TFTP Server via iRules

Jason Rahm, 2009-18-03

Last week in the forums, a user asked an intriguing question:

"Is it possible to write a simple TFTP server as an iRule to return static content? I see some examples for load balancing TFTP requests, but not for actually acting as a TFTP server. I have a very small and static TFTP file that I need to distribute, and rather than running a dedicated TFTP server it would be far better if it could be deployed as an iRule." (Forum thread is here)

My initial reaction to these types of questions is usually "Yep, you can do that with an iRule." This time, however, my initial thought was "probably not." Oh me of little faith...

TFTP Basics

...so I looked at the TFTP standard (defined in RFC 1350) and it didn't really look that complicated. Much simple and smaller (by design) than TFTP, for a user to grab a file really doesn't take that much work. There are five types of packets supported by TFTP: Read Request (RRQ, opcode 1), Write Request (WRQ, opcode 2), Data (DATA, opcode 3), Acknowledgement (ACK, opcode 4), and Error (ERROR, opcode 5). The formats for the messages we'll address in this tech tip are in the table below.

TFTP Message Format (Read & Data)

Read Request	opcode (RRQ=1) (2 bytes)	file name (N bytes)	zero byte	mode (N bytes)	zero byte
Data Message	opcode (DATA=3) (2 bytes)	block number (2 bytes)	data (0-512 bytes)		

- Bonus #1, the TCL wiki has a working example of a TFTP server here.
- Bonus #2, TMM handles the UDP communication, so we can focus on the TFTP protocol message handling.

The Read Request

The first thing we need to determine is what type of request the client is sending. Anything but a read should be rejected (or if you want to be cleaner, you can use the error packet and respond with human readable information for the client). Since the intent here is to fill requests for a small file and nothing more, there really is no benefit to providing the functionality. It's more overhead, and really, do you want users attempting to write files to your LTM? Didn't think so. So we can scan the UDP data for the opcode.

```
binary scan [UDP::payload] xc opcode
```

Binary scan in this case is scanning the UDP payload, skipping the first byte, storing the contents of the second byte as an 8-bit signed integer in the variable opcode. We know that the opcode field is two bytes, but since there is no valid opcode that would set any bits in the first byte, there is no reason to scan it. Now that we have the opcode, we can reject anything that isn't a request, or for troubleshooting purposes, add a log statement during development to trap the other options.

```
switch $opcode {
  1 {
    if { $::debug } { log local0. "Read request OK" }
  }
  2 {
    if { $::debug } { log local0. "Write request not supported here" }
    reject
  }
```

```
  3 {
    if { $::debug } { log local0. "Data receipt not supported here" }
    reject
  }
  4 {
    if { $::debug } { log local0. "Ack from client received" }
  }
  5 {
    if { $::debug } { log local0. "Error: $string" }
  }
  default {
    if { $::debug } { log local0. "Opcode $opcode is invalid" }
    reject
  }
}
```

I'm sure it's obvious by now, but I do a lot of logging during development, and I like the control method with a debug variable so you can enable/disable with one configuration change instead of commenting in/out multiple lines. OK, we have the opcode. Now, we want to take a look at the next two meaningful fields, the filename and the mode. Since the opcode is in the first two bytes, we don't care about those anymore, that decision has been made. We also know that the the filename can be any number of bytes, followed by a zero byte, followed by the mode (also any number of bytes) followed by another zero byte. Well, we can issue another binary scan to grab the data we need:

```
binary scan [UDP::payload] xxa* string
set file [lindex [split $string \000] 0]
set mode [lindex [split $string \000] 1]
```

xxa* dumps everything after the opcode into the string variable. Splitting on the zero byte puts the filename into the first list item and the mode into the second list item, so these can be extracted and set into variables, appropriately named file and mode. Now, we can provide some level of validation, such as making sure the mode is valid, making sure the file they are requesting is correct, etc. You could respond with the file contents regardless of the specified file, but it's a nice easy way to eliminate a bulk of the naughty behavior attempts.

```
if { ($mode == "octet" || $mode ==  "netascii") && !($file eq "") } {
  if { $::debug } { log local0. "File name $file specified, mode $mode is valid" }
} else {
  if { $::debug } { log local0. "File name not specified ($file) or mode $mode is invalid" }
}
```

OK, read request analyzed, now we can move on to passing some data back to the client.

The Data Message

This is the part that sourced my initial doubt. However, the creative juices started stirring through a typical (of the DC Community anyway) collaborative effort (thanks jquinby and natty76!) A really small file as used in the initial testing can be served out of an array, but for larger files, an external class is a good idea. Getting the formatting correct was a chore, but I finally found a conversion script here that maintained the CRLF formatting for the file (I dumped the UTF-8 hex output from the TFTP RFC in the external class). Practically, you could put small files in the same class and have <filename>, <encoded string> pairs, but I chose to maintain a separate class for each file and switch on filename:

```
switch $file {
  "test.txt" {
            set flen [string length $::tftp_file_contents]
     set total_blocks [expr {$flen / 1024.0}]
     if { [lindex [split $total_blocks "."] 1] > 0 } {
       set total_blocks [expr [lindex [split $total_blocks "."] 0] + 1 ]
       if { $::debug } { log local0. "block isn't integer, incrementing for final block" }
```

```
    } else {
  set total_blocks [lindex [split $total_blocks "."] 0]
  if { $::debug } { log local0. "Block is integer, no increment necessary" }
    }
    set str_index 0
    for { set x 1 } { $x <= $total_blocks } { incr x } {
  set data [binary format SSH* 3 $x [string range $::tftp_file_contents $str_index [expr {$str_index
  if { $::debug } { log local0. "data = $data, size = [string length $data]" }
  UDP::respond $data
  incr str_index 1024
    }
  }
  default {
  if { $::debug } { log local0. "File not found: $file" }
  }
}
```

Since we need to break the file into 512 byte blocks, we check the length of the file and store that length in the variable flen. Because the file has been converted to hex, each ascii character when passed through string length is two characters long in hex, so we need to double the 512 to 1024. In the expression (expr {$flen / 1024.0}), the .0 is added to get the floating point representation instead of the integer. This is necessary because we need to know the total blocks to pass to the client, even though the last block is less than 512 bytes. Any block less than 512 bytes is actually the signal to the client that the last block has been received. The logic is kind of kludgey, but I couldn't come up with a better way to round up in TCL (anyone? Please comment.) Now that we have the total blocks, we set a variable to index each block and then iterate through a for loop to respond to the client with each block of data, stopping on the last block which contains less than 512 bytes, thus signalling the client that the transfer is complete. The binary format SS takes the decimal opcode value (3 for data) and the block id (from the for loop variable) and converts them into 16-bit integers in big-endian byte order (per the format required in the data message) and the H* takes the class data per block and stores each byte in high-low order. The string range consumes the class contents from the str_index + 1023, so the block consumed increments through each iteration of the loop.

Conclusion

Aren't iRules cool? I continue to be amazed at the flexibility available in iRules. This rule is pretty basic, and could easily be extended to support multiple files (logic is already there with the switch on filename) and could be more complete in the ack and error handling. The full rule is here in the codeshare. Enjoy!

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