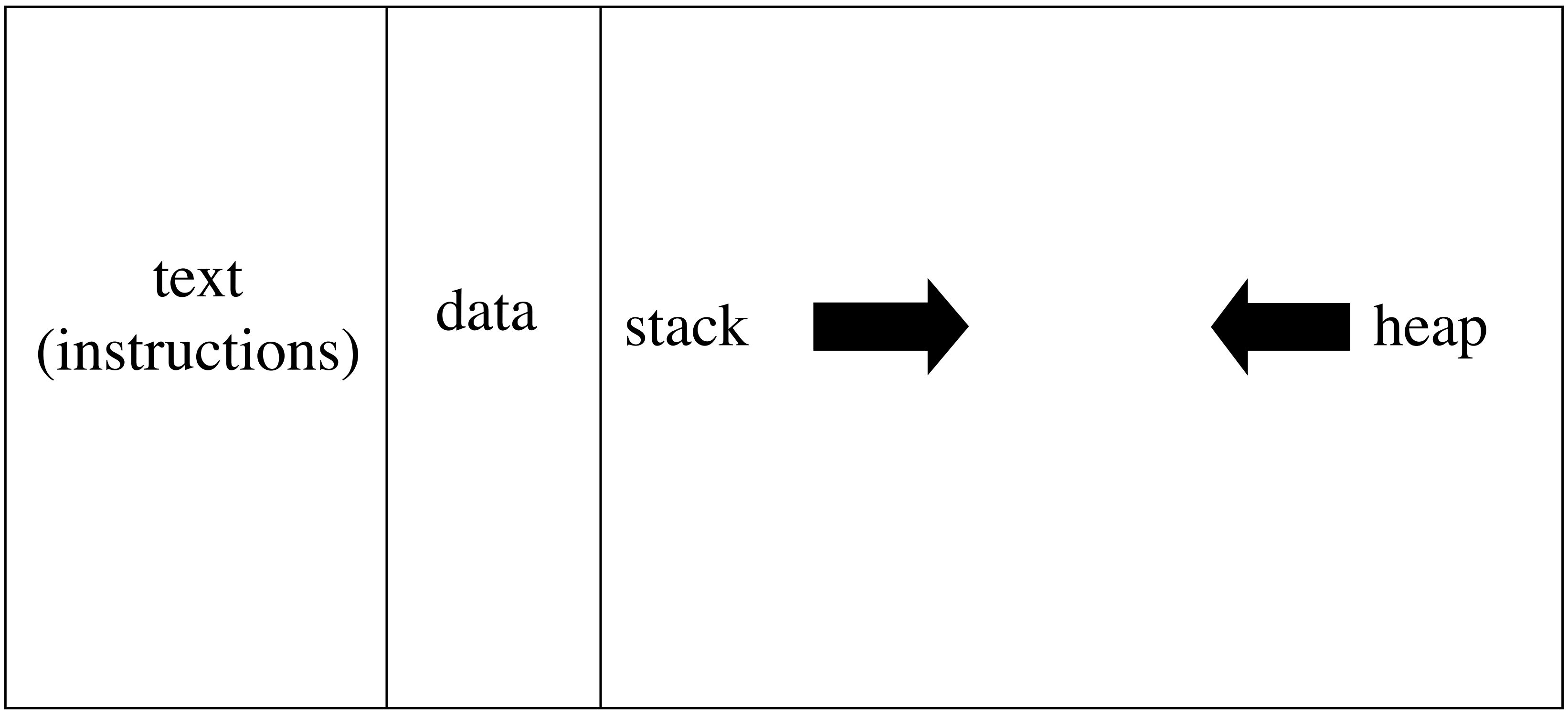
