DISSIMULAÇÃO DE HONEYPOTS VIRTUAIS UTILIZANDO USER-MODE LINUX

Eduardo Fernandes Piva[*]
Instituto de Computação
Univ. Estadual de Campinas
13083-970 Campinas - SP
eduardo@las.ic.unicamp.br

Martim d'Orey Posser de Andrade Carbone[†]
Instituto de Computação
Univ. Estadual de Campinas
13083-970 Campinas - SP
martim@las.ic.unicamp.br

Paulo Lício de Geus
Instituto de Computação
Univ. Estadual de Campinas
13083-970 Campinas - SP
paulo@las.ic.unicamp.br

RESUMO

Honeypots virtuais apresentam uma série de vantagens com relação a honeypots físicos no tocante a possibilidades de coleta de dados, custos de implantação e portabilidade. Possuem, no entanto, a desvantagem de que podem ser facilmente desmascarados se um intruso descobrir que a máquina invadida é uma máquina virtual, já que estas ainda são pouco utilizadas como servidores de produção. Este artigo propõe uma série de contra-medidas de implementação e de configurações para o gerenciador de máquinas virtuais User-Mode Linux*, de forma a fazer com que honeypots virtuais utilizando o mesmo aparentem ser sistemas Linux autênticos executados em computadores físicos, iludindo assim os intrusos a pensar que invadiram uma máquina de produção autêntica.*

ABSTRACT

Virtual honeypots have many advantages over physical honeypots with regards to data collection capabilities, deployment costs and portability. They do have, however, the disadvantage that they can be easily discovered if an intruder finds out that the compromised system is a virtual machine, since these are still not very popular as production servers. This paper proposes a series of implementation and configuration counter-measures for the User-Mode Linux VM manager, so that UML virtual honeypots appear to be authentic Linux systems running on physical computers, making the intruders believe that they have compromised authentic production machines.

1 Introdução

Um *honeypot* é definido como *um recurso computacional cujo valor está na sua sondagem, ataque ou comprometimento por invasores* [14, 13]. No cenário atual da segurança da informação, *honeypots* são reconhecidos como ferramentas de segurança potencialmente valiosas no estudo dos hábitos, ferramentas e psicologia dos *blackhats* [18]. Por não serem utilizados para fins de produção e serem totalmente dedicados ao mau-uso por parte de atacantes (eliminando assim a ocorrência de falsos positivos), *honeypots* constituem excelentes ferramentas para coleta de dados oriundos de atacantes.

Desde o ressurgimento da pesquisa em *honeypots*, em 1999, com a criação do *Honeynet Project*[1], um volume crescente de pesquisa tem sido realizado neste assunto, dando origem a diversos paradigmas de instalação, configuração e implantação de *honeypots*, cada um com seus méritos e ajustado para propósitos específicos. Estes podem ser divididos naqueles de *alta interatividade* e *baixa interatividade* dependendo do nível de interatividade que os atacantes tem com os mesmos (se chegam a invadi-los ou não). Também é possível particioná-los entre *físicos* e *virtuais*, dependendo da forma como são implementados.

Enquanto *honeypots* físicos seguem a abordagem convencional, sendo implementados como máquinas reais dotados de um sistema operacional e processos de usuário com os quais o atacante interage, *honeypots* virtuais [16] seguem uma abordagem diferente, sendo implementados como máquinas virtuais [12]. Neste artigo, chamaremos de gerenciador de máquinas virtuais qualquer mecanismo que forneça uma infraestrutura para que um sistema operacional *convidado* possa ser executado sobre um sistema operacional *hospedeiro*, sendo este último o sistema operacional da máquina física.

As vantagens na adoção de uma arquitetura baseada em máquinas virtuais na implantação de um *honeypot* são numerosas [18], e as principais são:

- **Custo de hardware reduzido**: Pode-se simular uma rede inteira de *honeypots* (constituindo assim uma *honeynet* virtual) em apenas um computador físico, através da execução simultânea de diversas máquinas virtuais;
- **Rápida restauração**: Os sistemas de arquivos de máquinas virtuais residem em arquivos

[*]Bolsista CAPES
[†]Bolsista CNPq
[1]http://www.honeynet.org

simples na máquina hospedeira, podendo assim serem restaurados rapidamente;

- **Portabilidade**: Pode-se configurar uma *honeynet* virtual inteira em um computador portátil, e conectá-lo a diferentes redes;
- **Monitoração facilitada**: É possível monitorar facilmente os eventos de uma máquina virtual, coletando dados sobre as atividades dos intrusos. Essa monitoração tende ser mais transparente que aquelas utilizadas em máquinas físicas, e mais difícil de ser detectada e desabilitada por um atacante.

Para a utilização de *honeypots* virtuais, podem ser utilizados gerenciadores de máquinas virtuais que implementam uma camada completa de virtualização, como o VMWare [3, 17] ou sistemas como o *User-Mode Linux* (UML) [2, 15], que implementam versões modificadas do *kernel* do Linux, sem entretanto recorrer a técnicas mais sofisticadas de virtualização. Os autores optaram pela utilização do UML tanto neste projeto como nos futuros, por este ser gratuito, de código fonte aberto e possuir recursos interessantes para uso em *honeypots* (a serem vistos na Seção 2)

Apesar das vantagens citadas acima, *honeypots* virtuais também possuem alguns problemas que limitam a sua disseminação. Entre eles, está o risco de que um atacante consiga romper o isolamento da máquina virtual e comprometer a segurança da máquina hospedeira; e a incapacidade de se emular sistemas mais exóticos (como roteadores Cisco, por exemplo, ou arquiteturas menos conhecidas) com máquinas virtuais [18].

