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, ?? de Novembro de 2001

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

Este projeto tem como objetivo a elaboração de um autômato finito configurável que possibilite ao usuário gerenciar um comportamento de fluxo de pacotes de rede utilizando, principalmente o conjunto de protocolos TCP/IP. Para tal decidiu-se elaborar, em uma primeira etapa, uma linguagem que descrevesse o comportamento deste fluxo para ser interpretada pela plataforma e que fosse executada pela mesma.

Assim, na primeira metade deste projeto, fez-se um estudo teórico sobre redes de computadores, segurança de redes, os ataques mais comuns, os protocolos TCP/IP e métodos de filtragem de pacotes.

Além deste estudo teórico, um estudo prático foi feito elaborando programas que utilizassem recursos de rede para uma melhor familiarização de um ambiente de rede. Entre estes programas existem alguns que se utilizam de falhas de segurança (sejam falhas do protocolo, sejam falhas de implementação) elementares. Destes programas pôde-se confeccionar uma biblioteca para manipulação e confecção de pacotes.

Para poder compreender melhor o fluxo de pacotes (tipos, cabeçalhos, tamanhos, serviços, etc), foi preciso utilizar ferramentas que capturassem os pacotes e os mostrassem de forma consistente, em um formato reconhecível para os olhos humanos (TCPDUMP, ETHEREAL). E para interagir neste fluxo de pacotes, uma ferramenta em especial, o NMAP[1], foi de extrema importância e serviu de inspiração para algumas seções da plataforma, uma vez que este consegue extrair informações de uma rede apenas inserindo pacotes nela e observando suas respostas.

[1] www.insecure.org/nmap

Possuindo uma visão de alguns problemas de segurança de redes, inicia-se o momento da implementação da plataforma, alvo deste projeto. E então vem a etapa de criação e implementação da linguagem que descreverá o comportamento do autômato finito configurável.

Até ao final da primeira metade deste projeto, tinha-se definido a estrutura de dados base da plataforma, estrutura que possuirá todas as informações da máquina de estados e que será percorrida por uma outra estrutura (o player) que executará suas ações. Também fora estabelecido um método de declaração de cabeçalho para que se alcançasse a flexibilidade inicialmente desejada . Além disso, tinha se definido as ferramentas (LEX e YACC, ou melhor, FLEX e BISON) que iriam ser utilizadas para a construção da linguagem e uma gramática rudimentar base.

2. Resumo das Atividades Referentes a este Relatório

Nesta segunda metade do primeiro ano deste projeto ficou estabelecido que seria terminada a definição da linguagem e a implementação da estrutura de dados que será percorrida e executada por um "player", também implementado nesta parte do projeto.

Ficou estabelecido também, que um método de construção e armazenamento de pacotes teria de ser definido e construído. Uma maneira descritiva que fosse interpretada e resultasse em pacotes prontos para serem enviados pela rede e armazenados em disco, ou seja, um construtor de pacotes.

Em linhas gerais, isto foi o desenvolvido nesta segunda parte e que será detalhado abaixo. A medida com que estes tópicos genéricos foram sendo implementados, novas necessidades e dificuldades vão aparecendo, o que faz com que estes tópicos se expandam.

3. Atividades Referentes a este Relatório

3.1 Linguagem

A versão final (até o momento) da linguagem foi definida da seguinte maneira no formato Bakus-Naur (BNF):

```
<start>          ::= <expression>
<expression>  ::= "init" NUMBER "states"           (1)
                 |   "load" "headers" ":" <EXPR>      (2)
                 |   "load" "packets" ":" <EXPR>      (3)
                 |   "create" "state" <STR>           (4)
                 |   "state" <STR> <action>           (5)
                 |   "show" <state_table>             (6)
                 |   "play"                           (7)
                 |   "bye"                            (8)

<action> ::= "send_pack" "{" "pack_id" ":" <NUMBER> "next" "state" ":" STR "}"      (9)
(10)        |  "wait" "{" <time> "next" "state" ":" <STR> "}"
(11)        | "recv_pack" "(" "rid" "=" <NUMBER> ")" "{" "bpf" "=" <EXPR> <branch_expr> "}"
(12)        | "finish"

<branch_expr> ::=  <b_expr>                            (13)
                 |    <b_expr> "or" <b_expr>           (14)

<b_expr> ::= "(" "next" "state" ":" <STR> "check" ":" <field_expr> ")"    (15)

<field_expr> ::= <f_expr>                              (16)
               |   <f_expr> "and" <f_expr>             (17)

<f_expr> ::= <STR> "." <STR> "=" <NUMBER>  (18)

<time> ::= <U_SEC> | <M_SEC>                           (19)

<state_table> ::= "all" | "used" | "state" <show_type>         (20)

<show_type> ::= <NUMBER> | <STR>                               (21)
```

