Proteção de dados sensíveis através do isolamento de processos arbitrários no kernel Linux

Otávio Augusto Silva[1], **André Grégio**[1,2], **Paulo Lício de Geus**[1]

[1] Instituto de Computação – Universidade Estadual de Campinas (Unicamp)
Av. Albert Einstein, 1251 – 13083-852 – Campinas – SP – Brasil

[2]Centro de Tecnologia da Informação Renato Archer (CTI/MCTI)
Rod. D. Pedro I (SP-65), KM 143,6 – 13069-901 – Campinas – SP – Brasil

{otavios, gregio, paulo}@lasca.ic.unicamp.br

Abstract. *Linux management policies raise issues such as security decisions being delegated to object owners. Existing frameworks aim to remediate these issues by restricting access to kernel objects (e.g., files) or replacing the kernel code for a more secure version. Although they may prevent a broad set of attacks, they do not mitigate privacy leaks, specially those that abuse the operating system trust base. In this article, we propose a mechanism to ensure that predefined rules are applied on sensitive processes for which the OS trust level is not enough, and the breaking of rules does not raise Mandatory Access Control conflict warnings. Our deployed module hooks the system call table, uses the LSM framework and extends other approaches (AppArmor and Gresecurity).*

Resumo. *As políticas de gerenciamento do Linux podem causar problemas, como decisões de segurança delegadas aos proprietários de um objeto. Mecanismos existentes tentam remedia-los via restrição de acesso a objetos de kernel e substituição deste por um mais seguro. Embora evitem alguns ataques, tais mecanismos não mitigam violações de privacidade, especialmente as que abusam a base de confiança do SO. Neste artigo, propõe-se um mecanismo que aplique regras pré-definidas em processos sensíveis cujo nível de confiança no SO não é suficiente e a quebra das regras não gere alertas de Mandatory Access Control. O módulo proposto intercepta a tabela de chamadas de sistema, usa o framework LSM e estende outras abordagens (AppArmor e Gresecurity).*

1. Introdução

A crescente modernização e monetização dos atacantes traz a necessidade de que todo *software* seja projetado com conceitos de segurança em mente. Assim, um ambiente computacional—doméstico, industrial, governamental ou corporativo—é tão seguro quanto seu *software* mais vulnerável. Como consequência da sofisticação dos ataques, os investimentos em segurança computacional têm crescido ano a ano. Relatórios como [Morgan 2015] preveem um salto no investimento de US$ 77 bilhões em 2015 para cerca de US$170 bilhões em 2020 no mercado norte-americano.

Parte do *software* altamente utilizado possui, em alguma etapa do desenvolvimento, a busca e resolução de falhas de segurança. Além disso, faz-se uso de mecanismos externos (e.g., *security frameworks*) para fornecer um certo nível de integridade e

determinismo durante a execução dos programas de forma a prover um ambiente mais seguro contra ataques. No entanto, mesmo esses ambientes preparados para resistir contra tentativas de violação à segurança podem ser comprometidos por vazamento de informações privilegiadas, brechas na interação baseada na confiança entre os processos do sistema operacional (como compartilhamento de memória) ou problemas em permissões no sistema de arquivos envolvendo processos de um mesmo usuário.

As violações da confidencialidade dos dados relacionados a um processo ocorrem devido as permissões de acesso a um objeto serem partilhadas entre processos de um mesmo grupo, ou entre aqueles que têm o mesmo nível de privilégio, como processos criados pelo mesmo usuário. Esse tipo de violação não é, em geral, causado por *bugs* existentes no sistema operacional, mas como consequência da otimização de recursos presentes nos sistemas operacionais modernos.

O projeto de sistemas operacionais modernos, como o *Linux*, visa muito mais do que fornecer uma interface para que o usuário interaja com o *hardware*. O objetivo é prover uma interface para diversos usuários simultaneamente, com mecanismos e estruturas para o compartilhamento transparente dos recursos limitados do sistema. Essa abstração na concorrência leva diversos processos (de vários usuários) a manterem instâncias de um mesmo recurso, por exemplo um arquivo em disco, em seus contextos de uso. Mediante a grande necessidade de otimização, minimização do uso e alocação de recursos no sistema operacional, tais contextos por diversas vezes são implicitamente compartilhados entre processos com os mesmos privilégios, principalmente aqueles de um mesmo usuário.

O exemplo mais simples de que a confiança atribuída por processos de um mesmo nível de privilégios pode ser abusada e levar ao vazamento de informações é o sistema de arquivos, haja vista que processos de um mesmo usuário podem, no mínimo, acessar os mesmos arquivos e assim abusarem da confiança dada pelo sistema operacional. Entretanto, a segurança provida a um processo pelo *kernel* ou por um *security framework* é passível de ser comprometida pela leitura das entradas pertencentes a processos de um mesmo usuário no *procfs* [Nguyen 2004].

O *procfs* é referido como um pseudo-sistema de arquivos que contém informação sobre processos e sobre o *Linux kernel*. Ele não contém arquivos em armazenamento secundário, mas arquivos virtuais referentes às informações do sistema em tempo de execução (gerenciamento de memória, dispositivos montados, etc.). Ele também contém entradas referentes a cada processo em execução, onde é possível encontrar informações como memória mapeada (incluindo a posição das bibliotecas compartilhadas), arquivos abertos, dispositivos montados pelo processo, entre outras. De posse dessas informações, um processo poderia descobrir quais áreas de memória um outro processo de um mesmo usuário está utilizando, sendo capaz de reduzir, ou mesmo anular, a complexidade de ataques contra a ASLR[1] de um determinado processo.

Neste artigo foram estudados os principais mecanismos de segurança capazes de proteger processos contra falhas de *software* ou de privacidade, seja no nível de usuário ou de *kernel*. Esse estudo revelou a existência de limitações quanto ao isolamento entre processos, onde a violação à privacidade não é tida como ataque pelos mecanismos de segurança avaliados. As principais contribuições do presente trabalho são:

[1] *Address Space Layout Randomization* [Hund et al. 2013].

