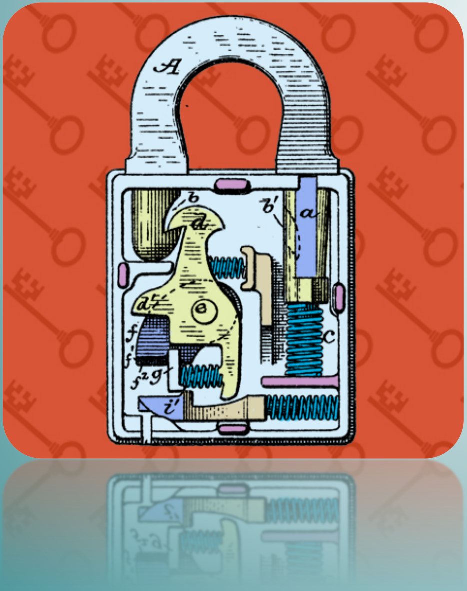
Garantía y Seguridad en Sistemas y Redes

Tema 7. Buffer Overflow



Esteban Stafford

Departamento de Ingeniería Informática y Electrónica

Contents

Buffer Overflows

Defending against buffer Overflows

Other forms of overflow attacks

Overflow vulnerabilities

- Overview
  - Known since 1988 (Morris Internet Worm).
  - Techniques for preventing them exist.
  - Still cause for many exploits.
  - `http://www.sans.org/top25-software-errors`
  - Called buffer|stack|heap overflow|overrun|smashing
- History
  - 1988 the Morris worm
  - 2001 the Code Red worm exploits MS IIS 5.0
  - 2003 the Slammer worm exploits MS SQL Server 2000
  - 2003 exploits for Xbox, PlayStation2 and Wii
  - 2004 the Sasser worm exploits MS Windows XP
  - ...

Basic buffer overflow

```
int main() {
   int valid = FALSE;
   char str2[8];
   char str1[8];
   strcpy(str1,"secret");
   gets(str2);
   if(strncmp(str1, str2, 7) == 0)
      valid = TRUE;
   printf("str1=’%s’␣str2=’%s’␣valid=%d\n",str1,str2,valid);
   return valid;
}
```

```
$ ./checkpasswd
12345
str1=’secret’ str2=’12345’ valid=0
$ ./checkpasswd
secret
str1=’secret’ str2=’secret’ valid=1
```

```
$ ./checkpasswd
1234567         1234567
str1=’1234567’ str2=’1234567␣␣␣␣␣␣␣␣␣1234567’ valid=1
```

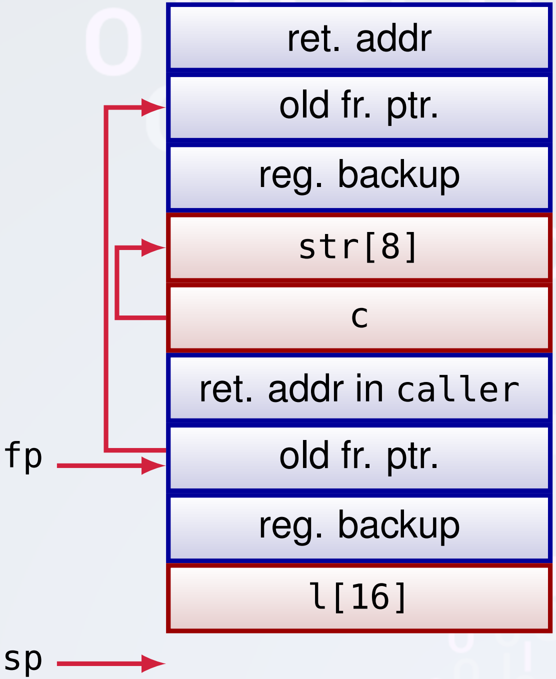
Basic buffer overflow

- How can this be possible?
  - Processor architecture is oblivious of data type length or structure.
  - Assembly and some high level languages (C) do not implement *strong typing*.
  - Strong typing and boundary checking is expensive.
- What can be done to prevent it?
  - Program carefully!
  - Identify vulnerabilities with *input fuzzing*
  - Use languages with strong typing: Java, ADA, Python...
  - Beware of legacy code.

Stack buffer overflow (Stack smashing)

- Overflowed buffer within the stack.
- Let’s refresh the function call mechanism.

```
int func(char *c) {
 char l[16];
 ...
}
int caller() {
 char str[8];
 func(str);
}
```



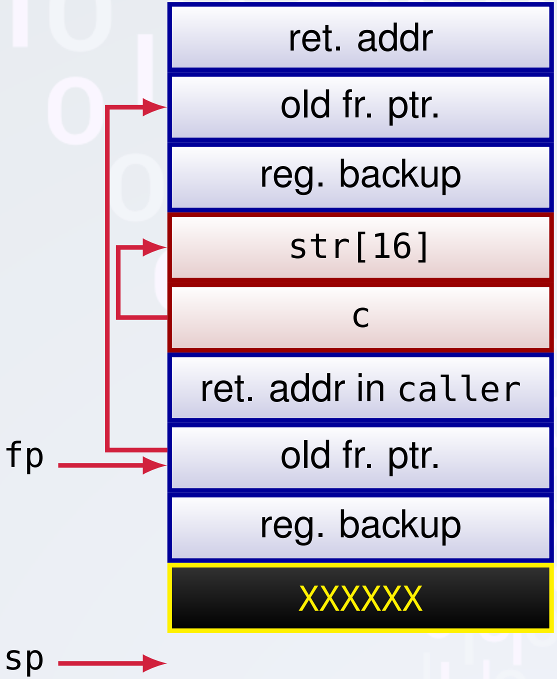
Stack buffer overflow (Stack smashing)

- Stack overflow aims to overwrite the return address and frame pointer in the stack.
- From an attacker's perspective:
  - Can cause segmentation when function returns: DoS.
  - Can execute code in the process' virtual space: Change program's behaviour.
- Finding what input causes the function call return to change is not easy
- Depends on processor architecture and compiler.
- Knowing the address of a desired piece of code is also hard.
- Can depend on operating system and runtime.
- Attackers can inject external code and run it (Shell code).

