Implementação de um Túnel ICMPv6

Guilherme Krzisch e Pedro Webber

24 de novembro de 2013

1 Introdução

O objetivo do trabalho é a implementação de um túnel ICMPv6 para transporte de pacotes TCP sobre IPv6. Isto é necessário porque haverá um firewall impedindo a passagem de pacotes TCP entre e o cliente e o servidor remoto, então iremos tentar "enganar" o firewall, mandando pacotes TCP encapsulados em ICMPv6 (do tipo "echo request" e "echo reply"), que ele deixará passar, e nosso proxy irá recebê-los.

O proxy irá então desencapsular a mensagem recebida e realizar a operação descrita (estabelecimento de conexão, finalização de conexão, envio de dados,...) com o servidor remoto.

O trabalho deverá ser realizado utilizando "socket raw"; e o protocolo do nível de aplicação que iremos utilizar é o protocolo HTTP, mais especificamente, o comando GET para requisição de uma página WEB.

Figura 1: Overview da comunicação no nosso trabalho.

2 Implementação

O objetivo do cliente é realizar requisições a um servidor remoto, por exemplo para o website "www.blanksite.com", pedindo um determinado arquivo, por exemplo "index.html". Esta requisição será feita via protocolo HTTP, com o comando GET, transportado sobre TCP. Como há um firewall bloqueando pacotes TCP, o cliente deverá encapsular estas mensagens com o protocolo ICMPv6, do tipo "echo request". Como o servidor remoto, se recebesse essas mensagens, mandaria de volta um "echo reply" (mecanismo do ping), e não o que estamos esperando, devemos enviar estes pacotes para uma máquina - o proxy - que irá

processá-los, desencapsulá-los, e fazer o devido encaminhamento para o servidor remoto. Quando o proxy receber dados do servidor remoto, e tiver que repassá-los para o cliente, irá encapsular eles também com ICMPv6, do tipo "echo reply", para passar pelo firewall.

2.1 Protocolo HTTP

O protocolo de aplicação que escolhemos foi o HTTP, pois já possuíamos um conhecimento prévio sobre ele, além de ser fácil e abrangente o seu uso. A ação que iremos requisitar ao servidor remoto é o método GET, que pede uma representação de um determinado recurso.

Requisição GET:
GET / HTTP/1.1
Host: www.blankpage.com

Resposta do servidor:

```
HTTP/1.1 200 OK
Date: Sun, 24 Nov 2013 15:52:46 GMT
Server: Apache/2.2.22 (Ubuntu)
X-Powered-By: PHP/5.3.10-1ubuntu3.6
Set-Cookie: uid=www529220ce7ab574.57500832; expires=Tue, 24-Dec-2013 15:52:46 GMT
Vary: Accept-Encoding
Content-Length: 2324
Connection: close
Content-Type: text/html
Set-Cookie: WEB=W1; path=/
```

```
 <html>

<head>
        <meta http-equiv="content-type" content="text/html; charset=utf-8" />
   <title>BlankPage.com</title>
        <script type="text/javascript">;
      if(self != top) top.location.href = 'http://'+location.hostname+'/?redir=frame&uid=www529220ce7ab574.57500832';
      </script>
      <script type="text/javascript" src="http://return.bs.domainnamesales.com/return_js.php?d=blankpage.com&s=1385308366"></script>

   <script type="text/javascript" src="http://ajax.googleapis.com/ajax/libs/jquery/1.5.2/jquery.min.js"></script>
   <script type="text/javascript">
      function GetParam(name) {
          var match = new RegExp(name +
                  "=*([^&]+)*", "i").exec(location.search);
          if (match == null)
              match = new RegExp(name + "=(.+)", "i").exec(location.search);
          if (match == null) return null;
              match = match + "";
          //**convert match to a string
          result = match.split(",");
          return decodeURIComponent(result[1]);
      }

      function logStatus(type) {
          $.ajax({
              cache: false,
              global: false,
              async: true,
              type: "POST",
              url: '/logpstatus.php',
              data: {uld: GetParam('uld'), type: type}
          });
      }
   </script>

   </head>
   <frameset cols="1,*,1" border=0>
      <frame name="top" src="tg.php?uld=www529220ce7ab574.57500832" scrolling=no frameborder=0 noresize framespacing=0 marginwidth=0 marginheight=0>
      <frame src="search_caf.php?uld=www529220ce7ab574.57500832&src=hydrabiggerbullets&abp=0" scrolling="auto" framespacing=0 marginwidth=0 marginheight=0 noresize>
      <frame src="page.php?www529220ce7ab574.57500832"></frame>
   </frameset>
   <noframes>
      blankpage.com has been connecting our visitors with providers of Build A Web Site, Cheap Web Design, Create Web Sites and many other related services for nearly 10 years. Join thousands of satisfied
visitors who found Ecommerce Web Design, Flash Web Design, Graphic Web Design, Multimedia Designs, and Search Engine Optimization.<br/>    </noframes>
</html>
```