O principal problema, no entanto, diz respeito à facilidade com que se pode reconhecer uma máquina virtual. Até o presente momento, máquinas virtuais não são utilizadas em larga escala para fins de produção na Internet, e assim, se um intruso descobrir que um sistema invadido se trata de uma máquina virtual, ele poderá suspeitar que o sistema é um *honeypot* [5]. Esta suspeita deve ser evitada a todo custo, pois pode comprometer seriamente a eficácia do *honeypot*. E ainda, o fato de que os atacantes já estão começando a ficar atentos para a autenticidade das máquinas invadidas (para evitar que sejam enganados por *honeypots*) [10, 9, 6, 7] só aumenta a gravidade do problema.

Tendo isso em mente, este artigo se propõe a abordar o problema de dissimulação de *honeypots* virtuais UML de alta interatividade. Ao longo do artigo, pretende-se abordar a maioria das técnicas já conhecidas para o reconhecimento de máquinas virtuais UML e propor contra-medidas que inibam estas técnicas de reconhecimento, enganando o intruso e dando a impressão de que este invadiu uma máquina física autêntica. É importante salientar, porém, que não é objetivo deste artigo descrever um sistema que dissimule perfeitamente *todos* os vestígios deixados pelo UML, pois estes são numerosos e com certeza há muitos que nem sequer são conhecidos. Pretende-se, sim, atenuar este problema, propondo uma série de técnicas de implementação/configuração que inibam a eficácia da geração atual de técnicas de reconhecimento, ganhando assim uma vantagem com relação aos intrusos [11].

Este artigo está organizado da seguinte maneira: inicialmente, na Seção 2, será descrito o funcionamento básico do UML, assim como os recursos de dissimulação voltados a *honeypots* que o mesmo já apresenta. Em seguida, na Seção 3, serão descritas as principais técnicas utilizadas no reconhecimento do UML, e serão propostas contra-medidas para inibir estas técnicas. Finalmente, na Seção 4, serão feitas algumas considerações finais sobre o trabalho realizado e serão propostos alguns trabalhos futuros.

2 User-Mode Linux

O UML consiste em um *patch* para o código fonte do kernel do Linux. Este *patch* modifica o kernel de tal forma, fazendo com que este kernel (convidado) possa ser executado como um processo de usuário em cima de outro kernel Linux (hospedeiro), constituindo assim uma máquina virtual. Com isso, consegue-se executar múltiplas instâncias do sistema operacional Linux na mesma máquina física simultaneamente. Ao contrário de gerenciadores como o VMware, o UML não implementa camadas mais sofisticadas de virtualização (emulando dispositivos de hardware, por exemplo).

A grande diferença entre um kernel Linux comum e outro modificado pelo UML é que, enquanto o primeiro atua como intermediário entre processos comuns e o hardware da máquina, o segundo é intermediário entre processos comuns e o kernel hospedeiro (Figura 1). Assim, costuma-se dizer que UML é uma versão do kernel Linux para a plataforma Linux. O kernel convidado (UML) atua interceptando as chamadas de sistema dos processos executados em seu contexto, realizando as modificações necessárias e enviando-as ao kernel hospedeiro. Em seguida, intercepta os códigos de retorno do kernel hospedeiro e os envia de volta aos processos que invocaram as chamadas originais.

O UML foi originalmente projetado e implementado para atuar como uma plataforma de testes de aplicações e de depuração de kernel. Suas aplicações, entretanto, de forma alguma se restringem a estes dois ítens, e recentemente a equipe de desenvolvimento do UML iniciou o trabalho em funcionalidades específicas para seu uso como *honey-*

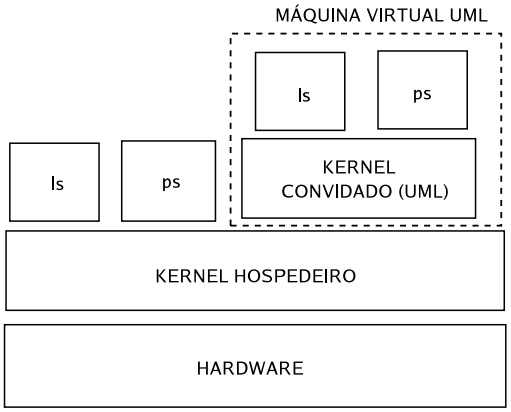

Figura 1: Funcionamento do User-Mode Linux

pots virtuais [5, 15]. Estas funcionalidades serão detalhadas nas próximas seções. Ao que tudo indica, esse é o único trabalho correlato em dissimulação de honeypots UML.

2.1 TTY logging

O UML altera o gerenciador de terminais do kernel, fazendo com que este capture e armazene na máquina hospedeira todos os comandos digitados por um usuário na máquina virtual ou, no caso de um *honeypot* virtual, um invasor [5]. Esta captura é feita em nível de kernel, o que dificulta muito a sua detecção e neutralização por parte de um invasor.

2.2 Separate Kernel Address Space (SKAS)

Em seu modo de operação padrão, o UML modifica o espaço de endereçamento dos processos da máquina virtual, fazendo com que o kernel UML ocupe os 500MB superiores do espaço de usuário (0xa0000000 - 0xc0000000). Essa abordagem possui diversas desvantagens, pois além de distorcer o espaço de endereçamento tradicional dos processos no Linux (o que poderia criar suspeitas por parte de um intruso), também possibilita que o estado do kernel seja lido e alterado sem qualquer restrição. Recentemente, a introdução da funcionalidade SKAS (ativado a partir de um *patch* que se aplica no kernel do sistema hospedeiro) fez com que o kernel UML passasse a residir em um espaço de endereçamento totalmente isolado de seus processos. Com isso, o espaço de endereçamento dos processos se tornou idêntico ao daqueles executados em uma máquina física e o kernel UML se tornou invisível e inviolável por processos executando no contexto da máquina virtual [5].

