Detecção e Recuperação de Intrusão com uso de Controle de Versão

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Gabriel Dieterich Cavalcante e aprovada pela Banca Examinadora.

Campinas, 16 de junho de 2010.

Prof. Dr. Paulo Lício de Geus (Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Miriam Cristina Alves – CRB8 / 5089

Cavalcante, Gabriel Dieterich
C314d Detecção e recuperação de intrusão com uso de controle de versão/Gabriel Dieterich Cavalcante-- Campinas, [S.P. : s.n.], 2010.

Orientador : Paulo Lício de Geus
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Redes de computação - Sistemas de segurança. 2. Sistemas operacionais (Computadores). 3. Software - Manutenção. 4. Software - Segurança.. I. Geus, Paulo Lício de. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: Intrusion detection and recovery with revision control systems.

Palavras-chave em inglês (Keywords): 1. Computer networks – Security systems. 2. Operating systems (Computing). 3. Software – Recover. 4. Software – Security/Protection.

Área de concentração: Segurança de Computadores

Titulação: Mestre em Ciência da Computação

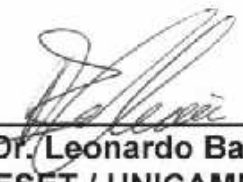
Banca examinadora: Prof. Dr. Paulo Lício de Geus (IC – UNICAMP)
Prof. Dr. Leonardo Barbosa e Oliveira (DTI – CESET – UNICAMP)
Prof. Dr. Ricardo Dahab (IC - UNICAMP)

Data da defesa: 05/05/2010

Programa de Pós-Graduação: Mestrado em Ciência da Computação

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 05 de maio de 2010, pela Banca examinadora composta pelos Professores Doutores:



Prof. Dr. Leonardo Barbosa e Oliveira
DTI-CESET / UNICAMP

Prof. Dr. Ricardo Dahab
IC / UNICAMP

Prof. Dr. Paulo Lício de Geus
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Detecção e Recuperação de Intrusão com uso de Controle de Versão

Gabriel Dieterich Cavalcante[1]

Junho de 2010

Banca Examinadora:

- Prof. Dr. Paulo Lício de Geus (Orientador)
- Prof. Dr. Ricardo Dahab
  Instituto de Computação - UNICAMP
- Prof. Dr. Leonardo Barbosa e Oliveira
  DTI - CESET - UNICAMP
- Prof. Dr. Mario Jino
  DCA - FEEC - UNICAMP (Suplente)
- Prof. Dr. Célio Cardoso Guimarães
  Instituto de Computação - UNICAMP(Suplente)

[1]Bolsista CNPq 2008-2010

Resumo

Existe uma grande quantidade de configurações presentes em sistemas atuais e gerir essas configurações é um trabalho árduo para qualquer administrador de sistema. Inúmeras configurações podem ser definidas para uma só máquina e as combinações entre elas implicam de forma significativa no seu desempenho. A partir do momento que um sistema em pleno funcionamento pára de funcionar, algo em sua estrutura pode ter mudado. Este cenário é comum no processo de desenvolvimento de *software* onde o código fonte pode deixar de compilar ou ainda uma funcionalidade pode se perder. Controladores de versão são usados para reverter o estado do código para uma data anterior, solucionando o problema. Verificadores de Integridade são utilizados para detectar estas mudanças porém não possuem mecanismos específicos para recuperação. Este estudo propõe e implementa uma arquitetura integrada que conbina verificação de integridade e mecanismos de recuperação. Foram executados testes para determinar a sobrecarga total deste método, além de estudos de caso para verificar a sua eficiência de recuperação.

Abstract

Current computer systems have a huge number of configurations that are hard to manage. The combinations of system configurations can impact on performance and behavior. From the moment that a system stops working correctly it is remarkable that something has changed. That is in common in software development, where changes made by the programmer may result in some features no longer working or the project not compiling anymore. Revision control systems can recover a previous state of the source code through revision mechanisms. Integrity checking is used to catch file modifications, however this technique does nothing toward recovering those files. This study proposes and implements an integrated architecture that combines integrity checking and restoring mechanisms. Tests were executed in order to measure the load imposed by the solution. In addition, analysis of three case studies shows the efficiency of the adopted solution.

Dedicatória

Aos meus dois tesouros,
Thays e Pietro.

Agradecimentos

Agradeço a Deus, por ter-me feito nascer pobre. A pobreza foi-me uma amiga benéfica; ensinou-me o preço verdadeiro dos bens úteis à vida, que sem ela não teria conhecido. Evitando-me o peso do luxo, devotei-me ao trabalho e ao estudo, além de cultivar os princípios da família e da humildade.

Longe de todos, eu Thays e Pietro viemos batalhar uma nova jornada. Gostaria de agradecer minha família pelo suporte incondicional e principalmente pelo carinho dado ao longo desses dois anos de curso, em especial minha mãe Clair, meu pai Valdizo, meu irmão Thiago e sua esposa Cristiane, meus sogros Sr. Mauro e D. Marilda, Rodrigo e Vanusa, José Wagner. Sei que não mediram esforços para nos fazer companhia, apesar da distância.

Gostaria de agradecer ao meu orientador, Prof. Dr. Paulo Lício de Geus, por me aceitar em uma hora incômoda, pois se preparava para uma longa viagem de estudos no exterior. E ainda em sua estadia fora do país, pela disposição em nossas videoconferências, apesar da diferença de horários tudo corria muito bem. Enfim, obrigado por ter me ajudado na conclusão deste trabalho.

Aos meus amigos do LAS-IC-UNICAMP, por me ajudarem a colocar as idéias no lugar, além dos momentos de descontração e café. Em especial, Ricardo, Miguel, Cléber, Arthur, Edmar, André Pereira, André Grégio, Luciano, Robson, Victor, João.

Aos meus grandes amigos desde a graduação, que também vieram buscar estudos aqui em Campinas, Ivo, Marcelo, Sidney e Leonardo. Aos meus colegas de mestrado, por me suportarem nas longas noites de estudo, Carlos Zampieri, Luciano Jerez, Juliana de Santi, Eriksson, Leonardo Ecco, Ricardo Pannagio, Fabrício, Geraldo Magela, Priscila, Andrei, Bruno Villar, e tantos outros. Aos amigos do IC, que apesar de não haver disciplinas em comum, construímos amizade, Fábio Faria e Rodrigo Tripodi.

Apesar de serem o alvo da dedicatória deste trabalho, gostaria de agradecer mais uma vez ao meu filho Pietro e minha esposa Thays, por às vezes sacrificarem suas horas de lazer familiar para que eu me dedicasse ao estudo. Obrigado, eu amo vocês!

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Enquanto especialistas desenvolvem novas técnicas de segurança, invasores estão ativamente envolvidos na criação de técnicas para injetar ou descobrir falhas latentes no código alheio. Isso significa que ameaças evoluem durante o ciclo de vida de determinado sistema em uso, que permanece exposto até que passe por uma atualização. Em um ambiente perfeito, o sistema evoluiria até ser livre de falhas, porém isto é muito difícil de acontecer, se não impossível.

Quando sistemas são comprometidos uma das mais árduas tarefas é a recuperação. A recuperação é feita após um ataque ser descoberto e tipicamente envolve vários passos como: instalação de uma nova imagem de sistema que inclui o sistema operacional e seus aplicativos; instalação de correções para as vulnerabilidades conhecidas e recuperar dados importantes corrompidos. Cada passo é manual e tedioso, além de tomar considerável quantidade de tempo. Além disso, é necessário certificar-se de que não existe nenhum código estranho embutido em algum arquivo executável do sistema, esse código pode comprometer o sistema a longo prazo ou ainda o sistema recém instalado.

Inegavelmente, computadores atuais—pessoais, estações de trabalho e servidores—acrescentam aos seus usuários uma quantidade crescente de recursos e serviços. Porém, também existe uma crescente quantidade de custos, investidos na solução de problemas e reparos de avarias em sistemas operacionais. Quando estes ocorrem há considerável perda de produtividade nos estudos, trabalho, lazer etc. Estudos revelam que suporte técnico envolve cerca de 17% do custo de vida total de um computador pessoal. A maior parte deste suporte é gasto com reparos no sistema operacional ou em seus aplicativos [58].

Muitos problemas são causados pelas configurações presentes em cada sistema. A perda de algumas configurações ou incoerência das mesmas pode acarretar seu mal funcionamento, ou até sua total queda. Mudanças nas configurações são frequentemente causadas por existir um depósito de dados compartilhados entre vários aplicativos do sistema e do usuário. Dois exemplos deste tipo de repositório são: os arquivos de configuração

Unix—*Unix Resource Files*—e o Sistema de Registros do *Windows*.

A função dos arquivos de suporte é prover uma vasta quantidade de recursos, naturalmente partilhados por várias aplicações, dentre os quais: fornecer meios para que aplicações instaladas em momentos diferentes possam interagir e se integrar umas com as outras; conter mecanismos para que aplicações sejam registradas em serviços base do sistema para utilizarem suas funcionalidades; permitir que pequenos componentes possam se acoplar a algum aplicativo para estender suas funcionalidades.

A quantidade de combinações entre esses arquivos e suas respectivas particularidades de configuração, geralmente são inúmeras. Gerenciar e achar a melhor configuração para o sistema é uma tarefa complicada e muitas vezes frustrante. Dentro dessa mesma perspectiva, a quantidade de tempo gasto para eliminar a fonte de um problema é tão grande quanto à de encontrar um melhor aproveitamento dos seus recursos—"melhor desempenho". Além disso, após uma invasão, uma das piores tarefas é a análise e recuperação do sistema que foi comprometido.

A trajetória de solução de problemas de sistemas frequentemente é acompanhada da afirmação: "funcionava ontem e não está funcionando hoje". Dito isso, é de se esperar que algo tenha mudado no sistema para ele deixar de funcionar. Encontrar o que foi alterado é de fato, o princípio da solução. Se de alguma forma, o sistema fosse preparado para que pudesse retornar a um estado anterior, seria economizada uma quantidade considerável de tempo.

Este tipo de histórico é uma coisa comum em ferramentas de controle de versão. Ferramentas desse tipo são usadas no processo de desenvolvimento de software, elas são responsáveis pelo acesso paralelo dos programadores ao código e guardam eficientemente a evolução do mesmo. Existe uma funcionalidade presente nestes versionadores que pode solucionar o caso da resiliência para configurações de sistemas operacionais é a revisão de código. A revisão de código permite que seja extraída uma versão do código baseando-se em uma data anterior, ou seja, mostrar o estado do repositório na data desejada. Isto é usado para reverter algum estado do repositório de código quando alterações não desejáveis acontecerem ou mesmo para fechar uma versão do código.

Sistemas de Controle de Versão possuem mecanismos para mostrar a diferença entre duas ou mais versões de um arquivo. Porém este não apresenta eficiência em determinar diferenças entre arquivos binários. Existem ferramentas que buscam diferenças entre os binários presentes em um sistema de arquivos com o intuito de checar a integridade dos mesmos. Esta é uma maneira útil de buscar alterações que podem ter sido feitas por um ataque bem sucedido, que visou implantar uma brecha permanente no sistema para viabilizar outros ataques.

Visando este cenário, esta dissertação dedica-se a buscar maneiras de tornar um sistema operacional *GNU/Linux* resiliente, ou seja, que ele seja capaz de voltar a estados

anteriores a quando algum problema afetar seu funcionamento. Desta forma o sistema estaria novamente disponível em menor espaço de tempo, deixando ao administrador apenas os passos para evitar a falha que ocorreu.

Para este fim será usada uma ferramenta de Controle de Versão, que fará o armazenamento do histórico bem como a cópia dos arquivos de configuração de um sistema. Se forem encontradas diferenças entre a versão atual de um arquivo e a última versão armazenada no Controle de Versão pode-se identificar uma alteração de configuração indesejada. Com a base de dados do Controle de Versão o sistema poderá ser recuperado automaticamente.

1.1 Justificativa e Motivação

A motivação para este projeto advém principalmente da possibilidade de diminuir o tempo gasto para serem diagnosticados e reparados problemas de sistemas operacionais e suas configurações. A diminuição no custo de produção de CPUs, dispositivos de rede e armazenamento, acabou mudando o ambiente de tecnologia no mundo. Agora nos deparamos com um cenário onde custo de recursos humanos supera sensivelmente os custos de recursos de tecnologia. Isso denota a necessidade de desenvolvimento de sistemas para automatizar a análise e recuperação, tendo como principal objetivo minimizar o esforço humano para este tipo de tarefa.