OBS: NUMBER, U_SEC e M_SEC são símbolos não terminais que representam número; STR e EXPR são símbolos não terminais que representam *string*;

Abaixo está a descrição das declarações da linguagem:

- (1): Inicia <NUMBER> estados que serão utilizados. Esta é uma das primeira informações a ser entregue à plataforma;
- (2): Carrega em memória as descrições dos cabeçalhos definidas pelo usuário nos arquivos contidos no diretório <EXPR>;
- (3): Carrega em memória os pacotes armazenados no arquivo <EXPR> que foram previamente construídos e armazenados pelo gerador de pacotes;
- (4): Aloca memória e define um estado com o nome <STR>;
- (5): Define a ação do estado <STR>;
- (6): Mostra a tabela de estados;
- (7): Inicia a máquina de estados;
- (8): Termina a plataforma e retorna ao sistema operacional;
- (9): Define como ação do estado o envio do pacote com número de identificação <NUMBER> e passa para o estado <STR>;
- (10): Define como ação do estado a espera um tempo <time> e passa para o estado <STR>;
- (11): Define como ação do estado o recebimento de pacotes e a definição do filtro BPF para auxiliar no critério de escolha para o próximo estado;
- (12): Define como ação do estado o término da máquina de estados;
- (20): Mostra a tabela de estados: Pode escolher mostrar toda a tabela, apenas as entradas válidas, ou apenas uma entrada desejada em particular;

Como ilustração segue o exemplo abaixo:

```
init 4 states;
load headers : "./headers";
```

```
load packets : "./packer/out_pack";
create state HELLO;
create state HOLD;
create state OKAY;
create state BYEBYE;

state OKAY send_pack {
  pack_id : 1
  next state : BYEBYE
};

state BYEBYE finish ;

state HELLO recv_pack (RID = 1) {
  bpf = "tcp and port 21"
  ( next state: OKAY
    check:        TCP.sport = 123 and
                  TCP.dport = 21
  ) or
  ( next state: HOLD
    check:        IP.saddr = 10.1.1.1 and
                  IP.daddr = 10.1.1.2
  )
};

state HOLD wait {
        210ms
        next state : HELLO
};
```

Esta descrição acima é referente à maquina de estados representada pela figura abaixo:

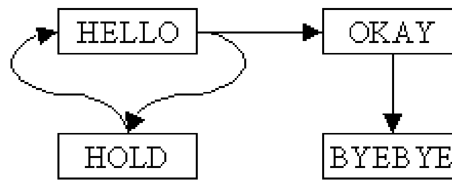

Figura 1

O primeiro estado declarado é sempre o estado que inicia a máquina de estados, neste caso, o estado HELLO. Dependendo do pacote que chegar à plataforma, o rumo a ser tomado pode ser o estado OKAY ou o estado HOLD (esta escolha será discutida em 3.7), o estado HOLD espera 210ms e retorna ao estado HELLO. O estado OKAY envia o pacote de número de identificação 1 em passa para o estado BYEBYE que termina a máquina de estados.

Pode-se perceber na declaração do estado que recebe pacotes (HELLO), no campo "check", que poderiam haver tantos critérios quanto fossem precisos, mas a análise sintática é linear e ao chegar ao início da declaração de *recv_pack*, não se sabe quanto de memória se deve alocar, pois não se sabe quantos são os critérios. Portanto, a medida com que vai se constatando campos de um pacote a se verificar, vai se alocando memória dinamicamente em forma de lista ligada. Ao final da declaração *recv_pack*, com o número de critérios já definido, estes dados em lista ligada são colocados em um vetor para poderem ser indexados afim de possuir um melhor rendimento.

