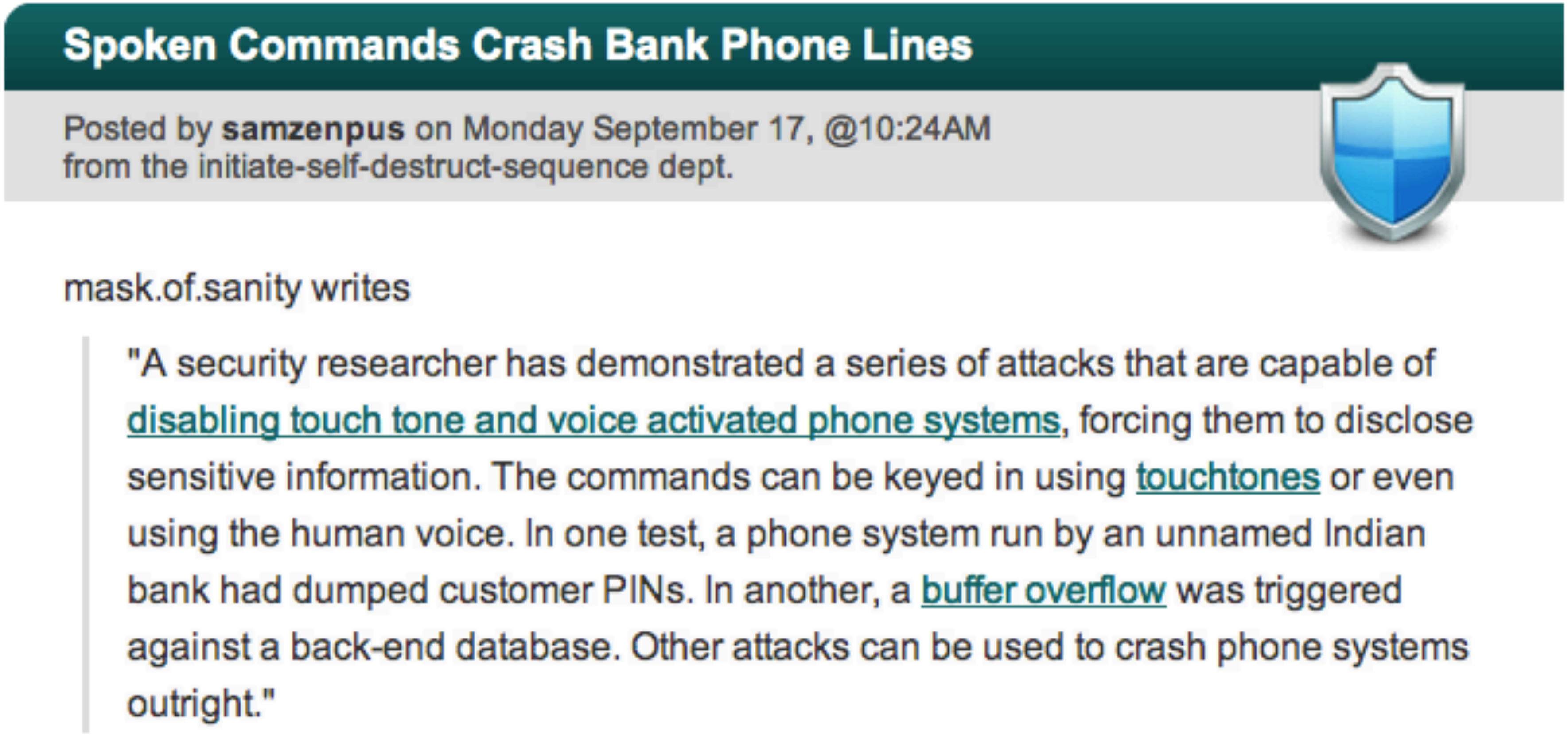


Spoken Commands Crash Bank Phone Lines

Posted by **samzenpus** on Monday September 17, @10:24AM
from the initiate-self-destruct-sequence dept.

mask.of.sanity writes

> "A security researcher has demonstrated a series of attacks that are capable of disabling touch tone and voice activated phone systems, forcing them to disclose sensitive information. The commands can be keyed in using touchtones or even using the human voice. In one test, a phone system run by an unnamed Indian bank had dumped customer PINs. In another, a buffer overflow was triggered against a back-end database. Other attacks can be used to crash phone systems outright."

Finding vulnerabilities

CS642: Computer Security

Professor Ristenpart

http://www.cs.wisc.edu/~rist/

rist at cs dot wisc dot edu

Finding vulnerabilities

Manual analysis

Simple example: double free

Fuzzing tools

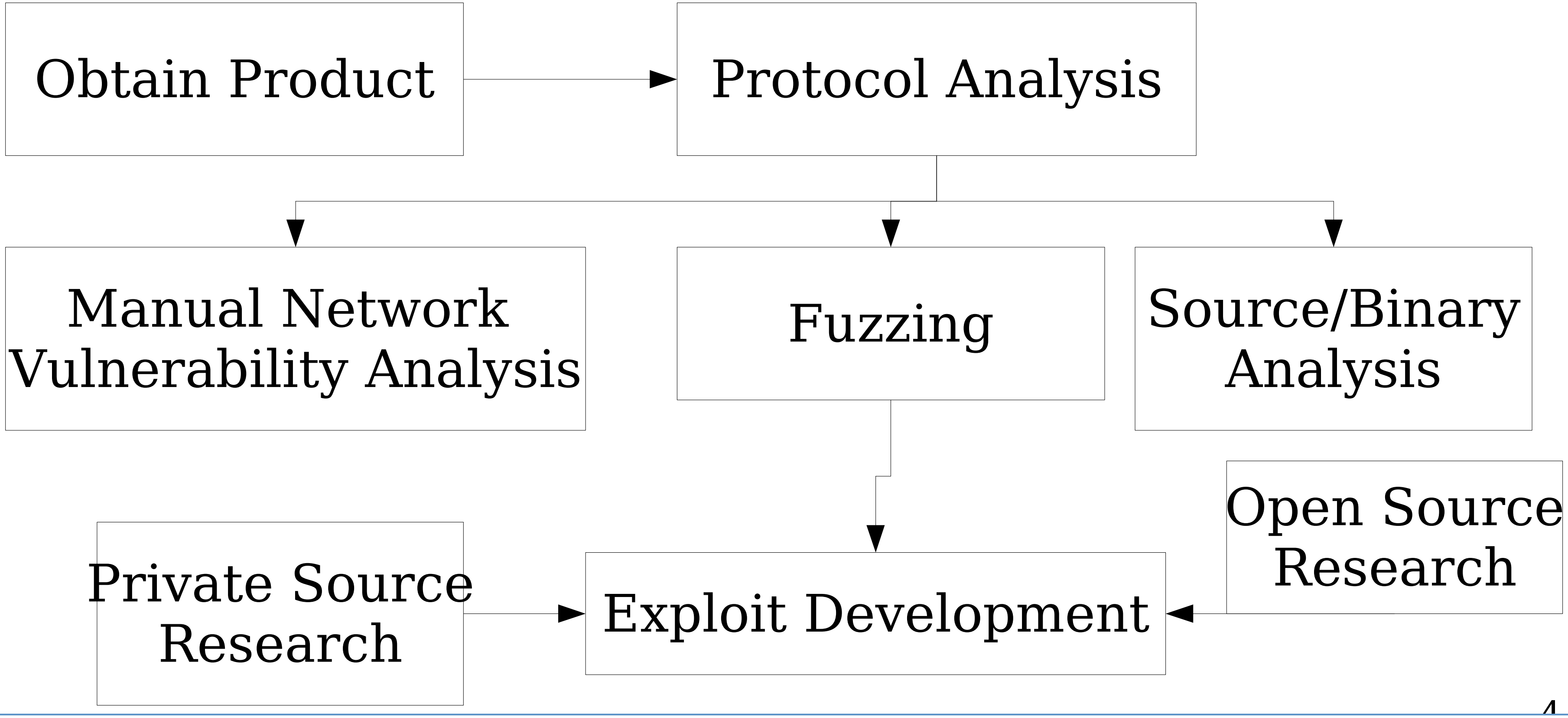
Static analysis, dynamic analysis

…

Hackers use People, Processes and Technology to obtain a singular goal: Information dominance

From “How Hackers Look for Bugs”, Dave Aitel

Take a sample product X and attack it remotely



Obtain Product

Protocol Analysis

Manual Network Vulnerability Analysis

Fuzzing

Source/Binary Analysis

Open Source Research

Private Source Research

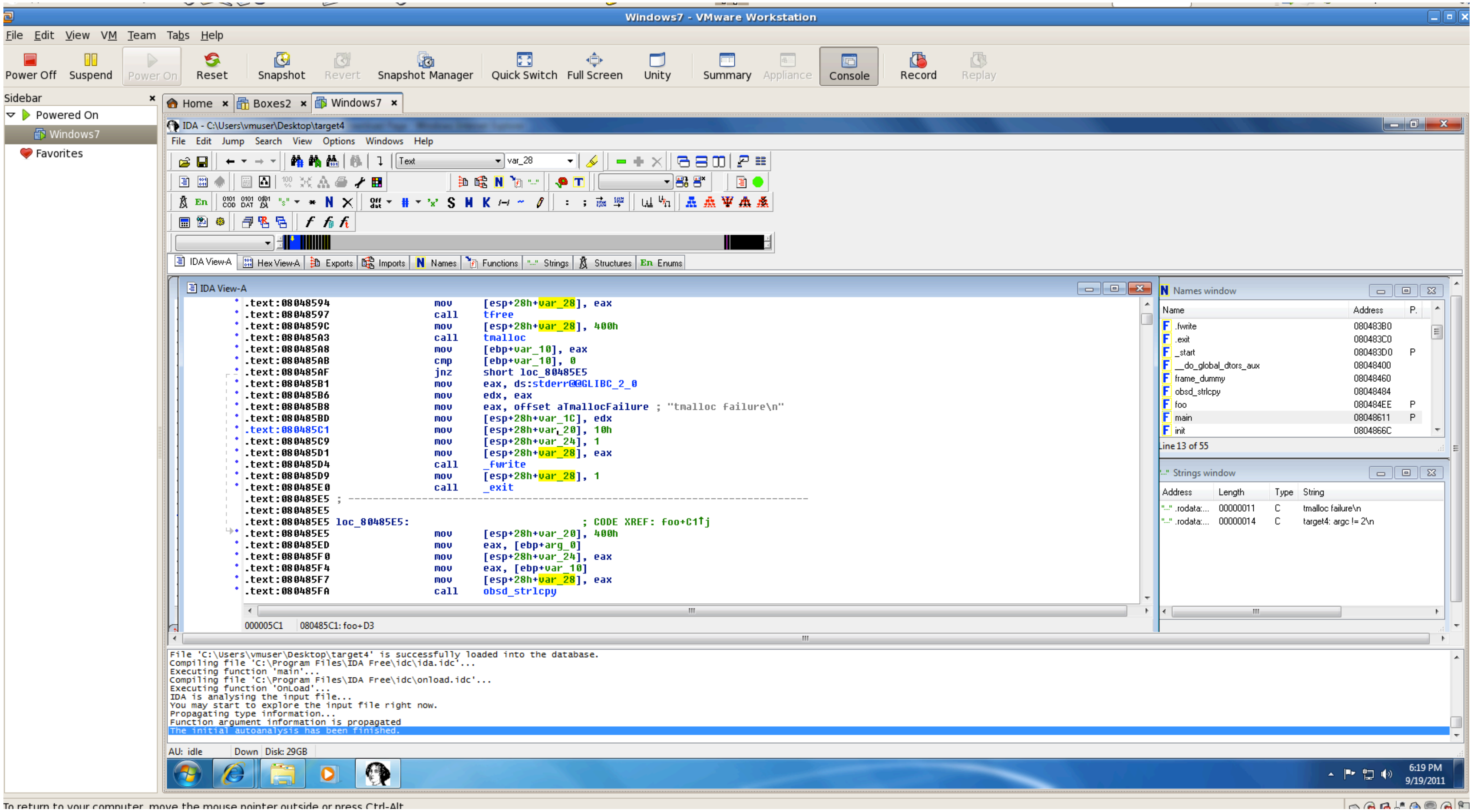
Exploit Development

From “How Hackers Look for Bugs”, Dave Aitel

Manual analysis

- You get a binary or the source code
- You find vulnerabilities

IDA Pro



```
.text:08048594                 mov     [esp+28h+var_28], eax
.text:08048597                 call    tfree
.text:0804859C                 mov     [esp+28h+var_28], 400h
.text:080485A3                 call    tmalloc
.text:080485A8                 mov     [ebp+var_10], eax
.text:080485AB                 cmp     [ebp+var_10], 0
.text:080485AF                 jnz     short loc_80485E5
.text:080485B1                 mov     eax, ds:stderr@@GLIBC_2_0
.text:080485B6                 mov     edx, eax
.text:080485B8                 mov     eax, offset aTmallocFailure ; "tmalloc failure\n"
.text:080485BD                 mov     [esp+28h+var_1C], edx
.text:080485C1                 mov     [esp+28h+var_20], 10h
.text:080485C9                 mov     [esp+28h+var_24], 1
.text:080485D1                 mov     [esp+28h+var_28], eax
.text:080485D4                 call    _fwrite
.text:080485D9                 mov     [esp+28h+var_28], 1
.text:080485E0                 call    _exit
.text:080485E5 ; ---------------------------------------------------------------------------
.text:080485E5
.text:080485E5 loc_80485E5:                            ; CODE XREF: foo+C1↑j
.text:080485E5                 mov     [esp+28h+var_20], 400h
.text:080485ED                 mov     eax, [ebp+arg_0]
.text:080485F0                 mov     [esp+28h+var_24], eax
.text:080485F4                 mov     eax, [ebp+var_10]
.text:080485F7                 mov     [esp+28h+var_28], eax
.text:080485FA                 call    obsd_strlcpy
```

000005C1 080485C1: foo+D3

Names window

Name	Address	P.
.fwrite	080483B0	
.exit	080483C0	
_start	080483D0	P
__do_global_dtors_aux	08048400	
frame_dummy	08048460	
obsd_strlcpy	08048484	
foo	080484EE	P
main	08048611	P
init	0804866C	