- Comparar os principais mecanismos de segurança quanto à aplicação, modelo de proteção e limitações.
- Apresentar a proposta e o desenvolvimento de um mecanismo de isolamento e segurança de processos que sane parte das limitações encontradas no estudo comparativo, tendo a privacidade dos dados de um processo como principal requisito.
- Introduzir um estudo de caso do mecanismo proposto em conjunto com o *Grsecurity* para proteção do ambiente em um cenário de ataques não previamente conhecidos pelo sistema.

Este artigo está dividido da seguinte forma: na Seção 2 são apresentados os *security frameworks* existentes na indústria e na academia utilizados para tratar de parte dos problemas de segurança aos quais um processo está sujeito; na Seção 3, são introduzidos o mecanismo proposto, sua arquitetura e aspectos da abordagem usada para aumentar a privacidade dos dados em um ambiente de execução; na Seção 4 são mostrados os testes feitos para comparar algumas abordagens existentes e o mecanismo proposto, bem como os resultados obtidos relacionados a efetividade e sobrecarga; na Seção 5, são feitas as considerações finais.

2. Trabalhos Relacionados

O *kernel Linux* fornece uma *API* para segurança—o *Linux Security Modules Framework*(*LSM* [Morris 2013a])—que permite a monitoração e controle do acesso aos objetos do *kernel* ou dos processos. Esses objetos podem variar de arquivos no disco até *kernel capabilities*, como a capacidade de interagir com *sockets*. Através desse *framework* é possível registrar um módulo no *kernel*, a fim de supervisionar ações em outros subsistemas (e.g., sistema de arquivos) e assim ser capaz de aplicar regras de segurança.

Exemplos de *security frameworks* que utilizam o *LSM* são o *SELinux* [Morris 2013b] e *AppArmor*[AppArmor 2013]. Esses dois sistemas criam, em conjunto com o *LSM*, um controle de acesso por restrição (*MAC*[2]) capaz de limitar o que um processo pode fazer, seja com objetos em nível de usuário ou do *kernel*. Com isso, incrementa-se o gerenciamento do acesso e as permissões já existentes no Linux, ao atribuir aos processos *labels* ou *tags* relacionadas às permissões concedidas.

O *SELinux* segue o modelo do menor privilégio. Em um cenário de práticas rigorosas, tudo é negado por padrão e, somente após a escrita de uma série de exceções nas políticas, libera-se o estritamente necessário para um processo funcionar. Tais políticas são escritas especificamente para cada executável cujos processos deseja-se controlar. Os executáveis são identificados pelos seus respectivos *inodes*, e as políticas aplicadas em dois regimes, *Enforcing* e *Permissive*. No modo *Enforcing*, o *SELinux* impede toda ação que infringe uma política de segurança e a reporta como um evento a ser logado; já no modo *Permissive*, os eventos são apenas reportados. O conjunto de regras e políticas possíveis é muito vasto, sendo considerado extremamente complexo quando comparado a outros sistemas que fazem uso do *LSM* e cumprem o mesmo objetivo, porém com regras mais simples e diretas.

O *AppArmor*, por sua vez, também faz uso do *LSM*, mas possui regras mais simples. Apesar de não ser o único, é o mais utilizado deles. Outros *security frameworks* que

[2]Mandatory Access Control

foram integrados ao *kernel vanilla* são o Smack [Schaufler 2011], que propõe um sistema de MAC simplista para ambientes embarcados com poucos recursos de *hardware*, e o Tomoyo [Corporation 2015], que possui uma aplicação balanceada tanto em auditoria e métricas do sistema, quanto em segurança. O *AppArmor* possui dois modos de operação, *Complain* e *Enforce*, onde o primeiro apenas registra violações às regras, enquanto o outro rejeita o acesso aos objetos controlados. Diferente do *SELinux*, o *AppArmor* utiliza *profiles* para gerenciar as regras de cada processo. Esses *profiles* são válidos para quaisquer processos cujo executável corresponda a um caminho específico. É nesses *profiles* que estão contidas todas as políticas de acesso, objetos controlados e mecanismo de acesso disponível para o executável em questão.

Apesar das diferenças na configuração, aplicação dos *labels* ou identificação dos objetos e processos protegidos, os *frameworks SELinux* e *AppArmor*, como usam o *LSM*, são equivalentes em termos de segurança fornecida. [Chen et al. 2009] demonstram a equivalência na segurança ao comparar os mecanismos utilizados por ambos os sistemas, porém com variações no *overhead* resultante de cada sistema. [SUSE 2016] apresenta uma tabela comparativa das principais *features* de ambos os sistemas.

O *Grsecurity* [Spengler 2015c] é um sistema de segurança cuja abordagem difere da usada por sistemas baseados em *LSM*, pois faz seus próprios *hooks* e pontos de entrada no *kernel* e, apesar de também fornecer um sistema equivalente ao *MAC*, sua proteção vai além: como o *Grsecurity* possui um conjunto de *patches* do *kernel Linux*, ele faz mudanças de segurança no código do *kernel* em quase todos subsistemas, aumentando a proteção nos níveis de usuário e de *kernel*. Com essa abordagem, o *Grsecurity* protege todo o sistema contra *exploits*, ataques inerentes ao funcionamento do *kernel Linux* (e.g.,jitspray [McAllister 2012]) e manipulação dos dados de um processo por outro. Os principais tipos de proteção alcançados por esse sistema encontram-se descritas em [Spengler 2015b], enquanto que [Fox et al. 2009a] comparam a proteção provida entre *Grsecurity* e *SELinux*.

