

# Memory safety attacks

- Buffer overflows
  - Can be used to read/write data on stack or heap
  - Can be used to inject code (ultimately root shell)
- Format string errors
  - Can be used to read/write stack data
- Integer overflow errors
  - Can be used to change the control flow of a program
- TOCTOU problem
  - Can be used to raise privileges

# What's wrong with this code?

**Suppose that it has higher privilege than the user**

```
int main() {
    char buf[1024];
    ...
    if(access(argv[1], R_OK) != 0) {
        printf("cannot access file\n");
        exit(-1);
    }

    file = open(argv[1], O_RDONLY);
    read(file, buf, 1023);
    close(file);
    printf("%s\n", buf);
    return 0;
}
```

# What's wrong with this code?

**Suppose that it has higher privilege than the user**

```
int main() {
    char buf[1024];
    ...
    if(access(argv[1], R_OK) != 0) {
        printf("cannot access file\n");
        exit(-1);
    }

    file = open(argv[1], O_RDONLY);
    read(file, buf, 1023);
    close(file);
    printf("%s\n", buf);
    return 0;
}
```

uid

euid

# What's wrong with this code?

**Suppose that it has higher privilege than the user**

```
int main() {  
    char buf[1024];  
    ...  
    if(access(argv[1], R_OK) != 0) {  
        printf("cannot access file\n");  
        exit(-1);  
    }  
  
    file = open(argv[1], O_RDONLY);  
    read(file, buf, 1023);  
    close(file);  
    printf("%s\n", buf);  
    return 0;  
}
```

uid

euid

**~attacker/mystuff.txt**

# What's wrong with this code?

**Suppose that it has higher privilege than the user**

**uid**

```
int main() {
    char buf[1024];
    ...
    if(access(argv[1], R_OK) != 0) {
        printf("cannot access file\n");
        exit(-1);
    }
}
```

**~attacker/mystuff.txt**

**euid**

```
file = open(argv[1], O_RDONLY);
read(file, buf, 1023);
close(file);
printf("%s\n", buf);
return 0;
}
```

# What's wrong with this code?

**Suppose that it has higher privilege than the user**

uid

```
int main() {
    char buf[1024];
    ...
    if(access(argv[1], R_OK) != 0) {
        printf("cannot access file\n");
        exit(-1);
    }
}
```

**ln -s /usr/sensitive ~attacker/mystuff.txt**

euid

```
file = open(argv[1], O_RDONLY);
read(file, buf, 1023);
close(file);
printf("%s\n", buf);
return 0;
}
```

# What's wrong with this code?

**Suppose that it has higher privilege than the user**

**uid**

```
int main() {
    char buf[1024];
    ...
    if(access(argv[1], R_OK) != 0) {
        printf("cannot access file\n");
        exit(-1);
    }
}
```

**ln -s /usr/sensitive ~attacker/mystuff.txt**

**euid**

```
file = open(argv[1], O_RDONLY);
read(file, buf, 1023);
close(file);
printf("%s\n", buf);
return 0;
}
```

**“Time of Check/Time of Use” Problem (TOCTOU)**

# Avoiding TOCTOU

```
int main() {  
    char buf[1024];  
    ...  
    if(access(argv[1], R_OK) != 0) {  
        printf("cannot access file\n");  
        exit(-1);  
    }  
  
    file = open(argv[1], O_RDONLY);  
    read(file, buf, 1023);  
    close(file);  
  
    printf(buf);  
}
```

uid

euid



# Avoiding TOCTOU

```
int main() {  
    char buf[1024];  
    ...  
    if(access(argv[1], R_OK) != 0) {  
        printf("cannot access file\n");  
        exit(-1);  
    }  
  
    file = open(argv[1], O_RDONLY);  
    read(file, buf, 1023);  
    close(file);  
  
    printf(buf);  
}
```

uid

euid

# Avoiding TOCTOU

```
int main() {  
    char buf[1024];  
    ...  
    if(access(argv[1], R_OK) != 0) {  
        printf("cannot access file\n");  
        exit(-1);  
    }  
    uid = geteuid();  
    uid = getuid();  
    seteuid(uid);    // Drop privileges  
    file = open(argv[1], O_RDONLY);  
    read(file, buf, 1023);  
    close(file);  
  
    printf(buf);  
}
```

uid

euid

# Avoiding TOCTOU

```
int main() {  
    char buf[1024];  
    ...  
    if(access(argv[1], R_OK) != 0) {  
        printf("cannot access file\n");  
        exit(-1);  
    }  
    uid = geteuid();  
    uid = getuid();  
    seteuid(uid);    // Drop privileges  
    file = open(argv[1], O_RDONLY);  
    read(file, buf, 1023);  
    close(file);  
    seteuid(uid);    // Restore privileges  
    printf(buf);  
}
```

uid

euid

# Defensive coding for Memory Safety

# *Defensive* coding practices

- Think defensive driving
  - Avoid depending on anyone else around you
  - If someone does something unexpected, you won't crash (or worse)
  - It's about *minimizing trust*
- Each module takes responsibility for checking the validity of all inputs sent to it
  - Even if you “know” your callers will never send a NULL pointer...
  - ...Better to throw an exception (or even exit) than run malicious code

# How to program defensively

- Code reviews, real or imagined
  - Organize your code so it is obviously correct
  - Re-write until it would be self-evident to a reviewer