Line 13 of 55

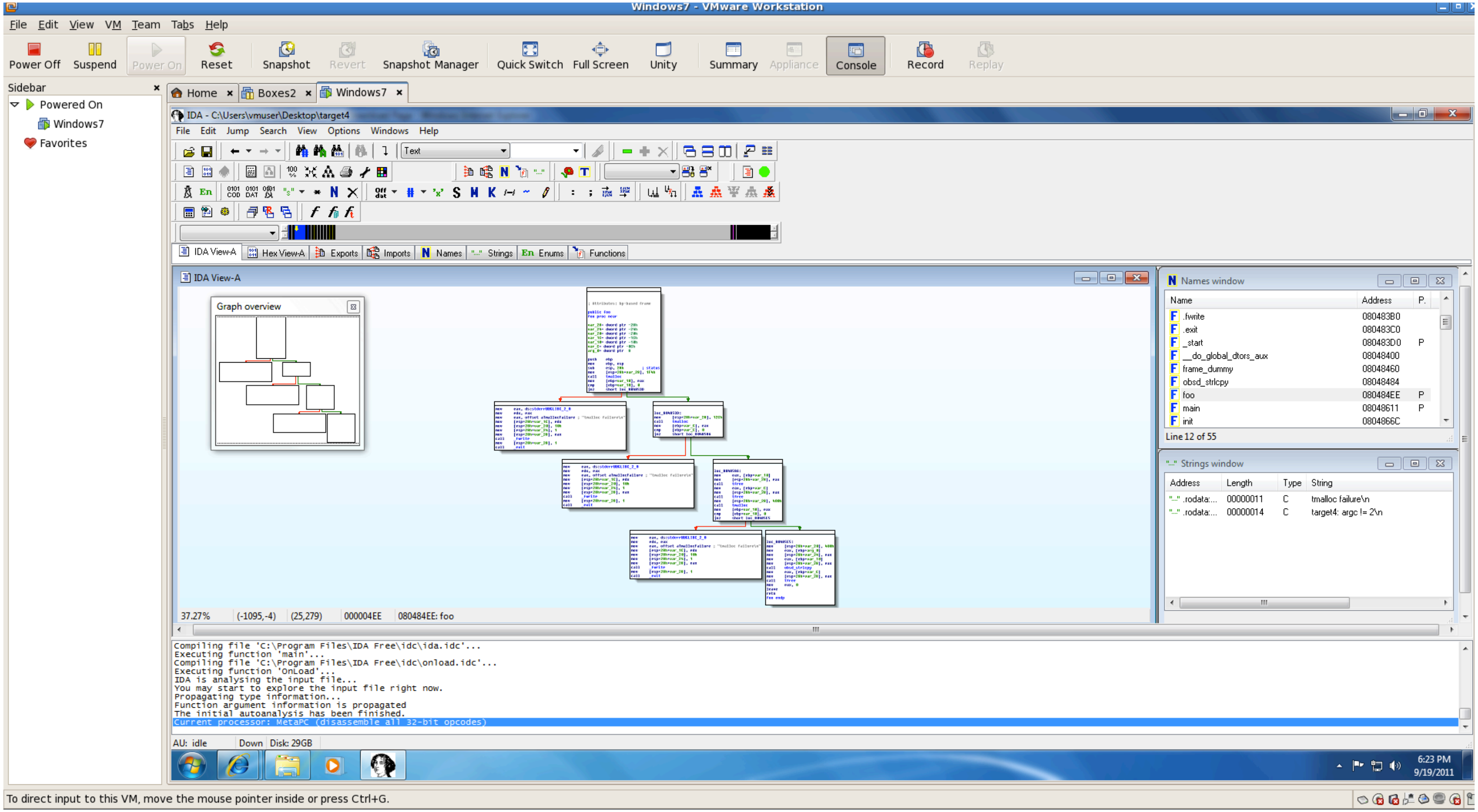
Strings window

Address	Length	Type	String
.rodata:...	00000011	C	tmalloc failure\n
.rodata:...	00000014	C	target4: argc != 2\n

```
File 'C:\Users\vmuser\Desktop\target4' is successfully loaded into the database.
Compiling file 'C:\Program Files\IDA Free\idc\ida.idc'...
Executing function 'main'...
Compiling file 'C:\Program Files\IDA Free\idc\onload.idc'...
Executing function 'OnLoad'...
IDA is analysing the input file...
You may start to explore the input file right now.
Propagating type information...
Function argument information is propagated
The initial autoanalysis has been finished.
```

AU: idle Down Disk: 29GB

IDA Pro

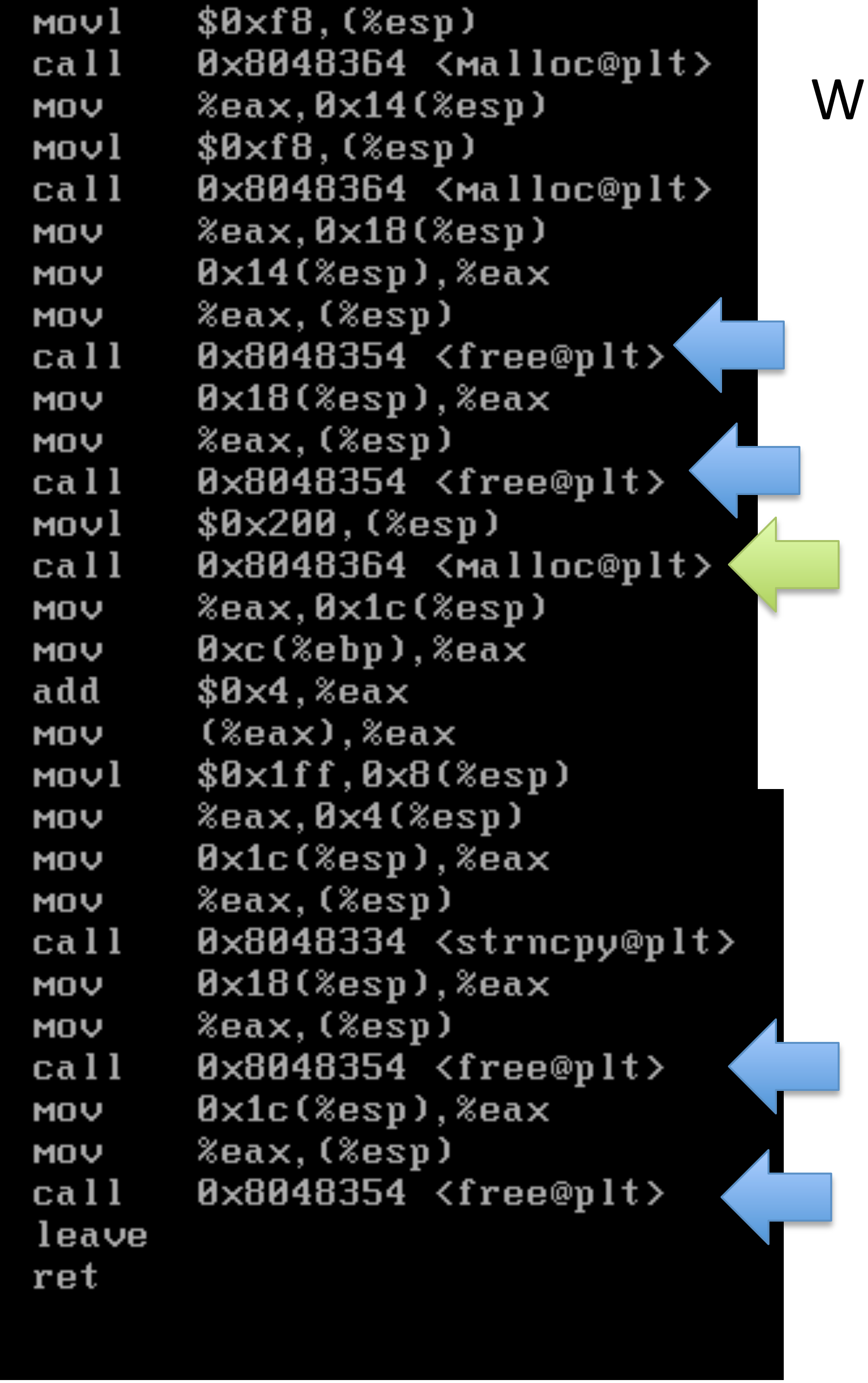


Name	Address	P.
.fwrite	080483B0	
.exit	080483C0	
_start	080483D0	P
__do_global_dtors_aux	08048400	
frame_dummy	08048460	
obsd_strlcpy	08048484	
foo	080484EE	P
main	08048611	P
init	0804866C	

Line 12 of 55

Address	Length	Type	String
.rodata:...	00000011	C	tmalloc failure\n
.rodata:...	00000014	C	target4: argc != 2\n

```
Compiling file 'C:\Program Files\IDA Free\idc\ida.idc'...
Executing function 'main'...
Compiling file 'C:\Program Files\IDA Free\idc\onload.idc'...
Executing function 'OnLoad'...
IDA is analysing the input file...
You may start to explore the input file right now.
Propagating type information...
Function argument information is propagated
The initial autoanalysis has been finished.
Current processor: MetaPC (disassemble all 32-bit opcodes)
```



```
movl   $0xf8,(%esp)
call   0x8048364 <malloc@plt>
mov    %eax,0x14(%esp)
movl   $0xf8,(%esp)
call   0x8048364 <malloc@plt>
mov    %eax,0x18(%esp)
mov    0x14(%esp),%eax
mov    %eax,(%esp)
call   0x8048354 <free@plt>
mov    0x18(%esp),%eax
mov    %eax,(%esp)
call   0x8048354 <free@plt>
movl   $0x200,(%esp)
call   0x8048364 <malloc@plt>
mov    %eax,0x1c(%esp)
mov    0xc(%ebp),%eax
add    $0x4,%eax
mov    (%eax),%eax
movl   $0x1ff,0x8(%esp)
mov    %eax,0x4(%esp)
mov    0x1c(%esp),%eax
mov    %eax,(%esp)
call   0x8048334 <strncpy@plt>
mov    0x18(%esp),%eax
mov    %eax,(%esp)
call   0x8048354 <free@plt>
mov    0x1c(%esp),%eax
mov    %eax,(%esp)
call   0x8048354 <free@plt>
leave
ret
```

What type of vulnerability might this be?