Para determinar o comportamento esperado de um processo e a decisão a ser tomada em caso de anomalia, o *Grsecurity* aplica aprendizado de máquina com regras baseadas em *RBAC* (*Role-Based Access Control*). O *PaX*, responsável pela gerência de segurança do processos em nível de usuário, possui dois modos de operação: aprendizado e produção. No modo aprendizado, o sistema é mantido fora de um ambiente "não-confiável" (produção), no qual todas as operações realizadas pelos processos são ditas confiáveis e farão parte da base de conhecimento.

A matriz de comparação presente em [Spengler 2015a] torna possível visualizar as diferenças entre *Grsecurity*, *SELinux* e *AppArmor*. Por exemplo, *Grsecurity* pode proteger um processo tal como qualquer sistema baseado em *LSM*, mas como protege o *kernel*, acaba por proteger também o subsistema do próprio *LSM*.

[Onarlioglu et al. 2013] propõem um sistema—*PrivExec*—que utiliza-se de mecanismos como restrições por IPC (*Inter Process Communication*) e *eCryptfs* [Kirkland 2012] para proteger um processo e seus dados. O *PrivExec* divide os processos entre os protegidos e os outros, onde a diferenciação ocorre na existência de uma *flag* adicional na *task_struct* e em uma chave simétrica, utilizada no *eCryptfs* naqueles que são protegidos. Esse sistema fornece o isolamento dos dados persistentes

através da separação da visão do *root filesystem*, com uso do *OverlayFS* [Brown 2015], e de uma versão alterada do *eCryptfs* para criptografar os arquivos do processo. Essa abordagem estende-se também ás páginas de *swap* de um processo, porém o *PrivExec* não protege diretamente nenhuma outra página de memória, com exclusão àquelas utilizadas em IPC. O isolamento por *IPC* ocorre através de restrições aos grupos de processos que podem iniciar essa comunicação. Um processo pertence a um grupo caso ele possua a chave simétrica compartilhada no grupo. Essa chave é compartilhada pelo primeiro processo a integrar o grupo, o qual usualmente é o processo a ser protegido, e cujas dependências foram incluídas neste grupo. A política implementada permite que processos do mesmo grupo se comuniquem livremente, enquanto outros processos recebem um erro de permissão, na leitura ou na escrita, quando feitas em outros grupos privados. As estruturas do *kernel Linux* que foram modificadas para respeitar essa política foram: *UNIX SysV & POSIX shared memory, message queues* e *UNIX domain sockets*.

O *PrivExec* consegue o isolamento de um processo arbitrário, garantindo o sigilo dos arquivos manipulados tanto no disco quanto no *swap* e da comunicação via *IPC*. Entretanto, essa implementação apenas provê segurança nos protocolos de *IPC* mais comuns, assim como não garante qualquer isolamento contra leitura ou vazamento de memória perante outros processos. As limitações do *PrivExec* incluem o fato de ser uma proposta conceitual na qual não se considera resistência em caso de ataques, falhas do sistema ou presença de *malware* previamente à sua instalação.

2.1. Limitações dos trabalhos relacionados

Os *security frameworks* apresentados representam o estado da arte na proteção de processos, seja contra ataques por meio de falhas de *software*, ou contra ataques que mudam o comportamento esperado do *software* ao longo de sua execução. Embora tais *frameworks* possam ser bem-sucedidos na prevenção da exploração de muitas vulnerabilidades, existem classes inteiras delas que não podem ser evitadas, como *kernel capabilities* desnecessárias a um processo ou *race conditions*.

Isto ocorre devido parcialmente à limitação inata de todos os sistemas de controle de acesso, como descrito por [Ausanka-Crues 2006]: eles precisam ser tão automáticos quanto possível e possuir configurações legíveis e facilmente editáveis, além de um rigoroso e estrito controle de objetos, de modo a fazer cumprir as políticas de segurança ao eliminar erros do administrador. Desta forma, ocasionalmente uma vulnerabilidade precisa ser tratada pela elaboração de novas regras, em geral porque novos objetos ou *capabilities* tornaram-se alvo dessa vulnerabilidade e as antigas regras não foram suficientes. Essa limitação está presente no *Grsecurity*, conforme mostrado por [Bugliesi et al. 2012] ao fazerem uma profunda análise, incluindo limitações e *trade-off*, da gerência por *RBAC* em relação a *MAC*. Em [Fox et al. 2009b], corrobora-se tais limitações ao se contrastar a implementação do *SELinux* e do *Grsecurity* quanto à gerência de acesso.

Cabe ressaltar que os *frameworks* acima mencionados não tratam dos recursos dos processos controlados por eles em relação a outros processos do sistema, caso esses últimos também não sejam controlados. Ambos, *LSM* e *Grsecurity*, controlam o que um processo controlado pode acessar, porém sem interferir no que um processo com os mesmos privilégios do processo controlado pode acessar, caso esse acesso não viole nenhuma política de segurança. Esse modo de projeto é um problema mais de privacidade

do que de segurança: caso um processo controlado seja subvertido, ele não será capaz de fazer mais do que ele é permitido pelas políticas de segurança. Porém, é possível um outro processo obter dados daquele processo controlado sem de fato influenciar em sua execução, isto é, sem causar correspondência com nenhuma regra de segurança, dado que não se trata de um cenário identificado como ataque pelos *frameworks*.

Um bom exemplo é o servidor Web *Apache*, o qual possui um único arquivo de configuração de políticas nos *frameworks* supracitados, mas geralmente é executado como mais de uma instância. Cada instância cuida geralmente de vários *virtual hosts* e é um processo de um mesmo usuário. Neste cenário, uma instância pode interferir com dados de outras instâncias, pois para os sistemas de segurança, todos as instâncias tem o mesmo nível de acesso e privilégios. Isso acontece devido à forma como cada processo é controlado com base no executável que deu-lhe origem, através do *path* ou *inode* no sistema de arquivos. Logo, uma instância do *Apache* pode sobrescrever arquivos de outra instância, tais como *logs*, *Unix socket lockfiles*, configurações, ou mesmo nos arquivos hospedados.

