ECS 153 Discussion Section

April 6, 2015

What We’ll Cover

Goal:

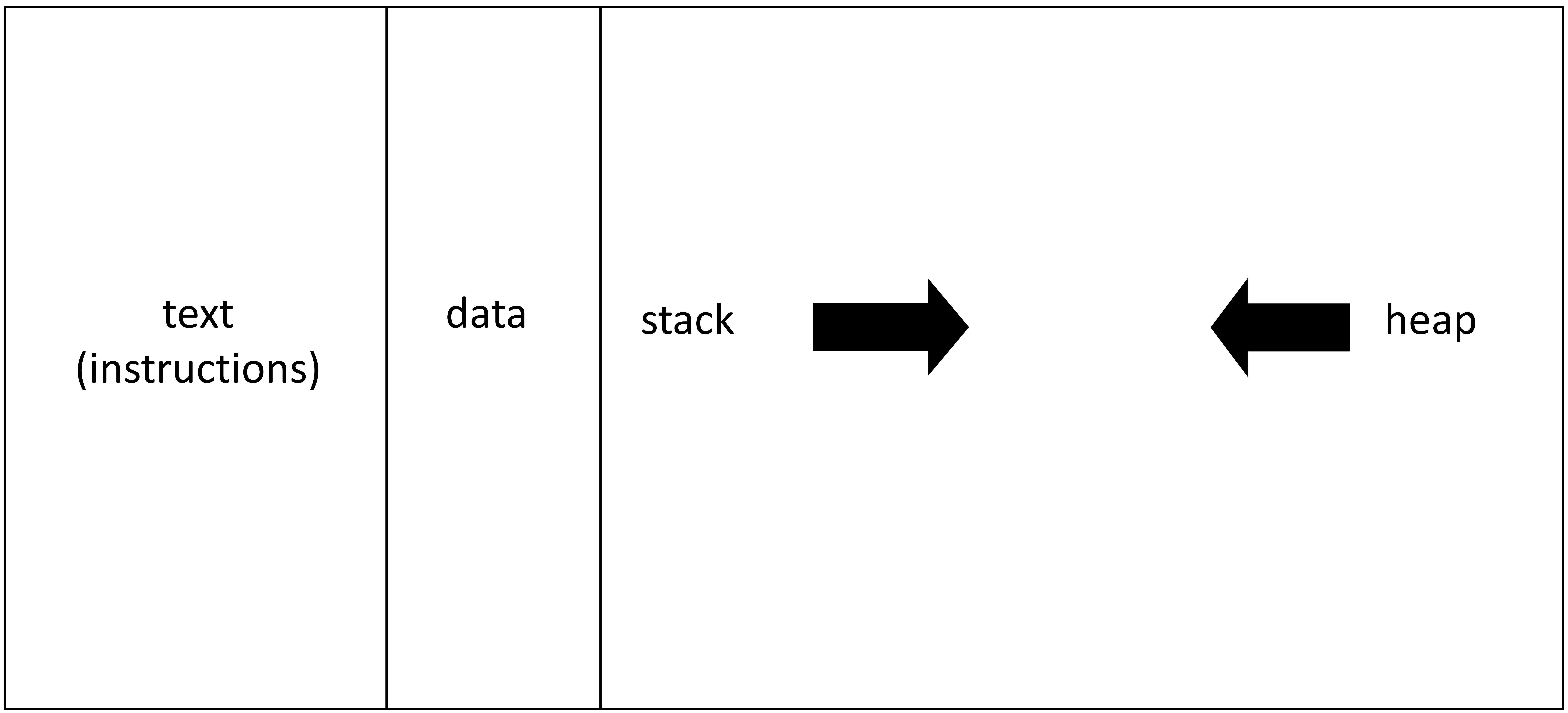
- To discuss buffer overflows in detail

- Stack-based buffer overflows
  - “Smashing the stack”: execution from the stack
  - ARC (or return-to-libc) attacks; ROP
- Data-based buffer overflows
- Ways to avoid and detect them

