Canon Printer Hacked To Run *Doom* Video Game

wiredog writes Security researcher Michael Jordon has hacked a Canon's Pixma printer to run *Doom*. He did so by reverse engineering the firmware encryption and uploading via the update interface. From the BBC: "Like many modern printers, Canon's Pixma range can be accessed via the net, so owners can check the device's status. However, Mr Jordon, who works for Context Information Security, found Canon had done a poor job of securing this method of interrogating the device. 'The web interface has no user name or password on it,' he said. That meant anyone could look at the status of any device once they found it, he said. A check via the Shodan search engine suggests there are thousands of potentially vulnerable Pixma printers already discoverable online. There is no evidence that anyone is attacking printers via the route Mr Jordon found."

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

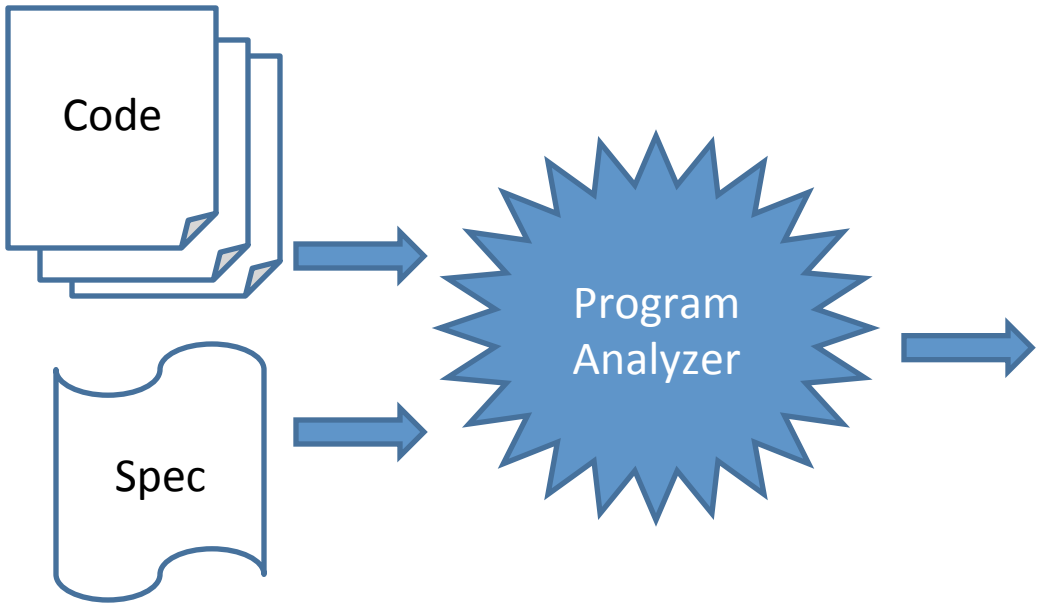
Static analysis, dynamic analysis

…

Demo example from Monday's class

```
#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] );
}
```

Program analyzers



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 analysis: false positives and false negatives

Term	Definition
False positive	A spurious warning that does not indicate an actual vulnerability
False negative	Does not emit a warning for an actual vulnerability

Complete analysis: no false negatives

Sound analysis: no false positives

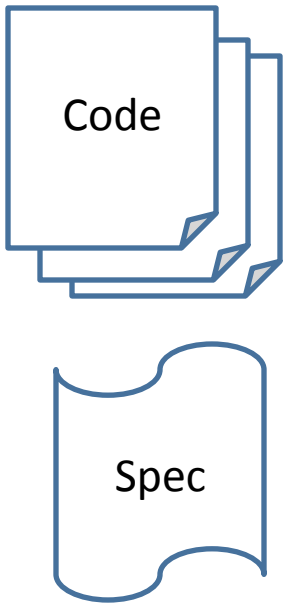
	Complete	Incomplete
Sound	Reports all errors Reports no false alarms No false positives No false negatives **Undecidable**	Reports all errors May report false alarms No false negatives False positives **Decidable**
Unsound	May not report all errors Reports no false alarms False positives No false negatives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

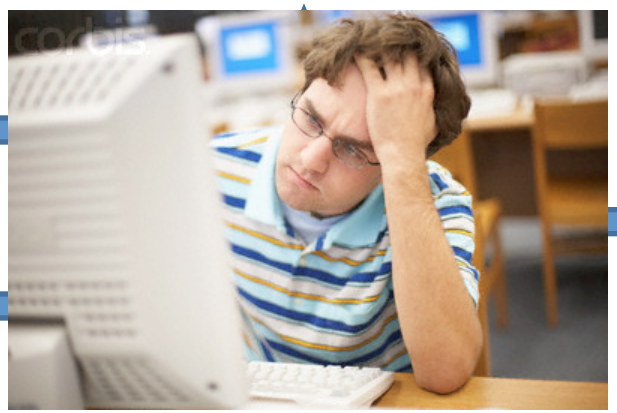
Slide credit: Prof Mitchell Stanford's CS 155

Example program analyzers

- Manual analysis (you are the analyzer!)
- Static analysis (do not execute program)
  - Scanners
  - Symbolic execution
  - Abstract representations
- Dynamic analysis (execute program)
  - Debugging
  - Fuzzers
  - Ptrace

Program analyzers





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

analyze large code bases

false alarm

false alarm

potentially reports many warnings

may emit false alarms

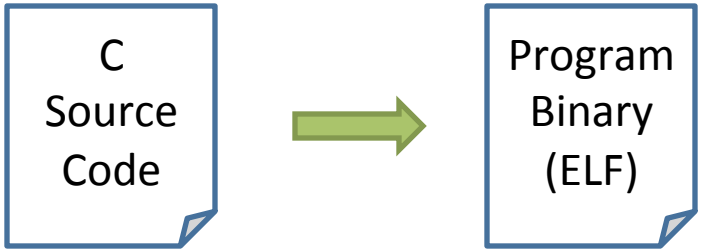
Manual analysis

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

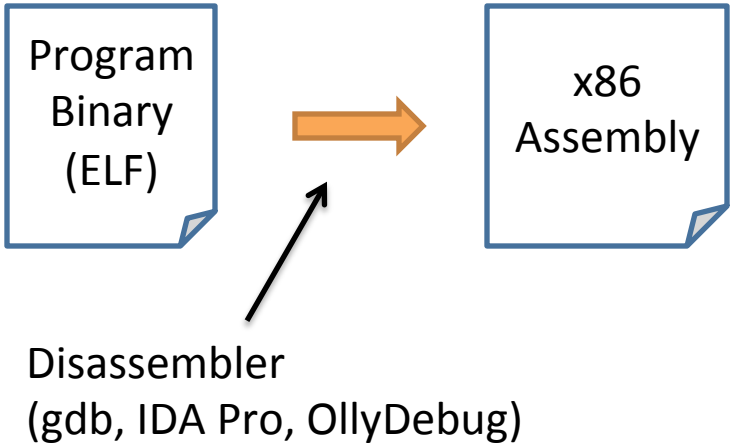
- Experienced analysts according to Aitel:
  - 1 hour of binary analysis:
    - Simple backdoors, coding style, bad API calls (strcpy)
  - 1 week of binary analysis:
    - Likely to find 1 good vulnerability
  - 1 month of binary analysis:
    - Likely to find 1 vulnerability *no one else will ever find*

Disassembly and decompiling

The normal compilation process



What if we start with binary?