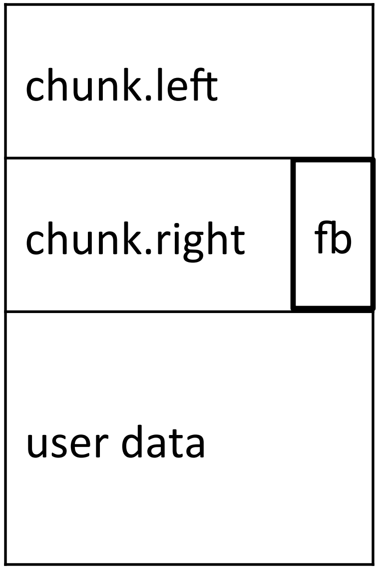
```
main(  int argc, char* argv[] ) {
 char* b1;
 char* b2;
 char* b3;

 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

Double-free vulnerabilities

Can corrupt the state of the heap management

Say we use a simple doubly-linked list malloc implementation with control information stored alongside data



chunk.left

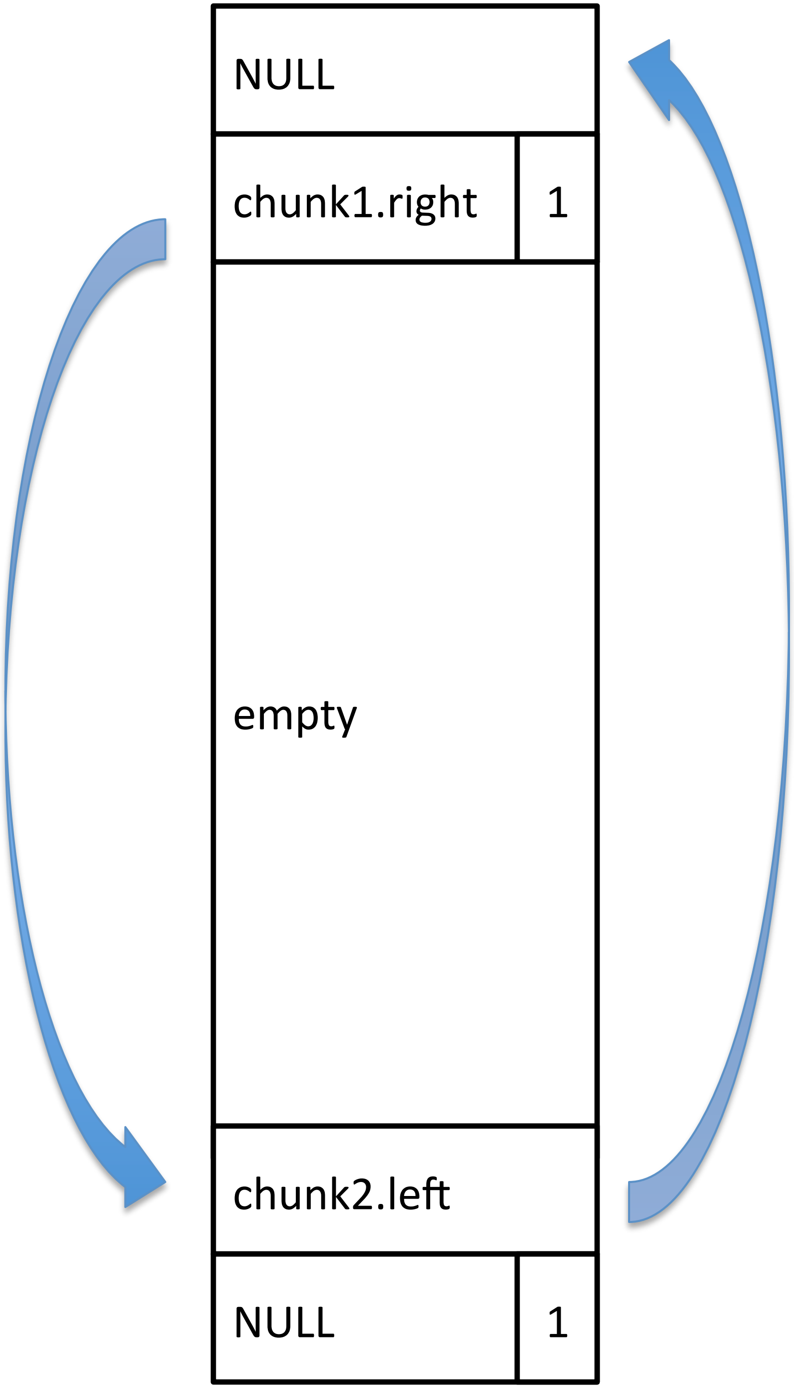
chunk.right

fb

user data

Chunk has:

1) left ptr (to previous chunk)
2) right ptr (to next chunk)
3) free bit which denotes if chunk is free
   this reuses low bit of right ptr
   because we will align chunks
4) user data



NULL

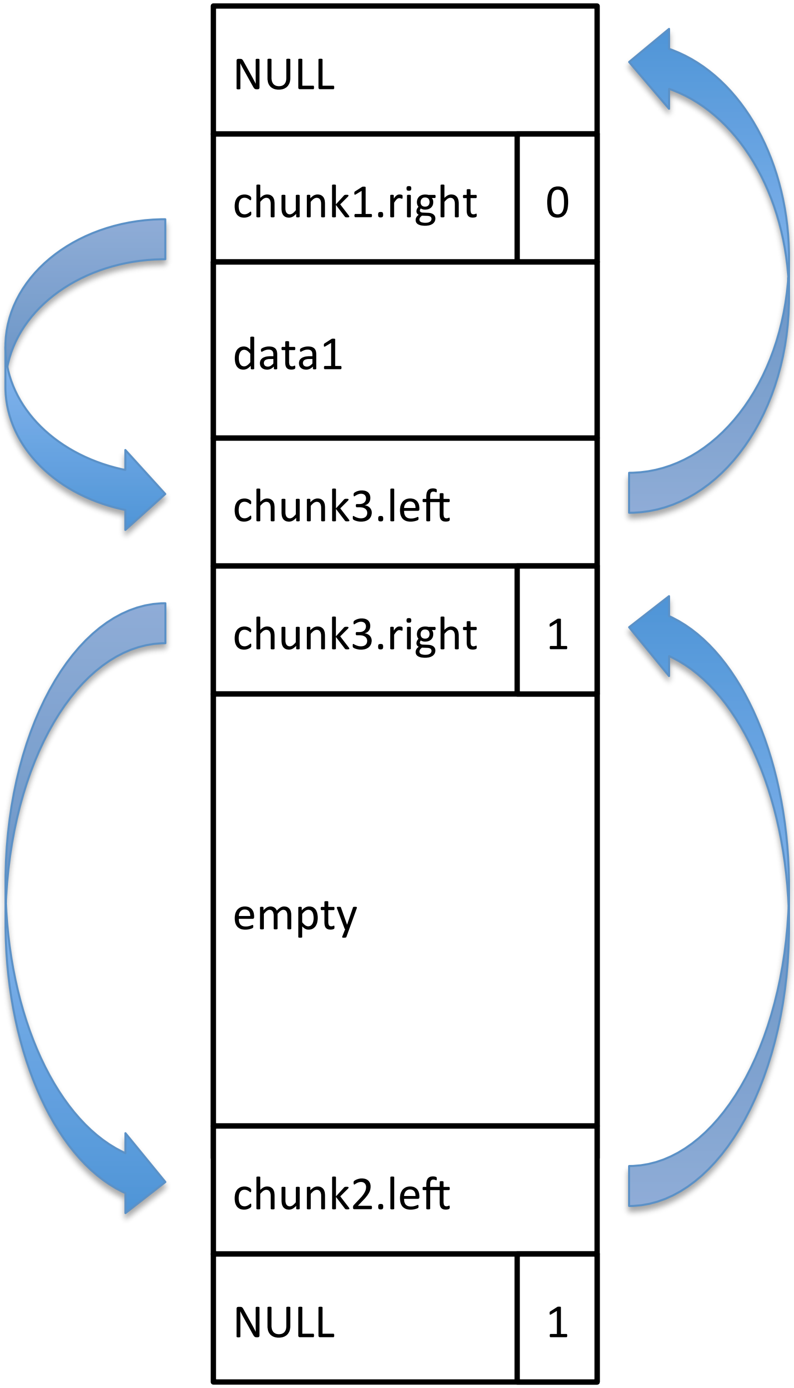
chunk1.right 1

empty

chunk2.left

NULL 1

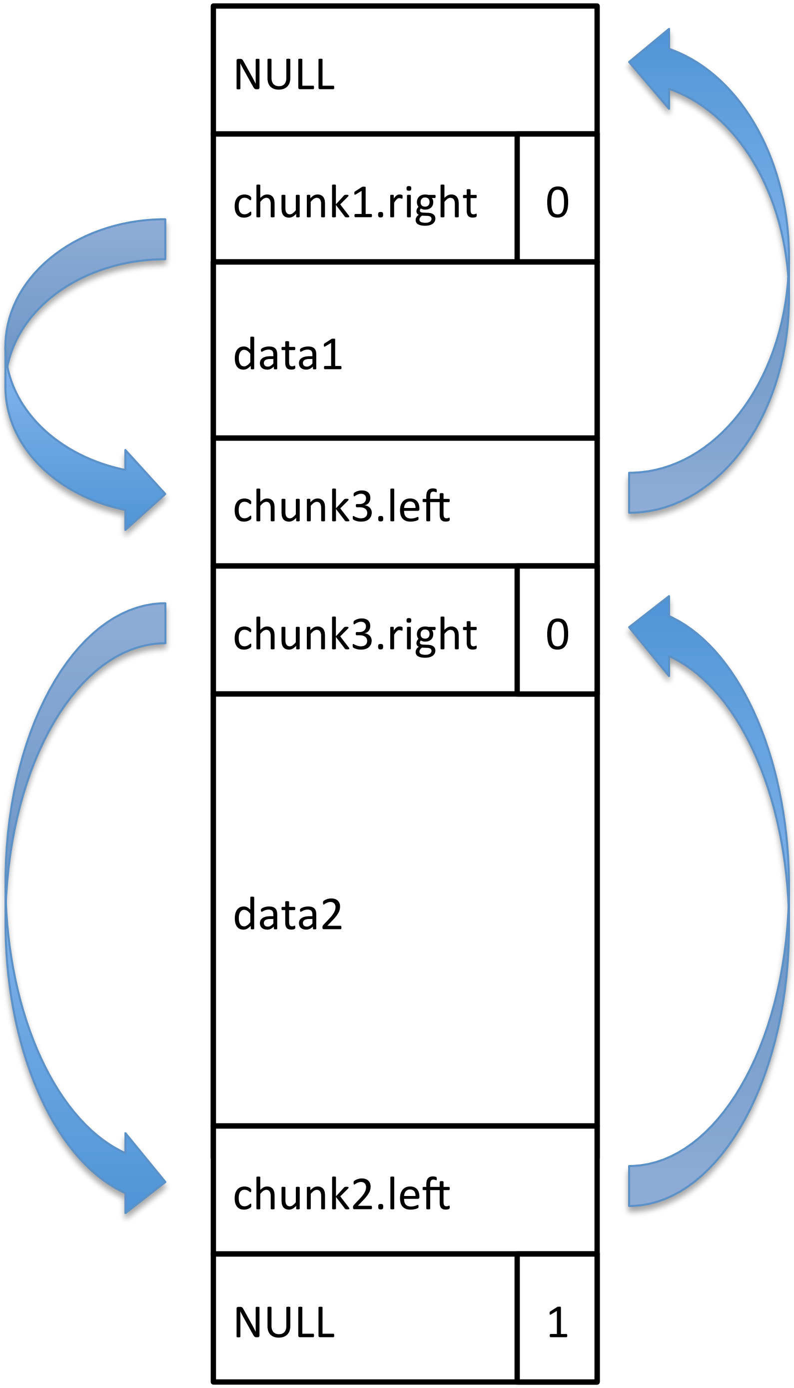
malloc()
- search left-to-right for free chunk
- modify pointers



NULL	
chunk1.right	0
data1	
chunk3.left	
chunk3.right	1
empty	
chunk2.left	
NULL	1

malloc()
- search left-to-right for free chunk
- modify pointers

b1 = malloc(BUF_SIZE1);



NULL

chunk1.right 0

data1

chunk3.left

chunk3.right 0

data2

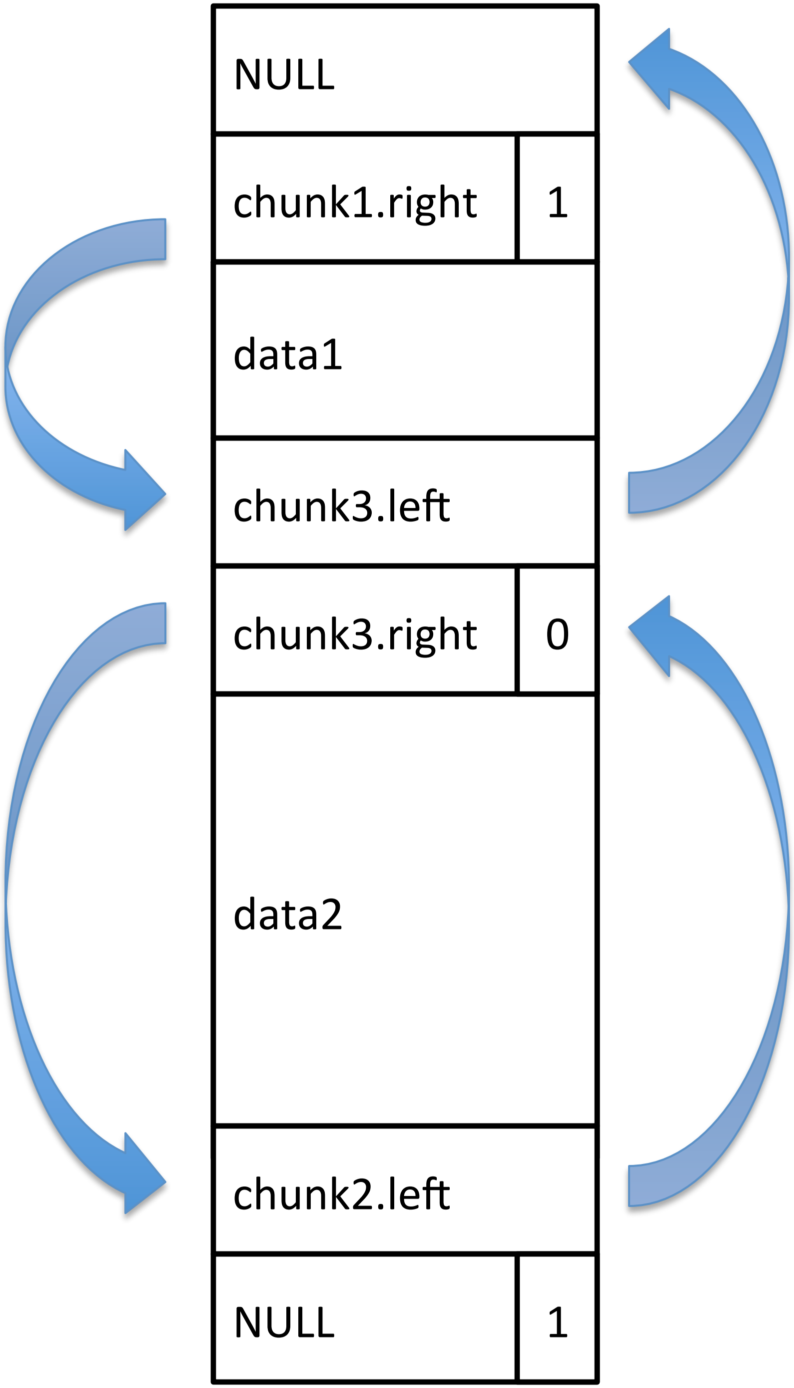
chunk2.left

NULL 1

malloc()
- search left-to-right for free chunk
- modify pointers

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

free()
- Consolidate with free neighbors



NULL

chunk1.right | 1

data1

chunk3.left

chunk3.right | 0

data2

chunk2.left

NULL | 1

malloc()

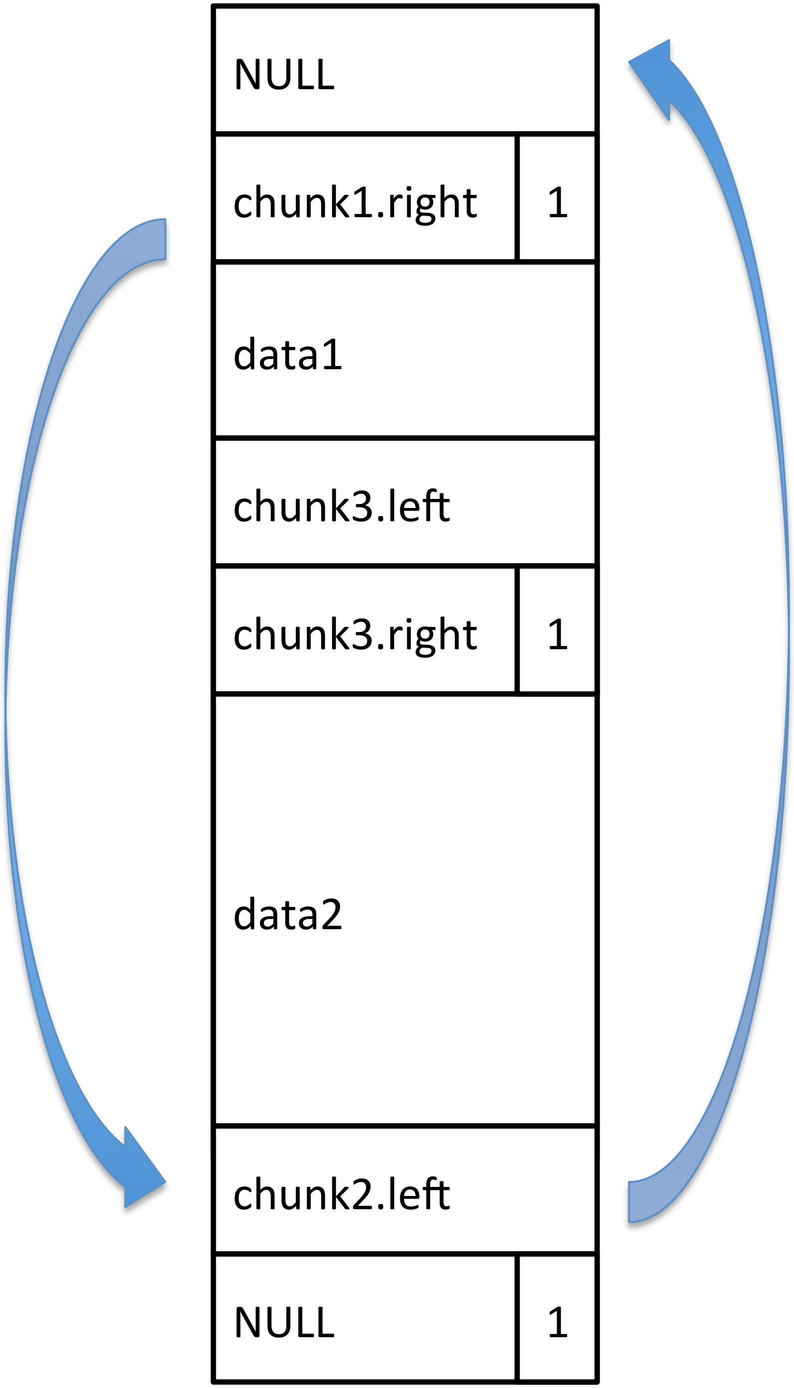
- search left-to-right for free chunk
- modify pointers

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

free()

- Consolidate with free neighbors

free(b1)



NULL

chunk1.right | 1

data1

chunk3.left

chunk3.right | 1

data2

chunk2.left

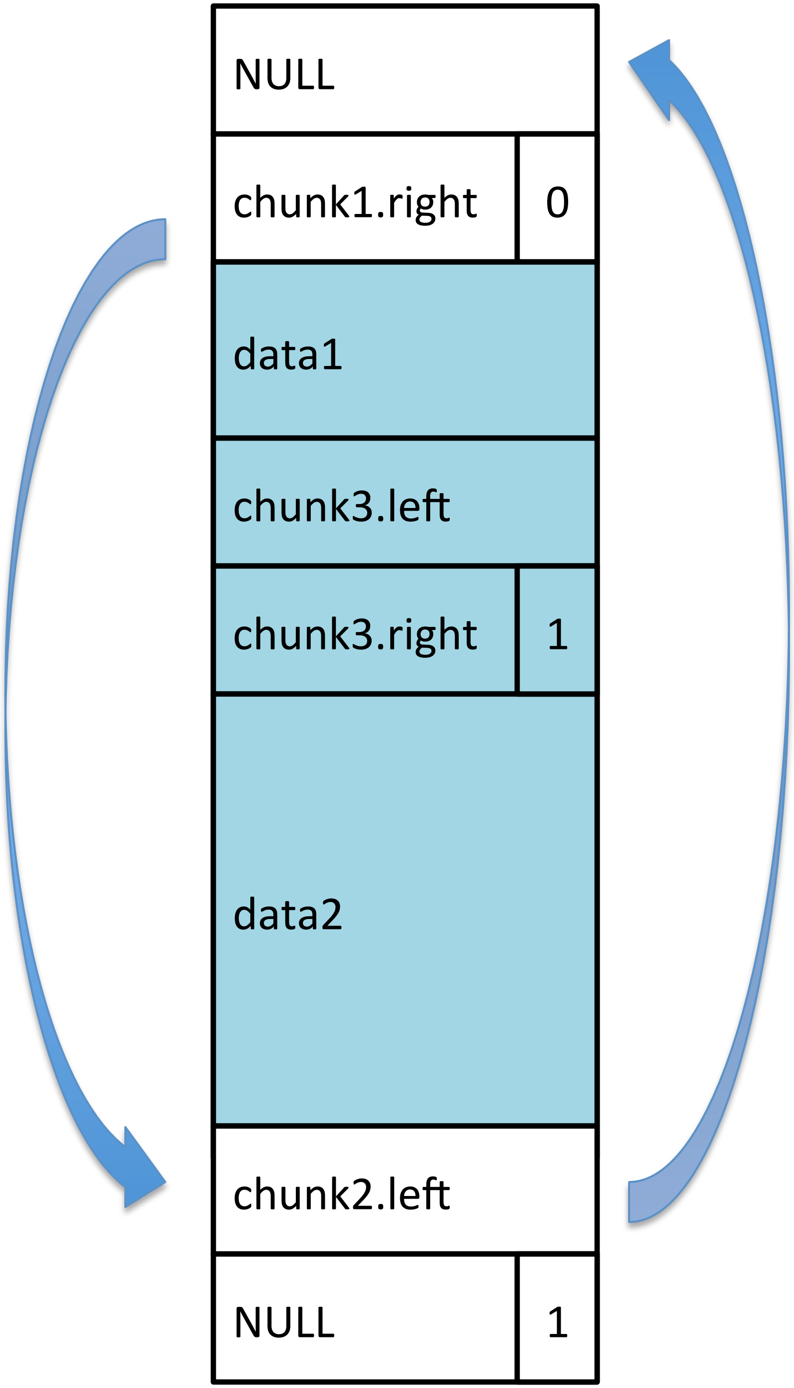
NULL | 1

malloc()
- search left-to-right for free chunk
- modify pointers

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

free()
- Consolidate with free neighbors

free(b1)
free(b2)

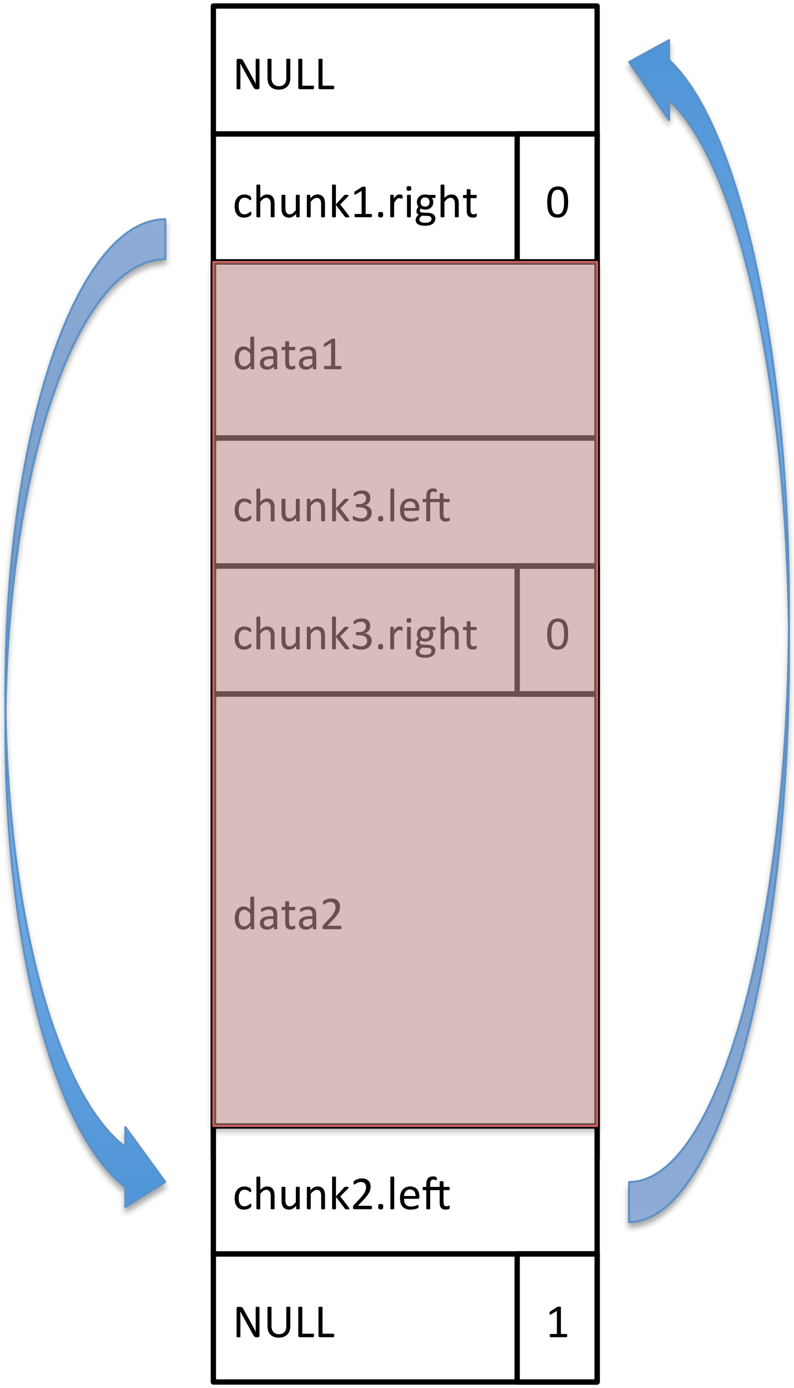

NULL
chunk1.right | 0
data1
chunk3.left
chunk3.right | 1
data2
chunk2.left
NULL | 1

malloc()
- search left-to-right for free chunk
- modify pointers

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

free()
- Consolidate with free neighbors

free(b1)
free(b2)
b3 = malloc(BUF_SIZE1 + BUF_SIZE2)



NULL

chunk1.right | 0

data1

chunk3.left

chunk3.right | 0

data2

chunk2.left

NULL | 1

malloc()

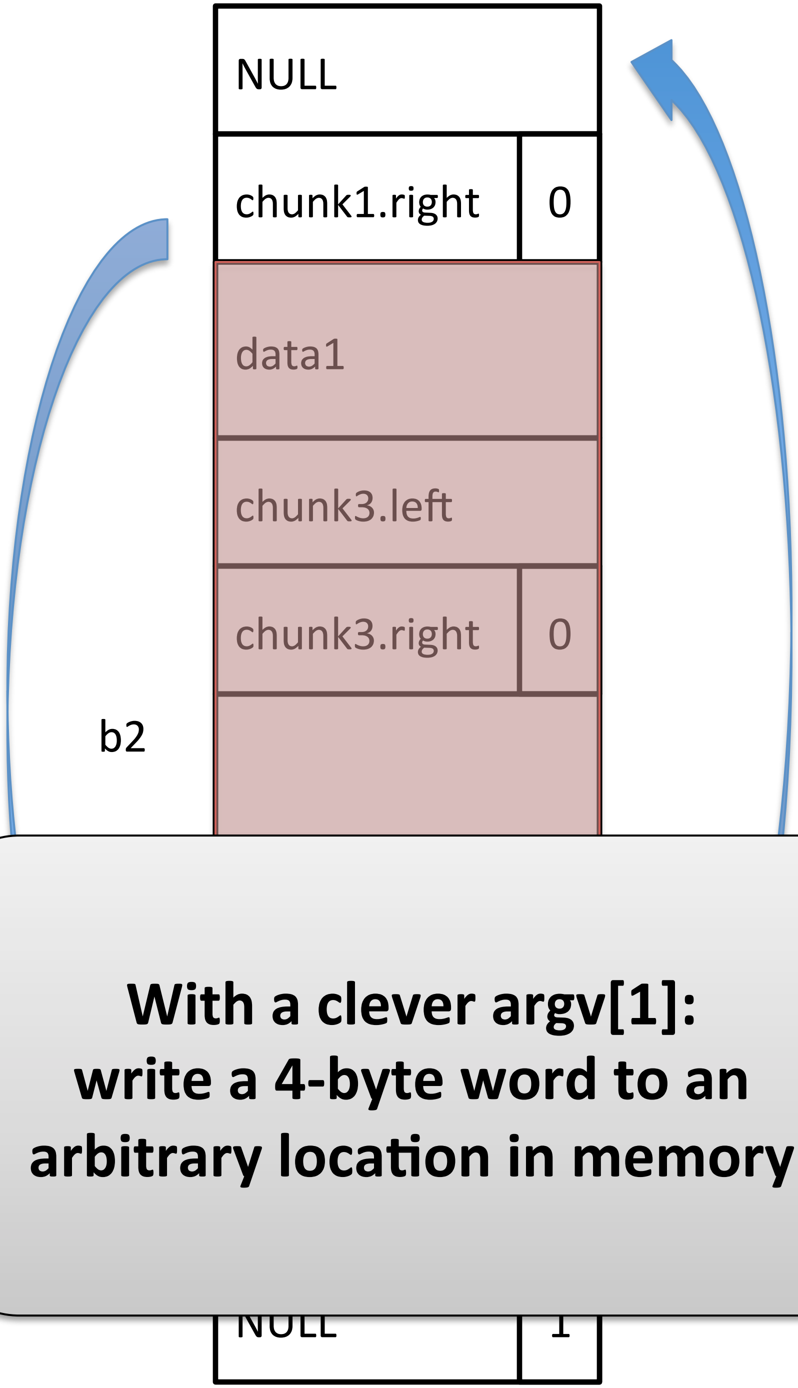
- search left-to-right for free chunk
- modify pointers

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

free()

- Consolidate with free neighbors

