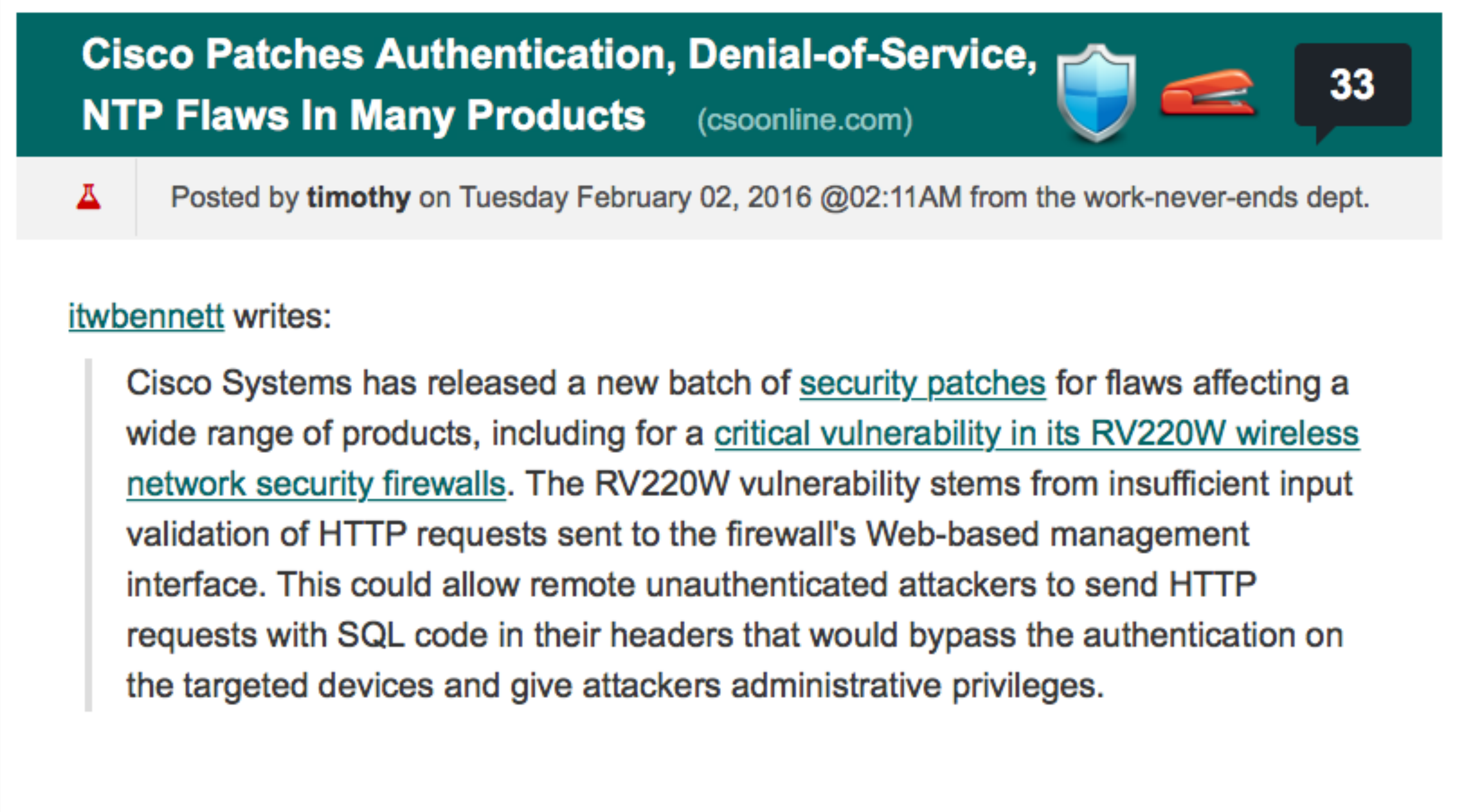


Cisco Patches Authentication, Denial-of-Service, NTP Flaws In Many Products (csoonline.com)

33

Posted by **timothy** on Tuesday February 02, 2016 @02:11AM from the work-never-ends dept.

itwbennett writes:

> Cisco Systems has released a new batch of security patches for flaws affecting a wide range of products, including for a critical vulnerability in its RV220W wireless network security firewalls. The RV220W vulnerability stems from insufficient input validation of HTTP requests sent to the firewall's Web-based management interface. This could allow remote unauthenticated attackers to send HTTP requests with SQL code in their headers that would bypass the authentication on the targeted devices and give attackers administrative privileges.

cs642 computer security

finding vulnerabilities

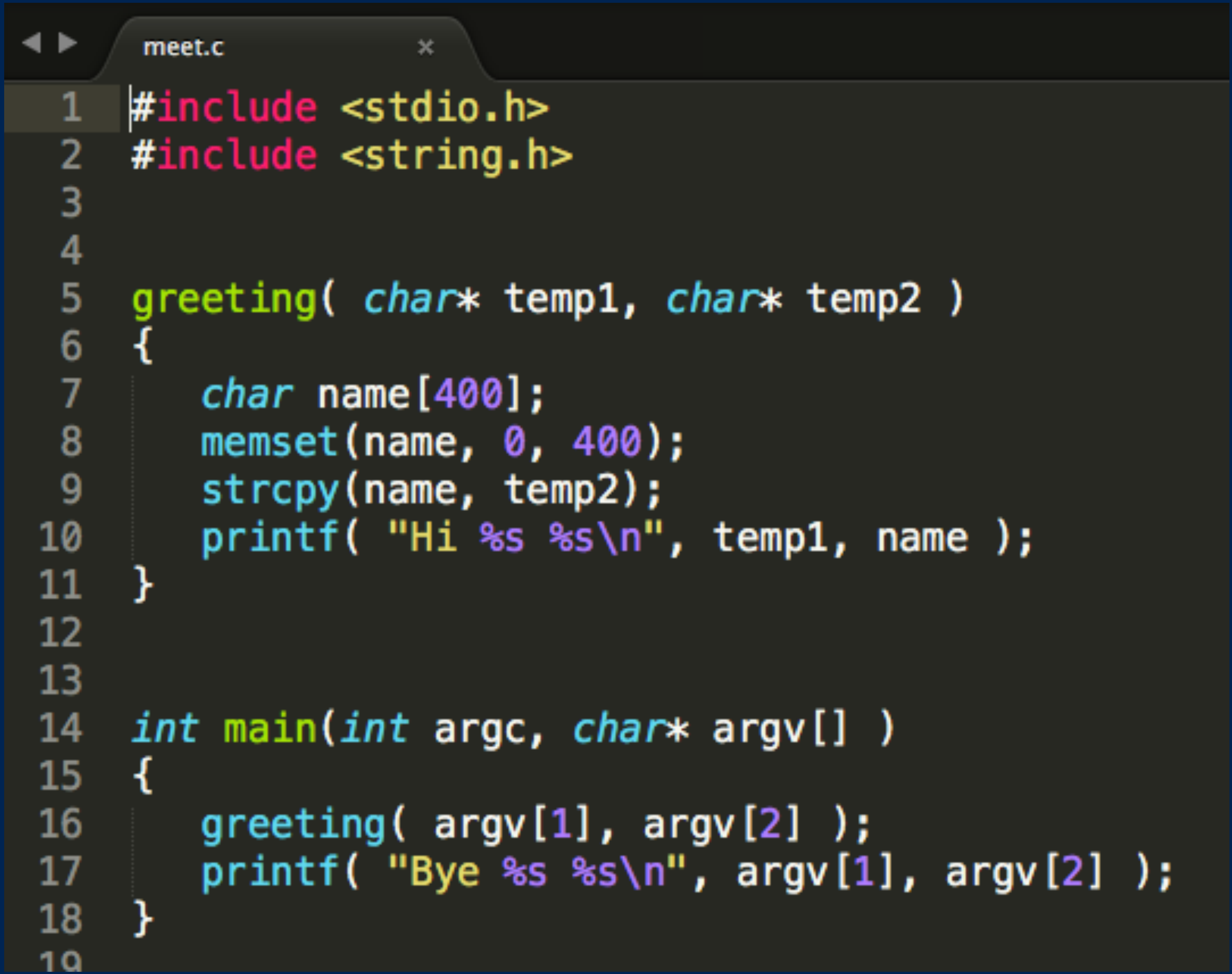
adam everspaugh
ace@cs.wisc.edu

hw1

- Homework 1 will be posted after class today
- Due: Feb 22
- Should be fun!
- TAs can help with setup
- Use Piazza as first step if you need help getting setup

today

- More vulnerabilities from Monday
- Program analyzers
- Software fuzzing
- Static + dynamic analysis