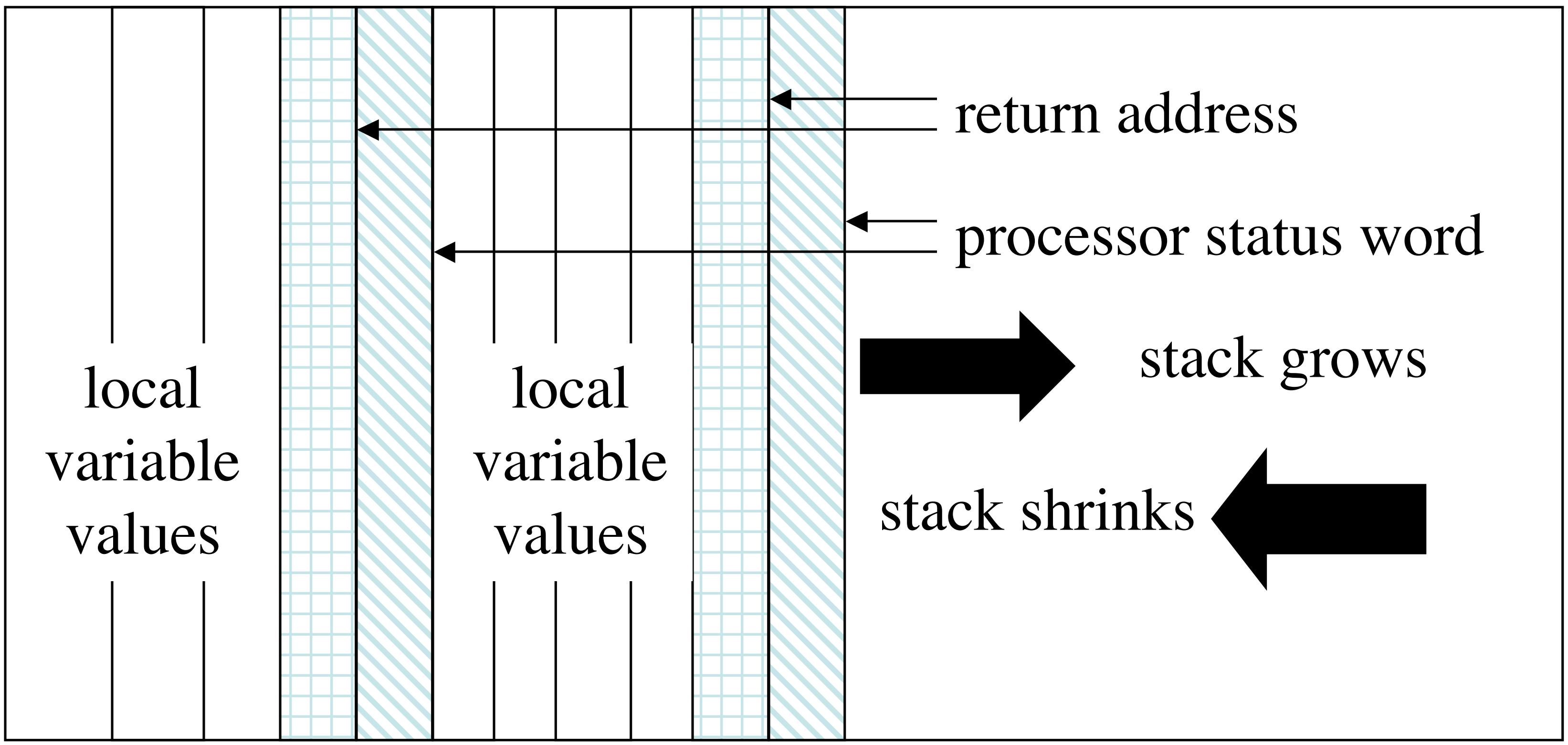
Buffer Overflows

