Faculty of Computers, Informatics and Microelectronics

Technical University of Moldova

SI

Laboratory work # 4

Timing attack experiment

Author:
Petru Negrei

Supervisor:
A. Railean

November 2014

1 Introduction

1.1 Objective

- Create a test environment to see whether a timing attack is feasible
- Analyze and document your findings
- Compare your results with those of your colleagues
- Write a constant-time string comparison function

1.2 Definition

In cryptography, a timing attack is a side channel attack in which the attacker attempts to compromise a cryptosystem by analyzing the time taken to execute cryptographic algorithms. Every logical operation in a computer takes time to execute, and the time can differ based on the input; with precise measurements of the time for each operation, an attacker can work backwards to the input.

2 Description

The following program analyzes necessary time to compare different strings and time needed for it.
We check the following cases:

- Compare two strings with == where the characters are the same until late in the string.
- Compare two strings with == where the characters differ early in the string.
- Same as 1 but with a ruby implementation of secure_compare.
- Same as 2 but with a ruby implementation of secure_compare.

We do 1000 tests of each case.

Due to the fact that the difference is not visible at small sizes, I implemented the following example that uses the 'TrurlAndKlapaucius' repeated 1000 times.

Ruby's benchmark module rounds off numbers quite agressively, so to see anything for the == cases, refer to the 'real' measurement.

In the first case, you should observe an order of magnitude or greater difference in using just ==, while you get measurements for both secure_compares that are the same to within the margin of error.

In the second case, the == are much closer, but should still be distinguishable.

You'll also notice that secure_compare is *much* slower and not super consistent, especially the pure Ruby one. This is the price you pay for not having a language-level secure byte comparison primitive.

3 Implementation

```ruby
require "benchmark"
require "digest"

def secure_compare(a, b)
  check = a.bytesize ^ b.bytesize
  a.bytes.zip(b.bytes) { |x, y| check |= x ^ y.to_i }
  check == 0
end
```

```
  def compare(base_str)
    early_str = base_str.clone
    early_str[0] = 'z'

    late_str = base_str.clone()
    late_str[late_str.length-1] = '!'

    Benchmark.bmbm do |b|
       b.report("==, early fail") { 1000.times { base_str == early_str } }
       b.report("==, late fail") {1000.times { base_str == late_str } }
       puts ""
       b.report("Ruby secure_compare, 'early'") { 1000.times { secure_compare(base_str, early_str) } }
       b.report("Ruby secure_compare, 'late'") { 1000.times { secure_compare(base_str, late_str) } }
       puts ""
       b.report("SHA512-then-==, 'early'") { 1000.times { Digest::SHA512.digest(base_str) ==
           Digest::SHA512.digest(early_str) } }
       b.report("SHA512-then-==, 'late'") { 1000.times { Digest::SHA512.digest(base_str) ==
           Digest::SHA512.digest(late_str) } }
     end
  end

  puts ""
  puts "==== Short text ===="
  puts ""
  compare('TrurlAndKlapaucius'*50)

  puts ""
  puts "==== Long text ===="
  puts ""
  compare('TrurlAndKlapaucius'*1000)
```

```
==== Short text ====
-----------------------------------------------------------
                           user   system   total     real
==, early fail          0.000000 0.000000 0.000000 ( 0.000115)
==, late fail           0.000000 0.000000 0.000000 ( 0.000158)
Ruby secure_compare, 'early' 0.220000 0.000000 0.220000 ( 0.224304)
Ruby secure_compare, 'late'  0.220000 0.000000 0.220000 ( 0.232572)
SHA512-then-==, 'early' 0.010000 0.000000 0.010000 ( 0.013119)
SHA512-then-==, 'late'  0.020000 0.000000 0.020000 ( 0.013085)


==== Long text ====
-----------------------------------------------------------

                           user   system   total     real
==, early fail          0.000000 0.000000 0.000000 ( 0.000117)
==, late fail           0.000000 0.000000 0.000000 ( 0.001374)
Ruby secure_compare, 'early' 4.360000 0.010000 4.370000 ( 4.384387)
Ruby secure_compare, 'late'  4.370000 0.010000 4.380000 ( 4.389848)
SHA512-then-==, 'early'    0.190000 0.000000 0.190000 ( 0.190479)
SHA512-then-==, 'late'     0.190000 0.000000 0.190000 ( 0.191458)
```

4 Conclussion

After making this laboratory work I learn more about timing attacks, and perform different comparision on strings to see the diference between recorded times needed to perform the 'comp' operation. Also implemented constant-time string comparison and saw why it is necessary. Analyzing the preliminary results, the time received for short strings (usually used for passwords and normal text) is too small and random to draw some conclusion about the used information, but if we use big strings the difference in time is visible for unsave comparision of strings.