Implementing The Exponential Backoff Algorithm To Thwart Dictionary Attacks

George Watkins, 2011-17-11

Introduction

Recently there was a forum post regarding using the exponential backoff algorithm to prevent or at the very least slow down dictionary attacks. A dictionary attack is when a perpetrator attacks a weak system or application by cycling through a common list of username and password combinations. If were to leave a machine connected Internet with SSH open for any length of time, it wouldn't take long for an attacker to come along and start hammering the machine. He'll go through his list until he either cracks an account, gets blocked, or hits the bottom of his list. The attacker has a distinct advantage when he can send unabated requests to the system or application he is attacking.

The purpose of the exponential backoff algorithm is to increase the time between subsequent login attempts exponentially. Under this scenario, a normal user wouldn't be able to type or navigate faster than the minimum lockout period and probably has a very low likelihood of ever hitting the limit. In contrast, if someone was to make a number of repetitive requests in a small timeframe, the time he would be locked out would rise exponentially.

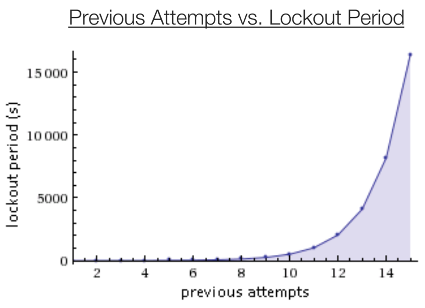
Exponential Backoff Algorithm

The exponential backoff algorithm is mathematically rather simple. The lockout period is calculated by raising 2 to the power of the number of previous attempts made, subtracting 1, then dividing by two. The equation looks like this:

$$E(c) = \frac{1}{2}\left(2^c - 1\right)$$

The effect of this calculation is that the timeout for the lockout period is small for the first series of attempts, but rise very quickly given a burst of attempts. If we assemble a table and a plot of previous attempts vs. lockout period, the accumulation becomes apparent with each subsequent attempt doubling the lockout period. If an attacker were to hit an application with 20 attempts in a short window, they would be locked out almost indefinitely or at least to the max lockout period, which we'll discuss shortly.

Attempts	Lockout (s)	Lockout (h, m, s)
1	0	0s
2	2	2s
3	4	4s
4	8	8s
5	16	16s
6	32	32s
7	64	1m 4s
8	128	2m 8s
9	256	4m 16s
10	512	8m 32s
11	1024	17m 4s
12	2048	34m 8s
13	4096	1h 8m 16s
14	8192	2h 16m 32s
15	16384	4h 33m 4s



Calculating Integer Powers of 2

A number of standard TCL math functions are disabled in iRules because of their ability to consume immense CPU resources. While this does protect the average iRule developer from shooting himself in the leg with them, it limits the ability to perform more complex operations. One function in particular would make implementing the exponential backoff algorithm much easier: pow(). The pow() provides the ability to perform exponentiation or raising a number (the base) to the power of another (the exponent). While we would have needed to write code to perform this functionality for bases larger than 2, it is actually a rather easy operation using an arithmetic shift (left in this case). In TCL (like many other modern languages) uses the << operator to perform a left shift (multiplication by 2) and the >> operator to employ right shift (division by 2). This works because all of the potential lockout periods will be a geometric sequence of integer powers of 2. Take a look at the effect of a left shift on integer powers of two when represented as a binary number (padding added to represent an 8-bit integer):

Binary number	Decimal number	TCL left shift (tclsh)
0 0 0 0 0 0 0 1	1	% expr {1 << 0} => 1
0 0 0 0 0 0 1 0	2	% expr {1 << 1} => 2
0 0 0 0 0 1 0 0	4	% expr {1 << 2} => 4
0 0 0 0 1 0 0 0	8	% expr {1 << 3} => 8
0 0 0 1 0 0 0 0	16	% expr {1 << 4} => 16
0 0 1 0 0 0 0 0	32	% expr {1 << 5} => 32
0 1 0 0 0 0 0 0	64	% expr {1 << 6} => 64
1 0 0 0 0 0 0 0	128	% expr {1 << 7} => 128

Even if the power function were available, a bitwise operation is almost certainly the most efficient way to perform this calculation. Sometimes the most obvious answer is not necessarily the most efficient. Had we not ran into this small barrier, this solution probably would not have emerged. Check out the link below for a complete list of available math functions, operators, and expressions in iRules.

List of TCL math function available in iRules

Implementing the Algorithm in iRules

Now that we know how to calculate integer powers of 2 using an arithmetic shift, the rest of the equation implementation should be straightforward. Once we replace pow() function with a left shift, we get an equation that looks as such:

```
set new_lockout [expr (((1 << $prev_attempts)-1)/2)]
```

Now when we run through a geometric series in TCL we'll get "almost" the number in the tables above, but they'll all have a value of one less than expected because we always divide an odd numerator resulting in a float that is truncated when converted to an integer. When this process takes place, the digits after the decimal place are truncated and only the integer portion remains. Given a random distribution of floats truncated to integers there would normally be an even distribution of those rounded "correctly" and "incorrectly." However in this series all of the solutions end with a decimal value of .500000 and are therefore rounded "incorrectly".