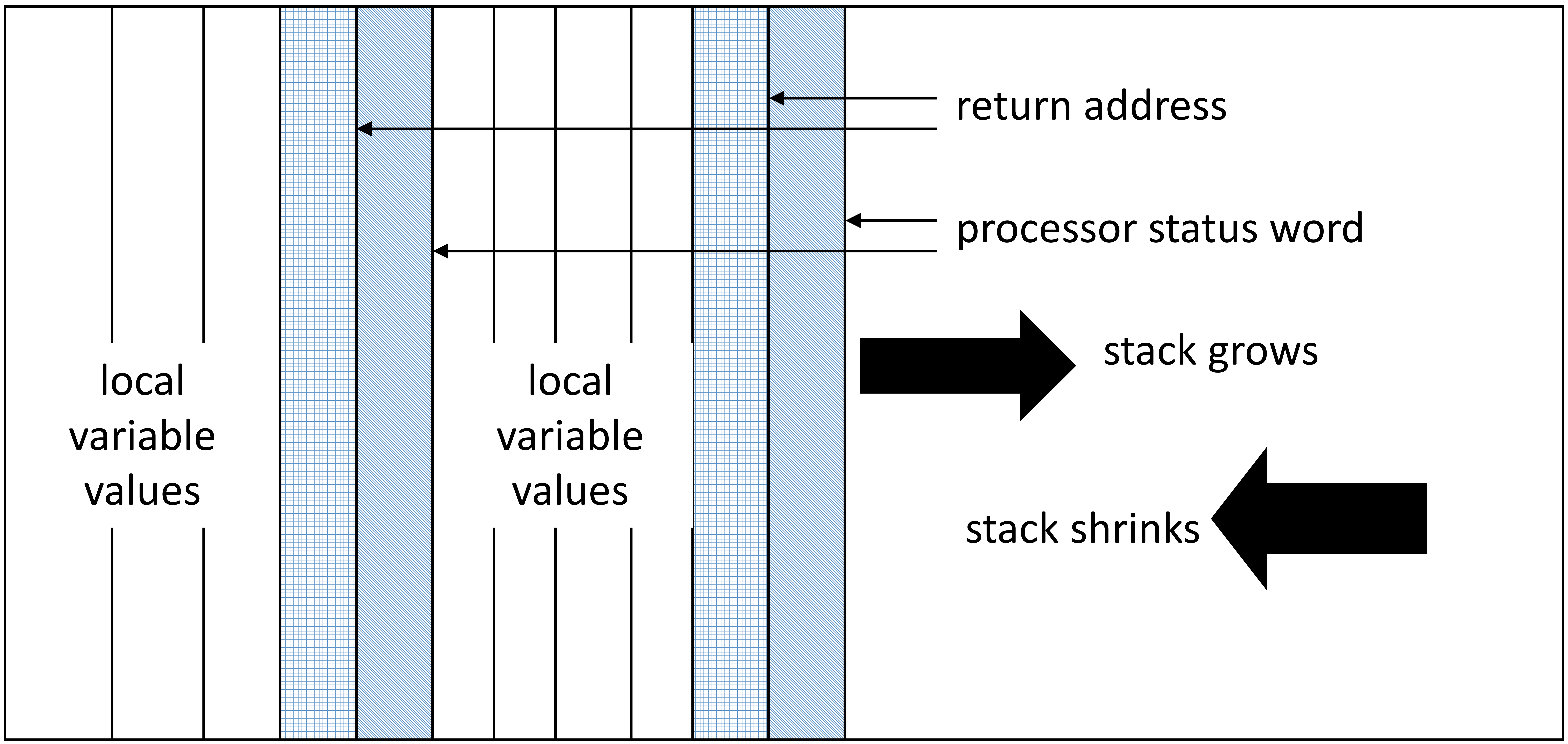
What They Are

- Traditionally considered as a technique to have your code executed by a running program
- Other, less examined uses:
  – Overflow data area to alter variable values
  – Overflow heap to alter variable values or return addresses
  – Execute code contained in environment variables (not fundamentally different, but usually stored on stack)

