

# 50.003 Elements of Software Construction

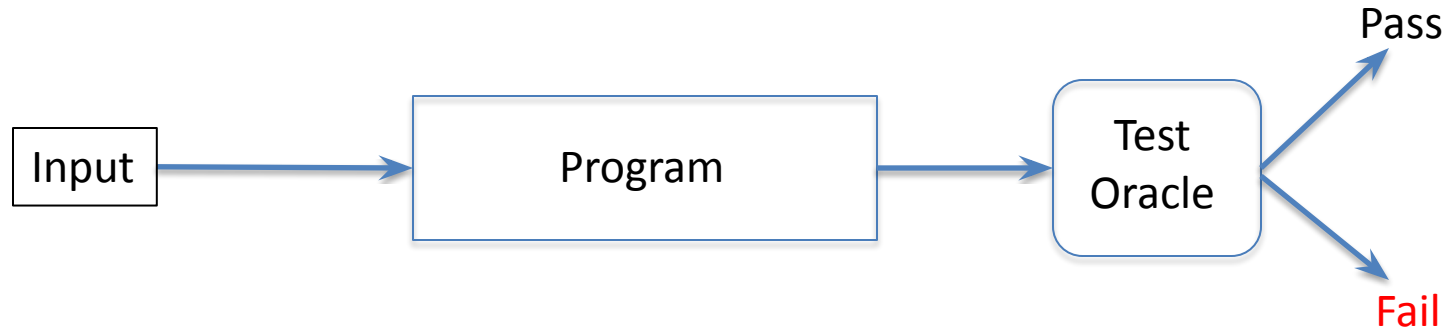
Auto test generation

# Learning Outcomes

By the end of this unit, you should be able to

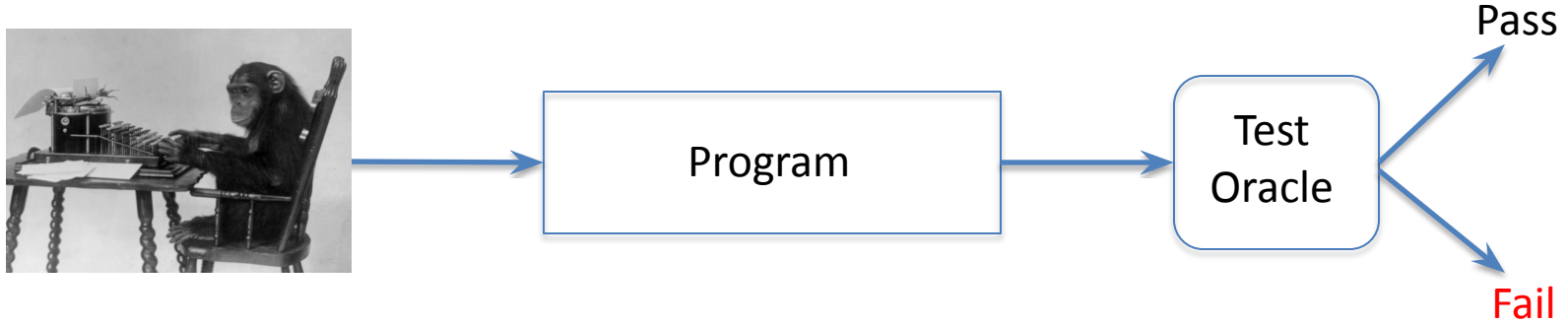
1. explain the roles and functionalities of the test generator.
2. apply mutation-based fuzzing to generate test cases.
3. apply generation-based function to synthesize test cases.

# Recap



Who gives the input?

