C/C++ Tools for Dynamic Buffer Overflow Prevention

Irene Muzi, Vasile Claudiu Perta

9 giugno 2009

Indice

1. Introduzione
2. Analisi Statica
   - Caratteristiche principali dei tools
   - Confronto dei tools e risultati
3. Prevenzione dinamica del buffer overflow
   - Attacchi
   - Approcci utilizzati
   - Libsafe
   - StackShield
   - StackGuard e ProPolice
   - Mudflap e MIRO
   - Altri tools
4. Test e conclusioni

Affrontare il problema del buffer overflow

- Come affrontare e risolvere il problema del buffer overflow?

Affrontare il problema del buffer overflow

- Come affrontare e risolvere il problema del buffer overflow?

Buffer Overflow Prevention

- approccio proattivo
- analisi statica
- analizzare i sorgenti per rilevare e correggere le potenziali vulnerabilità

Affrontare il problema del buffer overflow

- Come affrontare e risolvere il problema del buffer overflow?

Buffer Overflow Prevention

- approccio proattivo
- analisi statica
- analizzare i sorgenti per rilevare e correggere le potenziali vulnerabilità

Buffer Overflow Detection

- approccio reattivo
- prevenzione dinamica
- modificare il run-time del processo e rilevare il buffer overflow quando si sta per verificare

Analisi Statica

- assistere il programmatore
- fase di implementazione e testing del software
- migliorare la qualità del software dal punto di vista della sicurezza

Analisi Statica

- assistere il programmatore
- fase di implementazione e testing del software
- migliorare la qualità del software dal punto di vista della sicurezza

Analisi lessicale

- ITS4
- Rats, Flawfinder
- BOON
- UNO

Analisi Statica

- assistere il programmatore
- fase di implementazione e testing del software
- migliorare la qualità del software dal punto di vista della sicurezza

Analisi lessicale

- ITS4
- Rats, Flawfinder
- BOON
- UNO

Parsing, model checking

- Splint
- MOPS
- BLAST

Tools per l'analisi statica

- la maggior parte non richiedono modifiche al codice sorgente
- database delle vulnerabilità personalizzabile dall'utente
- disponibili pubblicamente online

Tools per l'analisi statica

- la maggior parte non richiedono modifiche al codice sorgente
- database delle vulnerabilità personalizzabile dall'utente
- disponibili pubblicamente online

Analisi lessicale

- più semplici, molto veloci - dei *grep* evoluti
- alto numero di falsi positivi
- flow-insensitive

Tools per l'analisi statica

- la maggior parte non richiedono modifiche al codice sorgente
- database delle vulnerabilità personalizzabile dall'utente
- disponibili pubblicamente online

Analisi lessicale

- più semplici, molto veloci - dei *grep* evoluti
- alto numero di falsi positivi
- flow-insensitive

Parsing, model checking

- numero ridotto di falsi positivi
- più complessi, flow-sensitive (problemi di usabilità)
- spesso richiedono annotazioni, oppure definire la proprietà di sicurezza

Confronto dei tools e risultati

- tools commerciali: `ARCHER`, `PolySpace`
- free tools (open source): `ITS4`, `Flawfinder`, `Splint`, `UNO`
- *test-suite* composta da 291 test-cases

(http://www.ll.mit.edu/mission/communications/ist/corpora/ComputerCorpora.html)

	ARCHER	PolySpace	Splint	ITS4	Flawfinder	UNO
Detections	264	290	164	20	32	151
False Allarms	0	7	70	4	2	0

Prevenzione dinamica del buffer overflow

- Modificare il contesto dell'esecuzione del programma
- Al verificarsi del buffer overflow il programma termina
- Overhead computazionale non indifferente
- Risolvono i *bug noti*

Obiettivo?

- Ma cosa deve impedire un tool di Prevenzione Dinamica?

Principalmente...

- Sovrascrittura del return pointer.
- Sovrascrittura di function pointer.
- ...

Esempio: obiettivo

parameters
return address
old base pointer
local variables: buf ...