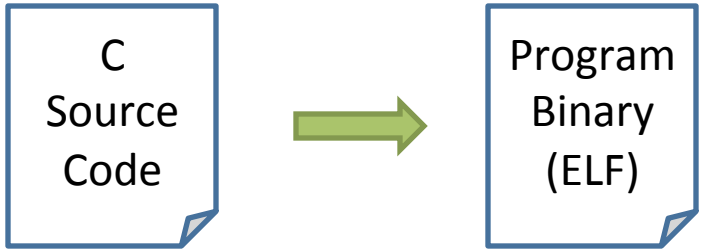
```
movl   $0xf8,(%esp)
call   0x8048364 <malloc@plt>
mov    %eax,0x14(%esp)
movl   $0xf8,(%esp)
call   0x8048364 <malloc@plt>
mov    %eax,0x18(%esp)
mov    0x14(%esp),%eax
mov    %eax,(%esp)
call   0x8048354 <free@plt>
mov    0x18(%esp),%eax
mov    %eax,(%esp)
call   0x8048354 <free@plt>
movl   $0x200,(%esp)
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?

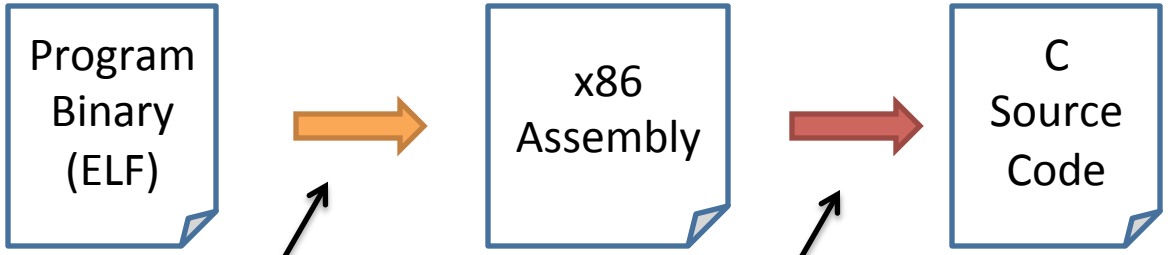
Double-free vulnerability

Disassembly and decompiling

The normal compilation process



What if we start with binary?

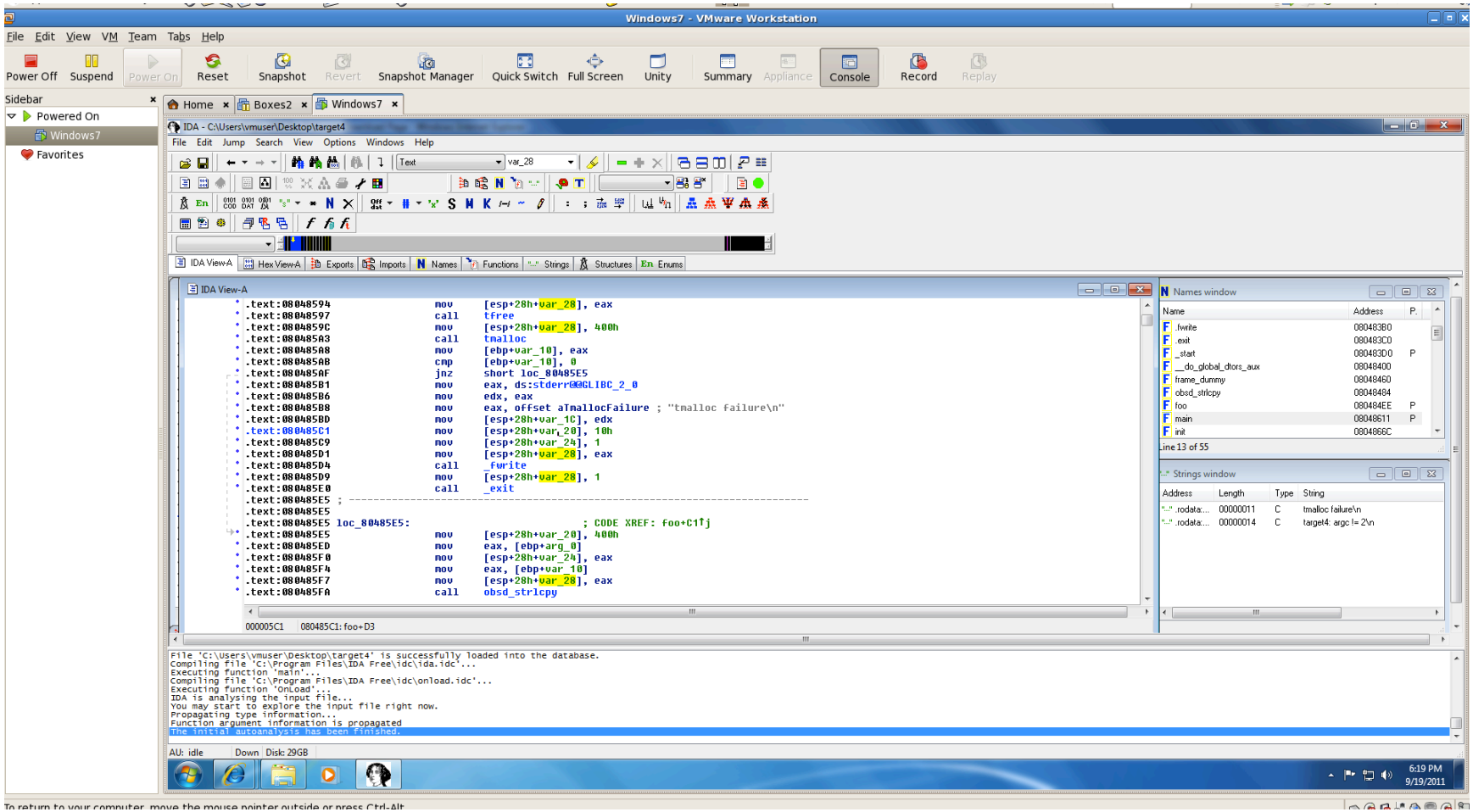


Disassembler
(gdb, IDA Pro, OllyDebug)

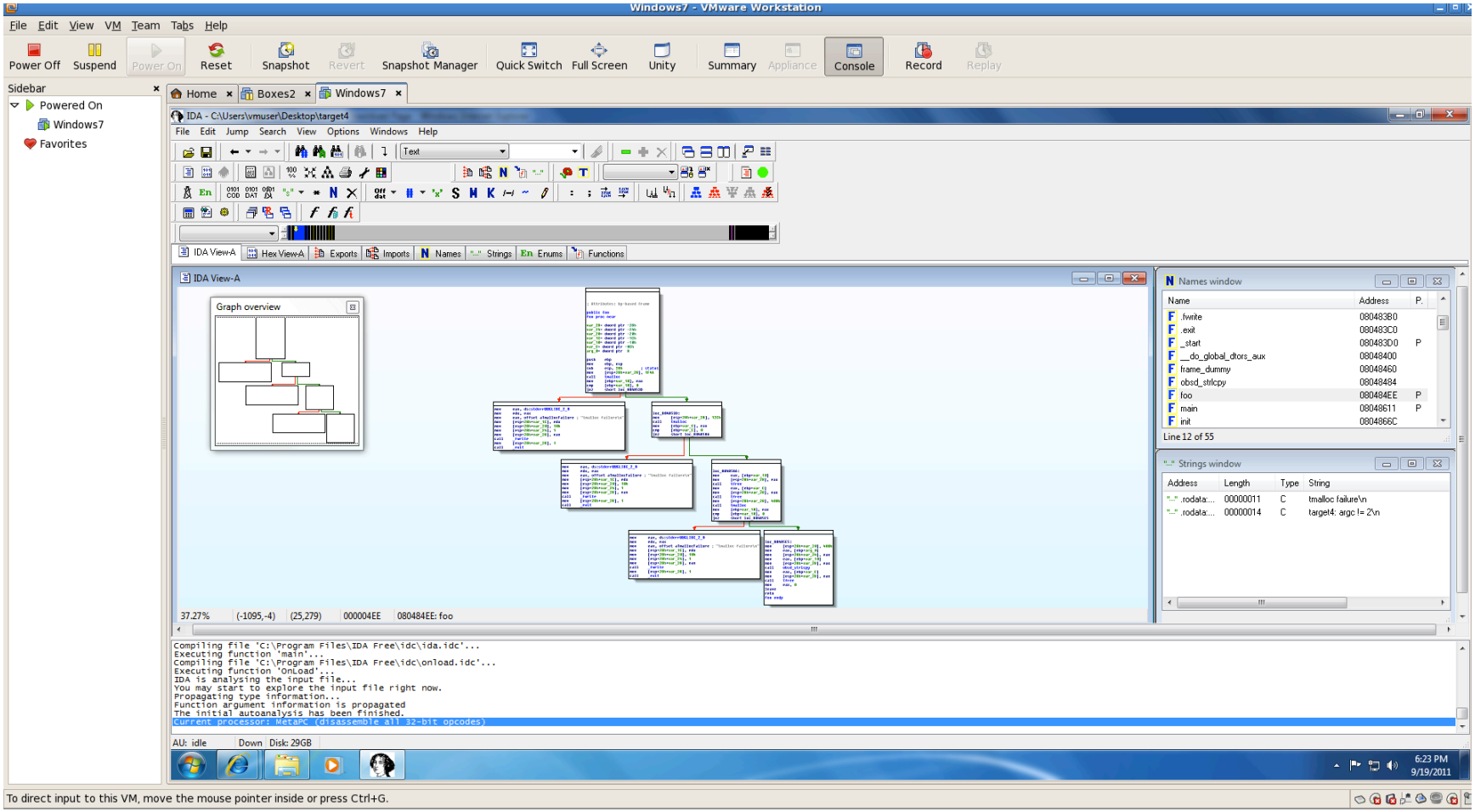
Decompiler
(IDA Pro has one)

Very complex, usually poor results

Tool example: IDA Pro



Tool example: IDA Pro



```
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?