*“Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”*

- Remove the opportunity for programmer mistakes with better languages and libraries
  - Java performs automatic bounds checking
  - C++ provides a safe `std::string` class

# Secure coding practices

```
char digit_to_char(int i) {  
    char convert[] = "0123456789";  
    return convert[i];  
}
```

**Think about all potential inputs, no matter how peculiar**

# Secure coding practices

```
char digit_to_char(int i) {  
    char convert[] = "0123456789";  
    return convert[i];  
}
```

**Think about all potential inputs, no matter how peculiar**

```
char digit_to_char(int i) {  
    char convert[] = "0123456789";  
    if(i < 0 || i > 9)  
        return '?';  
    return convert[i];  
}
```

**Enforce rule compliance at runtime**



# Rule: Use safe string functions

- Traditional string library routines assume target buffers have sufficient length

```
char str[4];  
char buf[10] = "good";  
strcpy(str, "hello"); // overflows str  
strcat(buf, " day to you"); // overflows buf
```

- Safe versions check the destination length

```
char str[4];  
char buf[10] = "good";  
strncpy(str, "hello", sizeof(str)); //fails  
strncat(buf, " day to you", sizeof(buf)); //fails
```

# Replacements

- ... for string-oriented functions

- `strcat`  $\Rightarrow$  `strlcat`
- `strcpy`  $\Rightarrow$  `strlcpy`
- `strncat`  $\Rightarrow$  `strlcat`
- `strncpy`  $\Rightarrow$  `strlcpy`
- `sprintf`  $\Rightarrow$  `snprintf`
- `vsprintf`  $\Rightarrow$  `vsnprintf`
- `gets`  $\Rightarrow$  `fgets`

`strncpy/strncat` do not  
NUL-terminate if they run  
up against the size limit

- Microsoft versions different
  - `strcpy_s`, `strcat_s`, ...

**Note: None of these in and of themselves are “insecure.”  
They are just commonly misused.**

## (Better) **Rule**: Use safe string library

- Libraries designed to ensure strings used safely
  - **Safety first**, despite some performance loss
- Example: Very Secure FTP (**vsftp**) **string library**

```
struct mystr; // impl hidden
```

<http://vsftpd.beasts.org/>

```
void str_alloc_text(struct mystr* p_str,  
                   const char* p_src);  
void str_append_str(struct mystr* p_str,  
                   const struct mystr* p_other);  
int str_equal(const struct mystr* p_str1,  
             const struct mystr* p_str2);  
int str_contains_space(const struct mystr* p_str);  
...
```

- Another example: **C++ std::string** safe string library

**Rule:** Understand pointer arithmetic

# **Rule:** Understand pointer arithmetic

```
int x;  
int *pi = &x;  
char *pc = (char*) &x;
```

# **Rule:** Understand pointer arithmetic

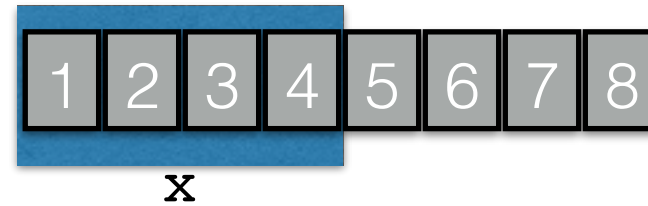
```
int x;  
int *pi = &x;  
char *pc = (char*) &x;
```