# Fuzz testing

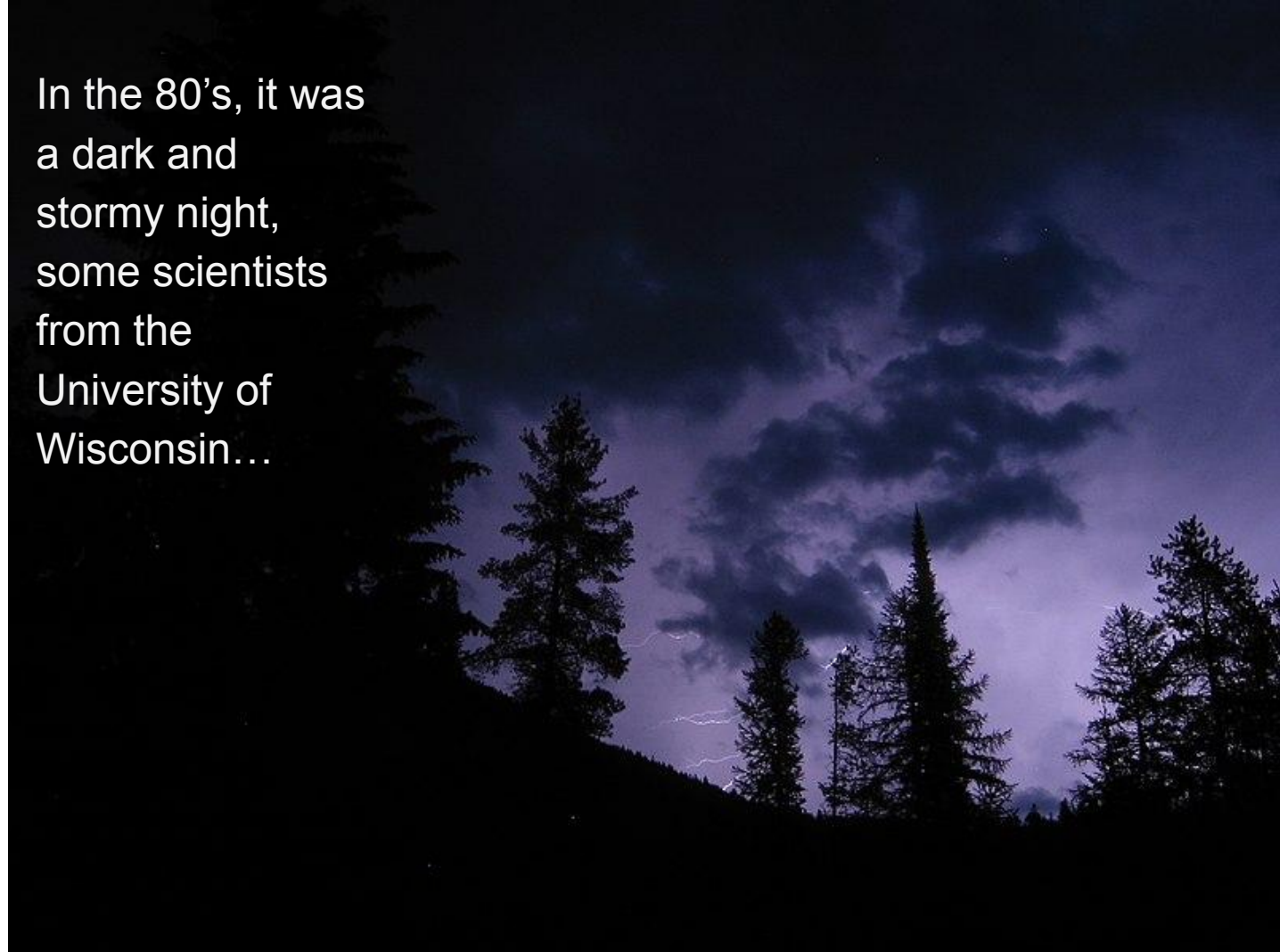


*Let's put a monkey (aka the fuzzer)*

# Origin of Fuzzing

In the 80's, it was  
a dark and  
stormy night,  
some scientists  
from the  
University of  
Wisconsin...

Image credits:  
wikipedia



# **An Empirical Study of the Reliability of UNIX Utilities**

*Barton P. Miller*  
*bart@cs.wisc.edu*

*Lars Fredriksen*  
*L.Fredriksen@att.com*

*Bryan So*  
*so@cs.wisc.edu*

## **Summary**

Operating system facilities, such as the kernel and utility programs, are typically assumed to be reliable. In our recent experiments, we have been able to crash 25-33% of the utility programs on any version of UNIX that was tested. This report describes these tests and an analysis of the program bugs that caused the crashes.

## **Content Indicators**

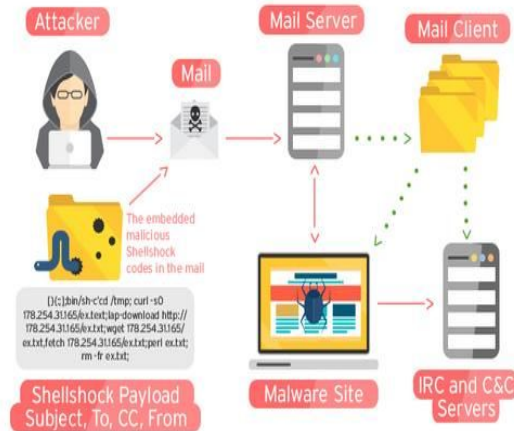
# Security is often a software issue.