Stack buffer overflow (Stack smashing)

```
int func(char *c) {
 char l[8];
 strcat(l,c);
 ...
}
int caller() {
 char str[16];
 gets(str);
 func(str);
}

char sh[]="/bin/sh";
char *args[]={"sh",NULL};
void sh_code() {
 execve(sh,args,NULL);
}
```



Shell code

- Code supplied by attacker with alternate behaviour:
  - Traditionally transferred control to a user command-line interpreter (shell)
  - Create a reverse shell that connects back to the intruder.
  - Disable firewall rules that could block other attacks.
  - Break out of `chroot` or jail environments, allowing full access.
- Its machine code:
  - Specific to processor and operating system
  - Traditionally needed good assembly language skills to create
  - More recently a number of sites and tools have been developed that automate this process.
- Metasploit Project: provides useful information to people who perform penetration, IDS signature development, and exploit research.

Defending against buffer overflows

- Development stage
  - Choose overflow free language.
  - Code overhead might not be suitable for all aplications.
  - Program not only for success or the expected.
  - Be constantly aware of what can go wrong. Graceful failure.
  - Avoid unsafe libraries or legacy code (OpenBSD).
  - `gets`, `sprintf`, `strcat`, `strcpy`...
  - When writing to a buffer, check for enough room.

```c
int copy_buf(char *to, int pos, char *from, int len) {
   for(int i=0; i<len; i++) {
      to[pos] = from[i];
      pos++;
   }
   return pos;
}
```

Defending against buffer overflows

- Compiling stage
  - Compiler extensions check boundaries automatically.
  - Good for static arrays. Not so much for pointers and dynamic arrays.
  - Unsafe C code can not be converted to safe C code.
  - Stack protection: Function call mechanism detects corruption in stack frame and aborts the process.
    - Non-standard stack frame: `-stack-protector` used by default in Ubuntu and the like.
    - Standard stack frame: Stackshield or Return Address Defender.

```
$ ./checkpasswd
1234567         12345671234567         1234567
str1='12345671234567␣␣␣␣␣␣␣␣␣1234567' str2='1234567␣␣'[...] valid=1
*** stack smashing detected ***: ./checkpasswd terminated
Aborted (core dumped)
```

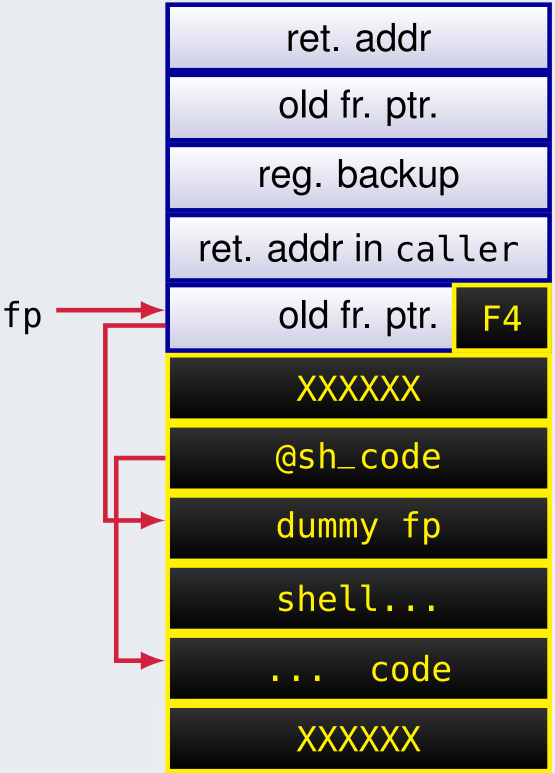
Defending against buffer overflows

- Execution stage
  - Run programs with `libsafe`. DLL that avoids return address overwriting.
  - Frobid execution of code in the stack (MMU)
  - Some programs need to execute on the stack: Just-in-time compilers, linux signal handlers.
  - Address space randomisation.
    - Place the stack on a different address on each process.
    - `malloc()` memory management randomisation.
    - Shuffle shared library loading order.
  - Guard pages: Insert empty frobidden pages within data segments.
  - ASCII armouring: Addresses with `00` can't be targeted with string overflows (binaries can).

Other forms of overflow attacks

- Return to system call, environment, heap or global
  - Response to non-executable stack defences
  - Overwrite return address with address of `system` or other libc function.
- Heap or global data overflow
  - May have function pointers can exploit.
  - Manipulate management data structures.
- Replacement stack frame
  - Used when have limited buffer overflow (Off-by-one)

Replacement stack frame



- This attack is difficult:
  - No nop sled. Dummy fp address guess must be perfect.
  - Local vars of caller become invalid. Process might crash.
- But possible!
- Use $\leq, <, \geq, >$ adequately.