Segundo Watson & Benn [61], sistemas de recuperação baseados em snapshots e backups, como os de Santry *et al* [46] e Soules *et al* [50], têm comum e bem entendido funcionamento. Esses métodos recompõem eficazmente um sistema, porém quando o aplicativo entra em funcionamento é que surgem suas desvantagens. Entre os arquivos que foram salvos para garantir a recuperação do sistema, existe uma grande quantidade de dados que não foram alterados e não precisam ser recuperados. Isso gera um desperdício de recursos alocados para o armazenamento desses dados.

Com os sistemas de versões apenas os arquivos que foram modificados terão de ser salvos. o que será possível graças ao sistema versão presente neste tipo de ferramenta. O sistema de revisão consegue montar uma versão dos dados armazenados tendo como entrada uma data. Dessa forma seria contornado o problema de armazenamento redundante de dados.

O princípio de restauração e recuperação de sistemas tem aspectos comuns em diferentes áreas da computação, principalmente pela automatização e maior disponibilidade que este método pode trazer aos sistemas. De fato, são necessárias pesquisas na área de Administração de Sistemas para que o processo de reconfiguração de sistemas seja mais eficiente. Na área de Segurança, técnicas aprimoradas para este fim são necessárias para conter mais rapidamente ataques futuros, como por exemplo eliminar – às vezes sem saber

de sua existência – um binário alterado por um atacante, além disso para fornecer meios para que sejam descobertas as ações do atacante.

Outra vantagem deste tipo de sistema é a possibilidade de ser utilizado em máquinas domésticas, pois armazenará seus dados localmente e sem utilização de meios externos. Simplificar este método para uso é um desafio necessário: se o usuário comum tiver uma máquina em funcionamento correto, ele não precisará saber como funcionam os arquivos de configuração, bastando apenas restaurar o estado anterior da máquina. A ausência de iniciativas em software livre para este tipo de facilidade também é um grande fator de motivação para a realização deste estudo.

1.2 Organização do Documento

O restante do documento está organizado da seguinte forma: o Capítulo 2 apresenta um breve estudo sobre resiliência e uma visão geral sobre sistemas desse tipo, bem como um cruzamento de conceitos de tolerância a falhas com este conceito; o Capítulo 3 apresenta o estudo realizado para escolher uma ferramenta de Controle de Versão para ser usada neste projeto; o Capítulo 4 mostra uma coleção de métodos de detecção de intrusão e alteração; o Capítulo 5 traz a arquitetura proposta bem como detalhes da implementação; o Capítulo 6 apresenta os resultados alcançados e por fim o Capítulo 7 apresenta a conclusão e os trabalhos futuros identificados ao longo do desenvolvimento deste trabalho.

Capítulo 2

Resiliência e Restauração de Sistemas

A noção de resiliência foi inaugurada pelas ciências exatas, principalmente física e engenharia, que a definiram como a energia de deformação máxima que um material é capaz de armazenar sem sofrer alterações permanentes em sua estrutura original, ou seja, é capaz voltar ao seu estado original sem qualquer deformação. Com o passar dos anos, esse termo assumiu significados também em outras áreas, não sendo mais exclusivo da física e engenharia. Atualmente, o termo é utilizado na saúde, na psicologia, na tecnologia e outros, inclusive é usado como característica de seleção de funcionários. Em recursos humanos o termo significa a capacidade humana de superar tudo, tirando proveito dos sofrimentos, e ser inerente às dificuldades [67].

Na Ciência da Computação o termo também tem significados particulares. Para McQuarrie *et al* [31], em Redes de Computadores o significado de perda de formato é traduzido para alterações na capacidade de funcionamento. Dessa forma, uma rede ou componente de rede resiliente deve possuir habilidades para responder e resistir a falhas, manter o fluxo e ter fácil configuração – tendo pouco ou nenhum impacto nos serviços que presta – possuindo alta disponibilidade. Segundo Johnson & Vanstone [23], em Criptografia, sistemas resilientes devem ser capazes de mesmo sob falhas catastróficas, manter a qualquer custo, a propriedade criptográfica dos dados que estão sob sua ação. Na área Segurança de Computadores, essa característica pode ser traduzida como a capacidade de um sistema a resistir a ataques maliciosos [36].

Em geral, resiliência é a capacidade de um sistema se recuperar após sofrer danos, que podem ser causados por erros no desenvolvimento do software, uso incorreto, erro humano ou ataque malicioso. Porém, existe uma corrente de desenvolvedores de software que trata resiliência como persistência, um sistema persistir em funcionamento, em busca de um dado etc. Para este projeto em particular, o termo resiliência deve ser interpretado como a capacidade de um sistema voltar a um estado anterior. Através disso, restaurar configurações e dependências para que seu funcionamento seja idêntico a uma data anterior

à que um erro foi encontrado.

2.1 Sistemas Resilientes

Bisset *et al* [8], caracteriza Sistema Resiliente todo sistema computacional capaz de funcionar – frequentemente com suas capacidades reduzidas – na presença de falhas e possuir capacidades de recuperação, para alcançar algum estado anterior de funcionamento pleno. Sistemas deste tipo possuem dois modos de operação, modo normal e modo de integridade/recuperação, não podendo operar em ambos simultaneamente. O sistema opera em modo normal quando qualquer defeito que aconteça não afete de modo drástico a vida do usuário. O que significa que o sistema deve estar acessível quando está em modo normal.

Para Bisset *et al* [8], um sistema tem sua integridade afetada quando um defeito causa perda de dados ou o corrompimento dos mesmos. Isto significa que um sistema operando em modo integridade deve estar configurado para evitar maiores perdas nos dados, mesmo que o sistema precise ficar inacessível para conseguir isto.

Existe uma característica geral que liga esse tipo de sistema a outros dois, a necessidade de possuir um componente que mantenha o sistema funcionando na presença de falhas. Por haver tal semelhança muitas vezes eles são confundidos entre si, tratam-se de sistemas tolerantes a falhas e sistemas tolerantes a desastres.

Para Weber [62], sistemas tolerantes a falhas pregam tanto disponibilidade e integridade quando confrontados com uma única falha, e em certas circunstâncias, quando se depara com múltiplas falhas. Sistemas desse tipo requerem algoritmos especiais ou componentes específicos, que garantam seu funcionamento correto mesmo com a presença de falhas. De acordo com Bisset *et al* [8], sistemas tolerantes a desastres vão além da tolerância a falhas, esses sistemas exigem que a perda de tempo de computação ocasionado por um desastre causado pelo usuário ou que tenho ocorrido naturalmente, não afete em hipótese alguma a disponibilidade do sistema, integridade dos dados ou perda dos mesmos.

2.2 Tolerância a Falhas

Termo que foi introduzido por Avizienis [3], apesar de estratégias de construção de sistemas mais confiáveis já ser pauta usada na construção dos primeiros computadores [35]. Mesmo envolvendo técnicas e estratégias tão antigas, a tolerância a falhas ainda não se tornou uma preocupação de projetistas e usuários, deixando suas áreas de aplicação geralmente restritas a sistemas críticos.

Dentre muitas classificações para as técnicas de tolerância a falhas a mais comum é

a classificação em quatro fases de aplicação de Anderson & Lee [28], são elas: detecção; confinamento; recuperação e tratamento. A partir deste ponto as informações foram retiradas de Weber [62] e complementadas com base em Anderson & Lee [28].

Essas fases podem ser interpretadas como uma sequência complementar de atividades, que são executadas após a ocorrência de uma ou mais falhas. A primeira fase é a detecção de um erro, todo erro advém de uma falha, antes da sua manifestação como erro, a falha está latente e não pode ser detectada. Eventualmente uma falha pode permanecer no sistema durante toda sua vida útil.

A latência entre uma falha se tornar um erro pode fazer com que dados inválidos se alastrem pelo sistema. O confinamento deve estabelecer limites para a propagação do dano. Durante o projeto devem ser previstas e implementadas restrições ao fluxo de informações, para evitar fluxos acidentais e estabelecer interfaces de verificação para detecção de erros.

A recuperação de erros ocorre após a detecção de erros, e envolve a troca do estado atual incorreto para um estado livre de falhas. Existem dois tipos principais de recuperação, por retorno e por avanço. A recuperação por avanço consiste em corrigir os componentes danificados e levar o sistema a um estado novo, para este tipo, danos devem ser previstos no projeto. Na recuperação por retorno, os componentes do sistema errôneo serão recuperados através da restauração de um estado correto, salvo anteriormente, esse método tem como desvantagem um alto custo para armazenar os estados do sistema.

A última fase envolve a prevenção do sistema a partir da correção da falha, ou seja, após recuperar-se do erro é preciso localizar a falha que o gerou. Nessa fase geralmente é considerada a hipótese de falha única, ou seja, uma única falha ocorrendo a cada vez. A localização da falha é realizada em duas etapas: localização grosseira e rápida (módulo ou subsistema); e localização fina, mais demorada, onde o componente falho é determinado. Após essa etapa é feito o reparo da falha, por substituição ou reparo do componente falho. Tanto o reparo quanto a localização da falha podem ser feitos de modo automático ou manual.

2.2.1 Recuperação por Retorno

A recuperação por retorno é uma das das técnicas de recuperação presentes na teoria de tolerância a falhas, esta técnica tem um propósito comum ao modelo apresentado neste trabalho, basear-se em estados anteriores para retornar o sistema a um estado livre de falhas. Posteriormente poderão ser tomadas medidas para evitar a falha da qual o sistema se recuperou. A seguir veremos um pouco a respeito desta técnica, identificando suas qualidades e pontos problemáticos.

Quando um erro ocorre, técnicas de recuperação por retorno conduzem o sistema a

um estado anterior correto, livre de falhas. Por meio de uma restauração ou *rollback* para um estado do sistema salvo anteriormente. Estes estados por sua vez são guardados em pontos de recuperação pré-determinados. Os de recuperação são estabelecidos através do armazenamento do estado do sistema em um determinado momento.

Quando há incidência de um erro o estado do sistema geralmente é restaurado com base no último ponto de restauração, e a operação do sistema é então retomada, seguindo um outro caminho para não se deparar com o mesmo erro. Segundo Jalote[22], se uma falha ocorrer após ser criado um ponto de restauração, este estado salvo é considerado livre de falhas.

Existe uma série de vantagens na utilização da recuperação por retorno, dentre as quais se destacam:

- Poder de recuperação de estados que não podem ser previstos no projeto, salvo se não estiverem afetando o sistema de recuperação [Randell & Xu *apud* Pullum[42]].
- O único conhecimento necessário para utilizar recuperação por retorno é que o último estado em questão deve ser livre de falhas [Levi *apud* Pullum[42]].
- A recuperação por retorno utiliza um esquema geral de recuperação, seu padrão uniforme é de detecção e recuperação de erro, é independente da aplicação e não se altera de programa para programa [Milli *apud* Pullum[42]].
- Não requer conhecimento dos erros no estado do sistema, pode ser apenas executado [Randell & Xu *apud* Pullum[42]].

Há também algumas desvantagens na sua utilização:

- O custo para restaurar um processo ou sistema a um estado anterior pode ser alto, levando a perda de desempenho [Zanardo *et al* [68]].
- A utilização da recuperação por retorno normalmente requer parada temporária do sistema [51].

De acordo com Pullum[42], a técnica de recuperação por retorno é a mais utilizada para recuperação de erros em tolerância a falhas de software, apesar de apresentar um considerável consumo de recursos computacionais – principalmente memória e tempo de processamento.

2.3 Recuperação e Restauração de Sistemas Operacionais

A finalidade de um sistema operacional prover de mecanismos de restauração é a possibilidade de devolver ao sistema um estado operacional sem haver a necessidade de uma reinstalação completa e sem comprometer arquivos de dados.

Para isto, devem ser criados pontos de restauração, que normalmente são criados em eventos específicos, como instalação de aplicativos, atualizações do sistema, instalações de novos drivers e instalação de novos dispositivos, ou ainda podem ser feitos de forma manual pelo administrador do sistema ou por agendamento de tal serviço.