```
free( b1 )
free( b2 )
b3 = malloc( BUF_SIZE1 + BUF_SIZE2 )
strncpy( b3, argv[1], BUF_SIZE1+BUF_SIZE2-1 )
```



malloc()
- search left-to-right for free chunk
- modify pointers

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

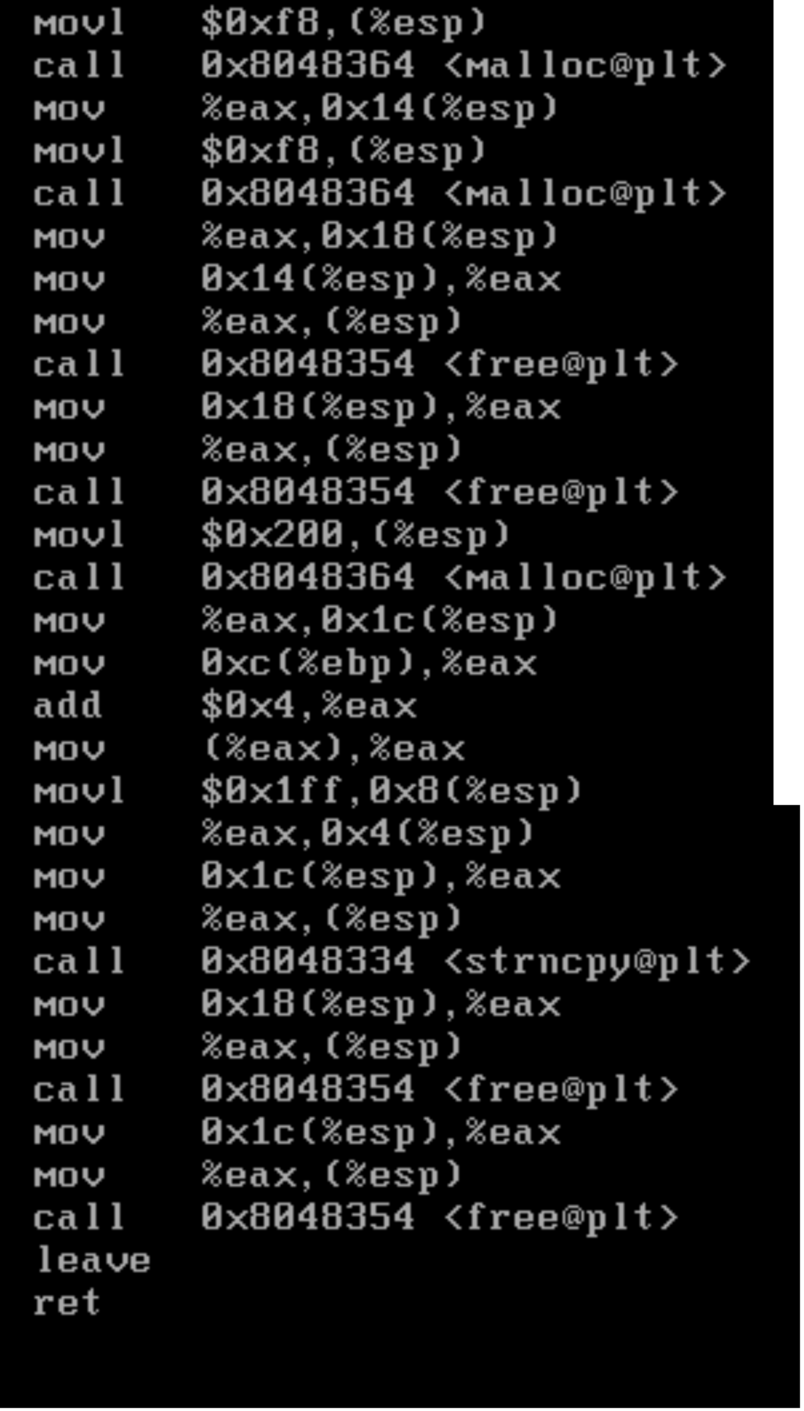
free()
- Consolidate with free neighbors

```
free( b1 )
free( b2 )
b3 = malloc( BUF_SIZE1 + BUF_SIZE2 )
strncpy( b3, argv[1], BUF_SIZE1+BUF_SIZE2-1 )
free( b2 )
```

Interprets b2-8 as a chunk3.left
Interprets b2-4 as a chunk3.right

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

**With a clever argv[1]:
write a 4-byte word to an
arbitrary location in memory**



