Verificação da integridade de arquivos pelo kernel do sistema operacional

Mauro A. Borchardt, Carlos A. Maziero

Janeiro, 2001

Programa de Pós-Graduação em Informática Aplicada
Pontifícia Universidade Católica do Paraná, 80.215-901, Curitiba, PR
E-mail:{borchardt,maziero}@ppgia.pucpr.br

Resumo

Os sistemas de informação atuais estão sujeitos a intrusões. 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 alterações, mas normalmente só conseguem fazê-lo de forma tardia, depois que o sistema foi comprometido. Apresentamos uma nova abordagem, levando para o kernel do sistema operacional a responsabilidade de verificar a integridade de todo arquivo antes de ser executado ou aberto. A técnica proposta aqui não destina-se a evitar que o sistema seja invadido ou adulterado, mas sim a detectar e inibir que as modificações realizadas comprometam a segurança do sistema. São enumeradas a dificuldades técnicas e decisões de projeto envolvidas na implementação da técnica proposta.

PALAVRAS-CHAVE: assinatura digital, executável, kernel, segurança, integridade.

1 Introdução

Um dos principais problemas dos sistemas de informação atuais é a possibilidade de intrusão. Burlando os mecanismos de segurança, invasores podem conseguir acesso indevido a recursos e informações. Quando o objetivo imediato não é a destruição pura e simples do sistema ou de seus dados, o invasor usa artifícios para dissimular sua presença e manter seu acesso futuro ao sistema. Uma das formas usuais de garantir esse acesso consiste em alterar parte do sistema operacional local, sejam arquivos de sistema, bibliotecas ou até mesmo partes do kernel, para esconder a presença do intruso e criar portas escondidas (*backdoors*) que permitam seu retorno.

Muitas técnicas foram propostas no sentido de detectar alterações nos arquivos de um sistema e facilitar sua reparação. Uma dessas técnicas baseia-se no cálculo de um *message digest*[1] para cada arquivo do sistema. Os *message digests* do sistema em seu estado inicial (após instalação) são armazenados em local seguro. Caso haja suspeita de violação do sistema, os *message digests* são recalculados e comparados com seus valores iniciais. Um produto conhecido que usa essa técnica é o *Tripwire* [6]. Um problema com essa técnica é que, devido à sua característica de execução em modo *batch*, a detecção de uma intrusão pode ser muito tardia.

[1] Resultado da aplicação de um algoritmo sobre um arquivo de tamanho variável, que tem a propriedade "teórica"de não gerar o mesmo valor para arquivos diferentes.

Neste artigo apresentamos uma nova abordagem para essa técnica, levando para o kernel do sistema operacional a responsabilidade de verificar a integridade de todo código executável ou arquivo aberto. Ao lançar um novo processo, carregar bibliotecas dinâmicas ou abrir um arquivo, o kernel efetua a verificação necessária, aprovando ou bloqueando sua execução ou acesso e tomando as medidas necessárias em caso de problemas. A seção 2 traz alguns conceitos importantes para a compreensão do problema, a seção 3 apresenta a proposta, a seção 4 enumera problemas a resolver para sua implementação e explora algumas possibilidades de solução. Finalmente, a seção 5 descreve as possibilidades de continuação deste trabalho.

2 Fundamentos

Os modelos de proteção baseados em níveis de privilégio [1, 2, 3] ou em confinamento de processos [4, 5, 8] têm como um de seus objetivos evitar que o sistema operacional seja modificado por ações inadvertidas ou intencionais. A proposta aqui apresentada visa detectar modificações antes que estas possam ter qualquer efeito sobre o sistema, evitando que partes do sistema comprometidas por um invasor possam ser ativadas. O restante deste artigo se apoia sobre a arquitetura dos sistemas UNIX, embora possa ser aplicado sem dificuldade a outras plataformas.