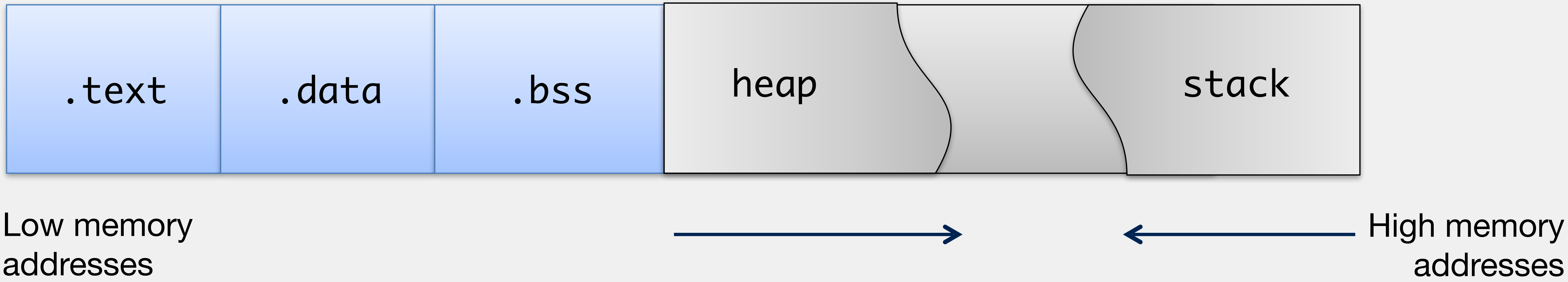


```c
#include <stdio.h>
#include <string.h>


greeting( char* temp1, char* temp2 )
{
    char name[400];
    memset(name, 0, 400);
    strcpy(name, temp2);
    printf( "Hi %s %s\n", temp1, name );
}


int main(int argc, char* argv[] )
{
    greeting( argv[1], argv[2] );
    printf( "Bye %s %s\n", argv[1], argv[2] );
}
```

example


.text

.data

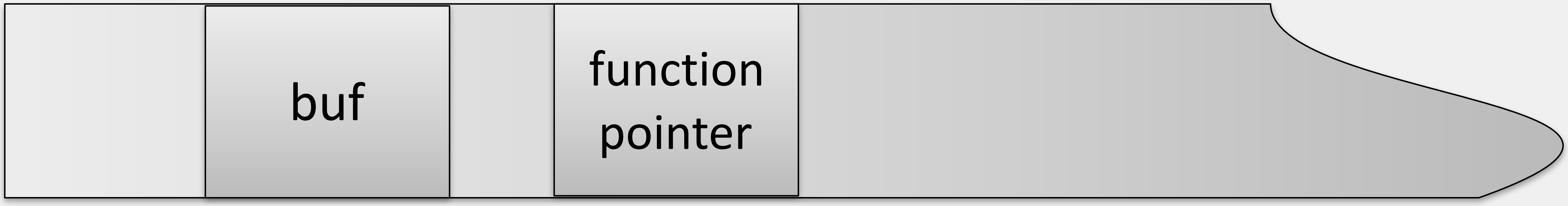
.bss

heap

stack

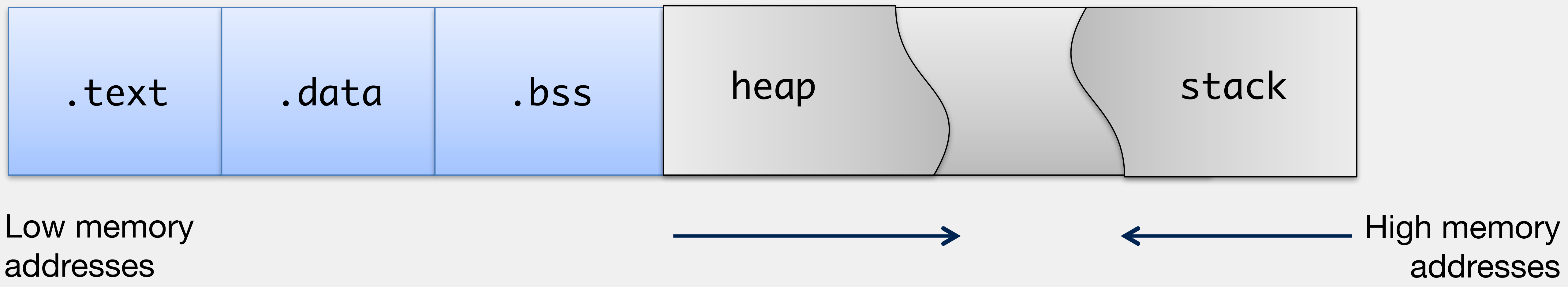
Low memory addresses

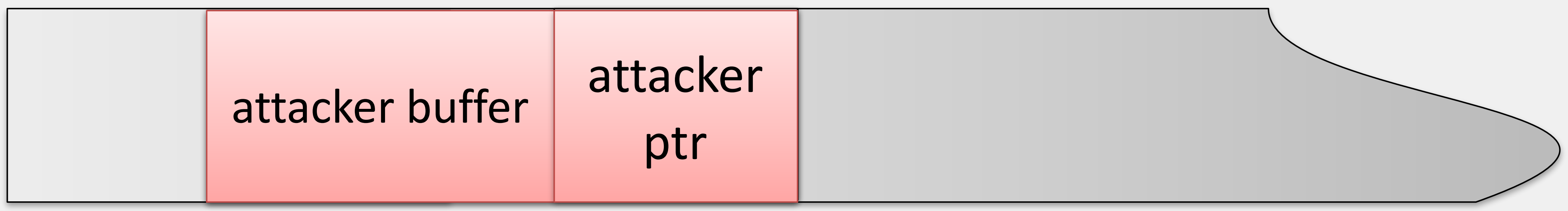
High memory addresses


buf

function pointer

heap overflows





heap overflows

format string vulnerabilities

```
void printf(const char* format, … )

printf("Hi %s %s\n", argv[1], argv[2]);
```

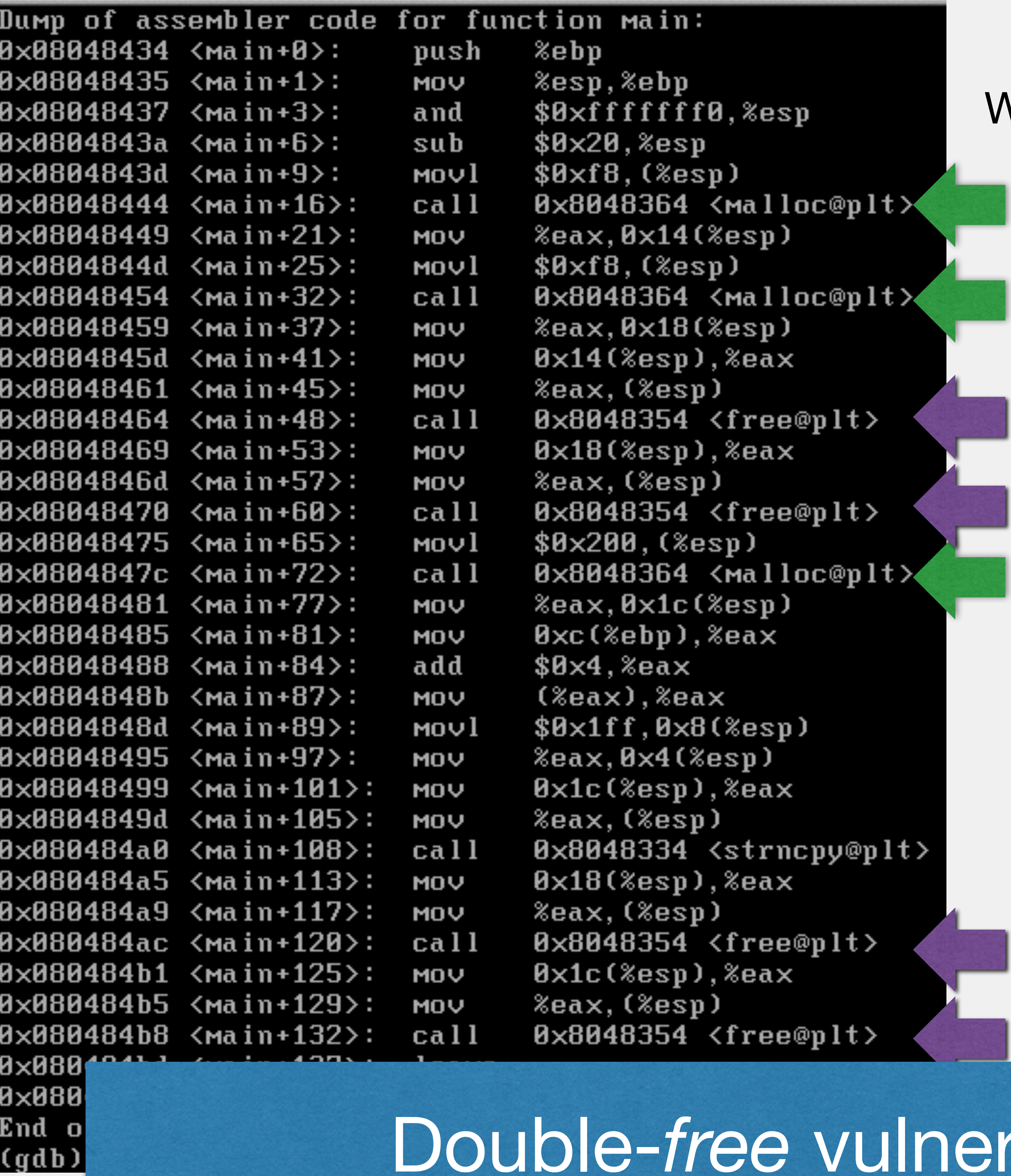
```
void main(int argc, char* argv[]) {
  printf(argv[1]);
}
```

```
What if? argv[1] = "%s%s%s%s%s%s%s%s%s%s%s"

        What if argv[1] = "%p%p%p%p%p%p%p%p%p%p%p"
```

