Mauro Augusto Borchardt

Uma Arquitetura para a Autenticação Dinâmica de Arquivos

Curitiba
2002

Mauro Augusto Borchardt

Uma Arquitetura para a Autenticação Dinâmica de Arquivos

Dissertação apresentada ao Programa de Pós-Graduação em Informática Aplicada da Pontifícia Universidade Católica do Paraná como parte dos requisitos para obtenção do título de Mestre em Informática Aplicada.

Área de Concentração:
Metodologia e Técnicas de Computação

Orientador:
Carlos Alberto Maziero

Curitiba
2002

Borchardt, Mauro Augusto

Uma Arquitetura para a Autenticação Dinâmica de Arquivos. Curitiba, 2002.
81p.

Dissertação (Mestrado) — Pontifícia Universidade Católica do Paraná. Programa de Pós-Graduação em Informática Aplicada.

1.Segurança 2.Kernel 3.Criptografia 4.Sistemas de Arquivos I.Pontifícia Universidade Católica do Paraná. Centro de Ciências Exatas e de Tecnologia.

Sumário

Lista de Figuras

Lista de Tabelas

Resumo

Atualmente os sistemas de informação estão sujeitos a invasões. Boa parte dos programas em uso foram criados para uso em ambientes de rede sem agentes maliciosos, ou seja, sem mecanismos adequados para proteção em um ambiente de rede hostil. Nos últimos anos, foram descobertos vulnerabilidades em muitos deles. Juntamente com o descuido dos administradores na atualização de seus sistemas, isso tem possibilitado a ocorrência de acesso não autorizado.

O invasor pode alterar partes do sistema operacional ou de aplicações para esconder sua presença e garantir seu acesso futuro ao sistema. Muitas técnicas foram propostas para detectar essas atividades, mas geralmente as vulnerabilidades são detectadas apenas quando o sistema já se encontra corrompido.

Este projeto visa a aplicação conjunta de técnicas de assinatura digital de arquivos e interceptação de chamadas do núcleo do sistema operacional, objetivando a verificação da integridade dos arquivos do sistema em tempo real. São também abordados tópicos relativos à avaliação dos problemas de desempenho, manutenção, usabilidade e segurança relacionados à sua aplicação da técnica proposta.

Abstract

The current information systems are subject to intrusions. Several programs in use today have been created to be used in safe network environments, without adequate mechanisms to operate in a hostile environment. In the last few years, many of them were found to be highly vulnerable and together with the oversight of administrators in keeping their systems up to date, unauthorized access may occur.

The intruder can modify parts of the operating system or its applications to hide himself and to guarantee his future access to the system. Many techniques have been proposed to detect these activities, but vulnerabilities are generally detected only when the system is already corrupted.

This project seeks for the joint application of digital signature techniques and system call interposition with the goal of verifing the integrity of system files in real time. Aspects related to the evaluation of performance, maintenance, usage, and security problems related to its application are also presented.

Capítulo 1

Introdução

Durante as últimas três décadas a sociedade tem migrado boa parte de seus dados para os sistemas de informação. Embora alguns destes dados ainda existam fisicamente em forma de papel, no dia-a-dia são manipulados quase que exclusivamente em sua forma eletrônica. Isto torna os repositórios destes dados muito interessantes por conterem informações históricas e operacionais sobre pessoas, empresas, governos e segredos militares. Sobre estas informações decisões são tomadas, podendo influenciar enormemente o futuro das pessoas envolvidas.

Em algumas áreas críticas, tais como mercado financeiro, transporte aéreo e comunicações, a dependência dos sistemas de informação é quase que completa, sendo pouco provável que sobrevivam se seus sistemas forem afetados.

Conseqüentemente, tais sistemas tornaram-se muito atrativos a elementos espúrios desta mesma sociedade que almejam lucros explorando as falhas de segurança, conseguindo acesso a informações privilegiadas. Este lucro não é necessariamente financeiro, podendo ir da fama alcançada por tal feito até a falência de um concorrente, passando por extorsão de favores, difamação e implante de informações falsas.

Enquanto os sistemas mantiveram-se isolados uns dos outros, estes eram mais simples e fáceis de serem gerenciados, porém quando foram interligados, a quantidade de serviços oferecidos aumentou e a complexidade ainda mais, alguns formando redes imensas em

escala mundial, o que tornou sua gerência uma atividade complexa e penosa.

A maioria dos programas em uso atualmente foram inicialmente concebidos para uso em um ambiente não-hostil. Aspectos relacionados à segurança raramente são previstos nas fases iniciais do projeto, pois consomem tempo e não são o objetivo fim. Para a construção de um programa seguro esta característica deve fazer parte de todas as etapas de desenvolvimento, da análise à manutenção o que mesmo assim não garante a inexistência de falhas. Um exemplo recente ocorreu durante o lançamento do banco de dados ORACLE 9.0, cuja propaganda dizia "O Inquebrável", do qual fazia parte até mesmo um desafio para que fosse encontrado algum problema de segurança no software. Esta campanha não durou duas semanas, quando a descoberta de um simples estouro de buffer permitiu acesso administrativo ao servidor de banco de dados. [28, 59]

Hoje pode-se afirmar com grande chance de sucesso que algum dia, determinado sistema terá sua segurança comprometida. Esta convicção está levando a uma nova linha de pesquisa, chamada *Survivability*, que se refere à capacidade do sistema de continuar oferecendo serviços essenciais mesmo na ocorrência de uma intrusão e de sua recuperação posterior. Este pensamento não é totalmente absurdo se levarmos em conta que a maioria dos sistemas é composta por milhares ou milhões de linhas de código, desenvolvidas por muitas pessoas que muitas vezes nunca se conheceram pessoalmente, espalhadas ao redor do mundo. Como códigos antigos raramente são re-escritos e funcionalidades são adicionadas a cada nova versão, é de se esperar que existam muitas falhas a serem exploradas.

Tendo-se esta realidade como base, partiu-se para a observação nos registros de incidentes de órgãos de segurança como o *CERT/CC* [14] e o *SANS* [36], onde constatou-se que na maioria das invasões alguns arquivos do sistema eram modificados, sejam arquivos de configuração ou binários (bibliotecas ou executáveis), ou novos arquivos eram introduzidos. Estas modificações iniciais tinham como objetivo garantir uma entrada posterior mais tranqüila e "invisível" (um "*backdoor*"), de onde o invasor pudesse realizar o que desejasse.

A partir dessa constatação, foram estudadas várias formas de proteção aos arquivos de sistema, sem no entanto fazer uso dos mecanismos de autenticação e autorização do sistema operacional, pois estes podem ser facilmente burlados pelos invasores. A melhor solução encontrada foi a mesma utilizada para garantir a integridade de documentos eletrônicos, ou seja, assinar os arquivos que precisam ser garantidos. As técnicas de assinatura digital são normalmente empregadas para certificar arquivos de texto, mas podem ser igualmente empregadas para a certificação de arquivos binários como executáveis e bibliotecas.

Esta dissertação está assim estruturada: no capítulo 2 será feito um apanhado histórico dos principais problemas levando ao comprometimento da segurança de um sistema. A seguir, no capítulo 3, será feita uma revisão das técnicas de assinatura digital de documentos. No capítulo 4 serão discutidas as técnicas conhecidas de modificação do núcleo para estender suas funcionalidades visando a implantação da arquitetura proposta. No capítulo 5 será detalha a arquitetura desenvolvida para certificação de arquivos implantada no núcleo de um sistema operacional. No capítulo 6 serão apresentados os resultados obtidos com a implementação da arquitetura em um núcleo real. Finalmente, no capítulo 7 serão apresentadas as considerações finais referentes ao trabalho, projetos correlatos à proposta e suas perspectivas futuras.

Capítulo 2

Vulnerabilidades e Ataques

No documento *Steps for Recovering from a UNIX or NT System Compromise* [10] do CERT/CC (*Computer Emergency Response Team/Coordination Center*), pode-se observar pelos detalhes apresentados que todas as modificações recaem sobre o sistema de arquivos. Esta é a principal fonte de informação para a Forense Computacional [26], onde o sistema de arquivos é analisado em busca de provas que evidenciem uma infração. Os vestígios normalmente procurados variam desde o tipo e conteúdo de um arquivo, como a de imagens de pornografia infantil, até o conteúdo do espaço livre [1] de um cluster de arquivo que pode indicar o antigo conteúdo desse cluster.

Um dos principais conjuntos de arquivos de um sistema é o conjunto de arquivos de log, pois neles são registrados todos os fatos mais importantes que ocorrem na vida operacional do sistema. Neles estão registradas as atividades dos usuários, dos processos, as conexões de rede, erros e alertas, ou seja, basicamente todos os eventos importantes do sistema, com exceção do conteúdo dos arquivos utilizados. Por este motivo, o sistema de log é o primeiro a ser modificado em uma invasão, objetivando esconder os rastros do invasor. Isto pode ocorrer não necessariamente através da modificação do programa responsável pela captura e armazenamento das mensagens de log (o `syslogd` no *UNIX*),

[1] O sistema de arquivos sempre aloca blocos de disco de tamanho fixo, como por exemplo 8192 bytes. Um arquivo de 1000 bytes deixaria 7192 bytes alocados mas sem uso, que podem conter informações de sua última utilização.

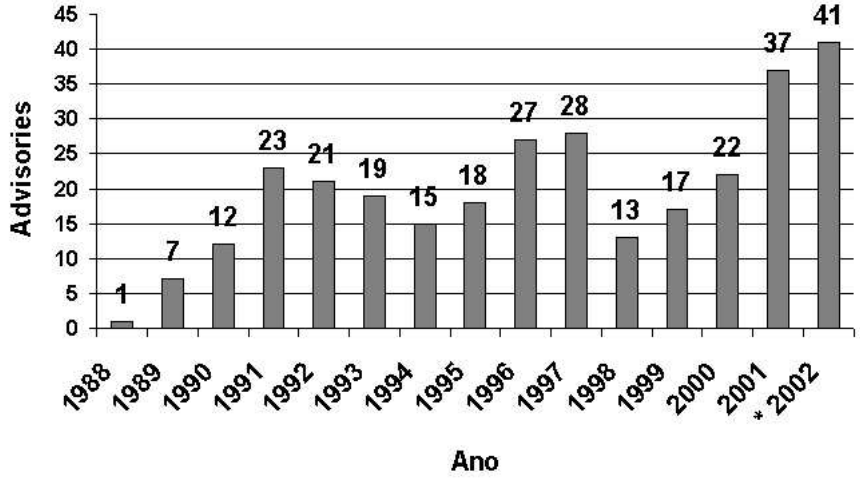

Figura 2.1: Estatística do CERT/CC: *Advisories* emitidos por ano.

mas modificando os programas que geram estas mensagens, como o programa `login`, por exemplo, que deixaria de registrar as mensagens de *login* para um determinado usuário (o invasor).

Excetuando-se a situação pouco provável do invasor conhecer a senha do administrador, ele irá se utilizar de alguma vulnerabilidade de um programa ou serviço do sistema para conseguir acesso ao mesmo, preferencialmente aquelas que possibilitem acesso administrativo. Segundo as estatísticas do CERT [16], figura 2.1, pode-se observar que o número de *advisories* emitidos tem crescido nos últimos quatro anos e as projeções para 2002 mantém esta tendência. O número de novas vulnerabilidades recebidas sobre programas também tem crescido neste mesmo período, conforme indica a figura 2.2. O CERT utiliza um sistema de pontuação conforme a gravidade, abrangência e outras considerações sobre a vulnerabilidade, que varia de 0 a 180 pontos. Normalmente um *advise* é emitido quando esta pontuação é superior aos 40 pontos. Não é possível determinar se o crescimento observado é decorrente de mais problemas surgindo devido ao crescimento do número de sistemas ou se é decorrente de uma maior atenção dada a segurança.

Independentemente do motivo, a possibilidade de se encontrar um sistema com algum problema de segurança é muito grande. Em pesquisa realizada pelo site *Zone-H.org*

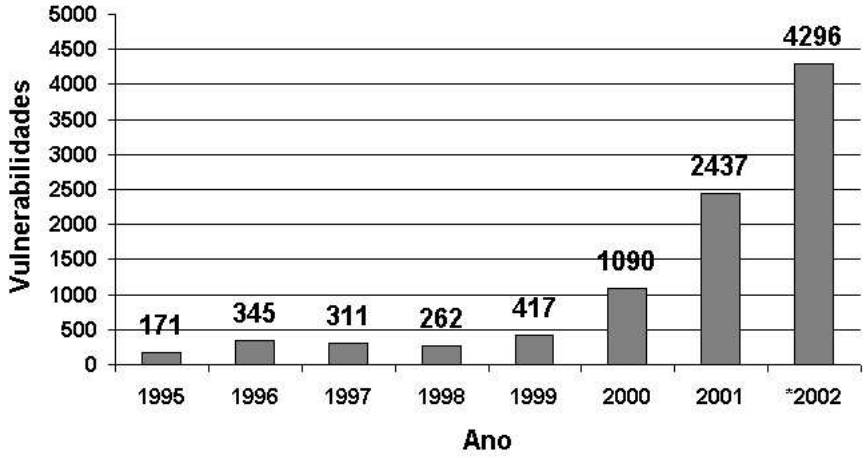

Figura 2.2: Estatística do CERT/CC: Vulnerabilidades detectadas por ano.

[68], especializado na publicação de relatos sobre invasões na Internet, no primeiro semestre de 2002 constatou-se que entre os problemas que levaram os sistemas a serem invadidos, 29,3% foi devido a uma vulnerabilidade conhecida mas cuja correção não foi aplicada pelo administrador, 20,4% foi por apresentar uma vulnerabilidade ainda sem correção, 15% foi por ataque de força bruta (sem o invasor conhecer detalhes a respeito do alvo), 13,3% por má configuração, 5,2% por engenharia social e 0,1% por outros motivos. Excetuando-se os ataques a vulnerabilidades sem correção, todas as demais poderiam ter sido evitadas.

Ainda nesta mesma pesquisa, a principal alegação que o invasor apresentou como motivação foi “fiz só para me divertir” com 46,6%, “sem razão específica” com 19,3%, “desafio pessoal” e “auto-promoção” somando 19,2% e em último “razões políticas”, “patriotismo” e “revanchismo” somando juntos 14%. Isto demonstra a sorte que tais sistemas tiveram, pois apenas estes últimos 14% tinham motivos sérios para efetuar a invasão, enquanto a grande maioria dos invasores o fez somente por brincadeira.

2.1 Levantamento de Informações

A primeira etapa de um ataque bem sucedido é conhecer a vítima, recolhendo dados sobre os recursos de máquina, topologias de rede, sistemas operacionais e suas versões, serviços disponíveis e suas versões, identificação dos usuários e seus perfis. Boa parte deste trabalho pode ser feito através de ferramentas específicas para tal, como:

- packet sniffers;
- scanners de rede;
- detecção de sistema operacional;
- scanners de sistema.

Normalmente a placa de rede recebe todos os pacotes que chegam até ela, descartando aqueles que não são endereçados à maquina local. Os "packet sniffer" alteram a configuração da placa de rede para capturar todos os pacotes chegando à placa. Serviços como *telnet*, *pop3*, *imap*, *ftp*, *rlogin*, *http* e muitos outros não usam criptografia, permitindo assim que seus dados sejam capturados pelo sniffer, tais como senhas, logins, números de cartão de crédito, etc.

Scanners de rede, como o *NESSUS* [23] e *SAINT* [19], permitem efetuar uma verificação completa da máquina alvo, remotamente. Estes softwares permitem identificar portas abertas, identificar versões de serviços e testar vários tipos de vulnerabilidades e más configurações dos mesmos. Suas bases de dados são constantemente atualizadas, refletindo as novas vulnerabilidades encontradas nos serviços.

Um técnica relativamente nova de levantamento de informações é a detecção do sistema operacional da máquina remota, bem como de sua versão, baseando-se somente no comportamento da pilha TCP/IP. Muito embora as RFCs descrevam como a pilha TCP/IP deva se comportar, cada desenvolvedor implementa detalhes não previstos na RFC à sua

própria maneira, fazendo-a reagir de maneira distinta em determinadas situações. Baseado no levantamento de diversas respostas possíveis da pilha, é possível ir refinando a busca até a determinação acurada da versão do sistema operacional na máquina remota. Em [24] é apresentada detalhadamente como é realizada a detecção no programa *NMAP*, um dos mais sofisticados scanners de rede, que utiliza 13 características diferentes na detecção.

Mesmo a máquina estando com os serviços de rede corretamente configurados e com versões não vulneráveis, pode ser que os serviços locais o estejam. Programas como *COPS* [29] e o *TIGER* [31] fazem a verificação de contas sem senha ou com senhas óbvias, permissões incorretas em arquivos e diretórios, diversos tipos de má configuração, SUID[2] de root em arquivos indevidos, entre várias outras situações de risco.

É possível obter muitas informações úteis sobre um sistema computacional sem fazer uso das ferramentas aqui descritas, mas simplesmente interagindo com as pessoas que usam o sistema. Esse método é conhecido como Engenharia Social. Neste método, as informações dos usuários são levantados de maneiras diversas, por telefone passando-se por alguém do suporte do sistema, ou através da ficha cadastral, ou pela internet. De posse destes dados, tenta-se descobrir um login e senha válidos para o sistema, pois observa-se que os usuários normalmente costumam utilizar-se de dados facilmente lembrados como datas de aniversário, número da identidade, nome da namorada ou esposa. Uma vez obtido acesso ao sistema através de um usuário válido, outras técnicas podem ser usadas para obter maiores privilégios.

2.2 Vulnerabilidades

De posse de informações suficientes, o invasor vai escolher um programa que explore a vulnerabilidade de algum dos serviços que escolheu. Estes programas, chamados de *ex-*

[2]No UNIX, um flag associado ao arquivo que indica para o sistema operacional, que processos criados a partir desse arquivo assumirão a identidade do dono do arquivo. Esse mecanismo permite que um simples usuário tenha direitos de administrador.

ploits, são disponibilizados aos milhares em sites da internet, bastando ao invasor escolher o que mais lhe agrada, não sendo necessário conhecimento técnico de como o programa em questão explora a vulnerabilidade. Isto levou ao surgimento de uma categoria de invasores conhecida como “script-kiddie” que, com conhecimentos mínimos e bons pacotes, invadiram muitos sistemas.

As técnicas que os programas/exploits utilizam variam muito, sendo especificas para cada tipo de vulnerabilidade. As técnicas mais conhecidas são:

Race Condition (condição de disputa): visa enganar o processo, levando-o a criar, ler ou alterar um arquivo que não pretendia, através de um *link* simbólico. O processo ao criar um arquivo temporário, não testa se o mesmo já existe e também não percebe que o mesmo é um *link* para outro arquivo. Existem outras variantes desta técnica.

Buffer Overflow (estouro do *buffer*): visa a introdução de um código executável na pilha de chamada de uma função. Programas que apresentam esta vulnerabilidade possuem um *buffer* para dados recebidos com tamanho pré-definido. Porém ao copiar os dados recebidos para o *buffer* não é verificado se o dado recebido cabe no mesmo. Quando o dado excede o *buffer*, normalmente é danificada parte da pilha da função, normalmente gerando uma exceção no programa conhecida pelo erro de “segmentation fault”. O que o *exploit* faz é introduzir um pequeno código executável junto com o dado passado para a função, na parte que excede o tamanho do buffer, sobre-escrevendo a pilha da função e introduzindo assim um novo endereço de retorno que apontará para este pequeno código extra, que fará o que o invasor programou. Boa parte dos *advisories* emitidos pelo CERT se enquadram nesta categoria.

Contas abertas Na instalação de alguns sistemas são criadas várias contas de sistema para determinados serviços. Sendo o processo de instalação automatizado, algumas destas contas ficam cadastradas com senhas padrão, ou mesmo sem senha. Os *CERT*®

Incident Note IN-2001-13 e *IN-2002-04* são relatos de incidentes relacionados à ação de dois *worms* que usam a vulnerabilidade *VU#635463*. Essa vulnerabilidade consiste em deixar a conta de administrador do servidor SQL da Microsoft com definição de senha como opcional, o que permite a qualquer usuário acessar o sistema como administrador, pois o login é "sa" sem senha.

IP Spoofing Baseia-se em fazer uma máquina se passar por outra confiável. Existem duas formas deste ataque acontecer: uma é a introdução de uma máquina em uma rede assumindo o endereço IP da máquina confiável, e assim respondendo pelas solicitações encaminhadas à mesma. Outra possibilidade é o ataque ao servidor de nomes, conhecido como servidor de DNS, inserindo o endereço da máquina intrusa nas tabelas do serviço de nomes. Em [58] são apresentados detalhes sobre este tipo de ataque sobre os servidores de DNS mais empregados atualmente. Uma vez fazendo-se passar pela máquina confiável, a máquina intrusa simula os serviços que a máquina de destino teria, conseguindo desta forma obter nomes de login e senhas dos usuários da maquina destino.