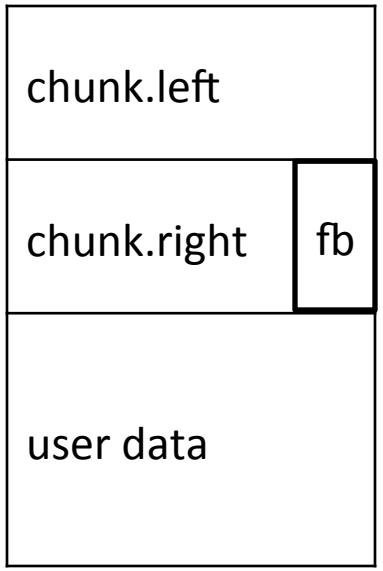
```
main(  int argc, char* argv[] ) {
  char* b1;
  char* b2;
  char* b3;

  if( argc != 3 ) then return 0;
  if( atoi(argv[2]) != 31337 )
      complicatedFunction();
  else {
      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

Digression: Double-free vulnerabilities

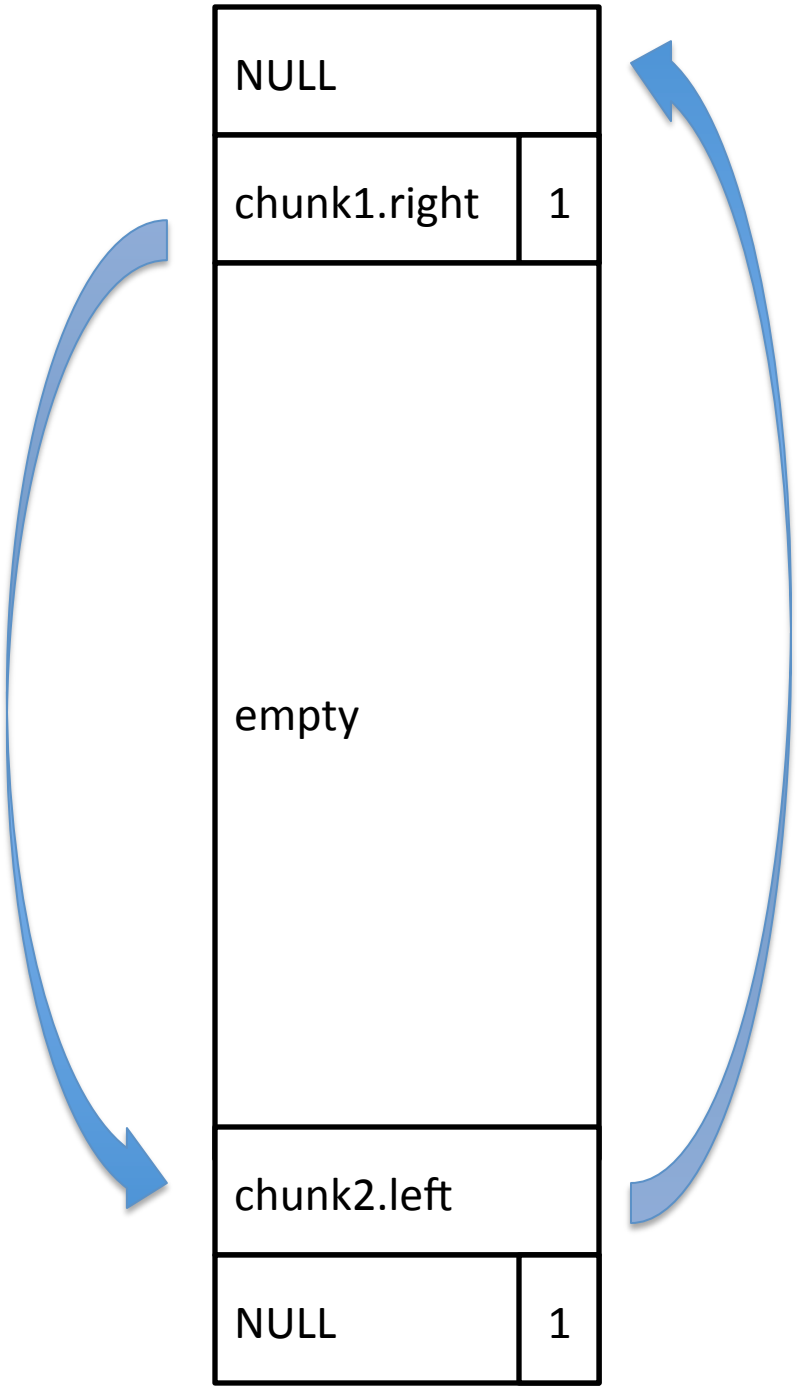
Can corrupt the state of the heap management

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



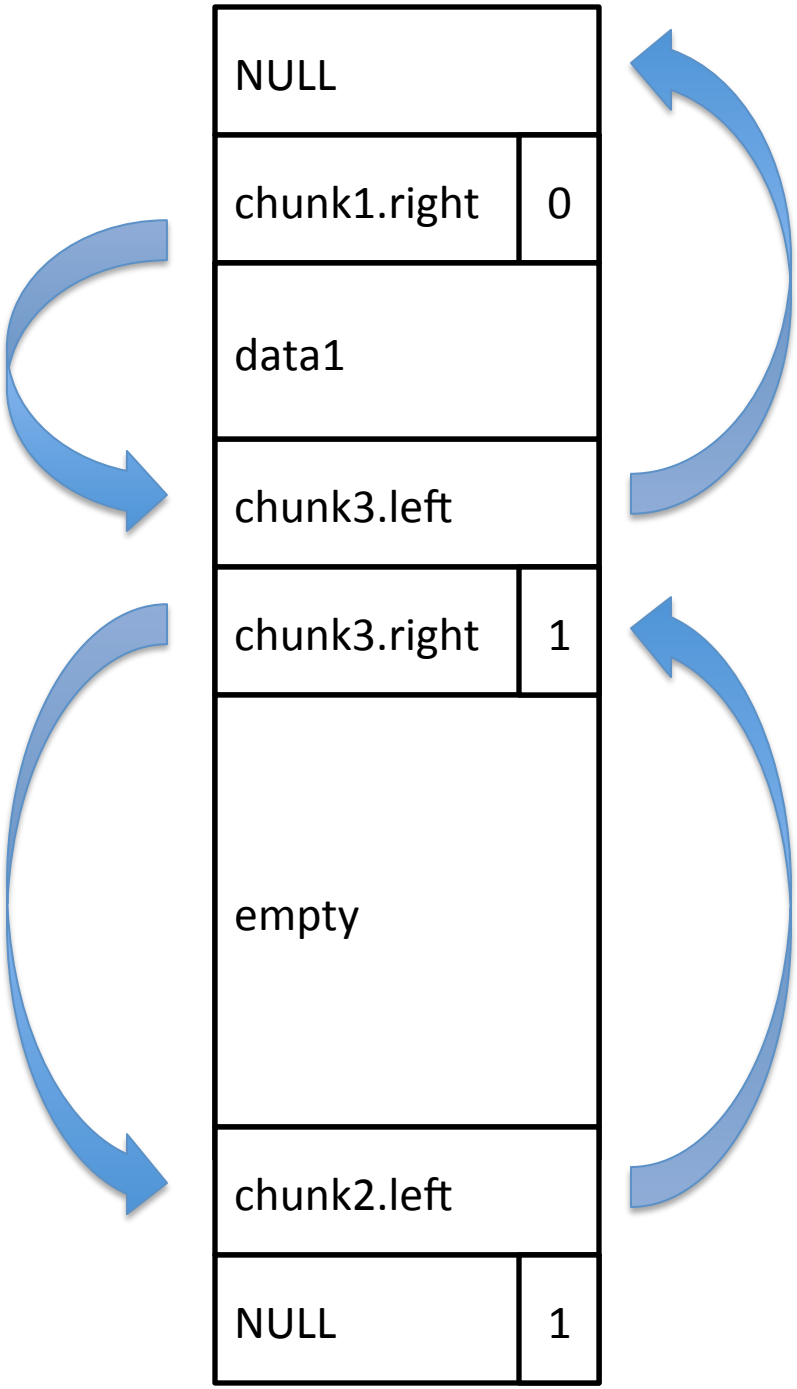
Chunk has:

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



malloc()

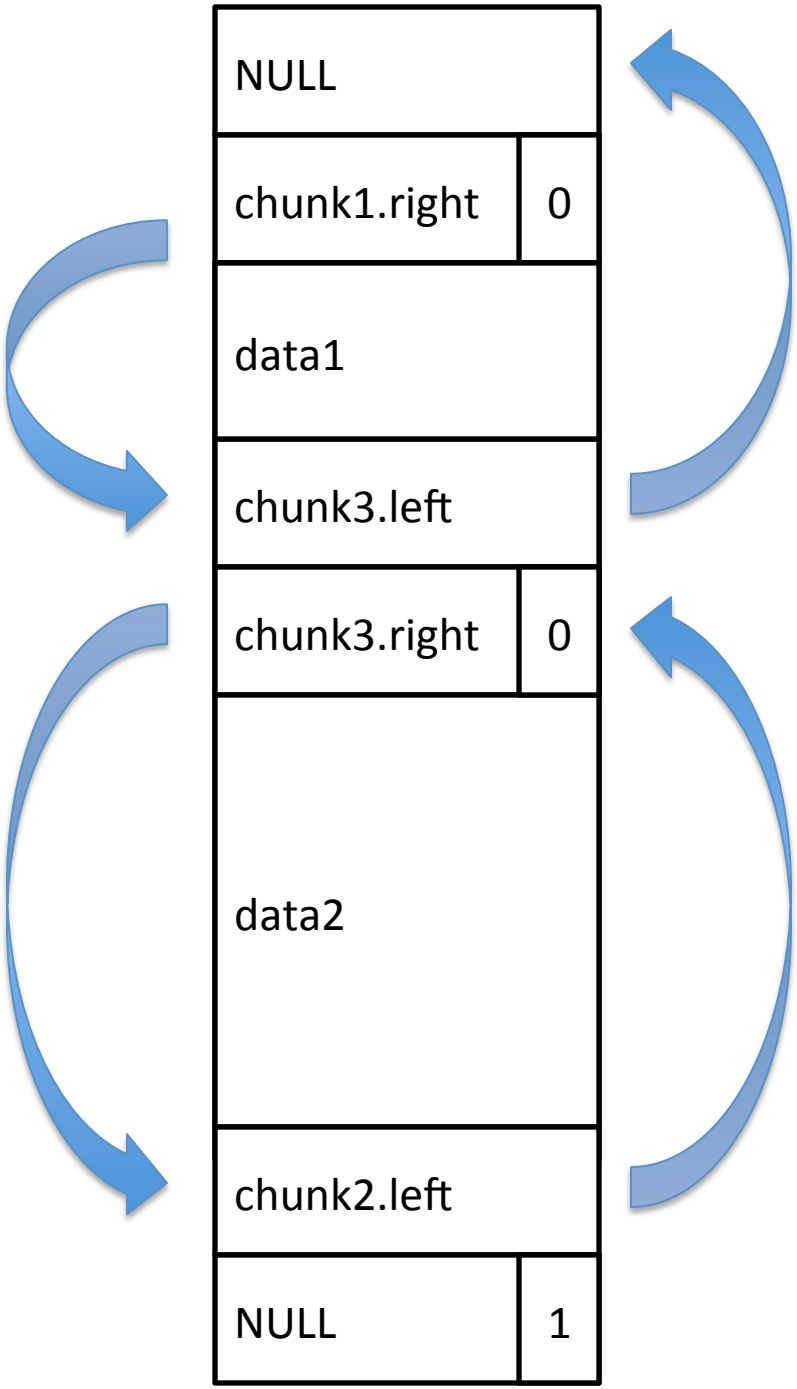
- search left-to-right for free chunk
- modify pointers



malloc()

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

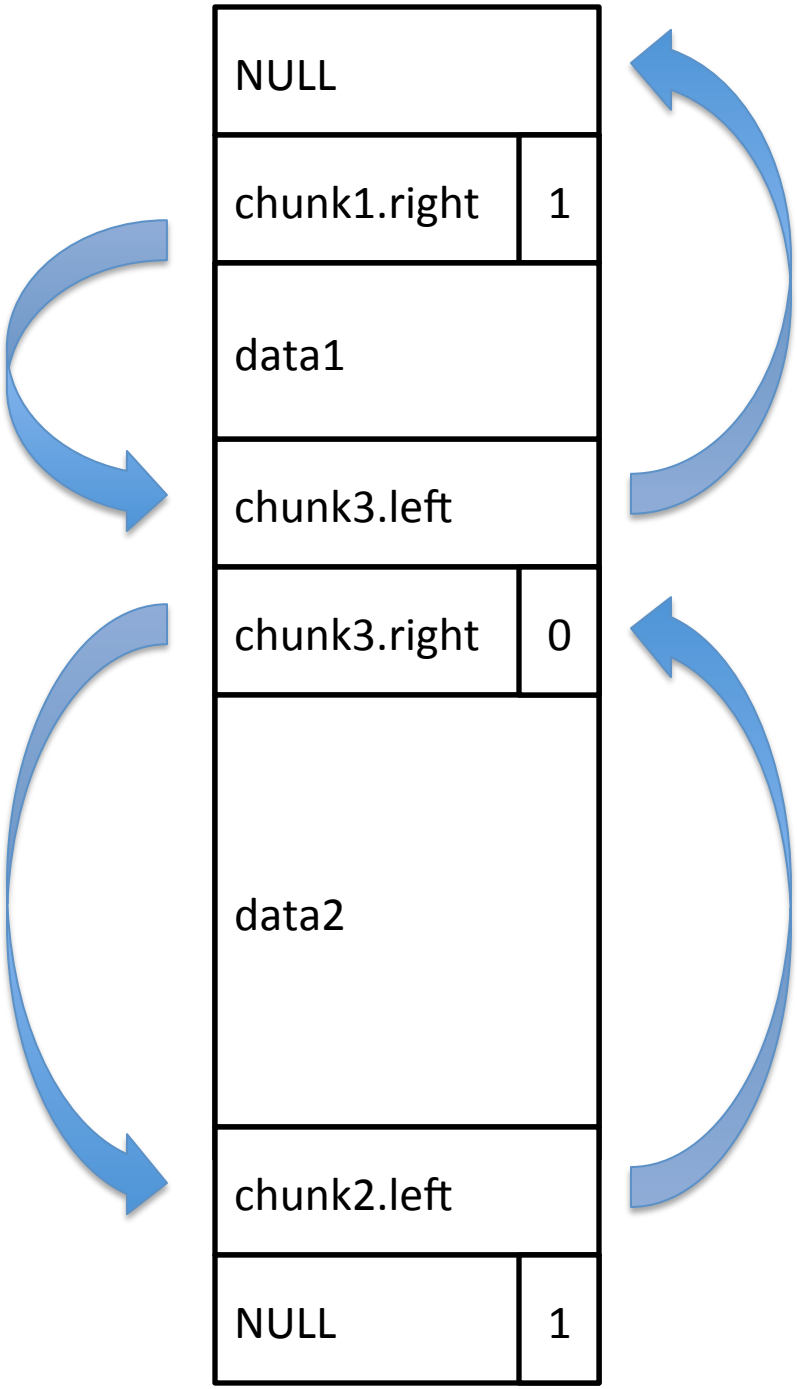
b1 = malloc(BUF_SIZE1);



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

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

free()
- Consolidate with free neighbors

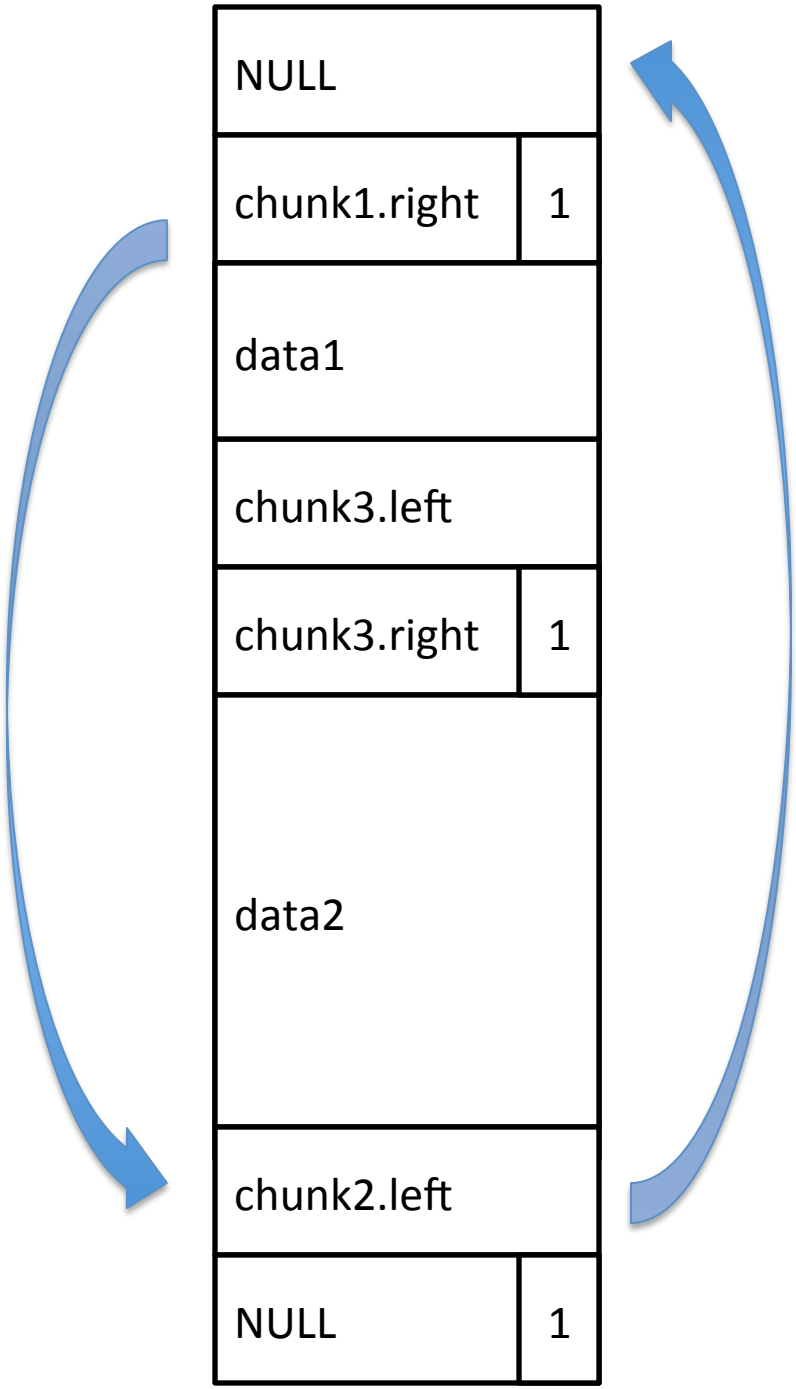


malloc()
- search left-to-right for free chunk