Adversary-controlled format string gives all sorts of control

Can do control flow hijacking directly



```
Dump of assembler code for function main:
0x08048434 <main+0>:     push   %ebp
0x08048435 <main+1>:     mov    %esp,%ebp
0x08048437 <main+3>:     and    $0xfffffff0,%esp
0x0804843a <main+6>:     sub    $0x20,%esp
0x0804843d <main+9>:     movl   $0xf8,(%esp)
0x08048444 <main+16>:    call   0x8048364 <malloc@plt>
0x08048449 <main+21>:    mov    %eax,0x14(%esp)
0x0804844d <main+25>:    movl   $0xf8,(%esp)
0x08048454 <main+32>:    call   0x8048364 <malloc@plt>
0x08048459 <main+37>:    mov    %eax,0x18(%esp)
0x0804845d <main+41>:    mov    0x14(%esp),%eax
0x08048461 <main+45>:    mov    %eax,(%esp)
0x08048464 <main+48>:    call   0x8048354 <free@plt>
0x08048469 <main+53>:    mov    0x18(%esp),%eax
0x0804846d <main+57>:    mov    %eax,(%esp)
0x08048470 <main+60>:    call   0x8048354 <free@plt>
0x08048475 <main+65>:    movl   $0x200,(%esp)
0x0804847c <main+72>:    call   0x8048364 <malloc@plt>
0x08048481 <main+77>:    mov    %eax,0x1c(%esp)
0x08048485 <main+81>:    mov    0xc(%ebp),%eax
0x08048488 <main+84>:    add    $0x4,%eax
0x0804848b <main+87>:    mov    (%eax),%eax
0x0804848d <main+89>:    movl   $0x1ff,0x8(%esp)
0x08048495 <main+97>:    mov    %eax,0x4(%esp)
0x08048499 <main+101>:   mov    0x1c(%esp),%eax
0x0804849d <main+105>:   mov    %eax,(%esp)
0x080484a0 <main+108>:   call   0x8048334 <strncpy@plt>
0x080484a5 <main+113>:   mov    0x18(%esp),%eax
0x080484a9 <main+117>:   mov    %eax,(%esp)
0x080484ac <main+120>:   call   0x8048354 <free@plt>
0x080484b1 <main+125>:   mov    0x1c(%esp),%eax
0x080484b5 <main+129>:   mov    %eax,(%esp)
0x080484b8 <main+132>:   call   0x8048354 <free@plt>
0x080484bd <main+137>:   leave
0x080
End o
(gdb)
```

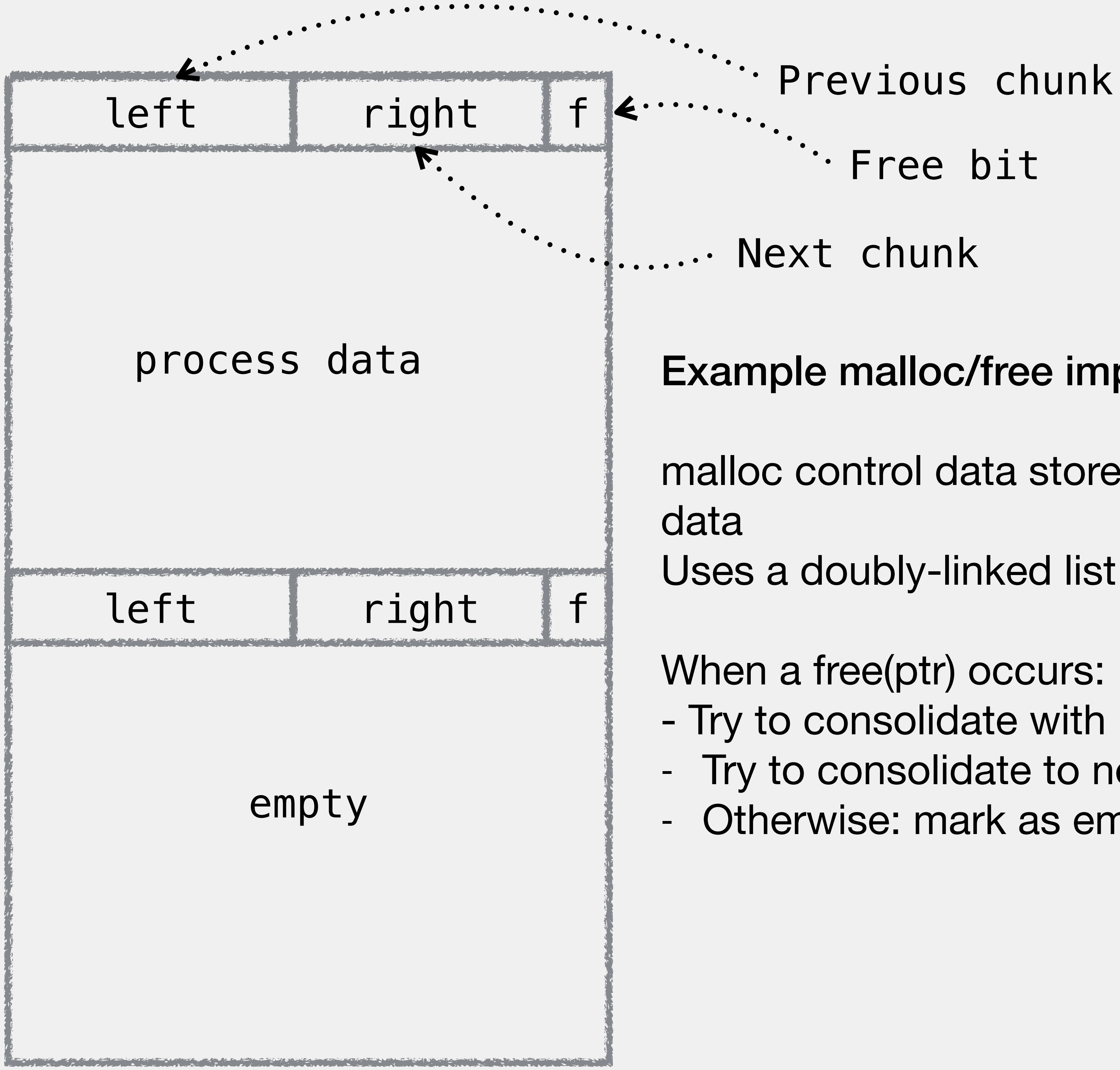
What type of vulnerability is this?

```
main(int argc, char* argv[]) {
  char* b1;
  char* b2;
  char* b3;

  b1 = (char*)malloc(248);
  b2 = (char*)malloc(248);
  free(b1);
  free(b2);
  b3 = (char*)malloc(512);
  strncpy( b3, argv[1], 511 );
  free(b2);
  free(b3);
}
```