Os pontos de restauração podem ser armazenados localmente ou em alguma mídia externa ao sistema. De certo modo, o número de estados salvos dentro do sistema local pode ser bem limitado de acordo com o espaço útil do sistema. Uma forma de contornar este problema é somente salvar arquivos que realmente importem para o funcionamento do sistema, deixando de lado, por exemplo, arquivos de usuários. É difícil imaginar uma situação em que arquivos do usuário sejam fundamentais para o funcionamento pleno do sistema.

Para que seja possível a restauração do sistema, mesmo que qualquer tipo de erro que tenha acontecido, o sistema deve dispor de um módulo adicional, independente, que possa alcançar um grau de funcionalidade mínimo para executar o sistema de recuperação e obter acesso aos dados do sistema principal.

Existem ferramentas deste tipo disponíveis, porém são todas ferramentas comerciais, dentre elas podemos citar: *Norton GoBack*^{TM}; *RollBack Rx Professional*; *Microsoft System Restore*; *TimeMachine*. Estes aplicativos permitem desfazer mudanças que tenham causado problemas no sistema, restaurando-o para um estado "saudável". Inclusive é possível restaurar alguns arquivos que foram eventualmente excluídos ou modificados. Vale ressaltar que estes sistemas são restritos a sistemas operacionais *Microsoft* e *MacOS*.

2.4 Restauração, Segurança e Administração de Sistemas

Frequentemente, instalações e configurações de servidores, são feitas por administradores de sistemas que trabalham sob extrema pressão. Especialmente quando há uma situação de recuperação de algum desastre. Além disso, no curso de vida normal da rede, administradores terão problemas e de tempos em tempos a configuração das máquinas terá de ser alterada[33].

Segundo o mesmo autor, administradores tem confiáveis sistemas de *backup* ou que

guardam *snapshots* para recuperar arquivos antigos ou recuperar erros. Anos atrás era uma solução plausível quando se tinha uma rede com menos de uma dezena computadores, em redes atuais onde temos estações de trabalho e servidores provendo múltiplos serviços, é necessária uma solução mais escalável e flexível.

Existem várias ferramentas para automatização de instalação de sistemas operacionais, que normalmente replicam o estado de uma máquina "saudável" para outras. Isso é útil quando a rede possui um padrão de máquinas, ou seja, o hardware é idêntico. Esta abordagem resolve apenas uma parte do problema. Além disso quando é feita uma operação destas, boa parte da banda da rede é ocupada. Normalmente os computadores da rede deverão passar por muitas mudanças de configuração, atualizações de sistema, de aplicativos e de segurança. O que pode gerar mais situações onde sistema de recuperação precisaria entrar em ação, consumindo recursos da rede.

Segundo Goel *et al.*[15], quando o sistema é comprometido por um ataque , uma das tarefas em que mais ocorrem erros e consomem tempo é a recuperação dos danos. A recuperação é feita após um ataque descoberto e envolve muitos passos: instalação da imagem do sistema e suas aplicações; instalar atualizações para conter a vulnerabilidade que resultou no ataque e recuperar dados de usuários corrompidos pelo ataque. Cada um desses passos é tedioso e demorado.

Paula[39], descrevendo uma arquitetura de segurança, destaca a restauração do sistema como um fundamental procedimento para tornar o sistema novamente seguro, principalmente para eliminar modificações em binários e arquivos de configuração ocorridos durante alguma intrusão. Além disso se houver maneira de restaurar as mudanças passo-a-passo no sistema, é possível identificar todos os passos do atacante. Uma análise forense desse tipo seria de grande interesse para administradores do sistema.

Como em Administração de Sistemas e Segurança há a necessidade de se recuperar o sistema podem ser utilizados os conceitos de recuperação que existem dentro de tolerância a falhas para especificar e modelar esses sistemas. O que traria benefícios, já que se trata de uma área de pesquisa tão antiga.

2.5 Conclusão

A resiliência está presente em várias áreas de pesquisa, apesar de possuir diferentes significados em cada uma delas é atributo desejável em todas. Na ciência da Computação várias áreas tentam particularmente representar resiliência, porém todas elas tem um cunho comum de recuperação. A resiliência está presente em vários tipos de software, aplicá-la em sistemas complexos como sistemas operacionais é sem dúvida um passo importante para a economia de recursos humanos.

Mesmo em algumas áreas em que o termo resiliência não é usado, existem metodolo-

gias de recuperação de sistema para um estado anterior previamente salvo. Certamente analisar as peculiaridades de cada um dos estudos dessas áreas é um ponto chave para conquistar maturidade no projeto de sistemas resilientes.

A recuperação por retorno é importante para segurança e administração de sistemas. Na administração, métodos de recuperação são úteis para evitar passos tediosos da recuperação manual, isto implica em economia de horas de trabalho. Em segurança, mecanismos de recuperação trariam o sistema novamente à um estado livre de modificações feitas por um ataque, assim podem-se implantar rapidamente novas políticas de segurança ou ainda atualizações no sistema, em ambos o resultado seria ganho em disponibilidade do sistema.

Capítulo 3

Sistema de Controle de Versão de Arquivos

Controle de Versão é a arte de gerenciar mudanças em informações. Na Engenharia de Software, Controle de Versão refere-se a um conjunto de práticas cujo objetivo principal é manter um histórico das modificações efetuadas em um determinado arquivo. Esta prática tem sido útil para equipes de programadores, que gastam seu tempo fazendo pequenas mudanças em muitos arquivos de código.

Ferramentas que proveem esta funcionalidade podem ser interpretadas como ferramentas de apoio ao processo de desenvolvimento de software, por atender diversas necessidades específicas dos desenvolvedores. Com ferramentas desse tipo, pode-se, por exemplo, manter um registro com todas as mudanças sofridas no código fonte ao longo do tempo, possibilitando a recuperação dos estados de arquivos em algum instante passado.

O objetivo deste capítulo é mostrar superficialmente o funcionamento dos sistemas de Controle de Versão—SCV—bem como uma revisão sobre funcionalidades, qualidades e defeitos específicos aos SCVs mais conhecidos atualmente, além de um breve estudo comparativo que auxiliou a escolha da ferramenta utilizada neste trabalho.

3.1 Visão Geral

Sistemas de Controle de Versão são ferramentas utilizadas para automatizar o processo de Controle de Versão sob documentos, códigos fonte e quaisquer arquivos utilizados dentro de um determinado ambiente. Com um SCV é possível gerenciar o histórico de um arquivo e obter simultaneamente duas ou mais versões do mesmo. Além disso, um SCV permite que duas pessoas trabalhem ao mesmo tempo em um documento, aumentando a produtividade conjunta. Para esta finalidade, contém uma série de ferramentas que auxiliam o gerenciamento de conflitos introduzidos pela edição mútua do arquivo.

O funcionamento de um SCV consiste basicamente em manter um repositório com os arquivos que devem ser controlados, além disso, no repositório também são armazenados os arquivos de controle e *logs*. Para otimizar o espaço em disco ocupado pelas diferentes versões dos arquivos, a maior parte dos SCVs utiliza o método de compressão *Delta* [7][45][55], que consiste em armazenar apenas a diferença entre versões sucessivas de um arquivo. O princípio geral da compressão *Delta* é aplicar sucessivamente as diferenças armazenadas sobre a versão inicial do arquivo, deste modo, partindo da versão inicial pode-se obter a mais recente ou qualquer outra intermediária[53].

A maioria dos SCVs trabalha com o modelo *check-in/check-out*[13] que tem base em um Controle de Versão individual para cada arquivo. Os arquivos versionados ficam armazenados dentro de um repositório e o usuário não pode interagir diretamente com eles. Primeiramente o usuário deve fazer um *check-out*—fazer a retirada—dos arquivos que ele deseja editar, ou seja, copiar os arquivos para um diretório de trabalho. Futuramente os arquivos modificados terão novas versões a partir do momento em que o usuário efetuar seu *check-in*—dar entrada—no repositório.

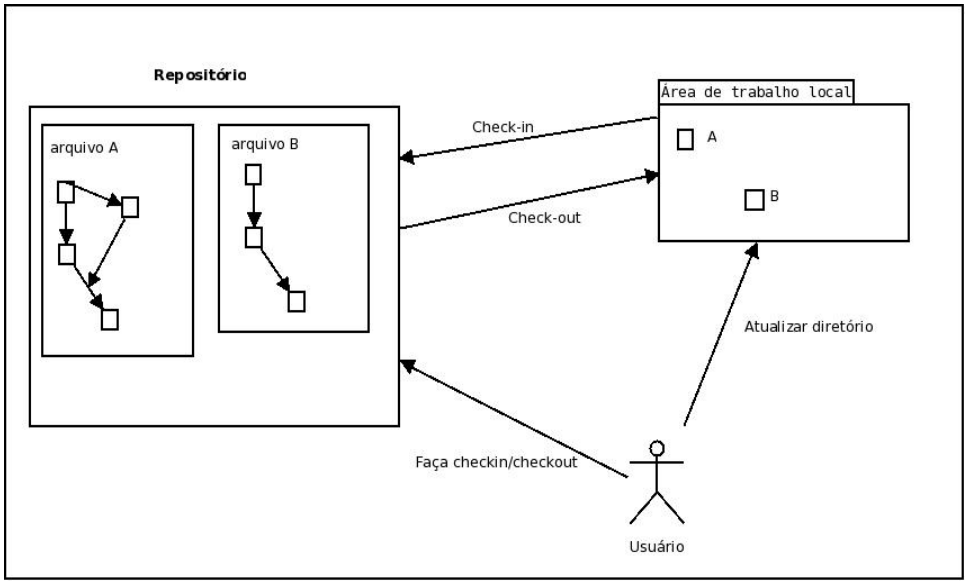


Figura 3.1: Exemplo do fluxo de dados do modelo check-in/check-out.

O usuário não age diretamente com os arquivos do repositório, apenas lhe dá o comando de *check-in/out* informando o local de cópia, conforme ilustra a Figura 3.1. A partir deste ponto, o controle de acesso dos arquivos na cópia de trabalho é exclusivo do usuário. O repositório também possui mecanismos para controlar a edição simultânea de um arquivo, normalmente possui também, ferramentas de auxílio ao processo de solução de possíveis conflitos de edição gerados pela edição simultânea.

3.2 Sistemas de Controle de Versão Centralizado e Descentralizado

Esta classificação advém da maneira em que é conduzido o armazenamento dos dados versionados, centralizada e descentralizada—será utilizado a partir daqui DSCV para o último. Nos SCVs centralizados as alterações sempre serão validadas e armazenadas em um servidor central único, ao passo que nos descentralizados cada desenvolvedor trabalha diretamente no seu repositório[14]. Pode-se dizer que os sistemas descentralizados usam um comunicação "ponto-a-ponto" e os centralizados utilizam comunicação cliente-servidor.

Os DSCVs trabalham com alterações locais, ou seja, cada usuário terá uma cópia fiel do repositório, este é seu princípio básico de funcionamento. O que é fundamentalmente diferente nos sistemas centralizados, que requerem que todas as alterações ocorram através de ligação com um servidor central.

Essa característica permite que os desenvolvedores utilizando um DSCV trabalhem em qualquer lugar, mesmo sem internet não perdem o Controle de Versão. Além disso, o modelo descentralizado torna-se mais rápido, pois a maioria das operações é executada localmente e posteriormente o usuário somente manda os dados resultantes da operação aos outros nós.

Os SCVs centralizados, apesar da arquitetura mais simples possuem algumas características inexistentes nos DSCVs. Naturalmente a maioria delas é resultado da simplicidade e facilidade imposta pelo controle de um repositório único. Neste modelo, cada desenvolvedor mantém uma cópia dos arquivos contidos no repositório, além de alguns metadados de controle que serão usados para cruzar as informações presentes na cópia e no repositório, tornando públicas as alterações através de um processo conhecido como *commit*. Além disso, é importante ressaltar o melhor controle de acesso aos arquivos do repositório, visto que as informações estão contidas em um único local.

A simplicidade que traz benefícios para este modelo também envolve alguns transtornos. Principalmente quando este se depara com ambientes complexos, onde desenvolvedores estão espalhados geograficamente e não há comunicação estável e permanente com o repositório. No SCVs centralizados os usuários ficam extremamente dependentes do repositório, de modo que um desenvolvedor não consegue sequer solicitar um histórico do arquivo, de qualquer forma ele consegue editar os arquivos.