- modify pointers

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

free()
- Consolidate with free neighbors

free(b1)

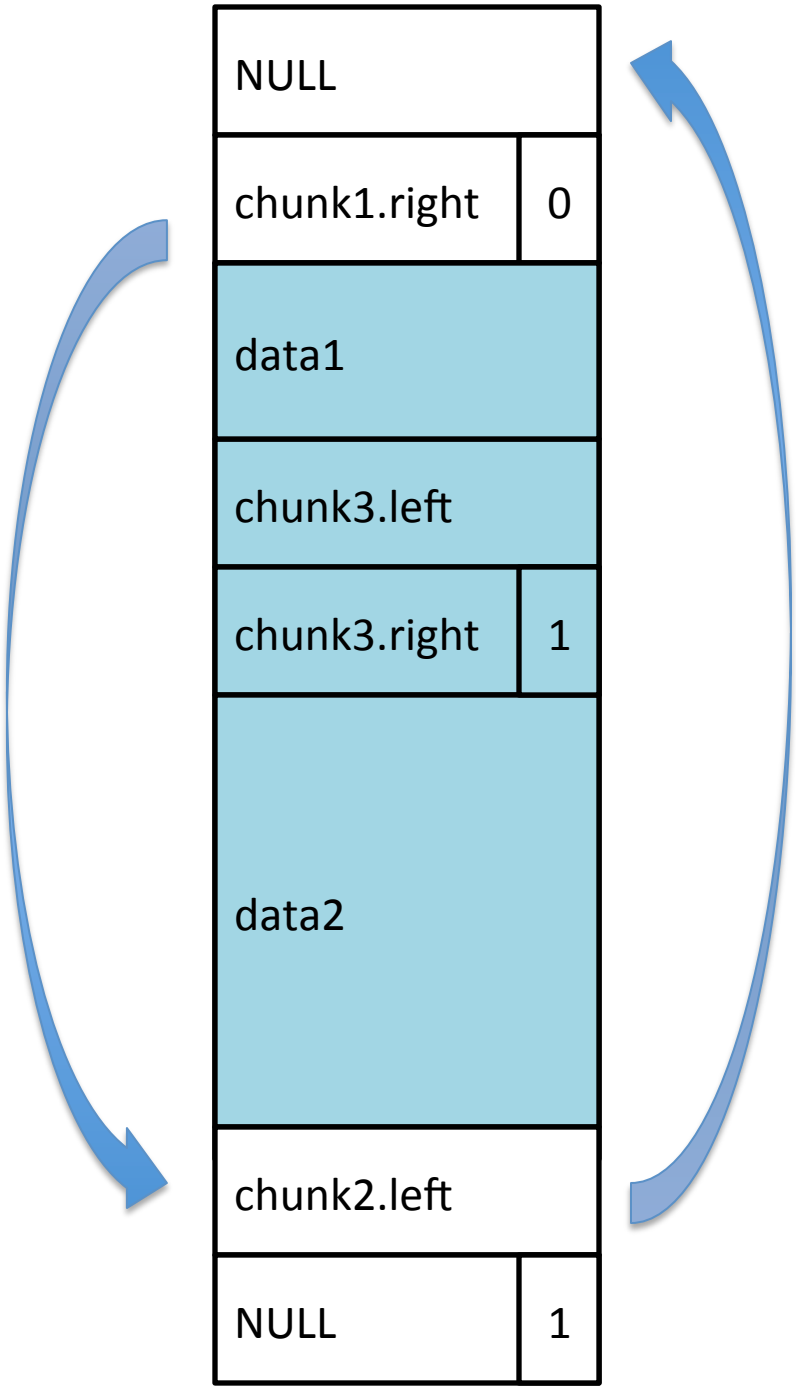


malloc()
- search left-to-right for free chunk
- modify pointers

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

free()
- Consolidate with free neighbors

free(b1)
free(b2)

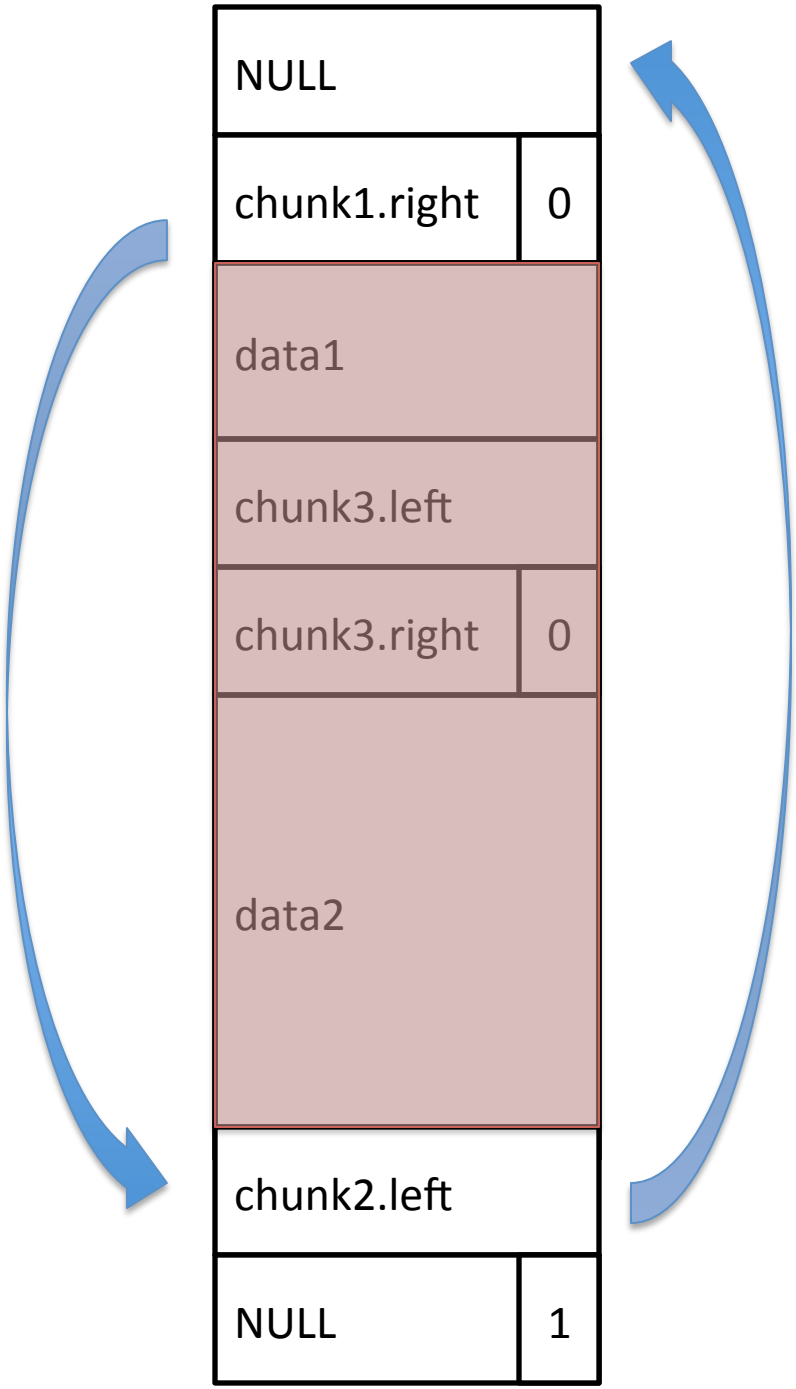


malloc()
- search left-to-right for free chunk
- modify pointers

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

free()
- Consolidate with free neighbors