In Deloitte's 2007 Global Security Survey, 87 percent of survey respondents cited poor software development quality as a top threat in the next 12 months.



CVE-2014-0160

Heartbleed



CVE-2014-6271

Shellshock



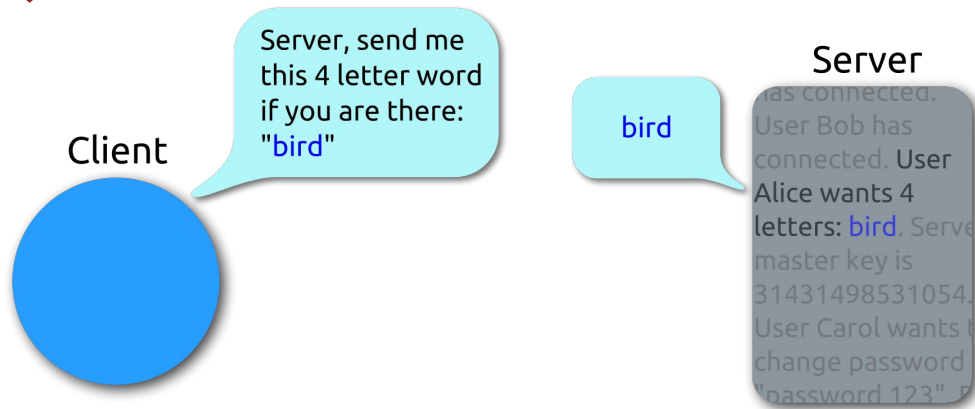
Multiple CVE

# Heartbleed

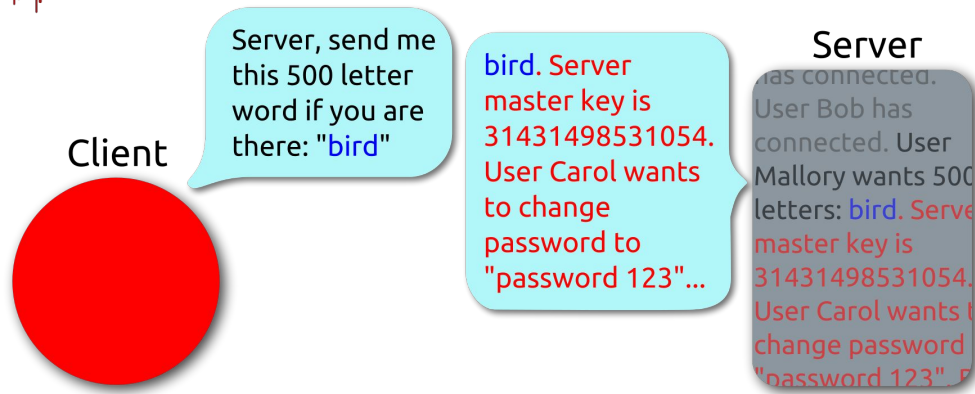
- Heartbleed is a security bug in the OpenSSL cryptography library, which is a widely used implementation of the Transport Layer Security (TLS) protocol.
- Details can be found at: <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160>



## Heartbeat – Normal usage



## Heartbeat – Malicious usage





# Heartbleed

## The Bug:

```
memcpy(bp, pl, pl_length);
```

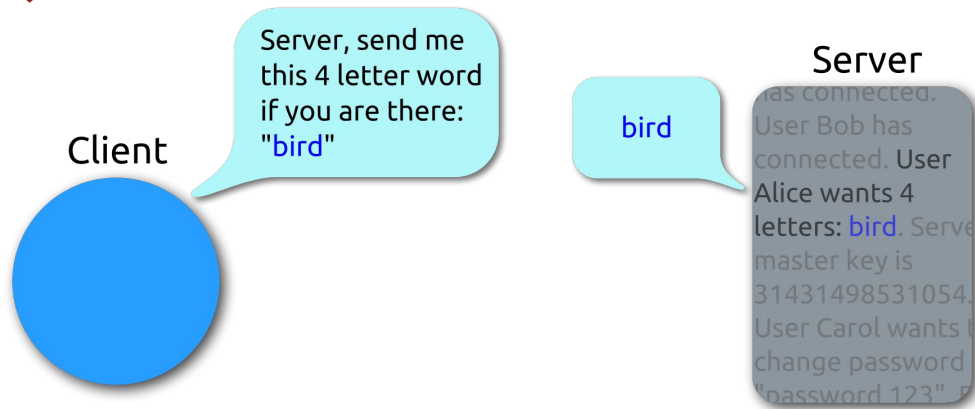
- where bp is a pointer to buffer storing the result, pl is a pointer to where the data the client requested for is, and pl\_length is a number that says how big pl is.