A complexidade interna dos sistemas atuais leva à presença de vulnerabilidades, muitas vezes extremamente sutis, que permitem níveis de acesso privilegiados. Tais vulnerabilidades, associadas a pacotes de intrusão sofisticados, como os *rootkits*[2], tornam a gestão da segurança uma tarefa árdua e minuciosa, exigindo atenção constante dos responsáveis.

Por diversas razões, a grande maioria das aplicações de sistema são de propriedade do administrador do sistema (ou *root*). No entanto, elas geralmente são executadas no domínio de proteção de cada usuário. Portanto, a aplicação não poderá acessar nada que o usuário não possa, o que confere maior segurança ao sistema contra falhas de implementação.

Todavia, algumas aplicações precisam ter acesso a recursos normalmente inacessíveis aos usuários normais. Este é o caso por exemplo das aplicações de sistema que efetuam spooling de impressão (`lpr`), agendamento de tarefas (`at`, `cron`) ou envio de correio (`mail`). Assim, essas aplicações precisam operar em um domínio de proteção mais privilegiado que aquele do usuário convencional. O sistema UNIX resolve esse problema através do mecanismo de *setuid bit*, que permite a determinadas aplicações executar sob o domínio de proteção de seu proprietário (geralmente o *root*) ao invés do domínio do usuário. Tais aplicações têm acesso privilegiado aos recursos do sistema e devem devem ser portanto ser muito bem escritas e testadas, para evitar que sejam usadas com objetivos diferentes daqueles inicialmente previstos. Boa parte das falhas de segurança de sistemas UNIX envolve programas com o bit *setuid* ativo, e portanto estes são alvos freqüentes de ataques.

3 A técnica proposta

A técnica aqui proposta é composta de duas partes igualmente importantes, sendo uma estática e outra dinâmica. A primeira parte, estática, é formada basicamente por uma lista de arquivos a monitorar, enquanto a segunda é constituída pelos mecanismos para a monitoração dos arquivos dessa lista. A lista de monitoração deve ser construída a partir do sistema em um estado inicial íntegro, ou seja, com o sistema operacional e demais aplicativos sendo instalados a partir de fontes confiáveis, portanto sem a presença

[2] Um *rootkit* é uma coleção de programas que permitem ao invasor criar portas de entrada ocultas, obter informações do sistema e comprometê-lo através substituição de diversos programas, como `ls`, `ps` e `netstat`, por versões alteradas, visando esconder sua presença, de seus arquivos, processos e conexões de rede.