```
movl   $0xf8,(%esp)
call   0x8048364 <malloc@plt>
mov    %eax,0x14(%esp)
movl   $0xf8,(%esp)
call   0x8048364 <malloc@plt>
mov    %eax,0x18(%esp)
mov    0x14(%esp),%eax
mov    %eax,(%esp)
call   0x8048354 <free@plt>
mov    0x18(%esp),%eax
mov    %eax,(%esp)
call   0x8048354 <free@plt>
movl   $0x200,(%esp)
call   0x8048364 <malloc@plt>
mov    %eax,0x1c(%esp)
mov    0xc(%ebp),%eax
add    $0x4,%eax
mov    (%eax),%eax
movl   $0x1ff,0x8(%esp)
mov    %eax,0x4(%esp)
mov    0x1c(%esp),%eax
mov    %eax,(%esp)
call   0x8048334 <strncpy@plt>
mov    0x18(%esp),%eax
mov    %eax,(%esp)
call   0x8048354 <free@plt>
mov    0x1c(%esp),%eax
mov    %eax,(%esp)
call   0x8048354 <free@plt>
leave
ret
```

What type of vulnerability might this be?

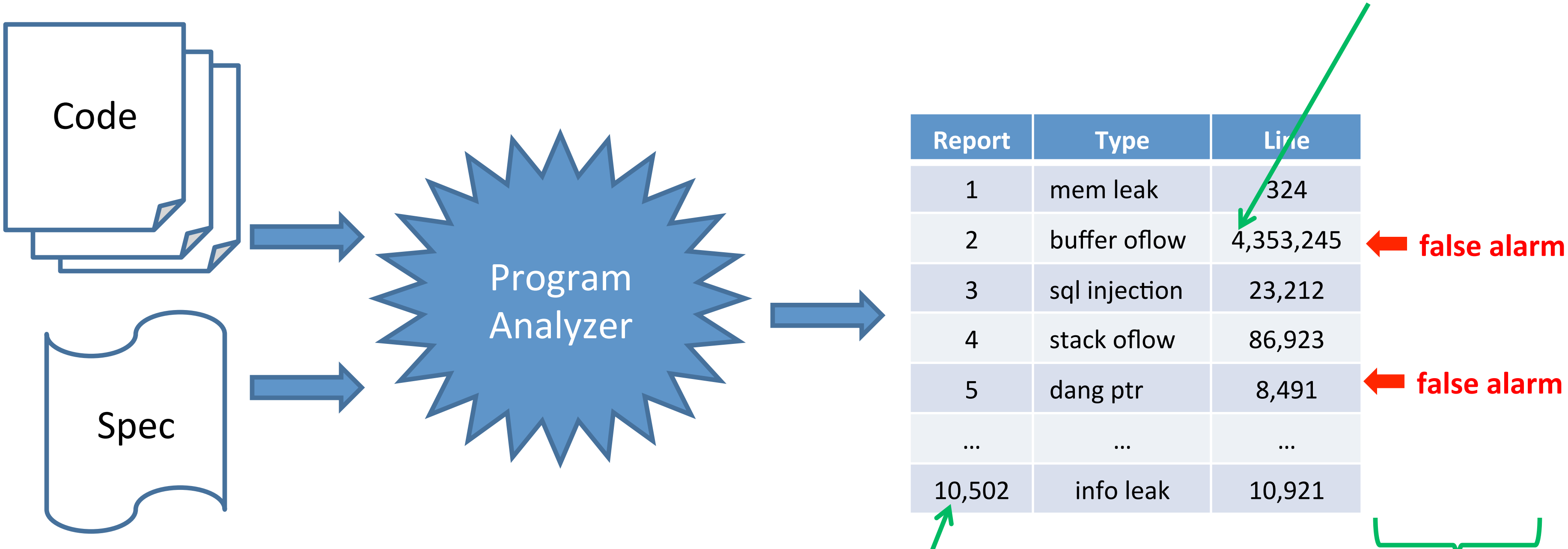
This is very simple example. Manual analysis is very time consuming.

Security analysts use a variety of tools to augment manual analysis

Security tools

- Testing tools for helping find security-critical bugs
  - Scanners
  - Taint trackers
  - Fuzzers
    - “dumb”, “smart”, whitebox
  - Static analysis tools
- Static analysis vs. dynamic analysis

Program analyzers


Code

Spec

Program Analyzer

Report	Type	Line
1	mem leak	324
2	buffer oflow	4,353,245
3	sql injection	23,212
4	stack oflow	86,923
5	dang ptr	8,491
…	…	…
10,502	info leak	10,921

false alarm

false alarm

analyze large code bases

potentially reports many warnings

may emit false alarms

Slide credit: Prof Mitchell Stanford’s CS 155

Program analyzers

- Static analysis
  - Do not execute program

- Dynamic analysis
  - Execute program on test cases

Soundness, Completeness

Property	**Definition**
Soundness	If the program contains an error, the analysis will report a warning. “Sound for reporting correctness”
Completeness	If the analysis reports an error, the program will contain an error. “Complete for reporting correctness”

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	Reports all errors May report false alarms No false negatives False positives Decidable
Unsound	May not report all errors Reports no false alarms False positives No false negatives Decidable	May not report all errors May report false alarms False negatives False positives Decidable

Slide credit: Prof Mitchell Stanford’s CS 155

Source code scanners

Look at source code, flag suspicious constructs

```
…
strcpy( ptr1, ptr2 );
…
```

Warning: Don't use strcpy

Simplest example: grep

Lint is early example

RATS (Rough auditing tool for security)

ITS4 (It's the Software Stupid Security Scanner)

Circa 1990's technology:

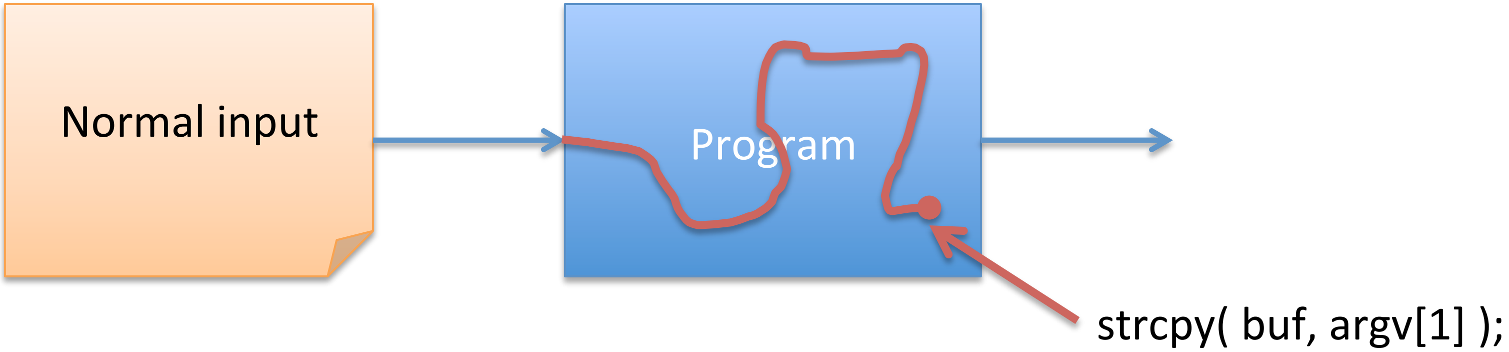
shouldn't work for reasonable modern codebases

Taint tracking

Track information flow from user input to it’s use

Can be either static or dynamic

Useful to augment manual testing

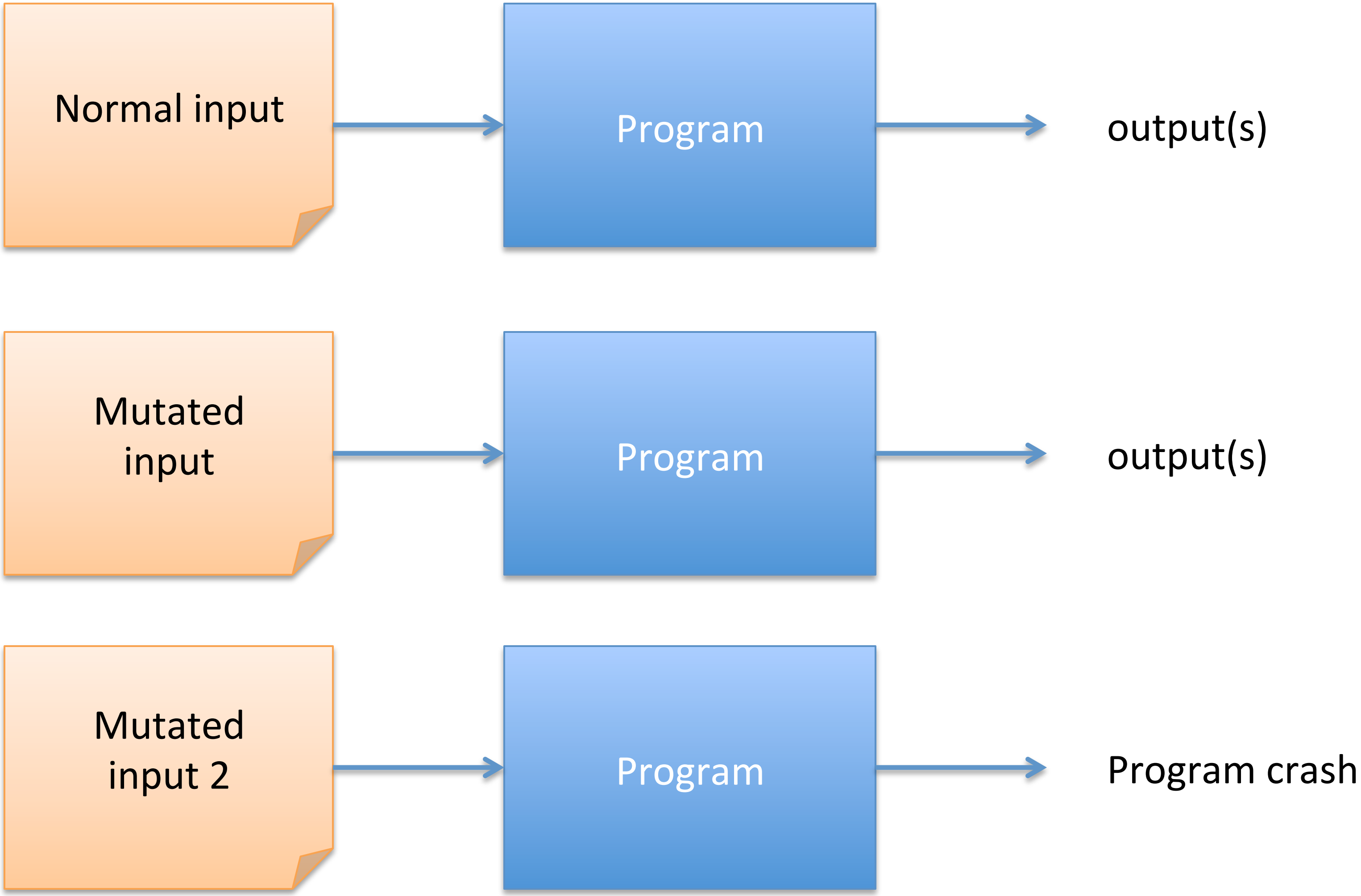


Fuzzing

“The term first originates from a class project at the University of Wisconsin 1988 although similar techniques have been used in the field of quality assurance, where they are referred to as robustness testing, syntax testing or negative testing.”
Wikipedia
http://en.wikipedia.org/wiki/Fuzz_testing

Choose a bunch of inputs
See if they cause program to misbehave
Example of dynamic analysis

Black-box fuzz testing



Normal input

Program

output(s)

Mutated input

Program

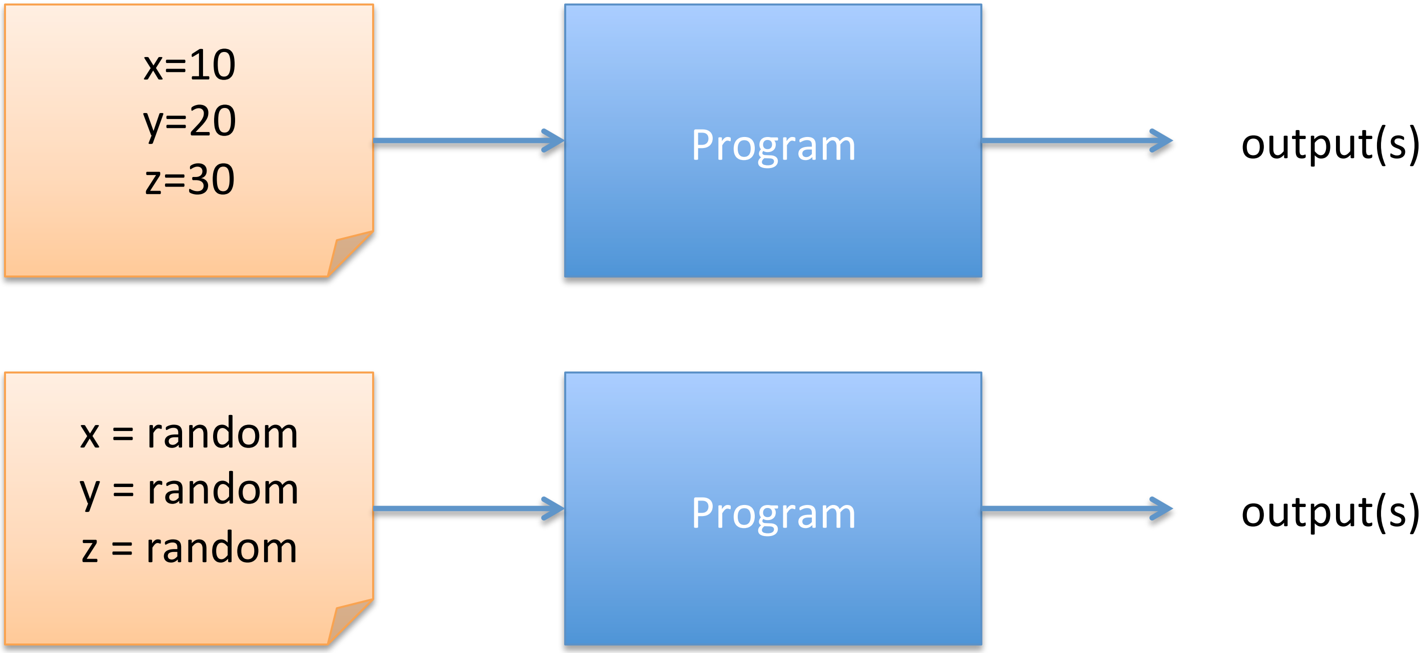
output(s)

Mutated input 2

Program

Program crash

Black-box fuzz testing



x=10
y=20
z=30

Program

output(s)

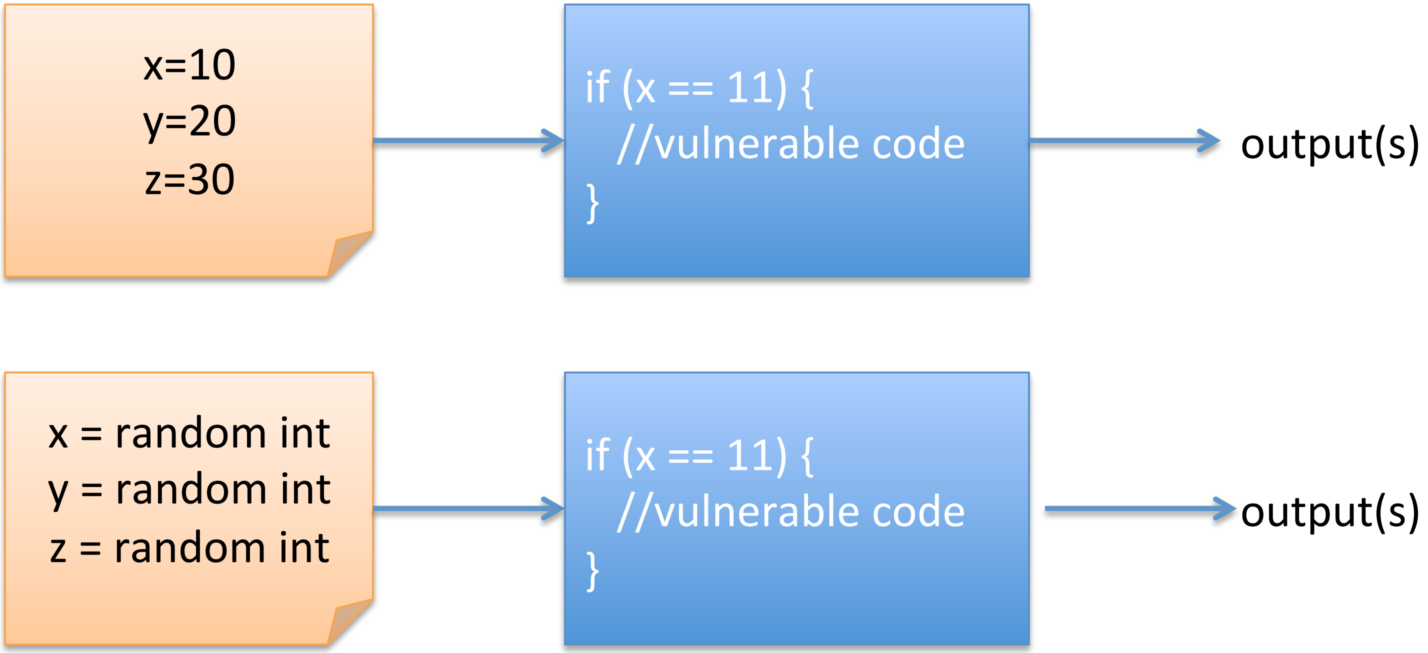
x = random
y = random
z = random

Program

output(s)

Achieving code coverage can be very difficult

Black-box fuzz testing



x=10
y=20
z=30