## The Fix:

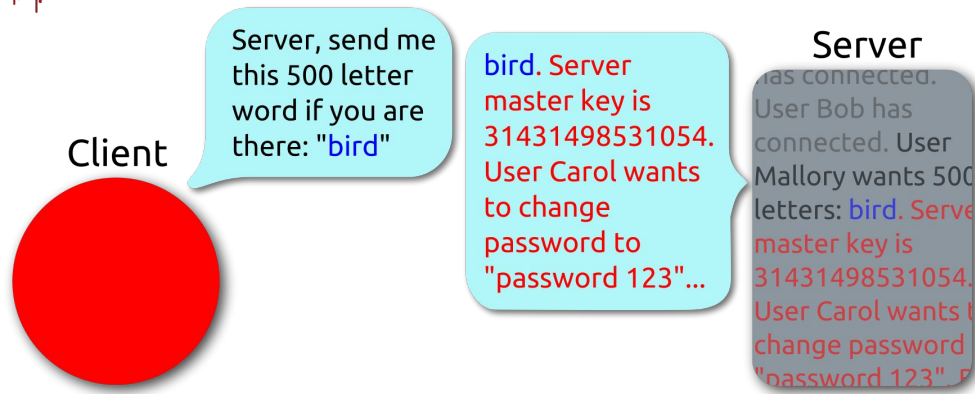
```
if (1 + 2 + pl_length + 16 > s->s3->rrec.length)
return 0;
/* silently discard per RFC 6520 sec. 4 */
```



## Heartbeat – Normal usage



## Heartbeat – Malicious usage



# Common errors causing system crashing

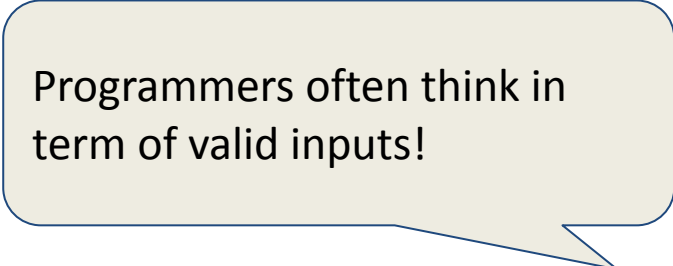
- Arrays and Pointers
- Not checking return code of a function
- Not checking copy length overflows
- signed vs unsigned problem
- ....

# Security Relevant Testing (Fuzzing)

Fuzzing or fuzz testing is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program.

- Very long or completely blank strings
- Maximum and minimum values for integers
- Null characters, new line characters, semi-colons
- Format string values (%n, %s, etc.)

Fuzzing aims to identify test inputs which reveal exploitable vulnerabilities.



Programmers often think in term of valid inputs!

# Why Fuzzing

- A study found that one-quarter to one-third of all utilities on every UNIX system that the evaluators could get their hands on would crash in the presence of random input.
- A study that looked at 30 different Windows applications found that 21% of programs crashed and 24% hung when sent random mouse and keyboard input, and every single application crashed or hung when sent random Windows event messages.
- A study found that OS X applications, including Acrobat Reader, Apple Mail, Firefox, iChat, iTunes, MS Office, Opera, and Xcode, were even worse than the Windows ones.