Double-*free* vulnerability

malloc implementation



left | right | f

Previous chunk

Free bit

Next chunk

process data

left | right | f

empty

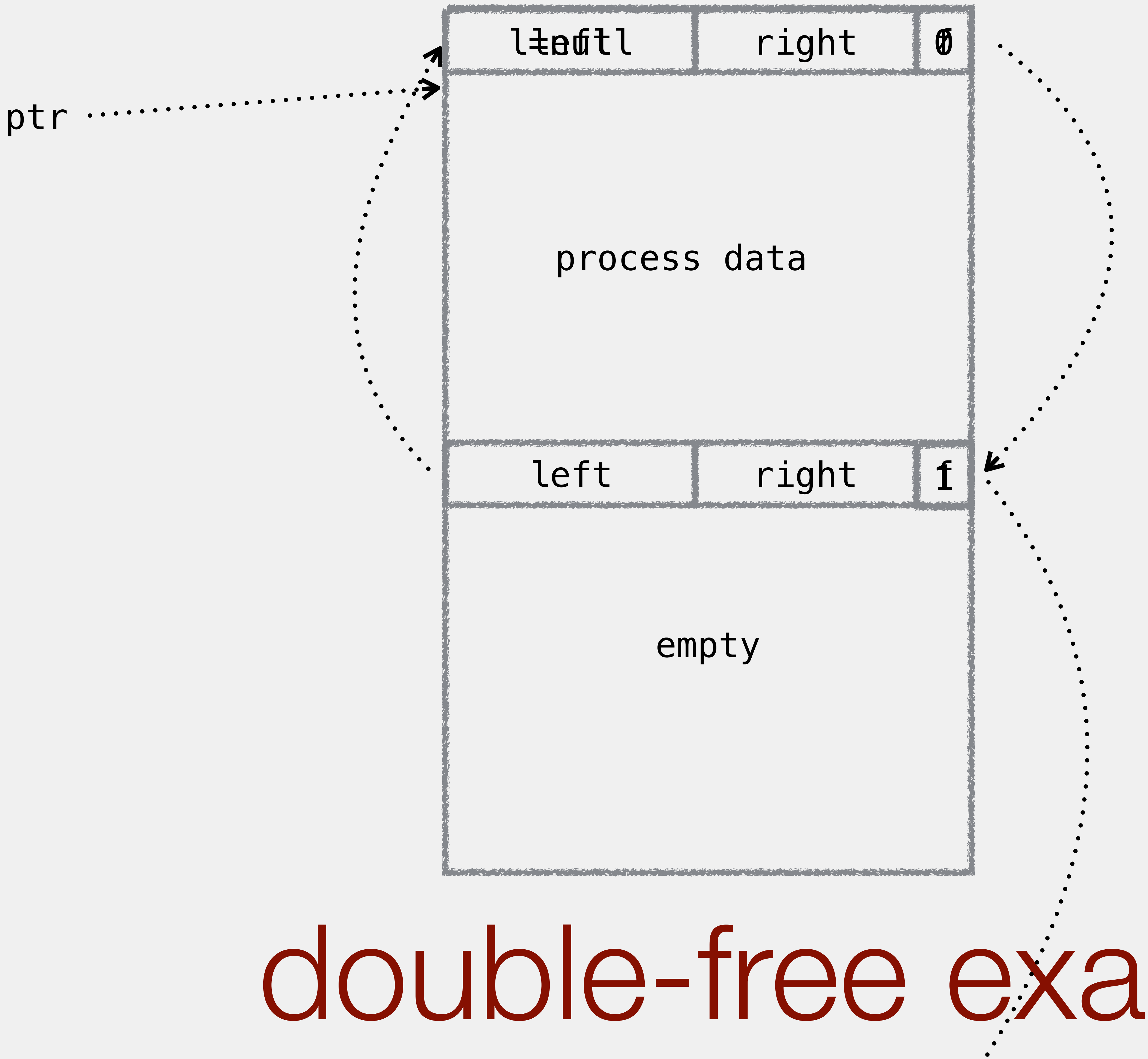
Example malloc/free implementation

malloc control data stored alongside process data

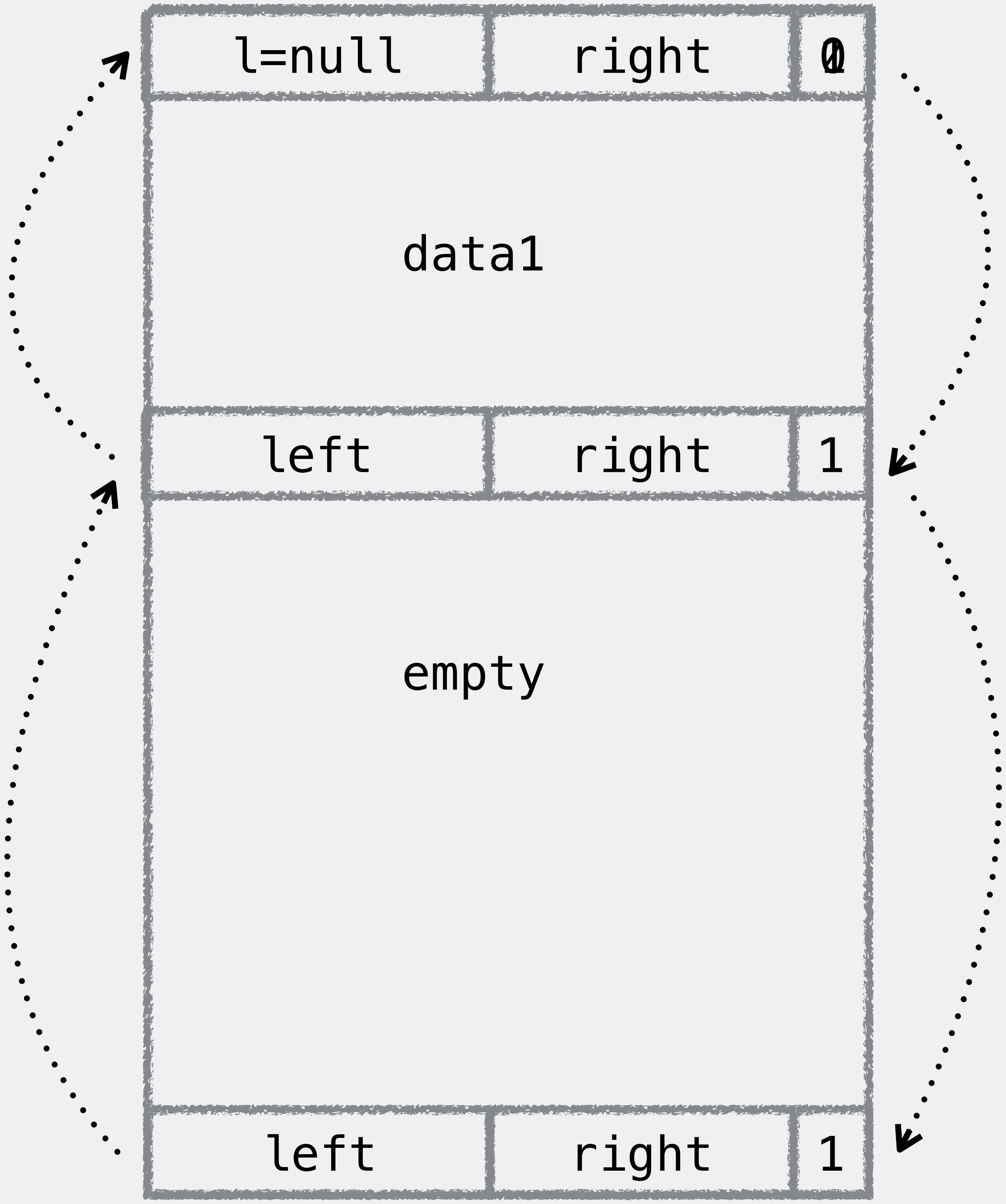
Uses a doubly-linked list to manage heap

When a free(ptr) occurs:

- Try to consolidate with prev if empty
- Try to consolidate to next if empty
- Otherwise: mark as empty



double-free example

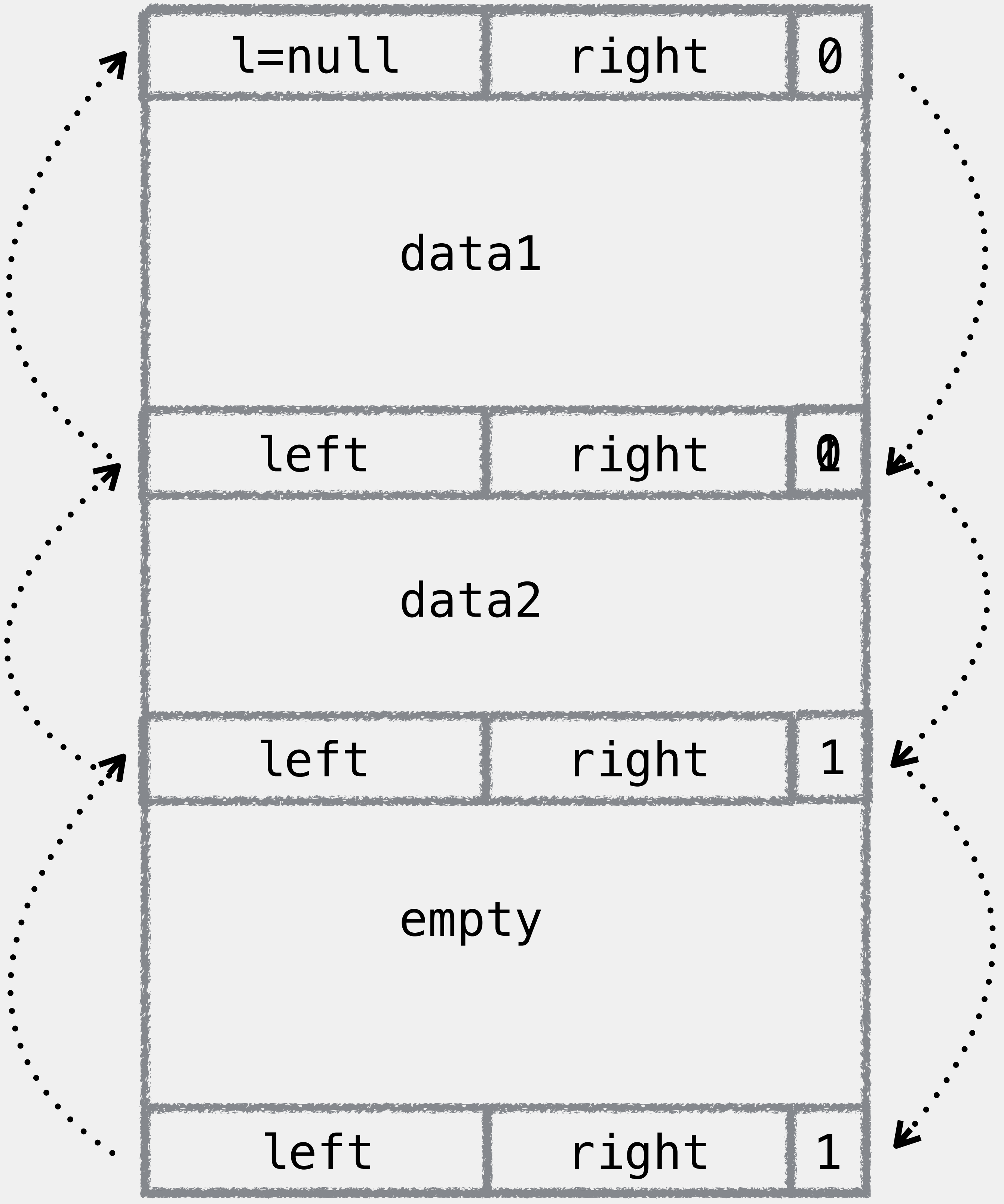


malloc()

- Searches left-to-right for free chunk
- Modifies pointers

```
b1 = malloc(BUF_SIZE1);
```

