Defense mechanisms

How do we stop this from happening?

Defenses

Different strategies:

- **Prevention**
- **Detection**
- **Reaction**

Ideally, you'd like to *prevent* problems, but typically you cannot, so you have to

- *make it harder* for the attacker
- *mitigate* the potential impact
- *detect* attacks and *react*

Defenses at different levels

- At the program level
  - to prevent attacks by removing the vulnerabilities
- At the compiler level
  - to detect and block exploit attempts
- At the operating system level
  - to make the exploitation much more difficult

First of all: the human Factor

- The main cause of buffer overflows are bad programmers, not the C language ;-)
  - educate programmers how to write secure code
  - test the programs with a focus on security issues

- Switch to more secure library functions
  - Standard Library: strncpy, strncat, ...
  - BDS's strlcpy, strlcat (boundary safe)
  - LibSafe: wrapper around a set of potentially “dangerous” libc functions
  - ContraPolice: libc extension to prevent heap overflow

Runtime checking: Libsafe

- Intercepts calls to dangerous functions that manipulate strings
  strcpy, strcat, getwd, gets, [vf]scanf, realpath, [v]sprintf
- Uses the frame pointer to approximate the buffer size:
  buffer size < |EBP – buff address|
- Adds runtime checks to make sure that any buffer overflows are contained within the current stack frame

Can you attack this function with a buffer overflow?

```
void f(....){
  long canary = CANARY_VALUE; // initialise canary
  ...
  ...
  if (canary != CANARY_VALUE) {
     exit(CANARY_DEAD); // abort with error code
  }
}
```

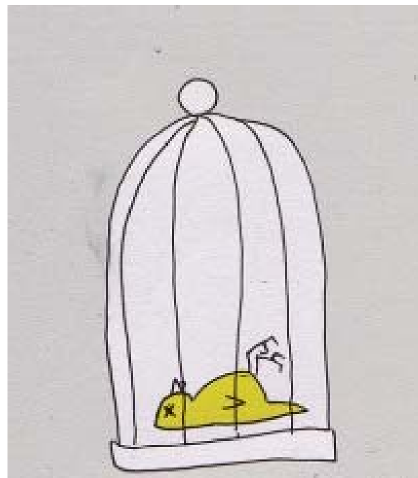
Stack protection with canaries

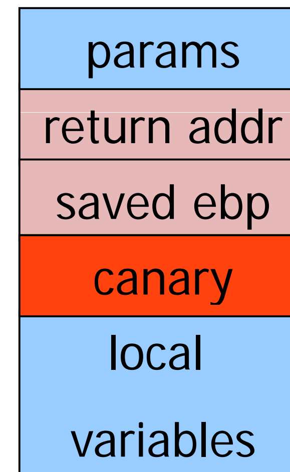
Goal:

protect the stack frame from being overwritten by the attacker

Idea: let the compiler insert code to

- add a "canary" value between the local variables and the saved EBP
- at the end of the function, check that the canary is “still alive”
- a different canary value means that a buffer preceding it in memory has been overflowed - ie. the stack has been smashed!





Stack canaries

You could add your own code to add & check stack canaries, by adding the lines below to the beginning and end of functions

```
void f(....){
   long canary = CANARY_VALUE; // initialise canary
   ...
   ...
   if (canary != CANARY_VALUE) {
       exit(CANARY_DEAD); // abort with error code
   }
}
```

It is easier & better to let the compiler add this code for you.
That can cover all the points where the function returns.
gcc does this with the `-fstack-protector` option

Stack canaries

You could add your own code to add & check stack canaries,
by adding the lines below to the beginning and end of functions

```
void f(....){
  long canary = CANARY_VALUE; // initialise canary
  ...
  ...
  if (canary != CANARY_VALUE) {
     exit(CANARY_DEAD); // abort with error code
  }
}
```

It is easier & better to let the compiler add this code for you.
That can cover all the points where the function returns.
gcc does this with the `-fstack-protector` option

Stack canaries

A clever attacker can try to put the correct canary value back.

Tricks to make this harder

- Generate a random canary value each time a program starts,
  so the attacker cannot predict the value it
- Make sure there is a null-terminator, ie. the character `\0',a somewhere in the middle of the canary value.

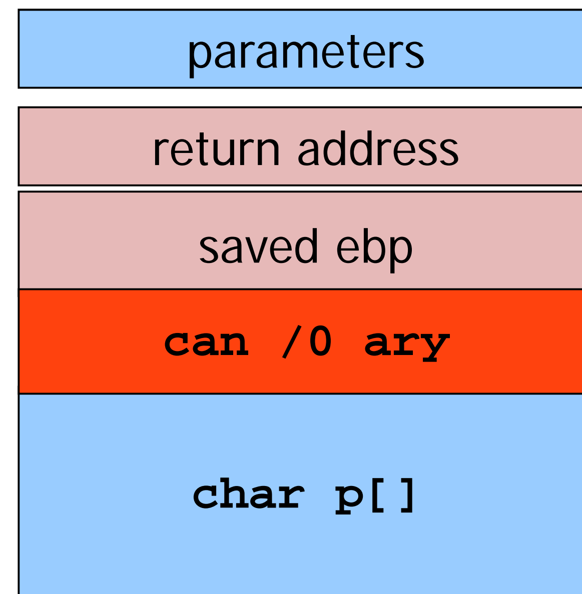
iSecLAB

Including the null terminator \0 in the canary

Can you over overflowing the array p with some string copying function, corrupting the return address but leaving the canary intact?