Connection Hijacking (roubo de conexão): permite que a conexão TCP/IP estabelecida entre um cliente e um servidor seja desviada (roubada) para um terceiro, podendo desta forma injetar pacotes TCP "válidos" com comandos na conexão, contornando assim até mesmo os mecanismos de autenticação tais como *tickets Kerberos* ou senhas. Esta ocorrência normalmente está associada aos serviços de seção não cifrados, como por exemplo, uma seção *telnet*.

Cada pacote transmitido entre os dois extremos de uma conexão TCP possui um número de seqüência de 32 bits, sendo estes números escolhidos progressivamente incrementados com um valor aleatório positivo durante o estabelecimento da conexão por ambas as partes. Entretanto, em algumas implementações do protocolo TCP/IP estes números não são suficientemente aleatórios [15]; a partir do conhecimento de um número de seqüência pode-se predizer o da próxima conexão ou do próximo pacote da seção cor-

rente. Desta forma, é possível desde a introdução de pacotes no meio de uma seção ou até mesmo roubar a seção, desfazendo a conexão do cliente original e assumindo o controle daquele ponto em diante. Em [5] são apresentados mais detalhes sobre esta técnica.

Format String Attack A função de biblioteca *printf* da linguagem C é utilizada na grande maioria dos programas para formatar dados para apresentação seja para tela ou arquivo. Explorando o comportamento normal desta função enquanto processa os dados de formatação e os parametros passados, permite que sejam inseridos dados arbitrários (por exemplo código executável) e que seja modificada a pilha de execução do programa. Desta forma, é possível enviar um pequeno programa pelos dados de entrada e que seja desviada a execução para ele, permitindo assim a entrada de um invasor. Em [46] é mostrado como esta vulnerabilidade pode ser explorada facilmente.

2.3 Rootkits

Uma vez obtido acesso administrativo ao sistema, o invasor precisa agora de ferramentas que o auxiliem a manter este acesso ou voltar a tê-lo mais facilmente e também para apagar todos os vestígios de sua ação e presença. Os *rootkits*, como o próprio nome sugere, são coleções de programas especialmente construídos para tal. Estes pacotes incluem programas para adulterar os atributos de arquivos, falsificando os tempos de modificação, acesso, tamanho e para criar dados nos arquivos modificados que produzam o mesmo valor de HASH do arquivo original. Também incluem ferramentas de remoção de entradas em vários arquivos de log, além de *sniffers* para captura de logins e senhas que trafegam sem criptografia.

Entretanto, são os programas modificados e aqueles com *backdoor* os mais comuns nessas coleções. Todos os programas que possam revelar a presença do invasor na máquina são substituídos, que em sistemas UNIX incluem os comandos: *ps*, *ls*, *find*, *su*, *ifconfig*, *netstat*, *syslogd*, *inetd* entre outros, além daqueles que prestam serviços de rede,

como *ftpd* e *sshd*. Também podem ser modificados arquivos de configuração para execução automática de scripts e serviços. Algumas vezes são criados diretórios com nomes especiais, tais como: “.. ”, “...”, “...^H^H^H” em diretórios pouco usuais como /dev ou /var/spool/mqueue, que dificilmente são detectados. Com isto, suas conexões de rede, diretórios de trabalho e processos não podem ser vistos normalmente no sistema e também não geram mensagens de log.

Os mais recentes *rootkits* para *Linux* são desenvolvidos como módulos de núcleo (LKM - Linux Kernel Modules), podendo ser dinamicamente carregados no núcleo em tempo de execução. Ao invés de alterar os programas que mostram a presença do invasor, eles alteram as chamadas de sistema que geram as informações que esses programas utilizam, obtendo-se assim o mesmo resultado. Normalmente os *backdoors* em modo núcleo também fazem parte dessas distribuições. Este tipo de *rootkit* é praticamente indetectável após carregado, principalmente a partir da máquina afetada. Em [45] é apresentada uma ferramenta especificamente voltada para sua detecção.

2.4 Conclusão

As estatísticas têm demonstrado que o número de invasões de sistemas tem crescido nos últimos anos, basicamente devido à também crescente quantidade de vulnerabilidades descobertas nos softwares de rede. Através da exploração dessas vulnerabilidades, através de diversas técnicas, um invasor consegue obter acesso administrativo ao sistema. Fazendo-se uso de ferramentas especiais, *rootkits*, modificam-se tais sistemas para conseguir-se acesso direto e indetectável, não sendo mais necessária a exploração das vulnerabilidades. Ao impedir o uso de arquivos adulterados ou mesmo a adulteração dos mesmos, é possível limitar a ação de um invasor e impedir que ele se dissimule no sistema.

Capítulo 3

Integridade de Dados

A integridade de dados é definida [44] como a propriedade de um dado não ter sido alterado de forma não autorizada desde o momento de sua criação, transmissão ou armazenamento a partir de sua fonte autorizada. Qualquer alteração, por menor que seja, torna o dado não-íntegro. A verificação da integridade de dados pode ser feita através das técnicas criptográficas apresentadas neste capítulo.

3.1 Criptografia

A criptografia é definida basicamente como o estudo de técnicas matemáticas que implementam as quatro propriedades listadas abaixo, tanto em seus aspectos teóricos como práticos.

- **Confidencialidade:** impede que pessoas não autorizadas tenham acesso ao conteúdo da informação;
- **Integridade:** garante que o conteúdo da informação não foi alterado;
- **Autenticidade:** garante a identidade de quem produziu a informação;
- **Não-repudiação:** impede que alguém negue o envio ou recepção de uma determinada informação;

As técnicas criptográficas normalmente fazem uso de mecanismos para cifrar (codificar) e decifrar (decodificar) a informação. Para as operações de ciframento de um objeto ou mensagem exige-se dois elementos: o algoritmo e a chave de ciframento. Normalmente os algoritmos de ciframento são conhecidos, residindo o segredo na escolha e manutenção das chaves criptográficas. Existem duas classes de sistemas criptográficos: os baseados em chave simétrica e os baseados em chave assimétrica. Podem também existir sistemas criptográficos baseados somente em algoritmo, que neste caso deve ser secreto, ou seja, conhecido somente entre as partes.

3.1.1 Criptografia Simétrica

Esta classe de algoritmos é baseada no uso de uma mesma chave tanto para cifrar como para decifrar a mensagem trocada entre as partes. Neste caso a chave deve ser conhecida somente pelas partes, entretanto o algoritmo criptográfico pode ser público. A figura 3.1 ilustra o modelo de funcionamento deste sistema.

Quando duas entidades desejam trocar mensagens cifradas, elas devem primeiramente encontrar um meio seguro para compartilharem uma mesma chave criptográfica. A partir deste ponto, Alice (origem) pode enviar para Bob (destino) mensagens que são cifradas com a chave K, transformando-as em novo conjunto de bits que podem ser transmitidos por qualquer meio, que Bob pode transformá-los novamente nas mensagens originais através do deciframento com a mesma chave K.

Os principais algoritmos desta classe são:

- *DES (Data Encryption Standard) [48]:* É o algoritmo simétrico mais difundido no mundo. Criado pela IBM em 1977, com um tamanho de chave de 56 bits, relativamente pequeno para os padrões atuais, foi quebrado por "força bruta" em 1997;

- *3DES [17]:* Uma variação do DES, utiliza 3 ciframentos em seqüência, empregando chaves com tamanho de 112 ou 168 bits, sendo recomendado no lugar do DES desde 1993;

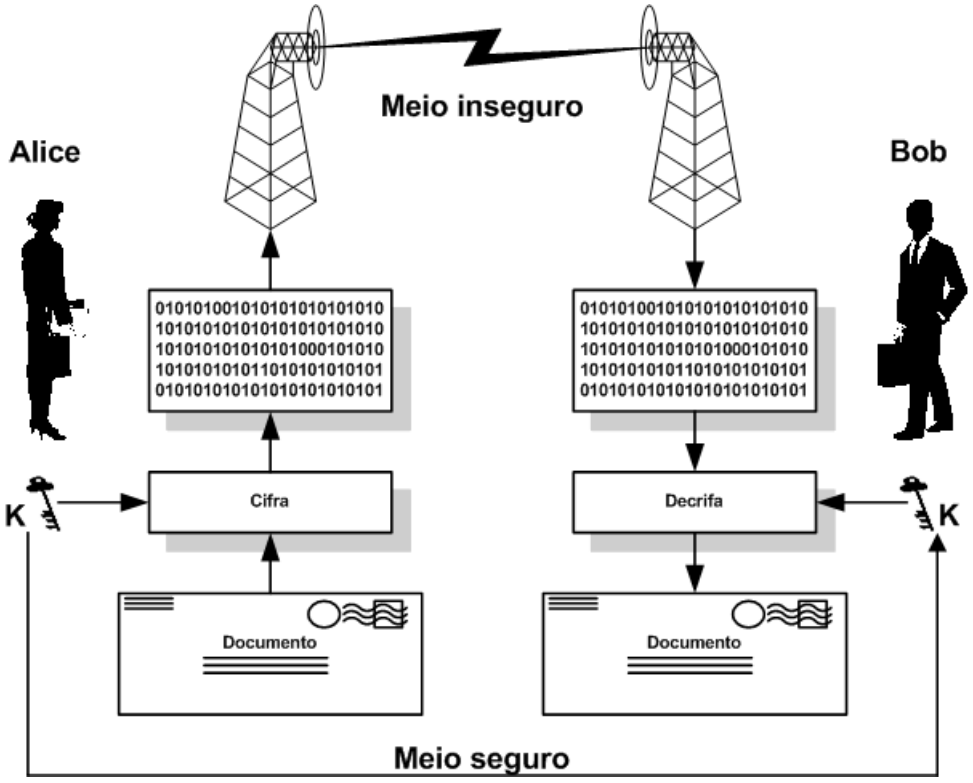

Figura 3.1: Modelo de funcionamento de criptografia baseada em chave simétrica.

- *IDEA (International Data Encryption Algorithm) [22]:* Criado em 1991, segue as mesmas idéias do DES, mas tem execução mais rápida que o mesmo.

- *AES (Advanced Encryption Standard)[50]:* É o padrão atual para ciframento recomendado pelo NIST (National Institute of Standards and Technology). Trabalha com chaves de 128, 192 e 256 bits, que adotou o cifrador *Rijndael*[21] após a avaliação de vários outros.

- *RC6 [54]:* A última versão de uma série de cifradores (RC2,RC3,RC4,RC5) desenvolvidos por Rivest[1]. Concorreu à adoção pelo padrão AES.

A criptografia simétrica apresenta algumas vantagens e desvantagens quando comparados a outros sistemas, que são:

- **Vantagens:** São muito rápidos, permitindo a cifragem rápida de grandes volumes

[1]Um do desenvolvedores do RSA.

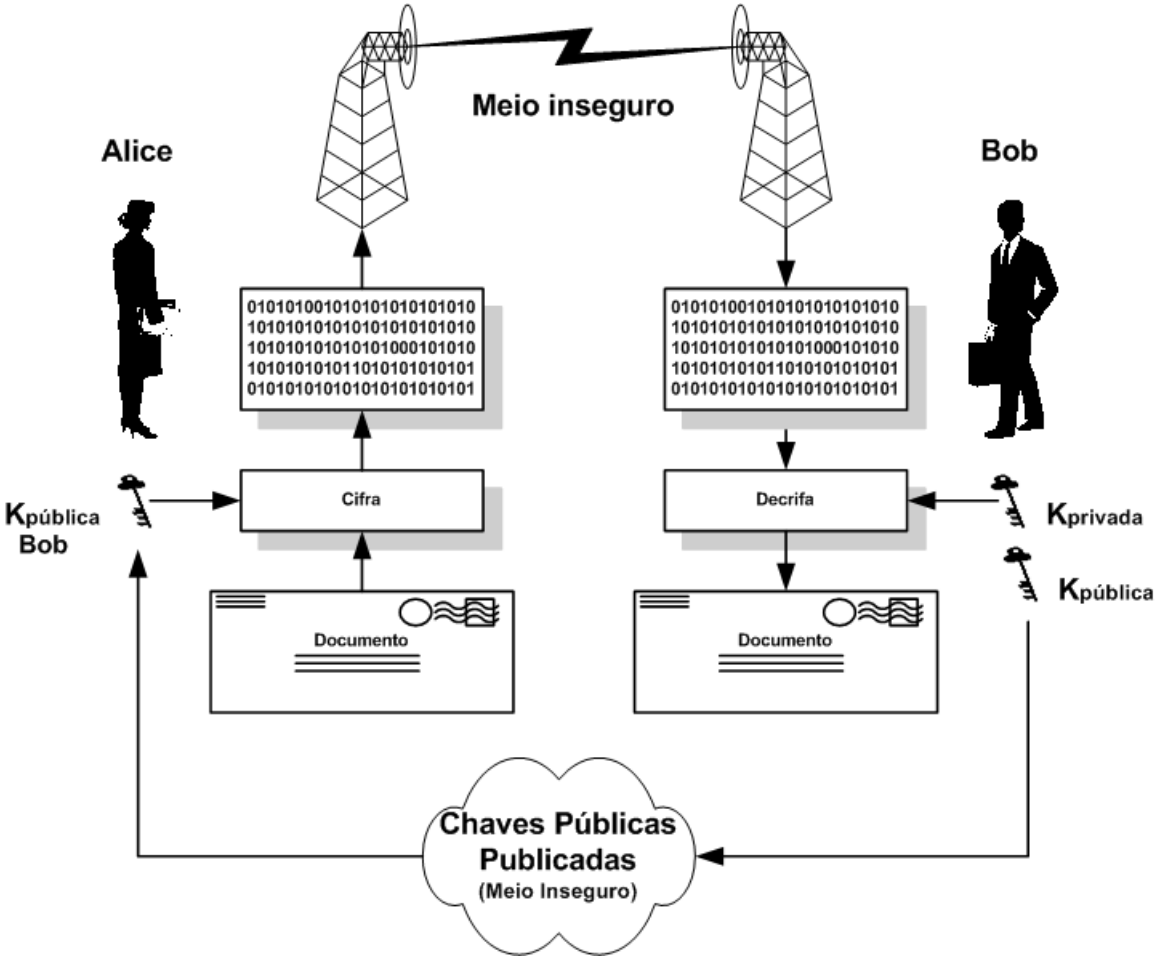

Figura 3.2: Modelo de funcionamento de criptografia baseada em chave assimétrica.

de dados. As chaves são relativamente pequenas bem como relativamente simples, podendo gerar cifradores muito robustos.

- **Desvantagens:** A comunicação é ponto-a-ponto e cada par de pontos comunicantes precisa manter uma chave secreta própria, desconhecida dos demais pontos. A gerência de chaves pode ser complicada em redes onde existam muitos pontos. Recomenda-se a troca freqüente das chaves. Não se garante a identidade da origem;

3.1.2 Criptografia Assimétrica

Esta classe de algoritmos é baseada no uso de um par de chaves, uma privada ou secreta e outra pública, como o próprio nome diz, que qualquer um pode conhecer.

A figura 3.2 ilustra o modelo de funcionamento deste sistema: quando duas entidades desejam trocar mensagens cifradas, elas devem primeiramente publicar suas chaves públicas em algum lugar acessível a ambos. A partir deste ponto, Alice (origem) busca a chave pública de Bob (destino); de posse dela então cifra as mensagens, transformando-as em novo conjunto de bits que podem ser transmitidos por qualquer meio. Ao receber os dados cifrados, Bob pode então transformá-los novamente nas mensagens originais através do deciframento com sua chave privada.

Os sistemas criptográficos de chave assimétrica se baseiam na dificuldade em se calcular a operação inversa de determinadas operações matemáticas. Atualmente existem três categorias de problemas matemáticos a partir dos quais são construídos algoritmos de chave assimétrica. Estas categorias são:

Fatoração Inteira (IFP). Dado um número n resultado da multiplicação de dois números primos p e q, a dificuldade consiste em encontrar p e q tendo-se somente n, sendo todos números inteiros positivos. Para números pequenos, um ataque de força bruta pode facilmente encontrar a solução, mas para valores maiores de n (da ordem de 1024 bits) isto já é não mais válido. Um dos esquemas criptográficos baseados neste problema é o *RSA (Rivest, Shamir e Adleman)* [55].

Logaritmo Discreto (DLP). Dada a equação $y = g^x \bmod p$, onde g é um número inteiro positivo e p um número primo, ambos conhecidos, a dificuldade é dado o valor de y calcular o valor de x. Da mesma forma que o *IFP*, para valores grandes de p, x, y e g, da ordem de centenas de bits, este cálculo torna-se muito difícil. Os esquemas criptográficos *ElGamal*[44] e *Diffie-Hellman*[25] são baseados neste problema.

Logaritmo Discreto sobre Curvas Elípticas (ECDLP). Dada a equação $Q = l.P \bmod n$, onde P é um ponto sobre a curva elíptica E, n a ordem do ponto P, e l um inteiro no intervalo $0 \leq l \leq n-1$, a dificuldade está em se encontrar l sabendo-se P e Q. Basica-

mente os mesmos algoritmos desenvolvidos para DLP, podem ser aplicados sobre curvas elípticas.

A criptografia assimétrica apresenta algumas vantagens e desvantagens quando comparada a outros sistemas, que são:

- **Vantagens:** Somente a chave privada precisa ser secreta, com isto a gerência de chaves é mais simples. Dependendo do modelo usado o par de chaves pode permanecer o mesmo por longos períodos. Os mecanismos de assinatura digital são relativamente eficientes. O número de chaves é pequeno ($O(2n)$) se comparado ao modelo do sistema simétrico ($O(n^2)$).

- **Desvantagens:** São de execução lenta, o que os torna ineficientes para grandes volumes de dados. O tamanho das chaves é grande para oferecer a mesma dificuldade computacional quando comparado ao sistema simétrico.

De todos os sistemas criptográficos desenvolvidos, o *ECDLP* é o que está sendo mais pesquisado no momento, devido a algumas características que o diferenciam dos anteriores. A maior parte de suas operações é de adição e não de multiplicação, como nos anteriores, permitindo assim que possa ser implementado em hardwares de baixo poder de processamento, como *PDAs* e *telefones celulares*. Vários órgãos têm padronizadas as curvas, algoritmos e tamanho de chaves, objetivando a interoperabilidade e a garantia de níveis de segurança. Estes padrões são o *FIPS 186-2* [49], *ANSI X9.62*, *IEEE P1363* [35] e *ISO/IEC 14888-3*, sendo o padrão *FIPS 186-2* o menos compatível seguindo até o padrão *ISO/IEC 14888-2*, compatível com todos os anteriores e definindo o padrão *ECDSA (Elliptic Curve Digital Signature Algorithm)*.

Na tabela 3.1, extraída de [37], pode-se ver as diferenças de tamanho, ou número de bits, das chaves utilizadas conforme o sistema criptográfico, para um mesmo nível de proteção (ou seja, apresentam a mesma complexidade computacional para a quebra da chave). Normalmente, quanto menos bits precisam ser manipulados, mais rápida é

Simétrico	RSA n	DSA p	DSA q	ECDSA n
56	512	512	112	112
80	1024	1024	160	161
112	2048	2048	224	224
128	3072	3072	256	256
192	7680	7680	384	384
256	15360	15360	512	512

Tabela 3.1: Equivalência aproximada do número de bits das chaves [37].

	ECDSA $GF(2^n)$	ECDSA $GF(p)$	RSA	DSA
geração de chave	13.0	5.5	1000	22.7
assinatura	13.3	6.3	43.3	23.6
verificação	68	26	0.65	28.3

Tabela 3.2: Comparação do ECDSA com outros algoritmos [65]. ECDSA (191 bits), RSA (1024 bits) e DSA (1024 bits). Tempos em ms.

a execução do algoritmo, mas isto não é uma constante. Normalmente o desempenho não é similar entre operações inversas: por exemplo se assinar um documento é rápido, o processo de verificação é normalmente lento. Isto pode ser observado na tabela 3.2, extraída de [65], onde o tempo de cálculo do valor de hash foi desconsiderado.

3.2 Resumos Digitais

Resumo digital é o resultado da aplicação de determinadas funções matemáticas facilmente calculáveis que mapeiam uma cadeia de bits de qualquer tamanho em um número determinado fixo de bits, que é o tamanho do resumo. Estas podem ser classificadas em duas categorias: as que não usam chave e as que usam chave [3].