Esempio: obiettivo

parameters
return address
old base pointer
local variables: buf ...

```
#include <stdio.h>
#include <string.h>

main()
{
  char buf[10];
  strcpy(buf, "Hello! I'm a buffer
        overflow");

}
```

Overflow!

parameters
return address
old base pointer
local variables: buf ↑
...

```
#include <stdio.h>
#include <string.h>

main()
{
  char buf[10];
  strcpy(buf, "Hello! I'm a buffer
        overflow");

}
```

Obiettivo

parameters
return address
old base pointer
local variables:
function pointer
buffer
parameters to function
return address
old base pointer
local variables

```
int whoops(int a)
{
  a++;
  printf("%d\n",a);
}

main(int argc, char ** argv)
{
  int (*p)(int)=NULL;
  char buf[10];

  p=&whoops;
  strcpy(buf,argv[1]);
  (*p)(1);
}
```

Obiettivo

parameters
return address
old base pointer
local variables:
function pointer buffer ↑
parameters to function
return address
old base pointer
local variables

```
int whoops(int a)
{
  a++;
  printf("%d\n",a);
}

main(int argc, char ** argv)
{
  int (*p)(int)=NULL;
  char buf[10];

  p=&whoops;
  strcpy(buf,argv[1]);
  (*p)(1);
}
```

Soluzioni run-time

Functions overloading
- Libsafe
- Libverify

Return address cloning
- MemGuard
- StackShield

Return address range-checking
- StackShield

Canary
- StackGuard
- ProPolice

Array bounds-checking
- Mudflap
- MIRO

Libsafe

Buffer Overflow Detection realizzato in tre fasi distinte:

- *Interception*
- *Safety check*
- *Violation handling*

Interception

`libsafe.so`

- shared library
- `libsafe.so` caricata prima di `libc.so` via `LD_PRELOAD`
- overloading, wrapper functions
- `strcpy()` sostituita con la `strncpy()`, altre sono reimplementate

Safety check

`__libsafe_raVariableP(void * addr)`

- restituisce 1 se `addr` è un indirizzo di ritorno oppure un frame pointer
- invocata per ogni parametro `%n` della `_IO_vfprintf()`

Safety check

`__libsafe_raVariableP(void * addr)`

- restituisce 1 se `addr` è un indirizzo di ritorno oppure un frame pointer
- invocata per ogni parametro `%n` della `_IO_vfprintf()`

`__libsafe_span_stack_frames(void * start_addr, void * end_addr)`

- restituisce 1 se `*start_addr` è `*end_addr` stanno in frame differenti
- parametri della funzione devono stare nello stesso stack frame

Safety check

`__libsafe_raVariableP(void * addr)`

- restituisce 1 se `addr` è un indirizzo di ritorno oppure un frame pointer
- invocata per ogni parametro `%n` della `_IO_vfprintf()`

`__libsafe_span_stack_frames(void * start_addr, void * end_addr)`

- restituisce 1 se `*start_addr` è `*end_addr` stanno in frame differenti
- parametri della funzione devono stare nello stesso stack frame

Limitazioni

- necessario determinare locazione e dimensione dgli stack-frames
- assunzioni sulla posizione dell'indirizzo di ritorno nello stack
- incompatibilità con *StackGuard*, *gcc -fomit-frame-pointer*

Violation handling

Azioni di default

- Termina il processo
- Aggiunge una entry in `/var/log/secure`
- Stampa un warning sullo standard error

Violation handling

temp@ubuntu~$ ./exploit

```
Libsafe version 2.0.16
Detected an attempt to write across stack boundary.
Terminating /home/temp/exploit
uid=1000, euid=1000, pid=29863
Call stack:
0xb802ba89 /home/temp/lib/libsafe.so.2.0.16
0xb802c5cc /home/temp/lib/libsafe.so.2.0.16
0x804859e /home/temp/exploit
0x80485ae /home/temp/exploit
0xb7ed66b1 /lib/libc-2.9.so
Overflow caused by strcpy()
29863 Killed
```

StackShield

- Implementato come preprocessore assembler

Vari tipi di controllo/protezione

- Return addres cloning
- Return range checking
- Function pointer protection

Return address cloning

- Salvare l'indirizzo di ritorno in una zona riservata di memoria
- *retarray* - contiene 256 locazioni
- *rettop* - indirizzo di memoria dove finisce `retarray`
- *retptr* - l'indirizzo dove verrà salvato il prossimo *return address*

```
.data
.comm shielddatabase,4,4

.data
.comm retptr,4,4
.comm rettop,4,4
.comm retarray,1024,4
```