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

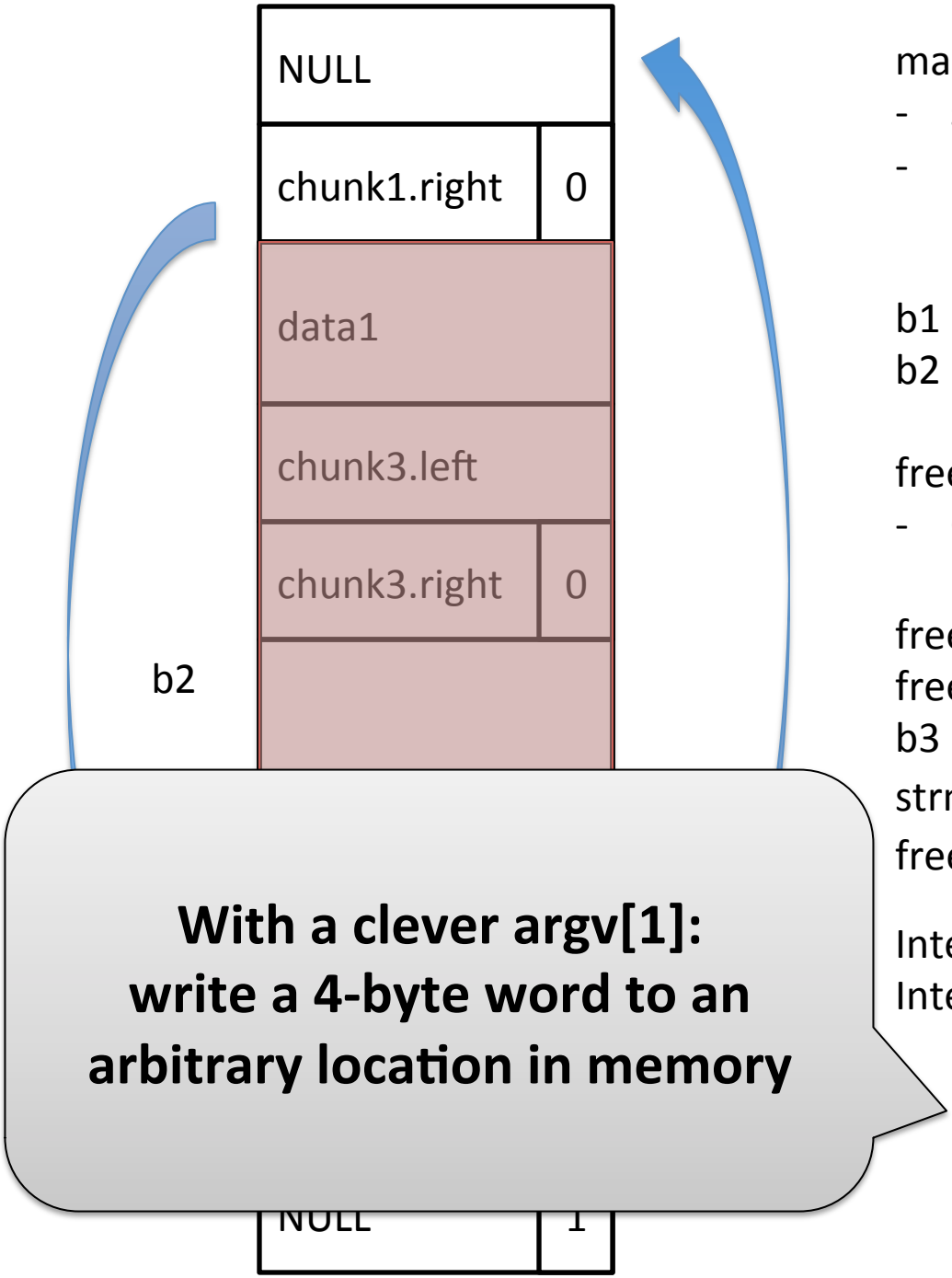


malloc()
- search left-to-right for free chunk
- modify pointers

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

Aiding analysts with tools

How can we automatically find the bug?

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

  if( argc != 3 ) then return 0;
  if( atoi(argv[2]) != 31337 )
      complicatedFunction();
  else {
      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);
   }
}
```

