Writing more secure code
or
How to avoid stack smashing

Vulnerability
specific flaw or oversight in a piece of SW that allows attackers to do something malicious

- expose or alter sensitive data
- disrupt or destroy a system
- take control of a computer system or program
our focus:
stack smashing

general idea:
a security vulnerability in a program allows an attacker to gain control of a program.

A buffer overflow bug permits this.
Here is the buggy program's stack and a piece of code.

```c
/* local variable */
char ar[16];
... 
strcpy(ar, src);
```

the buffer overflow: 
src has more than 16 bytes
In fact, the attacker knows the layout of the stack (this is easy, because it is well-known and always the same).

So, the attacker provides a `src` string that overflows the `dest` buffer.

This can:

1. change other local variables that are in the A.R.
(2) overwrite the rtn.addr with a more promising value

(2a) a system library function that does something useful for the attacker

(2b) Location that the attacker already controls. Contains code that does whatever the attacker wants. Now the shell code is running (under the guise of the innocent program).

(load) shell code

(local variable) array

prev%ebp

rtnaddr

new p1

new p2
Exploit code causes the program to do something unintended like

- write a file
- read (copy) private materials
- start another program
Ways to improve the situation:

1. Never use vulnerable library functions
   - `strcpy`, `strcat`, `sprintf` all keep going until src's `\0` is found .... so we cannot guarantee the dest buffer is big enough

2. ALWAYS check function return values
3 Implement preventative measures in the OS

3a Problem addressed: the stack looks the same for all programs.

```plaintext
%esp
same addr for all programs as they start
```

```plaintext
saved ebp
rtn addr
```

Attacker knows how much stack space is allocated AND its location, so exploit code has a spot...

```plaintext
goal: overwrite main's rtn addr to gain control
```
partial solution:
stack randomization

the OS allocates a random amount of space before invoking `main()`.

```
main's AR
{
    saved ebp
    rtnaddr
    ...
    ??
}
```

...... location not known & cannot be easily guessed
the concept of randomization can be expanded. All portions of program are randomly placed (in memory), so the attacker cannot acquire consistent knowledge of where "desirables" are.
3b. Have the compiler add extra code to a function to see if stack contents have changed.

Uses a canary value.

Just before return, code checks canary. If incorrect, buffer overflow or attacker tried halt program instead of using potentially bad return addr.

Make canary random/unique to invocation.
Make the HW/OS design include reduced locations of executable code.

In instruction fetch, HW checks PC against a per-"segment" bit that says executable or not.

Trap to OS code if PC has not executable address.
Take away lesson ...

all these exploits are provided by vulnerabilities in the code,

SO,

write better code!