Process Memory Structure



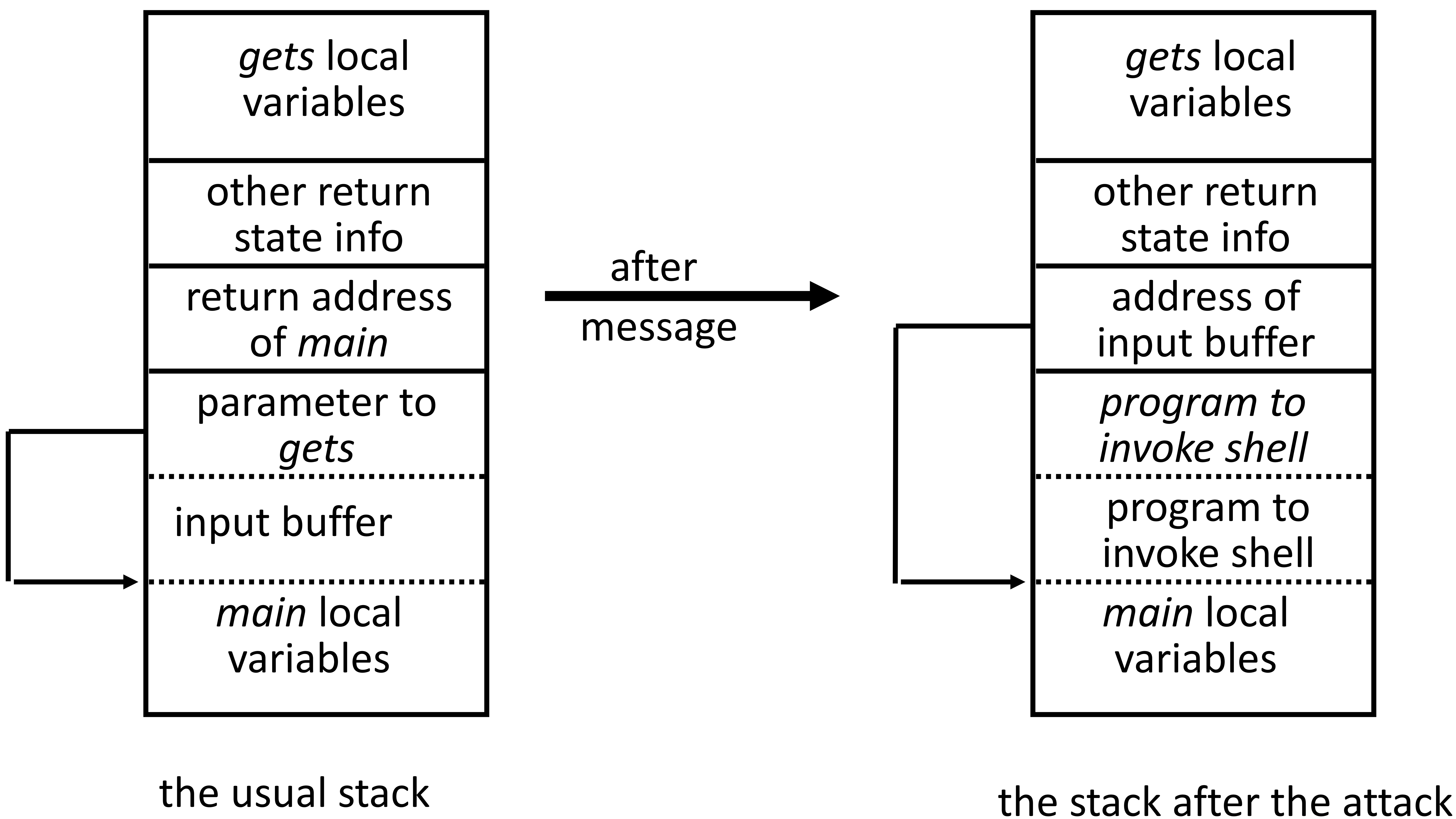
Typical Stack Structure



Idea

- Figure out what buffers are stored on the stack
- Write a small machine-language program to do what you want (exec a shell, for example)
- Add enough bytes to pad out the buffer to reach the return address
- Alter return address so it returns to the beginning of the buffer
  - Thereby executing your code …

In Pictures



gets local variables
other return state info
return address of *main*
parameter to *gets*
input buffer
main local variables
after
message
gets local variables
other return state info
address of input buffer
program to invoke shell
program to invoke shell
main local variables
the usual stack
the stack after the attack

In Words

- Parameter to *gets*(3) is a pointer to a buffer
  - Here, buffer is 256 bytes long
- Buffer is local to caller, hence on the stack
- Input your shell executing program
  - Must be in machine language of the target processor
  - 45 bytes on a Linux/i386 PC box
  - Pad it with 256 – 45 + 4 = 215 bytes
  - Add 4 bytes containing address of buffer
    - These alter the return address on the stack

Required

- Change return address
  - Best: you know how many bytes the return address is from the buffer
  - Approach: pad shell code routine with address of beginning of buffer
    - If not sure, put NOPs before shell code, and guess
    - Use buffer address as padding
      - Need to get alignment right, though

Also Required

- Machine language program to spawn subshell (or whatever) that does not contain either a newline or a NUL (string terminator)
  - If string loaded by standard I/O function (like *gets*(3)), no NULs or newlines allowed
  - If string loaded by string function (like *strcpy*(3)), no NULs allowed
    - *strncpy* terminates on NUL as well as length …
  - Many other problems (e.g., buffer may be massaged by *tolower*(), so can't contain upper case)

Quick Test

- If you overflow the return address with some fixed character, you are likely to load that location with an illegal address
- So, enter fixed data as input (or as arguments)
  - Usual value is sequence of ‘A’ (0x41)
- If the program crashes, you probably have a stack overflow
  - Go look at the stored address in the program counter; if it’s 0x41414141, you have an overflow

Where to Put Shell Code

- In the buffer
  - Get address by running *gdb*, *trace* or their ilk
    - Need access to system of same type as attacked system
- Somewhere else: environment list
  - Stored in standard place for all processes
  - Put shell code in last environment variable
    - Create new one
  - Calculate and supply this address

Variations on a Theme

- Return-to-libc (arc injection) attack
  - Change the return address to point to a library, or other function
    - On return you will jump to that routine
  - Set up the stack so when you jump to that function, it looks like a proper function call
    - Then the function executes … under *your* control

Return-Oriented Programmng (ROP)

- Like return-to-libc, but jump to code sequences rather than function entry points
  - Sequence terminates with a return
  - Enough of these sequences in standard C library to form a simple programming language
- Attack: chain these together using the stack to handle their invocation
  - Sounds complex, but can be made simple

