

gressgraph v0.2

Chris Forno (jekor)

October 21, 2008

`gressgraph` helps you visualize your iptables firewall. It acts as a filter, translating your firewall rules from iptables format into Graphviz graphing instructions.

You can create a simple graph of your firewall with:

```
$ iptables -L -vx | ingressgraph > iptables.twopi
$ twopi -Tsvg iptables.twopi > iptables.svg
```

(Use `-Tpng` instead of `-Tsvg` if you want raster output.)

The source code for the program begins here. It's written in Haskell98 and uses Glasgow extensions. It's been tested with GHC 6.8.2.

```
module Main where
import Data.List
```

We'll be using Parsec for parsing the iptables output.

```
import Text.ParserCombinators.Parsec
import Text.ParserCombinators.Parsec.Prim
```

```
gressgraphVersion :: Float
gressgraphVersion = 0.2
```

We need some basic parsers for iptables syntax. These are very permissive, trading simplicity for safety since we don't expect to receive malformed data.

An identifier is any sequence of characters, except for a space or comma.

```
identifier :: Parser String
identifier = many1 $ noneOf " ,\n"
```

Graphviz uses a limited set of ASCII characters for node identifiers. But it allows us to quote any identifier. We'll just quote everything to be safe.

```
quote :: String → String
quote n = "\"" ++ n ++ "\""
```

An iptables target actually describes the action that iptables will take. Don't confuse "target" with "destination". A target can be one of the 4 basic types (ACCEPT, DROP, QUEUE, or RETURN) or the name of another chain.

```

data Target = Accept | Drop | Queue | Return | Chain String
           deriving (Show, Eq)

target :: Parser Target
target = identifier >>= \s →
  case s of
    "ACCEPT" → return Accept
    "DROP"   → return Drop
    "QUEUE"  → return Queue
    "RETURN" → return Return
    _        → return (Chain s)

```

As with targets, there are 4 pre-defined protocols and one for protocol names.

```

data Protocol = TCP | UDP | ICMP | All | Protocol String
           deriving (Show, Eq)

protocol :: Parser Protocol
protocol = identifier >>= \s →
  case s of
    "tcp" → return TCP
    "udp" → return UDP
    "icmp" → return ICMP
    "all" → return All
    _     → return (Protocol s)

```

iptables allows for "extra" options. These are things like destination port, connection state, etc. This is where a lot of the meat of the rule is and is the (relatively) difficult part to parse.

Note that no other extra options are supported at this time and will result in parse errors.

```

data Extra = DPort DPort | CStates [CState] | None
           deriving Eq

extra :: Parser Extra
extra = choice [(try dport >>= return ∘ DPort ),
               (try cstates >>= return ∘ CStates),
               (try (many1 $ noneOf " \n") >>= return None)]

extras :: Parser [Extra]
extras = extra 'sepEndBy' (many1 $ char ' ')

```

A destination port has the form `udp dpt:bootps` or `tcp dpt:10000:10010`.

```

type DPort = (Protocol, String)

```

```

dport :: Parser DPort
dport = do p ← protocol
         spaces >> string "dpt"
         option ' ' (char 's') >> char ':'
         i ← identifier
         return (p, i)

```

A connection state can be NEW, RELATED, ESTABLISHED or INVALID. It allows iptables to determine whether or not to apply a rule by checking the connection tracking history.

```

data CState = New | Related | Established | Invalid
             deriving Eq

cstate :: Parser CState
cstate = identifier >>= λs →
        case s of
          "NEW"       → return New
          "RELATED"   → return Related
          "ESTABLISHED" → return Established
          "INVALID"   → return Invalid
          _           → unexpected "invalid state"

```

The state can be (and often is) a list of states separated by a comma.

```

cstates :: Parser [CState]
cstates = string "state" >> spaces >>
         cstate 'sepBy' (char ',') >>=
         return

```

Printing out the full state name can clutter the graph, so we'll use abbreviations.

```

instance Show CState where
  show s = case s of
    New       → "New"
    Related   → "Rel"
    Established → "Est"
    Invalid   → "Inv"

```

To output an extra option is pretty straightforward. For destination ports, print the protocol and port(s) separated by a colon. For states, print the states separated by commas.

```

instance Show Extra where
  show (DPort (p, ps)) = (show p) ++ ":" ++ ps
  show (CStates ss)    = intercalate "," (map show ss)
  show _               = ""