Example tools / approaches

Approach	Type	Comment
Lexical analyzers	Static analysis	Perform syntactic checks Ex: LINT, RATS, ITS4
Fuzz testing	Dynamic analysis	Run on specially crafted inputs to test
Symbolic execution	Emulated execution	Run program on many inputs at once, by Ex: KLEE, S2E, FiE
Model checking	Static analysis	Abstract program to a model, check that model satisfies security properties Ex: MOPS, SLAM, etc.

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

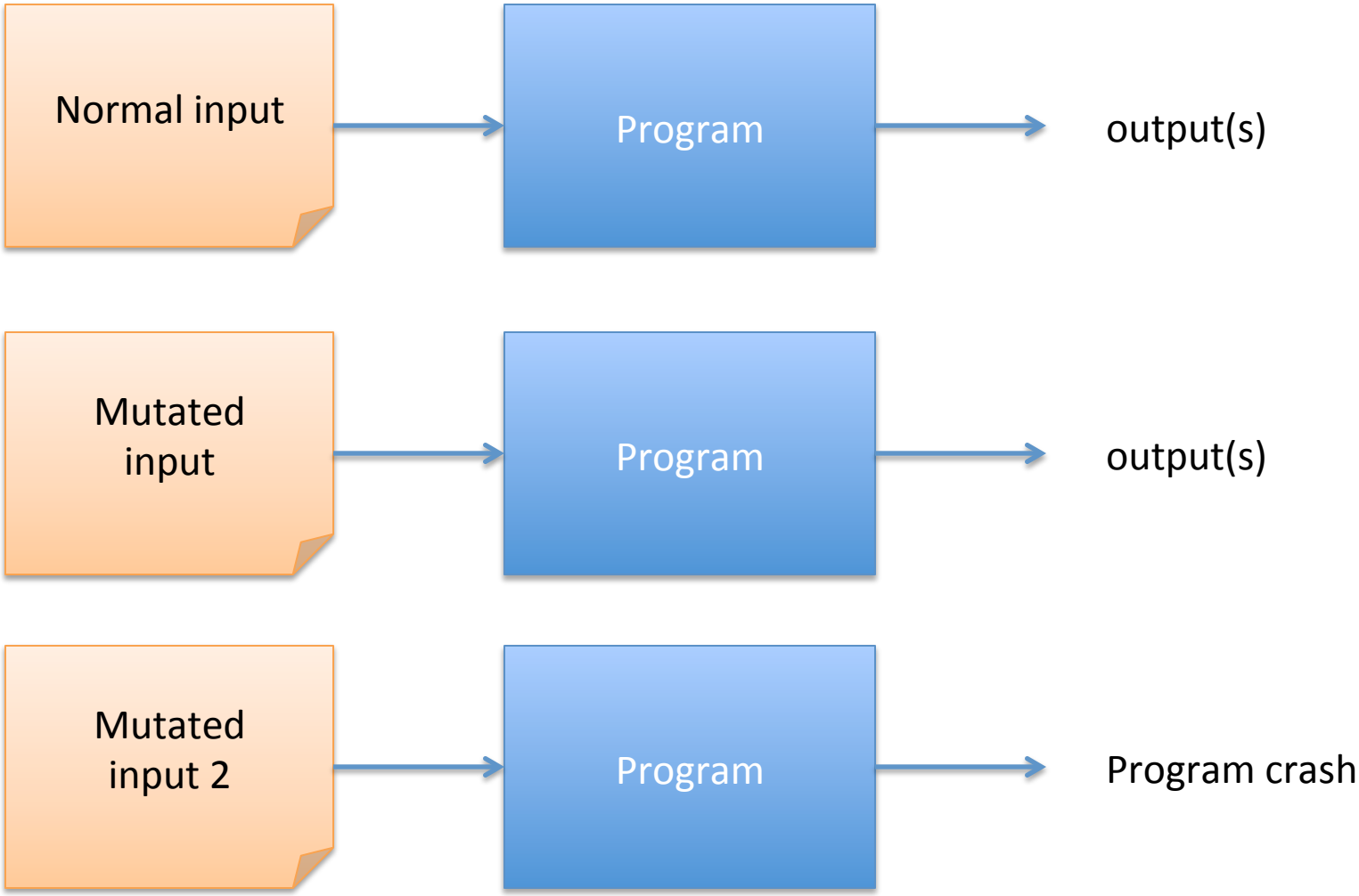
(… but probably will)

Dynamic analysis: Fuzzing

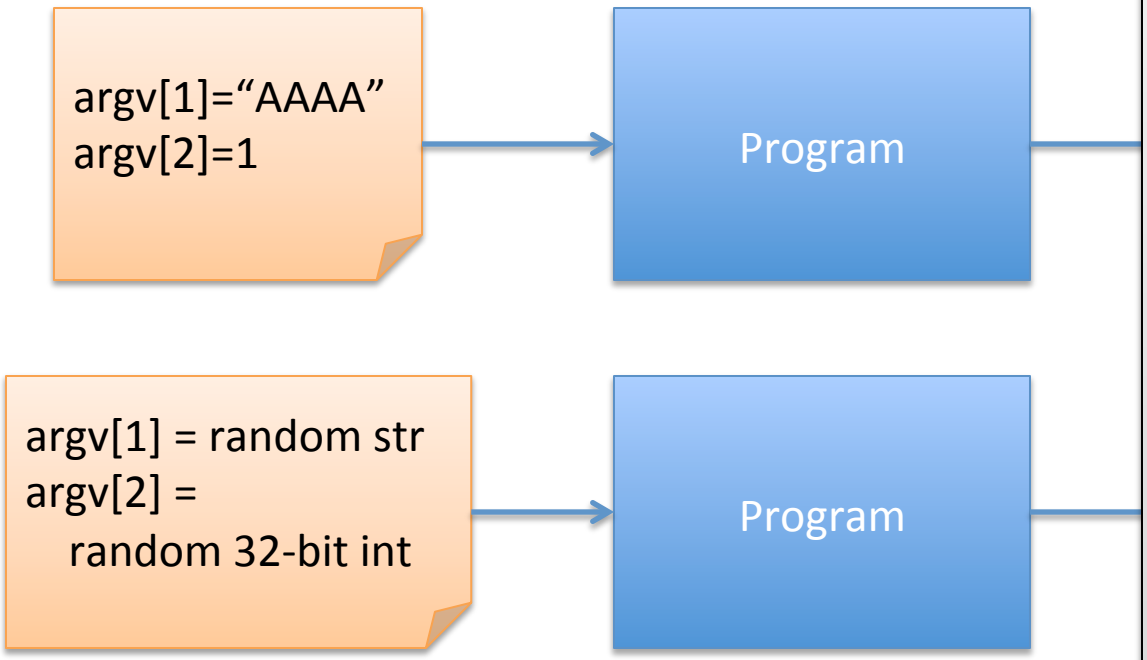
"The term first originates from a class project at the University of Wisconsin 1988 although similar techniques have been used in the field of quality assurance, where they are referred to as robustness testing, syntax testing or negative testing."
Wikipedia
http://en.wikipedia.org/wiki/Fuzz_testing