`(pi + 1) == (pc + 1) ???`

# Rule: Understand pointer arithmetic

```
int x;  
int *pi = &x;  
char *pc = (char*) &x;
```

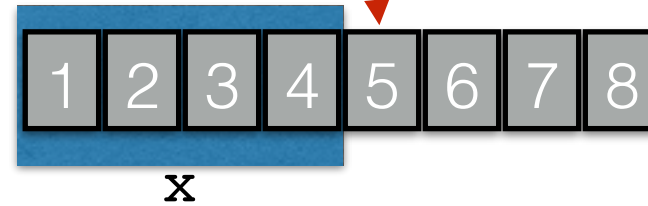
$(pi + 1) == (pc + 1) ???$



# Rule: Understand pointer arithmetic

```
int x;  
int *pi = &x;  
char *pc = (char*) &x;
```

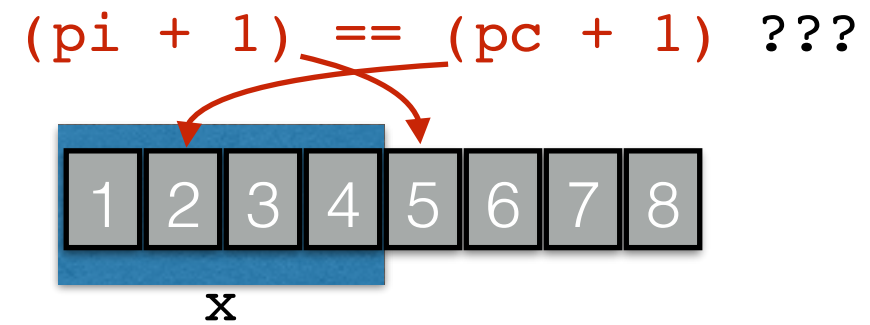
$(pi + 1) == (pc + 1) ???$





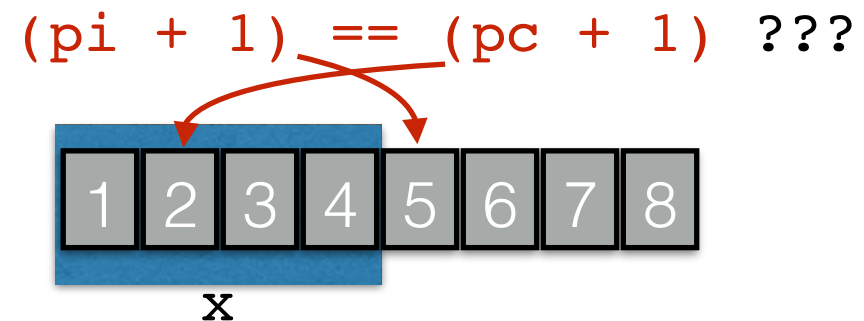
# Rule: Understand pointer arithmetic

```
int x;  
int *pi = &x;  
char *pc = (char*) &x;
```



# Rule: Understand pointer arithmetic

```
int x;  
int *pi = &x;  
char *pc = (char*) &x;
```

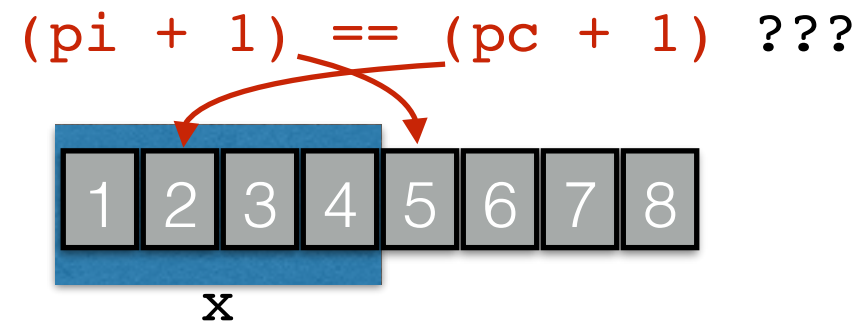


- `sizeof()` returns number of *bytes*, but pointer arithmetic multiplies by the `sizeof` the type

```
int buf[SIZE] = { ... };  
int *buf_ptr = buf;  
  
while (!done() && buf_ptr < (buf + sizeof(buf))) {  
    *buf_ptr++ = getnext(); // will overflow  
}
```

# Rule: Understand pointer arithmetic

```
int x;  
int *pi = &x;  
char *pc = (char*) &x;
```



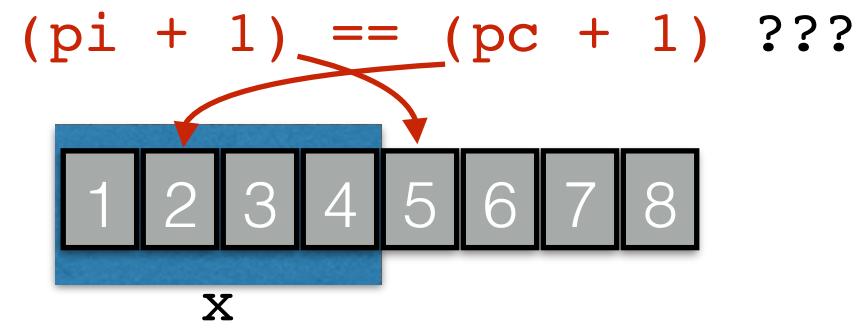
- `sizeof()` returns number of *bytes*, but pointer arithmetic multiplies by the `sizeof` the type

```
int buf[SIZE] = { ... };  
int *buf_ptr = buf;  
  
while (!done() && buf_ptr < (buf + sizeof(buf))) {  
    *buf_ptr++ = getnext(); // will overflow  
}
```

`SIZE * sizeof(int)`

# Rule: Understand pointer arithmetic

```
int x;  
int *pi = &x;  
char *pc = (char*) &x;
```



- `sizeof()` returns number of *bytes*, but pointer arithmetic multiplies by the `sizeof` the type

```
int buf[SIZE] = { ... };  
int *buf_ptr = buf;  
  
while (!done() && buf_ptr < (buf + sizeof(buf))) {  
    *buf_ptr++ = getnext(); // will overflow  
}
```




`SIZE * sizeof(int)`

- So, **use the right units**


```
while (!done() && buf_ptr < (buf + SIZE)) {  
    *buf_ptr++ = getnext(); // stays in bounds  
}
```

# Defend dangling pointers

```
int x = 5;  
int *p = malloc(sizeof(int));  
free(p);  
int **q = malloc(sizeof(int*)); //may reuse p's space  
*q = &x;  
*p = 5;  
**q = 3; //crash (or worse)!
```

x:   
p:   
q: 

Stack



Heap

# Defend dangling pointers

```
int x = 5;  
int *p = malloc(sizeof(int));  
free(p);  
int **q = malloc(sizeof(int*)); //may reuse p's space  
*q = &x;  
*p = 5;  
**q = 3; //crash (or worse)!
```

x: 5

p:

q:

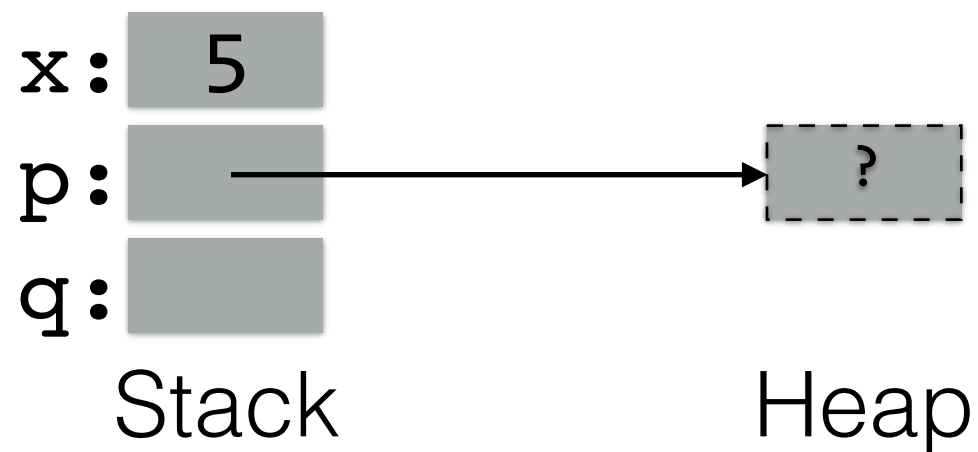
Stack

?

Heap

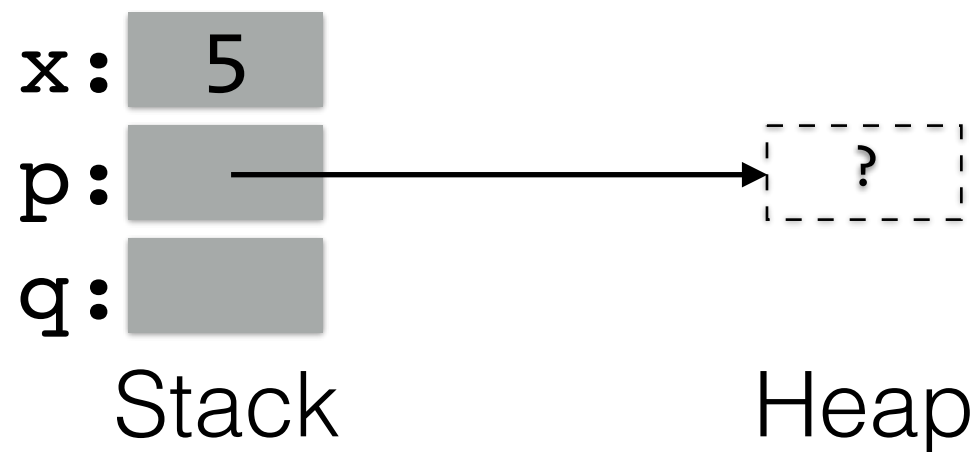
# Defend dangling pointers

```
int x = 5;  
int *p = malloc(sizeof(int));  
free(p);  
int **q = malloc(sizeof(int*)); //may reuse p's space  
*q = &x;  
*p = 5;  
**q = 3; //crash (or worse)!
```



# Defend dangling pointers

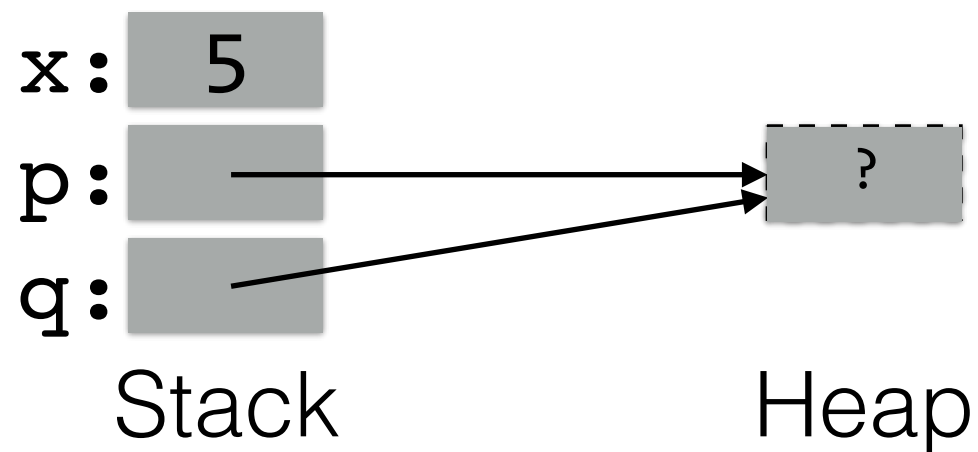
```
int x = 5;  
int *p = malloc(sizeof(int));  
free(p);  
int **q = malloc(sizeof(int*)); //may reuse p's space  
*q = &x;  
*p = 5;  
**q = 3; //crash (or worse)!
```