Return address cloning

```
function_prologue:
    pushl   %eax
    pushl   %edx

    movl    retptr, %eax   /* retptr is where the clone is saved */
    cmpl    %eax, rettop   /* if retptr" is higher than allowed */
    jbe     .LSHIELDPROLOG
    movl    8(%esp), %edx  /* get the return address from stack */
    movl    %edx, (%eax)   /* save it in global space */
.LSHIELDPROLOG:
    addl    $4, retptr     /* allways increment retptr*/
```

Return address check

```
function_epilogue:
    addl  $-4, retptr      /* allways decrement retptr */
    movl  retptr, %eax
    cmpl  %eax, rettop     /* is retptr in the reserved memory ? */
    jbe   .LSHIELDEPILOG
    movl  (%eax), %edx
    cmpl  %edx, 8(%esp)    /* compare clone to stack content */
    je    .LSHIELDEPILOG   /* if equals, keep going */
    movl  $1, %eax
    movl  $-1, %ebx
    int   $0x80            /* abort program execution (SYS_exit()) */

.LSHIELDEPILOG:
    popl   %edx
    popl   %eax
    ret
```

Return range checking

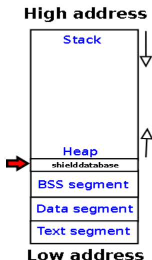
Nuova variabile `shielddatabase`

- indirizzo base di riferimento per i dati del programma
- bloccati tentativi di ritorno a locazioni più alte di memoria

Return range checking

Nuova variabile shielddatabase

- indirizzo base di riferimento per i dati del programma
- bloccati tentativi di ritorno a locazioni più alte di memoria




```
function_epilogue:
  cmpl $shielddatabase, (%esp)
  jbe  .LSHIELDRETRANGE /*return to a high address*/

  movl $1, %eax
  movl $-1, %ebx
  int  $0x80 /*abort program execution*/

.LSHIELDRETRANGE:
  ret /*jump to address on stack's top*/
```

StackGuard

- Implementato come una modifica al gcc
- *Terminator canary* vs *Random canary*

```
function_prologue:
    pushl  $0x000aff0d /* push the terminator canary into the stack */

function_epilogue:
    cmpl  $0x000aff0d, (%esp) /* check canary */
    jbe  .CANARY_CHANGED /* canary changed */
    addl $4, %esp
    ret

.CANARY_CHANGED:
    call __canary_death_handler /* abort the program with error */
```

Limitazioni StackGuard

"Emsi's attack"

- Sovrascrivere il *return address* saltando il *canary*
- Manipolare un puntatore
- Puntare al indirizzo di ritorno e poi sovrascriverlo
- *Terminator canary* inutile in questo caso

Emsi's attack - soluzione

Random canary

- Invece del *terminator canary* usiamo:
  - *Random canary* $\oplus$ *Return address*

Emsi's attack - soluzione

Random canary

- Invece del *terminator canary* usiamo:
  - *Random canary* $\bigoplus$ *Return address*

Crispy Cowan, autore di StackGuard:

"There is only one threat that the XOR canary defeats, and the terminator canary does not: the Emsi's attack. However, if you have a vulnerability that enables you to deploy Emsi's attack, then you have many other targets to attack besides functions return address values. Therefore, we dropped support for random canaries."

ProPolice

- Evoluzione di StackGuard
- Implementato come patch al compilatore
- Incluso nelle ultime versioni del gcc
  - `gcc -fstack-protector`

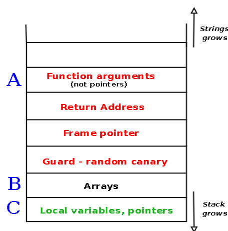
ProPolice

- Evoluzione di StackGuard
- Implementato come patch al compilatore
- Incluso nelle ultime versioni del gcc
  - `gcc -fstack-protector`

Design goal

- Introdurre un *modello sicuro* di utilizzo dello stack
- Trasformare lo stack del programma *riordinando* le variabili
- Cambiare lo stack, più vicino possibile al *layout ideale*

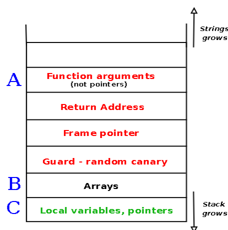
Safe Stack Usage Model



Ideal Stack Layout

- **A** non contiene array o puntatori
- **B** contiene solo array
- **C** contiene le variabili locali e i puntatori