# Baseline fuzzing - Random

```
function random_fuzz() {
  let res = "";
  // strings of any length between 0 and 1024
  let length = Math.floor(Math.random() * 1024);
  //generate a random character at each location of the string
  for (let i = 0 ; i < length ; i++) {
    //generate a character between ASCII 32 and 128
    let c = String.fromCharCode(Math.floor(Math.random() * 96) +
32);
    res = res + c;
  }
  return res;
}
console.log(random_fuzz())
```

```
.u/m:SKx[-:\3t3b?z87f)6'35!t1Y``fGR[_JS:&9^RSIA
6svw8G~!%!%_1HA9b\~a"cw$<|$2Gc^8&A*c$eT#wd<QUd}
Ylua>yDxh:=NC^3jI~KMivT'.j{Cp%cDco\2aG/cw3d$<Ih
|vIm,_4d,oE8;nV!VXc:[1T,X{F)pIh8=+_ [Xtm=_2X:9EW
U_Oo*$| ^0}nw~Dr1cQ(<dr}Y7/1H&mS$7?fX(F()j)^9#j8
.m2`<v_]="PAI{qvD+yxPusDx!/5ZXMff'b"Q,.pt7K\\?i
7Yj
8Tx)/E'`M[]=v{>GtDbijU%mwEZB?PG]Fo{{6}jXRD[0(<z
gbsahlJ!D&m\`aGQ8ehDbQN9>^C<I[AZ2;s'%_oj1}#RuCj
~i=O"vG]Q!FJ`UP:>{Y|]o\p#7zi'8\Ck@!vC./j:C.)iG
x>Q]@zfffZ24]lZw4L7BKk0dA{/
Z6`3v:NN8:h:@/qOC.oq{O^kgYD(#;|@_`i,l!x2P14G(|T
+KL!bml:<[P+Sh!*]JY|\y\dL[
```

# Baseline fuzzing - Random

```
function random_fuzz() {  
  let res = "";  
  // strings of any length between 0 and 1024  
  let length = Math.floor(Math.random() * 1024);  
  //generate a random character at each location of the string  
  for (let i = 0 ; i < length ; i++) {  
    //generate a character between ASCII 32 and 128  
    let c = String.fromCharCode(Math.floor(Math.random() * 96) +  
32);  
    res = res + c;  
  }  
  return res;  
}  
console.log(random_fuzz())
```

```
.u/m:SKx[-:\3t3b?z87f)6'35!t1Y``fGR[_JS:&9^RSIA  
6svw8G~!%!* 1HA9h\~a"cw$<!$2Gc^8&A*c$eT#wd<QUd}  
Y1ua>yDx! cw3d$<Ih  
|vIm,_4d, i=_2X:9EW  
U_Oo*$|^() ]j)^9#j8  
.m2`<v_] = pt7K\\?i  
7Yj  
8Tx)/E'`M XRD[0(<z  
gbsahlJ!I j1}#RuCj  
~i=O"vG]{ j:C.)iG  
x>Q]@zff?  
Z6`3v:NN& . . . . . :2P14G(|T  
+KL!bml:<[P+Sh!*]JY|\y\dL[
```



Most of the common programs reject invalid inputs.

# Leverage on valid inputs

- Mutation-based fuzzing
- Generation-based fuzzing

# Mutation-based fuzzing

Assuming the following is one of the valid input

ISTDisApillarInSUTDbutItsNameiSGoingToChange

The fuzzer randomly choose one character to update

ISTDisApillar**y**nSUTDbutItsNameiSGoingToChange

Or randomly trim off some characters

ISTDisApillarInSUTDbutItsName



# Common Mutation Operations

- Flipping a bit
- Trimming
- Swapping characters
- Inserting characters

We can random choose one of the operations to apply for each iteration

## Cohort Exercise (Graded)

Given an input string, implement a mutation operator that choose a random position in the string and swap the adjacent characters. Meaning if `SUTD` is an input string and 2 is chosen as the random position, the output should be `SUDT`. Careful about the string length bound check.

# Generation-based fuzzing

Leverage on the rules of valid inputs, to generate random valid inputs for fuzzing.

# Formal Grammar

For example, consider a language of arithmetic expressions which consists of  $+$ ,  $-$ ,  $*$ ,  $/$  as the operators and numbers as operands. In addition, we allow to use  $()$  to disambiguate ambiguous terms.

For instance,

$$5 + 2 * 3$$

# Formal Grammar

$S ::= \text{Expr}$

$\text{Expr} ::= \text{Expr} + \text{Term} \mid \text{Expr} - \text{Term} \mid \text{Term}$

$\text{Term} ::= \text{Term} * \text{Factor} \mid \text{Term} / \text{Factor} \mid \text{Factor}$

$\text{Factor} ::= -\text{Integer} \mid (\text{Expr}) \mid \text{Integer} \mid \text{Integer}.\text{Integer}$

$\text{Integer} ::= \text{Digit} \mid \text{IntegerDigit}$

$\text{Digit} ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

# Formal Grammar

$S ::= \text{Expr}$

← A rule

$\text{Expr} ::= \text{Expr} + \text{Term} \mid \text{Expr} - \text{Term} \mid \text{Term}$