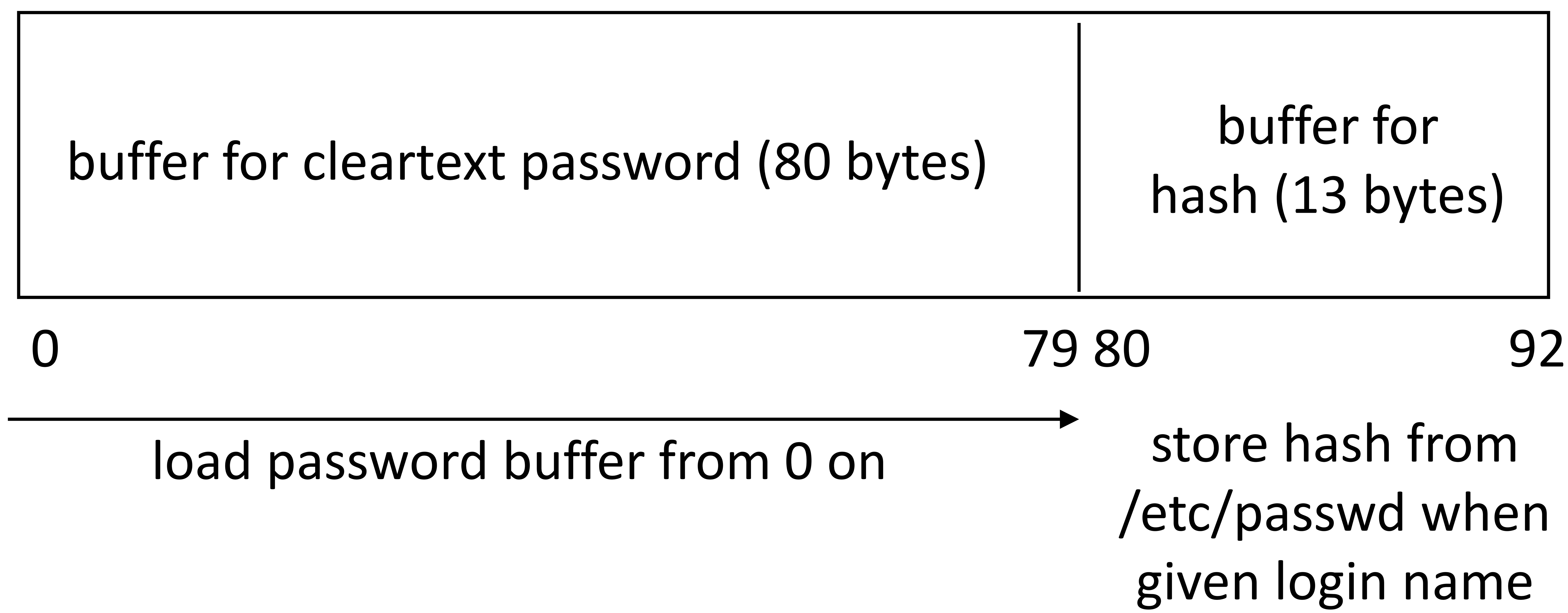
Data Segment Buffer Overflows

- Can’t change return address
  - Systems prevent crossing data, stack boundary
    - Even if they didn’t, you would need to enter a pretty long string to cross from data to stack segment!
- Change values of other critical parameters
  - Variables stored in data area control execution, file access
- Can change binary or string data using technique similar to that of stack buffer overflowing

Example: *login* Problem

- Program stored user-typed password, hash from password file in two adjacent arrays
- Algorithm
  - Obtain user name, load corresponding hash into array
  - Read user password into array, hash, compare to contents of hash array
- Attack
  - Generate any 8 character password, corresponding hash
  - When asked for password, enter it, type 72 characters, then type corresponding hash

In Pictures


buffer for cleartext password (80 bytes)
buffer for
hash (13 bytes)
0
79 80
92
load password buffer from 0 on
store hash from
/etc/passwd when
given login name

Requires

- Knowing what data structures are, and where
  – Need positions with respect to one another
  – If symbol table present, use *nm*(1)
- Knowing what data structures are used for
  – Use the source
  – Guess
  – Disassemble the code
- Knowing what a “good” value is
  – Good for the attacker and bad for the system

Example: The *syslogd* Bug

- On one system, *syslogd* read message from a socket
  - Does not use *gets*(), so no overflow at the read
- Message formatted with PID, date, *etc*.
  - Used *sprintf*(3) to write into an array called *line2* allocated with 2048 characters
- This array for *sprintf* can overflow

The Problem

- In the 3 cases we've seen, string put into array without being checked for overflow
  - fingerd
    - If *buf* not overflowed, stack uncorrupted and return made to *main*
  - login
    - If *passwd* not overflowed, hash not altered and correct hash used to validate password
  - syslogd
    - If *line2* not overflowed, stack uncorrupted and return made to caller