2.3 Honeypot Proc Filesystem (HPPFS)

Em um sistema Linux comum, o diretório `/proc` contém um sistema de arquivos virtual, denominado `proc`, que contém informações sobre configurações de kernel e dispositivos de hardware reconhecidos por este. No caso de uma máquina virtual UML, que não emula dispositivos de hardware e tem acesso limitado àqueles da máquina hospedeira, diversos arquivos que normalmente estariam presentes no sistema `proc` de máquinas físicas, estarão ausentes no `proc` de máquinas UML. Além disso, diversos arquivos seriam marcados com entradas referentes aos dispositivos utilizados pelo UML, como dispositivos UBD e interfaces de rede TUN/TAP. Tudo isso tornaria simples a identificação de uma máquina virtual UML através da análise do conteúdo de `proc`. Para resolver este problema, a equipe desenvolvedora do UML criou um sistema de arquivos `proc` especial, denominado *Honeypot Proc Filesystem* (HPPFS) [5], que possibilita a manipulação de quaisquer informações presentes no mesmo. Com isso, torna-se possível moldar o conteúdo do `proc` do *honeypot* virtual de forma a fazer com que o mesmo aparente ser característico de uma máquina física, através da inclusão, remoção e alteração de arquivos.

3 Reconhecimento e Dissimulação de UML

Muito embora o UML apresente funcionalidades para dissimular máquinas virtuais, como visto na Seção 2, cada uma delas trata de problemas de dissimulação específicos (do `proc` no caso do HPPFS e do espaço de endereçamento no caso do SKAS), deixando de lado outros vestígios não menos importantes. Além disso, no caso do HPPFS, a própria presença deste acaba criando novos vestígios, que devem ser ocultados a fim de que o invasor não note a sua presença.

Esta Seção foi organizada em quatro subseções. Na Seção 3.1 serão apresentadas as principais técnicas de reconhecimento baseadas no HPPFS e as modificações realizadas no mesmo, com vistas a configurá-lo para dissimular o sistema de arquivos `proc` e esconder os vestígios de sua presença. Em seguida, na Seção 3.2, técnicas de detecção e dissimulação do UBD (*User Block Device*), o dispositivo de bloco utilizado pelo UML, serão discutidas. Na Seção 3.3 os mecanismos utilizados para ocultar evidências do UML no *log* do kernel serão explicados. Finalmente, na Seção 3.4, alguns ítens de configuração especiais para dissimulação de um *honeypot* virtual serão discutidos. Em todos os testes, foi utilizada a versão 2.4.26 do kernel do Linux, tendo sido aplicados no mesmo os *patches* UML para esta mesma versão.

3.1 Modificações no HPPFS

As modificações realizadas no HPPFS foram divididas em dois grupos: modificações na configuração do HPPFS, e modificações na implementação do HPPFS.

As modificações na configuração do HPPFS foram realizadas com o intuito de esconder vestígios nos arquivos do sistema de arquivos `proc` que a configuração padrão do HPPFS não dissimula.

As modificações na implementação do HPPFS no kernel foram realizadas de forma a eliminar as evidências criadas pela presença do próprio HPPFS.

Nas seções seguintes serão analisadas as configurações realizadas no HPPFS, as técnicas de detecção de um sistema de arquivos do tipo HPPFS e por último, como dissimular a presença do HPPFS.

3.1.1 Configuração do HPPFS

O HPPFS é acompanhado por um script escrito em *perl*, responsável pela criação de dados forjados para o sistema de arquivos *proc*. Esse script, localizado no diretório `honeypot` do pacote `uml_tools`, atua criando diversos arquivos sockets no sistema hospedeiro, utilizados para negociar as saídas a serem exibidas nos arquivos do `proc` da máquina virtual, a partir de uma negociação de dados entre o sistema hospedeiro e a máquina virtual.

Existem basicamente dois tipos de negociações realizadas entre a máquina virtual e o sistema hospedeiro, no HPPFS. Pode-se clonar dados de um sistema de arquivos *proc* real em um sistema virtual ou filtrar informações do *proc* virtual, através do uso de expressões regulares.

As modificações realizadas no script *perl* fornecido pelo `uml_tools` foram feitas como extensões dos filtros de expressões regulares, removendo indícios da presença de dispositivos que tipicamente não estão presentes em máquinas reais, e que a configuração padrão do HPPFS não remove, como:

- `/proc/misc`: A linha contendo o string `net/tun` foi removida. Essa linha evidencia a existência de uma ou mais interfaces de rede do tipo TUN/TAP, muito utilizadas como interfaces de redes virtuais (emuladas) em máquinas virtuais UML;
- `/proc/devices`: Foi removida a linha que continha o string `vc`. Essa palavra evidencia a existência de consoles do tipo *virtual console*, utilizados principalmente em instalações de máquinas virtuais UML, e a sua presença pode levantar suspeitas em um atacante.

Este script *perl* está reproduzido integralmente no Apêndice A deste artigo.

3.1.2 Detecção do HPPFS

Após ocultar os vestígios comprometedores deixados nos arquivos do `proc`, alterando a configuração padrão do HPPFS, deve-se ocultar os vestígios deixados pela presença do próprio HPPFS. Afinal, a detecção da existência do HPPFS por parte de um intruso levantaria suspeitas imediatas de que a máquina se trata de um *honeypot*. Uma das maneiras de se detectar a presença do HPPFS é através da execução dos seguintes comandos:

```
mkdir /tmp/proc
mount none -t proc /tmp/proc
```

Após executar os comandos acima, compara-se o conteúdo dos diretórios `/tmp/proc` (onde estará montado o sistema de arquivos `proc` original do UML, sem quaisquer alterações provocadas pelo HPPFS) e `/proc` (onde estará montado o HPPFS). Qualquer diferença entre os dois evidenciará a presença do HPPFS.