double-free example

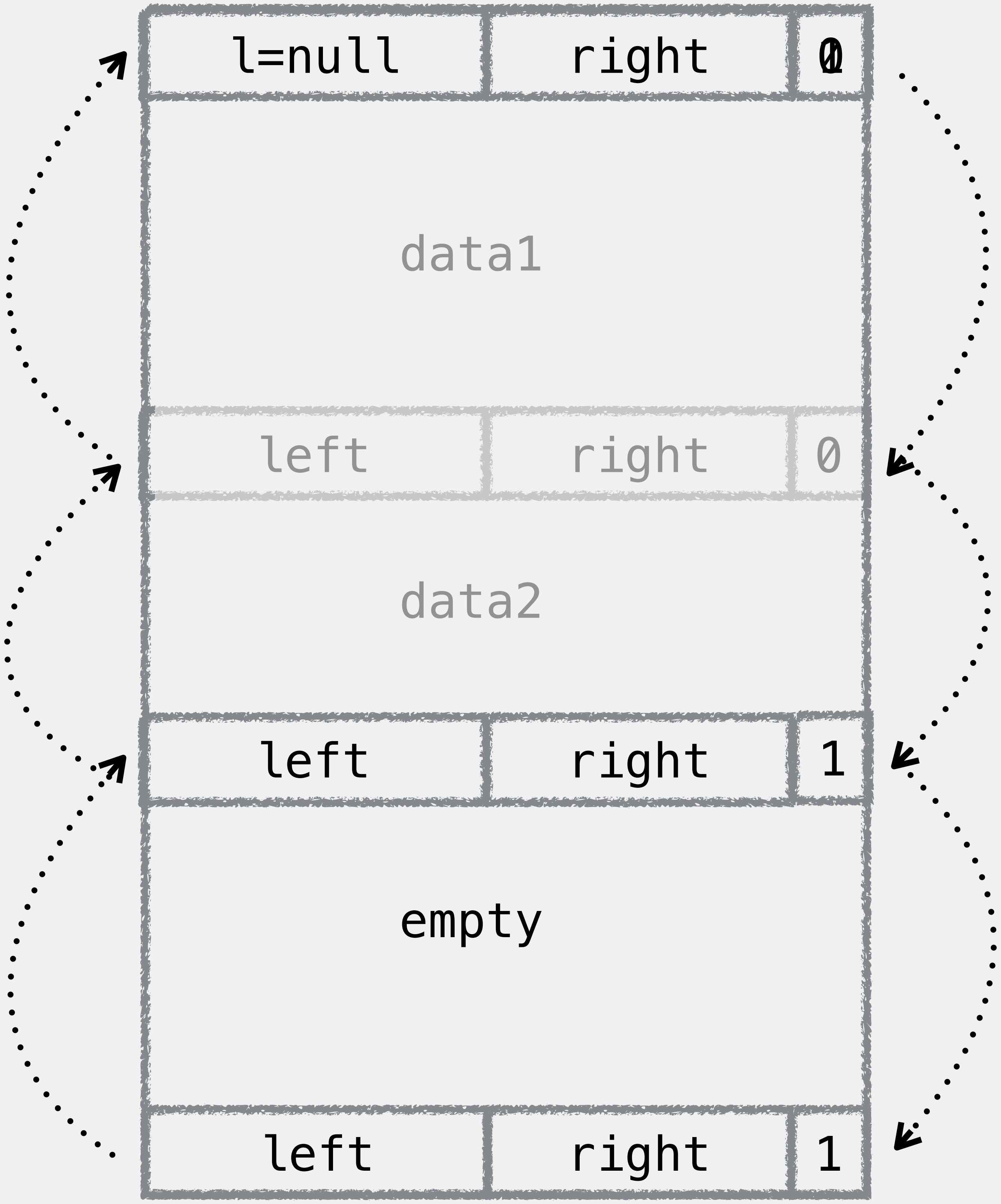


malloc()

- Searches left-to-right for free chunk
- Modifies pointers

```
b1 = malloc(BUF_SIZE1);
b2 = malloc(BUF_SIZE2);
```

double-free example



l=null | right | 0

data1

left | right | 0

data2

left | right | 1

empty

left | right | 1

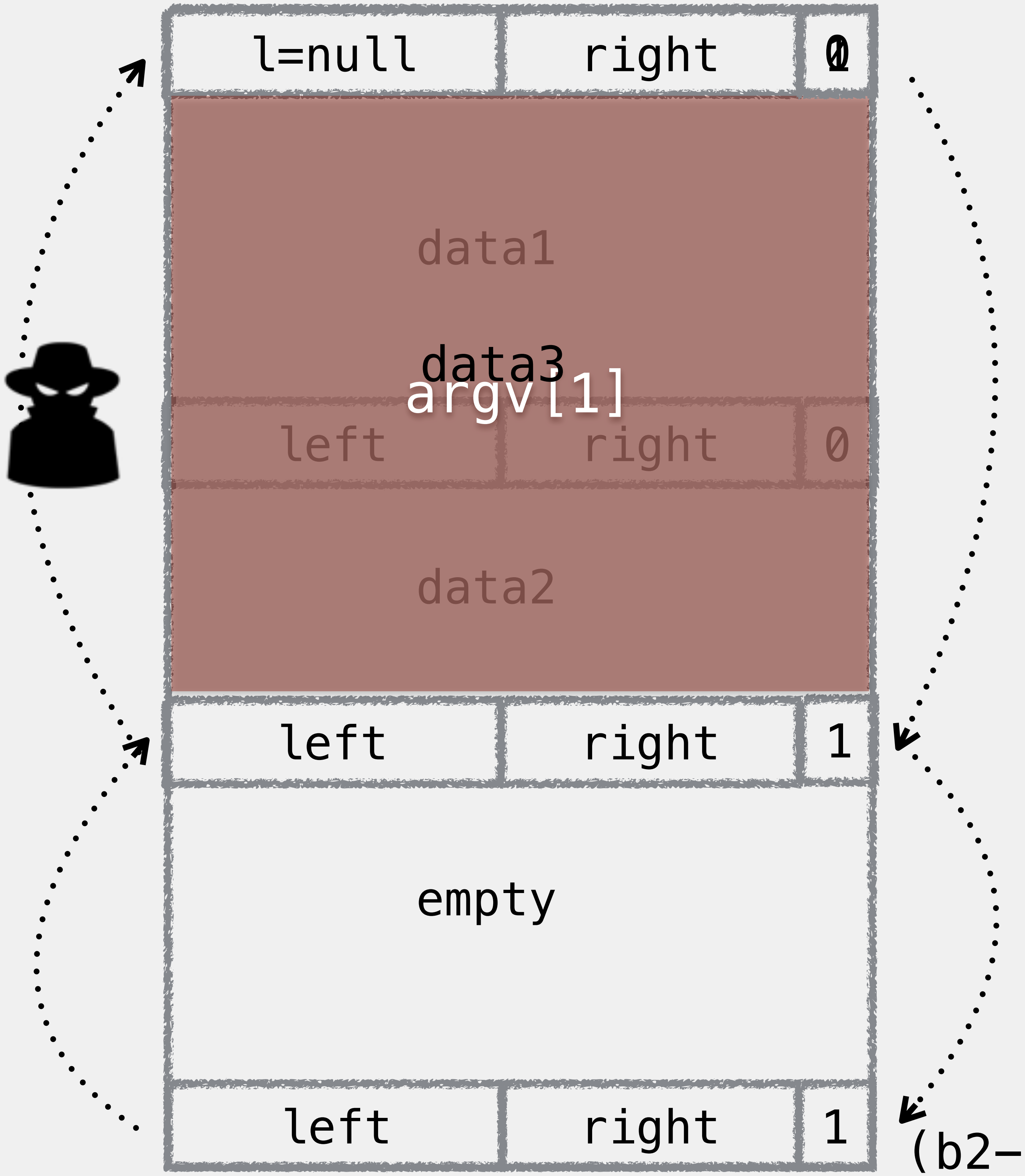
malloc()

- Searches left-to-right for free chunk
- Modifies pointers

```
b1 = malloc(BUF_SIZE1);
b2 = malloc(BUF_SIZE2);
free(b1);
free(b2);
```

double-free example

double-free example



malloc()

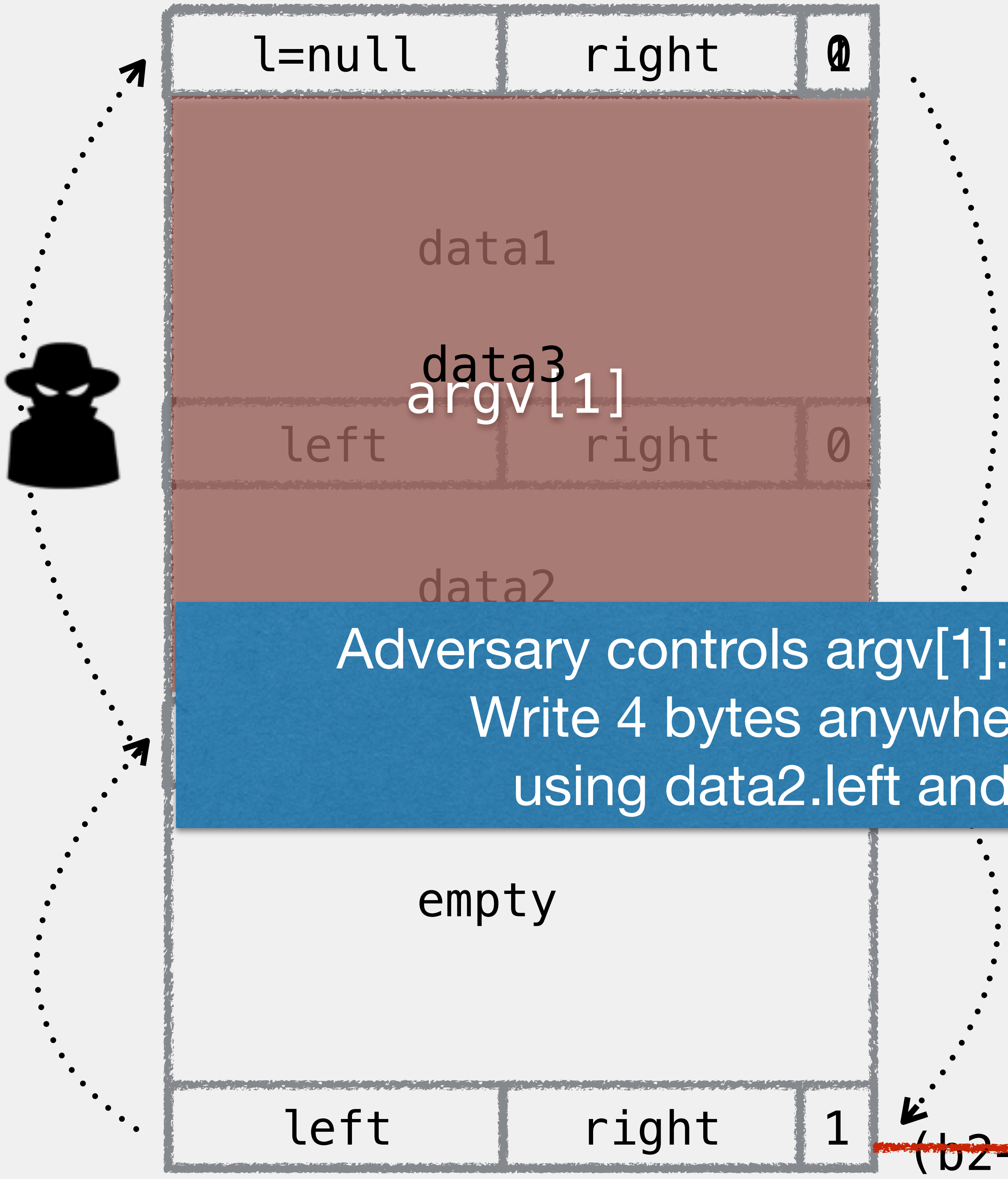
- Searches left-to-right for free chunk
- Modifies pointers

```
b1 = malloc(BUF_SIZE1);
b2 = malloc(BUF_SIZE2);
free(b1);
free(b2);

b3 = malloc(BUF_SIZE1 +
   BUF_SIZE2);
strncpy(b3, argv[1],
    BUF_SZ1+BUF_SZ2-1);
free(b2);
free(b3);
```

```
(b2-8)->left->right = (b2-8)->right
(b2-8)->right->left = (b2-8)->left
```