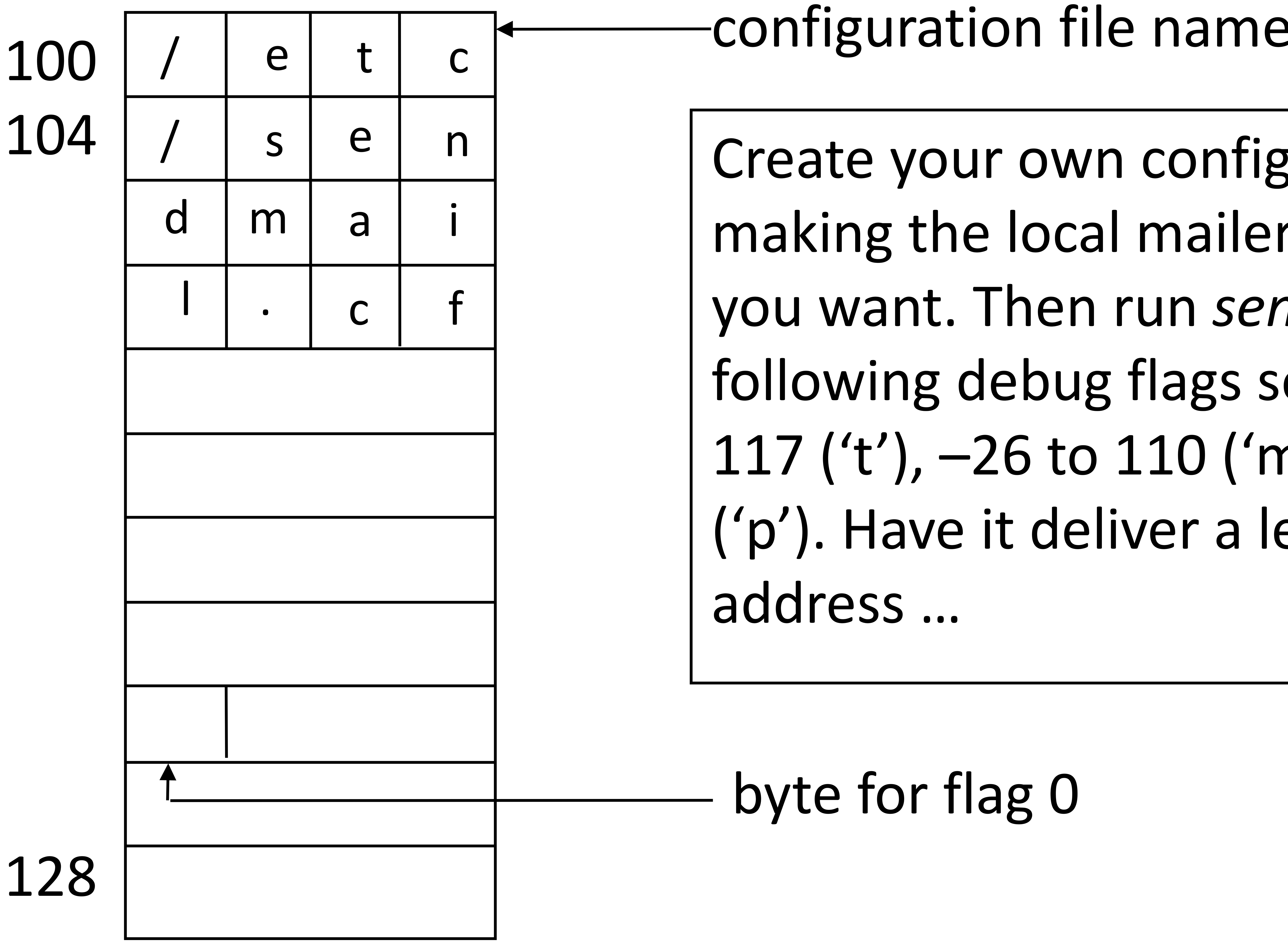
Selective Buffer Overflow

- Sets particular locations rather than just overwriting everything
- Principles are the same, but you have to determine the specific locations involved
- Cannot approximate, as you could for general stack overflow; need exact address
  - Advantage: it's fixed across all invocations of the program, whereas a stack address can change depending on memory layout, input, or other actions

Sendmail Configuration File

- *sendmail* takes debugging flags of form *flag.value*
  - sendmail -d7,102 sets debugging flag 7 to value 102
- Flags stored in array in data segment
- Name of default configuration file also stored in array in data segment
  - It's called *sendmail.cf*
- Config file contains name of local delivery agent
  - `Mlocal` line; usually /bin/mail …