No,you'd need two operations:

1. one to change the return address, and put last part of canary, `ary`, back. This will remove the `/0`.
2. a second one to write the first part of the canary, `can /0`, back



Question: how many overflows would be needed if there are 4 null terminators in the canary?

Stack canaries

A clever attacker could try to put the correct canary value back.

Tricks to make this harder

- Generate a random canary value each time a program starts,

  so the attacker cannot predict the value it
- Make sure there is a null-terminator, ie. the character `\0’,a somewhere in the middle of the canary value.

  This makes it impossible for the attacker to write the canary value back using a standard string writing functions, as these functions will stop at null-terminators.
- Let the canary be the XOR of some “master” canary value and the return address on the stack.

  If the attacker then changes the return address, then the old canary value is no longer correct.

OS Level: *N*on e*X*ecutable Stack (NX aka W⊕X)

- Does not block buffer overflows, but prevent the shellcode from being executed
  - can affect the execution of some programs that normally require to execute data on the stack
  - it makes use of hardware features such as the NX bit (IA-64, AMD64)

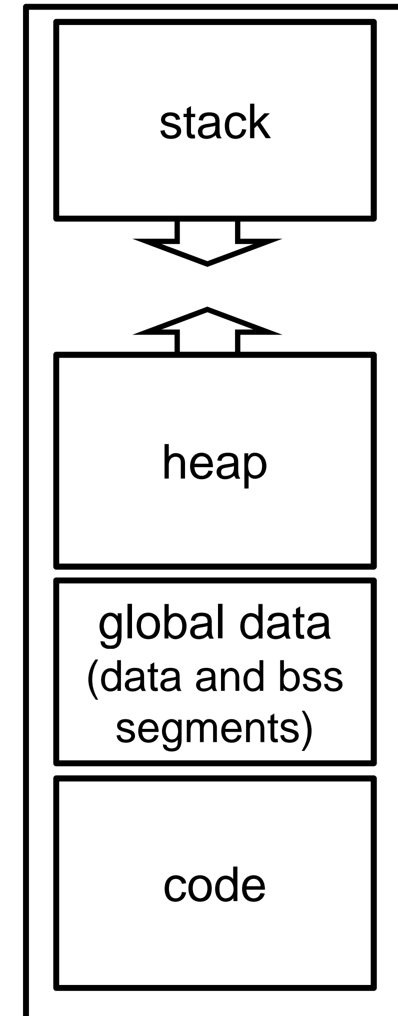
- Supported by many operating systems
  - MacOs X
  - Data Execution Prevention (DEP) on Windows
  - OpenBSD W^X
  - ExecShield and PAX patches in Linux

non-executable memory

Memory used by a process (program in execution) consists of different segments

The program counter should point to the code segment, not the heap, stack, or data segments.

Making these segments non-executable makes it impossible for an attacker to execute malicious code that they manage to get into some stack- or heap allocated buffer



OS Level: Address Space Randomization

- Introduce artificial diversity by randomly arranging the positions of key data areas:

  base of the executable, position of libraries, heap, and stack)

  This prevents the attacker from being able to easily predict target addresses

Program Level: Static Analysis

Tools that analyse code at compile-time to spot potential buffer overflows.

- Ccured
- Flawfinder
- Insure++
- CodeWizard
- Cigital ITS4
- Cqual
- Microsoft PREfast/PREfix
- Pscan
- RATS
- Fortify

Doesn’t this solve the problem?

Stack canaries and non-executable stack make attacks harder,
but not impossible.

For example

- attacker may be able restore the canary values
- attacker may be able to jump to existing code, eg in return to libc attacks, defeating the non-executable stack protection
- other interesting targets for the attacker might not be protected by these measures, eg
  - data on the stack in the current frame
  - function pointers allocated on the stack or the heap

    (We will not go into function pointers, or expect you to know this for the exam)

Windows 2003 Stack Protection

The subtle ways in which things can still go wrong...

Microsoft introduced stack canaries in 2003 in its compilers

- Enabled with /GS command line option
- When canary is corrupted, control is transferred to an exception handler
- Exception handler information is stored ... on the stack
  - http://www.securityfocus.com/bid/8522/info
- Countermeasure: register exception handlers, and don't trust exception handlers that are not registered or that are on the stack
- Attackers may still abuse existing handlers or point to exception handler outside the loaded module...