Figura 2: Resposta recebida do servidor, contendo a página HTML requisitada.

2.2 Protocolo TCP

O protocolo TCP é orientado a conexão, fim-a-fim e confiável. Provê comunicação entre processos situados em diferentes computadores. Em uma arquitetura

em camadas, ele se situa uma camada acima do protocolo IP.

2.2.1 Estabelecimento de conexão

Antes do cliente poder requisitar dados do servidor remoto, via proxy, ele terá que estabelecer uma conexão com o proxy, conforme a especificação do protocolo TCP. Para isso, deve ser usado o procedimento de "three-way handshake". Ele consiste no envio de um pacote SYN pedindo conexão de um dos hosts, no envio da resposta do outro host com um pacote SYN ACK (o ACK para confirmar o pedido de conexão, e o SYN para também requisitar uma conexão), e finalmente no envio de um pacote ACK, confirmando o pedido de conexão do segundo host, vindo do primeiro host.

Na nossa implementação é sempre o host cliente que começa a comunicação com o proxy, começando o procedimento de "three-way handshake". Portanto, o proxy começa em um estado de "esperando uma nova conexão". (O protocolo TCP também suporta o pedido de conexão simultânea por ambos os hosts; porém, como na nossa implementação é sempre o mesmo host - o cliente - que inicia a comunicação, não precisamos nos preocupar com isso.) Ao fim do estabelecimento da conexão, o proxy estará em um estado de "esperando dados do cliente". Nossa implementação é "single-threaded", portanto nosso proxy só consegue lidar com um cliente por vez. Se algum outro cliente requisita um início de conexão quando já existe uma conexão em andamento, este cliente não receberá nenhuma resposta. (Obs.: Estamos usando os dois processos - no cliente e no proxy - na porta 60.000).

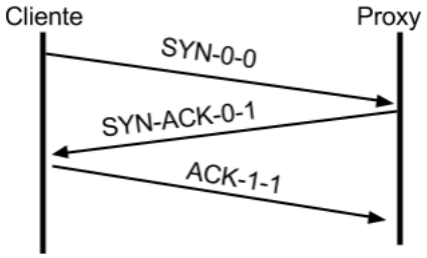

Figura 3: Mensagens trocadas entre o cliente e o proxy para o estabelecimento da conexão. (obs.: legenda das mensagens: [operação do TCP]* - [Número de sequência relativo] - [Número de acknowledgment relativo]).