In Pictures



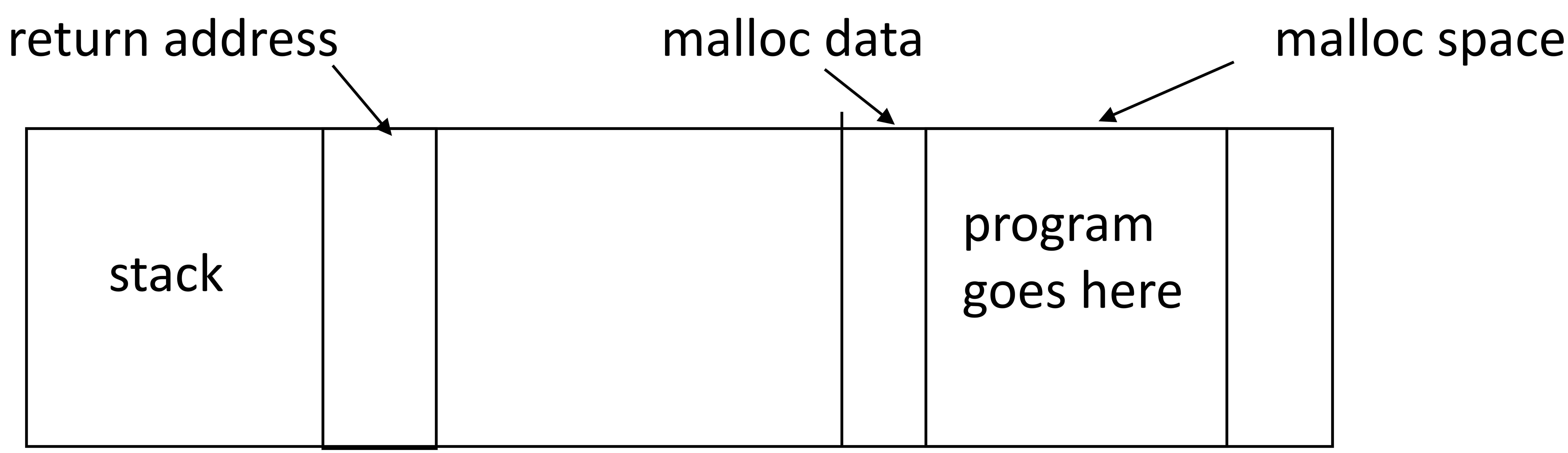
Create your own config file, making the local mailer be whatever you want. Then run *sendmail* with the following debug flags settings: flag –27 to 117 (‘t’), –26 to 110 (‘m’), and –25 to 113 (‘p’). Have it deliver a letter to any local address …

Problems and Solutions

- *Sendmail* won't recognize negative flag numbers
- So make them unsigned (positive)!
  - –27 becomes 2^{32} – 27 = 4294967269
  - –26 becomes 2^{32} – 26 = 4294967270
  - –25 becomes 2^{32} – 26 = 4294967271
- Command is:
  - sendmail -d4294967269,117 -d4294967270,110 \
    -d4294967271,113 …

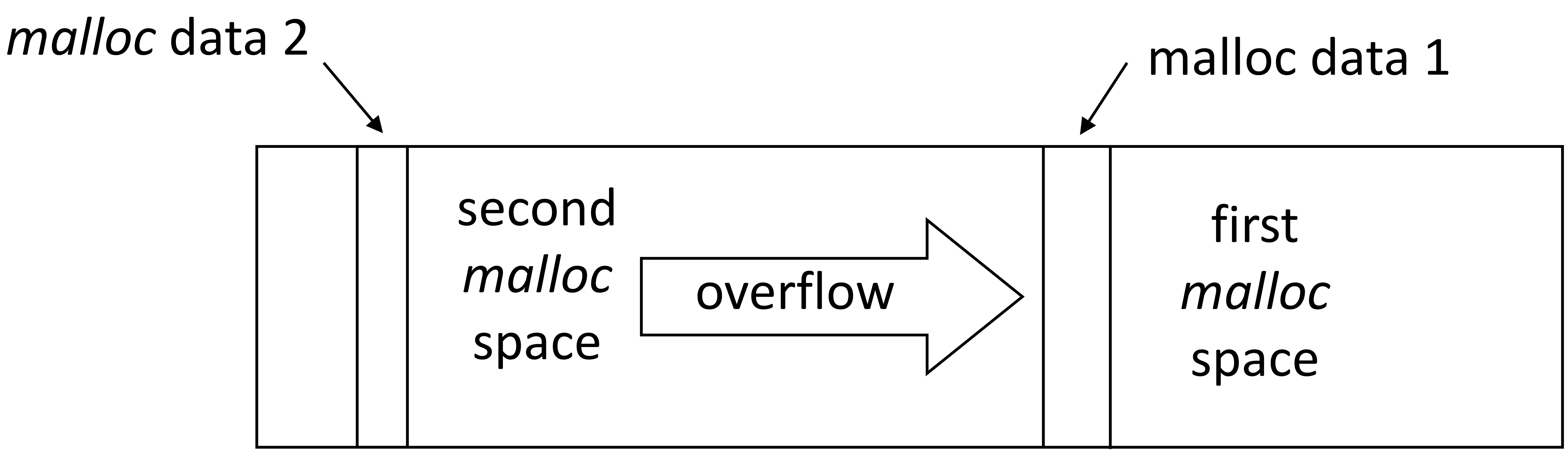
Attack: Whacking the Heap

- Like stack, except you find something on the heap that you can alter
  - Vendors protect stack from execution, but rarely the heap



Attack: Changing the Heap

- Like data segment, except overwrite other components on the heap
  - Mucks up storage allocators unless you figure out what the *malloc* information is



Things To Alter

- Function pointers
  - Look for places where these are stored on stack or heap
  - May be explicit (store function pointer in dynamically allocated array) or implicit (*atexit*(3))
- Fault handlers
  - Some are stored at the beginning of the heap, so just keep writing

Requires

- Knowing what allocations are performed, and where the allocators place the storage
  – Need positions with respect to one another
- Knowing where program stores function pointers
- Knowing where system stores function pointers
  – See *atexit*(3)
- Knowing what a "good" value is