```
if (x == 11) {
  //vulnerable code
}
```

output(s)

x = random int
y = random int
z = random int

```
if (x == 11) {
  //vulnerable code
}
```

output(s)

Achieving code coverage can be very difficult

If x is 32 bits, then probability of crashing is **at most**?

Fuzzing is a lot about code coverage

- Code coverage defined in many ways
  - # of basic blocks reached
  - # of paths followed
  - # of conditionals followed
- Mutation based
  - Start with known-good examples
  - Mutate them to new test cases
    - heuristics: increase string lengths (AAAAAAAAA…)
    - randomly change items
- Generative
  - Start with specification of protocol, file format
  - Build test case files from it
    - Rarely used parts of spec

Example from Miller slides

Multiplayer game
Fuzz for remote exploits

- Capture packets during normal use
- Replace some packet contents with random values
- Send to game, determine code coverage

Initial: 614 out of 36183 basic blocks

From Wikipedia:

Freeciv

Freeciv 2.1.0-beta3, with the SDL client

One big switch statement controlled by third byte of packet
Update fuzz rules to exhaust the values of this third byte

Improves coverage by 4x.
Repeat several times to improve coverage.
Heap overflow found.

Symbolic execution

```
void myfree(int* p, int x) {
	int y = 0;

	free( p );
	y = 10*x;
	if( y > 20 )  free( p );
}
```

Let ***x'*** be symbolic variable for x

y' = 2***x'*** is new symbolic variable

y' > 20 is constraint on reaching second free()

2***x'*** > 20 must hold for input x to cause double free

- Technique for statically analyzing code paths and finding inputs
- Associate to each input variable a special symbol
  - called symbolic variable
- Simulate execution symbolically
  - Update symbolic variable's value appropriately
  - Conditionals add constraints on possible values
- Cast constraints as satisfiability, and use SAT solver to find inputs

White-box fuzz testing

- Start with real input and do static analysis
  - Symbolic execution of program
  - Gather constraints (control flow) along way
  - Systematically negate constraints backwards
  - Eventually this yields a new input
- Repeat

Godefroid, Levin, Molnar. "Automated Whitebox Fuzz Testing"

Symbolic execution + fuzzing