Outra maneira de detectar a presença do HPPFS é simplesmente tentando montá-lo, através do seguinte comando:

```
mount none -t hppfs /tmp/proc
```

Caso o comando `mount` execute com sucesso, a presença do HPPFS no sistema terá sido comprovada.

3.1.3 Dissimulação do HPPFS

Para que não seja possível detectar a presença de um sistema de arquivos do tipo HPPFS, não se deve permitir que este sistema seja montado ou desmontado manualmente, conforme foi feito na Seção 3.1.2. Isto é, deve-se fazer com que a montagem deste sistema de arquivos ocorra de forma automática e transparente.

Para tanto, o mecanismo de montagem e desmontagem de sistemas de arquivos foi alterado. Foi utilizada a seguinte técnica: quando a montagem do sistema de arquivos *proc* for solicitada, este será montado, e logo em seguida o HPPFS será montado automaticamente sobre ele. Da mesma maneira, quando se desmontar um sistema *proc*, o sistema HPPFS, que foi montado sobre o *proc* de forma transparente, também será desmontado automaticamente. Com essa técnica, consegue-se inibir a primeira técnica de reconhecimento descrita na Seção 3.1.2, pois o intruso nunca conseguirá montar o sistema de arquivos *proc* original sem que o HPPFS seja montado sobre ele.

Para que o HPPFS não pudesse ser montado ou desmontado manualmente (inibindo assim a eficácia

da segunda técnica de reconhecimento descrita na Seção 3.1.2), fez-se com que o suporte a este ficasse desregistrado do kernel em todos os momentos, a não ser naqueles em que o sistema *proc* está sendo montado. Para tanto, modificou-se a função *init_hppfs*, em `arch/um/fs/hppfs/hppfs_kern.c`, desabilitando o registro automático do suporte ao HPPFS durante a inicialização do kernel. Feito isso, criou-se duas funções, *register_hppfs_support* e *unregister_hppfs_support*, responsáveis por registrar e desregistrar dinamicamente, sob demanda, o suporte ao HPPFS no VFS (*Virtual File System*) [4], através das funções *register_filesystem* e *unregister_filesystem*. Essas novas funções foram exportadas no kernel, possibilitando que fossem invocadas de qualquer ponto deste.

```
void unregister_hppfs_support(void)
{
unregister_filesystem(&hppfs_type);
}

void register_hppfs_support(void)
{
register_filesystem(&hppfs_type);
}
```

As funções *sys_umount* e *do_mount* (`fs/namespace.c`), responsáveis pela montagem e desmontagem de sistemas de arquivos, também foram alteradas, de forma a agir da maneira descrita anteriormente.

O código abaixo é a parte principal do código inserido na função *do_mount*.

```
if (!retval &&
    !strcmp(type_page, "proc")) {
  /* registro do sistema HPPFS */
  register_hppfs_support();
  retval = do_mount(dev_name,
                    dir_name,
                    "hppfs",
                    flags,
                    data_page);
  /* desregistro do sistema HPPFS */
  unregister_hppfs_support();
}
```

O código acima opera da seguinte maneira: se a operação a ser realizada for a de adição de um ponto de montagem[2], verifica-se se o sistema de arquivos é do tipo *proc*. Caso seja, e a montagem do sistema *proc* tenha ocorrido com sucesso (verificado através da variável `retval`), registra-se o suporte a HPPFS, monta-se o sistema HPPFS no mesmo diretório (atualizando o código de retorno da função *do_mount*), e, em seguida, retira-se o suporte a HPPFS do kernel. Isso faz com que o suporte a sistemas HPPFS esteja habilitado no kernel apenas durante o breve instante em que o sistema HPPFS está sendo montado e com isso inibe a eficácia da segunda técnica de detecção.

As modificações realizadas na função *sys_umount* contém lógica semelhante à de *do_mount*, contendo apenas algumas diferenças de implementação, devido às características do VFS. Abaixo segue o trecho mais significativo modificado:

```
parent = nd.mnt->mnt_parent;
retval = do_umount(nd.mnt, flags);

if(!retval &&
   !strcmp(parent->
           mnt_sb->
           s_type->name, "proc")) {
        retval = sys_umount(name, flags);
}
```

No código acima, a função *sys_umount* recebe dois parâmetros: o nome do diretório a ser desmontado, e as *flags* a serem utilizadas. Dentro dessa função, obtém-se, através do nome do diretório fornecido, uma estrutura do tipo *struct nameidata* (variável *nd*), onde estão todas as informações do diretório. Através da variável *nd*, obtém-se uma referência para uma estrutura do tipo *struct *vfsmount* (variável *parent*). Essa estrutura corresponde ao sistema de arquivos que está montado por baixo do sistema de arquivos que está sendo desmontado[3] [4]. Tendo essas informações, verifica-se se o sistema que está por baixo do sistema desmontado corresponde a um sistema do tipo *proc*. Caso afirmativo, e o sistema que estava sobre o *proc* foi desmontado com sucesso (no caso, o HPPFS), desmonta-se o mesmo diretório novamente. Com isso, consegue-se manter a ilusão se que o HPPFS representa o sistema de arquivos *proc* original da máquina, e consegue-se esconder do intruso o verdadeiro sistema *proc* da máquina virtual.

Para desmontar um ponto de montagem HPPFS não é necessário registrar o suporte a HPPFS no kernel, pois as referências às funções necessárias para desmontar esse ponto são determinadas na montagem do dispositivo, momento no qual o suporte a HPPFS está habilitado.

