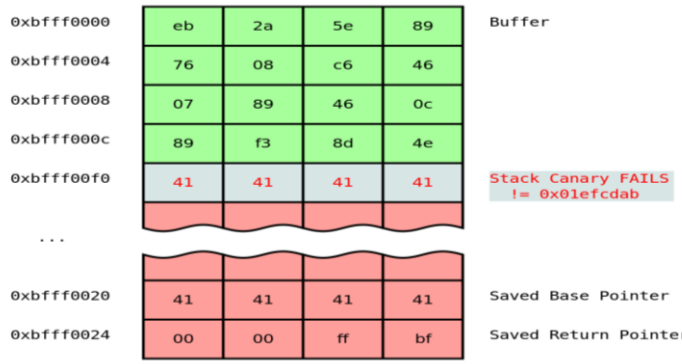
Stack:Canary found

NX: NXenabled

PIE:NoPIE



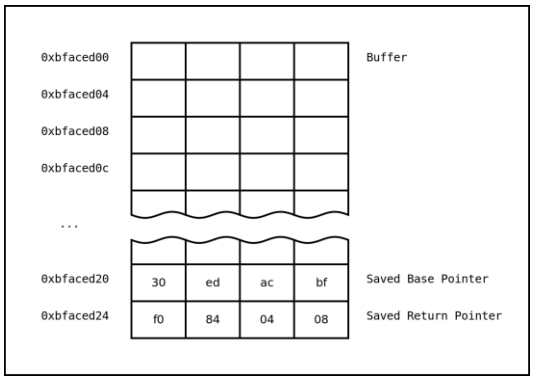
ISSN: 0971-1260
Vol-22-Issue-16-August-2019

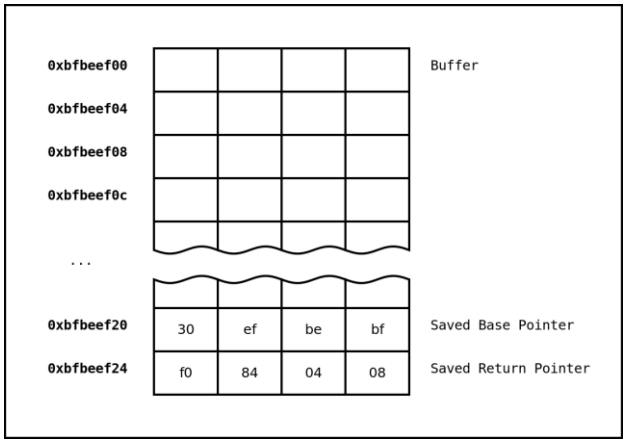


Here at addre placed, as a check,if changed even l in figure 2.

- **ASLR(Address Space LayoutRandomisation)**

  As we know that memory addresses are been mapped to the stack, in this protection technique the address of memory regions are randomised. So everytime the program is runned each time the addresses are different. Which makes it difficult for the attacker to place its shellcode and instruction at appropriate memory place.[14]





III. BPASSINGPROTECTIONS

- NXBit:

Techniques To Bypass

ISSN: 0971-1260
Vol-22-Issue-16-August-2019

- Rop
- Ret2libc Prior knowledge- CodeGadgets-

These are useful code sequences that end in 'ret' instructions are useful to construct a ROP chain. Which might look like:

'**0x0804f065 : pop esi ; pop ebp ; ret** '

These are called gadgets.We use this to create a chain ,where each gadget returns something to other gadget as they have return instruction in the end. Using this chain we try to control execution as accordingly.

- **Libc File**

The term "libc" is basically "standard C library", a information center of standard functions used by all C programs.

- **Linux Syscall**

These are the interface between the user space application and linux kernel.Using which linux kernal can be invoked by placing right argument in the right register.

Example- system() , execve()

We have to place address of system() into right register and the argument '/bin/sh' into its appropriate register.[/bin/sh/ spawn the shell when executed]

Note:In 64 bit binary file arguments are stored in registers whereas in 32 bit arguments are stored in stack.[11]

- **Rop (Return Oreinted Programming)**

We create a ROP Chain, which is a combination of some instructions which on execution will provide us shell/control.

*First step is EIP Control.

*Second Step is to find code gadgets.

*Third step is to generate ROP Chain, which can be automated using ROPgadget tool, which generates rop chain and python exploit.[12]

ubuntu@ubuntu#ROPgadget --binary 1_staticnx – ropchain

Same can be done using ropper tool.

ubuntu@ubuntu#ropper --file 1_staticnx --chain system

Fourth and final step is to complete the payload, by adding offsets and appropiate address. Finding the addresses of syscall can be done using readelf and objdump tools.[9]

- **Ret2libc (Return ToLibc)**

Sometime we receive a binary which is small in size. These are dynamically linkedfiles.