Same importance as for stack-based buffer overflows

General Rule

- Assume input may overflow an input buffer
  - Design to prevent overflow
- In general, don't trust input to be of the right form or length

Design Tip: Buffer Overflow

- Assume input may overflow an input buffer
  - Design to prevent overflow
- In general, don't trust input to be of the right form or length

Handling Arrays

- Use a function that respects buffer bounds
  - Avoid *gets*, *strcpy*, *strcat*, *sprintf*
  - Use *fgets*, *strncpy*, *strncat*, instead; no standard replacement for *sprintf* (*snprintf* on some systems)
  - Don't forget to add a NUL byte at the end of arrays if you use these functions, and watch those *n* values!
- To find good (bad) functions, look for those which handle arrays and do not check length
  - Checking for termination character is not enough
- Check array references
  - Not only when they are in loops

Puzzle

Consider the *strn* functions.

- What happens when *n* is negative?

  As the functions' *n* is an unsigned parameter, *n* is effectively infinite

- In *strncpy*, what happens if the first two arguments overlap?

  The behavior is undefined—so it varies from one system to another

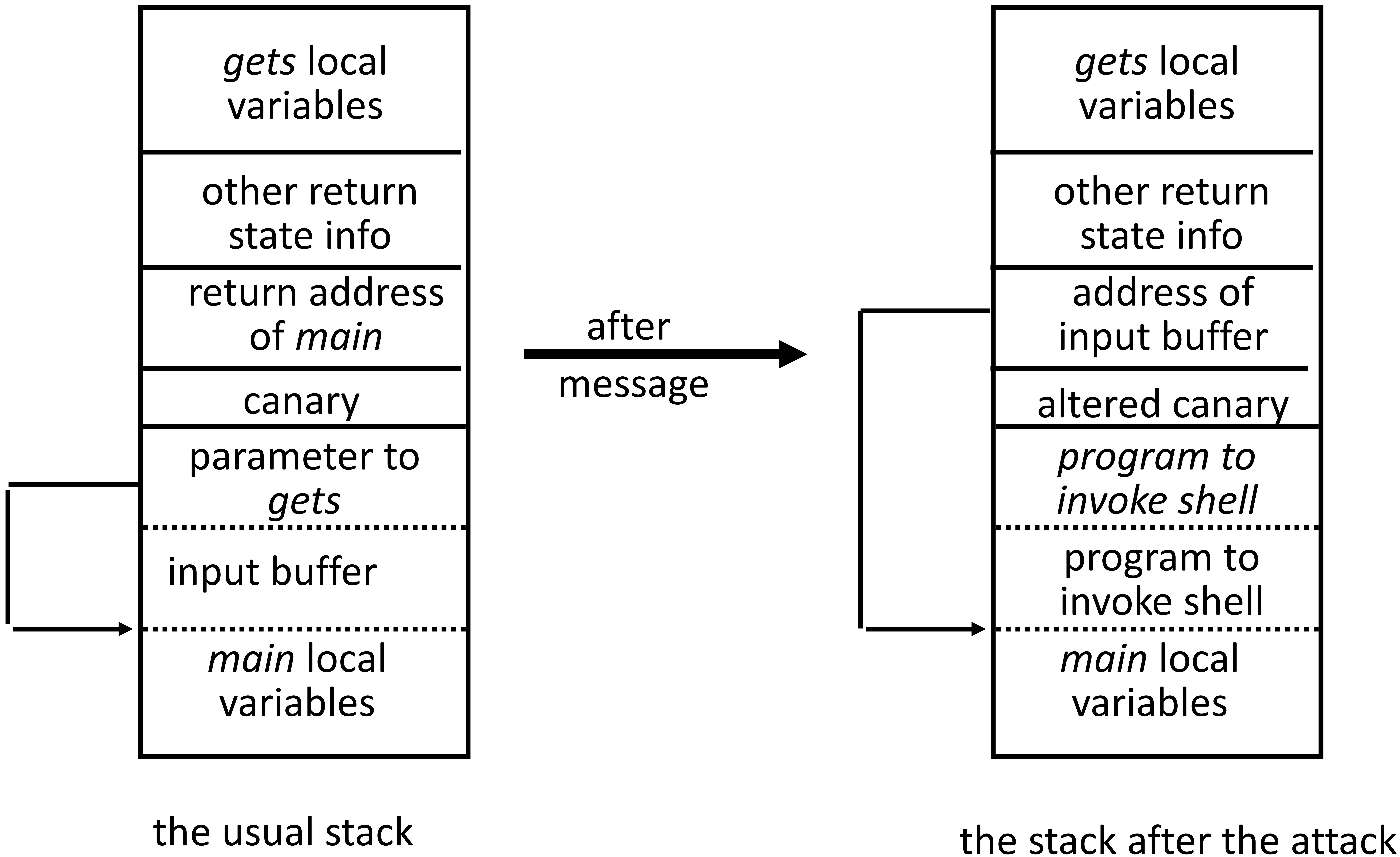
Common Error

- When writing error handlers, be sure you check for buffer overflows during formatting of error messages, even if the program provides the message
  - Sometimes you can manipulate the environment to force a bogus message
  - Source of message (file names printed, command-line arguments, *etc*.) are often under user's control