← An alternative

$\text{Term} ::= \text{Term} * \text{Factor} \mid \text{Term} / \text{Factor} \mid \text{Factor}$

$\text{Factor} ::= -\text{Integer} \mid (\text{Expr}) \mid \text{Integer} \mid \text{Integer}.\text{Integer}$

$\text{Integer} ::= \text{Digit} \mid \text{IntegerDigit}$

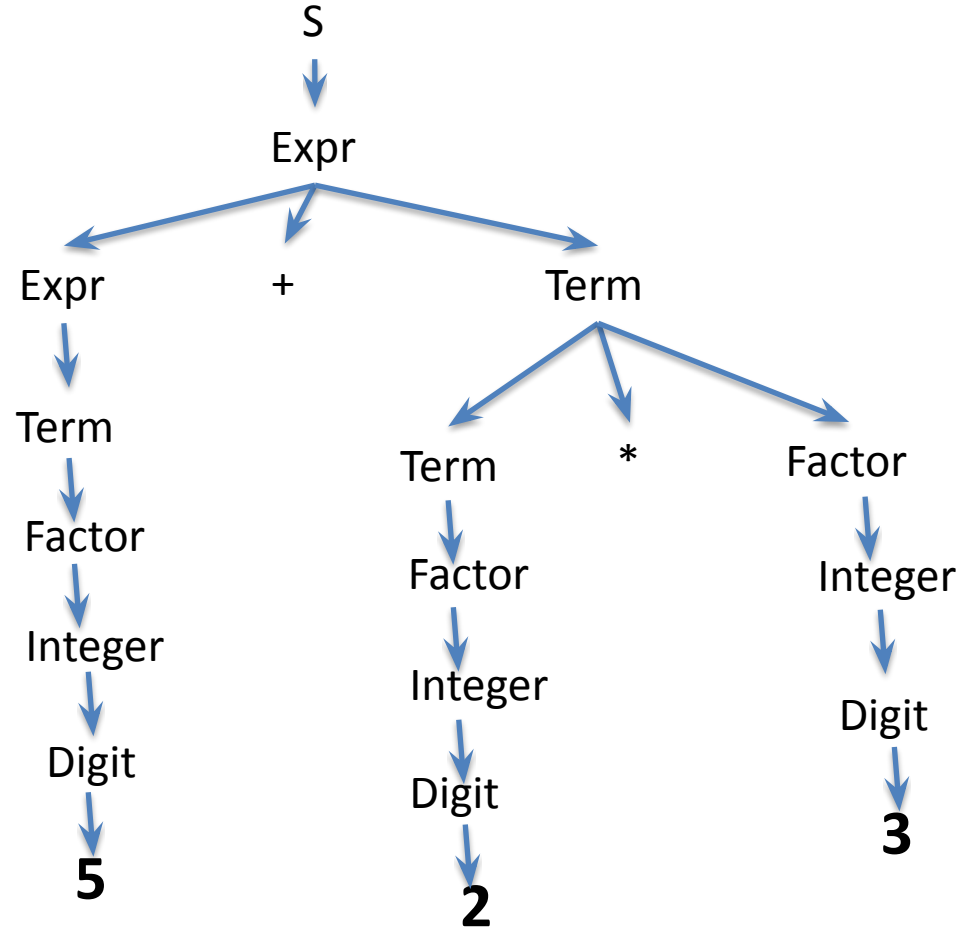
$\text{Digit} ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