ISSN: 0971-1260
Vol-22-Issue-16-August-2019

Dynamically Linked v/s Statically Linked Files:

Dynamically linked binary files are of lesser size and uses system`s libc files.
Statically linked binary files are of bigger size and are already compiled with libc file inside it.This can be checked by checksec tool. [14]
Issue in small sized binary file is that we are short of gadgets. There we try to use functions of libc which contains syscall such as system().
Now we can calculate the addresses of useful functions given the base address of libc. Since there is no randomisation due to ASLR being disabled, we can very easily find the addresses of useful things such as the `system()` function and the `/bin/sh` string by examining the libc shared object on its own.[17]

- **Stack Canary**

Stack canary is a reasonable method to alleviate ordinary stack crushing since it is genuinely difficult to figure an irregular worth. Be that as it may, spilling and savage constraining the canary are the two different ways which can help us in bypassing canary check.

- **LeakingCanary**

Here, the idea is to read the stack canary value and send it to program,as it remains same everytime. However,in Linux the first byte which is having eight bits of the stack canary is a NULL, function string will be stopped whenpointed to NULL.

  - User-controlled formatstring.
  - User-controlled length of anoutput.[13]

- **Bruteforcing Canary-**

The canary is set when the program is executed for the first time which means if the program forks, it keeps the same canary value in the child process.[7] This means that we can overwrite canary of child through input and check if it crashes. This way we can find 1 byte at a time.

- **BypassingASLR**

Now we have randomized addresses which makes it difficult to get the address of our vulnerable function or syscall and place it at exact return address.But also the question here arises, if addresses are randomized then how the program is executing normally, there have to be a way that the program is able to fetch the exact address.The answer is Yes. And the answer lies in Global Offset Table (GOT) And Procedure Linkage Table (PLT).[15]

ISSN: 0971-1260
Vol-22-Issue-16-August-2019

- **Global Offset Table**

Compiler sets a space to save the list of pointers(addresses) in the binary, to handle functions from dynamically loaded object. This space is divided into slots ,which is to be filled, called 'relocation' entry. The fact of GOT which is helpful for us is that this portion of memory is marked as readable which allows the values in slots to change during runtime. Lets take a program to understandthis:

Linux Terminal Output-

```
ubuntu@ubuntu-xenial:/$ readelf --relocs 1_clock
Relocation section '.rel.dyn' at offset 0x2dc contains 1 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
08049ffc  00000506 R_386_GLOB_DAT    00000000   __gmon_start__
Relocation section '.rel.plt' at offset 0x2e4 contains 5 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0804a00c  00000107 R_386_JUMP_SLOT   00000000   read@GLIBC_2.0
0804a010  00000207 R_386_JUMP_SLOT   00000000   printf@GLIBC_2
0804a014  00000307 R_386_JUMP_SLOT   00000000   puts@GLIBC_2.0
0804a018  00000407 R_386_JUMP_SLOT   00000000   system@GLIBC_2
0804a01c  00000607 R_386_JUMP_SLOT   00000000
libc_start_main@GLIBC_
```

Let's study the read entry in the GOT . If we analyse using gdb, and open the binary without running it, we can examine what actually is in the GOTinitially.

After derefencing(x/xw) that address from GOT , we get a value(0x08048643) which is also a address which is from PLT(Procedure Linkage Table). This is a mechanism to do lazy binding, which allows the program to resolve the randomised address or dynamic address to resolve when programs needs to. Now if we run the program and break before it ends, we can see the value now in GOT is completely different and points somewhere inlibc.

Procedure Linkage Table

Whenever we compile a program which uses libc function, the compiler instead of calling the particular function directly calls he PLT stub. Lets see at the disassembly of the read function in PLT.

gdb-peda$ disas read

Dump of assembler code for function read@plt: 0x08048340 <+0>: jmp DWORD PTR ds:0x804a00c

0x08048643 <+6>: push 0x0

0x0804834b <+11>: jmp 0x8048330

End of assemblerdump.

ISSN: 0971-1260
Vol-22-Issue-16-August-2019

How the excutionflows:

- The `read@plt` function is called.
- Execution proceeds and reaches 'jmp DWORD PTR ds:0x804a00c' and the address 0x804a00c is dereferenced and execution jumps. It is the address of the GOT entry of`read`.
- Since the GOT contained the value 0x08048643 initially, execution jumps to the next instruction of the `read@plt` function because that's where it pointsto.
- The program calls dynamic loader which changes the GOT with the resolvedaddress.
- And this resolved address is same thoughout execution, untilquit.

And that is how addresses are been randomised.

The fact that helps us is-It is the part of the binary and will be mapped at a static address which will not change. We can use it to exploit the program.

FORMAT STRINGVULNERABILITY

In this we take advantage of printf, sprintf function, to read and write arbitrary memoryaddress.