Mesmo com a vantagem do controle de acesso por possuir um só repositório, um problema pode surgir quando não há metodologia confiável de *backup* do mesmo. Vale ressaltar que o método de backup deve ser escolhido de acordo com o SCV usado, pois alguns podem um utilizar um banco de dados para manter suas informações, enquanto que outros podem utilizar uma árvore de arquivos logicamente ordenada[53].

Contrariamente aos SCVs centralizados, os DSCVs tem uma arquitetura muito mais

complexa e complicada de entender. Sob o ponto de vista operacional não há muita diferença, usuários fazem *commits* de suas modificações, verificam seu histórico etc. A diferença crucial é o fato de não haver um repositório, mas sim vários e um para cada usuário, dependendo da ferramenta utilizada. Deste modo, pode-se fazer com que repositórios filhos de um mesmo repositório pai compartilhe as mudanças entre si.

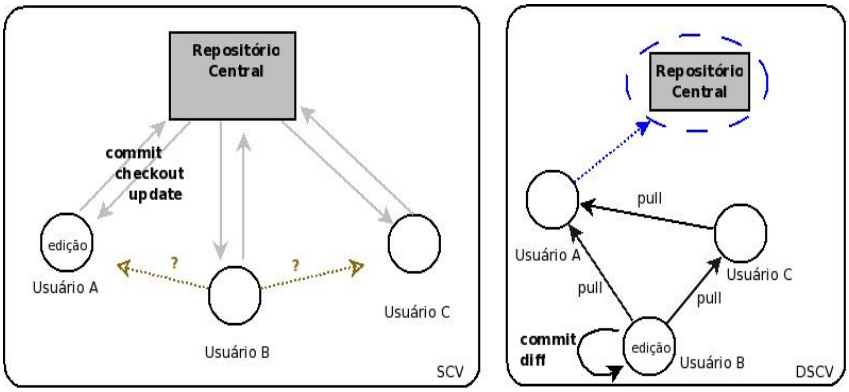


Figura 3.2: Exemplo de ambiente SCV e DSCV

Para efetuar alterações um usuário precisa primeiro efetuar um *commit* localmente e somente depois essas mudanças serão replicadas ao repositório pai e aos seus irmãos—Figura 3.2, existirem. Seguindo o mesmo raciocínio, ao receber este *commit* o pai pode transmití-lo ao seu pai e assim sucessivamente, tornando as modificações públicas para todos os outros membros do conjunto de repositórios. Utilizando-se um DSCV descentralizado pode-se ter acesso as informações de um repositório pai mesmo que não havendo conexão, porém, as informações serão referentes à última sincronização feita com um repositório pai ou irmão.

Da mesma maneira que no compartilhamento de arquivo por redes *peer-to-peer*, em um DSCV é complicado obter total controle de acesso dos dados, embora muitos dos DSCVs tenham implementado funcionalidades para resolver este problema.

3.3 RCS – *Revision Control System*

RCS teve o início do seu desenvolvimento por Walter Tichy da Universidade de Purdue no ano de 1980 e consiste em um conjunto de comandos *UNIX* que auxiliam um ou mais desenvolvedores em suas tarefas [55]. RCS é capaz de gerenciar a evolução de arquivos texto e configurações geradas a partir deles.

O compartilhamento e o gerenciamento de criação de novas versões é feito através da abordagem *check-out/check-in*[13], RCS contém mecanismos para que dois programadores não possam modificar simultaneamente o mesmo arquivo. A programação paralela é

simulada com a utilização de ramos, onde cada programador altera o arquivo da maneira desejada e após suas alterações serem confirmadas, o sistema faz a junção entre os ramos para criar um arquivo único[13].

RCS coloca indicadores nos arquivos para mostrar que eles estão em modo *check-out*, assim ficam bloqueados de forma que nenhum usuário consiga fazer modificações neles antes que seja feito seu respectivo *check-in*—modo bloqueante. O usuário que efetuou o *check-out*, por sua vez, deve certificar-se que suas alterações geram um código compilável e no momento que ele efetuar o *check-in* o RCS identifica quais arquivos foram modificados e cria uma nova revisão destes no repositório, mantendo os antigos no seu estado anterior.

3.4 CVS – *Concurrent Versions System*

Em estudos posteriores foram identificados dois problemas sérios no RCS: a metodologia de bloqueio dos arquivos impossibilitava desenvolvimento em paralelo e a ausência de suporte à hierarquia de diretórios dentro do repositório. Estas limitações incentivaram o desenvolvimento de um novo sistema, chamado Concurrent Versions System[7]. O CVS é uma camada construída entre o RCS e o usuário. Isso manteve as funcionalidades e a metodologia de armazenamento das versões dos arquivos do RCS.

O modelo utilizado pelo CVS também foi o de *check-in/check-out*, porém o comando *check-in* foi substituído pelo comando *commit*. Com esse comando é feita uma verificação diferente quando os arquivos vão retornar ao repositório. Antes de efetivar um *commit* o sistema verifica se a versão que foi dada o *check-out* é a mesma que está no repositório. Caso isso seja positivo é feito o *commit* normalmente, caso contrário é feita uma junção considerando os três arquivos, a versão atual do arquivo no repositório, o arquivo que foi feito *check-out* e o arquivo que o usuário quer atualizar no repositório. Essa abordagem é denominada *copy-modify-merge*[7].

3.5 SVN - *SubVersion*

SubVersion, lançado em 2004, foi criado com o objetivo principal de ser uma versão melhorada do CVS. Atualmente é desenvolvido por ex-desenvolvedores do CVS, contando então com a experiência de programadores responsáveis por uma das ferramentas de controle de versão mais conhecidas e utilizadas no mundo. O projeto Subversion começou em 2001, financiado pela empresas CollabNet e Red Hat, a qual disponibilizou um de seus funcionários por tempo indefinido para o projeto. Subversion levou em torno de 14 meses para ser capaz de se gerenciar os próprios arquivos do seu projeto.

Segundo Collin-Sussman *et al*[9], SubVersion adiciona algumas funcionalidades extras

comparado com CVS, por exemplo:

- Commits atômicos: se por algum motivo o servidor sofrer uma parada, ou a comunicação com o cliente for interrompida, o repositório continuará em estado confiável;
- Versionamento de diretórios: além de permitir hierarquia de diretórios dentro do repositório, subversion é capaz de manter informações das operações nos mesmos, deste modo é possível reorganizar uma árvore de dados sem perder as informações.
- Suporte a diversas camadas de rede: o CVS não foi projetado levando em conta as diversas arquiteturas de rede existentes atualmente;
- Uso eficiente da rede, evitando envio de dados redundantes entre dois pontos.

3.6 Bazaar-NG

Este controlador de versão permite tanto a topologia de servidor central quanto a de servidores distribuídos. O projeto surgiu com objetivo de sanar alguns problemas de arquitetura de outro projeto o *GNU Arch*, principalmente com relação a facilidade de uso. É desenvolvido na linguagem Python, o que significa que ele pode ser facilmente portado para qualquer plataforma. Uma das políticas problemáticas do Bazaar é que para cada projeto deve ser criado um novo repositório, isso gera um maior uso de disco para armazenar mais de um projeto. O desempenho ainda deixa a desejar e algumas operações são realmente um pouco lentas, principalmente se estiverem relacionadas com arquivos binários. Bazaar-NG não é tão maduro quanto outros projetos, porém recebe fundos da Canonical—que comercialmente suporta os custos de produção do Ubuntu—desde então vem crescendo rapidamente[64].

3.7 Mercurial-HG

Ferramenta de controle versão distribuída que oferece uma solução multi-plataforma—*Mac OSX*,*Microsoft Windows* e *Linux*. Implementado principalmente em *Python*, porém possui alguns binários escritos em C. É especificamente um aplicativo de linha de comando, onde são inseridos argumentos para o programa *hg*, em referência ao elemento químico. O projetista chefe e criador do Mercurial é Matt Mackall, e o código fonte está disponível sobe os termos da GPL2, que qualifica Mercurial como software Livre.

O primeiro anúncio sobre Mercurial foi feito em 19 de abril de 2005 por Mackal, motivado pela notícia de mês anterior de que um outro grande controlador perderia sua versão livre, o *BitKeeper*. *BitKeeper* era o controle de versões oficial do *Kernel Linux*,

isto fez com que duas frentes de desenvolvimento trabalhassem na tentativa de criar uma alternativa. Posteriormente GIT foi escolhido para ser o novo controlador de versão na árvore de código do *kernel*, porém agora Mercurial é usado por muitos outros projetos.

3.8 GIT

Git é um sistema de Controle de Versão distribuído criado por Linus Torvalds, tem como foco principal a velocidade, eficiência e usabilidade, mesmo quando aplicado a grandes projetos [65]. Dentre suas principais características podemos destacar:

- Rápida e eficiente junção de duas linhas de desenvolvimento, aliada a poderosas ferramentas de visualização e navegação dentro de um histórico;
- Eficiente suporte a grandes projetos, conta com um sistema de compressão de arquivos extremamente poderoso, além de garantir uma melhor organização e indexação dos arquivos do repositório;
- Histórico do projeto cifrado, o que garante que não se pode modificar manualmente o histórico do projeto.

3.9 Comparativo

RCS e posteriormente CVS demostraram pioneirismo no Controle de Versão, porém não possuem algumas funcionalidades importantes para este projeto. Principalmente quanto ao armazenamento de dados. O CVS foi projetado para armazenar apenas texto, para trabalhar com arquivos binários são necessários comandos não triviais. Além disso não é possível guardar os metadados de cada arquivo, ele trata somente do conteúdo dos mesmos. Outro aspecto relevante é a atomicidade das operações, tanto o RCS quanto o CVS não possuem controle para *commits* atômicos.

Baseando-se principalmente em quantidade de documentação e popularidade quatro ferramentas destacaram-se, SubVersion, GIT, Bazaar e Mercurial-HG. A Tabela 3.1 mostra alguns dos usuários mais notáveis destas ferramentas.

Através de comparação entre as funcionalidades das ferramentas—Figura 3.3— é possível analisar os pontos negativos e positivos de cada uma, lembrando sempre que a comparação leva em consideração a arquitetura proposta neste estudo. Primeiramente devem ser identificadas como foram tratadas as operações básicas do repositório de cada sistema e em que ponto cada uma destas atende melhor as necessidades da ferramenta proposta.

Ferramenta	**Usuários**
SubVersion	ASF, SourceForge, FreeBSD, Google Code, KDE, GCC, Ruby, Mono, PuTTY, Zope, Xiph, GnuPG, CUPS, Wireshark, TWiki, Django [1]
GIT	Linux kernel, GNOME, Perl 5, X.Org, Cairo, Qt Software, Samba, OpenEmbedded, Ruby on Rails, Wine, Fluxbox, Openbox, Compiz Fusion, XCB, ELinks, XMMS2, e2fsprogs, GNU Core Utilities [2]
Bazaar	Ubuntu, Launchpad, KatchTV, MySQL [3]
Mercurial-HG	Mozilla, OpenSolaris, OpenOffice, NetBeans [4]

Tabela 3.1: Usuários mais notáveis.

Git e Bazaar vão um pouco além do *commit* atômico, ao caso de uma operação ser interrompida, um buffer guarda o que foi feito até aquele ponto, para posteriormente continuar a operação de onde ela parou. Isso pode ser chamado de estado intermediário.

Outra característica interessante de um SCV é seu suporte à renomeação de arquivos e diretórios, todas as quatro ferramentas avaliadas possuem esta característica. Git e Mercurial vão um pouco além, pois guardam no histório do arquivo a renomeação, ou seja, são capazes de juntar histórico das operações do arquivo antes de depois de ser renomeado, SubVersion e Bazaar agem como se um arquivo fosse excluído e um novo criado.

Operação	**Bazaar**	**GIT**	**SubVersion**	**Mercurial**
Commits Atômicos	Sim	Sim	Sim	Sim
Renomear arquivos e diretórios	Sim	Sim	Sim	Sim
***Merge* inteligente**	Sim	Sim	Não	Sim
Cópia de arquivos e diretórios	Não	Sim	Sim	Sim
Permissões dentro do repositório	Básico	Sim	Sim	Sim

Tabela 3.2: Comparação entre operações.

