Linguagem C

Diagnóstico e correcção de problemas

José Pedro Oliveira
(jpo@di.uminho.pt)

Grupo de Sistemas Distribuídos
Departamento de Informática
Escola de Engenharia
Universidade do Minho

Sistemas Operativos I
2006-2007

Conteúdo

Introdução

Técnicas e ferramentas

- Arquitectura física
- Linguagem de programação
- Análise estática de programas C
- Compilador de C
- Bibliotecas de apoio a debugging
- Debuggers genéricos
- Debuggers especializados

Arquitectura física

Informação a ter em conta

- Tamanho da palavra do processador
- Endereçamento de memória
  - Processador com unidade de gestão de memória
  - Alinhamento
- *byte-order* do processador
  - little-endian vs big-endian
- `sizeof(int) != sizeof(void *)`

Linguagem de programação C

Algumas funcionalidades e propriedades a explorar

- Utilizar assert.h
- Substituir
  - `gets` por `fgets`
  - `strcpy` por `strncpy`
  - `strcat` por `strncat`
  - `sprintf` por `snprintf`
  - ...
- Substituir
  - `if (p == 0)` por `if (0 == p)`

Análise estática de programas C - splint

Utilização

```
$ splint +flag1 -flag2 ficheiro.c
```

Ficheiro de configuração: **.splintrc**

```
### Mode selection flags
#      weak, standard (default), checks, strict

+checks

### Display Flags

#+showscan
+showsummary
+stats
```

Compilador de C - gcc

Algumas opções úteis

-E	- Executa apenas o passo de pré-processamento
-S	- Gera o ficheiro assembly (.s)
-g	- Gera informação de debugging
-Wall	- Activa a grande maioria dos avisos
-Wextra	- Activa ainda mais avisos
-Werror	- Os avisos passam a ser considerados erros
-On	- Nível de optimização (por omissão: -O0)
-Wp,-D_FORTIFY_SOURCE=n	- detecção de buffer overflows

Compilador de C - gcc

Exemplos

```
gcc -E ...
gcc -S ...
gcc -Wall -Wextra ...
gcc -Wall -Wextra -std=c99 ...
gcc -Wall -Wextra -std=c99 -O0 -g ...
gcc -O2 -D_FORTIFY_SOURCE=2 ...
gcc -Wall -Wextra -Werror -O2 -Wp,-D_FORTIFY_SOURCE=2 ...
gcc -Wall -Wextra -g -lefence ...
gcc -Wall -Wextra -g -fmudflap -lmudflap ...
```

Fortify Source

FORTIFY_SOURCE

FORTIFY_SOURCE é uma característica adquirida pelo gcc e pela glibc que permite detectar e prevenir um subconjunto de buffers overflows.

Mecânica

Há diversas situações em que o compilador consegue saber qual a dimensão de um buffer (alocado estaticamente ou alocado dinamicamente via malloc). Com este conhecimento, diversas funções que sobre ele irão operar podem garantir a não existência de buffer overflows.

Fortify Source: exemplo 1

exemplo1.c - detecção: durante a compilação e a execução

```
1  #include <string.h>
2
3  int main(void)
4  {
5      char str[4];
6
7      strcpy(str, "1234");
8
9      return 0;
10 }
```

Fortify Source: exemplo 1

$ gcc -O2 -Wp,-D_FORTIFY_SOURCE=2 exemplo1.c

```
exemplo1.c: In function 'main':
exemplo1.c:7: warning: call to __builtin___strcpy_chk will \
                       always overflow destination buffer
```

$ ./a.out

```
*** buffer overflow detected ***: ./a.out terminated
======= Backtrace: =========
/lib/libc.so.6(__chk_fail+0x41)[0xa6ec45]
/lib/libc.so.6(__strcpy_chk+0x3f)[0xa6e2d7]
./a.out[0x80483bc]
/lib/libc.so.6(__libc_start_main+0xdf)[0x9a5d5f]
./a.out[0x804831d]
======= Memory map: ========
00973000-0098d000 r-xp 00000000 03:09 972530   /lib/ld-2.3.5.so
0098d000-0098e000 r-xp 00019000 03:09 972530   /lib/ld-2.3.5.so
...
```