exercise: think-pair-share



Adversary controls argv[1]: what can she do?
Write 4 bytes anywhere in memory
using data2.left and data2.right

malloc()

- Searches left-to-right for free chunk
- Modifies pointers

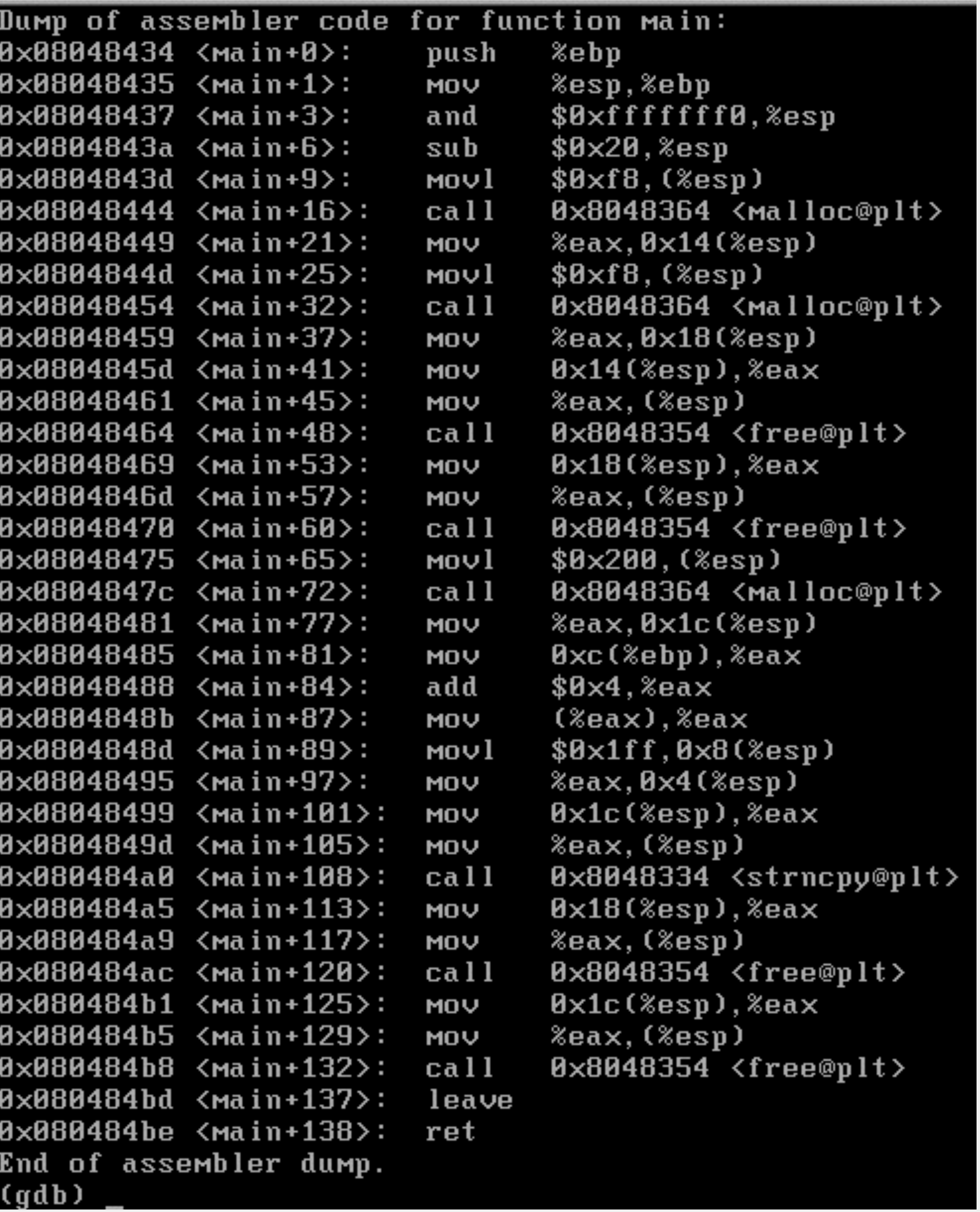
```
b1 = malloc(BUF_SIZE1);
b2 = malloc(BUF_SIZE2);
free(b1);

                  ZE1 +

strncpy(b3, argv[1],
     BUF_SZ1+BUF_SZ2-1);
```
`free(b2);`
```
free(b3);
```

~~(b2-8)->left->right = (b2-8)->right~~
```
(b2-8)->right->left = (b2-8)->left
```

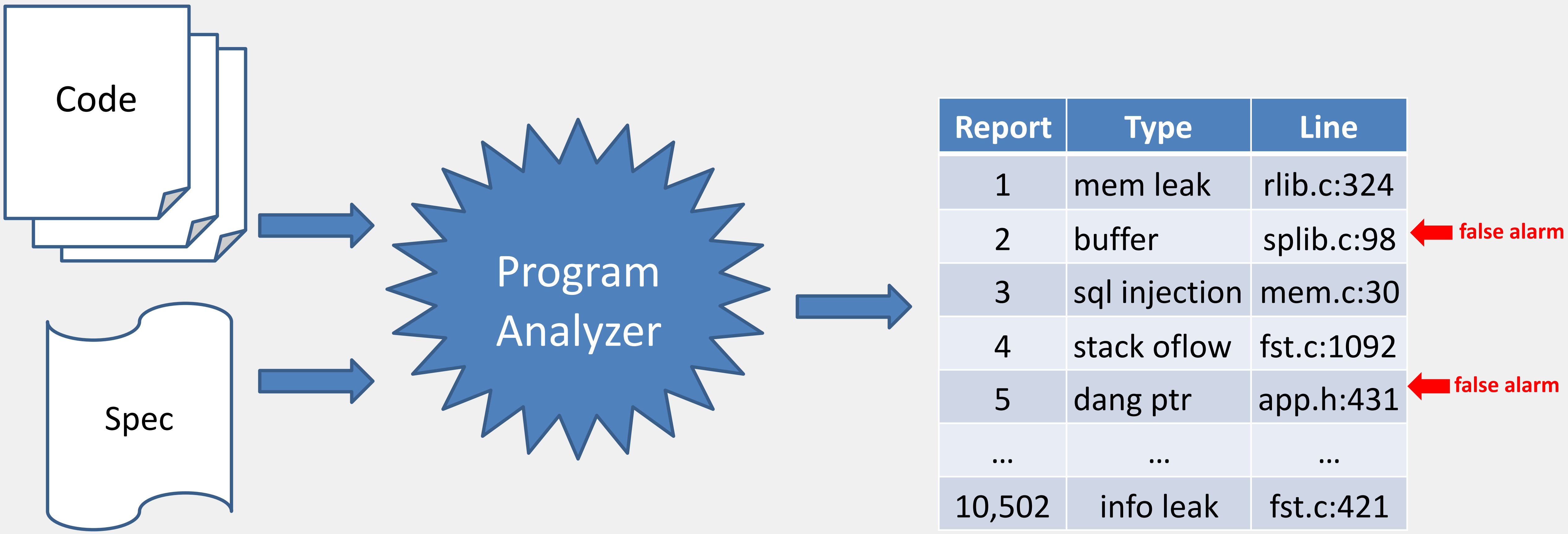
manual analysis



```
Dump of assembler code for function main:
0x08048434 <main+0>:    push   %ebp
0x08048435 <main+1>:    mov    %esp,%ebp
0x08048437 <main+3>:    and    $0xfffffff0,%esp
0x0804843a <main+6>:    sub    $0x20,%esp
0x0804843d <main+9>:    movl   $0xf8,(%esp)
0x08048444 <main+16>:   call   0x8048364 <malloc@plt>
0x08048449 <main+21>:   mov    %eax,0x14(%esp)
0x0804844d <main+25>:   movl   $0xf8,(%esp)
0x08048454 <main+32>:   call   0x8048364 <malloc@plt>
0x08048459 <main+37>:   mov    %eax,0x18(%esp)
0x0804845d <main+41>:   mov    0x14(%esp),%eax
0x08048461 <main+45>:   mov    %eax,(%esp)
0x08048464 <main+48>:   call   0x8048354 <free@plt>
0x08048469 <main+53>:   mov    0x18(%esp),%eax
0x0804846d <main+57>:   mov    %eax,(%esp)
0x08048470 <main+60>:   call   0x8048354 <free@plt>
0x08048475 <main+65>:   movl   $0x200,(%esp)
0x0804847c <main+72>:   call   0x8048364 <malloc@plt>
0x08048481 <main+77>:   mov    %eax,0x1c(%esp)
0x08048485 <main+81>:   mov    0xc(%ebp),%eax
0x08048488 <main+84>:   add    $0x4,%eax
0x0804848b <main+87>:   mov    (%eax),%eax
0x0804848d <main+89>:   movl   $0x1ff,0x8(%esp)
0x08048495 <main+97>:   mov    %eax,0x4(%esp)
0x08048499 <main+101>:  mov    0x1c(%esp),%eax
0x0804849d <main+105>:  mov    %eax,(%esp)
0x080484a0 <main+108>:  call   0x8048334 <strncpy@plt>
0x080484a5 <main+113>:  mov    0x18(%esp),%eax
0x080484a9 <main+117>:  mov    %eax,(%esp)
0x080484ac <main+120>:  call   0x8048354 <free@plt>
0x080484b1 <main+125>:  mov    0x1c(%esp),%eax
0x080484b5 <main+129>:  mov    %eax,(%esp)
0x080484b8 <main+132>:  call   0x8048354 <free@plt>
0x080484bd <main+137>:  leave
0x080484be <main+138>:  ret
End of assembler dump.
(gdb) _
```