Merge é o ato de fundir automaticamente duas ou mais versões de um arquivo automaticamente, SCVs que possuem *merge* inteligente conseguem lidar com situações em que um desenvolvedor tenta fazer um *commit* de um arquivo ou diretório que foi renomeado por outra pessoa. Dentre os quatro, apenas SubVersion não possui esta operação, portanto situações como esta necessitam de intervenção manual.

[1] Lista obtida em: http://subversion.tigris.org/testimonials.html
[2] Lista obtida em: http://git.or.cz/gitwiki/GitProjects
[3] Lista obtida em: http://bazaar-vcs.org/WhoUsesBzr
[4] http://mercurial.selenic.com/wiki/ProjectsUsingMercurial

Permissões internas no repositório são desejáveis quando partes do código devem ser mantidas em segredo, para este projeto em especial, esta característica é importante pois reflete em quesitos de segurança das informações contidas no repositório. No Bazaar algumas políticas básicas de permissão podem ser criadas através de um *script* externo, porém no site do desenvolvedor já está prevista a implementação desta característica. SubVersion e GIT por sua vez possuem controle de permissões completo.

GIT, Mercurial e Bazaar demonstram maior facilidade de uso que SubVersion, por exemplo, para criar um repositório basta realizar um comando dentro de um diretório e todos os arquivos e diretórios estarão automaticamente incluídos no Controle de Versão. A criação de um repositório no SubVersion é mais complicada, primeiramente deve-se criar o repositório, depois importar os arquivos para dentro dele, além disso existem algumas políticas de diretório dentro do repositório SubVersion, resta ao usuário escolher e usar o comando correto para seguir a desejada. Simplesmente iniciar o versionamento dentro de um diretório torna transparente a estrutura dos arquivos versionados, não deixando a difícil tarefa para usuário de montar a hierarquia do projeto versionado, ele pode apenas seguir a estrutura de seu diretório.

Os dados de controle GIT são todos relacionados com *hashes* dos conteúdos dos arquivos, na realidade para o GIT não existem nomes de arquivos e sim seus *hashes*, isso garante a integridade do repositório, pois toda e qualquer mudança pode ser identificada pelo hash. Esta é uma característica importante para este projeto pois é fundamental o controle de intrusão sobre os arquivos de controle, assim evita-se que o atacante possa modificar estes arquivos sem que a ferramenta saiba.

SubVersion apresenta menos eficiência para alguns comandos, pois precisa contactar o servidor através da rede para realizá-los, enquanto que os outros dois mantém cópias das informações de controle do repositório e não somente os arquivos. Existem vários tipos de comparações quanto ao desempenho dos SCVs, representaremos os mais relevantes a seguir.

3.9.1 Git e SubVersion

Westerbaan[63] em 2008 através de um estudo de caso de sua empresa, realizou alguns testes de eficácia quanto ao espaço gasto para armazenar o repositório.

Dentro de sua empresa eles possuiam um total aproximado de 115 projetos, armazenados em repositórios em GIT e SubVersion, ambos sincronizados.A Figura 3.3 representa os repositórios plotados de acordo com seu tamanho, SubVersion horizontalmente e GIT verticalmente. Cada X repesenta o cruzamento dos dois tamanhos encontrados para cada projeto.

Para facilitar a compreensão do gráfico foram adicionadas três linhas, a primeira

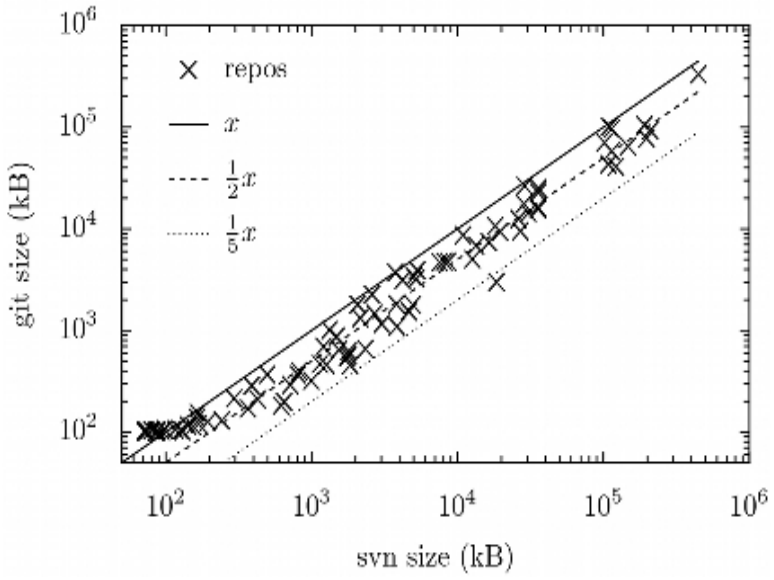


Figura 3.3: Comparativo dos tamanhos de repositório obtidos.

linha—preta contínua—indica os pontos em que os tamanhos seriam iguais, a segunda—tracejada—indica a posição em que o repositório SubVersion seria duas vezes maior que o repositório GIT, finalmente a terceira—pontilhada—indica a posição onde SubVersion seria cinco vezes mais ineficiente. Todos os repositórios onde GIT foi pior são menores do que 100 Kb, projetos em que GIT foi muito mais eficiente são os projetos de tamanho médio—10-100 Mb. Para o restante GIT foi aproximadamente 20% mais eficiente do que SubVersion

3.9.2 Git e Bazaar

Auvray[2] em 2008 realizou alguns testes entre três SCVs para demonstrar a melhor opção de uso para sua comunidade. Para isto ele criou cópias do repositório de um navegador de código aberto, o Mozilla Firefox[4]. Num total de 30000 arquivos aproximadamente e 70853 alterações. Os testes foram feitos numa máquina com processador AMD Athlon(tm) 64 3500+, 1 Gb de memória RAM rodando Linux Kubuntu 6.10 Edgy x86_64 em um sistema de arquivos ext3. Cada comando foi executado oito vezes, o melhor e o pior resultados foram descartados, além disso foi considerado o modelo de repositório local, ou seja, nenhuma sobrecarga de comunicação foi considerada.

O comando *status* mostra o estado atual do ambiente de trabalho, geralmente são identificados arquivos adicionados ao Controle de Versão e arquivos que foram alterados e ainda não houve efetivação dessas alterações. Já o comando *diff* mostra a diferença

[4]http://hg.mozilla.org/

Comando	Legenda	Bazaar (s)	GIT (s)
status	–	1,941	0,564
diff (todo o rep.)	op. 1	2.847	0.608
diff (dom/)	op. 2	2.037	0.116
diff (dom/src/)	op. 3	1.861	0.084
diff (dom/src/nsCOMClassinfo.cpp)	op. 4	2.135	0.085
clone	–	240.010	11.650

Tabela 3.3: Operações sobre o repositório do Firefox.

entre o arquivo—ou todos os arquivos de uma pasta—da cópia de trabalho e o arquivo do repositório. O comando *clone* consiste em clonar o repositório inteiro.

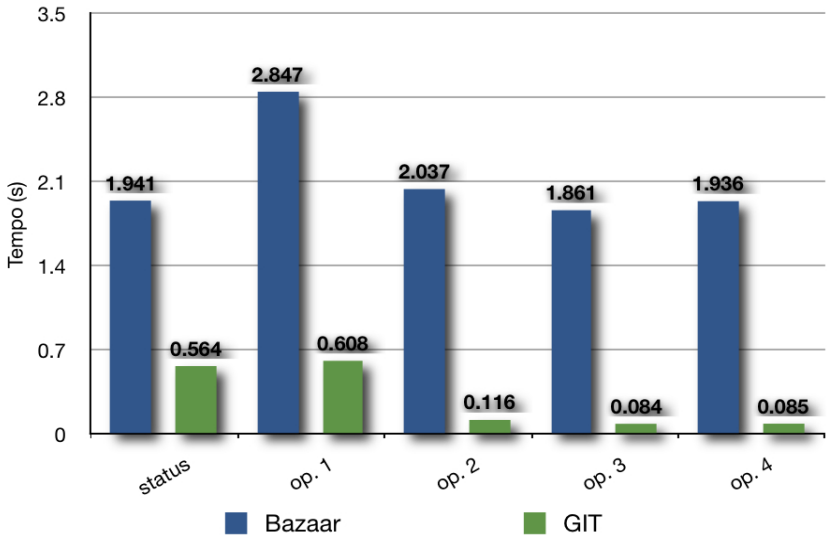


Figura 3.4: Comparativo de operações Auvray – 2008.

A Figura 3.4 mostra que GIT em todos os testes foi entre 70 a 95% mais rápido que Bazaar para todos os comandos. Jordan Mantha[30] um desenvolvedor do Ubuntu colocou em sua página pessoal uma série de testes envolvendo GIT e Bazaar, os resultados foram semelhantes aos de Auvray[2], onde GIT também se sobressaiu. Para fazer seus testes, Mantha criou repositórios contendo a árvore do kernel versão 2.6.0—que engloba todas as versões acima de 2.6.0 até aquela época—menos a versão 2.6.25.2. A partir deste ponto ele começou a executar as mesmas operações em cada repositório anotando o tempo total de cada uma delas.

Os tempos de inicialização do repositório foram muito parecidos e não foram considerados na tabela 3.4, Mantha adicionou entre os passos 2 e 3 a versão do kernel 2.6.25.2,

Operação	Comandos	Bazaar (s)	GIT (s)
1	Adicionar fonte	**4,852**	**7,267**
2	Commitar fonte	**43,968**	**10,263**
3	Diff (raiz)	**51,158**	**24,425**
4	Commit 2.6.24	**47,448**	**28,468**
5	Diff (sem alterações)	**47,027**	**2,343**
6	Status	**4,027**	**1,230**
7	Commit (pequena alteração)	**1,91**	**1,397**

Tabela 3.4: Operações sobre o repositório do Firefox.

isto gerou cerca de 300.000 novas linhas, criando assim um grande conjunto de mudanças em sua cópia de trabalho. Após isso foram executados comandos para efetivar a novas linhas ao repositório.

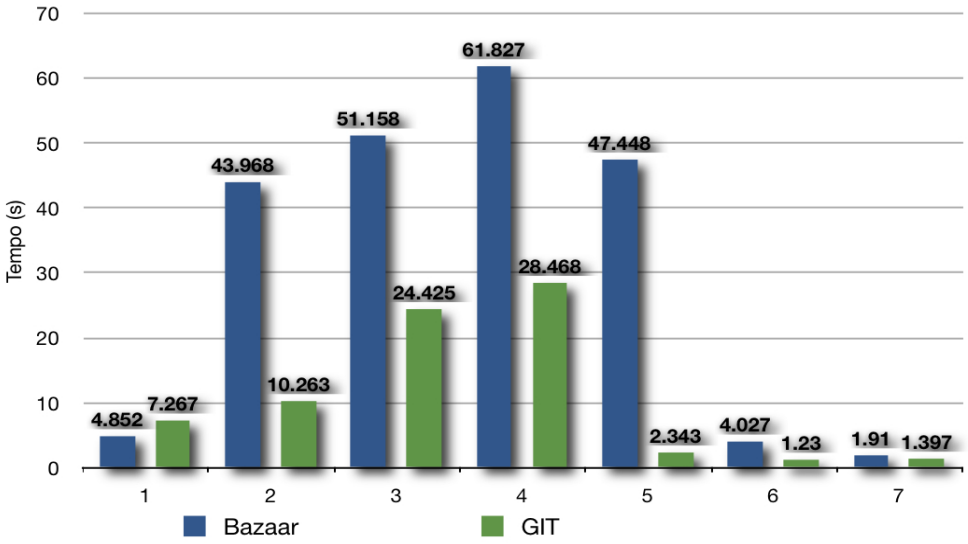


Figura 3.5: Comparativo de operações Mantha – Maio/2008.

Como mostra a Figura 3.5, GIT foi mais lento na primeira operação, porém extremamente mais rápido nas outras. Mantha explica que GIT não apenas marca um arquivo para ser adicionado ao Controle de Versão, ele cria em seus arquivos de controle estruturas complexas que fazem com que a sua eficiência seja maior em futuras operações no repositório. Visivelmente as duas ferramentas tem desempenho semelhante quando não há muita alteração no código. Percebe-se porém superioridade do GIT em até 50% quando foram realizadas grandes operações na cópia de trabalho.