- 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



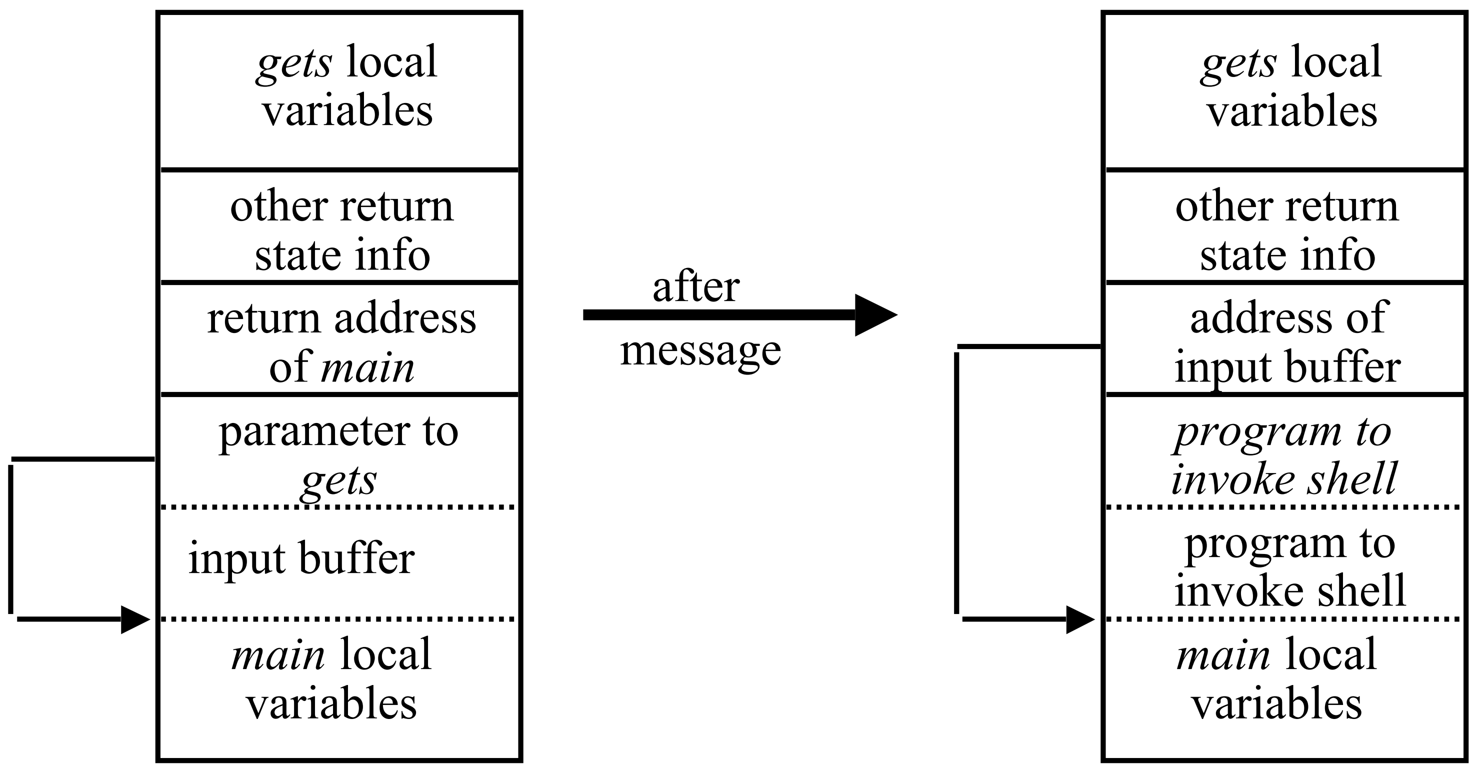
Typical Stack Structure



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



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