Embora o *Grsecurity*, diferente do *LSM*, não possua apenas um gerente de acesso para aplicar suas políticas e possa tratar cada processo como uma instância diferente no sistema operacional, ele não irá negar o acesso de uma instância à outra. Isso acontece porque o *Grsecurity* não pode coincidir esse comportamento com um perfil de ataque, fazendo com que apenas verifique os direitos de acesso dos recursos que o processo está autorizado a utilizar. Assim, ele também não consegue garantir a privacidade dos dados de um processo quando o acesso a estes é inerentemente permitido para outro processo. Este cenário sempre acontece quando o projeto do sistema operacional espera uma base de confiança na relação entre processos, tais como a relação pai-filho ou processos do mesmo usuário.

O mecanismo proposto neste artigo sanou algumas das limitações citadas ao retirar a proteção por meio de gerências de acesso e por perfil de ataques, passando-a para a gerência da posse dos dados. Por exemplo, de acordo com a gerência de permissões de um objeto, como um arquivo ou um *socket*, este só pode ser acessado caso o processo consiga decifra-lo, e não de acordo com seus privilégios no sistema operacional.

3. Arquitetura do Mecanismo Proposto

Neste artigo, propõe-se um mecanismo de isolamento de processos arbitrários a partir do contexto de outros processos, para assim aumentar a segurança e privacidade mesmo no caso de processos que compartilham os mesmos privilégios. A proposta é complementar ao tipo de proteção provida por outros *frameworks*, como o *Grsecurity*, pois resolve algumas de suas limitações, principalmente quando a base de confiança do sistema operacional não é suficiente, como mostrado anteriormente.

O mecanismo de segurança aqui proposto, chamado a partir de agora de OSPI (*Operating system Security Process Isolator*), foi desenvolvido como um módulo do *kernel Linux* para arquitetura x86 com suporte à extensão *AES-NI* e um gestor de nível de usuário. Sua arquitetura permite que os processos utilizem características especiais de segurança introduzidas no *kernel* pelo módulo, o qual tem por objetivo segregar os processos em grupos, de modo a aplicar políticas especiais de segurança—cada objeto manipulado por um processo, em nível de usuário ou não, é exclusivamente de sua propriedade ou compartilhado entre um grupo predefinido de processos. Para isso, o OSPI faz uso prin-

cipalmente dos seguintes recursos do *kernel*: *syscall hooking*, *LSM framework*, *CGroups (control groups)* e *CryptoAPI*. Além das estruturas do *kernel* citadas, o OSPI também inclui as seguintes estruturas lógicas: uma lista completa de processos controlados, uma lista global de estruturas do *kernel* controladas, uma lista de símbolos do *kernel* por processo e máscaras de bits por processo que mapeiam as permissões e políticas de cada processo.

A utilização das referidas estruturas permite que o módulo controle o acesso e as permissões de cada processo ou grupo de processos cujas permissões são compartilhadas. Esse controle é realizado nas estruturas cujo comportamento padrão deva ser alterado seguindo políticas de segurança, como a *syscall* `process_vm_readv`, que deve retornar sucesso, porém não deve retornar um bloco de memória quando destinada a um processo protegido, independentemente das permissões do usuário. As técnicas para interceptar e alterar as estruturas do *kernel* incluem o *hooking* ou acesso direto aos subsistemas do *kernel*, como no *virtual filesystem*, impedindo o mapeamento de um *incore inode* por um outro módulo do *kernel*, através da retirada de sua referência na lista de *incore inodes*.

Ao contrário de outros mecanismos de segurança do Linux, o controle de recursos proposto não utiliza os locais dos objetos (e.g., caminho no sistema de arquivos), mas irá utilizar as informações de posse presentes nas estruturas do módulo. Para conseguir isso, o módulo decidirá se, e como, um recurso será compartilhado, de acordo com a máscara de acesso do primeiro processo que obtiver a posse do objeto. Essa posse é atribuída através da relação entre processo controlado e objeto a ser manipulado, dado que tal objeto não foi manipulado por nenhum outro processo ou grupo de processos até então.

A abordagem de controle é feita no acesso da camada de *syscall*, fornecendo uma interface simples para conceder o controle sobre cada processo no sistema, pois permite mudar como um conjunto de *syscalls* se comportam, de acordo com o processo que as executarem. Com isso em mente, se um processo tentar manipular qualquer objeto no espaço de usuário, seu pedido terá de passar primeiro pelo OSPI, pois os *hooks* são colocados em *syscalls* como *open*, *read*,*write*, *exec*, *mmap*, etc. Com esses *hooks*, o módulo pode verificar se quem faz a chamada é um processo controlado, ou se ele está manipulando recursos controlados, aplicando restrições se necessário, e, em seguida, passar a execução para o código original da *syscall*. Deste ponto em diante, outras estruturas de segurança, como o *Grsecurity*, podem aplicar as suas políticas de segurança, de modo que o OSPI irá fortalecer um subsistema de segurança.

Dentro do *hook* de uma *syscall*, o módulo pode atuar de duas maneiras distintas: *managed* ou *restricted*. No modo *managed*, as políticas são aplicadas de acordo com um sistema simples de MAC ao conceder ou negar acesso aos objetos controlados, retornando um erro na *syscall* quando negado. Neste modo também é possível o uso de um objeto efêmero, criado exclusivamente como um objeto virtual. Seguindo a máscara de permissões, um processo pode ter todas as suas gravações confinados ao sistema de arquivos virtual, em se tratando de arquivos ou *buffers* geridos pelo *kernel*. Esta execução efêmera permite que um processo sustente uma execução privada de forma transparente, de modo que todas as alterações são confinadas na memória do *kernel* até o fim da execução, com o processo alheio a tal controle.