```

Here's the main unit of our graph: the iptables rule. In the `iptables` output it's almost a CSV line with spaces for delimiters, except for the "extra" information.

```

data Rule = Rule{ packets    :: Integer,
                  bytes      :: Integer,
                  action     :: Target,
                  proto      :: Protocol,
                  options    :: String,
                  inInterface :: String,
                  outInterface :: String,
                  source     :: String,
                  destination :: String,
                  extraOpts  :: [Extra] }

deriving Show

rule :: Parser Rule
rule = spaces >> many1 digit >>= λpackets' →
      spaces >> many1 digit >>= λbytes' →
      spaces >> target >>= λaction' →
      spaces >> protocol >>= λprotocol' →
      spaces >> identifier >>= λoptions' →
      spaces >> identifier >>= λinInt →
      spaces >> identifier >>= λoutInt →
      spaces >> identifier >>= λsource' →
      spaces >> identifier >>= λdest →
      (many $ char ' ') >> extras >>= λextras' →
      newline >>
      return (Rule
        { packets    = (read packets'),
          bytes      = (read bytes'),
          action     = action',
          proto      = protocol',
          options    = options',
          inInterface = inInt,
          outInterface = outInt,
          source     = source',
          destination = dest,
          extraOpts  = extras' })

```

Each rule is graphed as 3 edges:

- The hop between the the source address and the input interface.
- The hop between the input interface and the output interface.
- The hop between the output interface and the destination address.

We can break graphing a chain down into 3 parts.

1. Set the edge parameters.
2. Graph the first hop, with any necessary label (“extra” parameters like destination port).

3. Graph the second and third hop.

```

showRule      :: String → (Color, Rule) → String
showRule _ (c, r) = unlines
  [header,
   from ++ " -> " ++ inI ++ label',
   inI  ++ " -> " ++ outI ++ " -> " ++ to]
  where from = showAddress (inInterface r, source r)
        to   = showAddress (outInterface r, destination r)
        inI  = quote (inInterface r)
        outI = quote (outInterface r)
        extra' = intercalate " " (map show (extraOpts r))
        arrowhead = case action r of
          Drop → " arrowhead=tee"
          _    → " arrowhead=normal"
        header = "edge [color=\"" ++ c ++ "\" " ++
          "fontcolor=\"" ++ c ++ "\" " ++
          arrowhead ++ "]"
        label' = " [label=\"" ++ extra' ++ "\"]"

```

Some addresses have the same name but are conceptually different. A good example is “anywhere”. “Anywhere” means a different thing depending on which interface you’re talking about. Because of this, we treat addresses as an (interface, address) pair and graph them together (separated by an underscore).

```

type Address = (String, String)
showAddress  :: Address → String
showAddress (i, a) = quote (i ++ "_" ++ a)

```

A *Chain* is a named collection of rules. The rules are in order (even though we ignore that for graphing purposes).

```

type Chain = (String, [Rule])

```

A chain is terminated by a newline or the end of file marker.

```

chain :: Parser Chain
chain = do name ← chainHeader
          rules ← manyTill rule (newline [] (eof >> return '\n'))
          return (name, rules)

```

We ignore all of the information in the chain header except for its name.

```

chainHeader :: Parser String
chainHeader = string "Chain " >> identifier >>= λname →
  manyTill anyChar newline >>

```

```

manyTill anyChar newline >>
return name

```

Finally, the iptables output (our input) is a series of chains.

```

chains :: Parser [Chain]
chains = many1 chain

```

To graph a chain, we just graph its rules. We zip up each rule with a color to distinguish it from (most) other rules.

```

showChain          :: Chain → String
showChain (name, rules) = unlines (("// Chain " ++ name) :
                                   map (showRule name) (zip colors rules))
  where n          = length rules
        colors     = take n palette

```

We'll cycle the colors we're using indefinitely for as many rules as we need.

```

palette :: [Color]
palette = cycle spectral

```

spectral is a broad rainbow-like palette from the Graphviz documentation.

```

type Color = String
spectral :: [Color]
spectral = ["#9E0142", "#D53E4F", "#F46D43", "#FDAE61",
           "#FEE08B", "#FFFFBF", "#E6F598", "#ABDDA4",
           "#66C2A5", "#3288BD", "#5E4FA2"]

```

This program is just a simple filter that accepts an iptables dump as input and outputs a Graphviz representation.

```

main :: IO ()
main = getContents >>= graphviz ∘ parseChains

```

parseChains applies the parser we've built up until this point to the string it receives (an iptables dump). If there are any errors, it prints them on stderr (using Parsec's default error messages).

```

parseChains  :: String → [Chain]
parseChains x = case (parse chains "" x) of
  Left err → error (show err) >> []
  Right cs → cs

```

graphviz is responsible for creating the finished output for the `graphviz` program to parse.