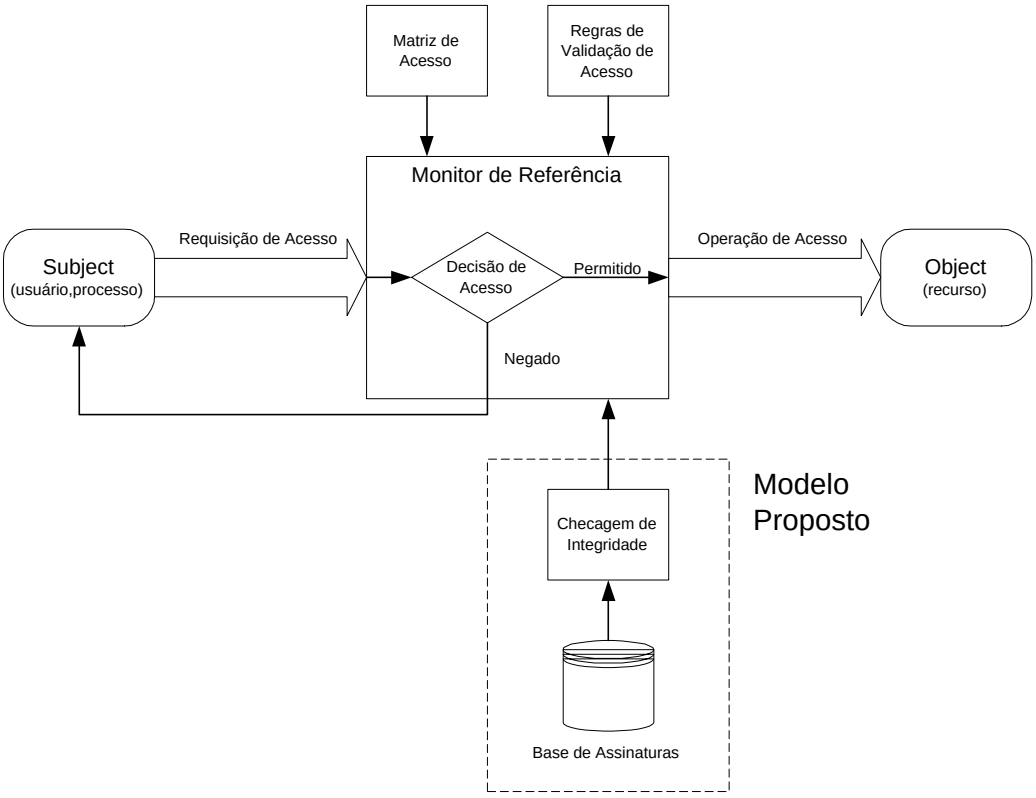

Figura 1: Representação do Monitor de Referência com incorporação do Modelo Proposto

de código intruso como *backdoors* e *trojans*[3]. Sobre esse estado inicial confiável é construída uma *base de assinaturas*, formada pelos nomes de caminho dos arquivos associados a informações que permitam a detecção de violações. A princípio, somente recursos estáticos, residentes no sistema de arquivos, devem ser cadastrados: arquivos executáveis, bibliotecas e arquivos de configuração.

Uma das maneiras de se representar um modelo de controle de acesso é através de um monitor de referência [7]. Independentemente do modelo utilizado, o núcleo do monitor não se altera, seja para um modelo discricionário, obrigatório, baseado em regras ou outro. A apresentação da proposta será baseada no modelo discricionário típico de um sistema UNIX. A figura 1 ilustra como a técnica proposta é incorporada ao monitor, sem alterações significativas do modelo básico.

São necessárias pequenas alterações em um kernel existente para a implantação do modelo proposto. Para sistemas operacionais respeitando o padrão POSIX, seriam duas as alterações previstas. A primeira é a interceptação da chamada ao sistema `execve`, responsável pela execução de novos processos. A segunda é a interceptação da chamada ao sistema `open`, responsável pela abertura de arquivos pelos processos.

A interceptação consiste na interposição de uma rotina entre a interface externa do kernel e suas chamadas de funções internas. Essa rotina tem por finalidade verificar a integridade do arquivo que pretende-se abrir ou executar. Se a integridade for confirmada então a função original é chamada . Caso contrário, registra-se uma falha na integridade e devolve-se um erro ao processo, negando-lhe assim o acesso.

Para se realizar uma verificação de integridade poderíamos comparar o arquivo atual com uma cópia

[3]Um aparentemente útil e inocente programa que contém código adicional escondido que permite acesso não autorizado. Este código pode ter sido implantado propositalmente pelo desenvolvedor ou através da modificação de um programa legítimo por um invasor.

Assinatura Digital	Path	Mat.A.	UID	GID	Timestamp
`2b79e60d8bfac03c66739c9ed0...`	/etc/passwd	rw-r--r--	root	root	21/12/00 17:34:32:3123
`7e07ddb8c823d8f04e5ad127da...`	/etc/shadow	r--------	root	root	21/12/00 17:34:32:3123
`63e6bfaf5d2e4c451c2b684d84...`	/etc/hosts	rw-r--r--	root	root	13/10/00 13:32:23:1256
...					