← A non-terminal

← A terminal

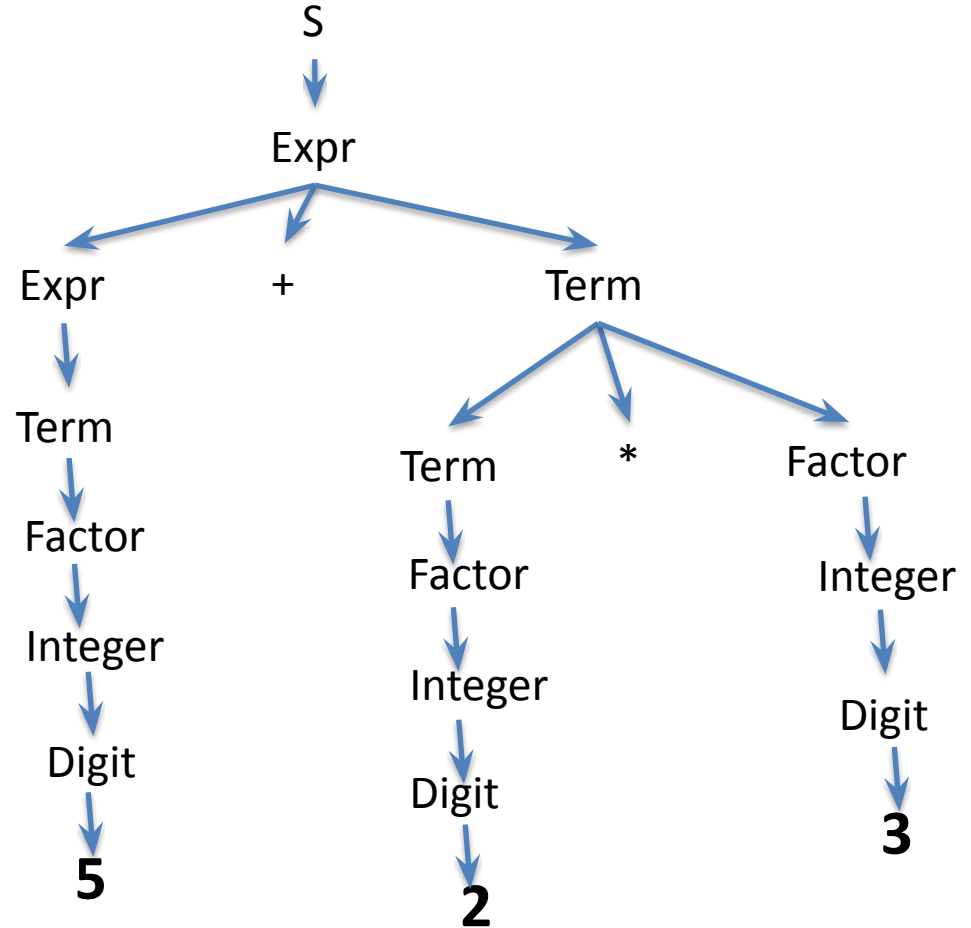
# Parse Tree

*Parses 5+2\*3*



# Parse Tree

*Generates 5+2\*3*

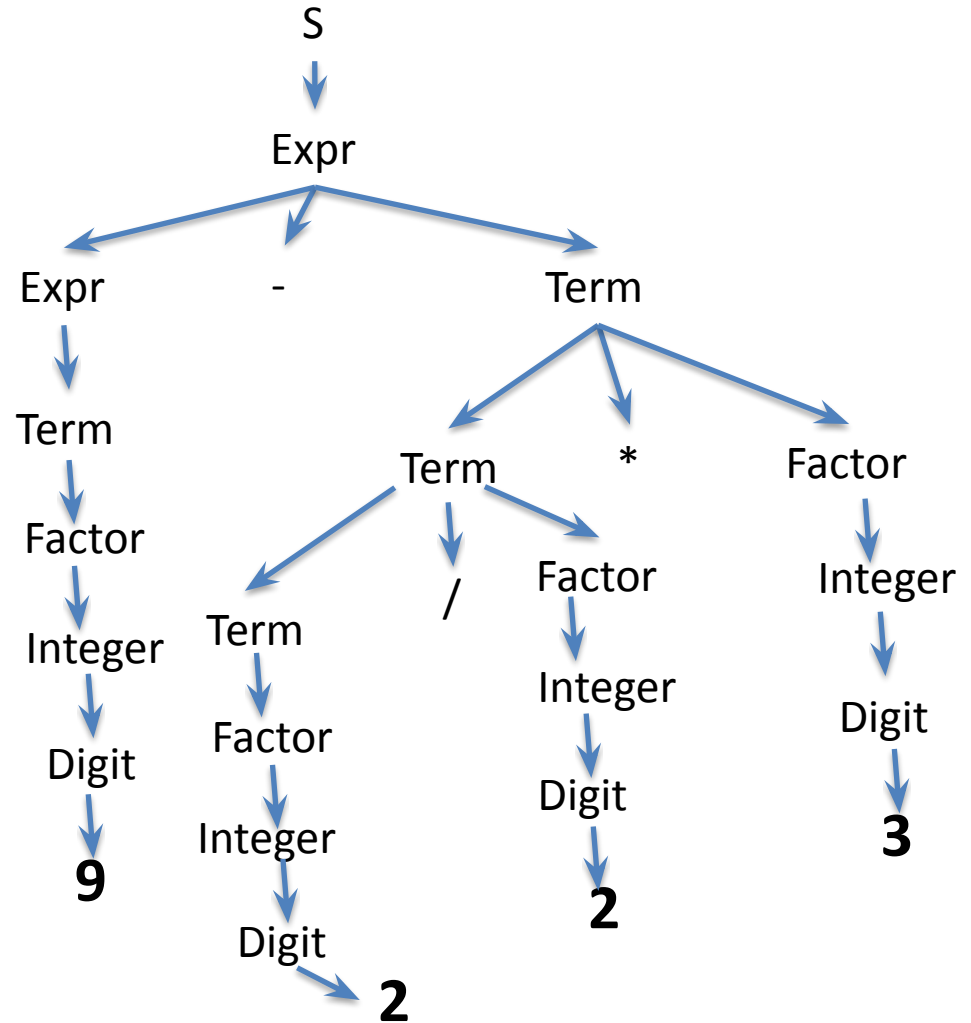




# The idea

- Starting from the starting rule's nonterminal, repeat
  - Randomly choose one grammar rule given the current non-terminal to generate a sub term (sub parse tree).
  - Take note of
    - Do not over expand with the recursive alternative.
    - Do not always expand the same sub-tree / node.

***Generates 9-2/2\*3***



# Cohort Exercise (Graded)

Use JavaScript to implement a fuzzer that will randomly generate inputs to the calculator conforming to the grammar. For now, you can hardcode the expression grammar.

Hint: Start with the initial rule  $S ::= \text{Expr}$  and at each point, apply a rule at random. For example, randomly choose any of the rules  $\text{Expr} ::= \text{Term}$ ,  $\text{Expr} ::= \text{Expr} + \text{Term}$  or  $\text{Expr} - \text{Term}$  in the next step. Continue until a valid expression for the calculator is obtained. Make sure you do not expand the rules forever to avoid infinite loop.

# Pros and Cons of fuzzing

## Pros

- Can provide results with little effort
- Can reveal bugs that were missed in a manual audit
- Provides an overall picture of the robustness of the target software

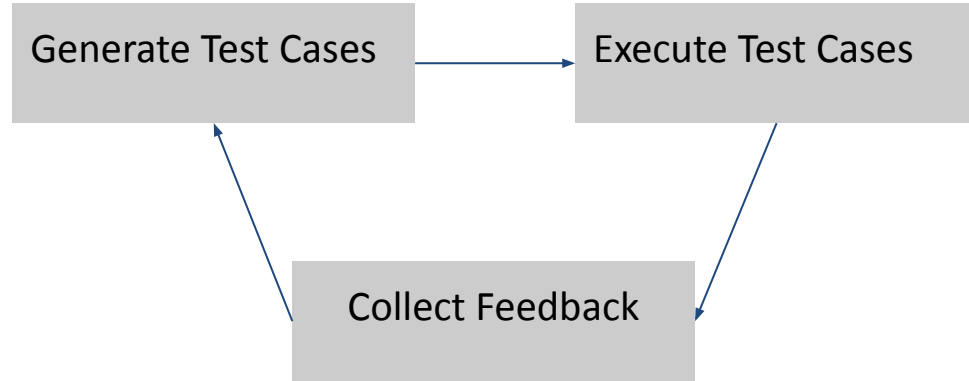
## Cons

- Will not find all bugs
- The crashing test cases that are produced may be difficult to analyse, as the act of fuzzing does not give you much knowledge of how the software operates internally
- Programs with complex inputs can require much more work to produce a smart enough fuzzer to get sufficient code coverage

# Feedback-based Fuzzing

## What feedback?

One common way to measure test effectiveness is to measure the code coverage.



The idea is to tune the test case generation problem into an optimization problem.

## Exercise (Not Graded)

Recall that in Jest, we can generate the coverage report in text or html. Note that it can also generate json report. With it, we can feed the json report back to the fuzzer to generate a better set of tests to increase the code coverage.

Discuss among your team members to think of a possible implementation to incorporate the feedback-based fuzzing

# API Fuzzing

Fuzz testing can be applied to API testing in several aspect

- Fuzzing the low level request - generates random bytes as HTTP requests to test the robustness of the API service.
- Fuzzing the routes - through fuzzers, we could generate sequences of random valid / invalid requests to the test the API routers.
- Fuzzing the high level request - generate GET/POST/DEL/PUT HTTP requests by fuzzing HTTP parameters or form parameters.

# UI Fuzzing

Fuzz testing can be applied to UI too.

- Fuzzing the UI element event handler by generating random input actions.
- Fuzzing the inputs to the HTML forms.