Fortify Source: exemplo 2

exemplo2.c - detecção: durante a execução

```
1  #include <string.h>
2
3  int main(int __attribute__((unused)) argc, char *argv[])
4  {
5      char str[4];
6
7      strcpy(str, argv[1]);
8
9      return 0;
10 }
```

Fortify Source: exemplo 2

$ gcc -O2 -Wp,-D_FORTIFY_SOURCE=2 exemplo2.c
$ ./a.out ok
$ ./a.out overflow

```
*** buffer overflow detected ***: ./a.out terminated
======= Backtrace: =========
/lib/libc.so.6(__chk_fail+0x41)[0xa6ec45]
/lib/libc.so.6(__strcpy_chk+0x3f)[0xa6e2d7]
./a.out[0x80483bd]
/lib/libc.so.6(__libc_start_main+0xdf)[0x9a5d5f]
./a.out[0x804831d]
======= Memory map: ========
00973000-0098d000 r-xp 00000000 03:09 972530   /lib/ld-2.3.5.so
0098d000-0098e000 r-xp 00019000 03:09 972530   /lib/ld-2.3.5.so
...
```

Fortify Source: referências

Referências

- **Limiting buffer overflows with ExecShield**
  `http://www.redhat.com/magazine/009jul05/features/execshield/`
- **Object size checking to prevent (some) buffer overflows**
  `http://gcc.gnu.org/ml/gcc-patches/2004-09/msg02055.html`

Biblioteca Electric Fence

Biblioteca Electric Fence

Permite detectar dois erros comuns associados à alocação dinâmica de memória:

- ultrapassar os limites do bloco de memória alocado
- aceder a memória já libertada com free()

Mecânica

Esta biblioteca utiliza o hardware de gestão de memória virtual do computador para colocar uma página de memória inacessivel imediatamente depois (ou antes) de cada bloco de memória alocado. Quando o software aceder em modo de leitura ou de escrita a uma das páginas inacessíveis, o hardware irá gerar um *segmentation fault*.

Biblioteca Electric Fence

Utilização

- Linkar o programa com a biblioteca **efence**
- Executar o programa modificando o comportamento da biblioteca através das variáveis de ambiente:
  - EF_ALIGNMENT
  - EF_PROTECT_BELOW
  - EF_PROTECT_FREE
  - EF_ALLOW_MALLOC_0
  - EF_FILL

Permitir a geração de ficheiros core

```
$ ulimit -c unlimited
```

Biblioteca Electric Fence: exemplo

exemplo.c

```c
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char *str = (char *) malloc( 5 );

    strcpy(str, "12345");
    *(str - 1) = '\0';
    free(str);
    *str = '\0';

    return 0;
}
```

Biblioteca Electric Fence: exemplo

Compilar o programa

gcc -Wall -Wextra -O0 -g -lefence -o exemplo exemplo.c

Executar o programa (gerar core dumps)

1. ./exemplo
2. EF_ALIGNMENT=1 ./exemplo
3. EF_PROTECT_BELOW=1 ./exemplo

Utilizar o debugger para localizar a fonte do problema

gdb exemplo core.pid

Biblioteca Electric Fence: exemplo

$ ./exemplo

```
  Electric Fence 2.2.0 Copyright (C) 1987-1999 Bruce Perens <bruce@per
Segmentation fault (core dumped)
```

$ gdb exemplo core.3542

```
GNU gdb Red Hat Linux (6.3.0.0-1.134.fc5rh)
...
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x0804849a in main () at efence.c:11
11              *str = '\0';
(gdb)
```

Valgrind

Valgrind 3.2

Valgrind é um sistema flexível para realizar o *debugging* e *profiling* de executáveis X86/Linux, AMD64/Linux, PPC32/Linux e PPC64/Linux. O sistema consiste num núcleo, que fornece um processador sintético, e num conjunto de ferramentas que permitem realizar tarefas especificas de debugging ou profiling. Uma das principais ferramentas é a **memcheck** que permite detectar problemas associados à gestão de memória de programas.