3.2 Dispositivos de bloco

Dispositivos de bloco (*block devices*) são arquivos especiais que são mapeados, de forma transparente, em um dispositivo de hardware. Dispositivos de

[2]Pois pode-se executar a função *do_mount* com o intuito de mover ou remontar um dispositivo

[3]Sempre há um, ainda que seja um pseudo-sistema de arquivos fornecido pelo VFS

bloco podem ser arquivos especiais criados estaticamente, através do comando `mknod`, ou arquivos virtuais criados dinamicamente, através de um sistema de arquivos virtual denominado *devfs*. Em ambos os casos, o funcionamento dos arquivos é semelhante, sendo que as duas maneiras podem coexistir. Neste artigo será abordada apenas a primeira abordagem de dispositivos de bloco já que a segunda ainda é pouco utilizada nas distribuições de Linux, e as modificações a serem feitas são análogas às da primeira abordagem.

Os arquivos especiais de dispositivos de bloco possuem dois atributos importantes, denominados identificador primário (*major number*) e identificador secundário (*minor number*). O identificador primário é utilizado para se identificar um dispositivo físico, e o identificador secundário é utilizado para se identificar partes de um dispositivo. No caso de discos IDE, o identificador primário identifica qual a porta IDE em que o disco está conectado, e o secundário identifica o disco inteiro, se 0 ou 64, ou alguma de suas partições, caso contrário [4]. Os identificadores primários são padronizados, possuindo valores fixos, definidos em `include/linux/major.h`.

No caso do UML, todos os discos são mapeados através de um dispositivo de bloco especial, de nome UBD (*User Block Device*). O UBD é capaz de mapear um arquivo de dados no sistema hospedeiro como se fosse um dispositivo de bloco na máquina virtual. Para tanto, utiliza-se o parâmetro *ubdX* durante a inicialização da máquina virtual, onde X denomina o valor que será utilizado como identificador secundário. Por padrão, o identificador primário de dispositivos UBD é 93.

Maneiras de se detectar a presença de um dispositivo UBD serão apresentadas na Seção 3.2.1, e na Seção 3.2.2 serão abordadas as técnicas para dissimular essas evidências.

3.2.1 Detecção de UBD

A primeira evidência de que dispositivos UBD estão sendo utilizados surge após a análise da listagem dos dispositivos que estão montados em um sistema UML. A existência do dispositivo `/dev/ubd0` montado na raiz do sistema (/) evidencia claramente o uso de dispositivos UBD. Mesmo que este dispositivo seja renomeado, o uso de UBD é denunciado pelo seu identificador primário, que pode ser obtido através do comando `ls`:

```
bash-2.05# ls -lh /dev/hda
brw-rw----    1 root     disk\
93,   0 Aug 30  2001 /dev/hda
```

Vê-se na saída acima que o dispositivo `/dev/hda` possui como identificador primário o valor 93, utilizado pelo UBD. Caso realmente fosse um dispositivo IDE, teria como identificador primário o valor 3.

Outra maneira de se identificar o UBD é efetuando uma listagem nas partições presentes no sistema, através do comando `fdisk -l`. Através deste comando, nota-se que o sistema não possui partições, o que é uma situação inexistente em sistemas reais.

A última evidência é fornecida pela análise do resultado da invocação da chamada de sistema *ioctl*. Essa chamada de sistema é utilizada na comunicação direta com dispositivos físicos. Como o UML não implementa uma camada de emulação de hardware, o sistema não responde a alguns comandos específicos da chamada *ioctl* da forma esperada, em especial a comandos do protocolo ATAPI [1].

Um exemplo simples de detecção de um dispositivo UBD através da utilização da chamada *ioctl* é através do comando `hdparm` (que se comunica com os dispositivos através desta chamada) [6], conforme segue abaixo:

```
bash-2.05# hdparm -I /dev/hda

/dev/hda:
 HDIO_DRIVE_CMD(identify)
    failed: Invalid argument
```

3.2.2 Dissimulação de UBD

Para se dissimular um dispositivo UBD, simulou-se a existência de um dispositivo IDE, dotado de uma tabela de partições e um esquema de partições factível.

Para isso, inicialmente trocou-se o valor da constante UBD_MAJOR pelo valor de IDE0_MAJOR em `include/linux/major.h`, a fim de se inibir a primeira técnica de detecção descrita na seção anterior. Feita a troca, criou-se então um arquivo em branco, no sistema hospedeiro, da seguinte maneira:

```
dd if=/dev/zero of=/tmp/block_device\
                bs=1048576\
                count=1000
```

Tendo criado o arquivo, inicializou-se o sistema virtual, utilizando como dispositivo `ubd0` o sistema de arquivos original, e como `ubd1` o arquivo recém-criado, conforme ilustrado abaixo:

```
./linux ubd0=root_fs \
  ubd1=/tmp/block_device
```

Após inicializar o sistema, este ficou com sua raiz em `/dev/hda` (identificador primário 3 e identificador secundário 0), e um dispositivo vazio com identificador primário 3 e secundário 16 (mapeando `/tmp/block_device`). Para acessá-lo, foram criados arquivos de dispositivos de blocos especiais, e o mesmo foi particionado. Os comandos abaixo criam arquivos especiais para o dispositivo e duas partições (identificadores secundários 16, 17 e 18, respectivamente).

```
mknod /dev/disco b 3 16
mknod /dev/disco1 b 3 17
mknod /dev/disco2 b 3 18
```

O particionamento foi feito da mesma maneira que em um disco real, através do comando `fdisk`, criando-se uma partição para a raiz do sistema e outra para a área de *swap* . Realizou-se então a formatação das partições mapeadas em `/dev/disco1` e `/dev/disco2`, através dos comandos `mke2fs` e `mkswap`. Fez-se então a clonagem de ambos os sistemas de arquivos, copiando o conteúdo de `/dev/hda` (sistema de arquivos original) para `/dev/disco1` (sistema de arquivos recém-criado). Em seguida, editou-se o arquivo `/etc/fstab`, colocando como partição raiz o dispositivo `/dev/hda1`, e como área de *swap* o dispositivo `/dev/hda2`. Reiniciou-se então a máquina virtual com o seguinte comando:

```
./linux ubd0=/tmp/block_device
```

Com isso, o dispositivo antes identificado como `/dev/disco`, com partições `/dev/disco1` e `/dev/disco2`, agora foi reconhecido como disco `/dev/hda`, com partições `/dev/hda1` e `/dev/hda2`.