One Way to Fix Them

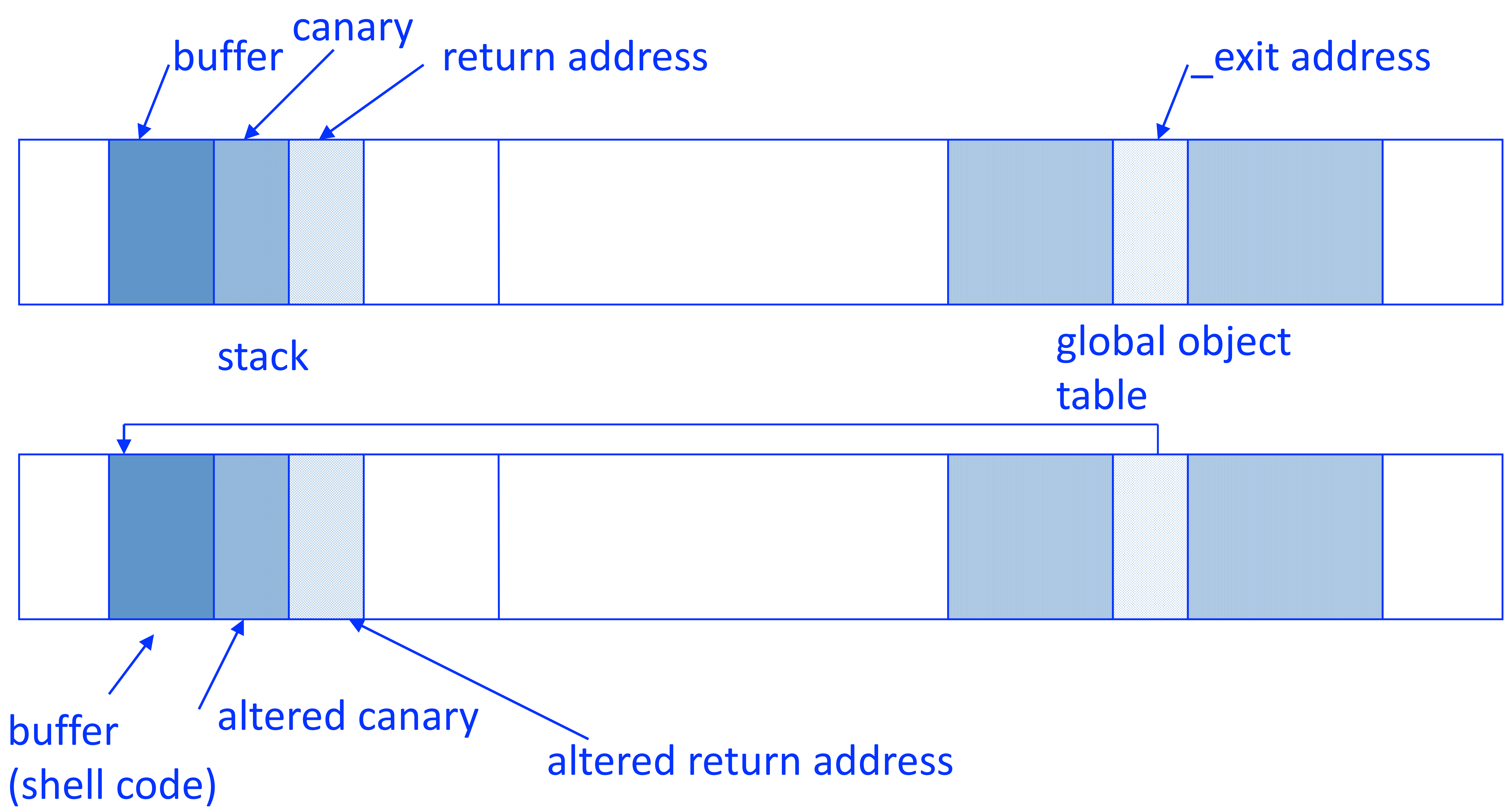
- Canaries
  - Install a special value (the "canary") on the stack before the return address
  - At exit of routine (but *before* you actually do the return), compare the canary to the special value
  - If they differ, call an error handling routine

Canaries


gets local variables
other return state info
return address of *main*
canary
parameter to *gets*
input buffer
main local variables
after
message
gets local variables
other return state info
address of input buffer
altered canary
program to invoke shell
program to invoke shell
main local variables
the usual stack
the stack after the attack

Puzzle

Will this always work?



Key Ideas

- Whenever you put data into an array (like a string), check that the length will not overflow that array
  - Look out for functions that manipulate arrays (strings) and do not check lengths
  - Look for other problems, such as calling a string manipulation function with bad numeric values
- When you compile, if you can turn off execute permission on the stack, do so
  - This will stop one type of buffer overflow attack
  - It does *not* stop many others, though