Algumas ferramentas

cachegrind - simulador de cache

memcheck - verificador de memória (ferramenta por omissão)

Valgrind

Opções de linha de comando

- ficheiro **.valgrindrc** no directório de trabalho
- ficheiro **.valgrindrc** na *homedir* do utilizador
- variável de ambiente **VALGRIND_OPTS**

Exemplo de algumas opções de linha de comando

-v - imprime informação adicional (verbose)

-q - modo silencioso; apenas imprime mensagens de erro (quiet)

--tool=callgrind - selecionar ferramenta callgrind

--leak-check=yes - detectar perdas de memória (valor por omissão: summary)

Valgrind: exemplo

exemplo.c

```c
#include <stdlib.h>

int main(void)
{
    int *p = malloc(10 * sizeof(int));

    p[10] = 0;
    *(p - 1) = 0;

    return 0;
}
```

Valgrind: exemplo

Compilar o programa

(desactivando optimizações e activando informação de debugging)

gcc -Wall -Wextra -O0 -g -o exemplo exemplo.c

Executar o programa

- valgrind exemplo
- valgrind -q exemplo
- valgrind --tool=memcheck exemplo
- valgrind --tool=memcheck -v exemplo
- valgrind --tool=memcheck --leak-checks=no exemplo
- valgrind --tool=memcheck --leak-checks=yes exemplo

Valgrind: exemplo

$ valgrind --tool=memcheck exemplo

```
...
==32764== Invalid write of size 4
==32764==    at 0x80483AE: main (exemplo.c:7)
==32764==  Address 0x1B92F050 is 0 bytes after a block of size 40 alloc'd
==32764==    at 0x1B909222: malloc (vg_replace_malloc.c:130)
==32764==    by 0x80483A1: main (exemplo.c:5)
==32764==
==32764== Invalid write of size 4
==32764==    at 0x80483BA: main (exemplo.c:8)
==32764==  Address 0x1B92F024 is 4 bytes before a block of size 40 alloc'd
==32764==    at 0x1B909222: malloc (vg_replace_malloc.c:130)
==32764==    by 0x80483A1: main (exemplo.c:5)
==32764==
==32764== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 13 from 1)
==32764== malloc/free: in use at exit: 40 bytes in 1 blocks.
==32764== malloc/free: 1 allocs, 0 frees, 40 bytes allocated.
==32764== For counts of detected errors, rerun with: -v
==32764== searching for pointers to 1 not-freed blocks.
==32764== checked 49152 bytes.
==32764==
==32764== LEAK SUMMARY:
==32764==    definitely lost: 40 bytes in 1 blocks.
==32764==      possibly lost: 0 bytes in 0 blocks.
==32764==    still reachable: 0 bytes in 0 blocks.
==32764==         suppressed: 0 bytes in 0 blocks.
...
```

Valgrind: referências

Referências

- **Valgring homepage**
  `http://valgrind.org/`
- **The Valgrind Quick Start Guide**
  `http://valgrind.org/docs/manual/quick-start.html`
- **Valgrind User Manual**
  `http://valgrind.org/docs/manual/manual.html`

Conteúdo

Overflow de buffers estáticos

Experimentar

- Utilizar o splint
- Activar avisos do gcc
- Compilar com a opção _FORTIFY_SOURCE
- Corrigir o programa

Exemplo 1 - overflow de um buffer estático

estatico_1.c

```
1  #include <string.h>
2
3  int main(void)
4  {
5      char str[4];
6
7      strcpy(str, "1234");
8
9      return 0;
10 }
```

Exemplo 2 - overflow de um buffer estático

estatico_2.c

```
1  #include <string.h>
2
3  int main(void)
4  {
5      char str[4] = "N:";
6
7      strcat(str, "01");
8
9      return 0;
10 }
```