Safe Stack Usage Model



Ideal Stack Layout

- **A** non contiene array o puntatori
- **B** contiene solo array
- **C** contiene le variabili locali e i puntatori

Stack integrity check

- Random canary all'inizio della funzione
- Se il check alla fine fallisce, termina il programma
- Emsi's attack non funziona più

ProPolice - Violation Handling

temp@ubuntu~$ ./exploit

```
*** stack smashing detected ***: ./exploit terminated
======= Backtrace: =========
/lib/libc.so.6(__fortify_fail+0x48)[0xb7efcf78]
/lib/libc.so.6(__fortify_fail+0x0)[0xb7efcf30]
./exploit[0x804860f]
======= Memory map: ========
08048000-08049000 r-xp 00000000 08:0d 1000080 /home/temp/exploit
08049000-0804a000 rw-p 00000000 08:0d 1000080 /home/temp/exploit
089a5000-089c6000 rw-p 089a5000 00:00 0 [heap]
b7e19000-b7e1a000 rw-p b7e19000 00:00 0
b7e1a000-b7f58000 r-xp 00000000 08:0c 353925 /lib/libc-2.9.so
b7f5a000-b7f5b000 rw-p 0013f000 08:0c 353925 /lib/libc-2.9.so
b7f5b000-b7f5f000 rw-p b7f5b000 00:00 0
b7f65000-b7f82000 r-xp 00000000 08:0c 392338 /usr/lib/libgcc_s.so.1
b7f82000-b7f83000 rw-p 0001c000 08:0c 392338 /usr/lib/libgcc_s.so.1
b7f83000-b7f85000 rw-p b7f83000 00:00 0
b7f85000-b7f86000 r-xp b7f85000 00:00 0 [vdso]
b7f86000-b7fa2000 r-xp 00000000 08:0c 354130 /lib/ld-2.9.so
b7fa2000-b7fa3000 r--p 0001b000 08:0c 354130 /lib/ld-2.9.so
bfc8f000-bfca4000 rw-p bffeb000 00:00 0 [stack]

Aborted
```

Mudflap

Checking Dereferenced Pointers

- Supporto per C e C++
  - `gcc -fmudflap -lmudflap`
- Modifiche al gcc per aggiungere i controlli sui puntatori
- Database delle regioni di memoria allocate
- Libreria runtime per effettuare i check: `libmudflap`

libmudflap - registrare una zona di memoria

```
void foo(int n)
{
    int a[4];
    int  *p, i = 3;

    __mf_register (&a, 16);
    __mf_register (&i, 4);

    /* ... */
}
```

```
void * malloc(size_t c)
{
    void * result = __real_malloc(c);

    if (result) {
        __mf_register (result, c);
    }

    return result;
}
```

libmudflap - check pointer dereferencing

`__mf_check()`

- La zona di memoria acceduta deve essere stata registrata
- Termina l'esecuzione del programma in caso contrario

```
void bar(int n)
{
    int a[4]; int i = 3; int *p;

    /* dereferencing p */
    __mf_check (p, 4);
    *p = 0;

    /* dereferencing a */
    __mf_check (&a, n);
    a[n] = 7;
}
```

libmudflap - Violation handling

temp@ubuntu~$ ./exploit

```
*******************
mudflap violation 1 (check/write): time=1244303540.812733 ptr=0xbfe5a12c size=16
pc=0xb7f3633d location=(strcpy dest)
/usr/lib/libmudflap.so.0(__mf_check+0x3d) [0xb7f3633d]
/usr/lib/libmudflap.so.0(__mfwrap_strcpy+0xcc) [0xb7f4382c]
./exploit(main+0xe1) [0x80488d1]
Nearby object 1: checked region begins 0B into and ends 6B after
mudflap object 0x8661ba0: name=exploit.c:8:7 (main) b
bounds=[0xbfe5a12c,0xbfe5a135] size=10 area=stack check=0r/3w liveness=3
alloc time=1244303540.812712 pc=0xb7f35acd
Nearby object 2: checked region begins 10B before and ends 5B into
mudflap object 0x8661b38: name=exploit.c:7:7 (main) a
bounds=[0xbfe5a136,0xbfe5a13f] size=10 area=stack check=0r/3w liveness=3
alloc time=1244303540.812711 pc=0xb7f35acd

number of nearby objects: 2
```