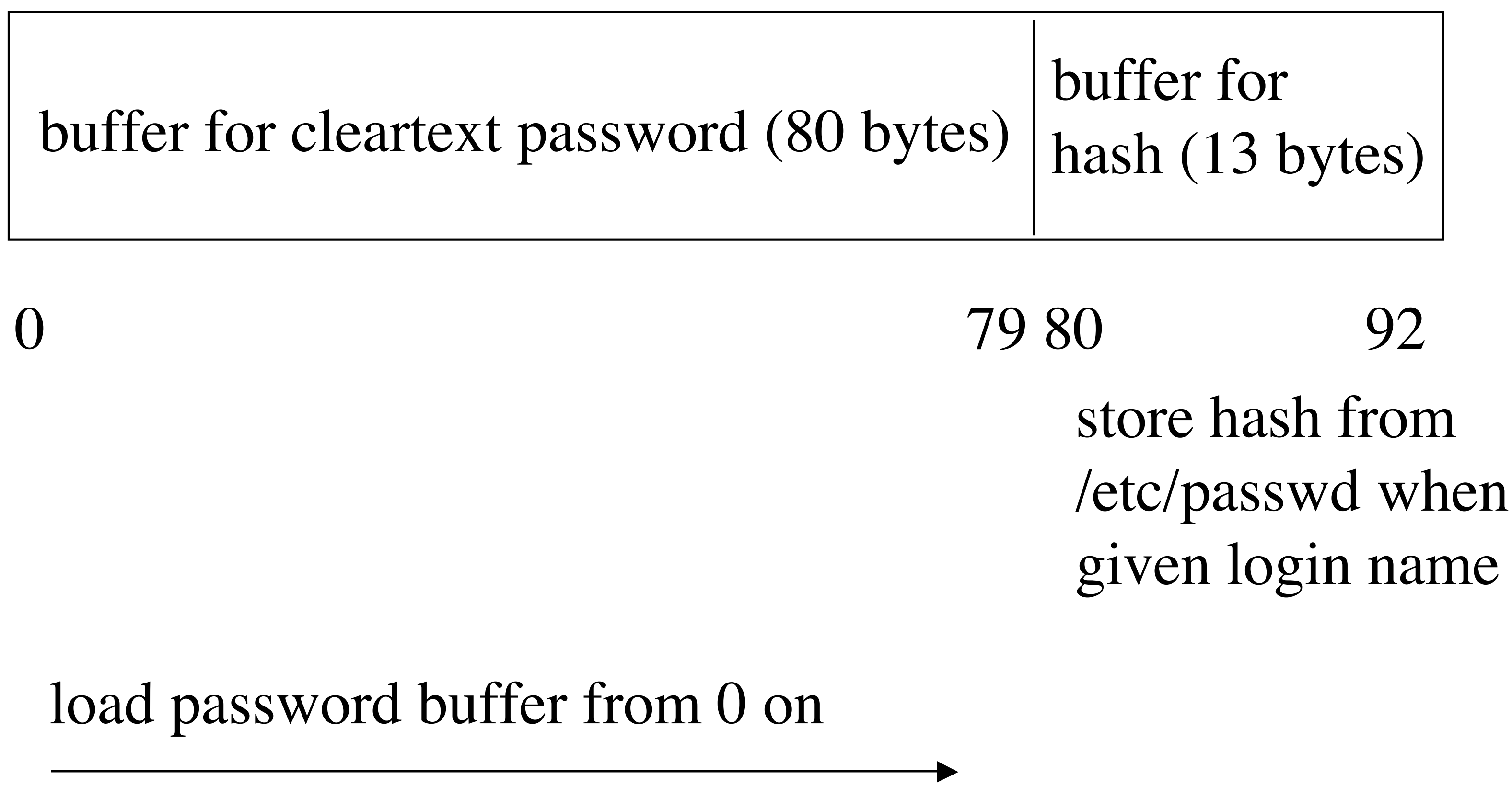
Data Segment Buffer Overflows

- 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
store hash from
/etc/passwd when
given login name
load password buffer from 0 on