As we know printf() has a number of format specifiers like %d

%s %p %x %n etc.

Example- we have a program which print a normal string-

```
int main()
{
  ....
  buffer[100]
  //user input taken in buffer printf(buffer)
  ....
}
```

Here we can see the user`s input is saved in buffer variable and is printed via printf. To exploit it we give '%p' 15 times and the printf function will think it has 15 arguments as we gave gave format specifier as our input. It will simply print the next 15 addresses on

Linux Terminal Output-

```
$ ./a.out "%p %p %p %p %p %p %p %p %p %p %p %p %p %p %p"
0xffffdddd 0x64 0xf7ec1289 0xffffdbdf 0xffffdbde (nil)
0xffffdcc4 0xffffdc64 (nil) 0x25207025 0x70252070
0x20702520 0x25207025 0x70252070 0x20702520
```

ISSN: 0971-1260
Vol-22-Issue-16-August-2019

thestack-

We see a repeating pattern of 0x252070 , these are our %p on the stack, to be sure we start our input string with 'AAAA' , which will be shown as 0x41414141.

Linux Terminal Output-

```
$ ./a.out "AAAA%p %p %p %p %p %p %p %p %p %p"
AAAA0xffffdde8 0x64 0xf7ec1289 0xffffdbef 0xffffdbee (nil) 0xffffdcd4 0xffffdc74 (nil) 0x41414141
```

Here we see our 'AAAA', we are confirm that we can write on stack using printf().

Now we can overwrite any address using '%n' format specifier.

[6][16]

IV. CONCLUSION

Buffer overflow attack are complex yet dangerous vulnerability attack, which may lead to the worst. More protection schemes may come but the other side of securityi.e. hackers also try hard to mitigate that protection.These Vulnerability arises due to bad coding practice using old methods to code. Secure coding practice should be encouraged as well as regular testing ,both manual as well as automated, should be done. Another fix could be bound check at the run time,which checks the amount of data written to buffer. Program should be made in such a way that even if exploited ,cannot to much worse to the system,by providing least privilege toprograms.

V. REFERENCES

[1] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole, “Buffer overflows: Attacks and defenses for the vulnerability of the decade,” *Found. Intrusion Toler. Syst. OASIS 2003*, no. June 2014, pp. 227–237, 2003.

[2] B. A. Kuperman, C. E. Brodley, H. Ozdoganoglu, T. N. Vljaykumar, and A. Jalote, “Detection and prevention of stack buffer overflow attacks,” *Commun. ACM*, vol. 48, no. 11, pp. 51–56, 2005.

[3] S. Gupta, “Buffer Overflow Attack,” *IOSR J. Comput. Eng.*, vol. 1, no. 1, pp. 10–23, 2012.

[4] E. Levy, “Smashing The Stack For Fun And Profit,” *Phrack Mag.*, no. 49, 1996.

[5] “Mudge”. How to Write Buffer Overflows

ISSN: 0971-1260
Vol-22-Issue-16-August-2019

https://insecure.org/stf/mudge_buffer_overflow_tutorial.html [6]Format String Exploitation-Tutorial By Saif El-Sherei

[7] klog. The frame pointer overwrite. Phrack Magazinehttp://phrack.org/issues/55/8.html

[8] P. A. FAYOLLE and V. GLAUME, "A Buffer Overflow Study," *Attacks & Defenses*, 2002.

[9] Z. M. Gu, J. D. Yao, and J. Qin, "Buffer overflow attacks on linux principles analyzing and protection," *Dcabes 2002, Proceeding*, no. Beijing 100081, pp. 385–387, 2002.

[10] P. V. Murugan and D. K. Alagarsamy, "Buffer Overflow Attack Vulnerability in Stack," *Int. J. Comput. Appl.*, vol. 13, no. 5, pp. 1–2, 2011.

[11] V. Chatole, "Buffer overflow : Mechanism and countermeasures," vol. 4, no. 6, pp. 526–529, 2018.

[12] M. Prandini and M. Ramilli, "Return-oriented programming," *IEEE Secur. Priv.*, vol. 10, no. 6, pp. 84–87, 2012.

[13] https://ctf101.org/binary-exploitation/stack-canaries/

[14] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee, "ASLR-guard: Stopping address space leakage for code reuse attacks," *Proc. ACM Conf. Comput. Commun. Secur.*, vol. 2015-October, pp. 280–291, 2015.

[15] H. Marco-Gisbert and I. Ripoll, "On the {Effectiveness} of {Full-ASLR} on 64-bit {Linux}," 2014.

[16] B. Overflows and F. S. Vulnerabilities, "Exploiting Format String Vulnerabilities," *Based a speech given 17th Chaos Commun. Congr. 17C32000*, vol. 4, no. 10, pp. 1–31, 2001.

[17] B. S. El-sherei, "Return - to - libc."