Tabela 1: Exemplo da base de assinaturas

íntegra previamente armazenada. Como essa situação é claramente inviável, podemos fazer uso de um *message digest* do arquivo em questão. Além disso, podemos também armazenar outros atributos do arquivo, como direitos de acesso, proprietário, grupo e datas. Para evitar que o invasor possa modificar o *message digest* de um arquivo que ele tenha adulterado é utilizada uma assinatura digital. Com o uso de algoritmos de chave assimétrica [7], cada *message digest* é criptografado com a chave privada definida para o sistema, formando uma assinatura digital. Com isso forma-se uma *base de assinaturas* dos arquivos, composta por tuplas formadas pela assinatura e pelos atributos. Somente a chave pública e esta base de assinaturas são utilizadas pela rotina incorporada ao kernel para verificação de integridade. A tabela 1 mostra um exemplo desta base.

A rotina de verificação de integridade no kernel é a mesma tanto para a chamada `open` como para `execve`, com a diferença que esta última também verifica as bibliotecas que o programa vai utilizar. Quando uma destas chamadas é invocada por um processo, a rotina localiza na base de assinaturas a assinatura que corresponde ao arquivo em questão e a descriptografa com a chave pública do sistema, obtendo-se o *message digest* do arquivo íntegro. Simultaneamente calcula-se o *message digest* do arquivo atual. Ao comparar os dois resultados, bem como os atributos, chega-se a conclusão sobre a integridade do arquivo.

4 Problemas do modelo

Muito embora a técnica proposta seja simples do ponto de vista conceitual, existem diversos problemas a resolver para que possa ser aplicada de forma viável. Os principais problemas levantados são:

4.1 Localização da Chave Pública

Se a chave pública puder ser trocada pelo invasor, ele poderá recriar a base de assinaturas com seu par de chaves e o sistema estará comprometido. A situação ideal seria que ela se localizasse em uma mídia que permitisse somente leitura, como um *cdrom*, um *floppy disk* protegido ou um *hardlock*, porém nem sempre será possível utilizá-los.

É possível compilar estaticamente a chave no código do kernel, tornando-a inacessível a qualquer código em modo usuário, ou seja, aos processos. Em alguns kernels monolíticos, como é o caso da maior parte das versões do UNIX, é possível a carga dinâmica de módulos ou drivers de dispositivo. Tal módulo teria acesso a todo o espaço de endereçamento do kernel e um módulo comprometido teria assim acesso total ao sistema. Neste caso chamadas ao sistema responsáveis pela carga desses módulos também precisam ser modificadas para verificação da integridade dos mesmos antes de carregá-los no kernel.

Também é possível manter a chave em um arquivo normal, porém mantendo-o escondido através de modificações nas chamadas ao sistema de arquivos (ou seja, usando a mesma técnica explorada pelos *rootkits*). Desta forma, esse arquivo não seria visível para nenhum processo do sistema. Essa abordagem

é sujeita aos mesmos problemas da alternativa anterior, mas tem a vantagem de não ser necessária a recompilação do kernel e a reinicialização do sistema em uma eventual necessidade de troca da chave.

Uma outra abordagem seria a busca da chave pública, e possivelmente da base de assinaturas, via rede em um servidor. Neste caso, grande parte do sistema já deveria estar ativo para que fosse possível o acesso à rede, e portanto algum código violado já poderia ter sido executado.

4.2 Localização da base de assinaturas

Essa estrutura é menos sensível que a chave pública com relação à integridade, mas é muito importante para a operação do sistema. Se ela for adulterada, a arquivo associado à entrada alterada ficará inacessível. Se ela for apagada, o sistema ficará inoperante. Portanto, sua proteção é essencial. Cabem aqui as mesma observações e ressalvas feitas com relação à chave pública, porém seria a ocultação pelas chamadas do sistema de arquivos a mais indicada, devido as dimensões provavelmente grandes desta base para possibilitar outra forma de armazenamento.