2.2.2 Envio e recebimento de dados

Após o estabelecimento da conexão, o proxy está pronto para receber dados do cliente. O dado que o cliente quer enviar é uma requisição GET para um servidor remoto. O proxy receberá esta requisição, enviará um ACK de confirmação de recebimento do pedido para o cliente (que ficará esperando os demais dados), e criará um socket (não "raw") para comunicação com o servidor requisitado (como

este não é um "socket raw", não precisamos nos preocupar com o estabelecimento da conexão, pois este socket trata disso sozinho). Após o envio da requisição para o servidor remoto, por parte do proxy, ele fica no aguardo da resposta com os dados. Ao receber os dados de reposta, ele repassa ao cliente somente os dados interessantes - ou seja, o HTML da página requisitada. O cliente pega estes dados e cria um arquivo HTML na pasta onde o programa está rodando. Assim o cliente poderá visualizar a página requisitada abrindo o arquivo por meio de algum "web browser".

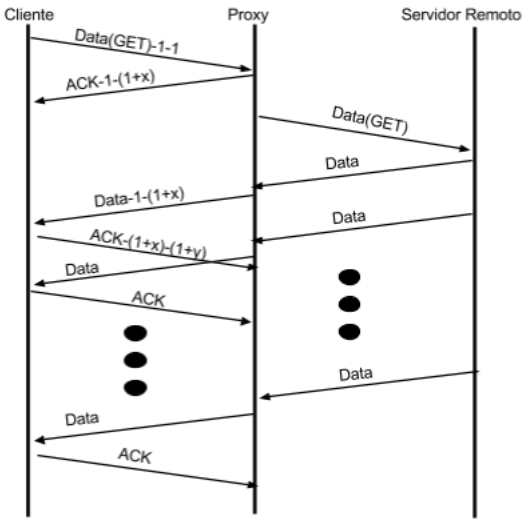

Figura 4: Mensagens de dados trocadas entre o cliente, o proxy e o servidor remoto. Obs.: "x" é igual ao tamanho do pacote de dados GET vindo do cliente. "y" é igual ao tamanho do primeiro pacote de dados do proxy. O proxy pode receber "n" respostas do servidor remoto, e irá encaminhá-las para o cliente.

2.2.3 Finalização da conexão

Conforme a especificação do TCP, um host deve pedir a finalização da conexão quando ele não tiver mais dados a enviar (note que ele ainda pode continuar a receber dados da outra parte; os dois hosts não precisam fechar a conexão simultaneamente, por isso o processo de finalização de uma conexão TCP é descrito como "half-close").

Após o proxy ter recebido todos os dados do servidor remoto e tê-los repassado ao cliente, ele primeiro fecha a comunicação com o servidor remoto, e após começa o processo de finalização da conexão com o cliente (após ter recebido o último ACK dos dados enviados ao cliente), enviando um pacote FIN, e esperando como resposta um ACK. Assim, estará fechada a conexão pela parte do proxy. Para o fechamento da conexão por parte do cliente, temos duas opções: ou pedimos o fechamento de conexão logo após enviarmos a requisição GET

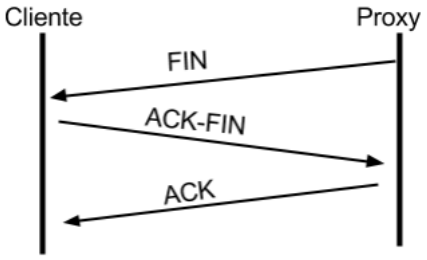

Figura 5: Mensagens trocadas entre o cliente e o servidor para o fechamento da conexão.

- pois não teremos mais dados a serem enviados, só a serem recebidos -, ou pedimos o fechamento após a finalização da conexão por parte do proxy.

Embora a primeira opção pareça mais correta, tendo em vista a especificação do TCP (fechamento da conexão quando "I have no more data to send."), implementamos a segunda opção, pela sua facilidade e simplicidade (e também por a termos codificado antes de perceber que poderíamos ter fechado a conexão por parte do cliente antes).

Os recursos então são liberados, o cliente possui o arquivo HTML da página requisitada, e o proxy está pronto para receber uma nova conexão.