Esta foi uma opção de implementação. Uma outra seria fazer com que o usuário informasse, de antemão, quantos critérios seriam utilizados e assim não precisaria haver este repasse de memória (lista ligada para vetor), mas haveria um certo comprometimento da linguagem.

3.2 Definição dos Cabeçalhos

A abordagem desta plataforma cobre a transação de pacotes de rede num baixo nível, tratando pacotes como apenas um aglomerado de bits a ser inserido no trafego de

rede. Com isso, não mais existem estruturas nem tipos de dados que compõem um pacote, nem a definição de protocolo é mencionada.

Para sustentar tal "deficiência" de informação para se formar um pacote, o usuário deve descrever em arquivos texto, de acordo com o formato abaixo definido, o formato que seu pacote deverá possuir. Este formato pode ser constituído de vários cabeçalhos que indicam a posição relativa dos campos ao início do cabeçalho.

Assim, utilizando a abstração de camadas (layers), pode-se concatenar vários cabeçalhos para construir um pacote. Esta característica exige a existência do campo "hlen" em todos os cabeçalhos (menos o último cabeçalho a ser concatenado) constituintes do pacote, pois o "offset" dos campos declarado é relativo ao início do cabeçalho em que residem, e portanto se faz necessário o conhecimento do tamanho dos cabeçalhos anteriores para se calcular a posição absoluta do campo dentro do pacote.

Alem disso, o usuário poderá escolher o "byte order" da informação a ser inserida no pacote a manuseá-la de acordo com sua preferência.

Para carregar os cabeçalhos, é necessário apenas o diretório dos cabeçalhos e todos os arquivos neste contidos serão analisados.

Esta análise é feita com uma leitura linear do arquivo colocando os dados em uma fila e contando o números de cabeçalhos. Com uma lista ligada e o número de cabeçalhos e campos faz-se uma tabela de "hashing" cuja chave é o nome dos campos.

Assim trabalham as funções abaixo descritas:

- void count_headers(char wdir, struct hdir_in **hdirs);

wdir (argumento de entrada): caminho do diretório que possui os arquivos que descrevem os cabeçalhos;

hdirs (argumento de saída): lista ligada que possui em cada elemento da lista as seguintes informações:

- caminho completo do arquivo que descreve o cabeçalho;
- o nome do cabeçalho;
- a camada em que residirá o cabeçalho;
- o número de campos que compõem a cabeçalho;

Esta função analisa o diretório dado por wdir e retorna uma lista ligada apontada por hdirs;

- u_int16_t count_noh(struct hdir_in *hd);
  hd (argumento de entrada): o mesmo que hdirs acima descrito;

Esta função retorna o número de cabeçalhos que serão utilizados;

- struct header_hash_table **mount_header_table(struct hdir_in *hd,u_int16_t noh);
  hd (argumento de entrada): o mesmo que hdirs acima descrito;
  noh (argumento de entrada): numero de cabeçalhos;

Esta função retorna um ponteiro para uma tabela de "hashing" de cabeçalhos, que será explicada mais detalhadamente em seguida.

struct header_info **mount_field_table(char *path, u_int16_t nof);
  path (argumento de entrada): caminho completo do arquivo que descreve o cabeçalho;
  nof (argumento de entrada): numero de campos de um cabeçalho;

Esta função retorna um ponteiro para uma tabela de "hashing" de campos de um determinado cabeçalho, que também será explicado mais detalhadamente em seguida.

```
- struct header_info *gimme_the_data(hname,fname,hdrtable,noh,layer)
    char                      *hname;       /* in  */
    char                      *fname;       /* in  */
    struct header_hash_table  **hdrtable;   /* in  */
    u_int16_t                 noh;          /* in  */
    u_int16_t                 *layer;       /* out */
```

hname (argumento de entrada): nome do cabeçalho;

fname (argumento de entrada): nome do campo;

header_hash_table (argumento de entrada): tabela de "hashing" de cabeçalhos;

noh (argumento de entrada): número de cabeçalhos;

layer (argumento de saída): camada em que reside tal cabeçalho;

Esta função retorna o offset do início do cabeçalho e o tamanho, em bits, do campo requisitado, e se este deve ou não ter sua "byte order" invertida.