As funções de resumo digital que não fazem uso de chaves são normalmente utilizadas para a detecção de modificações em dados. Estas devem apresentar a propriedade de "mão única", ou seja, dada uma entrada deve ser facilmente computado o valor de resumo, mas o cálculo no sentido oposto deve ser muito difícil, ou mesmo impossível. Para uso em criptografia elas também devem apresentar a propriedade de que uma pequena perturbação

na entrada gera uma grande mudança na saída. Também devem apresentar resistência à colisão, ou seja, dadas duas entradas diferentes, suas saídas devem ser distintas. Algumas funções desta categoria são as *MD4*, *MD5*, *SHA-1*, *RIPEMD-160* e *HAVAL*.

As funções *MD4* e *MD5* [53] apresentam a saída com tamanho de 128 bits, sendo as mais rápidas em termos de execução. Já as funções *SHA-1* [47] e *RIPEMD-160* [27] tem saída com 160 bits e tempos de execução maiores. A função *HAVAL* [67], por ser configurável apresenta saídas de 128 a 256 bits (em passos de 32bits), sendo os tempos de execução variáveis conforme o número de passos de cálculo escolhidos.

As funções que fazem uso de chaves normalmente são utilizadas para assinar documentos à medida em que são processados (vide seção 3.3). Como entrada, além do dado estas funções também recebem uma chave criptográfica. Além das características já apresentadas, estas funções devem também apresentar resistência à predição: a possibilidade de se poder calcular qual será o próximo valor do resumo após a entrada de um novo dado, tendo-se o conhecimento do valor atual do resumo e do próximo dado a ser processado.

Um estudo mais detalhado sobre as funções de resumo digital (*hash functions*) pode ser obtido em [44].

3.3 Assinatura Digital

Assinar digitalmente um documento consiste em usar um algoritmo criptográfico fazendo uso de uma chave secreta, de tal forma que se possa verificar se o documento recebido pelo destino não foi adulterado após sua criação, garantindo assim a autenticidade da origem e a integridade do documento.

O algoritmo básico de assinatura digital em todos os padrões de criptografia segue o apresentado na figura 3.3, mas cada padrão tem sua própria forma de implementá-lo. O formalismo de cada uma delas pode ser obtido junto a respectiva norma ou em [44]. Supondo-se que Alice queira enviar uma mensagem assinada digitalmente para Bob, ela primeiramente publicaria sua chave pública em algum lugar acessível a Bob. Aplicando-

se uma função de resumo sobre a mensagem obteria o resumo digital desta mensagem, que Alice cifraria com sua chave privada. Este resumo cifrado juntamente com a mensagem original formariam o corpo da mensagem assinada, que seria enviada para Bob por algum meio. Bob retiraria do corpo da mensagem assinada a mensagem original e sobre ela aplicaria a mesma função de resumo que Alice utilizou, obtendo assim o valor atual do resumo digital da mensagem. Utilizando-se da chave pública de Alice, Bob decifraria o resumo assinado que está no corpo da mensagem assinada e obteria o valor do resumo da mensagem enviada por Alice. Comparando-se os valores de ambos os resumos, o atual e o da Alice, saber-se-á se a mensagem foi adulterada ou não; se forem iguais a mensagem é a mesma, ou seja, está íntegra e pode-se assegurar que a mensagem foi gerada por Alice.

Funções *hash* que fazem uso de chaves simétricas também podem garantir a integridade do dado, mas não podem garantir a autenticidade, pois a mesma chave que é usada para assinar o documento é usada em sua verificação. Como a chave é compartilhada por mais de um indivíduo, não há como garantir que a mensagem não foi adulterada e assinada novamente no destino. Entretanto, se isto não representar um problema, elas apresentam a característica de serem mais simples, computacionalmente eficientes e muito rápidas.

No modelo baseado no par de chaves pública/privada, a velocidade das operações de ciframento e deciframento é bem menor, mas o fato de somente a origem poder assinar os dados, pois ela é que detém a chave privada, permite a qualquer um que tenha acesso à chave pública verificar a autenticidade e a integridade da mensagem. Uma grande vantagem destes sistemas é que os algoritmos são públicos e a chave de verificação é pública, a informação a assinar é pública e somente a chave privada é secreta, que a origem deve guardar (esconder).

3.4 Conclusão

Neste capítulo foram apresentadas algumas técnicas de criptografia utilizadas para garantir a integridade e autenticidade de dados nos quais são utilizadas chaves assimétricas,

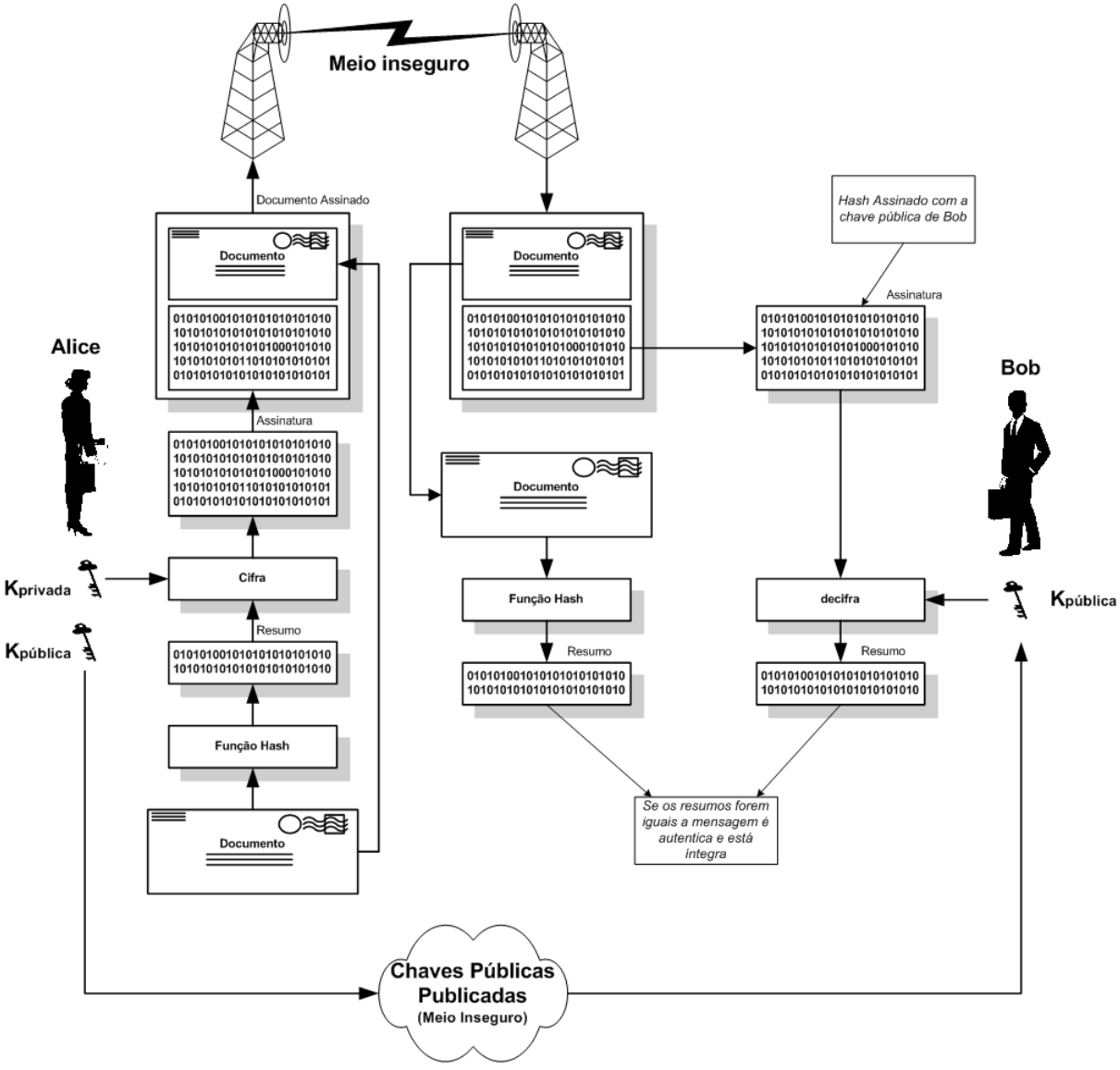

Figura 3.3: Modelo de funcionamento da assinatura digital.

resumo digital e assinatura digital. Para os objetivos deste trabalho, será utilizada a técnica de assinatura digital de arquivos que permitirá não somente garantir a integridade do arquivo, como também a autenticidade dos mesmos. Isto permitirá assegurar uma maior confiabilidade aos arquivos instalados no sistema e assinados digitalmente pelo seu administrador.

Capítulo 4

Extensão de Chamadas de Sistema

Quando um núcleo de sistema operacional é projetado, são previstos diversos serviços e funcionalidades que darão suporte para o sistema operacional construído sobre ele, como por exemplo o sistema de arquivos, serviços de rede, alocação de recursos, entre outros. Cada serviço é disponibilizado como uma série de funções com assinaturas próprias [1] e comportamento muito bem específico, sendo este conjunto chamado de API (*Application Programming Interface*) deste núcleo.

Os núcleos de diferentes sistemas operacionais possuem diferentes *APIs*, como a *Win32* [18] do *Microsoft Windows* e a *UNIX98* [32] da maioria dos núcleos *UNIX*, entretanto existe uma interface seguida por praticamente todos os núcleos que é a *POSIX 1003.1* [33]. Uma aplicação que faz uso somente das chamadas de sistema *POSIX* pode, em princípio ser compilada e executada em diferentes sistemas sem ser necessária qualquer modificação em seu código fonte; entretanto são raros os programas construídos desta forma.

O que caracteriza uma *API* é seu comportamento e funcionalidade muito bem conhecida e documentada. Quando novas versões do núcleo são desenvolvidas pode ocorrer o acréscimo de novas funções à *API* ou novas características serem adicionadas as fun-

[1] A assinatura de uma função é caracterizada pelo seu nome e pelo conjunto de parâmetros de entrada e de retorno; ela indica portanto como a função deve ser chamada.

ções já existentes. Essas modificações na *API* do núcleo podem implicar na necessidade de modificações nos programas de aplicação para se adequar às mesmas, o que pode ser inviável. Entretanto, existem várias maneiras de realizar estas alterações sem que seja necessária a modificação da *API* do núcleo, o que será visto a seguir.

4.1 Abordagens para a extensão da API do núcleo

Para se adicionar novos serviços ou modificar serviços existentes é necessário estender os serviços do núcleo e existem diferentes abordagens para isso, cada uma com seus benefícios e desvantagens.

As seguintes abordagens serão detalhadas, na seqüência:

- Modificação direta do núcleo;
- Dispositivos virtuais;
- User-Level Plugins;
- Alteração das bibliotecas dinâmicas;
- API's de interceptação de chamadas de sistema.

4.1.1 Modificação direta do núcleo

De todas as técnicas apresentadas aqui, a modificação do núcleo para incorporar as novas funcionalidades é certamente a mais complexa. O desenvolvedor irá necessitar do código fonte do núcleo[2], grande experiência em programação de baixo nível, provavelmente feita em linguagem *C* e/ou *assembly*, e não irá dispor de ferramentas de depuração para auxiliá-lo. O conhecimento do funcionamento do núcleo que se está alterando, se não de todo, pelo menos da parte que se está modificando, é essencial. Embora com tantos

[2]Praticamente, somente sistemas *Open-Source* tem o código fonte do núcleo liberado. Em outros o código pode ser licenciado, por quantias muito altas, restringindo o acesso somente a grandes corporações.

problemas de implementação, os benefícios são grandes, pois todas as aplicações usarão as novas facilidades, que não podem ser desligadas ou contornadas.

Esta abordagem é utilizada em vários projetos de maneiras distintas. No MOSIX [4], a extensão do núcleo adicionou a facilidade de migração de processos de uma máquina para outra de forma transparente para a aplicação, aproveitando assim os recursos de memória e processamento de máquinas ociosas.

Uma outra utilização foi apresentada em [56], em um projeto que implementou um *Intrusion Detection System* (IDS) baseado na ordem em que as chamadas de sistema são efetuadas por um determinado processo. Por exemplo, um processo servidor de e-mail (*sendmail*) apresenta sempre as mesmas seqüências de chamadas quando recebe uma mensagem, gerando assim uma máquina de estados. Se alguma operação "estranha" for realizada, esta não estará presente na máquina de estados da aplicação e poderá ser bloqueada ou gerar algum alerta.

No projeto *Domain and Type Enforcement* (DTE) [64], as permissões de acesso de usuários e programas foram modificadas através da modificação do núcleo, criando categorias de programas e agrupando-os em domínios. O objetivo principal do projeto é o de permitir a passagem de um domínio para outro mais privilegiado somente depois de uma rigorosa autenticação. Desta forma previne-se que um processo *daemon* de um serviço comprometido, mesmo sendo executado na conta do administrador, tenha acesso de escrita em diretórios de sistema se seu domínio não o permitir, por exemplo. Ou então, que um simples editor de texto tenha acesso aos serviços de rede, ou que o administrador tenha acesso ao arquivo de senhas, entre muitas outras regras que podem ser aplicadas.

4.1.2 Dispositivos virtuais

Se o núcleo estiver preparado para suportar o uso de dispositivos virtuais[3], é possível fazer uso dessa facilidade para a extensão, tornando possível adicionar e retirar novas

[3] Através de *device drivers*, aplicados a dispositivos emulados por software e não de hardware.

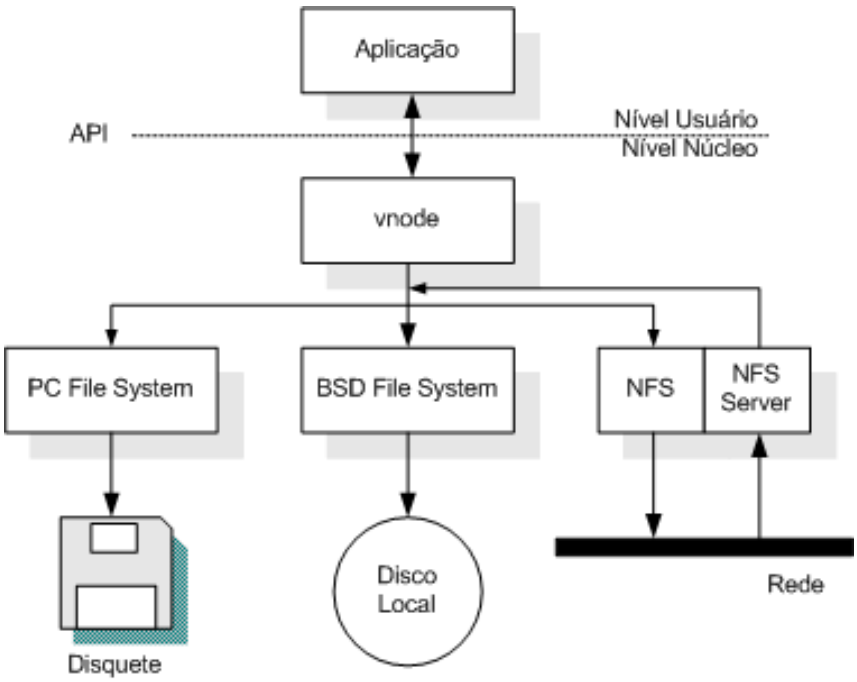

Figura 4.1: Arquitetura de blocos do Vnode.

funcionalidades em tempo de execução. Embora permaneçam os problemas de desenvolvimento da abordagem anterior, não é mais necessária a modificação e recompilação do núcleo para cada novo teste, reduzindo assim as dificuldades de desenvolvimento. O maior uso desta abordagem está no suporte a dispositivos físicos, tais como placas de vídeo, discos, rede e som, ou em dispositivos lógicos como sistemas de arquivo e protocolos de rede.

O projeto *Vnode* [42] é um dos projetos de maior sucesso no uso desta abordagem. A partir dele chegou-se ao conceito de *Virtual File System* (VFS), que permite a utilização de mais de uma dezena de sistemas de arquivos distintos sobre diferentes dispositivos físicos, indo desde o acesso a um disquete formatado em sistema *FAT* na máquina local, até o mapeamento de um servidor *CODA* do outro lado do planeta, com total transparência funcional para a aplicação. Este suporte é feito através de pequenos módulos que implementam cada sistema de arquivos e o mapeiam para o padrão do VFS. Na figura 4.1 pode-se observar a arquitetura do *Vnode*.

O projeto SLIC [30] simplificou um pouco o desenvolvimento usando esta aborda-

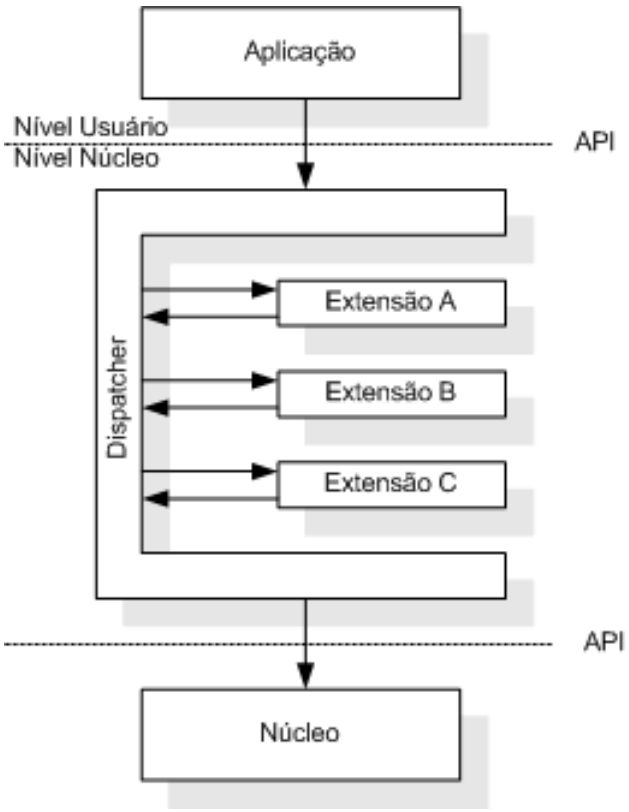

Figura 4.2: Design básico do SLIC.

gem, por se colocar como uma camada intermediária entre a *API* básica do núcleo e a *API* disponibilizada para a aplicação, como pode ser visto na figura 4.2. Todas as chamadas de sistema são capturas pelo *dispatcher* do SLIC. Quando uma chamada é feita pela aplicação, o *dispatcher* verifica se não existe uma extensão registrada para essa chamada, invocando-a caso exista. Uma característica interessante do *dispatcher* é que ele mantém a ordem de registro de cada extensão, permitindo que uma mesma chamada seja tratada por várias extensões diferentes, adicionando cada vez mais funcionalidades à mesma. Além disto, a extensão pode ser instanciada tanto em nível núcleo ou usuário, permitindo desta forma um fácil desenvolvimento da extensão em nível usuário e posterior uso em nível núcleo, para um melhor desempenho.

Versões recentes do núcleo *LINUX*, introduziram uma infra-estrutura de segurança chamada *Linux Security Modules* [66] (*LSM*) que incorpora mais de 500 pontos de interceptação espalhados dentro do núcleo, não se restringindo somente às chamadas de

sistema. Cada ponto repassa o controle para módulos externos ao núcleo, onde a ação e decisão é feita. Estes módulos possuem o conceito de registro parecido com o sistema *SLIC*, permitindo que módulos diferentes atuem sobre o mesmo ponto, e que possam ser carregados e descarregados em tempo de execução, porém a ordem de registro é importante.

Estrategicamente dispostos nos pontos onde haja necessidade de controle efetivo, a interceptação desses pontos praticamente não introduziu aumento nos tempos de execução, consistindo basicamente de chamadas a funções nos módulos registrados. É possível a construção de módulos que alteram drasticamente o comportamento do núcleo sem que seja necessária qualquer modificação no núcleo. Atualmente já foram portados para ele alguns sistemas que eram desenvolvidos através de outras formas de extensão, dos quais pode-se destacar o *DTE*, já citado anteriormente, o *LIDS* [61], um sistema de detecção de intrusão e um subset do POSIX 1e [34] que define *capabilities*; todos esses sistemas eram anteriormente baseados na modificação do núcleo.

Um indicativo do possível sucesso desta abordagem no mundo LINUX é o suporte e apoio efetivo da equipe de desenvolvimento do núcleo, em especial do próprio Linus Torvalds. Na figura 4.3, é exemplificado um ponto de interceptação no acesso a um arquivo, que internamente é representado por um i-node.