4.3 Arquivos não assinados e em diretórios de usuários

A técnica proposta trata apenas dos arquivos assinados, não prevendo nenhuma política com relação aos demais. Duas alternativas podem ser adotadas: uma seria permitir o acesso normal, conforme o modelo proposto, e outra seria mais restritiva, permitindo o acesso sob determinadas condições. Uma destas condições poderia exigir que os arquivos que podem sofrer modificações sejam cadastrados com um *flag* que indicasse que é modificável, como os arquivos de log do sistema. Outra seria o cadastro de chaves públicas também para os usuários, sendo o usuário obrigado a cadastrar seus arquivos em uma base de assinaturas, o que é especialmente útil para executáveis criados pelo usuário. Neste caso a busca da chave pública e da base do usuário pela rede seria viável.

4.4 Desempenho

Os algoritmos de *hash* e criptografia são computacionalmente pesados e podem levar a problemas de desempenho. Uma maneira de compensar este problema seria com o uso de um cache para armazenar os resultados anteriores, principalmente a descriptografia das assinaturas. Variações entre diferentes algoritmos de hash e tamanhos de chaves criptográficas podem alterar em muito o desempenho total do sistema.

4.5 Administração

Um sistema de informação está em constante evolução. Instalações de novos softwares ou atualizações são necessidades freqüentes, o que pode representar um problema para a manutenção da base de assinaturas.

Uma solução interessante seria o provimento, por parte do fornecedor do software, da assinatura de todos os arquivos relevantes do pacote, bem como de sua chave pública. Essa chave pública pode então ser estocada em uma base de chaves para os softwares instalados. O sistema deve buscar nessa base a chave pública a ser usada para verificar a integridade dos arquivos de um determinado fornecedor.

Para evitar que um intruso tente instalar ou atualizar software contendo código malicioso, a autenticidade da chave pública do fornecedor pode ser confirmada através do uso de um esquema de certificados,

semelhante ao usado nas páginas Web seguras. Isso exige no entanto que o sistema tenha acesso a chaves públicas de entidades certificadoras externas.

5 Conclusão

Neste artigo apresentamos uma técnica para melhorar a segurança de um sistema computacional através da certificação de arquivos, sejam eles executáveis ou não. Essa técnica não exclui a possibilidade de violação de um sistema, mas impede que programas alterados pelo intruso sejam executados. A abordagem proposta faz uso de técnicas de criptografia de chaves assimétricas e resumos digitais (*message digests*). Embora seja usado como contexto de trabalho um sistema UNIX típico, a proposta pode ser facilmente adequada a outros sistemas operacionais.

Ao longo do texto foram enumerados os principais problemas a ser resolvidos para a implementação da proposta, bem como soluções plausíveis a cada um deles. Como este é um trabalho em andamento, certamente outros problemas, mais sutis, serão encontrados no decorrer de sua implementação. Todavia, acredita-se que foram encontradas respostas adequadas às questões conceituais fundamentais para a implementação dessa proposta.

Referências

[1] D. E. Bell and L. La Padula. Secure computer system: Unified exposition and multics interpretation. Technical Report ESD-TR-75-306, Electronics Systems Division/AFSC, Bedford, MA, 1975.

[2] K. J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MS, 1977.

[3] D. Ferraiolo and R. Kuhn. Role-based access control. In *Proceedings of the 15th National Computer Secutity Conference*, USA, 1992.

[4] Timothy Fraser, Lee Badger, and Mark Feldman. Hardening cots software with generic software wrappers. In *IEEE Symposium on Security and Privacy*, Berkeley, USA, May 1999.

[5] D. P. Ghormley, D. Petrou, S. H. Rodrigues, and T. 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.

[6] 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.

[7] R. Summers. *Secure Computing: Threats and Safeguards*, chapter 4 and 5. McGraw-Hill, New York, NY, 1997.

[8] K. M. Walker, D. F. Sterne, M. L. Badger, M. J. Petkac, D. L. Sherman, and K. A. Oostendorp. Confining root programs with domain and type enforcement (dte). In *Sixth USENIX UNIX Security Symposium*. USENIX Association, 1996.