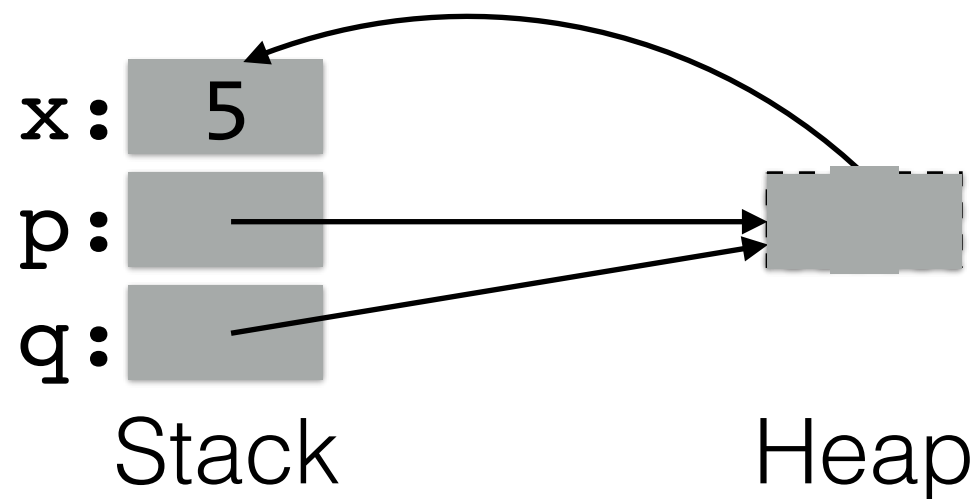
# Defend dangling pointers

```
int x = 5;
int *p = malloc(sizeof(int));
free(p);
int **q = malloc(sizeof(int*)); //may reuse p's space
*q = &x;
*p = 5;
**q = 3; //crash (or worse)!
```



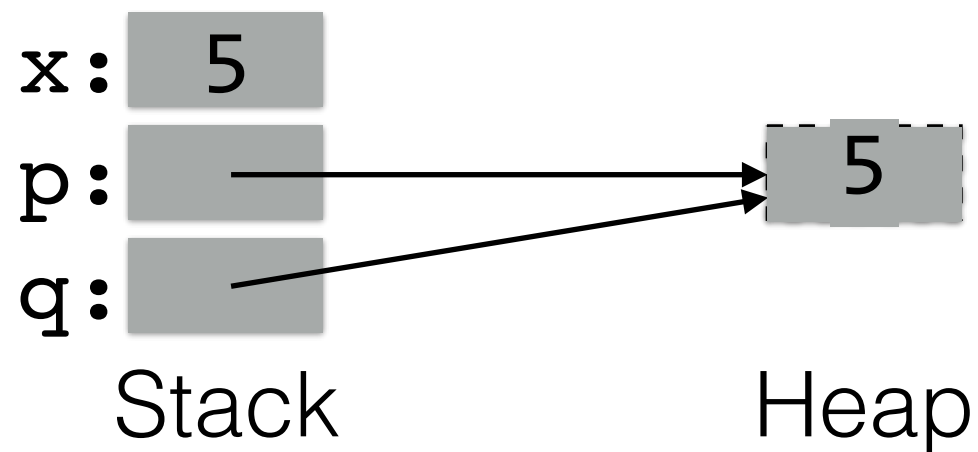
# Defend dangling pointers

```
int x = 5;  
int *p = malloc(sizeof(int));  
free(p);  
int **q = malloc(sizeof(int*)); //may reuse p's space  
*q = &x;  
*p = 5;  
**q = 3; //crash (or worse)!
```



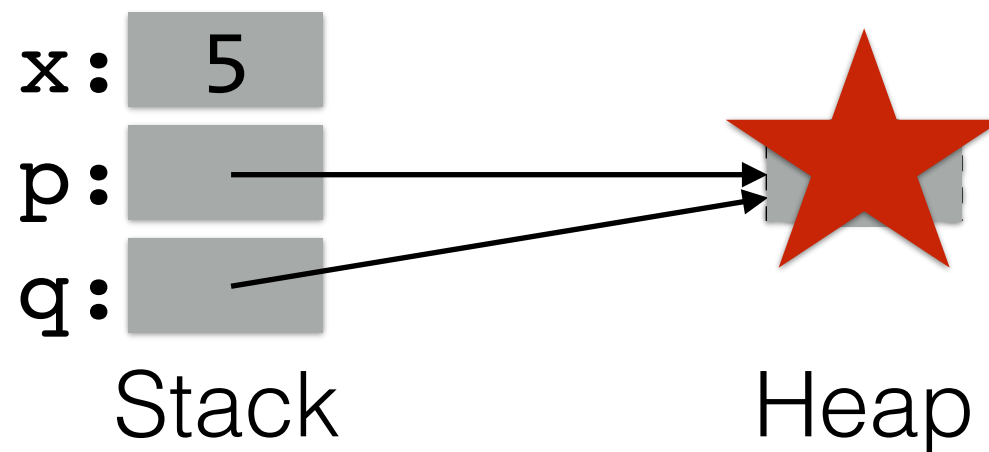
# Defend dangling pointers

```
int x = 5;  
int *p = malloc(sizeof(int));  
free(p);  
int **q = malloc(sizeof(int*)); //may reuse p's space  
*q = &x;  
*p = 5;  
**q = 3; //crash (or worse)!
```



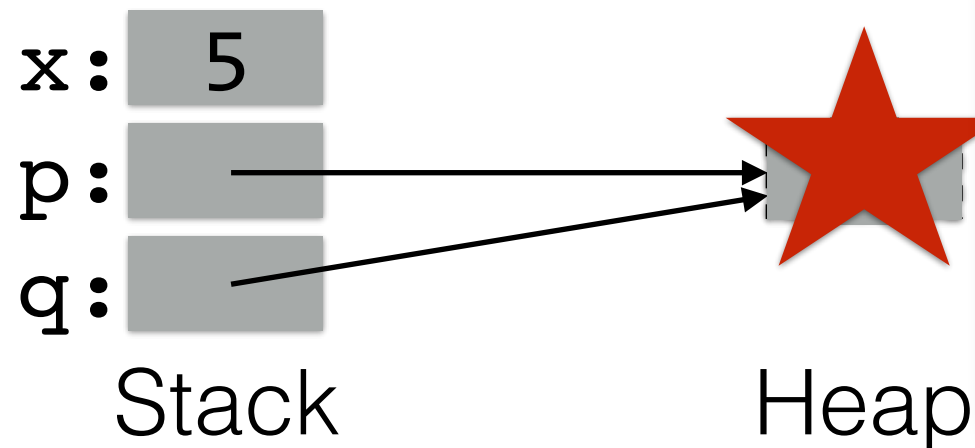
# Defend dangling pointers

```
int x = 5;  
int *p = malloc(sizeof(int));  
free(p);  
int **q = malloc(sizeof(int*)); //may reuse p's space  
*q = &x;  
*p = 5;  
**q = 3; //crash (or worse)!
```