Com essas medidas, neutralizou-se a eficácia das duas primeiras técnicas de detecção descritas na seção anterior, pois conseguiu-se criar um dispositivo `/dev/hda`, com identificadores primários e secundários coerentes, e dotado de um esquema partições comum a um disco rígido real.

Para que a simulação ficasse completa, entretanto, restou a criação de uma contra-medida para a terceira técnica de reconhecimento descrita anteriormente. Para tanto, implementou-se a simulação de respostas à chamada de sistema *ioctl*, que em uma máquina real possuindo discos IDE são geradas pelo próprio disco rígido, segundo o protocolo ATAPI.

Os comandos de disco cujas respostas foram simuladas são: HDIO_GET_IDENTITY (corresponde à opção -i do comando `hdparm`), HDIO_DRIVE_CMD com a opção IDENTITY (correspondente à opção -I do comando `hdparm`) e comandos do tipo *set/get* que manipulam atributos de DMA, *keep settings* e E/S de 32 bits.

Para que as respostas ficassem semelhantes às respostas geradas por discos reais, um aplicativo que solicita essas informações a dispositivos reais da máquina hospedeira foi escrito e executado. As respostas obtidas foram codificadas em estruturas de dados em C, que por sua vez foram armazenadas em arquivos *headers* de C, para que pudessem ser utilizadas como modelos de resposta. A função *ubd_ioctl*, contida em `arch/um/driver/ubd_kern.c`, foi estendida de forma a responder aos comandos utilizados na chamada de sistema *ioctl* listados anteriormente. Para isso, utiliza quando necessário a estrutura de modelos de respostas obtida anteriormente, presente no *header* `arch/um/include/ubd_atapi.h`. O código abaixo implementa a resposta ao comando HDIO_GET_IDENTITY, utilizando a estrutura *identity_struct*, contida no *header* de respostas.

```
case HDIO_GET_IDENTITY:
        if(copy_to_user((char *) arg,
           identity_struct,
           sizeof(identity_struct)))
                return(-EFAULT);
        return(0);
```

Para os comandos do tipo *get/set*, variáveis estáticas foram utilizadas. Essas variáveis são inicializadas com valores padrões factíveis, e sempre que um comando do tipo *set* é efetuado, seu valor é atualizado. Dessa forma, caso um atacante realize uma seqüencia de interações com um disco IDE, a implementação de block device irá simular essa interação, fornecendo os valores corretos. O código abaixo ilustra essa implementação, para o caso da operação de E/S de 32 bits:

```
case HDIO_GET_32BIT:
        if(copy_to_user((char *) arg,
           (char *)&hdio_32bit,
           sizeof(hdio_32bit)))
                return(-EFAULT);
        return(0);

case HDIO_SET_32BIT:
        hdio_32bit = arg;
        return(0);
```

Essas medidas resultaram na neutralização da terceira técnica de reconhecimento descrita na seção anterior, pois agora a execução do comando `hdparm` fornece respostas factíveis a alguns de seus principais comandos, dando a impressão de que o sistema possui um disco rígido real, ao invés de um dispositivo UBD virtual.

3.3 Dissimulação no *log* do Kernel

O kernel do Linux possui um *log* circular em memória, que pode ser acessado a qualquer mo-

mento através do comando `dmesg`. Este *log* contém informações que podem ser utilizadas para depuração ou alerta de eventos, como falhas em dispositivos.

Este *log* é alimentado através da função de kernel *printk*, semelhante à função *printf* da biblioteca padrão C. Todo código escrito no kernel, desde *drivers* de dispositivos até o sistema básico, pode escrever entradas nesse *log* através desta função.

Durante a inicialização de uma máquina virtual, diversas mensagens de *log* são escritas no buffer de *log* do kernel. Essas mensagens trazem consigo um grande número de evidências de que o kernel executado é um kernel virtual. Essas evidências se tornam mais alarmantes quando se utiliza UML em conjunto com HPPFS, pois nesse caso os dados do sistema de arquivos `proc` e do *log* do kernel se tornam conflitantes.

Um mecanismo para eliminar essas evidências foi criado, utilizando a infra-estrutura provida pelo utilitário `mconsole` do UML. O `mconsole` é um utilitário para o gerenciamento de atributos do kernel virtual, através do sistema hospedeiro, utilizando *sockets* como meio de comunicação e um protocolo do tipo cliente/servidor para a troca de mensagens.

Foi implementada uma nova operação no `mconsole`, denominada `cleanlog`, através da modificação dos arquivos `arch/um/include/mconsole.h`, `arch/um/drivers/mconsole_user.c` e `arch/um/drivers/mconsole_kern.c`. O trecho de código responsável pela limpeza do *buffer* de *log* segue abaixo:

```
void mconsole_cleanlog(struct mc_request *req)
{
  do_syslog(5, NULL, 0);
  mconsole_reply(req, "", 0, 0);
}
```

Feita a limpeza do *buffer* de *log*, pode-se inserir dados fictícios no mesmo. Para isso, um modelo pode ser criado e inserido através do `mconsole`, utilizando o comando nativo `log`. Dado um modelo, é possível inseri-lo da seguinte maneira:

```
uml_console socket log -f modelo
```

Como a maioria das informações fornecidas pelo HPPFS são baseadas no sistema hospedeiro, a maneira proposta de se criar um modelo é a partir das mensagens de *log* da máquina hospedeira, utilizando a mesma versão de kernel do *honeypot*, porém sem os *patches* do UML. Dessa forma, teremos informações coerentes entre o `proc` e a saída do *log* do kernel, pois esse *log* irá conter informações sobre dispositivos, CPU e portas de E/S da máquina hospedeira. Para se extrair um modelo, basta executar o comando `dmesg > modelo`.