- Manual analysis
- Can be effective, but time consuming
- Security analysts use tools to augment analysis

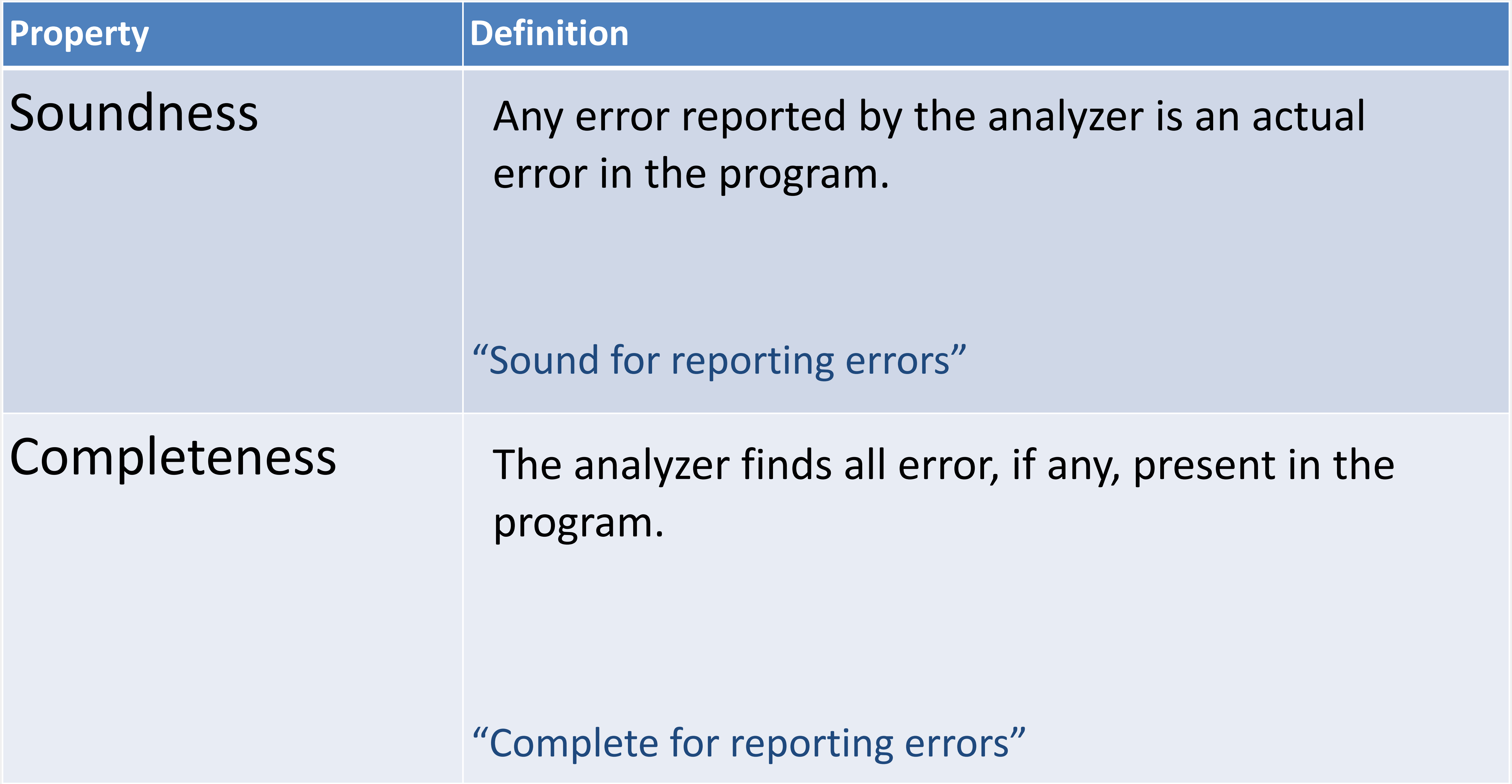
Slide credit: Prof Mitchell Stanford's CS 155



Report	Type	Line
1	mem leak	rlib.c:324
2	buffer	splib.c:98
3	sql injection	mem.c:30
4	stack oflow	fst.c:1092
5	dang ptr	app.h:431
…	…	…
10,502	info leak	fst.c:421

program analyzers

Slide credit: Prof Mitchell Stanford’s CS 155



Property	Definition
Soundness	Any error reported by the analyzer is an actual error in the program. “Sound for reporting errors”
Completeness	The analyzer finds all error, if any, present in the program. “Complete for reporting errors”

sound/complete

- True positive: Actual error
- False positive: Error reported, not a real error
- False negative: Missed an error; not reported

Report	Type	Line
1	mem leak	rlib.c:324
2	buffer	splib.c:98
3	sql injection	mem.c:30
4	stack oflow	fst.c:1092
5	dang ptr	app.h:431
…	…	…
10,502	info leak	fst.c:421

classifying error reports

Slide credit: Prof Mitchell Stanford’s CS 155

	Complete	Incomplete
Sound	Reports all errors Reports no false alarms No false positives No false negatives Undecidable	May not report all errors Reports no false alarms No false positives False negatives Decidable
Unsound	Reports all errors May report false alarms False positives No false negatives Decidable	May not report all errors May report false alarms False negatives False positives Decidable

soundness/completeness over error reports

- Static analysis
  / examines code (source or binary)
  / analyze program without running the program

- Dynamic analysis
  / runs programs on test inputs
  / examines run-time behavior
  / may instrument analysis target

program analyzers

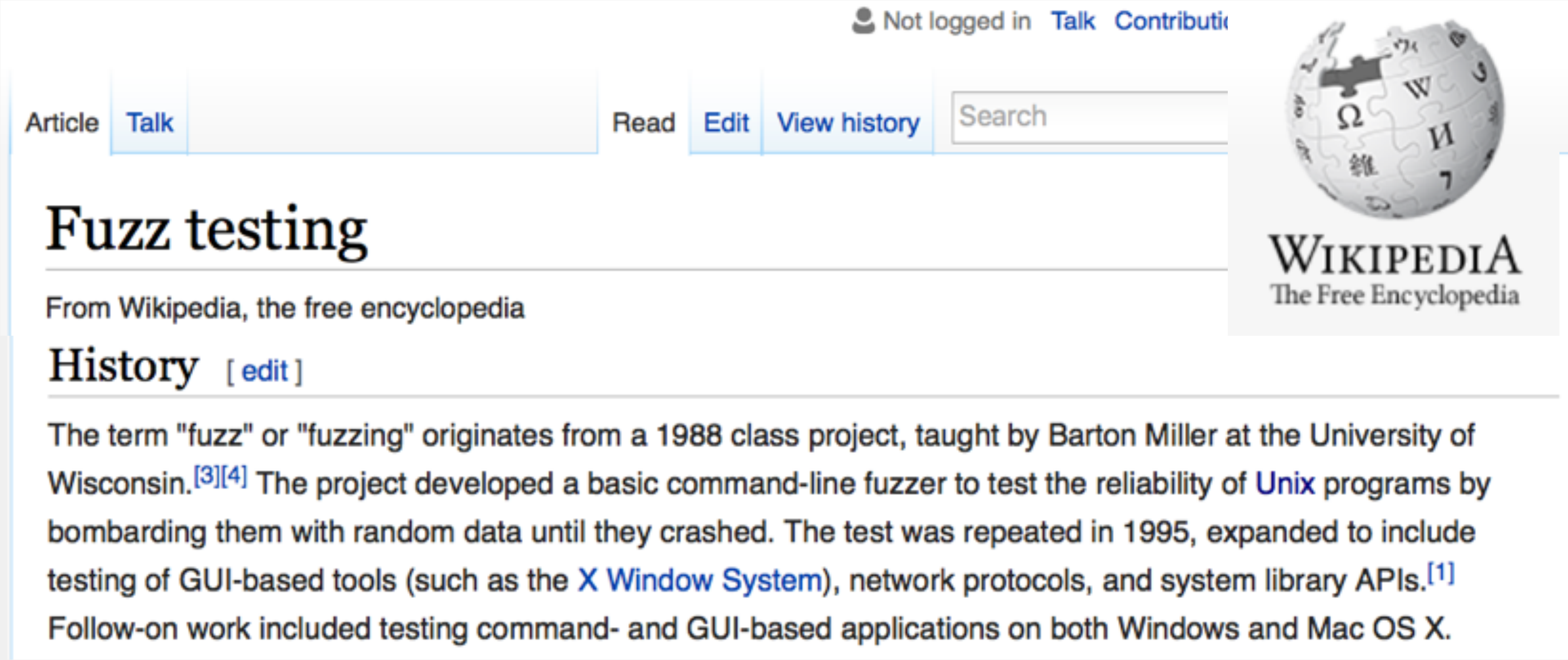
program analyzers

- Static analyzers
  - / grep, lint, gcc -Wall [source code scanners]
    - // Look for suspicious code/patterns (`strcpy, gets`)
  - / Klee, Fie [symbolic execution]
  - / Coverity