Previous attempts	**Calculated (float)**	**Calculated (integer)**
0	0	0
1	0.5	0
2	1.5	1
3	3.5	3
4	7.5	7
5	15.5	15

We could use the equation listed above, but our numbers would not line up with our projections. In order to get more accurate numbers and save additional CPU cycles, we can further reduce the equations to this:

$$E(c) = 2^{c-1}$$

Or as TCL like this:

```
set new_lockout [expr (1 << ($prev_attempts-1))]
```

Now we've got something that is super fast and serves our purposes. It would be virtually impossible to overload a box with this simple operation. This is a far more efficient and elegant solution than the originally proposed power function.

Maximums, Minimums, and State

The benefit of the exponential backoff algorithm is that it increased exponentially when probed repeatedly, but this is also the downside. As the timeout grows exponentially you can potentially lock out the user permanently and quickly fill the memory allocated for a 32-bit integer. The maximum value of a 32-bit integer that can be stored in iRules is 2,147,483,647, which equates to 68 years, 1 month, and change (far longer than a BIG-IP will be in service). For this reason, we'll want to set a maximum lockout period for a user so that we don't exceed the memory allocation or lock a user out permanently. We recommend a maximum lockout period of anything from an hour (3600s) to a day (86,400s) for most use cases. The maximum lockout period is defined by the static max_lockout variable in the RULE_INIT event.

On the flipside, you'll notice that there is a case where we get a lockout period of zero for the first request, which will cause a timing issue for the iRule. Therefore we need to establish some minimum for the lockout period. During our tests we found that 2 seconds works well for normal browsing behaviors. You may however decide that you never want anyone submitting faster than every 10 seconds and you'd like the added benefit of the exponential backup, therefore you would change the static min_lockout value in RULE_INIT to 10 (seconds).

Lastly, we use the session table to record the number of previous attempts and the lockout period. We define the state table name as a static variable in the CLIENT_ACCEPTED event and use a unique session identifier consisting of the client's IP and source port to track each session's behavior. Once we receive a POST request, we'll increment the previous attempts counter and the calculate a new timeout (lockout period) for the table entry. Once enough time has passed, the entry will timeout in the session table and the client may submit another POST request without restriction.

The Exponential Backoff iRule

Once we bring all those concepts together, we arrive at something like the iRule listed below. When applied to an HTTP virtual server, the exponential backoff iRule will count the POST requests and prevent users from firing them off two quickly. If a user or bot issues two tightly coupled POST requests they will be locked out temporarily and receive an HTTP response advising them to slow down. If they continue to probe the virtual server, they will be locked out for the next 24 hours on their 18th attempt.

```
when RULE_INIT {
  set static::min_lockout 2
  set static::max_lockout 86400
  set static::debug 1
}

when CLIENT_ACCEPTED {
  set static::session_id "[IP::remote_addr]:[TCP::remote_port]"
  set static::state_table "[virtual name]-exp-backoff-state"
}
```

```

when HTTP_REQUEST {
  if { [HTTP::method] eq "POST" } {
    set prev_attempts [table lookup -subtable $static::state_table $static::session_id]

    if { $prev_attempts eq "" } { set prev_attempts 0 }

    # exponential backoff - http://en.wikipedia.org/wiki/Exponential_backoff
    set new_lockout [expr (1 << ($prev_attempts-1))]

    if { $new_lockout > $static::max_lockout } {
      set new_lockout $static::max_lockout
    } elseif { $new_lockout < $static::min_lockout } {
      set new_lockout $static::min_lockout
    }

    table incr -subtable $static::state_table $static::session_id
    table timeout -subtable $static::state_table $static::session_id $new_lockout

    if { $static::debug > 0 } {
      log local0. "POST request (#[expr ($prev_attempts+1)]) from $static::session_id received during lockout period, updating lockout to ${new_lockout}s"
    }

    if { $prev_attempts > 1 } {
      # alternatively respond with content - http://devcentral.f5.com/wiki/iRules.HTTP__respond.ashx
      set response "<html><head><title>Hold up there!</title></head><body><center><h1>Hold up there!</h1><p>You're"
      append response " posting too quickly. Wait a few moments are try again.</p></body></html>"

      HTTP::respond 200 content $response
    }
  }
}
```
CodeShare: Exponential Backoff iRule

Conclusion

The exponential backoff algorithm provides a great method for thwarting attacks that rely on heavy volume of traffic directed at a system or application. Even if an attacker were to discover the minimum lockout period, they would still be greatly slowed in their attack and would likely move on to easier targets. Protecting an application or system is similar to locking up a bike in many ways. Complete impenetrable security is a difficult (and some would say impossible) endeavor. We can however implement a heavy gauge U-lock accompanied by a thick cable to protect the frame and other various expensive components. A perpetrator would have to expend an inordinate amount of energy to compromise our bike (or system) relative to other targets. The greater the difference in effort (given a similar reward), the more likely it will be that the attacker will move on to easier targets. Until next time, happy coding.

F5 Networks, Inc. | 401 Elliot Avenue West, Seattle, WA 98119 | 888-882-4447 | f5.com

F5 Networks, Inc. Corporate Headquarters info@f5.com	F5 Networks Asia-Pacific apacinfo@f5.com	F5 Networks Ltd. Europe/Middle-East/Africa emeainfo@f5.com	F5 Networks Japan K.K. f5j-info@f5.com