O modo *restricted* garante a privacidade e segurança de uma forma mais intru-

siva: ele criptografa os dados ao escrever e decifra ao ler, de todos os objetos tocados por um determinado processo. Isso impede que outros processos, independentemente de sua permissão, consigam ler dados em texto claro de um processo executado no modo restrito. Este modo pode ser persistente, de maneira que o processo ainda possa ler os dados manipulados mesmo se o sistema for reiniciado. Caso a persistência dos dados faça-se necessária, os diferentes processos lançados pelo mesmo executável terão a mesma chave de criptografia, invalidando assim o requisito que garante o isolamento de dados mesmo para diferentes instâncias do mesmo executável.

A cifragem e decifragem dos dados ocorrem em *syscalls* como `read` e `write`, ou `msgget` e `msgsnd`(IPC), usando a *CriptoAPI* [James Morris 2015] do *kernel Linux*. Essa *API* suporta diversos criptossistemas e funções de *hashing*, utilizando sempre as implementações e abordagens dos algoritmos que melhor aproveitem os dispositivos de *hardware* presentes. As operações sobre os objetos controlados ocorrem utilizando uma chave no criptossitema *AES128* disponibilizado pelo *kernel*. Uma vez que tais operações são feitas internamente nas *syscalls* controladas, essas são transparentes para os processos envolvidos. A chave utilizada é composta de duas outras chaves, a chave do módulo, que nunca é armazenada na memória principal, e a chave privada, única para cada processo controlado.

A geração da chave privada do processo pode ocorrer de duas formas, através do gerente em nível de usuário que devolve o *hash* MD5 do *ELF* relativo ao executável do processo, ou através do módulo utilizando informações da *task_struct* do processo. Devido a esse módulo ainda ser um trabalho em andamento, a gerência em nível de usuário funciona para quaisquer *kernels* da família *3* e *4*, enquanto a gerência exclusiva via módulo funciona apenas na família *3* do *kernel Linux*. O módulo do OSPI utiliza um tipo `mm_segment_t` da estrutura `thread_info` e um tipo `mm_struct` da estrutura `task_struct` para obter a seção *.text* do processo e assim calcular o seu *hash* MD5. Um detalhe importante referente a implementação é que o endereço apontado pela varável `mm_segment_t` deverá ser convertido para o endereço real, dado que a seção *.text* começaria no endereço virtual *0x00000000*. Essa conversão é feita pela tupla de funções do *kernel* `get_fs` e `set_fs`.

Independente da abordagem utilizada, o *hash* é capaz de representar todo o código disponível a ser executado pela *task* e assim, com alto grau de confiança, ser a chave única de um processo controlado mesmo quando a máquina reinicie e todos os identificadores do processo mudem. Caso a execução não tenha o bit de persistência, uma semente pseudo-aleatória provida pela entropia do *kernel* é adicionada como *salt* ao *hash*, tornando possível que processos de um mesmo executável mantenham instâncias privadas entre si. Uma vez que uma execução não persistente termine, todos os objetos manipulados pelo processo em questão podem ser considerados perdidos, em vista da criptografia utilizada e da perda da chave.

A chave privada do processo não pode ser usada diretamente como a chave no criptossistema *AES128*, pois o espaço de bits suscetíveis a colisões inerente ao *MD5* enfraqueceria a confidencialidade do *AES128*. Desta forma, a chave utilizada é o resultado da operação XOR (ou-exclusivo) das chaves privadas do processo e do módulo proposto. O uso de tal operação, para a geração de chaves únicas a partir de uma chave *tamper-proof* e outra sem tal garantia, não enfraquece ou expõe a primeira chave, ao passo que torna

as chaves geradas diferentes tanto quanto a segunda chave da operação. [Steel 2005] demonstra como podem ocorrer ataques a partir da extração de características das chaves geradas por meio de algoritmos que usam operações *XOR* em *Security APIs* largamente utilizadas na infraestrutura bancária. Portanto, a segurança contra a dedução da chave gerada no presente trabalho depende da segurança do algoritmo utilizado para o cálculo do *hash* e do algoritmo que gerou a chave do módulo. Tal segurança pode ser violada se não houver preservação mútua das características presentes nas saídas dos algoritmos, ou se as características da entrada não forem preservadas na saída.

O estudo supracitado apresenta outra fragilidade adicional à extração de características: a propriedade do inverso, na qual o resultado da operação será sempre "0" (zero). Esse ataque depende do conhecimento da chave resultante e de uma das chaves da composição, no caso a chave privada de um processo, em vista da facilidade de calcula-la. Porém, a chave do processo sozinha não tem a intenção de ser sigilosa, e de fato pode ser conhecida, dado que a chave resultante e a chave do módulo nunca residem a memória principal. Logo, ambas as chaves só podem ser conhecidas em nível de *kernel* por meio da leitura de registradores. A chave resultante é produto de instruções do processador e, portanto, é armazenada em registradores. Essa chave imediatamente após ser gerada serve como parâmetro no *AES128* através de instruções AES-NI (*Advanced Encryption Standard New Instructions*) presentes no processador. Isso faz com que ela não seja armazenada na memória principal.

A chave do OSPI é utilizada para todos os processos, mas não deve ser gerada pelo módulo e sim por uma boa fonte de entropia. Pretende-se, em uma versão posterior, que o módulo seja capaz de ler a chave nas primeiras fases do *boot*—por meio de um dispositivo de armazenamento de confiança ou um *hardware* gerador—e, após leitura preencha com "0" cada *buffer* usado para transporte da chave, armazenando-a, por fim, nos registradores de *debug* do processador (`DR0` a `DR3`). Esses registradores não podem ser utilizados fora do modo privilegiado do processador, a não ser que um *debugger* executando como *root* os acesse via capacidades especiais providas pelo *kernel Linux*.