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

Black-box fuzz testing: the goal



Black-box fuzz testing




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

  if( argc != 3 ) then return 0;
  if( atoi(argv[2]) != 31337 )
      complicatedFunction();
  else {
      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);
   }
}
```

If integers are 32 bits, then probability of crashing is **at most what**? $1/2^{32}$

Achieving code coverage can be very difficult

Code coverage and fuzzing

- Code coverage defined in many ways
  - # of basic blocks reached
  - # of paths followed
  - # of conditionals followed
  - gcov is useful standard tool
- 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: Fuzzing Freeciv (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

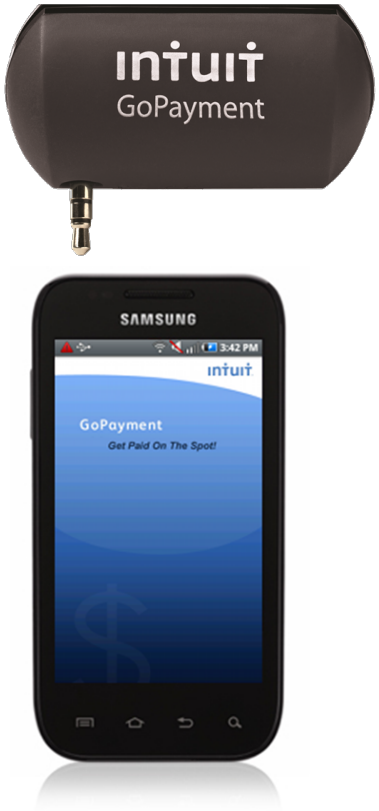
One big switch statement controlled by third byte of packet
Update fuzz rules to exhaust the values of this third byte

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

Example: IDTech Firmware

Point-of-sale credit card reader for smart phones

- No access to firmware binary, let alone source code



<u>What we did:</u>

- Scour web for documentation
- Download SDK for app developers (bingo!)
  - Use to build fuzzing functionality for each API call we could find
  - Use to discover undocumented API calls

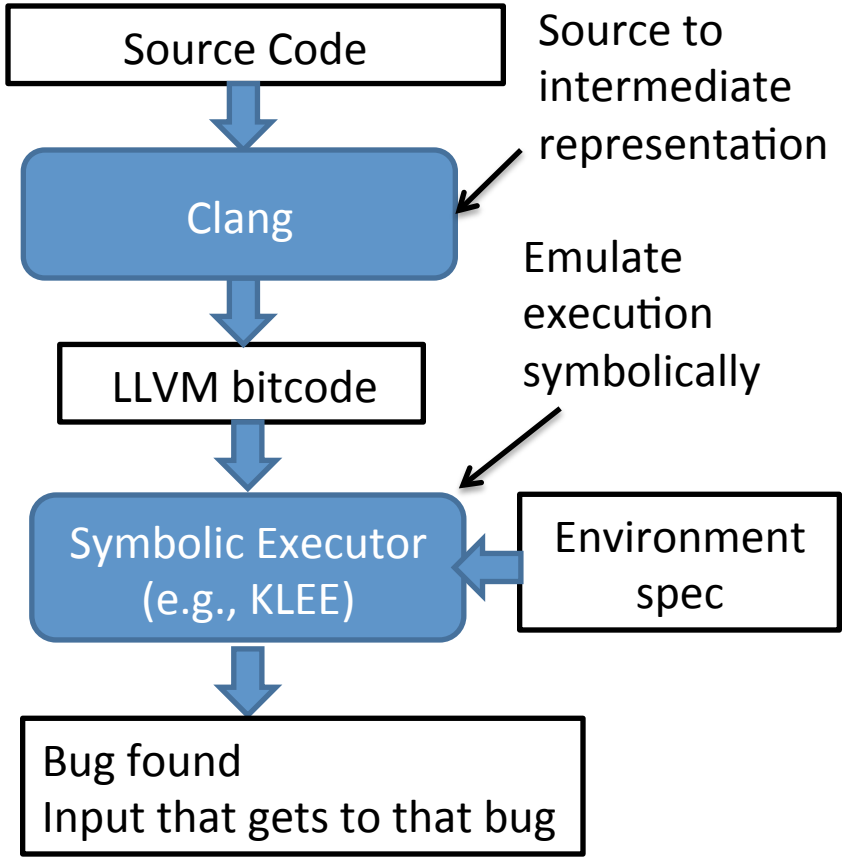
- Found unauthenticated API calls, buffer bound check problems, ultimately brick device, reveals crypto keys, cleartext credit card data)

http://pages.cs.wisc.edu/~rist/papers/pos.pdf

Example tools / approaches

Approach	Type	Comment
Lexical analyzers	Static analysis	Perform syntactic checks Ex: LINT, RATS, ITS4
Fuzz testing	Dynamic analysis	Run on specially crafted inputs to test
Symbolic execution	Emulated execution	Run program on many inputs at once, by Ex: KLEE, S2E, FiE
Model checking	Static analysis	Abstract program to a model, check that model satisfies security properties Ex: MOPS, SLAM, etc.

Symbolic execution



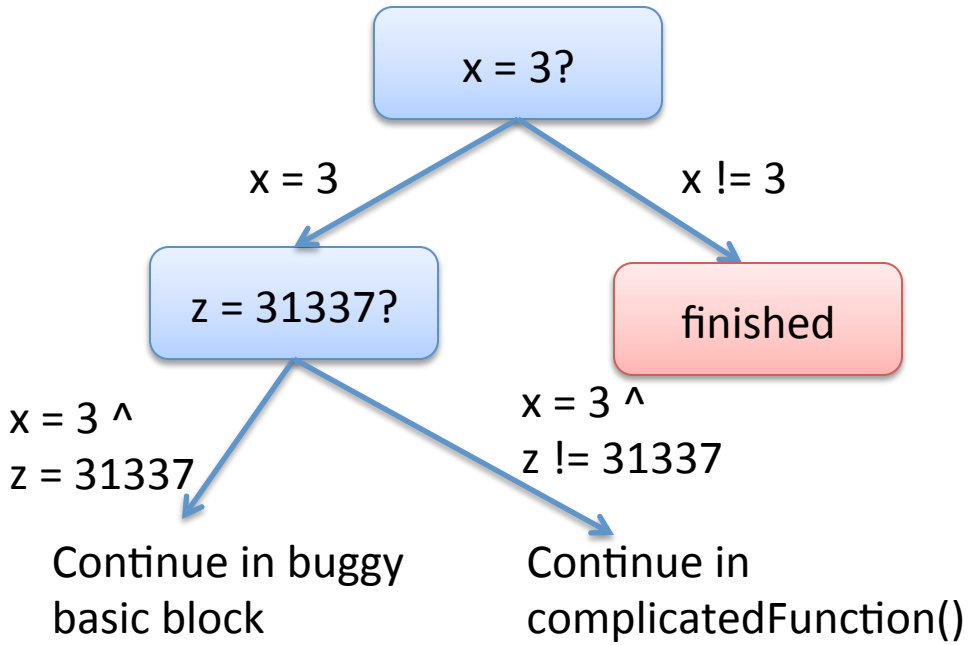
- Technique for statically analyzing code paths and finding inputs
- Associate to each input variable a special symbol
  - called symbolic variable
- Simulate execution symbolically
  - Update symbolic variable's value appropriately
  - Conditionals add constraints on possible values
- Cast constraints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

Symbolic execution

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

  if( argc != 3 ) then return 0;
  if( argv[2] != 31337 )
      complicatedFunction();
  else {
      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);
   }
}
```

Initially:

argc = x (unconstrained int)
argv[2] = z (memory array)



- Eventually emulation hits a double free
- Can trace back up path to determine what x, z must have been to hit this basic block

Symbolic execution challenges

- Can we complete analyses?
  - Yes, but only for very simple programs
  - Exponential # of paths to explore
- Path selection
  - Might get stuck in complicatedFunction()
- Encoding checks on symbolic states
  - Must include logic for double free check
  - Symbolic execution on binary more challenging (lose most memory semantics)

White-box fuzz testing

- Start with real input and
  - Perform symbolic execution of program
  - Gather constraints (control flow) along way
  - Systematically negate constraints backwards
  - Eventually this yields a new input
- Repeat
- In-use at Microsoft

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

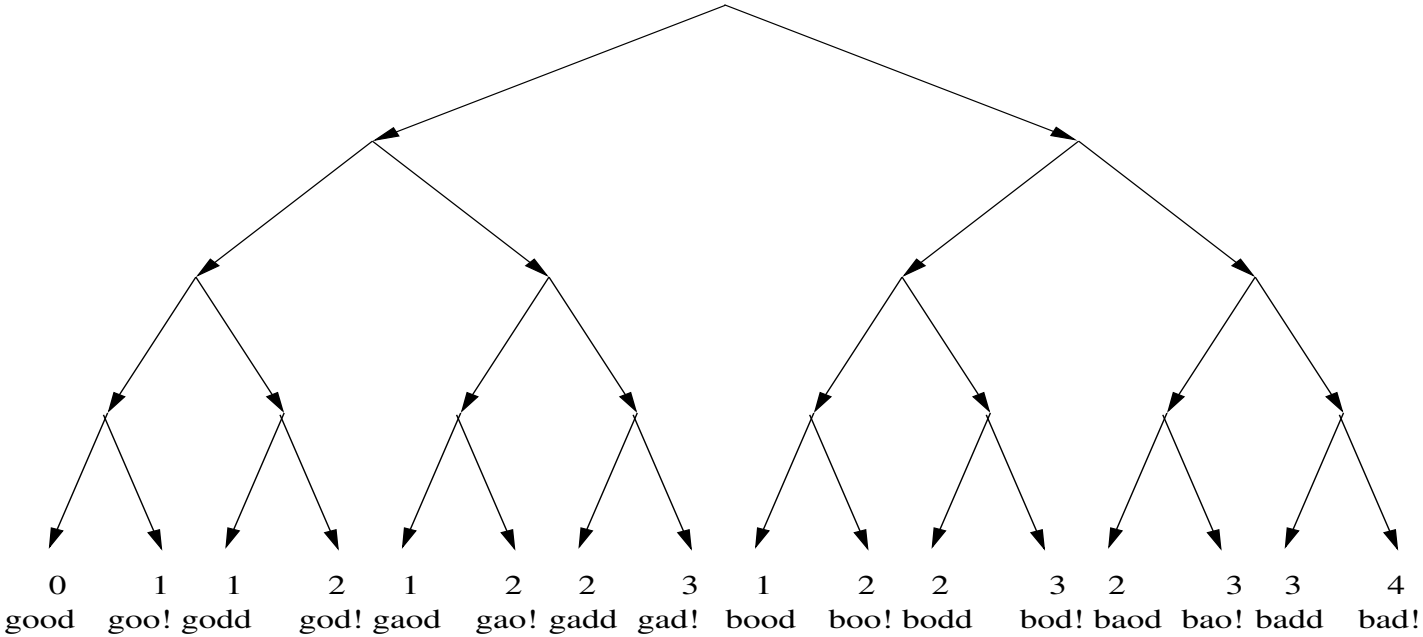


Figure 2. Search space for the example of Figure 1 with the value of the variable `cnt` at the end of each run and the corresponding input string.

Example from Godefroid et al.

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

In-use at Microsoft

Bug finding is a big business

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

Great article on static analysis in the real world:
http://web.stanford.edu/~engler/BLOC-coverity.pdf