4.1.3 User-level Plug-ins

Esta abordagem é uma solução mista entre os níveis usuário e núcleo, sendo colocados ganchos (pontos de interceptação) no núcleo para repassar as chamadas para processos em nível usuário. Estes ganchos são simples de desenvolver e apresentam baixa complexidade, tornando as modificações necessárias no núcleo mais fáceis. Toda a complexidade é transferida para os processos em nível usuário, o que apresenta diversas vantagens.

Como citado anteriormente, o projeto SLIC [30] permite que a extensão possa ser executada como um processo, como pode ser visto nas figuras 4.4 e 4.5. Mesmo sendo

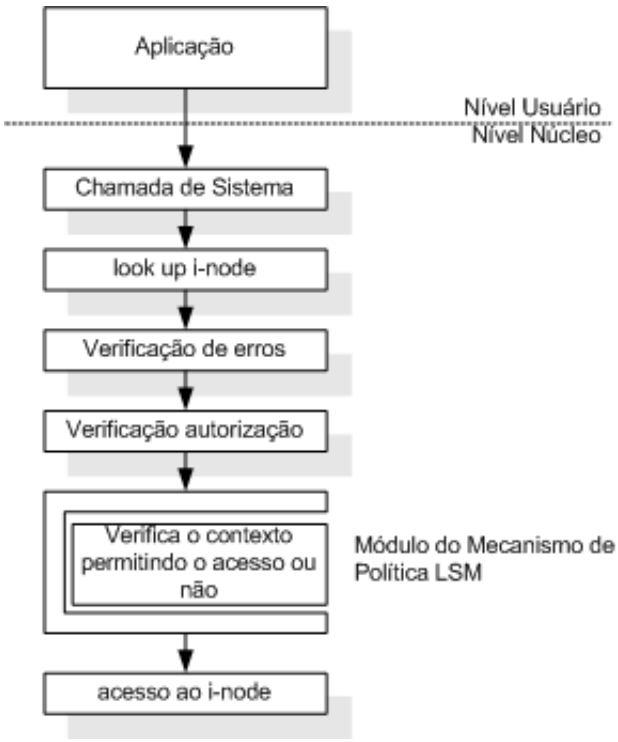

Figura 4.3: Mecanismo básico do funcionamento do LSM.

executado com um processo, a segurança do mesmo é garantida pela proteção que o próprio núcleo oferece a partir do isolamento do espaço de endereçamento entre os processos.

4.1.4 Alteração das bibliotecas dinâmicas

Normalmente uma aplicação não acessa diretamente o núcleo para usar algum serviço, mas faz uso de bibliotecas de função que implementam serviços com um grau de abstração maior que os oferecidos pelo núcleo, podendo implementar procedimentos complexos envolvendo várias chamadas de sistema. Portanto, se a biblioteca for modificada para oferecer mais funcionalidades, todas as aplicações que fazem uso dela serão beneficiadas.

Porém, existe um problema nesta abordagem: a aplicação pode contornar a biblioteca chamando diretamente o núcleo, o que em alguns casos não é desejável. Se o objetivo da extensão for a aplicação de restrições de acesso, formatação de parâmetros para o núcleo ou qualquer outra atividade relacionada a segurança, esta solução é totalmente inadequada.

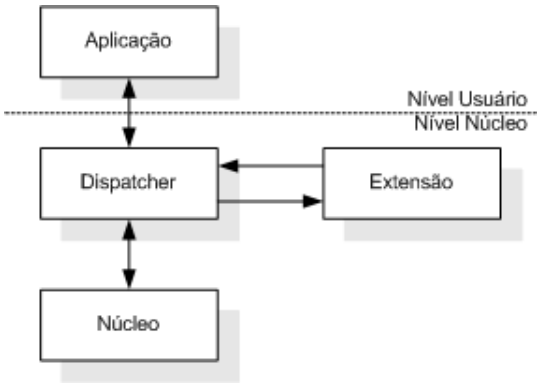

Figura 4.4: SLIC com extensão em nível núcleo.

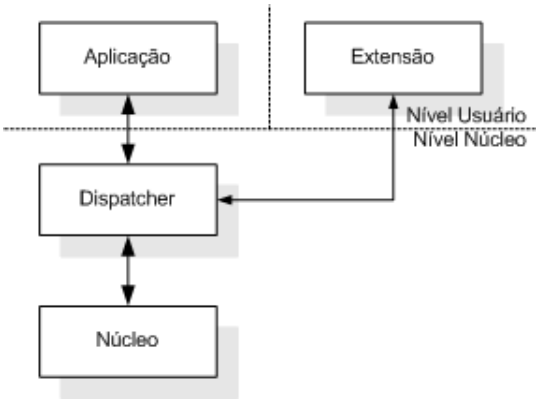

Figura 4.5: SLIC com extensão em nível usuário.

Em [60] é utilizada esta abordagem para prover a facilidade de uso de *checkpoints* a processos em execução. Isto permite que processos que tenham grande tempo de execução, como simulações por exemplo, possam ter seus estados intermediários salvos em disco, possibilitando sua re-execução a partir destes pontos. Também são construídos mecanismos para permitir a migração do processo para outra máquina através da mesma abordagem.

4.1.5 API's de interceptação de chamadas de sistema

Em alguns núcleos existe uma funcionalidade que permite que as chamadas de sistema de um processo sejam interceptadas e repassadas para outro processo. Esta abordagem é toda feita em nível usuário, dispensando qualquer acesso privilegiado no sistema.

Sua aplicação mais corriqueira é em depuração das chamadas de sistema de uma aplicação, através do programa *strace* dos sistemas *UNIX*, que usa a chamada de sistema *ptrace*. Esta chamada permite interceptar as chamadas de sistema de outros processos. Ele permite a visualização de cada chamada ocorrida, com seus parâmetros e resultados, mesmo através de processos filhos criados pelo processo monitorado. Pode-se inclusive escolher um conjunto específico de chamadas para monitoração.

Objetivando elevar o nível de abstração das chamadas de sistema, foi criado em [39] um conjunto de ferramentas que permitia a construção de extensões sob a forma de agen-

tes, possibilitando o reuso de código, com maior facilidade de modelagem do problema e de construção da extensão. Mas a solução apresenta um sério problema de desempenho, podendo o custo adicionado pelo procedimento de extensão chegar a mais de 50% do custo total, devido ao grande número de trocas de contexto.

Com esta abordagem é possível até mesmo montar um sistema de arquivos remoto com os direitos de um usuário simples[4], através da captura das chamadas de uma aplicação às funções do sistema de arquivos. Isto foi realizado no projeto *UFO Global File System* [1], que mapeia páginas Web e diretórios de FTP como diretórios do usuário.

4.2 Análise comparativa das técnicas

Cada uma das técnicas apresentadas possui benefícios e desvantagens dependendo do objetivo desejado para a extensão: por exemplo, se o objetivo é atingir todas as aplicações do sistema operacional de maneira incondicional, a modificação do núcleo ou o uso de um dispositivo virtual é uma solução excelente, porém somente o administrador pode fazê-lo. Porém se o objetivo é adicionar a facilidade de *checkpoints* e *rollbacks* a uma aplicação qualquer, a interceptação de chamadas de sistema é a ideal.

Cada uma das técnicas de extensão possui características próprias com relação ao que é modificado, ao nível de privilégio necessário, ao alcance das modificações e ao grau de interferência no desempenho, que são sumarizadas na tabela 4.1.

Durante o projeto de uma extensão deve-se escolher em que nível ela será executada, pois disto dependem todas as demais etapas de desenvolvimento. Cada nível tem características próprias, representando custos e benefícios para o projeto.

O desenvolvimento intra-núcleo caracterizado nas abordagens “Modificação direta do núcleo”, “Dispositivos virtuais"e “*API*s de interceptação de chamadas de sistema”, apresenta as seguintes vantagens e desvantagens:

- Vantagens:

[4]Normalmente um usuário simples não tem este direito, somente o administrador.

Método	Nível	Modifica fonte do núcleo	Requer Acesso de Administrador	Relinka Aplicações	Aplicações Atingidas	Custo no Desempenho
Modificação direta do núcleo	Núcleo	Sim	Sim		Todas	Muito Baixo
Dispositivos virtuais	Núcleo		Sim		Todas	Muito Baixo
User-level Plug-ins	Misto	Uma vez			Todas	Muito Baixo
Alteração das bibliotecas dinâmicas	Usuário			Algumas	As dinamicamente Linkadas	Baixo
API de interceptação de chamadas de sistema	Núcleo				Todas	Alto

Tabela 4.1: Funcionalidades e limitações dos diferentes métodos de extensão do sistema operacional. (Células em branco indicam "**Não**")

  - Acesso a todas as funções internas do núcleo, assim como áreas de memória dos processos.
  - Baixo custo de execução pois elimina a necessidade de cópias de dados entre os níveis.
  - As modificações não podem ser contornadas.

- Desvantagens:
  - Grande dificuldade de projeto e implementação.
  - Não é possível a utilização direta de bibliotecas de funções.
  - Necessidade de alteração do código fonte do núcleo, dificultando futuras atualizações.
  - Não é possível utilizar ferramentas de depuração.
  - O tempo de desenvolvimento é grande.

O desenvolvimento em modo usuário, caracterizado nas aborgagens "User-Level Plugins" e "Alteração das bibliotecas dinâmicas", apresenta as seguintes vantagens e desvantagens:

- Vantagens:
  - É possível utilizar-se do estado-da-arte em ferramentas de programação e depuração.
  - Não é necessário conhecer detalhes de funcionamento interno do núcleo.

  - Qualquer biblioteca de funções é passível de utilização.
  - Pequeno tempo de desenvolvimento.
  - O porte para outros núcleos é mais fácil.

- Desvantagens:
  - Existe um custo de desempenho originado pelas cópias de dados entre o nível núcleo e usuário, em cada chamada de sistema.
  - Dependo das modificações, estas podem ser contornadas pelas aplicações.

Se as modificações forem poucas e isoladas, o desenvolvimento inteiramente em nível núcleo é bem interessante, pois as vantagens superam em muito as desvantagens. Porém com muitas modificações, haverão muitos problemas, principalmente em depuração.

4.3 Conclusão

Existem diversas formas de se estender os serviços de um núcleo de sistema operacional, cada qual com diferentes características intrínsecas que as tornam indicadas para diferentes necessidades. As formas de extensão apresentadas neste capítulo contemplam a grande maioria das soluções para as necessidades conhecidas, sendo assim possível a escolha da mais indicada e viável.

Capítulo 5

A3 - Arquitetura de Autenticação de Arquivos

Conforme a exposição realizada no capítulo 2, uma das atitudes mais comuns em uma invasão é a adulteração de arquivos do sistema operacional visando a instalação de um *back-door* não detectável e a remoção das pistas deixadas pela invasão atual, através da substituição de alguns arquivos binários (executáveis e bibliotecas) e de configuração.

Teoricamente os mecanismos de autorização do sistema operacional deveriam impedir que essas modificações fossem realizadas, entretanto os ataques exploram justamente as vulnerabilidades de serviços que são executados com direitos administrativos, contornando assim os mecanismos de autorização. Isto não significa que tal sistema falhou, mas sim, que seu modelo é inadequado para trabalhar com serviços que podem apresentar falhas.

A arquitetura aqui proposta, denominada A3 (*Arquitetura de Autenticação de Arquivos*), visa monitorar determinados arquivos com o intuito de evitar seu uso caso sejam modificados. Em um segundo plano, é vetada a abertura para modificação destes mesmos arquivos monitorados, o que é particularmente interessante para arquivos de configuração que normalmente são modificados e não substituídos nas invasões. Este isoladamente, sem que sejam feitas quaisquer verificações de integridade, já inibiria a grande maioria

das modificações indevidas nos sistemas, mesmo sendo tão simples.

Desta forma consegue-se dar ao sistema duas boas características:

- a capacidade de detecção imediata de qualquer modificação do sistema;
- a capacidade de não operar em modo comprometido;

A primeira característica representa um grande avanço em relação aos sistemas atuais de verificação de integridade, tais como o *TRIPWIRE* [41, 62], cuja operação ocorre em modo *batch* em uma hora de baixa carga no sistema, normalmente na madrugada ou até mesmo semanalmente. Embora estes sistemas de verificação sejam muito eficientes, eles apresentam uma detecção tardia do ocorrido, o que pode ser muito tarde em alguns casos.

A segunda característica apresenta duas visões do mesmo aspecto, dependendo da forma de que se observa tal característica. O sistema impede que qualquer serviço opere em modo comprometido, sendo assim, todo o serviço disponibilizado necessariamente estará íntegro. Porém em algumas circunstâncias é interessante que um serviço continuasse operando, mesmo que comprometido, como um sistema de missão crítica onde é melhor algum controle do que nenhum.

A seguir será feita a apresentação da arquitetura proposta, na seção 5.1, seguida da descrição de seus módulos, na seção 5.2, e do roteiro básico de funcionamento, na seção 5.3.

5.1 Arquitetura Proposta

Em uma visão mais geral, a arquitetura proposta comporta-se como uma extensão ao núcleo, onde nada de seu comportamento é alterado com exceção da política de acesso aos arquivos, que agora também depende da integridade dos mesmos. Dentre as abordagens citadas na seção 4, esta arquitetura enquadra-se no modelo de dispositivos virtuais, conforme exposto na seção 4.1.2. Esta abordagem foi escolhida por apresentar menor complexidade, pois minimiza as modificações no núcleo existente, por acrescentar menor

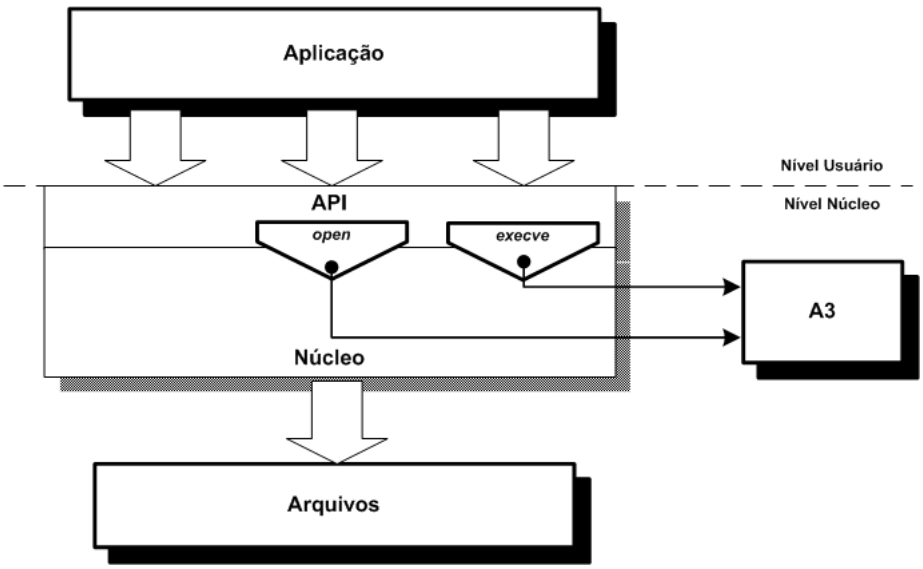

Figura 5.1: Arquitetura Básica

custo computacional no processo de interceptação da chamada, por não permitir ser desligada ou contornada. Embora o processo de desenvolvimento inteiramente em núcleo seja mais complexo que outras abordagens, os ganhos de desempenho compensa.

Na figura 5.1 pode-se observar quais são os pontos de interação do núcleo existente com a extensão, que basicamente são os pontos a onde as aplicações interagem com o sistema de arquivos.

Para a escolha destes pontos foi estudada a forma de como as aplicações acessam os arquivos e como o núcleo os mapeia e os utiliza.

O sistema de arquivos é uma abstração para a aplicação, que os vê como objetos endereçáveis a partir de um caminho, mas que fisicamente são setores em um disco, por exemplo. Quem realiza esta abstração é o núcleo, que fornece este serviço através das chamadas de sistema. Portanto para uma aplicação obter acesso a um arquivo, tem necessariamente de solicitar ao núcleo. Existem diversas chamadas de sistema disponibilizadas, mas somente duas são necessárias para se alcançar o objetivo proposto por esta arquitetura, pois estas outras após um tratamento dos parâmetros acabam chamando estas duas.

Para uma aplicação ser executada, deve-se efetuar uma chamada de sistema *execve*, fornecendo-se nela o caminho para o arquivo desejado e os seus parâmetros de execu-

ção. Para um arquivo ser aberto para leitura, escrita ou modificação deve-se efetuar uma chamada de sistema *open* passando-se nela o caminho para o arquivo e os parâmetros de abertura. Mesmo os arquivos de biblioteca de funções que o núcleo abre junto com a aplicação são feitos através da chamada *open*.

Portanto, realizando-se a verificação da integridade dos arquivos passados para estas duas chamadas de sistema, garante-se que nenhum arquivo adulterado estará sendo acessado. Assim sendo, estas duas chamadas de sistema foram interceptadas em nível núcleo, transferindo o controle ao sistema **A3** para verificação de integridade dos arquivos passados para estas chamadas através do processo de assinatura de documentos visto na seção 3, liberando-se ou não seu acesso para a aplicação conforme sua integridade.

Optou-se por assinar cada arquivo individualmente, gerando uma assinatura por arquivo. Uma outra solução seria a de assinar somente o arquivo onde se encontram os valores de HASH de cada arquivo do sistema. A solução proposta permite que se distribua a assinatura através de rede ou do disco, podendo-se fazer a busca individualmente. Também permite que seja usada uma infra-estrutura de PKI para que alguns usuários assinem seus próprios arquivos, embora isto não seja abordado aqui.

Por mais rápida que seja a máquina onde são executadas as rotinas de verificação de assinatura, seria impraticável realizá-las para cada acesso. São centenas de acessos por segundo e alguns dos arquivos são grandes, como as bibliotecas que possuem alguns MB, levando a latências de alguns segundos na inicialização de cada processo. Para contornar este problema foi introduzida na arquitetura um mecanismo de caching, na forma de uma lista com os arquivos já verificados (cujos mecanismos de manutenção da lista serão detalhados a seguir). Isto permitiu que após um período inicial de uso a arquitetura apresentasse uma sobrecarga muito baixa, conforme poderá ser visto na seção 6.

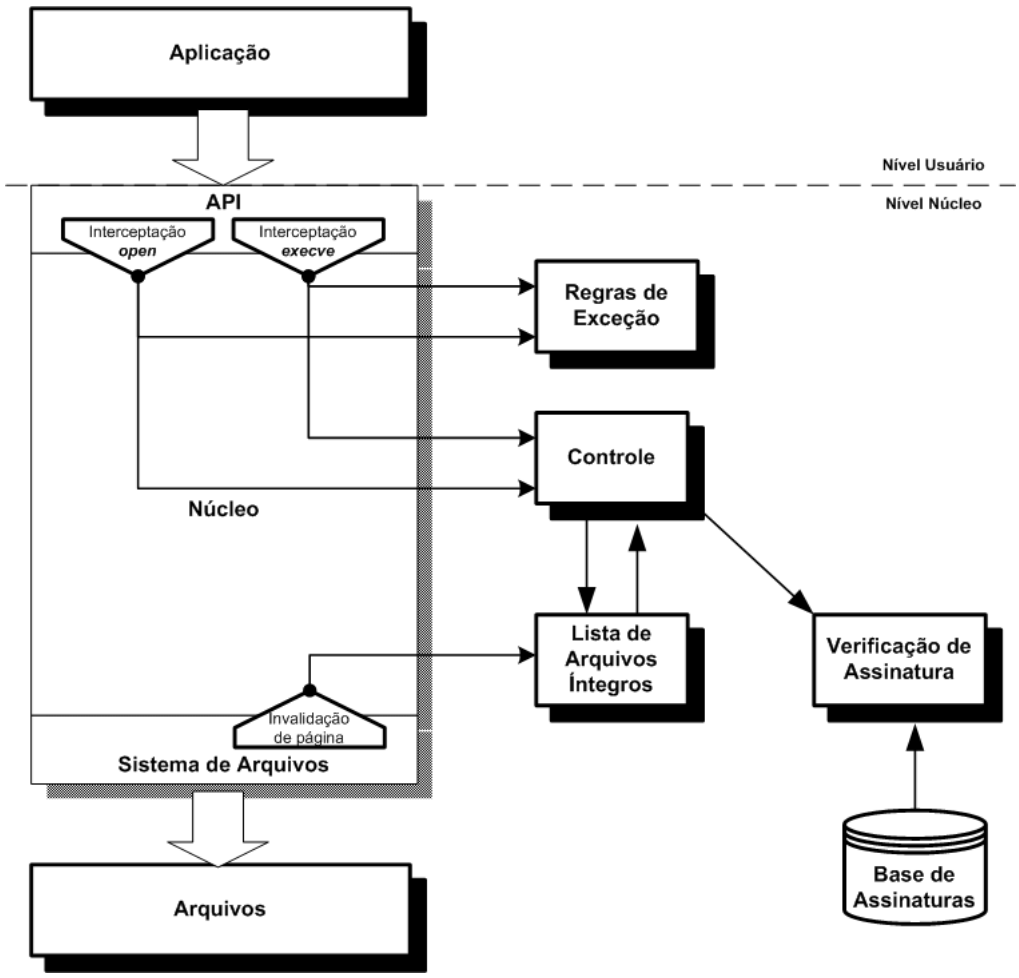

Figura 5.2: Arquitetura: Visão Modular

5.2 Descrição da Arquitetura

Por opção de projeto, escolheu-se o desenvolvimento inteiramente em nível núcleo, por apresentar melhores resultados, mesmo sabendo-se da maior dificuldade, conforme apresentado na seção 4. Na figura 5.2 pode-se observar a composição da arquitetura representada por seus sete módulos funcionais, que serão apresentados a seguir.