2.3 Protocolo ICMPv6

Os pacotes ICMPv6 servirão ao propósito de enganar o "firewall". Como ele não deixa passar pacotes TCP, iremos encapsulá-los em pacotes ICMPv6. O cliente enviará pacotes do tipo "echo request", e o proxy retornará pacotes do tipo "echo reply".

Uma maneira do "firewall" barrar este túnel ICMPv6 seria ele inspecionar todos os pacotes ICMPv6, checando se ele carrega alguma informação TCP; porém isso causaria um grande "overhead" de processamento, o que se tornaria inviável. Outra maneira seria bloquear pacotes ICMPv6, porém isso seria ainda mais inviável, pois perderíamos funções importantes do protocolo. A maneira mais realística de bloqueio seria permitir somente pacotes ICMPv6 de tamanho fixo.

2.4 Protocolo IPv6 e Ethernet

Por fim, o pacote será encapsulado no protocolo IPv6 e no protocolo Ethernet, ficando pronto para ser enviado.

2.5 Juntando tudo

Como a implementação do cliente e do proxy utilizam várias funções similares, fizemos uma classe que encapsula elas, que se chama "socketRawAPI.c". O

código do proxy está dentro da pasta "Receiver", no arquivo "receiver.c"; e o do cliente está dentro da pasta "Sender", no arquivo "sender.c". Para compilá-los, deve-se estar nas suas respectivas pastas e executar os seguintes comandos: "gcc -o receiver receiver.c" e "gcc -o sender sender.c".

Para rodar o proxy, estando na sua devida pasta, utiliza-se o comando "sudo ./receiver", e ele está pronto para receber novas conexões. O cliente precisa de mais informações para ser rodado: "sudo ./sender [IPv6 do servidor] [MAC do servidor] [URL] [path]" (exemplo: "sudo ./sender XXXX::XXXX:XXXX:XXXX:XXXX XX:XX:XX:XX:XX:XX www.google.com /").

Para a configuração do firewall foi utilizado o programa "ufw"[6] e seu equivalente gráfico "gufw"[7].

3 Resultados

Iremos demonstrar, por meio de imagens, nosso programa funcionando.

Configuramos o proxy em uma máquina dentro da mesma rede local em que o cliente estava. Configuramos o firewall para impedir a passagem de tráfego TCP entre eles. Fizemos uma requisição para este proxy (encontrado com seu IPv6 e endereço MAC), para acessar a página "www.candybox2.net/". O tráfego, do ponto de vista do cliente, pode ser visto nas seguintes imagens:

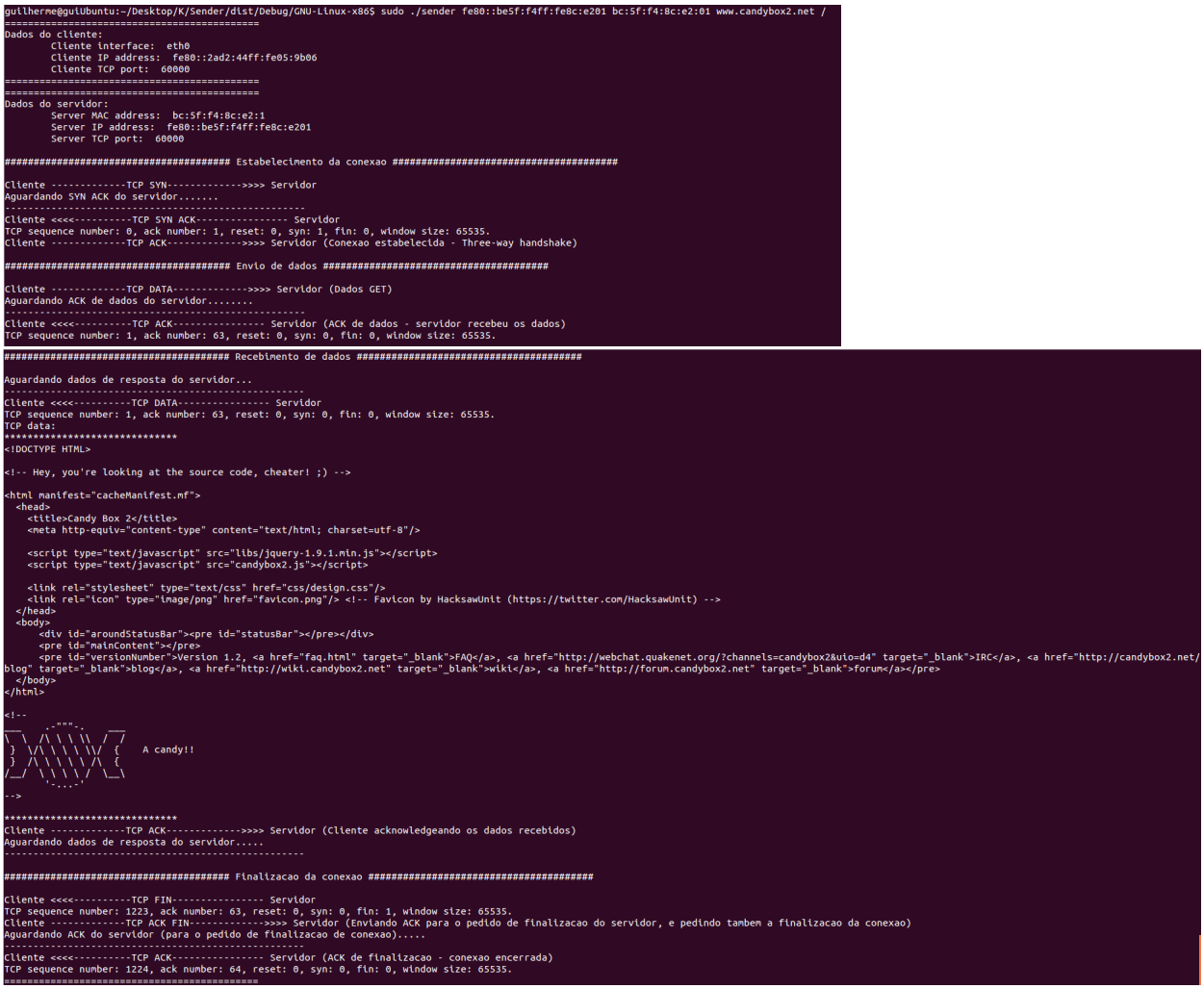

Figura 6: Tráfego entre o cliente e o proxy.

Tentamos deixar nosso código informando passo a passo o que ele está fazendo, com informações relevantes (por exemplo, em qual fase do protocolo TCP nos encontramos; qual pacote de mensagem está sendo enviado; quais foram os dados recebidos,...), por isso não explicaremos linha a linha o que aconteceu, pois basta ler as mensagens informadas na tela.

E como o que o cliente queria era o resultado do comando GET, mais precisamente o HTML da página requisitada, escrevemos esses dados recebidos em um arquivo ".html", neste caso "www.candybox2.net.html", no diretório em que está sendo executado o programa. Podemos observar que o HTML recebido é igual ao HTML da página, se acessarmos ela via "web browser".