```
void top(char input[4]) {
  int cnt=0;
  if (input[0] == ’b’) cnt++;
  if (input[1] == ’a’) cnt++;
  if (input[2] == ’d’) cnt++;
  if (input[3] == ’!’) cnt++;
  if (cnt >= 3) abort(); // error
}
```

Example from Godefroid et al.

Start with some input.
Run program for real & symbolicly
Say input = “good”

i0 != ‘b’
i1 != ‘a’
i2 != ‘d’
i3 != ‘!’

i0,i1,i2,i3 are all symbolic variables

This gives set of constraints on input
Negate them one at a time to generate a new input that explores new path

Example
i0 != ‘b’ and i1 != ‘a’ and i2 != ‘d’ and i3 = ‘!’
input would be ``goo!’’

Repeat with new input

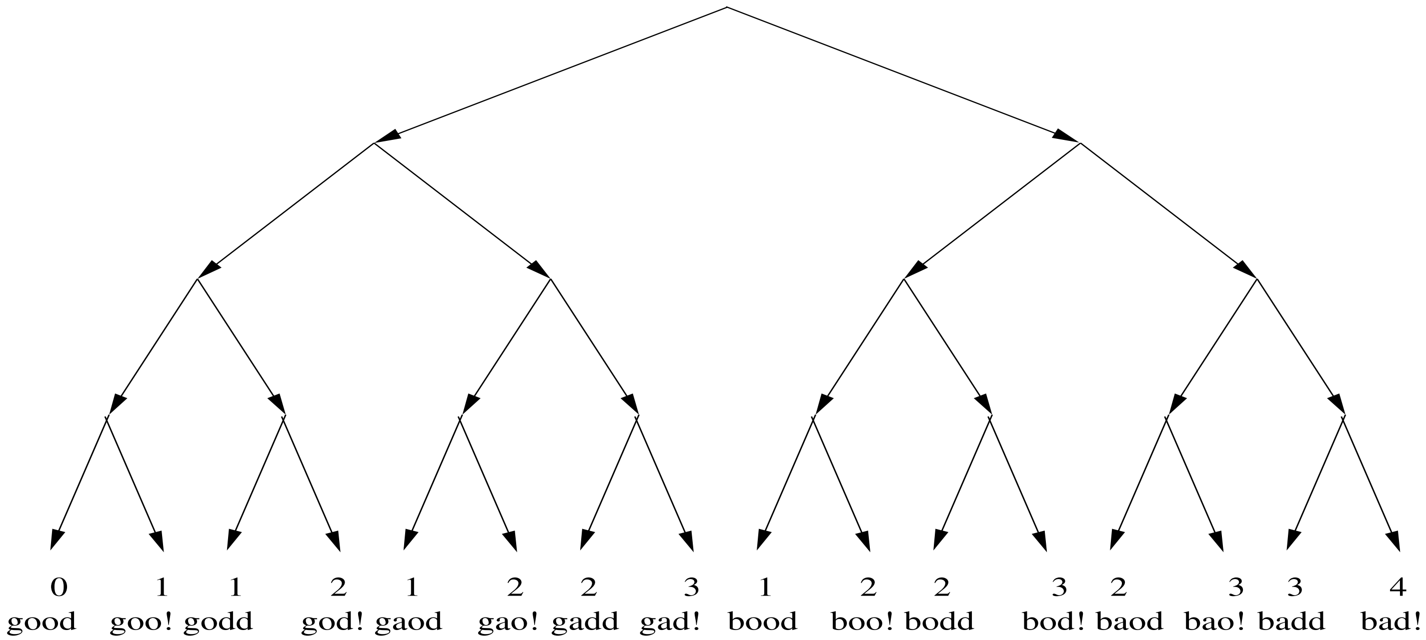


Figure 2. Search space for the example of Figure 1 with the value of the variable `cnt` at the end of each run and the corresponding input string.

Example from Godefroid et al.

Larger programs have too many paths to explore so they specify various heuristics

In-use at Microsoft

Formal verification

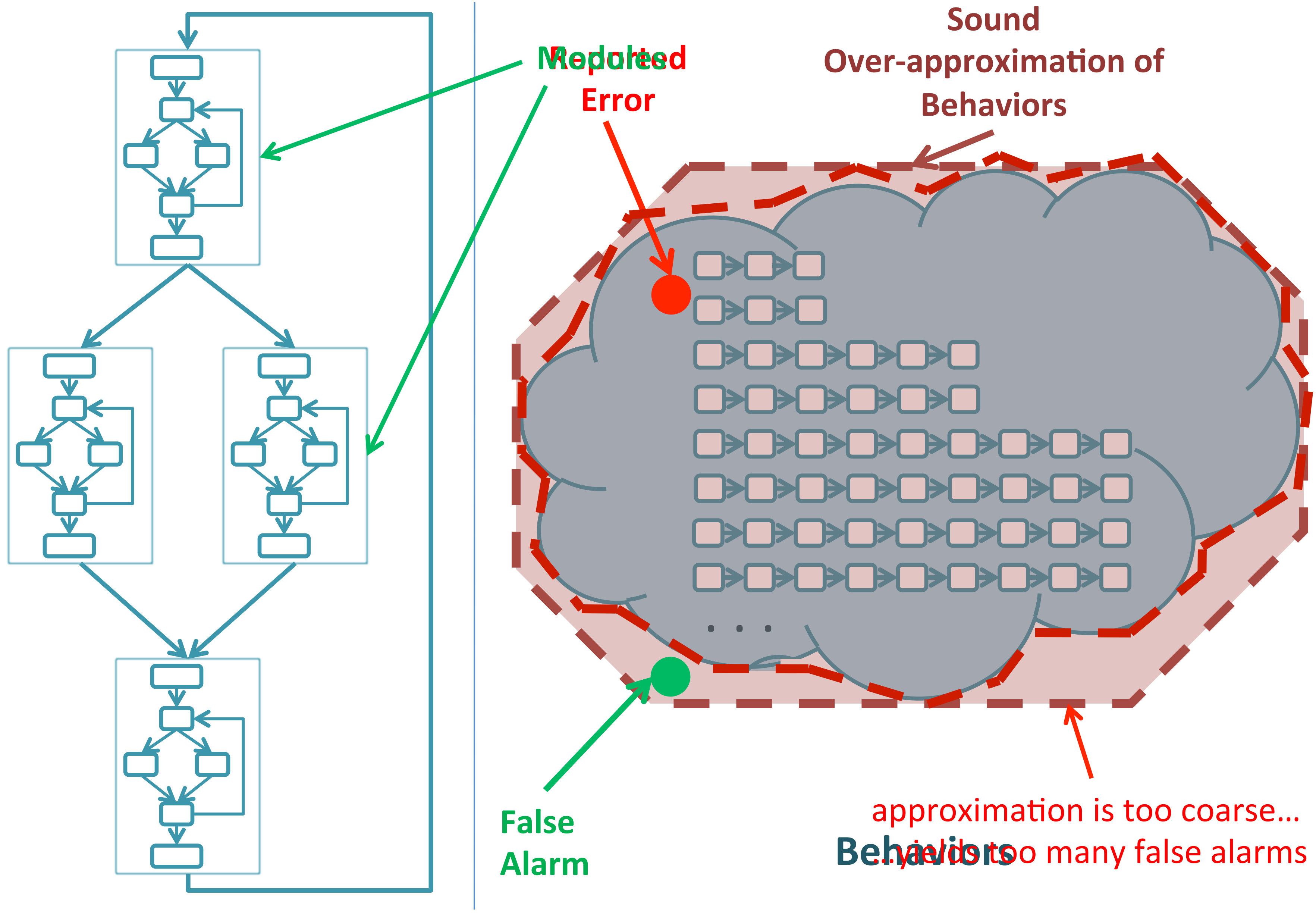
- Can we prove absence of security vulnerabilities?
- No…but we might rule out certain specific vulnerabilities

Model checking

1) Specify a property (typically via FSA)
    e.g., temporal safety property
    such as "drop privileges" before
    running untrusted program
2) Prove that no path exists that violates
    property (on some sound abstraction)

Bug finding is a big business

- Grammatech (Prof Reps here at Wisconsin)
- Coverity (Stanford startup)
- Fortify
- many, many others…



Modules

Reported Error

Sound Over-approximation of Behaviors

False Alarm

approximation is too coarse…

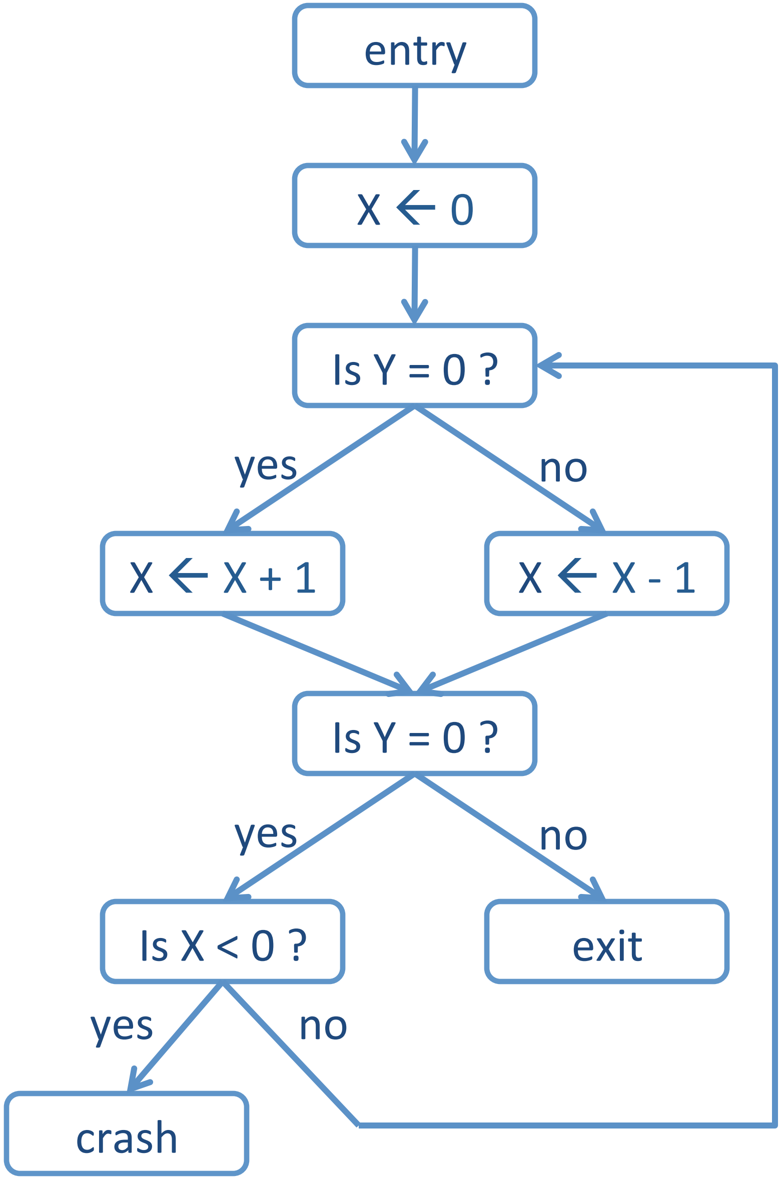
…yields too many false alarms

Behaviors

Software

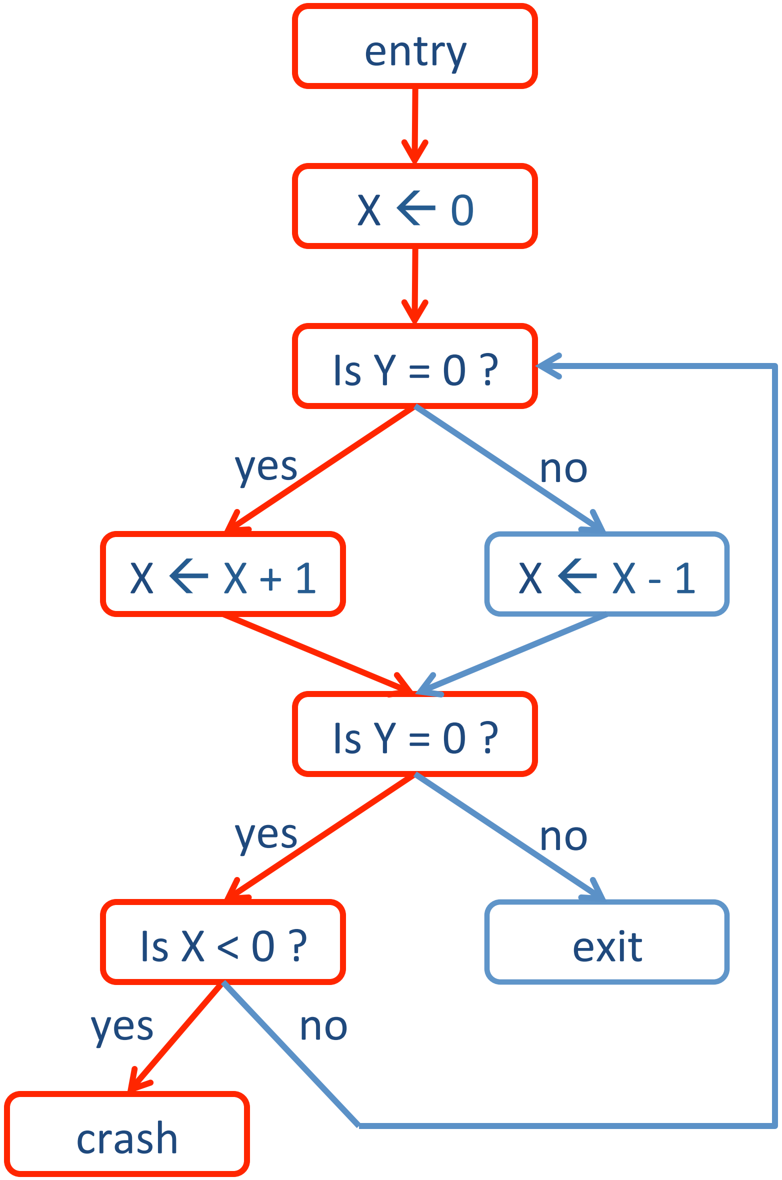
Slide credit: Prof Mitchell Stanford’s CS 155

Does this program ever crash?



Slide credit: Prof Mitchell Stanford's CS 155

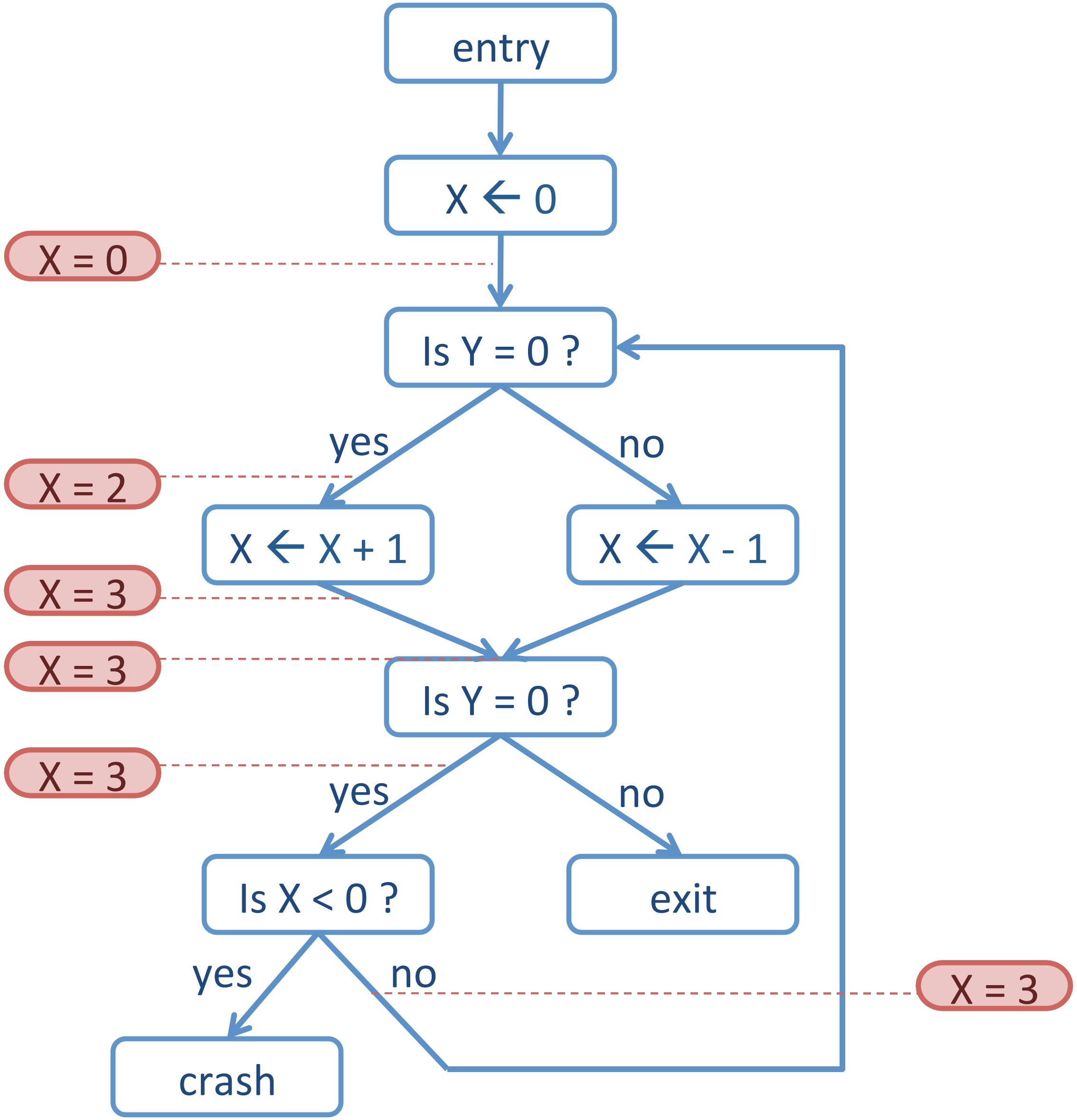
Does this program ever crash?



infeasible path!
… program will never crash

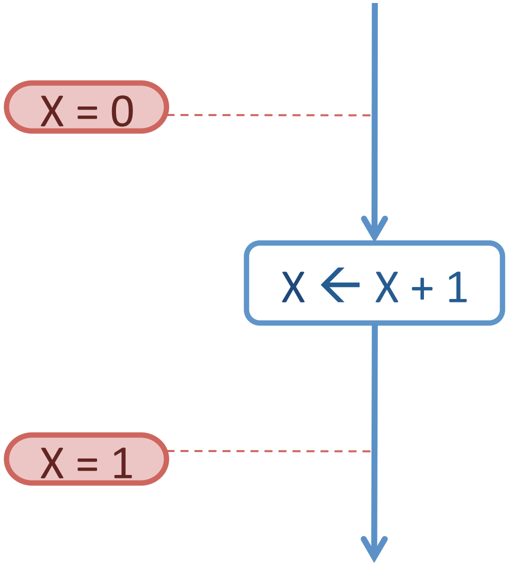
Slide credit: Prof Mitchell Stanford’s CS 155

Try analyzing without approximating…



non-termination!
… therefore, need to approximate

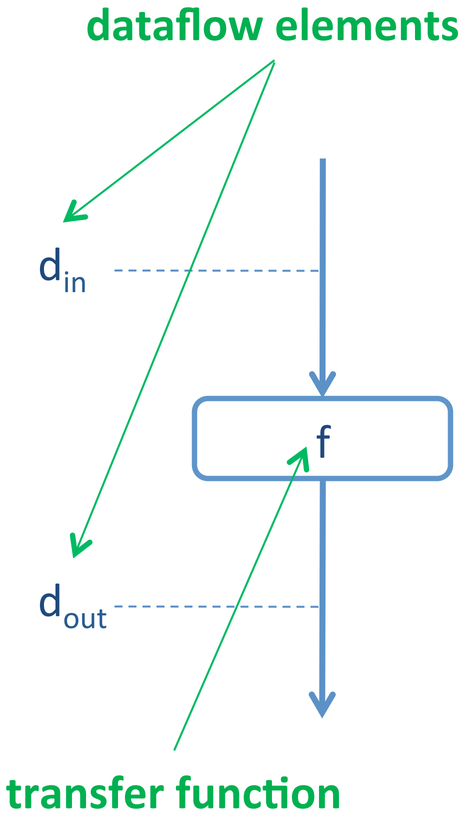
Slide credit: Prof Mitchell Stanford's CS 155



$X = 0$

$X \leftarrow X + 1$

$X = 1$



dataflow elements

d_{in}

f