5.2.1 Módulo de interceptação da chamada de sistema *open*

Neste módulo é interceptada a chamada de sistema *open*, responsável pela abertura, criação e modificação de arquivos, diretórios e dispositivos. Ela apresenta a seguinte assinatura de função:

```
int open (const char *path, int flag, mode_t mode)
```

A grande maioria dos aplicativos faz uso de bibliotecas de funções que são carregadas dinamicamente pelo núcleo quando necessário, como forma de reutilização de código, bem como de diminuição do tamanho da aplicação, entre outras. Embora estas bibliotecas tenham seu código executado, elas são carregadas em memória através da chamada de sistema *open*, sendo portanto também verificadas quanto à sua integridade.

O núcleo *LINUX* permite a carga durante seu funcionamento, de drivers de dispositivo ou então pequenas partes do núcleo, chamados *módulos* que foram compilados separadamente. Durante este processo o próprio núcleo faz uso da chamada de sistema *open*, o que por sua vez evita que módulos ou drivers comprometidos sejam executados.

A seguir será feita uma apresentação desta chamada com ênfase nos detalhes relevantes ao seu uso na proposta, sendo que uma descrição mais completa pode ser obtida nos manuais do próprio sistema[1].

O primeiro parâmetro é o caminho completo para o arquivo que se deseja acessar, representado por uma cadeia de caracteres, como por exemplo: `/etc/passwd`. No segundo parâmetro tem-se os flags que descrevem a forma de abertura deste arquivo, podendo ser por exemplo: de somente leitura, escrita, inclusão no final, dentre várias outras formas. O terceiro parâmetro descreve o modo de abertura na criação de um novo arquivo.

Nos sistemas de arquivos utilizados em sistemas *UNIX* existe um tipo especial de arquivo chamado *link* simbólico. Um *link* simbólico nada mais é do que uma referência para um arquivo que fisicamente está em outro lugar, ou seja, é um apontador.

Haviam duas formas de se tratar este tipo de arquivo: uma seria de forma especial com suas peculiaridades e a outra seria como um arquivo normal, resolvendo a referência para o arquivo para o qual aponta. Neste caso o caminho para o arquivo é mantido inalterado, porém o conteúdo é do arquivo apontado. Esta segunda forma foi a escolhida, pois

[1] Através do comando: "*man 2 open*", em sistemas UNIX.

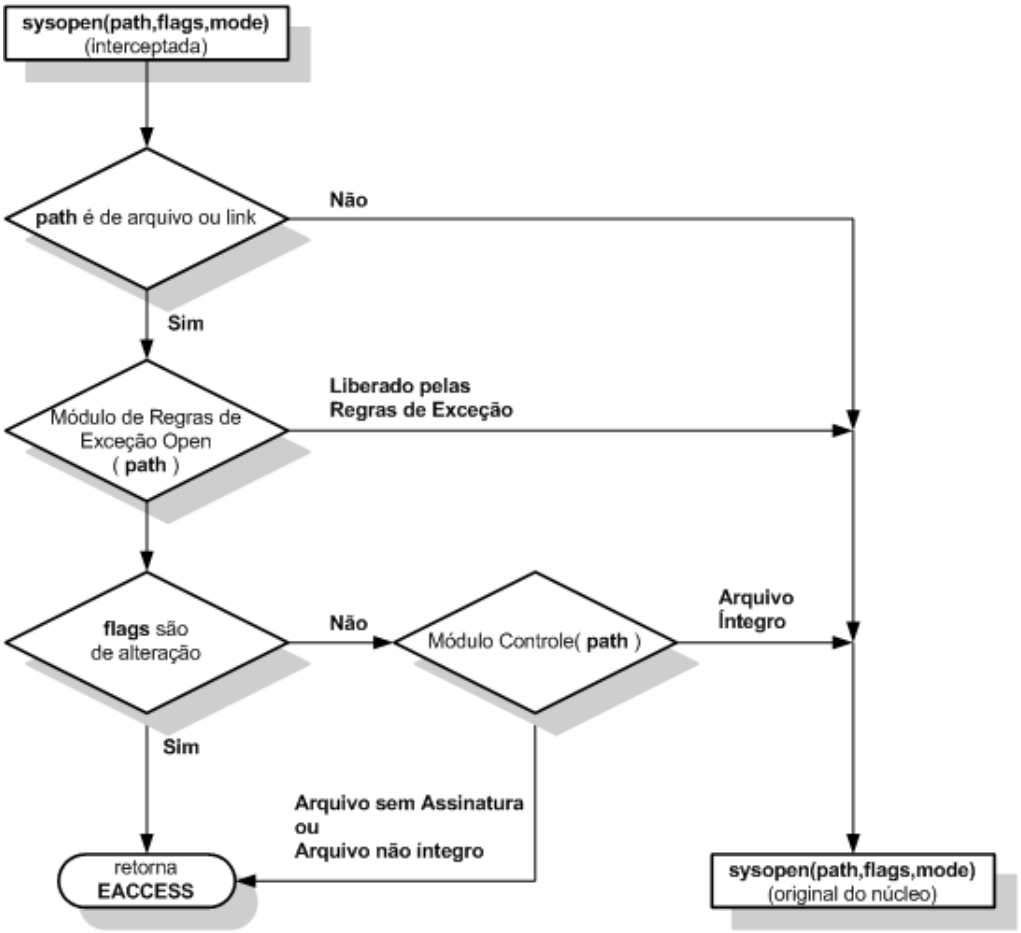

Figura 5.3: Arquitetura: Interceptação da chamada de sistema *open*.

simplifica a arquitetura e não altera a proteção proporcionada.

Como citado anteriormente, esta mesma chamada de sistema é também utilizada na abertura de outros recursos como dispositivos e diretórios e não somente arquivos. Em virtude disto, é aplicado um filtro para que somente os arquivos e *links* sejam repassados ao módulo de controle.

Sua lógica de funcionamento é simples, como pode ser visto na figura 5.3. Primeiramente inspeciona-se o caminho passado como parâmetro, verificando se aponta para um arquivo ou *link* para arquivo; em caso negativo, o controle é devolvido à chamada de sistema *open* original do núcleo.

Em seguida, consulta-se o módulo de regras de exceção, seção 5.2.6, para verificar se é possível liberar imediatamente o acesso ao arquivo, sem que seja preciso verificar

sua integridade. Caso liberado é chamada a rotina original do núcleo. Mas em caso contrário, antes de ser feita a verificação de integridade é efetuado mais um teste que melhora muito a resistência do sistema a ataques. Partindo-se do princípio de que um arquivo assinado não deve ser alterado a não ser pelo administrador através das rotinas de manutenção, envolvendo inclusive a geração de uma nova assinatura para o arquivo, não existem motivos para permitir que este arquivo seja aberto para alterações. Assim sendo, são analisados os *flags* passados na chamada, verificando se podem levar a alterações, tais como escrita e truncamento para inclusão a partir do fim do arquivo. Caso alterações sejam possíveis a chamada retornada à aplicação com o código de erro EACCESS.

Finalmente, é chamado o Módulo de Controle, que irá conduzir os procedimentos de verificação de integridade, que retornarão uma resposta simples: “íntegro”, “não íntegro” ou “sem assinatura”. Se a resposta for “não íntegro” ou “sem assinatura” é retornado imediatamente para a aplicação o código de erro EACCESS, que indica negação de acesso. Este é exatamente o mesmo código de erro que a chamada de sistema original do núcleo retornaria em caso de negação de acesso por falta de direitos sobre o arquivo, o que é interessante pois não acrescenta-se nenhum código de erro novo, sendo o código empregado coerente com o tipo de erro.

Estando o arquivo íntegro e sem problemas com os flags, é então evocada a chamada de sistema original do núcleo, e o seu resultado retornado à aplicação.

5.2.2 Módulo de interceptação da chamada de sistema *execve*

Neste módulo é interceptada a chamada de sistema *execve*, responsável pela carga de um arquivo executável, alocação dos recursos necessários, tais como: memória, estruturas de núcleo e a carga de bibliotecas dinâmicas. Ela apresenta a seguinte assinatura de função:

```
int execve(struct pt_regs regs)
```

Seu único parâmetro é uma estrutura de dados contendo diversos valores utilizados pela chamada, porém o único valor que é utilizado por este módulo é a entrada *regs.ebx*, que é o caminho do arquivo que será executado. Mais detalhes com relação aos demais parâmetros podem ser obtidos nos manuais do próprio sistema[2].

Existem diversas outras chamadas de sistema com o radical *exec*, mas com diferentes assinaturas, entretanto internamente estas chamadas tratam seus parâmetros preenchendo a estrutura *regs* e invocam a chamada *execve* (isto é verdade para o núcleo *LINUX*, mas não é possível afirmar essa característica nos demais núcleos).

Sua lógica de funcionamento também é simples, como pode ser visto na figura 5.4. Primeiramente, recupera-se o caminho para o arquivo. Em seguida é consultado o módulo de regras de exceção, seção 5.2.6, para saber se é possível ocorrer a liberação imediata para execução. Caso não seja, é chamado o Módulo de Controle que irá conduzir os mesmos procedimentos já descritos na seção anterior. Estando o arquivo íntegro é evocada a chamada de sistema original do núcleo, e o seu resultado é então retornado à aplicação. Estando o arquivo adulterado, é retornado imediatamente para a aplicação o código de erro `EACCESS`, impedindo sua execução.

Diferentemente do padrão adotado para o módulo da chamada de sistema *open*, aqui é bloqueado o acesso e a conseqüente execução de arquivos que não tenham assinaturas, retornando-se para a aplicação o código de erro `EACCESS`. Este bloqueio impede que novos arquivos sejam executados sem que o administrador do sistema tenha conhecimento. Execuções de novos arquivos são muito comuns quando são instalados *rootkits* e *exploits* em invasões. Isso não impede no entanto os usuários desenvolvedores de compilar e executar seus programas. Para isto é necessário apenas que o diretório de trabalho dos usuários (normalmente `/home`) esteja habilitado por uma regra de exceção.

[2] Através do comando: "*man 2 execve*", em sistemas UNIX.

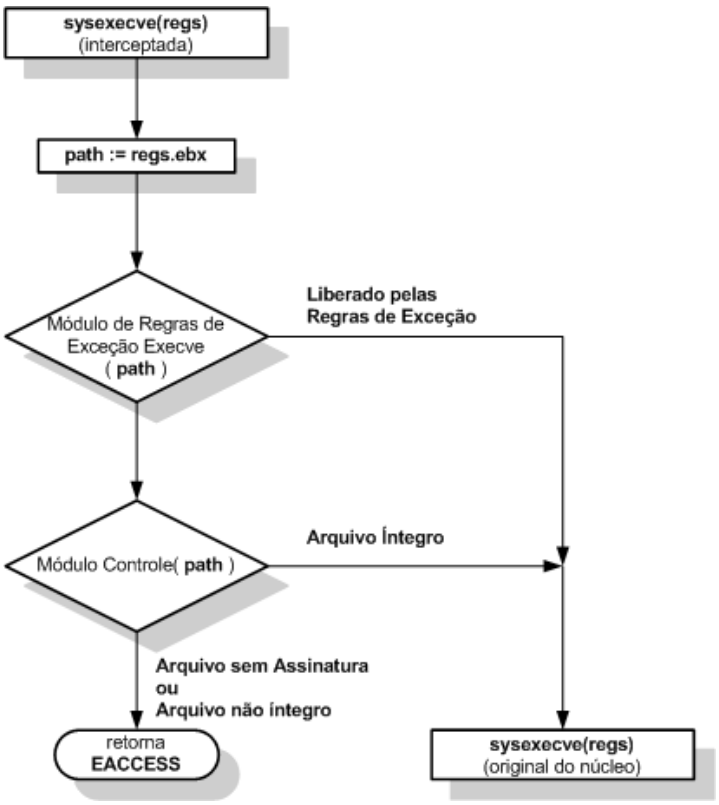

Figura 5.4: Arquitetura: Interceptação da chamada de sistema *execve*.

5.2.3 Módulo de controle

Neste módulo são realizadas as chamadas aos demais módulos, cuja lógica de controle pode ser vista na figura 5.5.

Primeiramente é convertido o caminho para o arquivo de sua localização lógica (caminho textual) para sua localização física, isto é, são obtidos o *i-node* e o *device* que possuem fisicamente o conteúdo do arquivo. De posse destes valores, é consultado o módulo *Lista de Arquivos Íntegros*, que possui uma lista com todos os arquivos já verificados e que ainda se encontram íntegros. Caso o arquivo ainda existir na lista deste módulo, é retornado imediatamente ao módulo de interceptação a informação de que este arquivo está íntegro.

Caso contrário, é solicitado ao *Módulo de Verificação de Integridade* que teste este arquivo. Este fará todos os procedimentos necessários e retornará um dos três estados: íntegro, não íntegro e sem assinatura. Caso esteja íntegro, será adicionada uma entrada na

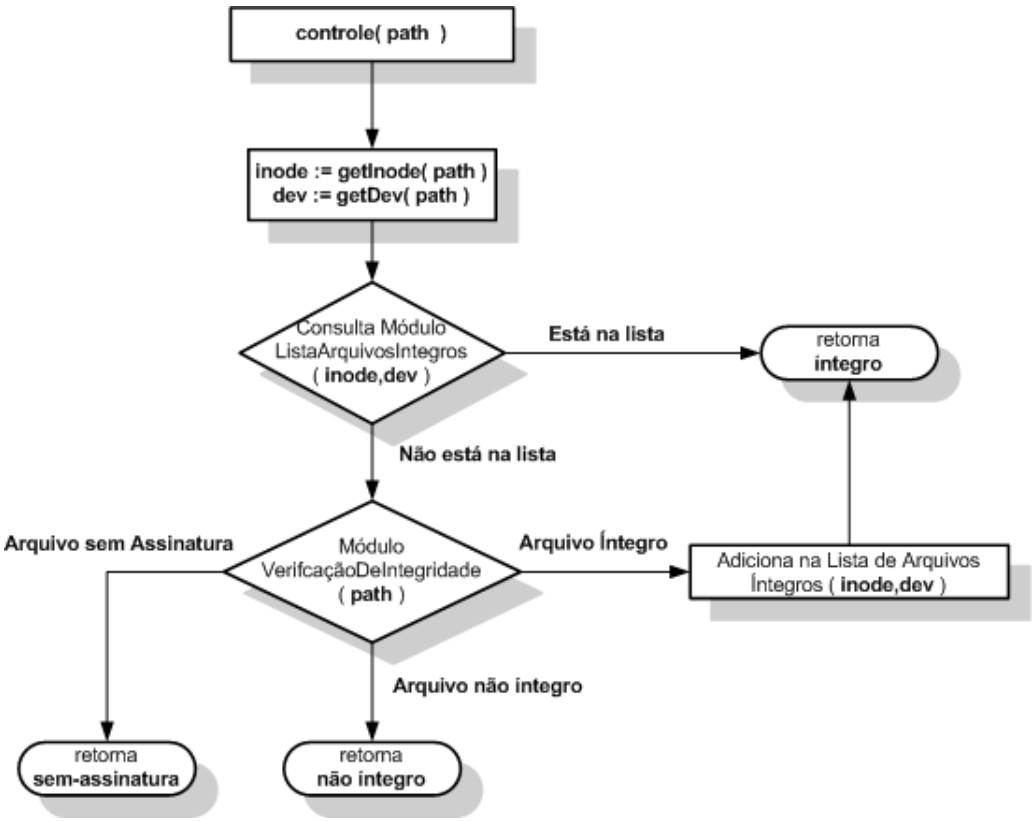

Figura 5.5: Arquitetura: Controle.

Lista de Arquivos Íntegros, através do módulo correspondente.

5.2.4 Módulo lista de arquivos íntegros

Este módulo é responsável por controlar a lista de arquivos que já foram verificados e ainda se encontram íntegros. Existem 3 operações possíveis: consulta, inclusão e remoção. Cada entrada nesta lista é composta por um par (*i-node* | *device*), que identifica unicamente o arquivo no núcleo.

Cada vez que um arquivo é acessado, os módulos de interceptação passam a consulta para o módulo de controle, que realiza a busca nesta lista. Se uma entrada não for encontrada nesta lista significa que o arquivo nunca foi verificado antes ou que o arquivo foi modificado.

Quando o núcleo é carregado, esta lista está vazia. Ela é preenchida sob demanda pelo módulo de controle, ou seja, à medida em que os arquivos vão sendo verificados e estando

íntegros, suas entradas vão sendo inseridas nela. Não é mantido nenhum histórico sobre estados anteriores dos arquivos, tampouco sobre execuções anteriores do núcleo.

A remoção ocorre quando o módulo de interceptação de invalidação de página (seção 5.2.5) detecta que o arquivo foi modificado em memória e vai ser modificado fisicamente. Isto faz com que, em um novo acesso, o módulo de controle faça uma nova verificação desse arquivo.

Esta lista foi implementada na forma de uma tabela hash. Para a configuração desta tabela foram analisadas várias instalações de distribuições Linux [3] e observou-se que em média são instalados 128 mil arquivos entre binários, bibliotecas e arquivos de configuração. Em função disto foi escolhida uma tabela com 4096[4] entradas, o que implica em uma profundidade de busca média de 32.

Por estas operações sobre a lista serem naturalmente paralelas e assíncronas, foram protegidas com semáforos, em um esquema leitores-redatores com prioridade para redatores. Várias consultas podem ocorrer simultaneamente, porém as operações de inclusão e remoção bloqueiam novas consultas depois que encontram o local exato de modificação e esperam pelo término das operações em curso, quando então assumem exclusivamente as modificações sobre a tabela. Este é o único ponto em toda arquitetura onde ocorre serialização na execução.

Como não houve detecção de problemas de desempenho devido ao bloqueio completo da tabela, não foi necessário o uso de um bloqueio mais granulado, como o bloqueio por entrada na tabela, por exemplo.

5.2.5 Módulo de interceptação da função de invalidação de página

Todo arquivo acessado pelo núcleo é mapeado pelo *Virtual File System*(VFS) em seus *buffers* e *caches* como páginas na memória. Tudo o que as demais partes do núcleo enxergam é esta visão que o *VFS* fornece. Quando um arquivo não é mais necessário, suas

[3] Distribuições Linux RedHat 7.1 e Conectiva 6.0

[4] Esta foi uma escolha empírica, baseado no número de arquivos das distribuições.

páginas são marcadas como descartáveis, liberando espaço para outros arquivos. Porém quando o arquivo é modificado, estas páginas são marcadas como sujas, para que posteriormente possam atualizar o dispositivo físico através da gravação destas páginas. Existe ainda uma terceira situação que ocorre quando o dispositivo físico é modificado por outro mecanismo, sendo estas páginas também marcadas como sujas para serem lidas novamente a partir do dispositivo físico. Esta situação ocorre por exemplo quando é montado um diretório via *NFS* e o arquivo é modificado remotamente no servidor, sendo esta modificação propagada para este cliente.

No núcleo existe uma função, pertencente ao *VFS*, por onde passam todas as chamadas de invalidação de página, chamada `__mark_inode_dirty`. Os parâmetros dessa função indicam a localização física do arquivo mapeado e o tipo de invalidação que está sendo feita. Esta função é específica dos núcleos *LINUX* mas existem equivalentes em outros núcleos, porém com outros nomes. Em todos os casos, arquivos abertos cujo conteúdo foi modificado têm suas páginas marcadas como sujas, mais especificamente pelo flag `I_DIRTY`.