3.4 Configurações especiais para dissimulação

Além das técnicas para dissimulação descritas nas Seções 3.1 e 3.2, ainda há uma série de configurações especiais às quais um administrador de *honeypots* virtuais UML deve ficar atento a fim de não levantar suspeitas nos intrusos. Estas seguem abaixo:

- Desabilitar a utilização do sistema de arquivos *devfs*, utilizando no lugar arquivos de dispositivos criados manualmente com o uso do comando *mknod*. Este procedimento é necessário para que as técnicas descritas na Seção 3.2 funcionem, pois estas não contemplam o *devfs*;
- Antes de compilar o kernel UML, editar o arquivo `include/linux/version.h`, e retirar o sufixo *-um* de todas as linhas que especificarem versões do kernel. Este indicador apareceria na saída de comandos como o `uname` e é um indicador bastante claro de que o kernel sendo executado se trata de UML;
- Utilizar na máquina virtual o mesmo esquema de partições da máquina hospedeira, a fim de que as informações presentes em arquivos do diretório `/proc` e no *log* do kernel (que são baseados nos da máquina hospedeira) estejam condizentes com a formatação das partições virtuais;
- Colocar no diretório `/boot` da máquina virtual os arquivos `vmlinuz` e `System.map` da mesma versão de kernel utilizada pelo UML, substituindo eventuais versões antigas que já estejam presentes (padrões da distribuição de Linux instalada no sistema de arquivos). Fazer o mesmo com o código fonte do kernel no diretório `/usr/src/linux`, mas obviamente sem aplicar o *patch* do UML no mesmo. Com essas medidas, evita-se que haja discrepâncias entre a versão de kernel sendo executada e a versão em disco;
- Desabilitar o suporte ao carregamento de módulos. Isso evita que o intruso tenha acesso à tabela de símbolos do kernel (presente em `/proc/ksyms`), que pode ser utilizada para se identificar o UML; e que não possa ter acesso de leitura e escrita às estruturas internas do kernel;
- Desabilitar o suporte a interfaces do tipo *dummy*, removendo a opção `CONFIG_DUMMY` na configuração do kernel UML. Isso é necessário para que esta interface não apareça listada na saída do comando `ifconfig`, sugerindo que o sistema seja UML [18];
- Alterar o endereço MAC das interfaces de redes virtuais para algum diferente do padrão, evitando com isso o reconhecimento. Isso pode ser

feito no próprio comando de carregamento do UML, no parâmetro de especificação de cada interface de rede virtual (assumindo interfaces do tipo TUN/TAP):

```
eth<n>=tuntap,,<MAC address>,
<host IP address>
```

- No carregamento da máquina virtual UML, utilizar a opção `fakehd=true`, de forma a fazer com que todas as referências internas a dispositivos UBD sejam trocadas por dispositivos HD (IDE). O uso desta opção é indispensável para a eficácia das técnicas descritas na Seção 3.2.2;
- Aplicar o *patch* do SKAS no kernel hospedeiro para que suporte a execução do kernel UML em modo SKAS. Como já foi discutido na Seção 2, o uso do SKAS é importante para a dissimulação correta do espaço de endereçamento dos processos.

4 Conclusões e Trabalhos Futuros

Neste trabalho foram apresentadas diversas técnicas de detecção de máquinas virtuais UML, acompanhadas de contra-medidas para inibir a eficácia destas técnicas. Algumas destas contra-medidas envolvem implementação e/ou alteração do código do UML, enquanto outras se limitam a configurações especiais.

Conseguimos, com sucesso, configurar o HPPFS de forma a esconder os vestígios deixados pelo UML, eliminar as evidências deixadas pelo próprio HPPFS, esconder os vestígios deixados pelo uso do UBD e manipular o *log* circular do kernel em nosso favor, dando a crer que a máquina em questão se trata de uma máquina física. Também pudemos esconder diversos vestígios menores deixadas pelo UML através de configurações especiais.

Como resultado, obtivemos um sistema UML capaz de dissimular as principais características de máquinas virtuais UML, e portanto passível de ser utilizado como um *honeypot* virtual, diminuindo consideravelmente o risco de que seja descoberto por um intruso.

É importante ressaltar, contudo, que algumas das contra-medidas implementadas possuem limitações, e não resistiriam a uma análise mais minuciosa feita por um intruso habilidoso e determinado. Um exemplo de limitação é o tratamento rudimentar da chamada *ioctl* feito na dissimulação de dispositivos UBD. Um intruso poderia executar o comando `hdparm` com um parâmetro que utiliza um comando ATAPI não tratado e receberia uma mensagem de erro, o que poderia levantar suspeitas. Na realidade, qualquer comando que realize acesso direto a dispositivos de hardware provocaria um erro, já que o UML não implementa uma camada de emulação de hardware. Outra possível fonte de suspeitas é o fato de o carregamento de módulos estar desabilitado no *honeypot*, impedindo a instalação de *rootkits* residentes em memória de kernel. Acreditamos, contudo, que, em face do grande número de complicações que seriam criadas caso o suporte a módulos fosse habilitado, o risco apresentado por esta alternativa é aceitável.