3.9.3 Bazaar, SubVersion e GIT

Existem vários comparativos entre diferentes SCVs, porém nenhum que leve em consideração as quatro ferramentas analisadas nesta seção. Para este fim, realizamos um comparativo entre elas. Foi usado como base de testes um repositório SubVersion com aproximadamente 250Mb e mais de 8000 alterações. Tanto Bazaar quanto GIT possuem ferramentas para migrar um repositório SubVersion para seu formato, desta forma foi possível manter todos os arquivos e seus históricos para os três repositórios. Mercurial apresentou problemas na conversão da base de dados, portanto foi desconsiderado.

Os testes foram feitos numa máquina com processador AMD Turion 64 ML-37, 2 Gb de memória RAM, rodando Gentoo Linux x86_64 com kernel 2.6.30-r1 em um sistema de arquivos ext3. Antes de cada operação foram excluídos os caches de leitura de disco. A tabela 3.5 contém os resultados e a relação das operações realizadas.

Operação	Comandos	GIT (s)	SubVersion (s)	Bazaar (s)
1	status	**3,548**	**123,039**	**4,993**
2	diff	**0,789**	**88,349**	**4,738**
3	diff (alterações)	**1,002**	**91,385**	**4,830**
4	status (alterações)	**3,250**	**93,466**	**4,699**
5	commit	**3,832**	**28,132**	**9,826**

Tabela 3.5: Lista de Operações sobre GIT, SVN e Bazaar.

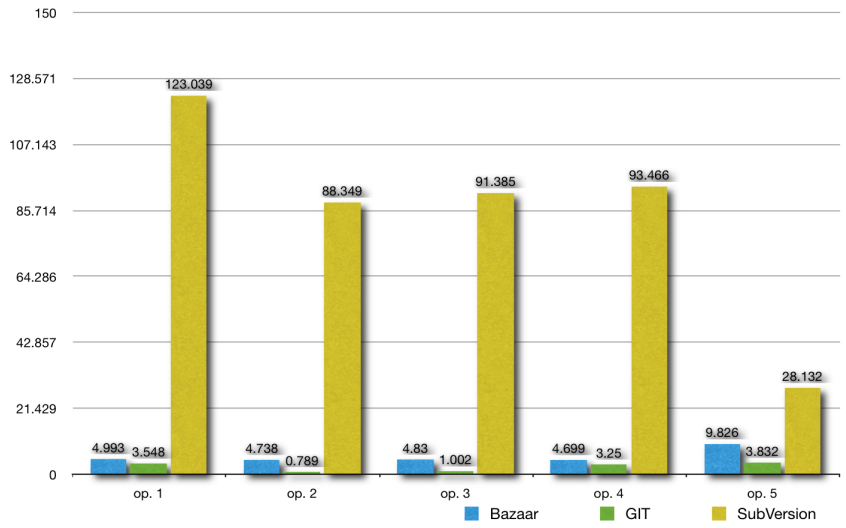


Figura 3.6: Comparativo de Operações sobre GIT, SVN e Bazaar.

As alterações feitas no passo 3 consistiram em concatenar dois arquivos de texto no

final de um terceiro e excluir os mesmos, resultando em um conjunto de alterações de aproximadamente 4000 linhas de texto.

SubVersion não cria grande quantidade de metadados para buscar por diferenças entre arquivos da cópia de trabalho e o repositório, isso faz com que a necessidade de buscas no repositório o mantenha distante da performance de GIT e Bazaar, que por sua vez são distribuídos e geram resultados com base em operações com metadados locais. A diferença entre os resultados apresentados por GIT e Bazaar seguem o padrão de Mantha[30], demonstrando superioridade do GIT em torno de 50% sobre as operações do Bazaar.

3.10 Conclusão

Sistemas de Controle de Versão são ferramentas eficazes para guardar o histórico de alteração de arquivos texto. Com o passar dos anos se tornaram ferramentas obrigatórias no ambiente de produção de *software*. A constante evolução deste tipo de ferramenta permitiu a criação de novas metodologias de uso, centralizando e descentralizando o acesso mútuo ao código armazenado.

Um grande quantidade de ferramentas de Controle de Versão podem ser obtidas comercialmente ou gratuitamente, porém como em toda categoria de software, algumas merecem destaque e conseguem conquistar grande parte do público alvo. Dentre as mais populares podemos destacar GIT, Bazaar e SubVersion.

Através de testes de eficiência e comparação de características de implementação foi constatado que GIT é escolha ideal para uso neste projeto, pois além de ser mais eficiente possui mecanismos de segurança desejáveis para este trabalho.

Capítulo 4

Sistemas de Detecção de Intrusão e Recuperação de Sistemas

Os computadores atuais estão infestados de vulnerabilidades, aplicações e sistemas operacionais possuem falhas em muitas de suas camadas. Sob ponto de vista dos paradigmas tradicionais de segurança, é possível eliminá-las através de extensível uso de métodos formais e melhores práticas de engenharia de software. Este ponto de vista baseia-se em: políticas de segurança explicitamente e corretamente especificadas; programas corretamente implementados, e que estes por sua vez possam ser corretamente configurados e utilizados.

Apesar de essas afirmações parecerem bastante coerentes, na prática nenhuma delas pode ser completamente garantida. Sistemas modernos não são estáticos, são intensamente modificados, novos programas são instalados, configurações são alteradas e o hardware pode ser atualizado. Isso torna a verificação formal impraticável. Sem verificação formal ferramentas de controle de acesso, *firewalls* e ferramentas de criptografia podem apresentar falhas, o que torna a implementação de políticas de segurança perfeitas impossível na prática.

Assumindo que não é possível implementar a prática de segurança perfeita, pode-se incrementar o nível de segurança através do uso de sistemas de detecção de intrusão—IDS. O abordagem de detecção de intrusão baseia-se na hipótese de que um sistema pode não ser completamente seguro, porém violações de segurança podem ser captadas pelo monitoramento e análise de seu comportamento.

Há diferentes maneiras de monitorar o comportamento de um sistema. Os dois primeiros tipos de IDS foram: o baseado em *host*—*Host-Based Intrusion Detection System (HIDS)*—e o baseado em rede—*Network-Based Intrusion Detection System (NIDS)*—a evolução fez com que surgissem novas alternativas como: o IDS híbrido—*Hybrid* IDS—que aproveita as melhores características das duas abordagens; finalmente, como com-

plemento dos HIDS e NIDS surgiu o *storage-based* IDS, que utiliza dados armazenados separadamente do sistema para realizar verificações mesmo que o *host* esteja comprometido.

Por outro lado, existem certas ferramentas que executam uma sistemática análise das configurações da máquina, com objetivo de encontrar falhas comuns que podem torná-la vulnerável à intrusões ou mau uso. Essas ferramentas são chamadas de sistemas de análise de configuração—*Configuration analysis systems*.

Neste capítulo serão discutidas técnicas e ferramentas que almejam a segurança do sistema, especialmente para ferramentas baseadas em host que fornecem base para recuperação, buscam por ameaças no sistema de arquivos e dão suporte forense para análise posterior ao ataque. Essa discussão servirá de base para a arquitetura que será apresentada no capítulo 5.

4.1 *HIDS – host-based intrusion detection system*

Um sistema de detecção de intrusão baseado em *host* monitora qualquer comportamento ou estado dinâmico de uma máquina. Para isto, utiliza em sua análise arquivos de *log* ou a saída de agentes de auditoria, podendo assim avaliar alterações no sistema de arquivos, modificações no controle de acesso de usuários, comportamento de processos do sistema, uso de recursos entre outros aspectos. Sumariamente pode também fazer inspeções na rede em que a máquina está ligada, sendo limitado aos pacotes que chegam à ela.

Uma outra característica presente em muitos *HIDS* é o *hash* do sistema de arquivos. Através de uma lista de arquivos importantes, o *HIDS* cria uma base de dados inicial com valores *hash* dos arquivos desejados, posteriormente recalcula os valores de hash e compara com os dados armazenados. Se alguma comparação entre os *hashes* não for bem sucedida alguma alteração ocorreu. Esta estratégia é bem eficiente para detectar instalação de *rootkits*, adição de novos usuários ou alteração de configurações.

Após as análises de dados o *HIDS* gera alertas ou coloca em *logs* todas as atividades suspeitas que ele encontrar. Dentre as vantagens de um HIDS pode-se destacar[43]:

- Associa programas e usuários com os seus efeitos em um sistema, podendo dizer quem e quando executou certa aplicação ou comando;
- Faz parte do alvo de ataque, assim consegue grande quantidade de informações durante o ataque;
- Confronta apenas ameaças direcionadas para a máquina onde reside—não captura todo o tráfego da rede;
- Tem acesso direto ao sistema, não dependendo de outro meio para acessar dados;

- Pode detectar ataques que ocorrem fisicamente no host;
- Não necessita de *hardware* adicional.

Dentre as desvantagens de um *HIDS* considera-se[48] :

- são difíceis de gerenciar pois necessitam de configuração e observação em todos os *hosts*, impactando em escalibilidade;
- É dependente do sistema operacional;
- Da mesma maneira que tem benefícios de acesso a informação por fazer parte do ambiente do *host*, é suscetível a ataques. Seus dados podem ser corrompidos ou até excluídos;
- Para haver correlação de dados de ataque a respeito de uma rede inteira, os dados de todos os *hosts* precisam ser concentrados e submetidos à alguma outra ferramenta que faça esta análise;
- Dependendo da profundidade da análise dos dados, ou do nível de coleta de informações a ferramenta pode causar diminuição de desempenho no *host* monitorado.

4.1.1 Verificadores de integridade

A integridade é uma das chaves principais para confiabilidade, ou seja, se um arquivo é íntegro, podemos confiar no seu conteúdo. Portanto, pode-se concluir que um sistema é íntegro se todos os seus arquivos também são. O sistema de arquivos está presente na maioria dos sistemas operacionais, nele estão contidos dados do usuário, arquivos executáveis, arquivos de configuração e informações de controle de acesso. Usualmente, também está contida a base de executáveis do próprio sistema operacional.

Existe uma classe de ataques denominada ataques de integridade, que tem o intuito de alterar arquivos sem detecção. O atacante se aproveita de um sistema vulnerável—muitas vezes temporariamente vulnerável—alterando arquivos para garantir sua reentrada ou para simplesmente comprometer o sistema.

Há muitos anos atrás os administradores de sistema deixaram de conseguir gerenciar mudanças indesejadas no sistemas de arquivos, esta necessidade aumentou a demanda de ferramentas com objetivo de assegurou a integridade dos arquivos de um sistema, para assim, buscar ameaças instaladas no sistema ou simplesmente guardar uma configuração padrão.

Surgiram ferramentas utilizando diferentes mecanismos para flagrar mudanças indesejadas no sistema de arquivos. Em geral são baseadas no *hash* de arquivos, além disso

existem algumas práticas de assinatura de arquivos que tem funcionamento semelhante. Algumas ferramentas e suas estratégias serão apresentadas a seguir:

Ferramentas de modo usuário, a história de Tripwire[25]

Uma das mais usadas ferramentas para detecção de alterações indesejáveis em arquivos. Tripwire baseia-se em um arquivo de configuração que define de quais arquivos ele irá manter informações, além de algumas políticas de análise para cada arquivo. Por exemplo, guardar informações de metadados para logs, já que são arquivos que estão em constante crescimento. Para este caso somente são analisados o tamanho do arquivo—ele deve crescer, nunca diminuir—e dados do controle de acesso ao arquivo. Em 2002 os criadores desta ferramenta abriram uma empresa e pararam de desenvolver ela abertamente, tornando-a proprietária. Tripwire foi pioneiro, porém após alguns anos após foi fechado para a comunidade.

Muitas ferramentas copiaram a metodologia de Tripwire e servem como solução aberta para o problema. Aide[44] é considerada a continuação de Tripwire na comunidade de software livre. Aide adicionou *daemons* de monitoramento com uso de *threads*, porém as características gerais de uso são as mesmas.

Um dos problemas críticos desta abordagem é a frequência das análises. Certamente alterações somente serão detectadas no momento que um arquivo alterado for examinado. O problema está na intermitência das verificações, ou seja, se um arquivo for infectado e este for utilizado antes da próxima análise. Dependendo da periodicidade das análise pode haver um grande tempo livre para ataques, que podem causar sérios danos aos dados do usuário.