O OSPI usa diretamente as funções do *kernel* para desabilitar e impedir o uso dos registradores de *debug* por meio das seguintes funções: `arch_uninstall_hw_breakpoint`; `hw_breakpoint_restore`; `arch_install_hw_breakpoint` (irá sofrer *hook* e chamar `hw_breakpoint_restore`). Desta forma, inviabiliza-se o uso de *hardware breakpoints* ao mesmo tempo em que se aumenta o nível de segurança do sistema, pois há trabalhos que mostram como *rootkits* abusam dessa funcionalidade de forma a persistir no sistema alvo e comprometer tanto o OSPI quanto o *Grsecurity* [Emil Persson 2012], [Phrack.0x0c41 2008].

A Figura 1 ilustra as interações entre os diversos níveis de processos presente em um sistema. As interações representadas pelas arestas pretas pertencem a processos não-controlados, enquanto que as em escala de cinza representam os processos sujeitos a alguma forma de controle. Arestas cinza-claro ou cinza-escura são interações com objetos pertencentes aos grupos "0" ou "1", cujos dados são cifrados ou decifrados em decorrência do parâmetro presente em uma *syscall* controlada. As interações negadas são representadas por arestas esbranquiçadas, nas quais os processos não possuem entrada na lista de processos controlados e, portanto, não possuem uma chave simétrica.

Caso fossem processos do *grupo0* tentando acessar objetos do *grupo1*, mesmo sem uma regra específica no *MAC* para coibir esse tipo de atividade, o objeto acessado em questão seria decifrado pela chave errada, não revelando assim suas informações (o resultado de tal decifragem seria ininteligível). As arestas cinza-médio indicam uma interação anômala, enquanto que a aresta tracejada indica um ataque não-detectado pelo *Grsecurity*.

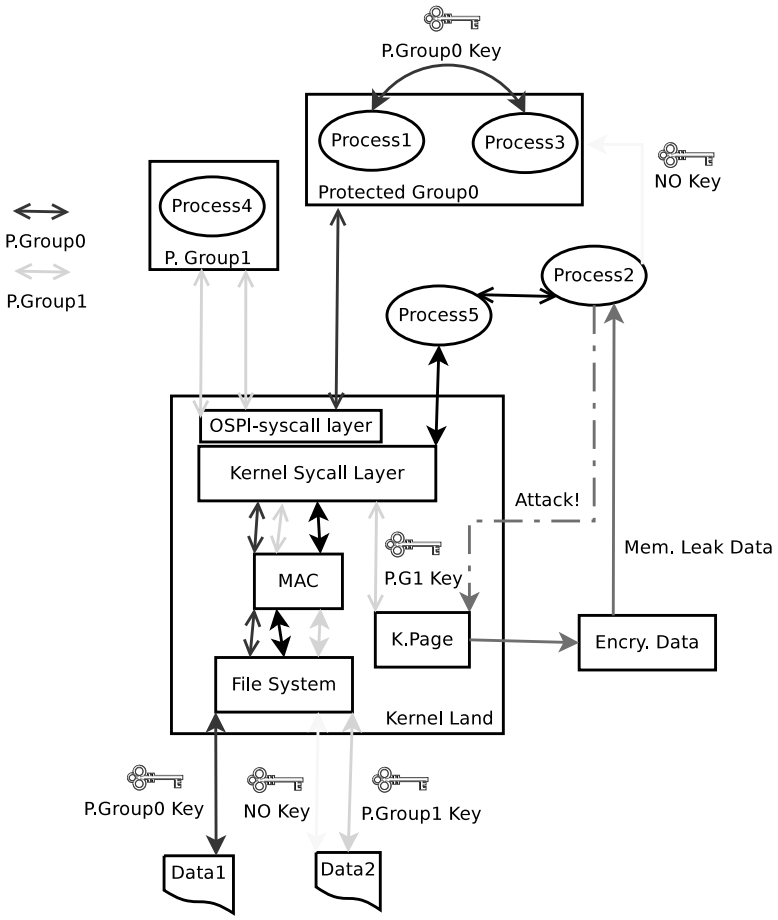

Figura 1. Diagrama de interações entre grupos de processos controlados e suas chaves com processos fora do escopo de controle do OSPI.

4. Testes e Resultados

O desenvolvimento do módulo está em andamento, almejando um mecanismo com fina granularidade e proteção em baixo nível de forma a impor as necessidades de segurança de processos sensíveis ou especiais. O mecanismo foi projetado para ser capaz de aumentar o nível de privacidade dos dados nos casos onde os mecanismos de proteção existentes falham. Além disso, a combinação da abordagem proposta neste artigo com o *Grsecurity* faz com que um determinado processo possa ser protegido contra violações de privacidade e ataques em cenários incomuns.

Para testar o protótipo desenvolvido em sua versão mais atual, bem como a atuação de outros mecanismos de segurança disponíveis para instalação, utilizou-se um cenário de exploração de virtualização com *Docker*: *docker container breakout* [Turnbull 2014].

Uma vez que o ataque em questão não possui uma regra de detecção correspondente a ataque conhecido até o momento da escrita deste artigo, tanto o *Grsecurity* quanto o *AppArmor* foram incapazes de evitar que este fosse completado com sucesso. O *profile* do *AppArmor*, até a versão do *Docker 0.11*, permitia bloquear *kernel capabilities* através de sua inserção em uma *blacklist*. Porém, a falha contou com uma combinação de *syscalls* e *capabilities* que não estavam na *blacklist* até que o ataque ficou conhecido. Esta combinação causou uma brecha que permitiu a um *docker* executado pelo *root* ser capaz de ler arquivos e diretórios localizados fora do isolamento.