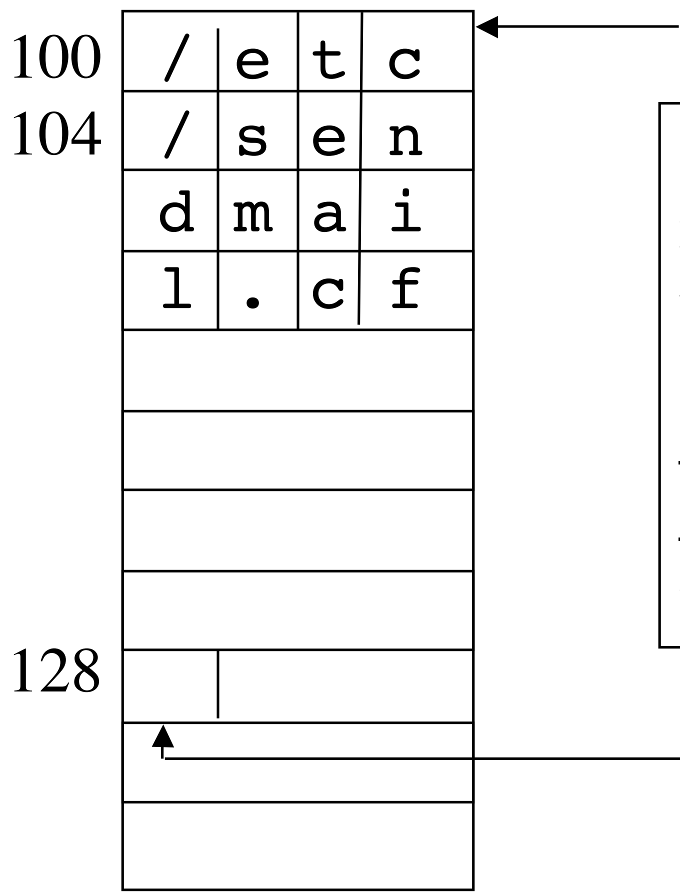
Selective Buffer Overflow

- 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 232 – 27 = 4294967269
  - –26 becomes 232 – 26 = 4294967270
  - –25 becomes 232 – 26 = 4294967271
- Command is:

```
sendmail -d4294967269,117 -d4294967270,110 \
   -d4294967271,113 …
```

Numeric Overflows

- Program may assume a particular value stays in a bound
  - May depend on assumptions about operating system or other interfaces
- Look for ways to overflow or underflow them
  - Proper programs will check for errors
  - Common error: ignore overflow ($> 2^{32}-1$)
  - Type punning helpful (especially signed and unsigned integers)

Attack: NFS UIDs

- UNIX UIDs are 16 bits on many systems
- NFS uses a 32-bit UID
  – Done specifically for portability
- NFS server invokes UNIX kernel with UID of remote user
  – Kernel does access control checking
- NFS disallows UID 0
  – Mapped into 65534 (or –2), the user *nobody*, before kernel invoked
  – You can override this in a configuration file, but administrators rarely do (and should not, in general)

Obvious Question

- What happens at the NFS server if NFS client user's UID is 2^{17}?
  - Can't give this directly to UNIX kernel, as the latter takes only UIDs of $2^{16}-1$ or less
- Hypothesis: UID is truncated to 16 bits by NFS server
  - Assumes maximum UID for server system is $2^{16}-1$
  - Give it to NFS and see …
- Idea: check all programs that take UIDs as integers

Results of the Attack

- NFS client sends request, UID to NFS server
- NFS server takes UID, checks that it is not 0
  - As $2^{17} \neq 0$, UID is not remapped
- NFS gives UID to UNIX kernel for access control
- UNIX kernel discards high-order bits …
  - As $2^{17} =$ 0000 0000 0000 0001 <u>0000 0000 0000 0000</u>, the UID that the kernel sees is 0
  - Presto! *root* access to files

strn Functions

- What happens when *n* is negative?
  - Proper behavior: nothing, or error message
  - Usual behavior: goes until NUL encountered (effectively the same as *strcpy* and *strcat*, *etc*.)
- Suppose first, second arguments overlap?
  - Manual says they “must not overlap”
  - Behavior varies from system to system