Após ter implementado as funções que lidam com os cabeçalhos, decidiu-se estabelecer o arquivo de definição dos cabeçalhos da seguinte maneira:

- Na primeira linha é estabelecido o nome o cabeçalho e a camada a qual pertence;

- Nas linhas seguintes define-se, nesta ordem, o nome do campo, o offset em bits do início do cabeçalho, o tamanho em bits do campo e a necessidade ou não de "swap" a ordem de bytes (big endian/little endian);

Para ilustrar tal estrutura de dados faz-se a utilização do seguinte exemplo (os cabeçalhos abaixo serão extremamente simplificados para uma melhor visualização dos resultados):

- dentro de um diretório definido pelo usuário devem existir apenas os arquivos de definição dos cabeçalhos, que são:

/.../<nome do diretório>/IP :

```
#PACKMAN HEADER: IP LAYER 1
#name   ofsset  size   swap?(byte order)
hlen    0       4      0
ver     4       4      0
TOS     8       8      0
```

/.../<nome do diretório>/TCP :

```
#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

- no ambiente da plataforma digitar: load headers : "/.../<nome do diretório>" <enter>

E então será obtida a seguinte estrutura de dados montada em memória:

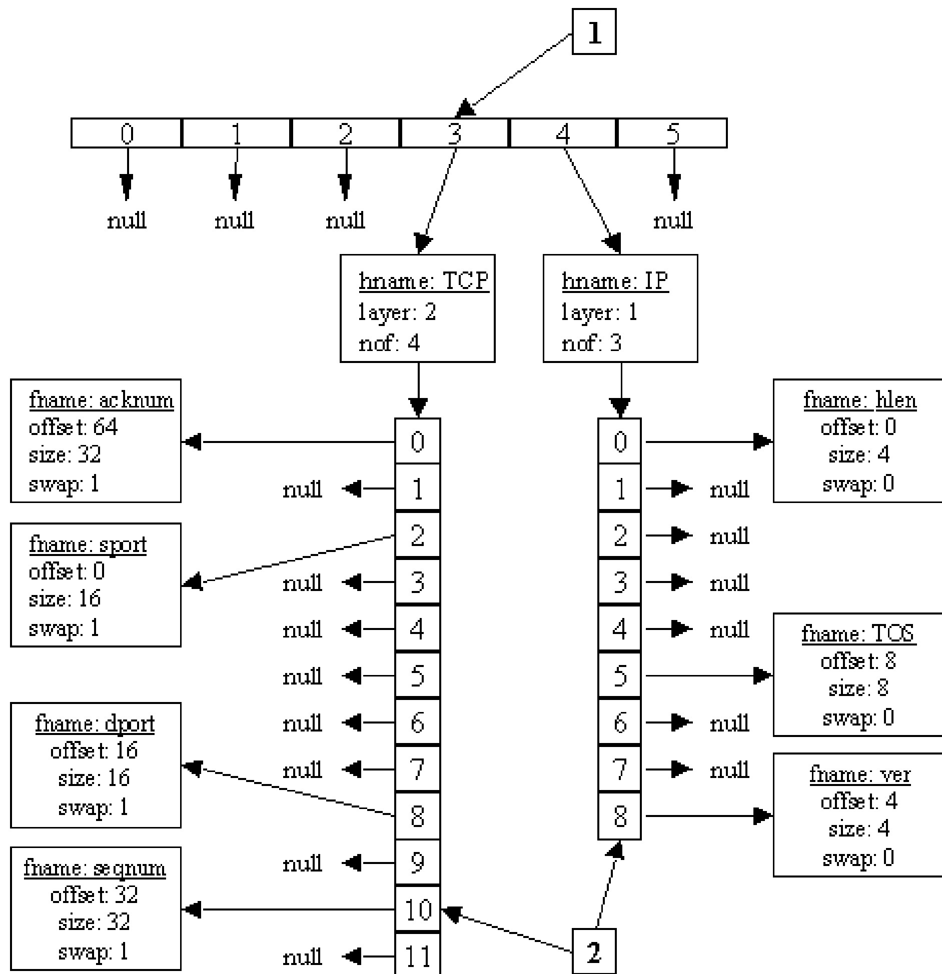

Figura 2

Verifica-se na figura acima que há duas tabelas de "hashing" combinadas. A primeira (1), que está em um nível mais elevado, é utilizada para localizar as informações de um cabeçalho como um todo, já a segunda (2 – as duas tabelas inferiores) é utilizada para localizar as informações especificas sobre o cabeçalho definido pela tabela anterior.

Assim fica justificada a necessidade de se entregar à função "gimme_the_data", o nome do cabeçalho e o nome do campo.

- Com tal estrutura de dados em memória pode-se obter as informações sobre os campos dos cabeçalhos, sendo uma maneira direta de fazê-la seria digitando no ambiente da plataforma, por exemplo: resolve TCP.dport; <enter>. E seria obtida a seguinte resposta: "2 dport 16 16 1", camada, nome do campo, posição relativa, tamanho e swap? respectivamente.

3.3 CONSTRUTOR DE PACOTES

O processo de construção de pacotes, aqui empregado, é análogo ao processo de montagem de um programa. O usuário digita um arquivo texto com uma descrição em alto nível dos pacotes e obtém, ao final do processo, um arquivo com os pacotes em binário.

A gramática do montador de pacotes (em BNF) é a seguinte:

```
<start> ::= <statement>

<statement> ::= "out_pack_file" ":" <EXPR> "{" <p_expression> "}"
              |   <load_h>

<p_expression> ::= <expression>
                 |   <p_expression> <expression>
<expression>::= "packet""#" <NUMBER>"("<NUMBER> "bytes" ")" "{" <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>
```

<load_h> ::= "load" "headers" ":" <EXPR>

Também aqui, faz-se o uso das definições de cabeçalhos e das estruturas de dados discutidas em 3.2.

Um ponto importante a ser verificado é a questão da concatenação dos cabeçalhos, pois a informação sobre os campos de cada cabeçalho é relativo ao início de cada um deles e nem todos iniciam no começo do pacote. Para fazer tal ajuste de posição fez-se o uso da abstração de camadas, e assim, ao examinar cada cabeçalho e ao passar pelo campo "hlen" (justificando o porquê de todos os cabeçalhos, menos o último, terem que possuir um campo com o nome "hlen") tal valor relativo a este campo é armazenado. Dessa forma, para se calcular o valor absoluto da posição de um campo verifica-se o valor da soma dos tamanhos dos cabeçalhos anteriores.

O arquivo que contem os pacotes em binário está organizado da seguinte forma:

[Número de pacotes - 4 bytes][posição da tabela de pacotes - 4 bytes]

[k pacotes concatenados contiguamente - n bytes]

[tabela de pacotes - k*12 bytes]

Nesta tabela de pacotes há a localização no arquivo, o tamanho e a informação sobre o tamanho de cada cabeçalho dos pacotes, indexados por seus números de identificação.

As funções que executam estas tarefas são:

- int put_in_a_file(char *fname, struct pack *l_pack, u_int32_t nop);

fname (argumento de entrada): nome do arquivo;

l_pack (argumento de entrada): lista ligada dos pacotes a serem colocados no arquivo;

nop (argumento de entrada): número de pacotes;

Inicialmente o montador percorre o arquivo em que estão definidos os pacotes formando uma lista ligada dos pacotes em binário a serem colocados em arquivo. Então a função "put_in_a_file" é chamada para armazená-los em disco.

- struct pack_vect *load_packs(char *fname);
  fname (argumento de entrada): nome do arquivo;

Esta função carrega os pacotes armazenados em "fname" em um vetor indexado pelos números de identificação dos pacotes, assim eles estão prontos (ou quase, pois, checksum, ack number, etc, podem ter de ser calculados no momento de envio) em memória para serem enviados.

Uma facilidade provida pela linguagem C é a possibilidade de se manusear bits dentro de um conjunto de bytes (operações bit-a-bit). A seguinte função insere, dado um offset em bits, informação em um conjunto de bytes apontado por "pack".