# Defend dangling pointers

```
int x = 5;
int *p = malloc(sizeof(int));
free(p);
int **q = malloc(sizeof(int*)); //may reuse p's space
*q = &x;
*p = 5;
**q = 3; //crash (or worse)!
```




# Rule: Use NULL after free

```
int x = 5;
int *p = malloc(sizeof(int));
free(p);
p = NULL; //defend against bad deref
int **q = malloc(sizeof(int*)); //may reuse p's space
*q = &x;
*p = 5;  //(good) crash
**q = 3;
```

x:   
p:   
q: 

Stack



Heap

# Rule: Use NULL after free

```
int x = 5;  
int *p = malloc(sizeof(int));  
free(p);  
p = NULL; //defend against bad deref  
int **q = malloc(sizeof(int*)); //may reuse p's space  
*q = &x;  
*p = 5; //(good) crash  
**q = 3;
```

x: 5

p:

q:

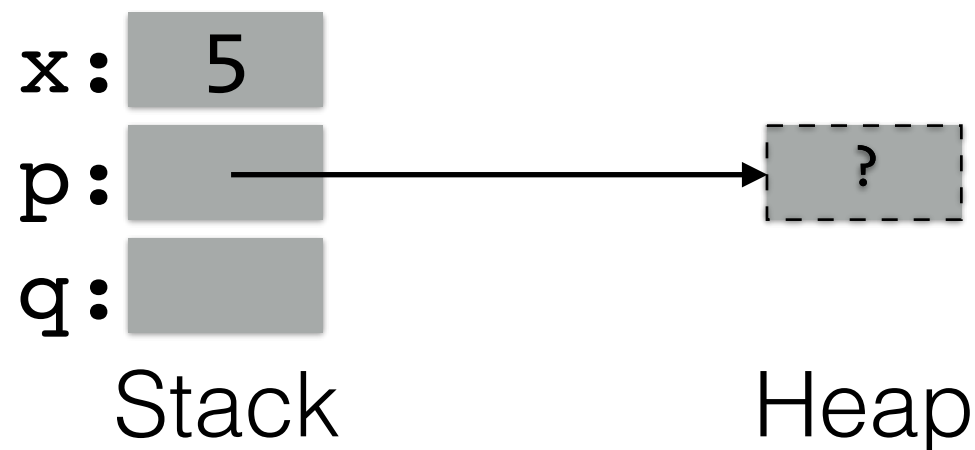
Stack

?

Heap

# Rule: Use NULL after free

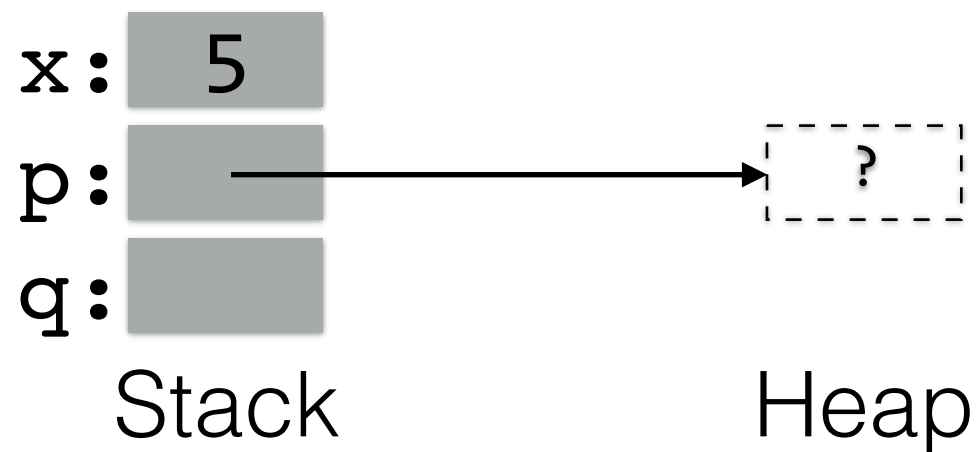
```
int x = 5;  
int *p = malloc(sizeof(int));  
free(p);  
p = NULL; //defend against bad deref  
int **q = malloc(sizeof(int*)); //may reuse p's space  
*q = &x;  
*p = 5; //(good) crash  
**q = 3;
```





# Rule: Use NULL after free

```
int x = 5;  
int *p = malloc(sizeof(int));  
free(p);  
p = NULL; //defend against bad deref  
int **q = malloc(sizeof(int*)); //may reuse p's space  
*q = &x;  
*p = 5;  //(good) crash  
**q = 3;
```



# Rule: Use NULL after free

```
int x = 5;
int *p = malloc(sizeof(int));
free(p);
p = NULL; //defend against bad deref
int **q = malloc(sizeof(int*)); //may reuse p's space
*q = &x;
*p = 5;  //(good) crash
**q = 3;
```

x: 5

p: 0

q:

Stack

?