O que este módulo faz é capturar todas as chamadas a esta função cujo flag seja `I_DIRTY` e invalidar a entrada correspondente a esse arquivo na lista de arquivos íntegros (seção 5.2.4). Embora este procedimento envie operações de remoção de entradas sobre a lista de arquivos íntegros para arquivos que não existem nela, isto não causa problemas, pois a lógica de busca só bloqueia a lista caso encontre a entrada nela, caso contrário a remoção funciona somente como uma busca sem êxito.

5.2.6 Módulo de regras de exclusão

Neste módulo verifica-se a existência de alguma regra que libere o acesso ao arquivo diretamente, sem que seja preciso passar por todas as regras de verificação de integridade. As regras estão dispostas em um arquivo com a seguinte estrutura:

(`caminho, tipo de acesso, usuário, grupo`)

A sintaxe dos campos é a seguinte:

- *caminho* é a máscara de liberação. Se a máscara for de um arquivo, com em: `/home/mauro/bin/pgp.sh`, então somente para ele será liberado o acesso. Mas se apontar para um diretório, como em `/home/mauro/bin/`, então toda a sub-árvore de diretórios deste ponto para baixo será liberado.

- *tipo de acesso* indica a forma de acesso liberado, ou seja, *open* e/ou *execve*.

- *usuário* e *grupo* indicam para qual grupo e usuário está liberado o acesso. "`*`" indica para todos e "`-root`" indica para todos exceto se o usuário for o *root*; também pode ser indicado um usuário ou grupo específico, com em "`mauro, devel`".

Desta forma é possível liberar aos usuários a alteração/manipulação de seus arquivos, bloqueando a execução de novos binários, com exceção dos usuários desenvolvedores, por exemplo em:

```
(/home/ , open , * , *)
(/home/devel/mauro/projetos/ , exec , mauro , kernel_devel)
(/home/devel/joel/projetos/ , exec , joel , db_devel)
```

5.2.7 Módulo de verificação de integridade

Neste módulo é feita a verificação da integridade do arquivo através do algoritmo apropriado de assinatura digital de documentos, como descrito na seção 3.3. Para tanto, foi utilizada a biblioteca de funções de criptografia *MIRACL* [43]. Desta biblioteca de foram extraídos somente os códigos necessários para o uso da verificação de assinatura explicitamente utilizados neste módulo. Estes códigos foram adaptados para que fosse possível seu uso em nível núcleo, pois sua aplicação original era em nível usuário.

A assinatura digital de cada arquivo está localizada em um único arquivo texto contendo todas as assinaturas, composto pelos caminhos dos arquivos e suas assinaturas.

Este arquivo é carregado em memória durante a carga do núcleo, montando-se assim uma tabela (no capítulo 7 são apresentadas alternativas a esta solução). As assinaturas dos arquivos foram gerados com o uso da chave privada do par de chaves gerado para o sistema. Somente a chave pública fica disponível no sistema para uso deste módulo.

De posse do caminho completo do arquivo é localizada sua assinatura na tabela de assinaturas. Caso esta não seja encontrada é retornado ao módulo de controle a informação de que não existe assinatura para este arquivo.

Existindo a assinatura, então é calculado o valor de hash atual para o arquivo. Este valor é calculado através da aplicação do conteúdo do arquivo e de seus atributos sobre a função de hash *SHA-1*. Os atributos são constituídos dos seguintes dados: número do *inode*, número do *device*, identificador do usuário, identificador do grupo, do tamanho do arquivo em bytes, valor da matriz de acesso e valores de data e hora de criação e última modificação.

Com a chave pública do sistema e da assinatura digital deste arquivo é aplicado o algoritmo de verificação, a partir do qual obtém-se o valor original do hash do arquivo quando foi assinado. Caso este valor seja igual ao valor do hash obtido do arquivo agora então ele não foi modificado e estará íntegro, retornando-se esta informação para o módulo de controle. Caso contrário será retornado a informação de que não está íntegro.

5.3 Roteiro Básico de Funcionamento

Aqui será exposta a seqüência lógica dos eventos que podem ocorrer na arquitetura durante o funcionamento do sistema.

Na figura 5.6 estão representadas as ações que ocorrem quando da abertura de um arquivo, cujos passos são apresentados a seguir:

1. a aplicação solicita a abertura de um arquivo para o sistema, e este o faz através de uma chamada de sistema *open*;

2. esta chamada é capturada pelo módulo de interceptação *open*, §5.2.1;
3. verifica-se o parâmetro é um arquivo ou link; caso contrário a rotina original é ativada (passo 10);
4. consulta ao módulo de regras de exceção, §5.2.6, onde é verificado se o acesso pode ser liberado imediatamente; em caso positivo é chamada a rotina original do núcleo (passo 10);
5. analisa os flags da operação *open*, verificando se estes podem levar a modificação do arquivo; em caso positivo é negado acesso ao arquivo (passo 9);
6. consulta ao módulo *lista de arquivos íntegros*, §5.2.4. Estando o arquivo ainda íntegro, é chamada a rotina original do núcleo (passo 10);
7. caso o arquivo não conste na lista, solicita ao módulo de verificação de integridade, §5.2.7, o teste do arquivo. Este módulo aplicará o algoritmo de assinatura digital para verificar a integridade do arquivo;
8. estando o arquivo íntegro, é adicionada uma nova entrada na lista de arquivos íntegros, §5.2.4, através de seu módulo; na seqüência a rotina original do núcleo é ativada (passo 10). Caso o arquivo não esteja íntegro ou não possua assinatura, o módulo de interceptação negará o acesso ao arquivo (passo 9);
9. retorna para a aplicação o código de erro EACCESS, indicando negação de acesso ao arquivo;
10. a rotina *open* original do núcleo é ativada com os mesmos parâmetros que recebeu na chamada inicialmente interceptada;

Na figura 5.7 estão representadas as ações que ocorrem quando da execução de um arquivo, através da chamada *execve*. A seqüência de passos é assim descrita:

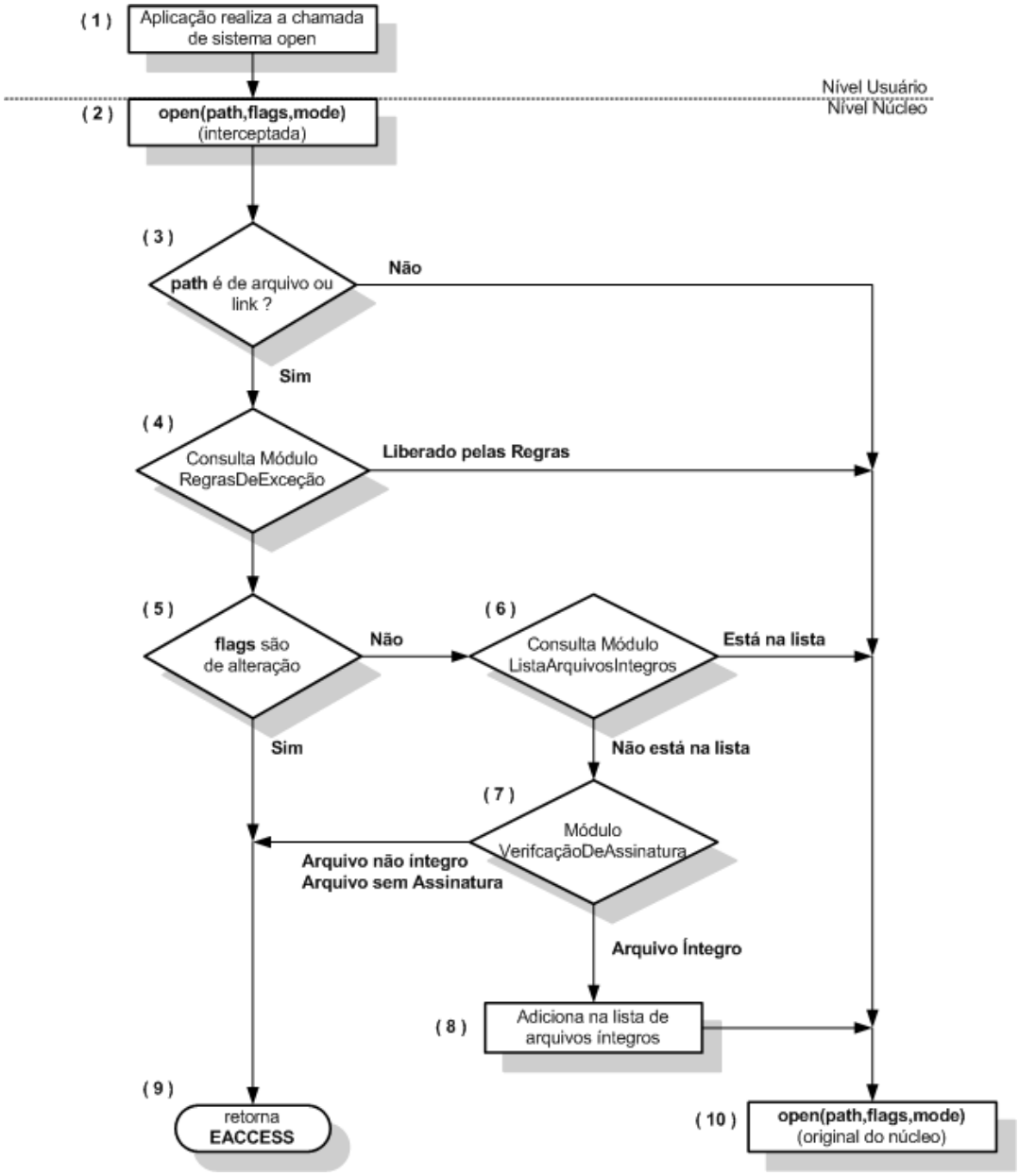

Figura 5.6: Roteiro de funcionamento para abertura de arquivo.

1. quando é solicitada a execução de um arquivo para o sistema, este o faz através de uma chamada de sistema *execve*;

2. esta chamada é capturada pelo módulo de interceptação *execve*, §5.2.2;

3. isolando-se do parâmetro o nome de arquivo, é consultado o módulo de regras de exceção, §5.2.6, para liberação imediata caso o arquivo esteja habilitado por uma regra de exceção (passo 8);

4. consulta à *Lista de Arquivos Íntegros*, §5.2.4;

5. caso o arquivo não esteja na *Lista de Arquivos Íntegros*, solicita-se ao módulo de verificação de integridade, §5.2.7, que por sua vez aplicará o algoritmo de assinatura digital para verificar sua integridade;

6. caso o arquivo esteja íntegro, é adicionada uma entrada para ele na *lista de arquivos íntegros*, §5.2.4;

7. retorna para a aplicação o código de erro EACCESS, indicando negação de acesso ao arquivo, caso o arquivo não esteja íntegro;

8. é chamada a rotina original *execve* do núcleo com o mesmo parâmetro que a chamada interceptada recebeu;

5.4 Conclusão

Neste capítulo foi apresentada a arquitetura proposta, bem como sua descrição funcional. Ela foi implementada em um núcleo *LINUX*, versão 2.4.4 . Como citado anteriormente, o modelo de extensão aqui proposto pode ser enquadrado na categoria de dispositivos virtuais (conforme classificação proposta no capítulo anterior). Para tanto o código do núcleo foi levemente ajustado para suportar a inserção e remoção dos mecanismos propostos durante a execução do sistema. Em versão futuras do núcleo essa alteração

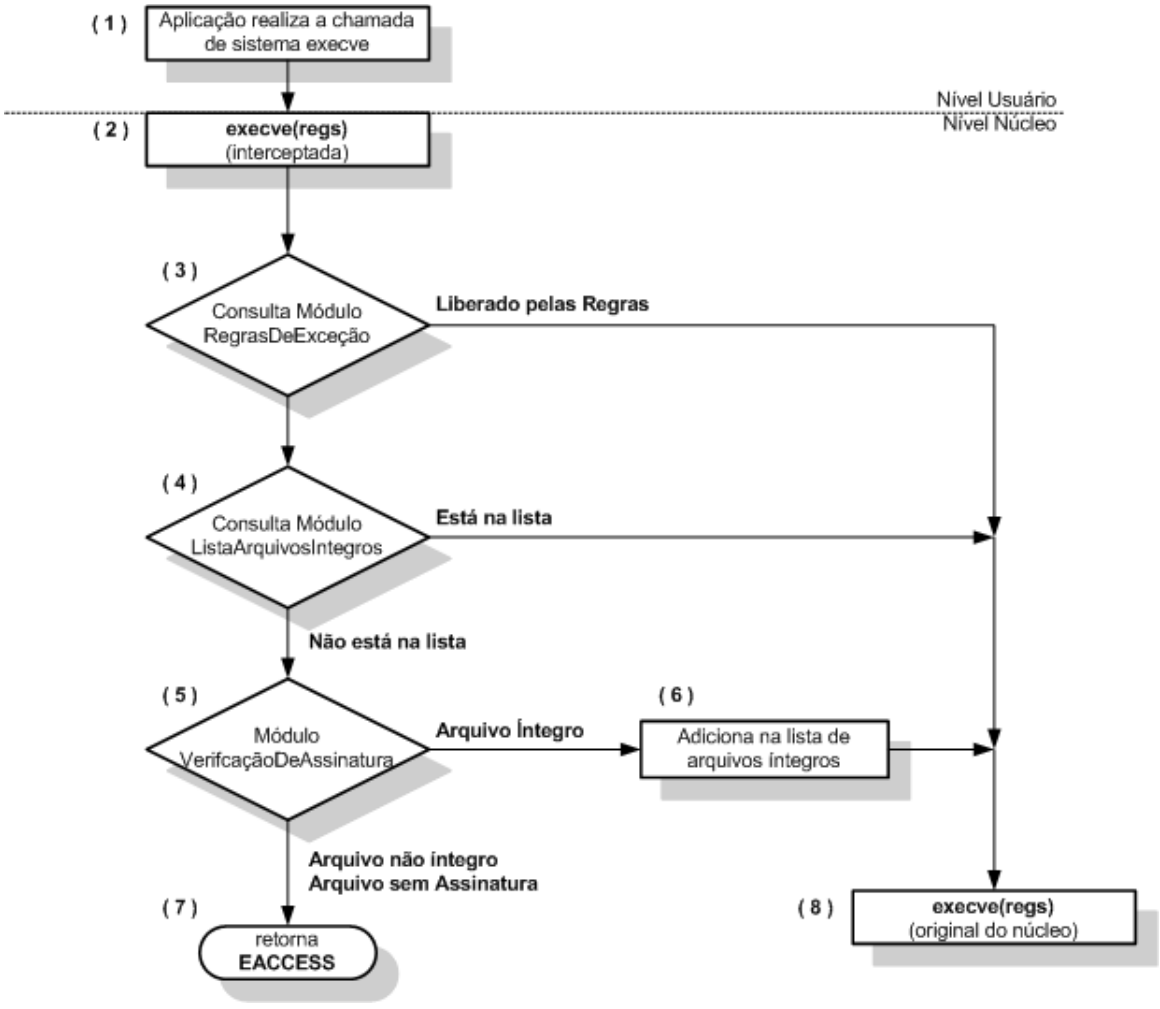

Figura 5.7: Roteiro de funcionamento para execução de arquivo.

não será mais necessária, uma vez que o núcleo irá suportar nativamente alterações dinâmicas através dos *Linux Security Modules* [66]. Para a implementação do protótipo foram gastos três meses de desenvolvimento, consumindo aproximadamente 700 horas de trabalho e gerando um código fonte com cerca de 2 mil linhas em linguagem C.

Capítulo 6

Validação da Arquitetura Proposta

Neste capítulo serão apresentados os ensaios da implementação da arquitetura, na seção 6.3, bem como alguns testes anteriores que ajudaram na sua definição, seção 6.2. O comportamento da arquitetura em situações reais de ataque será abordado na seção 6.4.

6.1 Ambiente de Desenvolvimento e Testes

A necessidade de acesso ao código fonte do núcleo para os estudos e modificações levou à opção pelo núcleo *LINUX* [40]. Foi utilizada a versão 2.4.4[1] do núcleo, que era a versão estável mais recente na época do desenvolvimento do projeto. Este núcleo segue a especificação POSIX 1003.1 [33] e a especificação UNIX 98 [32] do OpenGroup, nos quais são definidos os serviços do sistema e suas respectivas interfaces (chamadas de sistema); estas normas também são seguidas pela grande maioria dos sistemas UNIX.

Outras vantagens em se usar o núcleo *LINUX* incluem sua gratuidade, a grande quantidade de documentação disponível e o acesso a todos os fontes não somente do núcleo mas também dos demais programas que compõem o sistema operacional[2].

[1] Em verificações de versões posteriores, até a 2.4.16, foi constatado que ainda era mantida a compatibilidade.

[2] Excetuando-se alguns poucos programas aplicativos e drivers de dispositivo que somente são fornecidos em sua forma binária.

Todos os ensaios foram realizados em um computador classe IBM-PC, com processador Pentium III de 450MHz, com 128MB de memória RAM e com disco IDE ULTRA DMA33 com tempo de busca de 10ms. Como sistema operacional foi utilizada a distribuição LINUX RedHat[3] 7.1.

Foram realizados os seguintes testes:

- Desempenho das funções HASH;
- Desempenho da arquitetura para operações habituais;
- Comportamento em situações de ataque.

Em cada ensaio foram feitas pelos menos dez amostras de medida, e os resultados apresentados são suas médias aritméticas. Amostras que apresentaram valores díspares das demais foram descartadas.

6.2 Desempenho das funções HASH

A premissa inicial era a utilização do sistema criptográfico que fosse o mais rápido possível e ainda assim garantisse a segurança necessária, mesmo que ele não se enquadrasse em nenhuma norma vigente. Para isto foram realizados vários testes com diversas funções de hashing, tentando localizar a mais rápida para diferentes tamanhos de arquivos.

O primeiro ensaio realizado foi o levantamento do tempo despendido somente com a execução das funções de hash em relação a diferentes tamanhos de arquivos. Foram utilizadas somente as funções MD5[53], SHA-1[47] e HAVAL[4][67], seção 3.2, sendo as duas primeiras padronizadas e utilizadas pelas normas nos processos de assinatura digital[49] e a última por ter a execução mais rápida, segundo a literatura.

[3] `http://www.redhat.com`

[4] A função HAVAL permite a escolha do número de iterações e do tamanho do valor de hash; nos ensaios foram utilizadas 3 iterações e hash de 128 bits.

Arquivo	MD5	SHA-1	HAVAL
1KB	0,0010s	0,0004s	0,0002s
2KB	0,0010s	0,0007s	0,0003s
16KB	0,0037s	0,0051s	0,0020s
64KB	0,0149s	0,0200s	0,0080s
512KB	0,1169s	0,1607s	0,0645s
2048KB	0,4671s	0,6434s	0,2600s
8192KB	1,8709s	2,5717s	1,0316s

Tabela 6.1: Tempos médios de execução das funções de hashing

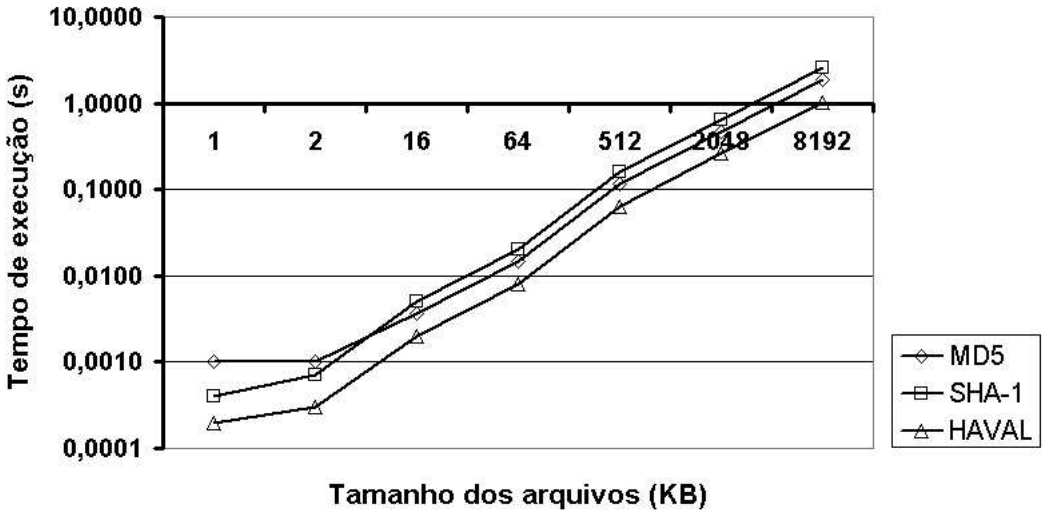

Figura 6.1: Tempos médios de execução das funções de hashing

O ensaio foi realizado com o sistema operacional em modo mono-usuário, objetivando evitar qualquer interferência nos resultados obtidos. Esta situação apresenta os melhores tempos que podem ser obtidos. Para evitar que os caches de disco e de sistema operacional influenciassem nos resultados, para o ensaio de cada função a máquina era re-inicializada.