```
void pack_it(unsigned char *pack, u_int32_t info, u_int16_t off, u_int16_t size){
    u_int16_t myword = off/32;
    u_int8_t  start_bit= off%32;
    u_int32_t *wrk = (u_int32_t *)pack;

    wrk[myword] &= ~(mask(size)<<(start_bit));    (1)
    wrk[myword] |= info<<start_bit;               (2)
}
```

A variável "myword" armazena qual conjunto de 4 bytes dentro do pacote que será manuseado. Em "start_bit" está o bit, dentro destes quatro bytes escolhidos, que iniciará a seqüência a ser inserida. Na primeira linha os bits a serem preenchidos são "zerados" e na segunda linha a informação é inserida no pacote.

Esta descrição acima feita justifica a limitação de 4 bytes para o tamanho máximo de um campo e a não possibilidade de bits de um mesmo campo estarem fragmentados entre conjuntos de quatro bytes distintos.

Abaixo segue um exemplo para ilustrar o acima descrito:

```
out_pack_file : "out_pack" {
 packet #1 (40 bytes) {
  header : IP {
     hlen       5
     ver        4
     TOS        0
     totlen     40
     id         620
     frag       0x4000
     TTL        64
     proto      6
     cksum      0x2060
     saddr      10.1.2.1
     daddr      10.1.2.2
  }
  header : TCP {
     sport      1036
     dport      79
     seqnum     0x2
     acknum     0
     hlen       5
     unused     0
     flags      0x2
     win        32120
     cksum      0x1609
     urgp       0
  }
 }

 packet #2 (40 bytes) {
```

```
    header : IP {
        hlen        5
        ver         4
        TOS         0
        totlen      40
        id          620
        frag        0x4000
        TTL         64
        proto       6
        cksum       0x2060
        saddr       10.1.2.3
        daddr       10.1.2.4
    }
    header : TCP {
        sport       1036
        dport       79
        seqnum      0x2
        acknum      0
        hlen        5
        unused      0
        flags       0x2
        win         32120
        cksum       0x1609
        urgp        0
    }
  }
}
```

O que resulta em um arquivo com a seguinte configuração binaria (112 bytes):

[número de pacotes = 2:4 bytes]

[offset da tabela de pacotes = 88: 4bytes]

[2 pacotes: 80 bytes]
[informação sobre a posição dos pacotes: 24 bytes]

3.4 O RECEPTOR

Uma importante característica desta plataforma é a recepção de pacotes provenientes da rede e a 'decisão' do rumo a ser tomado a partir deste estímulo. Assim, uma gramática foi desenvolvida para que o usuário possa configurar como a plataforma deve agir. A decisão deve ser tomada através de comparações de campos dos pacotes definidos pelo usuário, como abaixo exemplificado:

```
    state HELLO recv_pack (RID = 1) {
     bpf = "tcp and port 21"
     ( next state:  AGAIN
       layers:      layer 1 = IP
       check:       TCP.sport = 123 and
                    TCP.dport = 21
     ) or
     ( next state:  ADIOS
       check:       IP.saddr = 10.1.1.1 and
                    IP.daddr = 10.1.1.2
     )
};
```

O exemplo acima esta definindo a ação de receber pacotes do estado HELLO. O pacote recebido terá o número de identificação (RID) 1. O filtro BPF (Berkeley Packet Filter) está sendo utilizado com a seguinte política: "tcp" (apenas pacotes utilizando protocolo TCP serão considerados); "port 21" (apenas pacotes com porta (destino ou origem) 21 serão considerados).

Na primeira política declarada, a maquina de estados é levada ao estado AGAIN caso um pacote com o campo sport do cabeçalho TCP seja 123 e o campo dport do mesmo seja 21. Na política seguinte, a maquina é levada para o estado ADIOS caso o campo saddr

do cabeçalho IP seja 10.1.1.1 (0x0a010101) e o campo daddr do mesmo seja 10.1.1.2 (0x0a010102). A preferência de políticas vem de baixo para cima, ou seja, a ultima declarada é a primeira a ser testada.

O campo "layers" também se faz necessário pois é preciso saber qual (ou quais) cabeçalho(s) está (ou estão) acima do cabeçalho a ser examinado, pois a informação previamente declarada sobre os cabeçalhos provê apenas o "offset" relativo ao início do cabeçalho em questão, e portanto é necessário saber qual o "offset" do cabeçalho dentro do pacote para calcular a posição absoluta do campo a ser examinado.

A filtragem está sendo utilizada através da biblioteca pcap (packet capture) também utilizada pelo TCPDUMP e terá um enfoque mais aprofundado no momento em que o executor da máquina de estados for construído.

3.5 CHECKSUM

Apesar de se estar operando os pacotes TCP/IP em um nível mais baixo, sem abstração de protocolos, campos, etc, mas apenas os considerando como um conjunto de bits em um aglomerado de bytes, há momentos em que é preciso lidar com informações específicas de um protocolo como por exemplo o cálculo do campo checksum.

Para não sair da filosofia de não se utilizar a palavra "protocolo", que seria um nível um pouco superior ao enfoque que se esta tomando neste projeto, algumas funções de cálculo de checksum foram construídas e são providas como métodos de cálculo do mesmo. Dessa forma o usuário deve informar qual o método de cálculo de checksum a ser utilizado no cabeçalho.

4. STATUS

Primeiro Ano	Jan	Fev	Mar	Abr	Mai	Jun	Jul	Ago	Set	Out	Nov	Dez
Estudo inicial	o	o										
Filtros de captura de pacotes e de acesso	o	o	o									
Estados e temporização				o	o	o	o	o	o	o	-	-
Gerador de pacotes										o	o	o

o: Parte do cronograma executado.

-: Parte restante do cronograma.

Por este projeto ter sido idealizado para uma duração de dois anos e por haver a necessidade de pedido de renovação, este relatório está sendo entregue antes do prazo determinado, e assim faltam alguns ajustes para que o cronograma do primeiro ano seja completado. Nestes últimos dias a integração de todos os módulos prontos será feita, o que resultará em uma primeira versão da plataforma.

4.1 Plano de Trabalho e Cronograma para a Etapa Seguinte

Este projeto foi concebido para uma duração de dois anos, e como foi anteriormente declarado em outro relatório, esta próxima etapa seria a fase da criação do ambiente (gráfico ou não) e de aplicação da plataforma como uma ferramenta de teste de segurança de redes. Isto será possível pois toda a base da plataforma estará pronta, totalmente configurável por uma linguagem descritiva de estados.

Nesta segunda etapa serão discutidas formas de interfaces para uma manipulação amigável do usuário com a plataforma. Apesar da linguagem obtida até o momento ser razoavelmente de alto nível, este projeto tem o objetivo de abstrair ao máximo o usuário da implementação em linguagem de programação.

Possuindo todo o ambiente pronto para ser utilizado, inicia-se uma série de testes para verificar a corretude da plataforma. Neste caso, algumas funcionalidades da pilha TCP/IP e de algumas aplicações serão implementadas através da plataforma. Assim poderá ser verificado se a plataforma atuará como o desejado.

Estando funcionando como o esperado, uma série de ataques conhecidos serão executados através da plataforma para verificar a eficiência da mesma como teste de segurança e integridade dos componentes em baixo nível de uma rede de computadores (normalmente a implementação da pilha TCP/IP).

Então, ao final deste projeto, existirá uma ferramenta capaz de manipular cada bit da estrutura atômica de uma rede de computadores, o pacote, e colocá-lo em um contexto real para se testar a resposta de outros sistemas contidos na rede mediante um valor inesperado em algum campo de algum cabeçalho.

Segundo Ano	Fev	Mar	Abr	Mai	Jun	Jul	Ago	Set	Out	Nov	Dez	Jan
Interface e ambiente	x	x	x	x	x	x						
Teste de funcionalidades elementares de pilhas TCP/IP e aplicações						x	x	x	x			
Utilização dos ataques para teste								x	x	x	x	
Possíveis ajustes à plataforma										x	x	x

E assim faz-se um pedido de renovação da bolsa de IC à FAPESP por um prazo de 1 ano.

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.

Fyodor. The Art of Port Scanning e Remote OS Detection via TCP/IP Fingerprinting, 1997 e 1999, http://www.insecure.org.

Request for Coments e artigos científicos

Linux Documentation Project, *Howtos e Mini-howtos, http://www.linuxdoc.org*