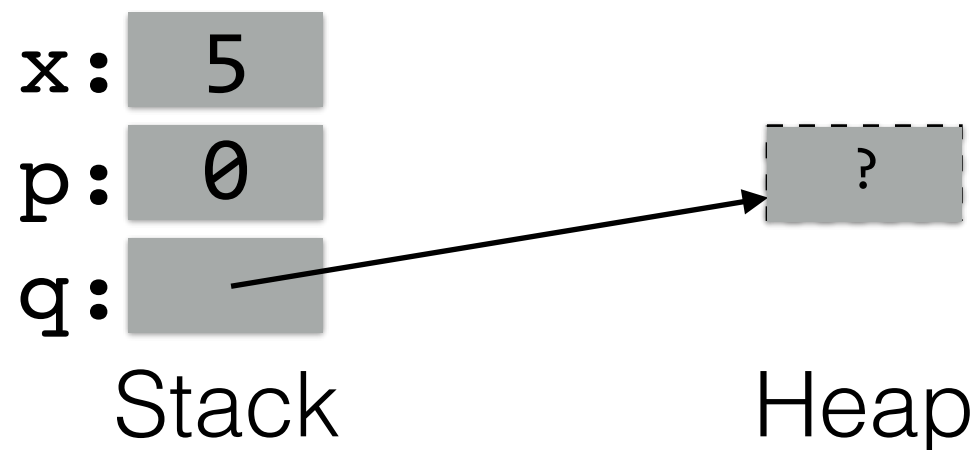
Heap

# Rule: Use NULL after free

```
int x = 5;
int *p = malloc(sizeof(int));
free(p);

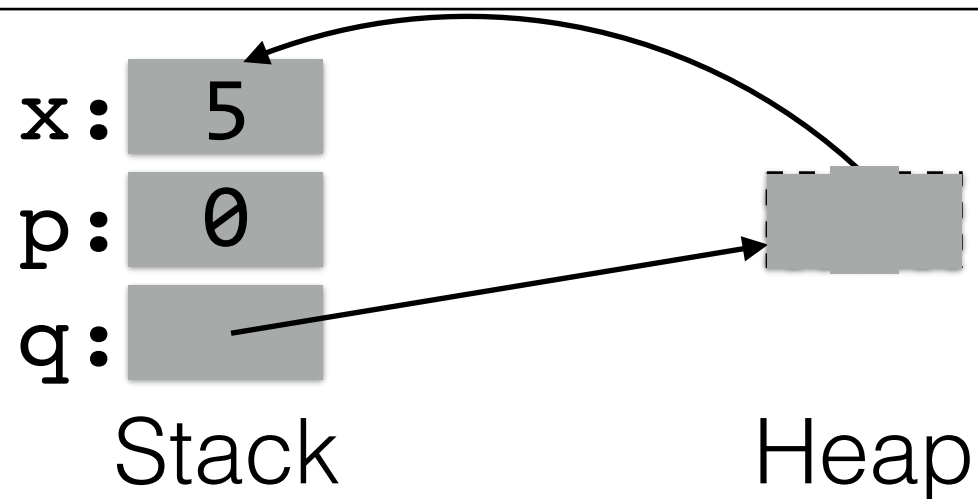

p = NULL; //defend against bad deref


int **q = malloc(sizeof(int*)); //may reuse p's space
*q = &x;
*p = 5;  //(good) crash
**q = 3;
```



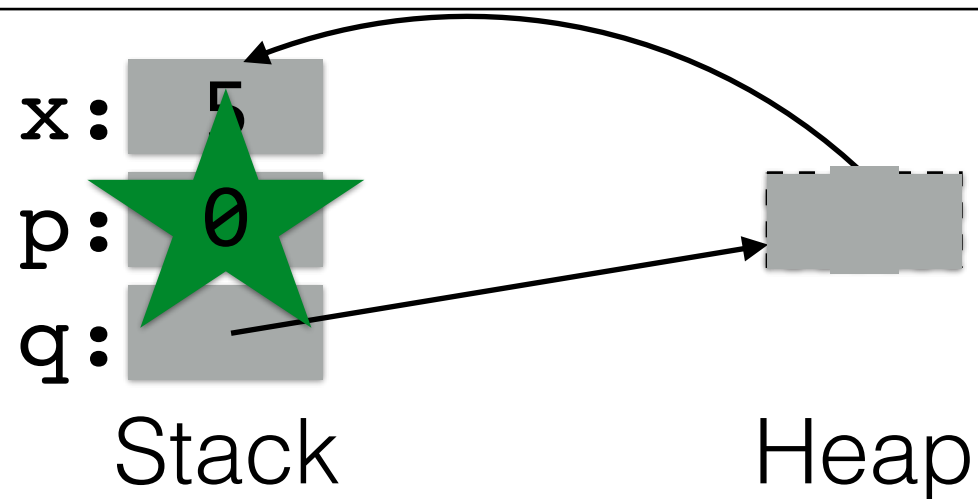
# Rule: Use NULL after free

```
int x = 5;  
int *p = malloc(sizeof(int));  
free(p);  
p = NULL; //defend against bad deref  
int **q = malloc(sizeof(int*)); //may reuse p's space  
*q = &x;  
*p = 5; //(good) crash  
**q = 3;
```