O motivo do sucesso deste ataque em sistemas com *Grsecuriy* e *AppArmor* deu-se pelo fato de que, mesmo que o processo isolado não seja executado como *root* internamente ao *docker*, um processo com privilégio de *root* executa *syscalls* como tal. Isto ocorre devido a virtualização em espaço de usuário compartilhar a mesma *syscall layer* do resto do sistema. Para barrar esse ataque, *AppArmor* recebeu um novo *profile*, em que as *capabilities* foram bloqueadas por padrão (exceto para aqueles em uma *whitelist*).

O OSPI foi capaz de bloquear o ataque sem sequer conhecê-lo. Portanto, o módulo proposto já é capaz de atenuar uma classe inteira de ataques semelhantes, devido à utilização do modo restrito no processo protegido. Assim, um processo que quebre o isolamento provido pelo *Grsecurity* não será capaz de ler qualquer objeto que não seja seu, de acordo com a política de acesso do *OSPI*. Mesmo que o processo consiga subverter as restrições do *MAC* e ter acesso ao objeto, dado que o conteúdo do objeto está cifrado, esse acesso não será bem sucedido no contexto de um ataque de violação da confidencialidade. O cenário da escrita funciona de forma semelhante: como o processo que possui a posse do objeto sempre o recebe na leitura após o modulo decifra-lo, a escrita direta deste objeto por um ataque resultará em dados ininteligíveis para o processo protegido. Desta forma, o processo subvertido não poderá escrever o que pretende, pois mesmo que haja alteração dos dados, esta corrompe o arquivo, mitigando o ataque (por exemplo, a escrita de credenciais em */etc/shadow* resultaria em uma linha inválida e, portanto, inútil para um *login* do atacante).

Embora o *OSPI* ainda não tenha sido amplamente testado, o sucesso alcançado no bloqueio de alguns ataques que não são tratados por outros *frameworks* sem configurações específicas mostra que a abordagem é promissora. Os cenários avaliados—*rootkits* e *docker leak*—valida sua eficácia como um mecanismo complementar para reforçar a segurança de um sistema operacional e seus processos cuja proteção depende dos *frameworks* tradicionais.

Ao longo do projeto, foram feitas aferições acerca do *overhead* resultante dos mecanismos utilizados pelo módulo, a fim de se escolher a abordagem mais eficiente que, ao mesmo tempo, não implicaria em redução na segurança do sistema. As abordagens implementadas, tais como utilizar a fila de processos e objetos estruturadas em uma tabela *hash* e realizar a conversão dos endereços virtuais para os reais na *OSPI syscall layer* previamente à chamada original do *kernel*, reduziram muito o *overhead* e o *turnaround time*[3]. das chamadas de sistema efetuadas. A Tabela 1 ilustra algumas *syscalls* executadas no mesmo *kernel*, mas com *frameworks* de segurança diferentes. O *turnaround time* de uma *syscall* é representado em milissegundos, enquanto que o *overhead* em porcentagem

[3]Tempo necessário para completar um processo, do lançamento para execução até o retorno de sua saída

da chamada em relação ao *kernel* sem nenhuma proteção. Foi utilizado o `SystemTap` e uma adaptação do *script iotime.stp* [RedHat 2007] para obtenção do tempo gasto nas chamadas.

Foram feitas 1024 chamadas de cada uma das *syscalls* representadas na Tabela, utilizando-se *1KB* de dados por meio da ferramenta *syscall fuzzer Trinity*[Jones 2015] quando necessário. A chamada `null I/O` corresponde a execuções das *syscalls* `read` e `write`, porém com o *buffer* vazio. Todas as chamadas realizadas foram feitas de modo pseudo-aleatório, sempre seguindo a semântica dos parâmetros necessários para cada *syscall*. Com isso, elas representam tanto o comportamento esperado quanto o inesperado, simulando uma anomalia que possivelmente é tratada pelos *frameworks* de segurança testados.

Tabela 1. Tabela comparativa com o *turnaround time* em milissegundos das *syscalls* em um sistema protegido pelos *frameworks* OSPI, Grsecurity ou AppArmor em comparação à chamada sem proteção (base). As medidas de*overhead* correspondem à porcentagem do *turnaround time* da chamada protegida em relação à chamada desprotegida.

Syscall	Base	OSPI	Overhead	Grsecurity	Overhead	Apparmor	Overhead
`null I/O`	0,13	0,13	1%	0,13	5%	0,13	12%
`open/close`	2,05	2,05	4%	2,09	2%	4,28	109%
`fork`	131,17	131,43	2%	130,23	1%	134,71	27%
`execve`	386,52	575,91	49%	552,72	43%	587,51	52%
`pipe`	6,67	7,33	10%	7,67	15%	7,53	13%
`read`	8,38	11,98	43%	9,30	11%	9,14	9%
`write`	14,09	22,12	57%	15,49	10%	15,27	8%

5. Conclusão

Neste artigo, discutiu-se os *frameworks* mais utilizados para a proteção da privacidade dos dados de processos em execução em sistemas operacionais Linux, como *Grsecurity* e *AppArmor*. Foram levantadas as limitações de tais *frameworks* em face de alguns ataques que exploram vulnerabilidades de programas causando a exposição de informações sensíveis, seja por vazamento de dados ou pela presença de *malware* em nível privilegiado do sistema operacional. Com a finalidade de aumentar o nível de segurança do usuário, impedindo certas violações à confidencialidade, foi proposto o OSPI, um mecanismo de segurança que isola processos arbitrários do sistema, protegendo assim o contexto de seus dados. Apresentou-se a arquitetura do mecanismo proposto, bem como testes preliminares e seus resultados. Foi possível notar que o mecanismo proposto alcançou, em média, um baixo *overhead* quando comparado com o *AppArmor*, mas obteve taxa de *overhead* superior ao *Grsecurity*. Entretanto, o último resultado era esperado devido ao uso do critpossistema *AES128*, que por si só demanda de um esforço computacional superior ao demandado pelo *Grsecurity*. Porém, a diferença obtida em relação ao *AppArmor*, que possui em média um *overhead* inferior ao *SELinux*, foi notável, principalmente ao se considerar que as chamadas *open/read* apresentaram mais do que o dobro de *turnaround time*. Tal diferença é causada principalmente pelo tempo para se fazer o *parsing* dos arquivos de políticas, inexistente no *OSPI* ou *Grsecurity*. Como um protótipo, os resultados foram promissores e indicam o potencial da continuidade do projeto como um mecanismo viável de proteção complementar.