Além disso, todas as ferramentas desta categoria realizam massivo uso do disco durante suas operações, o que pode causar profundo impacto no *host*. Isto ocorre devido a necessidade de se realizar *hashes* dos arquivos analisados, o que requer leitura de todo ele. Geralmente, as verificações nos servidores, são agendadas em horários onde o número de usuários dependentes do serviço é menor.

Samhain[27]

Uma ferramenta aberta multi-plataforma com intuito de centralizar a verificação de integridade. Foi projetada para monitorar centralizadamente múltiplos *hosts* com diferentes sistemas operacionais instalados. Também pode ser usada como aplicação simples em uma só maquina. Como diferencial, Samhain possui bases de dados e comunicação com os clientes criptografada, além de um modo "invisível" de operação, como contra-medida a possíveis ataques sobre seus arquivos e operação.

Osiris[37]

Osiris é um sistema de monitoramento de integridade que realiza análises periódicas de um ou mais *hosts*. Ele mantém *logs* das mudanças no sistema de arquivos, usuários, grupos e módulos de *kernel* instalados. Osiris pode ser configurado para enviar esses *logs* por *e-mail* e, se desejado, ele os mantém de modo a suportar análises forenses sobre eles.

Radmind[10]

Consiste em um conjunto de comandos *UNIX* e um gerenciador projetado para disparar verificações em várias outras máquinas cliente. Radmind utiliza Tripwire em seu núcleo, porém adiciona ferramentas para possibilitar a recuperação da máquina onde houveram alterações.

Soluções *In-Kernel*

Com objetivo de sanar o problema de detecção em tempo real surgiu a abordagem *In-kernel* de verificação de integridade. Que consiste verificar cada arquivo antes de permitir que o kernel acesse seus dados ou modificar as políticas de acesso para determinados arquivos. LIDS[41]—*Linux Intrusion Detection System*—-implementa outro controle de acesso de arquivos muito mais rígido, que impede as vezes até o próprio proprietário do arquivo de modificá-lo.

Uma abordagem similar é a do LSM[66]—*Linux Security Modules*—que consitem em um grande framework com chamadas de sistema, para que possam ser implementados vários mecanismos de segurança dentro do kernel. LSM não usa nenhuma política de segurança, mas permite adicionar ao kernel um completo sistema de segurança.

Por sua vez I^{3}FS[38]–*In-kernel Integrity Checker and Intrusion Detection File System*—implementa uma sobreposição das chamadas de kernel ao sistema de arquivos, permitindo que verificações de segurança sejam realizadas no ato da leitura—para execução. O funcionamento de I^{3}FS consiste em checar o *hash* do arquivo antes do kernel executá-lo, isso pode fazer com que alterações sejam descobertas antes do executável infectar o sistema operacional. Esta sobreposição é feita por um módulo que pode ser carregado pelo kernel, portanto nenhuma alteração dentro do kernel é necessária. I^{3}FS permite que sejam feitas verificações a cada N vezes que o arquivo é lido, para evitar grande sobrecarga no *host*.

Assinaturas de Arquivo com formato binário

Arbaugh *et al.*[57] propôs um método de assinatura e verificação de binários ELF[25]—*Executable and Linkable Format*—para o sistema operacional *Linux*. As assinaturas dos

binários são calculadas a partir de um função de hash MD5 e armazenadas sob um esquema RSA de assinaturas digitais. As assinaturas são adicionadas como um segmento ELF que cobre a parte executável dos arquivos. Quando um executável é carregado, o gerenciador ELF extrai a assinatura do segmento superior e calcula o *hash* do segmento em que está a parte executável do código. Se a verificação falhar a execução do arquivo não é permitida.

Hardware Criptográfico

TPM—Trusted Platform Module—é uma especificação de segurança definida pelo Grupo de Computação Confiável—Trusted Computing Group[17]. Sua implementação consiste em um chip fisicamente acoplado em placas-mãe, esse chip é controlado a partir comandos bem definidos dados por *softwares* instalados no sistema operacional, permitindo ao computador tirar proveito de recursos de segurança avançados. Um dos recursos presentes no TPM é armazenamento seguro de pequenas porções de dados. Sistemas operacionais utilizam esta porção de dados para armazenar valores relativos *hashes* de setores de disco, quando a máquina for iniciada verificações de integridade são feitas com base nesses valores. Por possuir uma interface simples e estar situado em hardware, este sistema é resistente a ataques de software[6].

4.2 *NIDS – network-based intrusion detection system*

Os NIDS podem monitorar dados coletados do segmento de rede em que ele está instalado ou ainda agrupar dados coletados em vários *hosts*—monitoramento *multi-host*—da rede. Não sendo limitados à escolha de uma das duas abordagens, NADIR[19] e DIDS[18] por exemplo, são ferramentas que utilizam as duas abordagens. Segundo Hofmeyr *et al*[20], apesar de algumas particularidades a arquitetura de um NIDS é geralmente separada em três partes: um coletor de dados; um gerenciador e um módulo responsável pela comunicação dos resultados da análise.

O coletor pode ser formado por um conjunto de sensores espalhados pela rede, são responsáveis pela captura, formatação dos dados e análise do trafego da rede. Este é o componente que tem responsabilidade de realizar o preprocessamento da análise. No coletor as informações de entrada serão transformadas e normalizadas, para que possam futuramente ser comparadas com as informações de perfil padrão de anomalia.

Após a padronização da entrada de dados, o mecanismo de classificação—presente no gerenciador—identifica as entradas como eventos de comportamento intrusivo ou eventos normais. A classificação dos eventos depende do modelo de detecção, sendo que os dois modelos principais são os baseados em mau uso e o de detecção de anomalias. O

gerenciador também é responsável pela administração integrada dos sensores e definir os tipos de resposta a serem utilizados para cada tipo de comportamento da rede. O módulo mais básico é o comunicador que envia os resultados de análise baseando-se em políticas definidas pelos administradores.

A detecção de anomalias é feita com a captura e análise dos cabeçalhos e conteúdos dos pacotes, normalmente esses pacotes são comparados com padrões ou assinaturas conhecidos. O NIDS é eficiente contra vários tipos de ataque e pode bloquear alguns deles em tempo real[34].

Na detecção por mau uso, os mecanismos de classificação comparam os dados de entrada com regras e outros tipos de descritores de comportamento. Por sua vez, em modelos de detecção baseados em anomalias, a comparação normalmente é feita com perfis estatísticos do comportamento histórico do usuário ou sistema. É traçado um comportamento padrão para cada usuário, após isso os dados capturados de cada usuário são comparados com o seu padrão[47].

Dentre os benefícios dos NIDS se destacam[43][48]:

- Não causa impacto no desempenho da rede;
- Os ataques pode ser identificados em tempo real, dando a chance do administrador responder rapidamente;
- Captura tentativas de ataques que ainda não obtiveram resultado;
- É independente do sistema operacional, enxerga todos eles como um só, desde que todos estejam ligados a uma rede com um mesmo padrão;
- Podem ser invisíveis ao atacante, é possível que um NIDS monitore a rede sem revelar sua presença, enquanto que um HIDS certamente deixa traços no sistema onde está instalado;
- Maior campo de atuação com menor infraestrutura.

Os pontos negativos mais importantes são[43][48]:

- Dificuldade de compreensão de protocolos de aplicações específicas;
- Incapacidade de monitorar tráfego cifrado;
- Dificuldade de utilização em redes fragmentadas, principalmente com *switches*;
- Quando o tráfego da rede ultrapassa a capacidade de coleta, muito tráfego deixará de ser analisado;

- Para a detecção de alguns tipos de ataque é preciso armazenar dados, por exemplo em conexões TCP são armazenar os estados das conexões, dependendo do fluxo da rede, o NIDS necessita de grande memória para armazenar esses estados.

4.3 *Hybrid IDS*

O tipo híbrido de IDS consiste em uma interessante mistura dos pontos fortes e fracos dos HIDS e NIDS. Na prática eles operam como um NIDS, coletando e processando tráfego da rede— pacotes—para detectar ataques. Porém como um HIDS eles fazem isso baseando-se em cada *host*, processando somente os pacotes que são endereçados ao próprio sistema. Resolvendo desta forma o problema de desempenho comum nos NIDS. Porém preservando o problema da escalabilidade, pois um IDS híbrido deve ser instalado em cada equipamento.

Outra corrente de pesquisadores[60] divide tanto HIDS quanto NIDS em duas categorias citadas anteriormente, detecção baseada em padrões e baseada em desvio de comportamento. E considera um IDS híbrido aqueles que combinam os dois métodos de detecção. A estratégia de padrões gera um pequeno número de falsos alertas, porém não identificam ataques inéditos. Já a abordagem baseada em comportamento gera um grande número de alertas que não consistem em ataques, porém podem detectar novos tipos de ataque.

Além disso um IDS pode ser considerado híbrido por permitir um gerenciamento centralizado, ou seja, localizar sensores em vários segmentos da rede e outros IDS baseados em host são usados em máquinas. A central de gerenciamento pode controlar as regras de ambos os tipos de IDS, formando um IDS híbrido[34].

4.4 *Storage-based Intrusion Detection System*

Essa classe de IDS consiste em monitorar características das modificações no sistema de arquivos, ou seja, monitorar os pedidos de escrita de dados enviados pelo sistema operacional ao sistema de armazenamento de dados. Isso permite que o IDS possa identificar várias ações comuns de um atacante como instalar *backdoors*, inserir cavalos-de-tróia no sistema, adulterar ou excluir arquivos de *log*. Todo sistema de armazenamento consegue enxergar as operações que serão realizadas em dados persistentes, permitindo transparente análise de mudanças indesejáveis e gerar alertas para cada cliente do sistema de armazenamento—alguns sistemas de armazenamento via rede possuem vários clientes[4].

Este tipo de IDS tem algo em comum com os NIDS, a possibilidade de ser independente do sistema operacional monitorado, dessa forma o atacante que obteve acesso privilegiado

na máquina monitorada não pode desativar o sistema de monitoramento, a não ser que consiga acesso privilegiado no servidor de armazenamento[52].

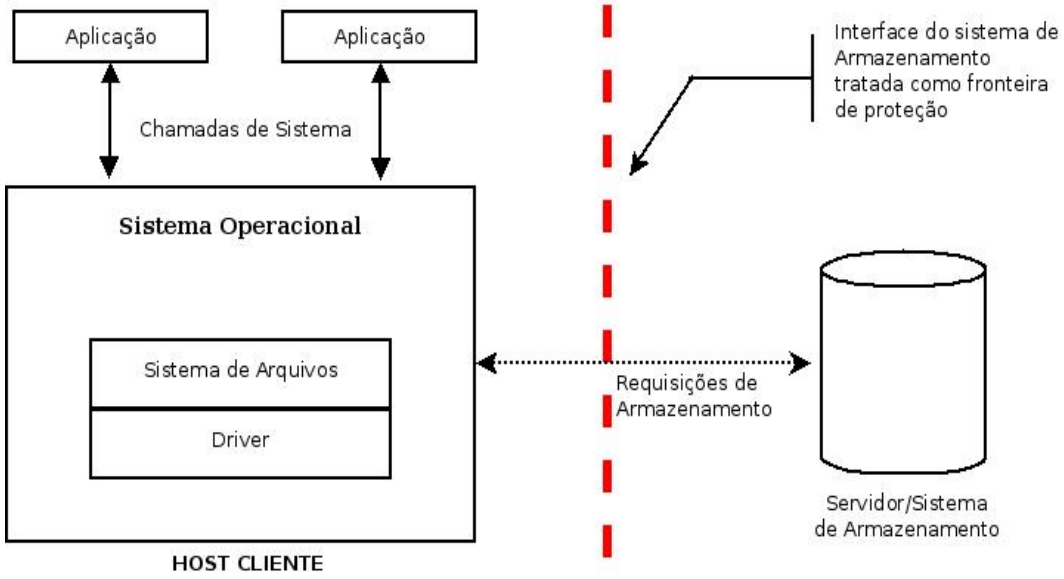


Figura 4.1: Esquema de storage-based IDS por Strunk *et al.*[52].