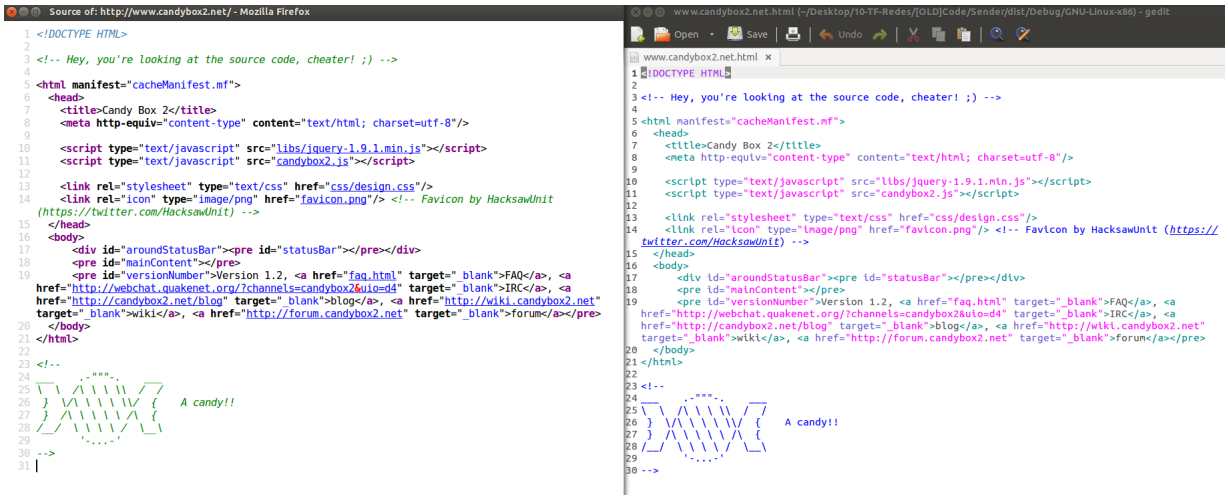

Figura 7: A tela da esquerda mostra o HTML da página acessada via "web browser". A tela da direita mostra o HTML recebido por nosso cliente via proxy. Podemos observar que o conteúdo é idêntico.

Pacotes da comunicação capturados pelo Wireshark:

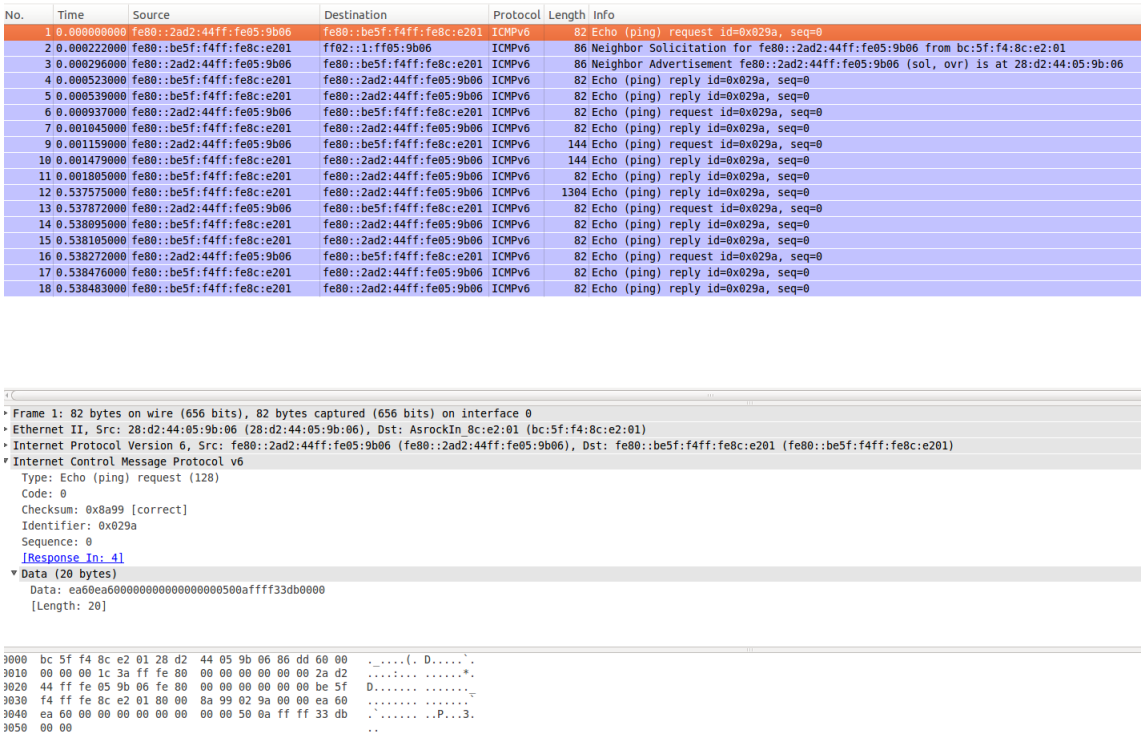

Figura 8: Pacotes capturados pelo Wireshak.