Além das limitações expostas acima, como foi discutido na Seção 1, é possível que haja maneiras ainda não descobertas de se identificar *honeypots* UML que as técnicas descritas neste artigo não cobrem. O que realmente importa, nesse caso, é que as contra-medidas implementadas são capazes de neutralizar a geração atual de técnicas de reconhecimento de máquinas UML, ou seja, foi dado um passo a frente dos intrusos na eterna corrida que é a pesquisa em *honeypots* [11].

Dando continuidade ao trabalho, pretendemos adaptar as contra-medidas aqui descritas para a arquitetura do kernel 2.6 e expandir implementação feita do tratamento da chamada *ioctl* para dissimulação de dispositivos UBD que, no estágio atual, deixa de tratar alguns comandos.

No campo da coleta de dados em *honeypots*, também pretende-se explorar a vantagem que máquinas virtuais apresentam através da adaptação do mecanismo descrito em [8] para a arquitetura do UML. Este mecanismo atuará interceptando todas as chamadas de sistema executadas dentro da máquina virtual que modificam o sistema de arquivos e gravando-as no sistema de arquivos da máquina hospedeira, aos moldes do *TTY logger* descrito na Seção 2. Com esses dados em mãos, será possível realizar reconstituições das evidências deixadas por intrusos no sistema de arquivos do *honeypots* com uma alta granularidade temporal e informacional.

Referências

[1] Especificação do padrão ATAPI. http://www.ata-atapi.com.

[2] User-mode linux. http://user-mode-linux.sourceforge.net.

[3] Vmware. http://www.vmware.org.

[4] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, Sebastopol, CA, USA, 2. edition, Dec. 2002.

[5] Christopher Carella, Jeff Dike, Naomi Fox, and Mark Ryan. UML Extensions for Honeypots

in the ISTS Distributed Honeynet Project. In *Proceedings from the 5th IEEE SMC Information Assurance Workshop*, pages 130–137, West Point, NY, USA, June 2004. IEEE Computer Society Press.

[6] Joseph Corey. Advanced Honeypot Identification. *Phrack Inc. (edição falsa)*, 11(63), 2004.

[7] Joseph Corey. Local Honeypot Identification. *Phrack Inc. (edição falsa)*, 11(62), 2004.

[8] Martim d'Orey Posser de Andrade Carbone and Paulo Lício de Geus. A Mechanism for Automatic Digital Evidence Collection on High-Interaction Honeypots. In *Proceedings from the 5th IEEE SMC Information Assurance Workshop*, pages 1–8, West Point, NY, USA, June 2004.

[9] Maximillian Dornseif and Thorsten Holz Christian N. Klein. NoSEBrEaK–Attacking Honeynets. In *Proceedings from the 5th IEEE SMC Information Assurance Workshop*, pages 123–129, West Point, NY, USA, June 2004.

[10] Neal Krawetz. Anti-Honeypot Technology. *IEEE Security & Privacy Magazine*, 2(1):76–79, Jan-Feb. 2004.

[11] Bill McCarthy. The Honeynet Arms Race. *IEEE Security & Privacy Magazine*, 1(6):79–82, Nov.-Dec. 2003.

[12] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. John Wiley & Sons, New York, NW, USA, 6. edition, 2002.

[13] Lance Spitzner. *Honeypots: Tracking Hackers*. Addison-Wesley, Boston, MA, USA, 2002.

[14] Lance Spitzner. Honeypots: Definitions and Values. Disponível em World Wide Web (Agosto de 2004): <http://www.tracking-hackers/papers/honeypots.html>, May 2003.

[15] The Honeynet Project. Know Your Enemy: Learning with User-Mode Linux. Disponível em World Wide Web (Agosto de 2004): <http://www.honeynet.org/papers/honeynet/index.html>, Dec. 2002.

[16] The Honeynet Project. Know Your Enemy: Defining Virtual Honeynets. Disponível em World Wide Web (Agosto de 2004): <http://www.honeynet.org/papers/honeynet/index.html>, Jan. 2003.

[17] The Honeynet Project. Know Your Enemy: Learning with VMWare. Disponível em World Wide Web (Agosto de 2004): <http://www.honeynet.org/papers/honeynet/index.html>, Jan. 2003.

[18] The Honeynet Project. *Know Your Enemy: Revealing the Security Tools, Tactics, and Motives of the Blackhat Community*. Addison-Wesley, Indianapolis, IN, USA, 2. edition, 2004.

A Script *perl* de configuração do HPPFS

Abaixo segue o script perl utilizado pelo HPPFS. As duas linhas adicionadas são aquelas nas quais é feita a filtragem dos arquivos `/proc/devices` e `/proc/misc`, através do uso da função `remove_lines`.

```
# Copyright (C) 2002, 2003 Jeff Dike (jdike@karaya.com)
# Licensed under the GPL
#

use hppfs;
use hppfslib;
use strict;

my $dir;

@ARGV and $dir = $ARGV[0];

my $hppfs = hppfs->new($dir);

my $remove_filesystems = remove_lines("hppfs", "hostfs");

$hppfs->add("cmdline" => proc("cmdline"),
            "cpuinfo" => proc("cpuinfo"),
    "dma" => proc("dma"),
    "devices" => remove_lines("ubd", "vc"),
    "exitcode" => "remove",
    "filesystems" => $remove_filesystems,
    "interrupts" => proc("interrupts"),
    "iomem" => proc("iomem"),
    "ioports" => proc("ioports"),
    "mounts" => $remove_filesystems,
    "pid/mounts" => $remove_filesystems,
    "stat" => proc("stat"),
    "uptime" => proc("uptime"),
    "version" => proc("version"),
    "kcore" => $remove_filesystems,
    "misc" => remove_lines("net/tun"),
    dup_proc_dir("bus", $dir) );

$hppfs->handler();
```