# Rule: Use NULL after free

```
int x = 5;  
int *p = malloc(sizeof(int));  
free(p);  
p = NULL; //defend against bad deref  
int **q = malloc(sizeof(int*)); //may reuse p's space  
*q = &x;  
*p = 5; //(good) crash  
**q = 3;
```



# Manage memory properly

```
int foo(int arg1, int arg2) {
    struct foo *pf1, *pf2;
    int retc = -1;

    pf1 = malloc(sizeof(struct foo));
    if (!isok(arg1)) goto DONE;
    ...
    pf2 = malloc(sizeof(struct foo));
    if (!isok(arg2)) goto FAIL_ARG2;
    ...
    retc = 0;

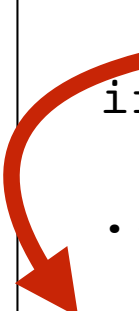
FAIL_ARG2:
    free(pf2); //fallthru
DONE:
    free(pf1);
    return retc;
}
```

- Common approach in C: *goto chains* to avoid duplicated or missed code
  - Like try/finally in languages like Java
- Confirm your logic!...

# Anatomy of a goto fail

```
static OSStatus
SSLVerifySignedServerKeyExchange(...)
{
    OSStatus      err;
    ...

    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
    goto fail; // triggers if if fails: err == 0
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;
    ... // SSL verify called somewhere in here
fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err; // returns err = 0 (SUCCESS), without SSL verify function
}
```



# Rule: Use a safe allocator

- ASLR challenges exploits by making the base address of libraries unpredictable
- **Challenge heap-based overflows** by making the **addresses** returned by `malloc` **unpredictable**
  - Can have some negative performance impact
- Example implementations:
  - **Windows Fault-Tolerant Heap**
    - [http://msdn.microsoft.com/en-us/library/windows/desktop/dd744764\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/dd744764(v=vs.85).aspx)
  - **DieHard** (on which fault-tolerant heap is based)
    - <http://plasma.cs.umass.edu/emery/diehard.html>



# **Rule:** Favor safe libraries

- **Libraries** encapsulate **well-thought-out design**.  
*Take advantage!*
- **Smart pointers**
  - Pointers with only safe operations
  - Lifetimes managed appropriately
  - First in the Boost library, now a C++11 standard
- **Networking:** Google protocol buffers, Apache Thrift
  - For dealing with network-transmitted data
  - Ensures input validation, parsing, etc.
  - Efficient

# Automated testing

# Automated testing techniques

- **Static code analysis**

- Detects most bugs
- Not automatable: model checking or theorem proving

- **Dynamic code analysis**

- Monitor execution (in a vm?) for memory safety: valgrind, address-sanitizer
- But only checks those executions
- High overhead: not suitable for deployed code

- **Penetration testing**

- actively generate inputs to exploit vulnerabilities
- applicable to programs, applications, network, servers
- **Fuzz testing**: many many random inputs

# Fuzz testing

- **Black box**
  - Tool knows nothing about program or its input
  - **Easy to use**, but most likely **explores only shallow states**
- **Grammar-based**
  - Generates inputs informed by a grammar
  - More work to use, but can explore deeper states
- **Mutation**
  - Take a legal input and mutate it (subject to a grammar)
  - Legal input from human or automated (eg, grammar)
- **White box**
  - Generate inputs (partly) informed by the target program
- **Combinations** of above

# Examples: Radamsa and Blab

- **Radamsa** is a *mutation-based, black box fuzzer*
  - It mutates inputs that are given, passing them along

```
% echo "1 + (2 + (3 + 4))" | radamsa --seed 12 -n 4
5!++ (3 + -5))
1 + (3 + 41907596644)
1 + (-4 + (3 + 4))
1 + (2 + (3 + 4
% echo ... | radamsa --seed 12 -n 4 | bc -l
```

- **Blab** generates inputs according to a grammar (*grammar-based*), specified as regexps and CFGs

```
% blab -e '([wrstp][aeiouy]{1,2}){1,4} 32){5} 10'
soty wypisi tisyro to patu
```

# Network-based fuzzing

- Fuzzer can act as
  - an endpoint of a communicating pair
  - a “man-in-the-middle” of a communicating pair
- Inputs can be generated from
  - replays of previously recorded interactions
  - protocol grammar
- Examples
  - **American Fuzzy Lop**: mutation-based white-box fuzzer
  - **SPIKE**: library for creating network-based fuzzers
  - **Burp Intruder**: customized attacks against web apps
  - **BFF**,
  - **Sulley**
  - ...

# You fuzz, you crash. Then what?

Try to find the **root cause**

Is there a smaller input that crashes in the same spot? (Make it easier to understand)

Are there multiple crashes that point back to the same bug?

Determine if this crash represents an **exploitable vulnerability**

In particular, is there a buffer overrun?

# Finding memory errors

1. **Compile** the program with **Address Sanitizer (ASAN)**

- Instruments accesses to arrays to check for overflows, and use-after-free errors
- <https://code.google.com/p/address-sanitizer/>

2. **Fuzz it**

3. Did the program **crash with an ASAN-signaled error**? Then worry about exploitability

- Similarly, you can *compile with other sorts of error checkers* for the purposes of testing
  - E.g., `valgrind memcheck` <http://valgrind.org/>