E se i buffer sono adiacenti?

```
#include<stdio.h>
#include<string.h>

int main(int argc, char *argv[])
{
    char a[1024], b[10];
    char *s = b;
    char *ss = argv[1];

    /* Overflow a if argv[1] is big enough! */
    for (i = 0; i < strlen(ss); i++) {
        *s = *ss;
        s++; ss++;
    }
    return 0;
}
```

MIRO

Mudflap Improved with Reference Object

- Aggiunge il controllo sull'aritmetica dei puntatori
- Migliora il supporto per C++
- Patch al gcc, ancora non incluso nella release standard

```
int a[4]; int i = 3; int *q;
int *p = &a[0];

q = p + i;

/* base pointer: p */
q = __bounds_arith (p, p+i);
```

Altri tools

MemGuard

- Protegge le pagine di memoria contenenti dati sensibili
- Installa un `trap_handler()` che intercetta gli accessi in scrittura
- Blocca tutte le write sulle variabili protette e termina il processo
- Scritture fino a 1800 più lente

PointGuard

- Usa la cifratura
- Genera una chiave `k` nell'`exec()`
- I puntatori in memoria sono cifrati ($\oplus$ `k`)
- Decifratura solo quando vengono caricati nei registri
- Se succede un overflow viene dereferenziato un dato random

Altri tools

TIED + LibsafePlus

- Type Information Extractor and Depositor
- Estrae informazioni dall'eseguibile (ELF)
- Richiede la compilazione con `-g`
- Inserisce nel binario informazioni per il run-time bound-checking
- LibsafePlus - intercetta le funzioni vulnerabili, ed effettua il bound-checking

LibsafeXP

- Stesso meccanismo di TIED e LibsafePlus
- Non necessita le informazioni di debugging, overhead minimo
- Bound sugli array solo *approssimativi*
- Non previene del tutto i buffer overflow ma riduce le loro dimensioni

Altri tools - non testati

计算机软件新技术国家重点实验室(南京大学)
State Key Laboratory for Novel Software Technology at Nanjing University

首 页 | 实验室概况 | 实验室组织 | 科学研究 | 人才队伍 | 学术交流 | 开放课题 | 其 它
2009年6月7日

科学研究

- 定位
- 研究方向
- 总体目标
- 科研项目
- 代表性成果
- 论文
- 专著
- 获奖
- 专利
- 软件著作权

LibsafeXP: A Practical and Transparent Tool for Run−time Buffer Overflow Preventions

论著名称：LibsafeXP: A Practical and Transparent Tool for Run−time Buffer Overflow Preventions

作者：Zhiqiang Lin, Bing Mao and Li Xie

第一单位：计算机软件新技术国家重点实验室

国际会议：7th Annual IEEE Information Assurance Workshop (IAW'06)

Test su varie tipologie di buffer overflow

	ProPolice	Mudflap	MIRO	libsafe
Return address	detected	detected	detected	detected
Function pointer	detected	detected	detected	missed
Function pointer as parameter	detected	detected	detected	missed
Longjmp buffer	detected	detected	detected	missed
Longjmp buffer as parameter	detected	detected	detected	missed
Function pointer on the heap	missed	detected	detected	missed
Longjmp buffer on the heap	missed	detected	detected	missed
Format string	missed	missed	missed	abnormal
Direct access to buffer	missed	detected	detected	missed

Test su vulnerabilità note

- 14 vulnerabilità note
- bind(4), sendmail(7) wu-ftpd(3)

(http://www.ll.mit.edu/mission/communications/ist/corpora/ComputerCorpora.html)

	ProPolice	Mudflap	MIRO	libsafe	StackShield
bind	1	1	4	1	1
sendmail	3	1	7	2	3
wu-ftpd	1	2	3	2	2

Overhead introdotto dai tools run-time

- ctags compilato con i vari tools
- 1Mb di files sorgenti (*capwap*)
- 200Mb di files sorgenti (*linux filsystem kernel*)

ctags	ProPolice	Mudflap	MIRO	libsafe	gcc
1 Mb	*0.08s*	*1.73s*	*1m 53s*	*0.08s*	*0.08s*
200 Mb	*4.87s*	*2m 21s*	*still executing...*	*19.1s*	*4.5s*