First, it tells Graphviz how we want the graph to be drawn. We want a directed graph (`digraph`) with good spacing and color.

Next, it creates an invisible root node to act as a central hub for the graph and connects all the network interface nodes to it.

Finally, it draws out all of the interfaces, addresses, and the connections between them based on the rules in all of the chains.

```
graphviz    :: [Chain] → IO ()
graphviz cs = putStr $ unlines
  ["// Generated by gressgraph v" ++
   (show gressgraphVersion) ++ " <http://jekor.com/gressgraph/>",
   "",
   "digraph gressgraph {",
   graphAttributes,
   "",
   "// Invisible root node",
   "rootNode [root=true style=invis]",
   "",
   "// Interfaces",
   unlines $ map interfaceNode (zip interfaces interfaceSizes),
   unlines $ map toRoot interfaces,
   "// Addresses",
   unlines $ map addressNode (zip addresses addressSizes),
   "// Rules",
   unlines $ map showChain cs,
   "}"]
```

where

```
inInterfaces = getMembers inInterface cs
outInterfaces = getMembers outInterface cs
sources      = getMembers source      cs
destinations = getMembers destination cs
inAddresses  = zip inInterfaces sources
outAddresses = zip outInterfaces destinations
(interfaces, interfaceSizes) = sizes size (inInterfaces ++ outInterfaces)
(addresses, addressSizes)   = sizes size (inAddresses ++ outAddresses)
```

getMembers is a helper function that extracts a list of members from a list of chains. It takes an accessor function *f* and applies it to each rule of each chain. You can think of it as a specialized *map*.

For instance, *getMembers action cs* will return the list of actions present in *cs* (where *cs* is a list of chains), such as [*Drop, Drop, Accept, Drop, Accept*] (for a very short set of chains!).

```
getMembers    :: (Rule → a) → [Chain] → [a]
getMembers f cs = concatMap (map f ∘ snd) cs
```

More active nodes need be drawn larger than nodes with few connections. *sizes* takes a list as returned by *getMembers* and converts it into a list of weighted members. It does this by counting up each distinct member and applying a scaling function *f* to it.

Continuing with our *action* example from above,

```
sizes id (getMembers action cs) = [(Drop, 3), (Accept, 2)].
```

```
sizes      :: Ord a => (Int -> Float) -> [a] -> ([a], [Float])
sizes f xs = (map head xs', map (f ∘ length) xs')
  where xs' = group (sort xs)
```

Even though an address is a combination of interface and address, for the graph it's nicer to display just the address component. This shouldn't cause any confusion for someone viewing the graph.

We also set a height for the node to keep the edges on it from bunching up.

```
addressNode      :: (Address, Float) -> String
addressNode (a, diameter) = showAddress a ++ " [label=\\"" ++ snd a ++
  "\" height=\"" ++ show diameter ++ "]"
```

We set the width in addition to the height for interface nodes. This gives a visual cue for distinguishing between interfaces and addresses. This is a little risky because we may set a width that's too small for the label of the node, but interface names are usually very short.

```
interfaceNode    :: (String, Float) -> String
interfaceNode (i, diameter) = quote i ++ " [height=\"" ++ show diameter ++
  "\" width=\"" ++ show diameter ++ "]"
```

To keep the interface nodes in a circular configuration, we need to connect each of them to an invisible root (hub) node.

```
toRoot  :: String -> String
toRoot i = quote i ++ " -> rootNode [style=invis]"
```

To keep edges from bunching up on a node and becoming a tangled mass, we need to increase the size of nodes with more connections. But we don't want busy nodes to dominate the graph. We'll use a naïve logarithmic scaling. To get the size (either the width or height) of a node (in inches), we use:

$$size = \log_{10} n + m$$

Where n is the number of edges that touch the node and m is the minimum size (in inches).

```
size  :: Integral a => a -> Float
size n = logBase 10 (fromIntegral n) + minimumSize
```

$\frac{1}{4}$ inch is about the size that Graphviz uses as a default height.

```
minimumSize :: Float
minimumSize = 0.25
```

By default, Graphviz places nodes too close together when using the twopi layout algorithm.

```
rankSep :: Float  
rankSep = minimumSize * 12.0
```

We want a background color upon which colors will stand out well. Grey is best for that.

```
backgroundColor :: String  
backgroundColor = "#808080"
```

```
graphAttributes :: String  
graphAttributes = "graph [ranksep=" ++ show rankSep ++  
" bgcolor=\"\" ++ backgroundColor ++ "\"]"
```