A tabela 6.1 apresenta os resultados obtidos para as três funções consideradas. Como pode-se observar, os tempos obtidos apresentam as características esperadas: MD5 mais rápida que a SHA-1, e a HAVAL mais rápida que as demais. Entretanto os resultados ainda são muito altos, demonstrando que as funções de hashing não podem ser executadas para cada novo acesso ao arquivo, motivo pelo qual foi introduzida a lista de arquivos íntegros descrita no capítulo 5.

Pode-se constatar, através da figura 6.1, que as funções apresentam um tempo de execução linear com o aumento do tamanho do arquivo. Também observa-se que mesmo a mais rápida função HASH apresenta um tempo de execução muito elevado. Por esta razão optou-se pela utilização de um sistema criptográfico padronizado, o que levou à escolha da função SHA-1 (norma FIPS 186-2 [49]), utilizado pelo algoritmo de assinatura escolhido (ECDSA). A escolha deste algoritmo ocorreu em na fase inicial do projeto, onde não se dispunha de dados concretos comparativos com os demais algoritmos exceto que pela teoria, seção 3.1.2, ela é exige menos recursos de processamento. Posteriormente encontrou-se um quadro comparativo, tabela 3.1.2, onde foi constatado que para a operação de verificação da assinatura, que é o mais impactante e constantemente realizado no A3, é justamente a operação mais demorada no ECDSA. Porém o protótipo já estava concluído e optou-se por mantê-lo assim, visto que no caso de escolha de outro algoritmo com melhores tempos, no caso o RSA, os resultados seriam bem melhores.

Como a proposta propunha o desenvolvimento da arquitetura e não das tecnologias utilizadas, fez-se o uso de uma biblioteca de criptografia para as operações que envolviam a geração e verificação das assinaturas digitais dos arquivos. A biblioteca utilizada é a *MIRACL* [43], que implementa as operações de criptografia em curva elíptica (ECDSA) [38].

Ela teve que ser adaptada para uso no núcleo, uma vez que sua compilação original era como biblioteca de aplicação em modo usuário.

6.3 Desempenho da arquitetura

Embora pudessem ter sido realizados ensaios de maneira semelhante ao efetuado na seção 6.2 para a avaliação do desempenho da arquitetura, optou-se pela escolha de situações de utilização mais próximas da realidade. Para tanto, foram escolhidos três casos típicos da ocorrência de grandes quantidades de acesso a disco e alto uso de processamento.

Os ensaios foram feitos em três situações distintas:

- **execução sem verificação:** ensaio realizado com um núcleo original, sem qualquer modificação, para que os resultados obtidos sejam usados como referência para as outras duas situações.

- **primeira execução com verificação:** ensaio realizado com o núcleo modificado, mas sem que qualquer arquivo do caso em questão fosse acessado antes, ou seja, sem qualquer entrada lista de arquivos íntegros.

- **segunda execução com verificação:** ensaio realizado novamente após o término da primeira execução, onde os arquivos já verificados e não modificados, não são validados novamente por já estarem na lista de arquivos íntegros.

Nas situações "sem verificação" e "primeira execução" os ensaios foram efetuados com o sistema recém inicializado. Para melhorar o entendimento dos ensaios, foram separados os tempos nos níveis usuário, núcleo e total, possibilitando assim avaliar onde ocorreram as principais variações nos tempos. Para facilitar os ensaios, as regras de bloqueio foram relaxadas, ou seja, todos os arquivos são verificados, mas se o arquivo não está íntegro ou não possui entrada na base de assinaturas, mesmo assim o acesso a ele é liberado. Nesta situação ocorre a pior condição de tempo de execução, porque algumas verificações podiam ser abortadas antes de ser necessário o acesso ao disco. Nas sessões seguintes serão apresentados os resultados encontrados para cada caso e situação.

6.3.1 Caso 1: Compilação

O processo de compilação de um programa é extremamente complexo, demorado, utiliza grandes quantidades de memória e processamento e faz uso intensivo de disco. Foi escolhido como programa teste a compilação do próprio núcleo do LINUX, por ser o maior programa disponível para compilação que se tinha à época. Ele é composto por aproximadamente 9600 arquivos, ocupando 135MB em disco.

Tempo	Execução sem verificação	Primeira execução	Segunda execução
Nível Usuário	663,12s	664,80s	662,24s
Nível Núcleo	41,42s	134,70s	90,01s
Total	704,54s	799,50s	752,25s
Custo adicional	—	13,47%	6,77%

Tabela 6.2: Tempos médios para compilação

Operação	Ocorrência	1ª Execução		2ª Execução	
		Hit	Miss	Hit	Miss
open	115494	98,2%	1,8%	99,3%	0,7%
execve	2344	99,2%	0,8%	99,9%	0,1%

Tabela 6.3: Taxas de acerto da lista de arquivos íntegros.

Na tabela 6.2 estão os resultados obtidos para cada situação. Pode-se observar que o tempo no nível usuário permaneceu praticamente constante (663seg) nas três situações, o que não ocorreu com o tempo no núcleo.

Na primeira execução houve um acréscimo no tempo total (13,47%) mas ainda não suficientemente grande para inviabilizar seu uso.

Na segunda execução ainda ocorre algum acréscimo no tempo total (6,77%), mas menor. O principal motivo deste incremento no tempo total é devido à verificação dos códigos-objeto resultantes da compilação, que são novamente verificados por terem sido alterados pela compilação.

Na tabela 6.3 é possível observar a influência da lista de arquivos íntegros nos resultados. Nela estão o número de operações *open* e *execve* que foram efetuadas neste ensaio e os percentuais de acerto (*hit*), ou seja, foi encontrado o arquivo na lista, e de não-acerto (*miss*), quando não foi o arquivo, na lista, tendo sido necessária a verificação de sua integridade.

O número de arquivos abertos é grande porque o mesmo arquivo é aberto varias vezes, como os arquivos de include, o que faz com que o custo adicional da primeira execução seja baixo, 13,47%, porque somente é validado no primeiro acesso.

Tempo	Execução sem verificação	Primeira execução	Segunda execução
Nível Usuário	55,28s	55,75s	55,27s
Nível Núcleo	2,56s	293,33s	3,31s
Total	57,84s	349,08s	58,58s
Custo adicional	—	503,5%	1,28%

Tabela 6.4: Tempos médios para armazenamento

		1ª Execução		2ª Execução	
Operação	Ocorrência	Hit	Miss	Hit	Miss
open	8876	0,14%	99,86%	100%	0%
execve	2	0%	100%	100%	0%

Tabela 6.5: Taxas de acerto da lista de arquivos íntegros.

6.3.2 Caso 2: Armazenamento

O processo de armazenamento de uma estrutura de diretório também faz uso intenso de disco e processamento. A aplicação escolhida para o teste foi o *tar*, com a seguinte linha de comando "*tar -czf arquivo diretório*" que armazena em um *arquivo* comprimido o conteúdo da árvore de *diretório*. Novamente foi utilizado o código fonte do núcleo como diretório para o ensaio, cujas características foram apresentadas na seção anterior.

Na tabela 6.4 estão apresentados os resultados obtidos. Como pode ser observado novamente, na primeira execução ocorre um aumento significativo no tempo de execução, mas na segunda execução o incremento de tempo é insignificante.

Este caso é muito interessante pois, pode-se induzir grosseiramente qual seria o acréscimo de tempo esperado caso não se utilizasse a lista de arquivos íntegros, seção 5.2.4, pois todos os arquivos acessados são verificados. Para a arquitetura de hardware e software do ensaio o incremento de tempo seria da ordem de 500%, ou seja, quase 6 vezes mais lento que o normal, o que é totalmente inviável.

Também pode-se induzir que os 1,28% de acréscimo na segunda execução é a perda de desempenho imposta por toda a arquitetura, que pode certamente ser desprezada.

Na tabela 6.5 é possível observar a influência da lista de arquivos íntegros nos resultados. Nela estão o número de operações *open* e *execve* que foram efetuadas neste

Tempo	Execução sem verificação	Primeira execução	Segunda execução
Nível Usuário	20,56s	20,83s	20,59s
Nível Núcleo	1,52s	11,31s	1,57s
Total	22,08s	32,14s	22,16s
Custo adicional	—	45,56%	0,36%

Tabela 6.6: Tempos médios para execução

Operação	Ocorrência	1ª Execução		2ª Execução	
		Hit	Miss	Hit	Miss
open	117	39,3%	60,7%	100%	0%
execve	2	0%	100%	100%	0%

Tabela 6.7: Taxas de acerto da lista de arquivos íntegros.

ensaio e os percentuais de acerto (*hit*), ou seja, foi encontrado o arquivo na lista, e de não-acerto (*miss*), quando não foi o arquivo, na lista, tendo sido necessária a verificação de sua integridade.

6.3.3 Caso 3: Execução

Este caso é um exemplo real de utilização de um aplicativo em uma estação de trabalho, o aplicativo *ghostview*[5]. Neste ensaio fez-se a execução do aplicativo, abertura do arquivo PostScript, visualização de todas as páginas do documento e término de execução. Embora o aplicativo dê a impressão de ser simples, suas operações são bastante complexas, envolvendo a carga de bibliotecas, fontes de caracteres e a renderização do documento.

Os resultados obtidos estão apresentados na tabela 6.6, que possuem características semelhantes as obtidas nos casos anteriores.

Na tabela 6.7 é possível observar a influência da lista de arquivos íntegros nos resultados. Nela estão o número de operações *open* e *execve* que foram efetuadas neste ensaio e os percentuais de acerto (*hit*), ou seja, foi encontrado o arquivo na lista, e de não-acerto (*miss*), quando não foi o arquivo, na lista, tendo sido necessária a verificação

[5]Um visualizador de arquivos PostScript.

de sua integridade.

6.4 Comportamento em ataques

Para os testes de verificação de funcionamento a ataques reais, foi utilizado o *worm* RAMEN [13, 11, 12], que explora a vulnerabilidade de três serviços de rede em sistemas Linux, o *LPRng*[6], o *wu-ftpd*[7] e o *rpc.rstatd*[8]. Para realizar os testes, foram instaladas versões destes serviços que apresentavam as vulnerabilidades. O vírus, bem como toda a análise de seu funcionamento, foi obtido no site *Whitehats* [63].

Foram utilizadas duas máquinas isoladas, com versões vulneráveis destes serviços. Em uma das máquinas foi aberto e executado o arquivo *ramen.tgz*[9] contendo o vírus, contaminando desta forma esta máquina. Para testar a eficácia do vírus, a segunda máquina foi conectada em rede com a primeira, sendo também contaminada. Na seqüência, a segunda máquina foi novamente instalada e preparada com os serviços vulneráveis.

Para o teste da arquitetura proposta, foram geradas as assinaturas de todos os arquivos do sistema, arquivos binários e de configurações. As regras de bloqueio foram novamente colocadas no modo padrão[10]. Esta segunda máquina foi novamente colocada em rede com a primeira, e como era esperado, sequer os comandos do "instalador" do vírus foram executados com sucesso.

A seguir será feita uma análise do processo de instalação deste vírus, e suas possíveis variações e suas interações com o sistema A3. Os três serviços vulneráveis são executados na conta do administrador (root); quando são exploradas suas vulnerabilidades (*buffer overflow*, seção 2.2), é possível passar comandos para a máquina atacada, com direitos administrativos, que no caso do RAMEN, são os seguintes:

[6] Serviço de impressão de rede

[7] Serviço de FTP.

[8] Um dos módulos do serviço de RPC.

[9] Formato de arquivo comprimido, similar em funcionalidade aos ZIP, RAR, BZ2 e ARJ.

[10] Leitura e gravação permitidas nos diretórios `/var` e `/tmp`, mas execução bloqueada. Demais diretórios liberados para leitura e execução somente de arquivos assinados.

1. `mkdir /usr/src/.poop`
2. `cd /usr/src/.poop`
3. `export TERM=vt100`
4. `lynx -source http://IPdaOrigem:27374 > /usr/src/.poop/ramen.tgz`
5. `cp ramen.tgz /tmp`
6. `cd /tmp`
7. `gzip -d ramen.tgz`
8. `tar -xvf ramen.tar`
9. `./start.sh`

O primeiro comando falharia, pois não é possível a criação de qualquer arquivo ou diretório na sub-árvore `/usr`, por não haver regra de exclusão liberando esta ação. O segundo também falharia, pois o diretório não foi criado. O terceiro seria executado com sucesso. O quarto e quinto falhariam, pois o diretório `.poop` não foi criado, portanto o arquivo não foi baixado da máquina de origem, portanto não existe o arquivo `ramen.tgz` para ser copiado. O sexto seria feito com sucesso. O sétimo, oitavo e nono falhariam pois os arquivos não existem para serem processados.

Pode-se supor uma variação, no qual a transferência fosse para o diretório `/tmp`, que é liberado pelas regras para leitura e escrita, ao invés do diretório `/usr/src/.poop`. Todos os comandos seriam executados com sucesso, com exceção do comando `./start.sh`. Este não seria executado por estar bloqueado pelas regras e não possuir assinatura.

Mas supondo-se que a execução de arquivos no `/tmp` estivesse liberada pelas regras de exceção, os diversos programas que compõem o worm RAMEN, tentariam alterar/criar/apagar os seguintes arquivos:

1. `/etc/rc.d/rc.sysinit`
2. `/etc/inetd.conf`
3. `/sbin/asp`

4. `/etc/xinetd.d/asp`
5. `/etc/hosts.deny`

A modificação do primeiro e segundo arquivos seriam bloqueadas por estarem modificando um arquivo assinado. O terceiro e quarto arquivos seriam criados, mas não poderiam ser executados/acessados por não possuírem assinatura associada. O quinto arquivo seria realmente removido.

Pode-se constatar que mesmo em uma situação muito ruim, sendo permitida a execução do instalador do vírus com direitos administrativos, o sistema não perdeu sua integridade, apenas ficou mais aberto, devido a remoção do arquivo `/etc/hosts.deny`. Desta forma sua eficiência a ataques reais pode ser comprovada.

6.5 Conclusão

Neste capítulo foram vistos os resultados alcançados com a atual implementação da arquitetura, onde foi possível observar sua viabilidade e segurança alcançada. O protótipo construído permitiu a validação da proposta, embora ainda necessite de ajustes e melhorias para tornar-se um produto apto à distribuição ao público.

Capítulo 7

Conclusões e Perspectivas

Este capítulo apresenta algumas das dificuldades encontradas, avalia os resultados alcançados, apresenta alguns projetos correlatos à proposta e sugere algumas possibilidades de continuação deste trabalho.

7.1 Resultados

A verificação ativa da integridade dos arquivos mostrou-se eficaz no bloqueio aos tipos conhecidos de ataques que fazem uso de modificações no sistema. A proteção oferecida pela arquitetura proposta coíbe adulterações nos arquivos do sistema, seja através da ação de algum vírus, seja através execução de algum aplicativo malicioso pelo administrador, seja por algum usuário mal intencionado.

O bloqueio funciona em duas frentes: uma evitando que um arquivo monitorado[1] seja modificado de forma indevida, e a outra, assumindo que o se o sistema foi modificado de alguma forma, evitar que os arquivos adulterados sejam acessados.

Desta forma obtemos um sistema com algumas características de *survivability* como referenciado em vários artigos pelo *CERT-CC*[2]. Primeiramente ele é capaz de resistir ao ataque, mas uma vez sucumbindo as partes afetadas são isoladas e ficam inacessíveis, po-

[1] Arquivo que possua uma assinatura digital.
[2] Na internet: `http://www.cert.org/research/`.

rém as partes ainda íntegras permanecem operacionais e prestando os seus serviços. Isto é muito interessante em máquinas que disponibilizam vários serviços simultaneamente, que mesmo tendo comprometidos alguns de seus serviços é ainda capaz de prestar serviços corretamente.

O impacto esperado sobre o desempenho deve ser muito baixo quando esta arquitetura for utilizada em servidores correntes, como: e-mail, páginas WEB, arquivos, impressão e de aplicativos. Por raramente sofrerem modificações em seus arquivos binários ou de configuração, permanecendo por meses em contínua operação, raramente serão necessárias novas verificações de integridade de seus arquivos.

Em servidores de desenvolvimento a situação é um pouco diferente, pois sempre será necessária a assinatura e verificação do novo aplicativo que o usuário criou, ou então liberar a execução em alguns diretórios através das regras de exceção. Isto trará um certo impacto sobre a rotina do usuário desenvolvedor, pois antes de executar um arquivo o mesmo terá de ser assinado. Este procedimento deve ser tão simples quanto assinar digitalmente um e-mail antes de enviá-lo, porém esta abordagem ainda não é aplicável no atual estágio de desenvolvimento da arquitetura.

Sendo assim, a arquitetura proposta é bastante viável para ser utilizada na prática, em sistemas reais. Todavia, por se tratar de uma arquitetura em desenvolvimento, ainda não está pronta, restando diversos pequenos detalhes a serem resolvidos para torná-lo um produto de prateleira, conforme apresentado mais a frente na seção 7.4.

Como sub-produtos deste trabalho pode-se incluir ainda a geração de dois artigos [8, 7]. Novas publicações estão em preparação.

7.2 Considerações acerca da implementação

Durante o desenvolvimento constatou-se que alguns detalhes que no início pareciam ser de fácil localização ou solução, na realidade não o eram. Primeiramente, embora tenha-se o acesso total aos códigos fontes do núcleo, este não é comentado e o desenvol-

vimento intra-núcleo é bastante obscuro. No primeiro semestre de 2001 a documentação do núcleo utilizado era praticamente inexistente, restringindo-se a uma breve e superficial descrição de algumas poucas partes do núcleo e em sua maioria disponibilizada de forma online e para versões muito antigas. Basicamente, a cada três dias de desenvolvimento da arquitetura, dois foram gastos com problemas decorrentes da falta de documentação do núcleo ou de detalhes sobre como codificar corretamente para ele. Ironicamente, todas as informações necessárias que não existiam nas documentações oficiais foram encontradas em uma fonte "não confiável", um site hacker. Em [51] estavam todas as instruções didaticamente expostas, passo-a-passo, em como criar-se um vírus de núcleo *LINUX*, que foram fundamentais no sucesso da implementação e no entendimento do núcleo.

Em segundo lugar, a inexistência de ferramentas de depuração para o núcleo atrapalhou muito a localização dos erros de codificação. A única forma de depuração encontrada e utilizada foi através do uso de instruções que indicavam por onde a linha de execução passou. No núcleo existe uma função usada no sistema de logs, com sintaxe e funcionalidade semelhante a função *printf*, chamada *printk*. Esta maneira de depuração mostrou-se útil na maioria das situações em que a codificação estava mais estável, apenas com pequenos erros de lógica. Porém, em erros mais sérios de codificação, o núcleo se paralisava sem indicar onde ou porque, ou apresentava uma tela de *kernel panic* que paralisa e mostra o estado dos registradores do processador. Nestes casos foi necessário localizar e corrigir o erro sem se saber exatamente o que corrigir.

Existe uma versão de núcleo que é instanciado em nível usuário, o *User Mode Linux* [57], onde ele é executado como um simples processo e não como núcleo real, portanto passível de depuração pelos ambientes existentes. Entretanto as modificações que seu desenvolvedor realizou para que o núcleo pudesse ser instanciado como processo, modificavam justamente as chamadas de sistema utilizadas pela arquitetura proposta e portanto não foi possível utilizá-lo no desenvolvimento.

7.3 Trabalhos Correlatos

Foram localizados apenas três projetos destinados exclusivamente à verificação de integridade dos arquivos de um sistema. Existem outras formas de realizar a verificação a partir de programas separados que podem ser agrupados, como por exemplo: com o uso de um programa que gere o MD5 (hashing) de um arquivo e scripts é possível construir uma ferramenta que realize a verificação de integridade em modo *batch*.