- Dynamic analyzers
  - / `assert()`
  - / Valgrind, Purify
  - / Fuzzers

Soundness and completeness of these tools?

fuzzing



Fuzz testing

From Wikipedia, the free encyclopedia

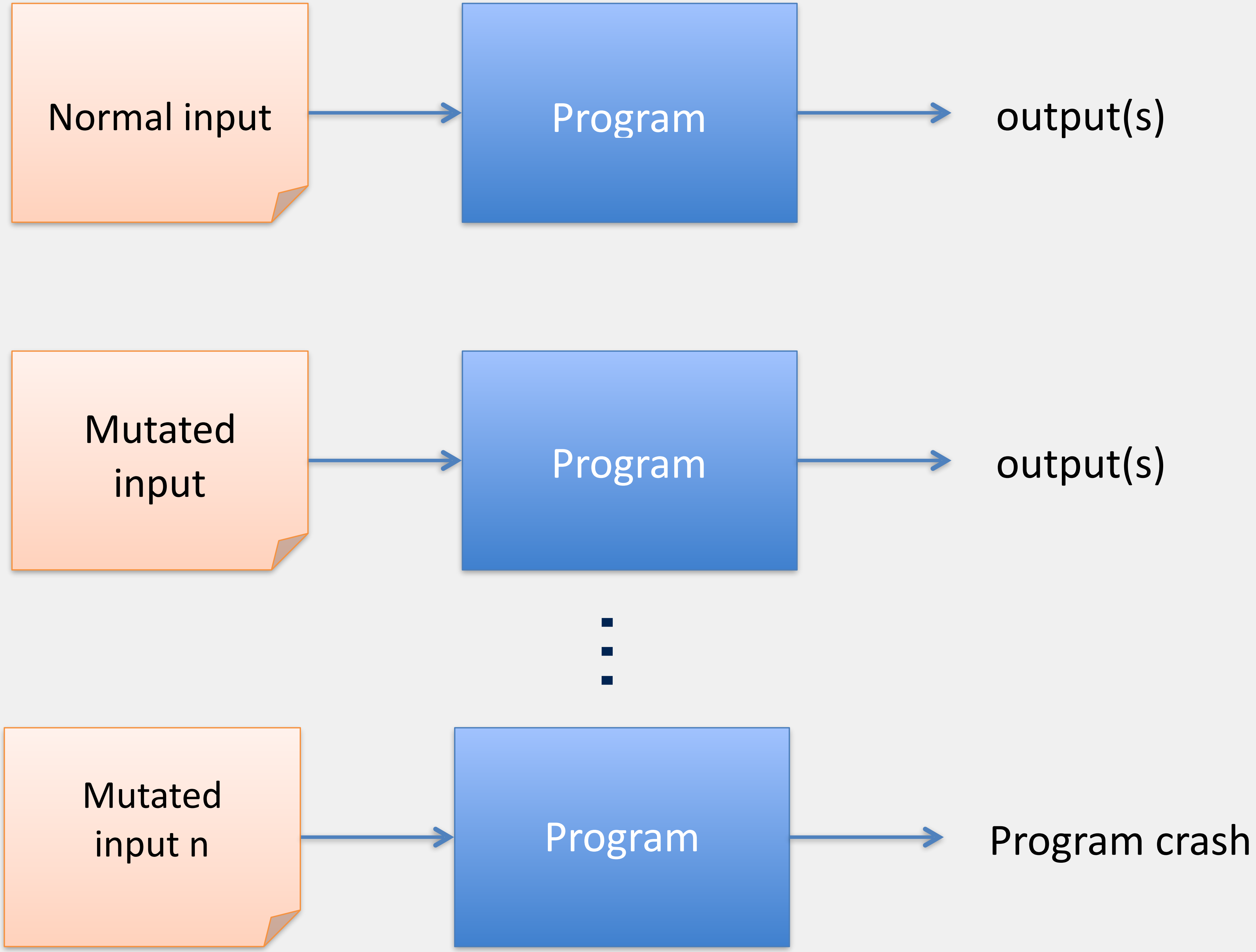
History [edit]

The term "fuzz" or "fuzzing" originates from a 1988 class project, taught by Barton Miller at the University of Wisconsin.[3][4] The project developed a basic command-line fuzzer to test the reliability of Unix programs by bombarding them with random data until they crashed. The test was repeated in 1995, expanded to include testing of GUI-based tools (such as the X Window System), network protocols, and system library APIs.[1] Follow-on work included testing command- and GUI-based applications on both Windows and Mac OS X.

Feed randomized inputs into a program.

Does this work? Why would it work?

What are the limitations?



Normal input

Program

output(s)

Mutated input

Program

output(s)

⋮

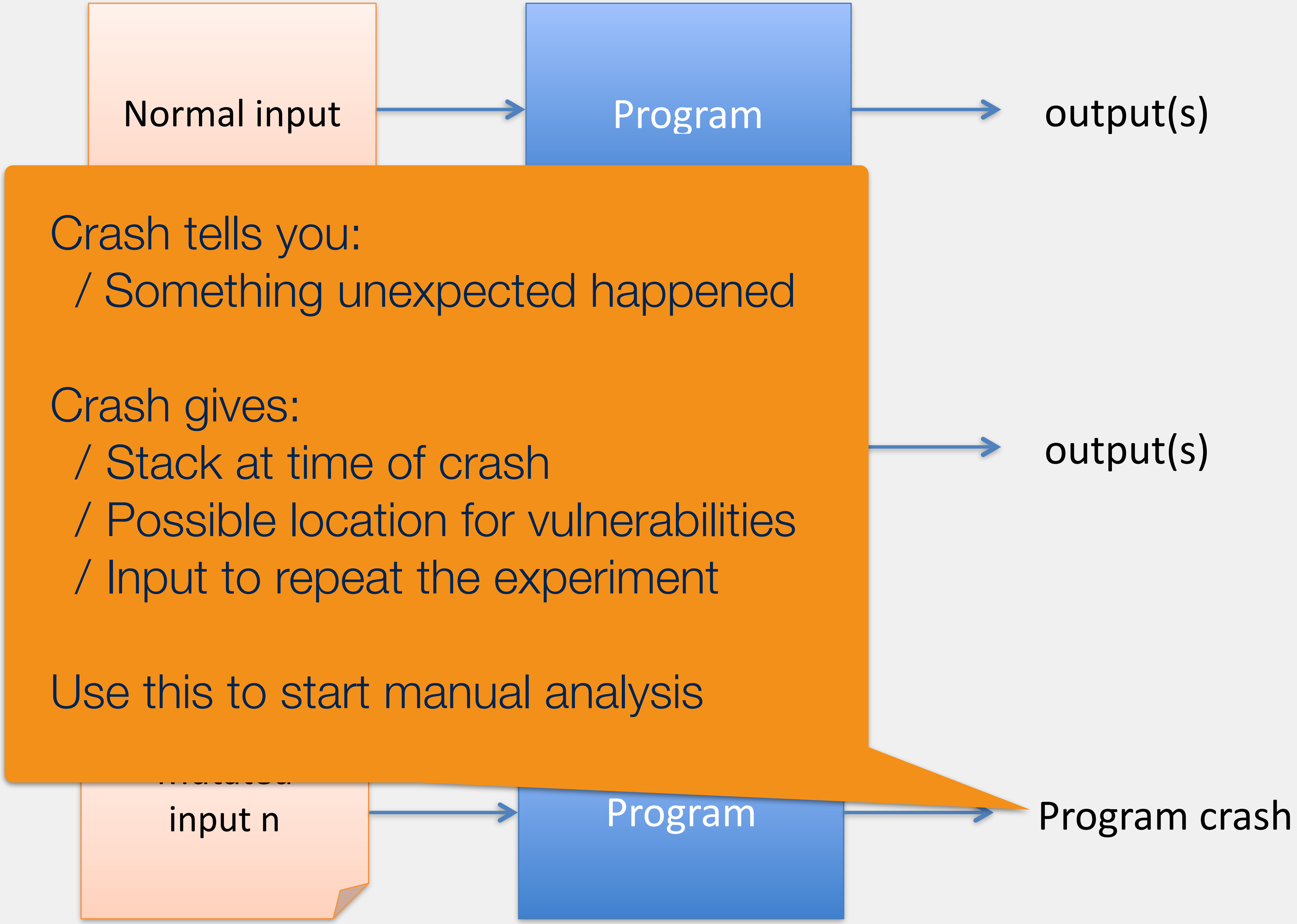
Mutated input n

Program

Program crash

black box fuzzing

black box fuzzing



Crash tells you:
/ Something unexpected happened

Crash gives:
/ Stack at time of crash
/ Possible location for vulnerabilities
/ Input to repeat the experiment

Use this to start manual analysis

- When do you stop fuzzing?

- Can use code-coverage to determine how effective your fuzzing is
  / # of LOCs executed
  / # of basic blocks reached
  / # of paths follows
  / # of conditionals followed

code coverage

- Heap overflows, format string vulnerabilities
- Double-free vulnerabilities
- Program analyzers
  - Soundness + completeness
  - Static + dynamic analyzers
  - Fuzzing

recap