Exemplo 3 - overflow de um buffer estático

estatico_3.c

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main(void)
5  {
6      char str[4] = "N";
7
8      sprintf(str, "%s:%d\n", str, 1);
9
10     // printf("String = <%s>\n", str);
11
12     return 0;
13 }
```

Exemplo 4 - overflow de um buffer estático

estatico_4.c

```
1  #include <stdlib.h>
2
3  int main(void)
4  {
5      int i;
6      int a[10];
7
8      for (i=0; i<=10; i++) {
9          a[i] = 0;
10     }
11
12     return 0;
13 }
```

Overflow de buffers alocados dinamicamente

Experimentar

- Utilizar o splint
- Activar avisos do gcc
- Compilar com a opção _FORTIFY_SOURCE
- Utilizar a biblioteca Electric Fence
- Correr o programa com o Valgrind
- Corrigir o programa

Exemplo 1 - overflow de um buffer alocado dinamicamente

dinamico_1.c

```c
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char *str = (char *) malloc( 4 );

    strcpy(str, "1234");

    free(str);

    return 0;
}
```


Exemplo 2 - overflow de um buffer alocado dinamicamente

dinamico_2.c

```c
#include <stdlib.h>

int main(void)
{
    int i, *p = malloc(10 * sizeof(int));

    for (i=0; i<=10; i++) {
        p[i] = i;
    }

    free(p);

    return 0;
```


Exemplo 3 - overwrite de buffers

dinamico_3.c

```c
#include <stdlib.h>

int main(void)
{
    int i;
    int *p1 = malloc(10 * sizeof(int));
    int *p2 = malloc(10 * sizeof(int));

    for (i=0; i<10; i++) { p2[i] = 2; }
    for (i=0; i<15; i++) { p1[i] = 1; }

    return 0;
}
```


Conteúdo

Outros problemas associados à alocação de memória

Experimentar

- Utilizar o splint
- Activar avisos do gcc
- Compilar com a opção _FORTIFY_SOURCE
- Utilizar a biblioteca Electric Fence
- Correr o programa com o Valgrind
- Corrigir o programa

Exemplo 1 - problema associado à alocação de memória

memoria_1.c

```c
#include <stdlib.h>

static void f(void) {
    int *p1 = NULL;

    p1 = (int *) malloc(sizeof(int));
    *p1 = 0;
}

int main(void) {
    f();
    return 0;
}
```


Exemplo 2 - problema associado à alocação de memória

memoria_2.c

```c
#include <stdlib.h>

static void f(void) {
    int *p1 = NULL;

    p1 = (int *) malloc(sizeof(int));
    free(p1);
    *p1 = 0;
}

int main(void) {
    f();
    return 0;
}
```


Exemplo 3 - problema associado à alocação de memória

memoria_3.c

```c
#include <stdlib.h>

static void f(void) {
    int *p1 = NULL;

    p1 = (int *) malloc(sizeof(int));
    *p1 = 0;
    free(p1);
    free(p1);
}

int main(void) {
    f();
    return 0;
}
```

Exemplo 4 - problema associado à alocação de memória

memoria_4.c

```c
#include <stdlib.h>

static void f(void) {
    int *p1 = NULL, *p2;

    p1 = (int *) malloc(sizeof(int));
    *p2 = 0;
    free(p1);
}

int main(void) {
    f();
    return 0;
}
```

Exemplo 5 - endereço de memória inválido

memoria_5.c

```c
#include <stdlib.h>

int * f(int i) {
    int p[10];
    p[i] = 1;
    return &p[i];
}

int main(void) {
    int *pi = f(4);
    int i = *pi;
    return 0;
}
```

Exemplo 6 - endereço de memória inválido

memoria_6.c

```c
#include <stdlib.h>

int * f(int i) {
    int *p = (int *) malloc(10 * sizeof(int));
    p[i] = 1;
    free(p);
    return &p[i];
}

int main(void) {
    int *pi = f(4);
    int i = *pi;
    return 0;
}
```