4 Dificuldades encontradas

A maior dificuldade encontrada foi a programação na linguagem C, que às vezes pode ser trabalhosa, por termos que nos preocupar com alocação de memória. Além disso, para testarmos se o programa estava funcionando, tivemos que utilizar duas máquinas, configurando os endereços de MAC/IP origem/destino corretamente, além da configuração do "firewall", o que exigia um certo tempo e esforço.

A requisição para determinadas páginas de alguns websites às vezes gera um erro ("ERROR! sendto()") que não conseguimos corrigir. Páginas/Websites que testamos e deram certo: "www.google.com/", "www.candybox2.net/". Páginas/Websites que testamos e deram errado: "www.gnu.org/". Páginas/Websites que testamos e às vezes deram certo e às vezes errado: "www.blanksite.com/".

5 Trabalhos futuros

O que poderíamos melhorar em um trabalho futuro é deixar o código mais limpo e modular, de forma que seja possível estender suas funcionalidades para

deixar o programa mais "user-friendly". Talvez permitir ao usuário escolher quais operações/pacotes ele deseja enviar ao servidor/proxy, como por exemplo: estabelecer conexão (pedir IP e MAC destino), enviar dados (tipo de dado, dados), fechar conexão,...

Outro ponto seria fazer uma implementação "multi-thread" do nosso proxy, de forma a permitir conexões simultâneas de diferentes clientes.

Tentamos também, com relativo sucesso, de que quando o cliente recebesse todo o HTML do proxy, ele abrisse o navegador "default" para a visualização da página. Testamos esse método com o navegador Firefox, porém, toda vez que abríamos ele, nosso método desconfigurava o navegador. Descobrimos que isso acontecia porque nosso código do cliente executava com permissões "root", por meio do comando "sudo", e, para o correto funcionamento do navegador, deveríamos rodá-lo como um usuário comum da máquina. Como para o presente trabalho não precisávamos desta funcionalidade, deixamos este problema de lado, porém gostaríamos de tê-lo resolvido.

Na questão de finalização da conexão TCP, poderíamos fazer o cliente pedir o fechamento da sua parte da conexão logo após ele ter enviado a requisição GET, pois, após isso, ele não terá mais nenhum dado a enviar para o proxy. Ou poderíamos permitir ao cliente fazer múltiplas requisições GET dentro de uma mesma conexão com o proxy, assim evitando o "overhead" de ter que passar pelas fases de estabelecimento e finalização da conexão a cada requisição.

Resolver o erro que a requisição a alguns websites gera seria uma das prioridades, em questões de implementação.

6 Conclusão

Acreditamos que conseguimos um bom resultado, com uma implementação satisfatória do cliente e do proxy. A requisição GET do cliente está sendo corretamente enviada para o proxy (passando pelo firewall), que realiza a conexão com o servidor remoto, devolvendo os dados de resposta para o cliente, que pode então visualizar a página HTML requisitada.

Referências

[1] http://tools.ietf.org/html/rfc793

[2] https://en.wikipedia.org/wiki/Transmission_Control_Protocol

[3] https://en.wikipedia.org/wiki/ICMP_tunnel

[4] http://packetlife.net/blog/2010/jun/7/understanding-tcp-sequence-acknowledgment-numbers/

[5] https://en.wikipedia.org/wiki/File:Tcp_state_diagram_fixed.svg

[6] https://help.ubuntu.com/community/UFW

[7] https://help.ubuntu.com/community/Gufw