Referências

[AppArmor 2013] AppArmor, W. (2013). Apparmor project wiki. http://wiki.apparmor.net. Acessado em: 30 Mar. 2016.

[Ausanka-Crues 2006] Ausanka-Crues, R. (2006). Methods for access control: Advances and limitations. Technical report, Harvey Mudd College.

[Brown 2015] Brown, N. (2015). Overlayfs documentation. https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt. Acessado em: 03 Abr. 2016.

[Bugliesi et al. 2012] Bugliesi, M., Calzavara, S., Focardi, R., and Squarcina, M. (2012). Gran: Model checking grsecurity rbac policies. In Chong, S., editor, *CSF*, pages 126–138. IEEE.

[Chen et al. 2009] Chen, H., Li, N., and Mao, Z. (2009). Analyzing and comparing the protection quality of security enhanced operating systems. In *NDSS*. The Internet Society.

[Corporation 2015] Corporation, N. D. (2015). About tomoyo linux. http://tomoyo.osdn.jp/about.html.en. Acessado em: 30 Mar. 2016.

[Emil Persson 2012] Emil Persson, J. M. (2012). Debug register rootkits. a study of malicious use of the ia-32 debug registers. Technical report, School of Computing,Blekinge Institute of Technology.

[Fox et al. 2009a] Fox, M., Giordano, J., Stotler, L., and Thomas, A. (2009a). Selinux and grsecurity: A case study comparing linux security kernel enhancements.

[Fox et al. 2009b] Fox, M., Giordano, J., Stotler, L., and Thomas, A. (2009b). Selinux and grsecurity: Mandatory access control and access control list implementations.

[Hund et al. 2013] Hund, R., Willems, C., and Holz, T. (2013). Practical timing side channel attacks against kernel space aslr. In *IEEE Symposium on Security and Privacy*, pages 191–205. IEEE Computer Society.

[James Morris 2015] James Morris, David S. Miller, H. X. (2015). Linux kernel cryptographic api. http://lxr.free-electrons.com/source/Documentation/crypto/api-intro.txt. Acessado em: 30 Mar. 2016.

[Jones 2015] Jones, D. (2015). Trinity: Linux system call fuzzer. https://github.com/kernelslacker/trinity. Acessado em: 30 Mar. 2016.

[Kirkland 2012] Kirkland, D. (2012). Ecryptfs documentation. http://ecryptfs.org/documentation.html. Acessado em: 03 Abr. 2016.

[McAllister 2012] McAllister, K. (2012). Attacking hardened linux systems with kernel jit spraying. https://lwn.net/Articles/525609. Acessado em: 30 Mar. 2016.

[Morgan 2015] Morgan, S. C. (2015). Cybersecurity market report q4 2015. http://cybersecurityventures.com/cybersecurity-market-report.

[Morris 2013a] Morris, J. (2013a). Linux security module framework. https://www.linux.com/learn/overview-linux-kernel-security-features. Acessado em: 30 Mar. 2016.

[Morris 2013b] Morris, J. (2013b). Selinux project wiki. http://selinuxproject.org. Acessado em: 30 Mar. 2016.

[Nguyen 2004] Nguyen, B. (2004). Linux proc file system. http://www.tldp.org/LDP/Linux-Filesystem-Hierarchy/html/proc.html.

[Onarlioglu et al. 2013] Onarlioglu, K., Mulliner, C., Robertson, W., and Kirda, E. (2013). PrivExec: Private Execution as an Operating System Service. In *IEEE Symposium on Security and Privacy*, San Francisco, CA USA.

[Phrack.0x0c41 2008] Phrack.0x0c41 (2008). Mistifying the debugger ultimate stealthness. http://phrack.org/issues/65/8.html. Acessado em: 30 Mar. 2016.

[RedHat 2007] RedHat (2007). Systemtap script iotime. https://sourceware.org/systemtap/examples/io/iotime.stp. Acessado em: 30 Mar. 2016.

[Schaufler 2011] Schaufler, C. (2011). Smack project description. http://schaufler-ca.com/description_from_the_linux_source_tree. Acessado em: 30 Mar. 2016.

[Spengler 2015a] Spengler, B. (2015a). Grsecurity comparison matrix. https://grsecurity.net/compare.php. Acessado em: 30 Mar. 2016.

[Spengler 2015b] Spengler, B. (2015b). Grsecurity features. https://grsecurity.net/features.php. Acessado em: 30 Mar. 2016.

[Spengler 2015c] Spengler, B. (2015c). Whats is grsecurity. https://grsecurity.net/about. Acessado em: 30 Mar. 2016.

[Steel 2005] Steel, G. (2005). Deduction with XOR constraints in security API modelling. In Nieuwenhuis, R., editor, *Proceedings of the 20th International Conference on Automated Deduction (CADE'05)*, volume 3632 of *Lecture Notes in Artificial Intelligence*, pages 322–336, Tallinn, Estonia. Springer.

[SUSE 2016] SUSE (2016). Apparmor and selinux comparison. https://www.suse.com/support/security/apparmor/features/selinux_comparison.html. Acesso em 11 de abril de 2016.

[Turnbull 2014] Turnbull, J. (2014). Docker Container Breakout Proof-of-Concept Exploit. https://blog.docker.com/2014/06/docker-container-breakout-proof-of-concept-exploit/.