- **BSIGN:** Criado em 1998 por Oscar Levi com o objetivo de detectar falhas de *hardware* em disco IDE que corrompiam os programas. Utilizando uma característica existente no formato de arquivo ELF[3] para executáveis de sistemas UNIX, foi acrescentada uma seção com a assinatura digital em cada arquivo executável do sistema. A verificação da integridade é realizada através de um processo *batch* executado em um horário programado, normalmente na madrugada, recebendo o administrador o relatório por email. Sua maior vantagem é sua simplicidade e também sua maior desvantagem. Somente arquivos executáveis que tenham o formato ELF podem ser assinados, todos os demais ficam desprotegidos, como outros formatos de executáveis, arquivos de configuração e dados. Os atributos dos arquivos assinados também não verificados, como *time-stamp* e permissões. Não foi localizada qualquer publicação a seu respeito. Ele pode ser encontrado somente em forma de download no site relacionado em [9], mas a última atualização data de maio/2000. A pouca documentação que o acompanha resume-se a quatro páginas, que abrangem uma descrição do projeto, instruções para instalação e operação, explicitamente direcionadas para o sistema operacional GNU/Linux.

- **TRIPWIRE:** Com motivações semelhantes as que motivaram esta dissertação, o TripWire foi desenvolvido no *COAST Laboratory* da *Purdue University* no início

[3]Este formato de arquivo permite que sejam criadas várias seções dentro do mesmo arquivo. Por exemplo, uma seção pode conter código executável, outra dados, outra código executável e neste caso também a assinatura digital do programa.

da década de 90. Em [41] é apresentada a descrição completa do projeto, abrangendo desde os ensaios das funções de hashing até os modelos de administração. Nestes anos ele foi sendo atualizado[4], representando hoje o estado-da-arte em verificação de integridade de arquivos de sistemas, mas operando em modo *batch*. O seu uso é altamente recomendado pelo CERT [14] e demais entidades de que tratam de segurança [36]. Diferentemente do Bsign, aqui todos os arquivos do sistema são verificados. É montada uma base de dados, que compreende além do *path* do arquivo, todos os seus atributos e o Message Digest (hashing). O arquivo que contém a base de dados inteira é assinada digitalmente para proteção contra adulterações, mas não cada arquivo do sistema. Em um horário programado, novamente de madrugada, é verificada a integridade de cada arquivo do sistema e gerado um relatório ao final.

- **SOFFIC:** É o projeto mais próximo da proposta do *A3*, com exatamente os mesmos objetivos, ou seja, fazer a verificação de integridade em tempo de execução e está sendo desenvolvido na UFRGS/PPCG [20]. Porém sua implementação e características são muito diferentes do *A3*. É utilizado um arquivo único, contendo os valores de hashing de cada arquivo, os números de *device*, *inode* e um flag indicando o tipo de algoritmo de hashing que foi utilizado na geração, sendo somente este arquivo assinado digitalmente e sua implementação enquadrada na modalidade de extensão *User-Level Plug-in*, seção 4.1.3. Com relação aos testes de funcionamento desconhece-se os resultados já alcançados.

- **Outros sistemas:** Foram encontrados outros sistemas que fazem a proteção das chamadas de sistema. O sistema Remus [6] faz a verificação dos parâmetros passados para as chamadas de sistema e, baseando-se em uma coleção de regras de acesso, libera ou nega o acesso à chamada de sistema. Estas chamadas são agru-

[4]Ele pode ser encontrado no site `http://www.visualcomputing.com/`, onde o projeto é mantido.

padas em níveis conforme seu grau de periculosidade para o sistema, bem como se elas são chamadas através de um terminal do sistema ou se são chamadas através de um *daemon*[5]. Um outro sistema é o Systrace [52], que implementa regras de acesso muito mais detalhadas que as existentes no sistema de autorização padrão do sistema operacional. Estas regras permitem um grande controle e refino sobre o que é liberado ou não para cada aplicativo ou usuário.

7.4 Pesquisas Futuras

Tanto a arquitetura, quanto sua implementação, foram desenvolvidas basicamente para validar a idéia de que a autenticação dos arquivos era viável, oferecendo segurança quanto à ativação somente de arquivos íntegros e que não ocorreriam problemas de desempenho, entretanto ainda restam alguns pontos que podem ser melhorados:

7.4.1 Mídia segura para imagem de Boot

Apesar da arquitetura bloquear a modificação de arquivos assinados, ela não impede que o arquivo seja removido, movido ou substituído, pois estas atribuições são do sistema de autorização do núcleo. Portanto, esta talvez seja a única vulnerabilidade da arquitetura proposta, a troca do arquivo binário do núcleo ou do arquivo que contém a chave pública utilizada para verificação das assinaturas, ou de ambos. É interessante que a inicialização seja realizada somente a partir de fontes seguras, existindo basicamente duas maneiras para isso: a primeira seria o boot a partir de uma mídia que permitisse somente leitura, como um disquete protegido contra escrita ou de um CD-ROM. Desta forma, a troca teria que ser feita fisicamente junto à máquina, o que deixaria de ser um problema de segurança de software para tornar-se um problema de segurança física. Esta solução está longe de ser uma solução elegante, mas é muito funcional e segura.

[5]Programa residente em segundo plano, como por exemplo: syslogd, sshd, httpd, etc.

A outra maneira, esta sim elegante, seria que o próprio hardware (através da *BIOS*) armazenasse ou fosse buscar na rede o arquivo de inicialização. Isto seria uma extensão da *BIOS*, exatamente como ocorre com uma placa de rede que possuí *BOOT-PROM*[6], porém com mecanismos que garantam a integridade do que está sendo executado durante o boot. Infelizmente o único trabalho conhecido na área é o projeto AEGIS[2] que é uma extensão da *BIOS* com todos os procedimentos necessários para se efetuar o boot de maneira segura a partir da rede e desconhece-se sua aplicação em sistemas não acadêmicos. Um outro projeto que pode ser adaptado para este fim é o LinuxBIOS, desenvolvido nos laboratórios do centro de computação avançada de Los Alamos [7] para uso em seus clusters, objetivando reduzir o tempo de inicialização dos nodos. A *BIOS* original da placa mãe é substituída por outra que faz uma inicialização primária dos hardware e carrega uma imagem comprimida do núcleo LINUX. Esta imagem comprimida poderá conter o núcleo da arquitetura proposta. Esta abordagem está sendo mantida comercialmente por uma empresa e já está disponível para alguns modelos de placa mãe, permitindo seu uso imediato.

7.4.2 Integração com o LSM (Linux Security Modules)

Como a versão do núcleo não suportava diretamente a interceptação de algumas partes de suas funções internas, foi criado um pequeno *patch* para isto. Portanto a implementação atual da arquitetura está codificada em um modo proprietário, embora ainda seja um *device driver*(4.1.2), mas o porte para a infra-estrutura aberta do LSM [66], permitirá a fácil integração atual e futura em núcleos LINUX. O LSM somente foi incorporado na versão 2.5 do núcleo, disponibilizada durante o ano de 2002.

[6] Uma pequena *ROM* que contém instruções de como efetuar o boot a partir de uma imagem armazenada em um servidor na rede. Embora funcional, estas instruções possuem mecanismos de autenticação bem simples.

[7] Na internet: `www.acl.lanl.gov/linuxbios/`

7.4.3 Adotar um modelo de regras mais robusto

O módulo de regras foi incorporado para resolver o problema de liberação seletiva de alguns arquivos, mas é muito primitivo ainda, embora funcional. Um interessante complemento à arquitetura seria o projeto SysTrace [52], que implementa uma ACL por aplicativo, permitindo que se defina exatamente o que cada um deve e pode acessar, podendo desta forma centralizar as regras, com um maior poder de expressão, controle e segurança, pois os projetos são complementares.

7.4.4 Adotar uma PKI para as assinaturas

O processo de assinatura do arquivo, do local de armazenamento destas assinaturas e de uma infra-estrutura de PKI estão todas relacionados. Com a adoção de uma PKI é possível que os arquivos sejam assinados por várias entidades diferentes: o administrador da máquina, o administrador do sistema de informação, os usuários e os fornecedores dos pacotes de software que são instalados. Este último já ocorre parcialmente, pois alguns fornecedores assinam seus pacotes através do uso do PGP [8]. Entretanto, dependendo da forma como a PKI seja implementada, será mais um serviço a ser atacado e mais um ponto de entrada no sistema. A pesquisa deve concentrar-se mais nos aspectos humanos do que computacionais. Como foi observado através dos anos, os usuários deixam de fazer determinadas atividades se não forem obrigados, portanto a melhor solução será aquela que um usuário comum possa realizar sem problemas ou dúvidas e da forma mais natural possível, que não seja necessariamente a melhor solução técnica.

7.4.5 Substituição da biblioteca criptográfica

A biblioteca em uso no protótipo da arquitetura possui restrições quanto ao seu uso comercial, sendo necessário o licenciamento para tal. Além disso, como os tempos de execução para a primeira execução são relativamente altos. Uma avaliação do impacto

[8] Na internet: `http://www.gnupg.org`

da troca do padrão de assinatura utilizada por outras, ou seja, trocar a *ECDSA* pela *DSA* ou *RSA* padrão, permitiria abandonar a biblioteca *MIRACL* pelas de código aberto, como a *OPENSSL* [9], podendo-se disponibilizar a arquitetura para a comunidade sem que haja problemas de licenciamento das mesmas e talvez pouco afete o desempenho da arquitetura.

7.4.6 Localizações alternativas da base de assinaturas

A centralização da base de assinaturas em um único arquivo não permite a adoção de uma PKI, pois teria-se que permitir que usuários normais pudessem alterá-la. Além disso, imediatamente após a inicialização, o núcleo começa a ler os arquivos de configuração da máquina, seguido da carga de diversos drivers de dispositivo e bibliotecas. Logo, a assinatura destes deve estar disponível localmente na máquina, pois a rede somente torna-se ativa mais tarde, pouco antes de ser passado o controle para o usuário. É possível também armazenar remotamente estas assinaturas, mas a cópia local é indispensável para o processo de inicialização. Outra consideração a respeito das assinaturas é que elas devem ser necessariamente geradas na máquina onde os arquivos residem, isto porque os dados acerca da localização física do arquivo na máquina, bem como seus demais atributos, são parte dos dados de integridade verificados.

[9] Na internet: `http://www.openssl.org`

Referências Bibliográficas

[1] Albert D. Alexandrov, Maximilian Ibel, Klaus E. Schauser, and Chris J. Scheiman. Extending the operating system at the user level: the UFO Global File System. In *Proceedings of the 1997 USENIX Technical Conference*, Anaheim, USA, January 1997.

[2] William A. Arbaugh, David J. Farber, and Jonathan M. Smith. A secure and reliable bootstrap architecture. In *Proceedings 1997 IEEE Symposium on Security and Privacy*, pages 65–71. IEEE, May 1997.

[3] Shahram Bakhtiari, Reihaneh Safavi-Naini, and Josef Pieprzyk. Keyed hash functions. In *Cryptography: Policy and Algorithms*, pages 201–214, 1995.

[4] Amnon Barak and Oren La'adam. The MOSIX Multicomputer Operating System for High Performance Cluster Computing. Journal of Future Generation Computer Systems, March 1998.

[5] Steven M. Bellovin. Security problems in the tcp/ip protocol suite. *Computer Communications Review*, 2(19):32–48, April 1989.

[6] Massimo Bernaschi, Emanuele Gabrielli, and Luigi V. Mancini. REMUS: A security-enhanced operating system. In *ACM Transactions on Information and Security*, volume 5, pages 36–61, February 2002.

[7] Mauro Augusto Borchardt and Carlos Alberto Maziero. Uma arquitetura para a autenticação dinâmica de arquivos. In *Simpósio Segurança em Informática*, pages 101–108, São José dos Campos, 2001.

[8] Mauro Augusto Borchardt and Carlos Alberto Maziero. Verificação da integridade de arquivos pelo kernel do sistema operacional. In *Workshop em Segurança de Sistemas Computacionais*, pages 31–36, Florianópolis, 2001.

[9] Projeto Bsign. ftp://ftp.buici.com/pub/bsign, February 2001.

[10] CERT Coordination Center. Steps for Recovering from a UNIX or NT System Compromise. Na internet `http://www.cert.org/tech_tips/root_compromise.html`, April 2000.

[11] CERT Coordination Center. CERT Advisory CA-2000-17 Input Validation Problem in rpc.statd. Na internet `http://www.cert.org/advisories/CA-2000-17.html`, August 2000.

[12] CERT Coordination Center. CERT Advisory CA-2000-22 Input Validation Problems in LPRng. Na internet `http://www.cert.org/advisories/CA-2000-22.html`, December 2000.

[13] CERT Coordination Center. Cert advisory ca-2000-13 two input validation problems in ftpd. Na internet `http://www.cert.org/advisories/CA-2000-13.html`, July 2000.

[14] CERT Coordination Center. http://www.cert.org/, 2001.

[15] CERT Coordination Center. CERT Advisory CA-2001-09 Statistical Weaknesses in TCP/IP Initial Sequence Numbers. Na internet `http://www.cert.org/advisories/CA-2001-09.html`, May 2001.

[16] CERT Coordination Center. CERT/CC Statistics 1988-2002. Na internet `http://www.cert.org/stats/cert_stats.html`, July 2002.

[17] D. Coppersmith, D. B. Johnson, and S. M. Matyas. A proposed mode for triple-des encryption. *IBM Journal of Research and Development*, 40(2):253–262, 1996.

[18] Microsoft Corporation. *Win32 Programmer's Reference*. Microsoft Corporation, 1993.

[19] SAINT Corporation. Saint. Na internet `http://www.wwdsi.com/`, August 2002.

[20] Vinícius da Silveira Serafim and Raul Fernando Weber. Um verificador seguro de integridade de arquivos. In *Simpósio Segurança em Informática, CTA/ITA/IEC*, October 2001.

[21] J. Daemen and V. Rijmen. AES Proposal: Rijndael. First Advanced Encryption Standard (AES) Conference, California, August 1998.

[22] Joan Daemen, René Govaerts, and Joos Vandewalle. Weak keys for IDEA. *Lecture Notes in Computer Science*, 1994.

[23] Renaud Deraison. The "nessus" project. Na internet `http://www.nessus.org/`, August 2002.

[24] Remote OS detection via TCP/IP Stack FingerPrinting. http://www.insecure.org/nmap/nmap-fingerprinting-article.html, February 2001.

[25] W. Diffie and M.E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):650, November 1976.

[26] Marcelo Abdalla do Reis and Paulo Lício de Geus. Forense computacional: Procedimentos e padrões. In *Simpósio Segurança em Informática, CTA/ITA/IEC*, October 2001.

[27] Hans Dobbertin, Antoon Bosselaers, and Bart Preneel. RIPEMD-160: A strengthened version of RIPEMD. In *Fast Software Encryption*, pages 71–82, 1996.

[28] Info Exame. Oracle pediu, hacker invadiu. Na internet `http://www2.uol.com.br/info/aberto/infonews/012002/18012002-4.shl`, January 2002.

[29] D. Farmer and E. H. Spafford. The cops security checker system. In *Proceedings of the Summer 1990 USENIX Conference*, pages 165–170, June 1990.

[30] Douglas P. Ghormley, David Petrou, Steven H. Rodrigues, and Thomas E. Anderson. SLIC: An extensibility system for commodity operating systems. In *Proceedings of the USENIX 1998 Annual Technical Conference*, pages 15–19, Berkeley, USA, June 1998. USENIX Association.

[31] CIS Network Group. Tiger. Na internet `http://net.tamu.edu/network/tools/tiger.html`, July 2002.

[32] The Open Group. *The Single UNIX Specification, Version 2*. The Open Group, 1998.

[33] IEEE. *IEEE P1003.1: POSIX - Standard Portable Operating Systems Interface for Computer Environments*. IEEE, 1988.

[34] IEEE. *IEEE P1003.1e: POSIX - Security Interface Extensions*. IEEE, 1988.

[35] IEEE. *Standard Specifications for Public-Key Cryptography*. IEEE Computer Society, 2000.

[36] SANS Institute. http://www.sans.org/, February 2001.

[37] D.B. Johnson. ECC: Future resiliency and high security systems. *Certicom*, 1999.

[38] Don Johnson and Alfred Menezes. The elliptic curve digital signature algorithm ECDSA. Technical Report CORR 99-34, University of Waterloo, Canada, February 2000.

[39] Michael B. Jones. Interposition agents: Transparently interposing user code at the system interface. In *Symposium on Operating Systems Principles*, pages 80–93, 1993.

[40] Linux Kernel. http://www.kernel.org/, 2001.

[41] G. H. Kim and E. H. Spafford. The design and implementation of Tripwire: A file system integrity checker. Technical Report CSD–TR–93–071, Purdue University, November 1993.

[42] Steve R. Kleiman. Vnodes: An architecture for multiple file system types in SUN UNIX. In *USENIX Summer*, pages 238–247, 1986.

[43] Shamus Software Ltd. Miracl library version 3.2x. Na internet `http://indigo.ie/~mscott/`, April 2001.

[44] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[45] Nelson Murilo and Klaus Steding-Jessen. Métodos para detecção local de rootkits e módulos de kernel maliciosos em sistemas UNIX. In *Simpósio Segurança em Informática, CTA/ITA/IEC*, October 2001.

[46] Tim Newsham. Format string attacks, September 2000.

[47] National Institute of Standards and Technology. *FIPS 180-1 – Secure Hash Standard*. Federal Information Processing Standards Publication, April 1995.

[48] National Institute of Standards and Technology. *FIPS 46-3 – Data Encryption Standard (DES)*. Federal Information Processing Standards Publication, October 1999.

[49] National Institute of Standards and Technology. *FIPS 186-2 – Digital Signature Standard*. Federal Information Processing Standards Publication, January 2000.

[50] National Institute of Standards and Technology. *FIPS 187 – Advanced Encryption Standard*. Federal Information Processing Standards Publication, November 2001.

[51] Pragmatic/THC. Complete linux loadable kernel modules. Na internet `http://www.thehackerschoice.com/papers/LKM_HACKING.html`, July 2002.

[52] Niels Provos. Systrace - interactive policy generation for system calls. Na internet `http://www.citi.umich.edu/u/provos/systrace/`, July 2002.

[53] R. Rivest. The MD5 Message-Digest Algorithm. IETF RFC 1321, April 1992.

[54] Ron Rivest, Matt Robshaw, Ray Sidney, and Yiqun Lisa Yin. The Security of the RC6 Block Cipher. Na internet `http://www.rsasecurity.com/rsalabs/rc6/`, 1998.

[55] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[56] R.Sekar and O. Uppuluri. Synthesizing fast intrusion prevention/detection system from high-level specifications. In *Proceedings 8th Usenix Security Symposium*, Washington DC, USA, August 1999. USENIX Association.

[57] Rusty Russell. User-mode linux. Na internet `http://user-mode-linux.sourceforge.net/`, 2001.

[58] Vagner Sacramento, Thais Batista, Guido Lemos, and Claudio Monteiro. Uma estratégia de defesa contra ataques de DNS Spoofing. In *Simpósio Segurança em Informática, CTA/ITA/IEC*, October 2001.

[59] NGS Software. Hackproofing Oracle Application ServerNGS Software. Na internet `http://www.nextgenss.com/papers/hpoas.pdf`, February 2002.

[60] Johny Srouji, Paul Schuster, Maury Bach, and Yulik Kuzmin. A transparent checkpoint facility on NT. In *Proceedings of the 2nd USENIX Windows NT Symposium*, Seattle, Washington, USA, August 1988. USENIX Association.

[61] LIDS Linux Intrusion Detection System. http://www.lids.org/, February 2001.

[62] Tripwire. Na internet `http://www.tripwire.com`, April 2002.

[63] Max Vision. Ramen internet worm analysis. *Na internet `http://www.whitehats.com/library/worms/ramen/index.html`*, 2001.

[64] Kenneth M. Walker, Daniel F. Sterne, M. Lee Badger, Michael J. Petkac, David L. Shermann, and Karen A. Oostendorp. Confining root programs with domain and type enforcement. In *Proceedings of the 6th USENIX UNIX Security Symposium*, San Jose, California, USA, July 1996. USENIX Association.

[65] Erik De Win, Serge Mister, Bart Preneel, and Michael Wiener. On the performance of signature schemes based on elliptic curves. *Lecture Notes in Computer Science*, 1423:252, 1998.

[66] Chris Wright, Crispin Cowan, James Morris, Stephen Smalley, and Greg Kroah-Hartman. Linux security modules: General security support for the Linux Kernel. In *USENIX Security Symposium*, 2002.

[67] Zheng, Pieprzyk, and Seberry. HAVAL – A one-way hashing algorithm with variable length of output. In *AUSCRYPT: Advances in Cryptology–AUSCRYPT '90, International Conference on Cryptology*. LNCS, Springer-Verlag, 1992.

[68] Zone-h. Statistics. Na internet `http://www.zone-h.org/en/stats`, August, 2002.