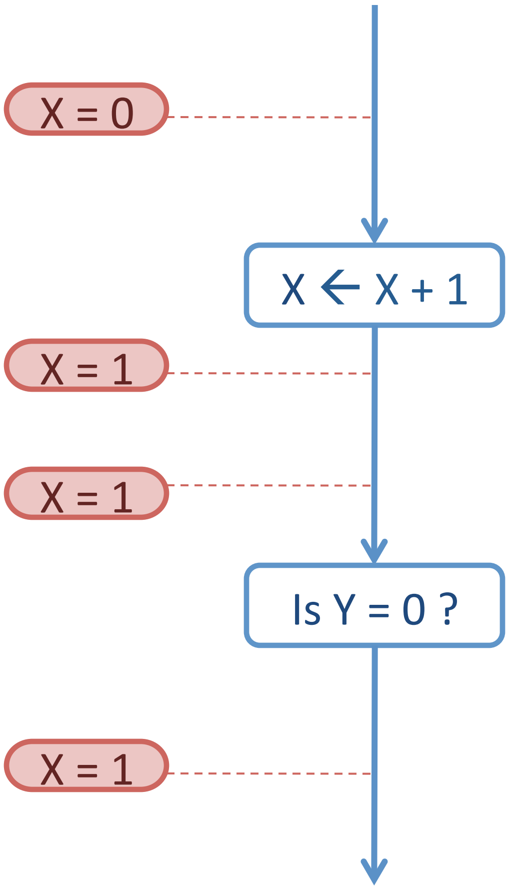
d_{out}

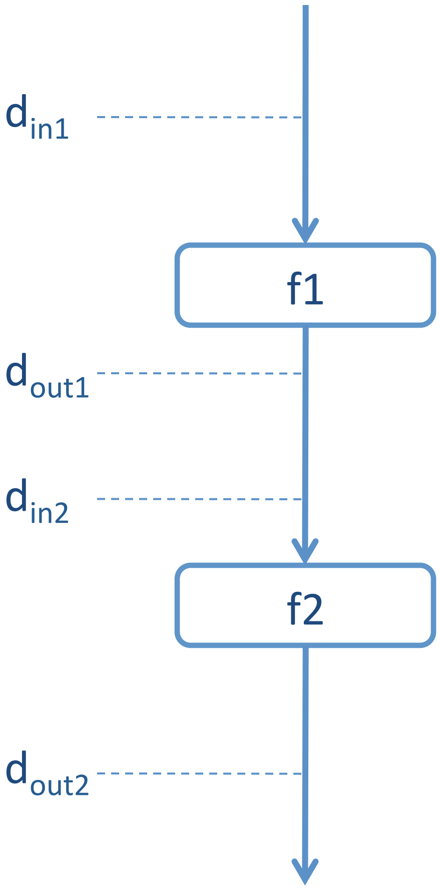
transfer function

$$d_{out} = f(d_{in})$$

dataflow equation

Slide credit: Prof Mitchell Stanford’s CS 155



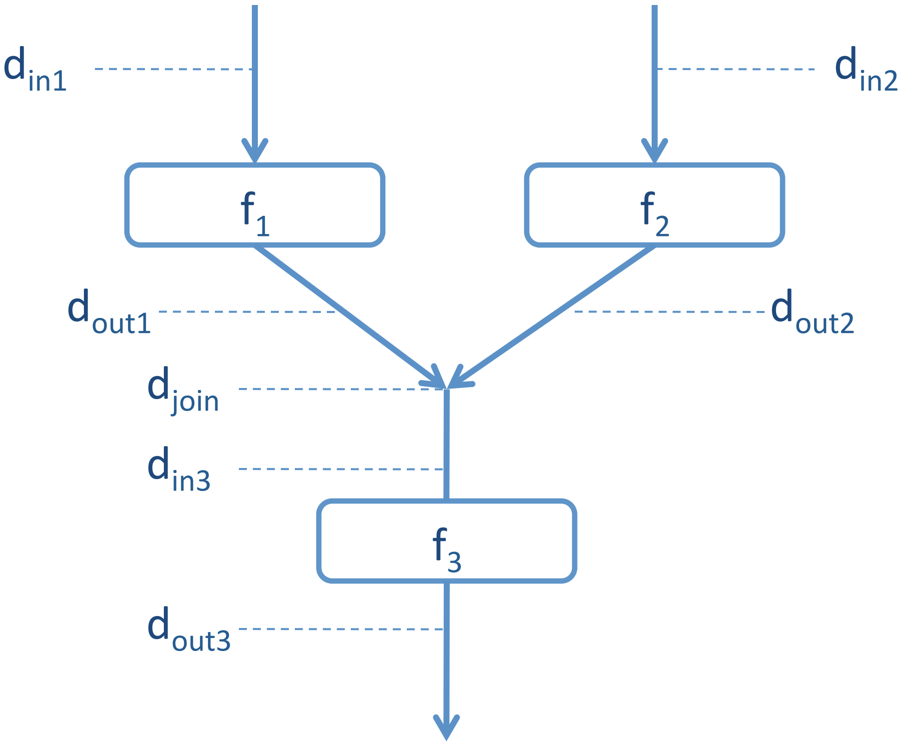


$$d_{out1} = f_1(d_{in1})$$

$$d_{out1} = d_{in2}$$

$$d_{out2} = f_2(d_{in2})$$

Slide credit: Prof Mitchell Stanford’s CS 155



$$d_{out1} = f_1(d_{in1})$$

$$d_{out2} = f_2(d_{in2})$$

$$d_{join} = d_{out1} \sqcup d_{out2}$$

$$d_{join} = d_{in3}$$

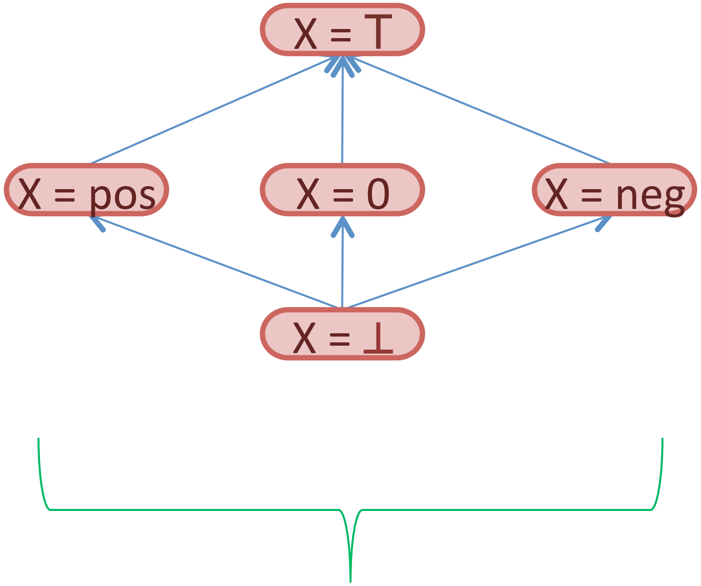
$$d_{out3} = f_3(d_{in3})$$

least upper bound operator
Example: union of possible values

What is the space of dataflow elements, Δ?
What is the least upper bound operator, $\sqcup$?

Slide credit: Prof Mitchell Stanford's CS 155

We give a lattice (partially ordered list with elements representing union and intersection) to specify the possible values we assign to symbolic variables



X = T

X = pos

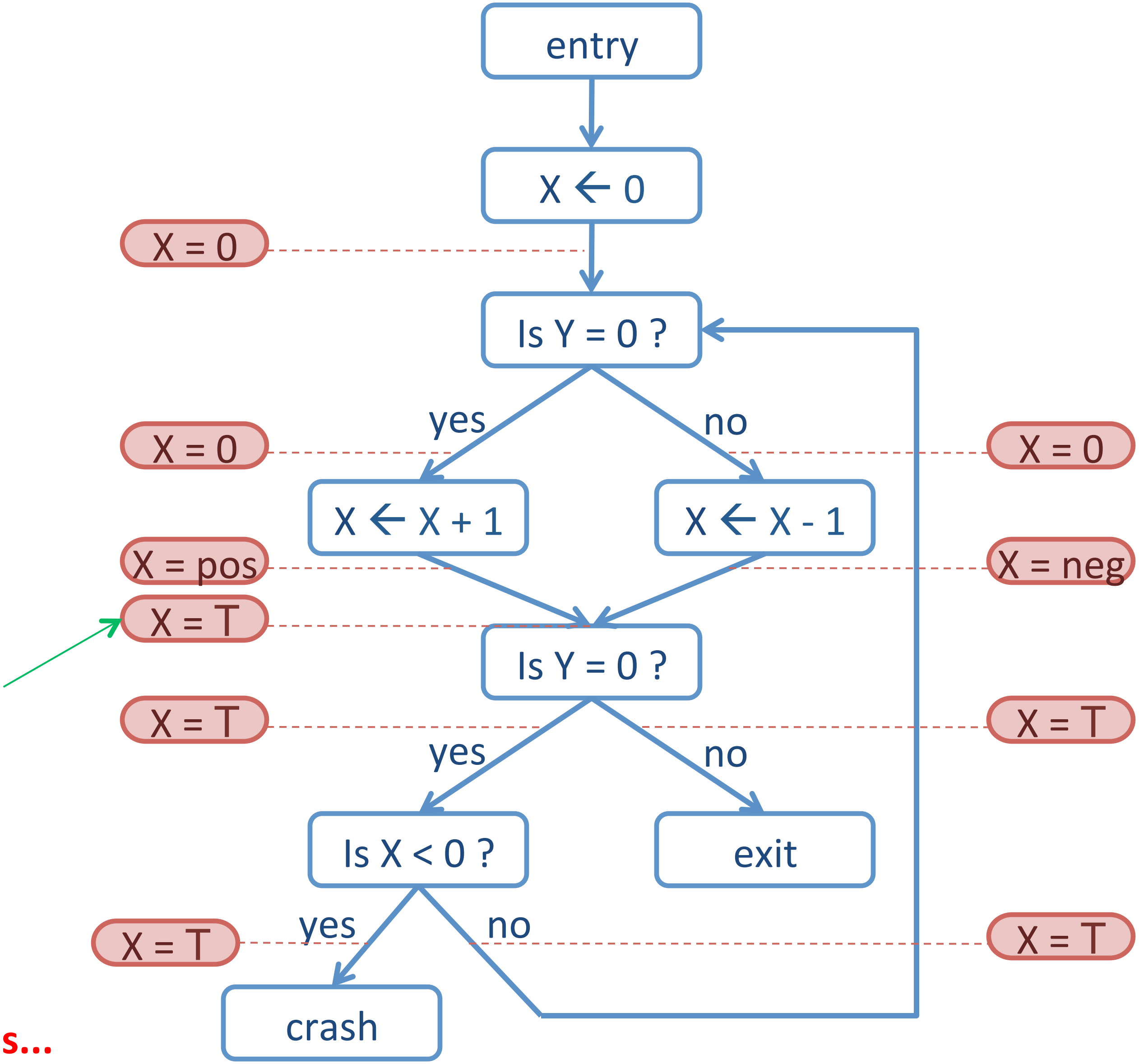
X = 0

X = neg

X = ⊥

signs lattice

Try analyzing with “signs” approximation…

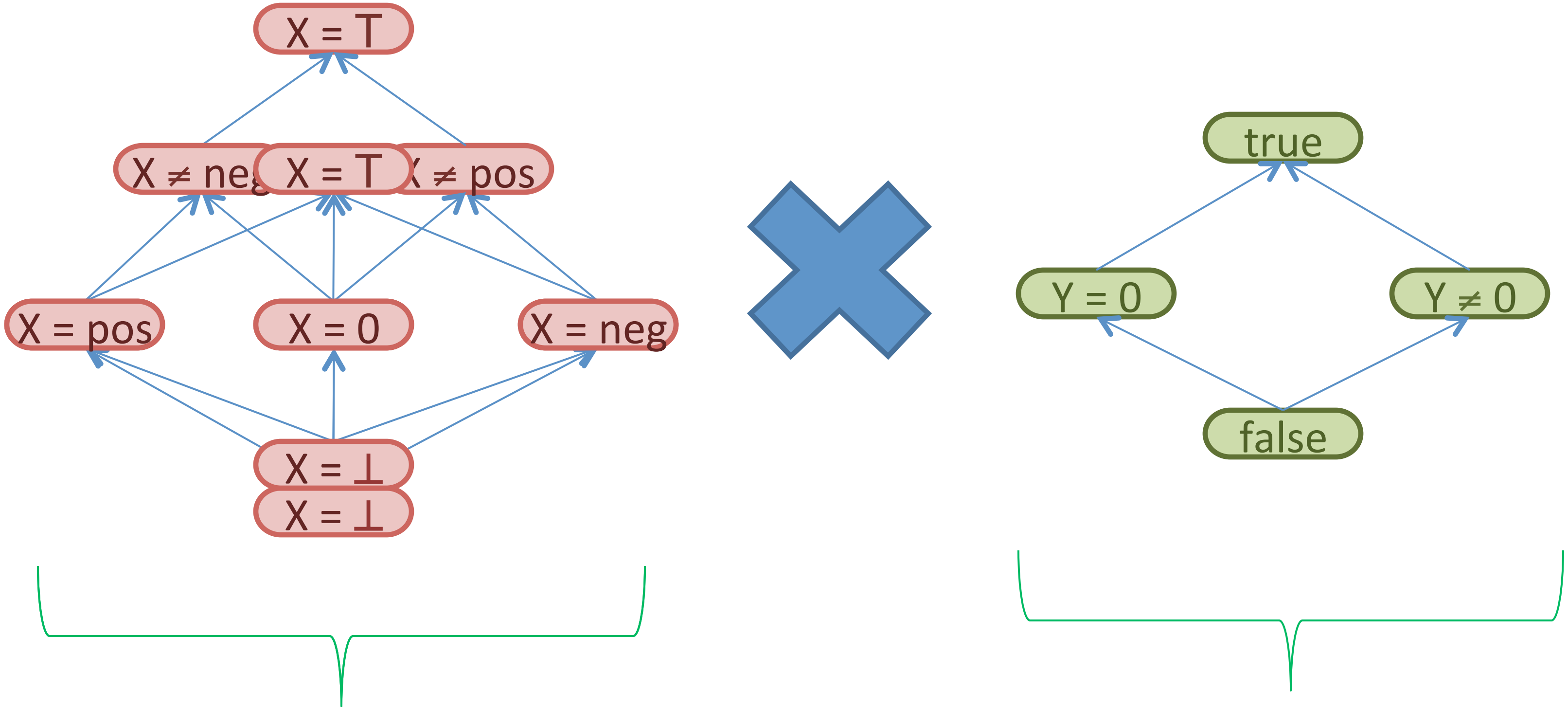


lost precision

terminates…
… but reports false alarm
… therefore, need more precision

Slide credit: Prof Mitchell Stanford’s CS 155

We give a lattice (partially ordered list with elements representing union and intersection) to specify the possible values we assign to symbolic variables

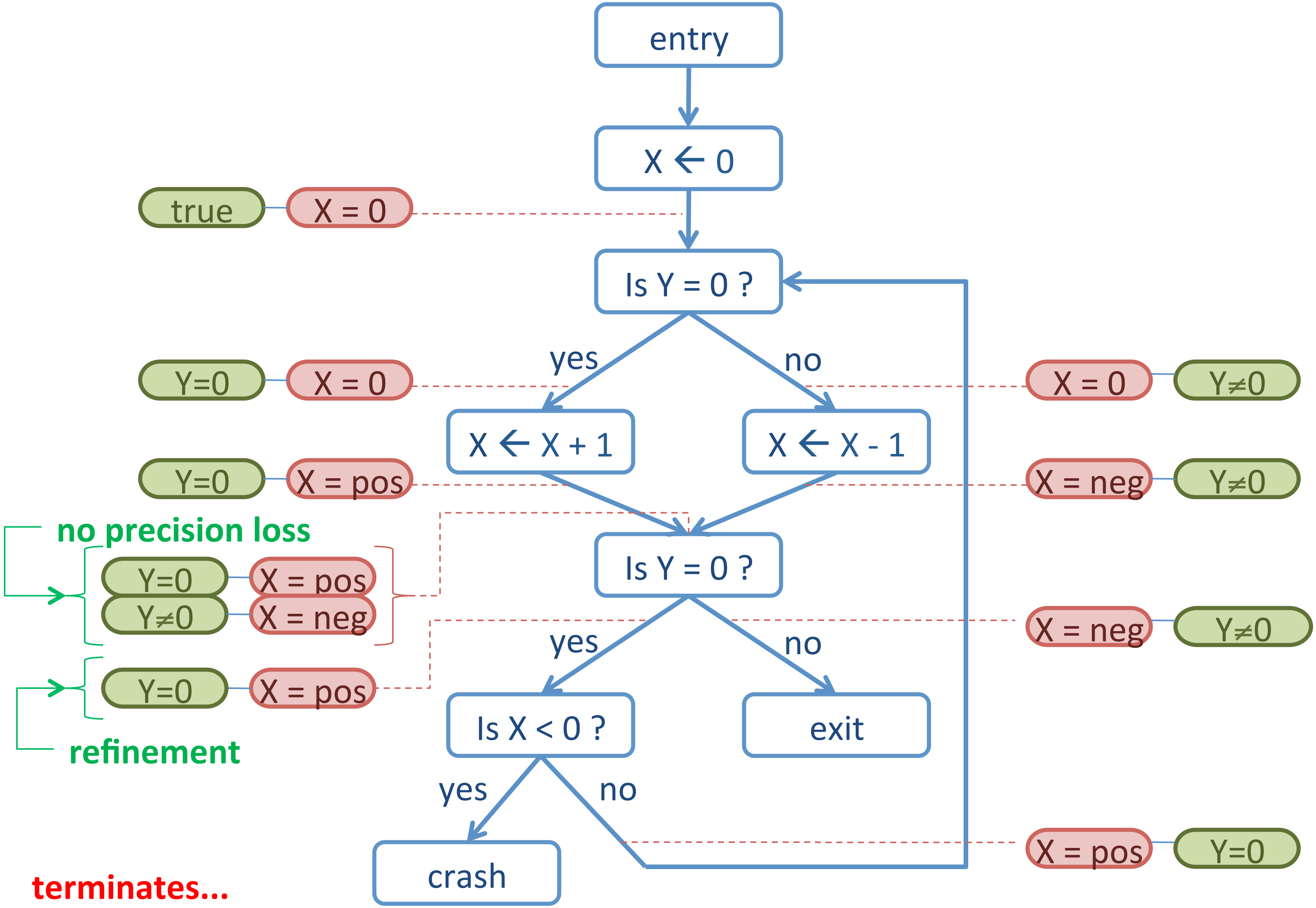


refined sign lattice

sign lattice

Boolean formula lattice

Try analyzing with "path-sensitive signs" approximation…



terminates…

… no false alarm

… soundly proved never crashes

Slide credit: Prof Mitchell Stanford's CS 155

Tales in insecurity…

"The most critical servers contain malicious software that can normally be detected by anti-virus software," it says. "The separation of critical components was not functioning or was not in place. We have strong indications that the CA-servers, although physically very securely placed in a tempest proof environment, were accessible over the network from the management LAN."

DigiNotar

All CA servers were members of one Windows domain and all accessible with one user/password combination. Moreover, the used password was simple and susceptible to brute-force attacks.

http://www.net-security.org/secworld.php?id=11570