A arquitetura mostrada pela Figura 4.1 só pode ser implantada quando o sistema de armazenamento faz parte de um hardware separado, rodando código separado do sistema operacional do *host*. As duas entidades devem ser ligadas por uma simples interface de comunicação, com comandos bem definidos—SCSI, NFS, CIFS. Esta estrutura coloca o sistema de armazenamento em posição ideal para proteger os dados do sistema [52].

Apesar de necessitar de recursos de armazenamento para isto, os *storage-based* IDS podem guardar históricos das operações que ocorrem no servidor de arquivos. Isso garante futuramente a possibilidade de recuperação no caso de invasão efetiva, que pode causar danos aos dados armazenados.

Pennington *et al.*[40] implementou um protótipo deste tipo de IDS criando um módulo para o *NFS* server, determinando políticas de segurança para arquivos dos sistema, como: monitorar arquivos *append-only*—*logs*; desvio do padrão de escrita—por exemplo, um arquivo que contém senhas costuma armazenar 7 novas linhas por vez, que consistem em um novo registro; observar modificações nos *timestamps* dos arquivos—às vezes um atacante consegue modificar um *timestamp* para ocultar seu acesso. Experimentos realizados demonstraram que 15 de 18 ferramentas reais de intrusão puderam ser detectados apenas por suas alterações no sistema de arquivos[69].

Zhang *et al*[69] afirma que os storage-based IDS consistem em conjuntos de regras detecção e recuperação no nível de arquivo ou de blocos, não havendo integração entre

os dois níveis. O autor propôs um framework para detectar intrusões no protocolo iSCSI de armazenamento via rede, o sistema é capaz de detectar alterações no nível de arquivo e efetuar recuperações no nível de blocos. Como vantagem o modelo apresenta maior quantidade de informações ao usuário quando ocorrem incidentes—o que não ocorre na detecção por análise de blocos—com um sistema de recuperação mais rápido e eficiente que consiste em armazenar mudanças no nível dos blocos da partição.

4.5 Configuration Analysis Systems

Nesta categoria estão inseridas as ferramentas que realizam uma análise sistemática do *host* para buscar por problemas comuns de configuração incorreta. De certa forma, os controles de acesso que sistemas operacionais implementam são inúteis se arquivos e diretórios críticos possuem vulnerabilidades. Muitas ferramentas de análise de configuração foram desenvolvidas para realizar análises periódicas buscando por configurações e arquivos vulneráveis. Por exemplo, COPS[12]—*Computer Oracle Password and Security System*— que possui uma série de componentes que buscam por problemas comuns no sistema analisado como senhas fracas, arquivos e diretórios de usuário com permissão livre para escrita e execução, arquivos e diretórios do sistema com permissões de escrita e execução para qualquer usuário etc.

Um analisador de segurança baseado em regras denominado SU-KUANG [26] é o principal componente do COPS. Este analisador utiliza o raciocínio de atacante para procurar por brechas no sistema. Para isto o administrador dá como entrada um objetivo de ataque—por exemplo, obter acesso de root ao sistema—e um conjunto de privilégios iniciais que serão garantidos ao atacante. O analisador através de encadeamento recursivo quebra o objetivo principal em um ou mais sub-objetivos. Se o objetivo principal eventualmente for reduzido aos privilégios iniciais, então um ataque em potencial foi encontrado.

4.6 Estratégias de Recuperação de Sistemas

Ferramentas de detecção de intrusão e de diagnóstico pós-ataque com certeza fazem parte de uma estratégia inteligente de segurança. Um eficiente planejamento para a recuperação entretanto é uma necessidade. Mesmo em redes bem administradas, em que todas as máquinas sempre estão atualizadas, incidentes de segurança podem inevitavelmente acontecer, por isso é importante que a estratégia de recuperação seja eficiente e completa.

Nos sistemas convencionais, a recuperação de incidentes de segurança é uma tarefa complicada e que consome muito tempo, tanto em termos de recursos humanos quanto de indisponibilidade do serviço oferecido pela máquina prejudicada. Cada passo da recupe-

ração requer muita atenção, pois qualquer erro de configuração pode criar outra brecha de segurança no sistema.

Normalmente ferramentas de análise de intrusão fornecem mecanismos para que o sistema possa ser automaticamente ou manualmente recuperado, para isto podem definir um conjunto de arquivos, porção de memória ou mesmo processos que estão infectados. A seguir serão listados algumas técnicas de recuperação.

4.6.1 Redundância

A mais fundamental técnica para recuperação é a redundância. Isto significa que qualquer elemento de uma informação é armazenada de forma redundante em algum outro sistema de maneira que ela possa ser reconstruída a partir de alguns outros elementos armazenados no sistema. Esta redundância pode significar *backups* espalhados pela rede, por exemplo.

De certa forma, a estratégia de Santry *et al*[46] pode ser considerada uma técnica de redundância. Santry *et al*[46] projetou um sistema de arquivos que utiliza um armazenamento de dados inteligente, criando marcos de recuperação para cada arquivo quando ele é aberto em modo escrita. O principal problema desta abordagem é o ineficiente uso do disco, pois todo marco guarda o arquivo inteiro novamente, e não somente a diferença entre as versões.

4.6.2 Recuperação por Retorno

No caso de erros onde onde nenhuma ação corretiva pode ser determinada ou mesmo não se pode determinar a extensão do problema esta técnica pode ser empregada. Este tipo de recuperação usa mecanismos de bancos de dados como os *logs* de *undo/redo*, que é usado para apagar transações recentes e restaurar o banco de dados para um estado anterior.

Zhu & Chiueh[70] implementou um sistemas de arquivos com suporte a este tipo de recuperação, permitindo ao usuário recuperar o estado dos arquivos para uma data anterior. Para isto criaram um sistema de *log* detalhado de qualquer operação de escrita no sistema de arquivos, isto faz com que qualquer operação deste tipo seja reversível. Entretanto não foram consideradas ações conflitantes, quando por exemplo se apaga um arquivo e um novo arquivo sobrescreve os dados do anterior.

Por sua vez, Taser[15] implementa a recuperação seletiva, onde reverte-se o sistema de arquivos para um estado anterior e logo após pode-se refazer somente as operações que forem desejadas. Taser tem como fornecedor de informações o sistema de análise de intrusões Forensix[16]. Forensix marca um conjunto de processos que foram infectados por uma intrusão e a partir de *logs* dos processos consegue verificar quais arquivos foram possivelmente infectados, um analisador identifica quais são as operações do sistema de arquivos que não devem ser refeitas. O algoritmo que determina a abrangência da intrusão

contém duas fases de operação, a de rastreamento e a de propagação. Na fase de rastreamento é definido o conjunto de arquivos e processos que foram fonte do ataque, além do estimado período em que o ataque teve início, esta fase requer intervenção manual. A partir deste conjunto de processos e arquivos a etapa de propagação constrói um grafo das operações realizadas a partir da data estimada do ataque, os nós deste grafo são os arquivos e processos e suas arestas são as chamadas de sistema. Através de uma tabela o analisador consulta quais as operações—arestas—podem oferecer risco de contaminação e, considerando somente estas arestas encontra o fecho transitivo de todos os vértices inicialmente marcados como contaminados. Após esta análise o algoritmo devolve um conjunto de arquivos contaminados.

ReVirt[11] utiliza o conceito de máquina virtual para evitar que um ataque possa comprometer o sistema de *log* que permite a recuperação. Revirt encapsula o sistema monitorado em uma máquina virtual UMLinux e faz uma cópia inicial do seu disco virtual. A partir deste ponto uma versão modificada do UMLinux cria *logs* de todas as suas operações sobre o disco. A partir desta imagem inicial do disco e do conjunto de operações sobre ele o sistema pode ser restaurado até a data necessário, ou seja, quando um ataque ocorre o administrador determina uma data para o sistema ser reconstruído e após isso pode fazer atualizações para corrigir a vulnerabilidade explorada pelo atacante. O autor considera que novas imagens possam ser tiradas do disco a cada semana, para não deixar o processo de restauração lento.

4.6.3 Versionamento

Consiste em manter árvores de versões para cada arquivo, em para cada versão existem detalhes da transação feita para armazenar somente a diferença entre elas, o que permite uma recuperação eficiente e elegante. O sistema de Zhu & Chiueh[70] pode ser considerado um método de versionamento também, porém não guarda as operações baseando-se no conteúdo de cada arquivo, seu método analisa as operações no nível de bloco, desconsiderando qual arquivo está sendo manipulado na operação.

4.6.4 Particionamento Dinâmico de elementos de Informação

O objetivo principal é separar os arquivos suspeitos para serem verificados em um local seguro e independente do sistema que será recuperado, após análise serão eleitos os candidatos que podem ser reintegrados ao sistema principal. SEE[54] cria uma cópia temporária do sistema de arquivos principal e faz com que os programas suspeitos executem modificações sobre esta cópia. As modificações feitas na cópia não podem ser revertidas para o sistema de arquivos original.

Hsu *et al*[21] descreve um framework que isola processos não confiáveis, as modificações que este processos executam no sistema são visíveis somente para eles, mantendo o sistema íntegro para os processos confiáveis. Se for decidido que um processo se tornou confiável, suas alterações vão ser efetivadas e ficarão visíveis ao conjunto de processos confiáveis. Esse sistema é complementado por um detector de *malware* responsável por classificar os processos.

4.7 Conclusão

O objetivo deste capítulo era contextualizar as principais estratégias de detecção de intrusão bem como as técnicas de recuperação de sistemas comprometidos. Com a eminente evolução das técnicas de ataque pode-se concluir que sistemas de detecção são ferramentas necessárias para qualquer ambiente computadorizado. A implantação de métodos de segurança devem ser cuidadosamente planejada, levando em consideração as técnicas existentes e os efeitos colaterais que elas possam causar.

Vários tipos de ataques são possíveis através de modificação maliciosa nos sistemas de arquivos. O alvo principal dos atacantes é modificar binários executáveis com o objetivo de instalar *backdoors*, cavalos-de-tróia etc, para obter acesso permanente ao sistema atacado. Se modificações não autorizadas forem detectadas em um menor tempo, o dano causado pela intrusão pode ser reduzido ou até prevenido. Uma vasta gama de ferramentas utilizam *hashes* de arquivos para eficientemente detectar modificação ou substituição não autorizada. Além disso uma outra classe de ferramentas utiliza-se de técnicas de armazenamento seguro para não somente detectar mudanças mas também fornecer mecanismos para recuperar a máquina em caso de modificação dos arquivos

Capítulo 5

Projeto de um Sistema Resiliente

Neste capítulo, será apresentada uma arquitetura para viabilizar o desenvolvimento de um aplicativo de detecção de intrusão e recuperação de sistemas GNU/Linux. Para isto será utilizada a detecção de intrusão baseada em integridade de arquivos e ferramentas de Controle de Versão de arquivos e diretórios para armazenar os diferentes estados de configuração do sistema. Criando assim um ambiente resiliente.

Os maiores problemas enfrentados no desenvolvimento desta arquitetura consistiram em: como estabelecer pontos de recuperação localmente com esse tipo de ferramenta; como prover um módulo de restauração do sistema – para quando o erro ocasionar a queda do sistema; onde e quando criar pontos de restauração; como garantir a segurança dos dados de controle utilizados pela ferramenta e como prover facilidade de uso ao usuário.

Outra importante questão é a metodologia de escolha dos arquivo que devem salvos, ou seja, quais arquivos são de fundamental importância para o sistema e que são difíceis de obter novamente. Essa abordagem de limitar o ponto de restauração ajuda conter a possibilidade de os estados salvos comprometerem a capacidade de armazenamento do usuário. Apesar de a capacidade de armazenamento de discos rígidos atuais ser elevada.

Também serão mostrados detalhes da implementação de um protótipo que será usado no Capítulo 6 para validar o modelo proposto neste trabalho.

5.1 Formalização do Problema

A qualidade de uma ferramenta de detecção de intrusão depende de quão bem ela trata as realidades de grandes ambientes computacionais. Isso inclui portabilidade em máquinas com diferentes propósitos, facilidade de uso e flexibilidade de configuração de diferentes políticas de segurança. Outro quesito importante é que qualquer ferramenta deve possuir mecanismos internos de segurança, além de assegurar o sistema onde será usada. Nesta seção serão discutidos algumas características desejáveis para a ferramenta proposta neste

trabalho.

5.1.1 Questões administrativas

É comum um administrador ser responsável por um conjunto grande de computadores interligados via rede e ou internet. Essas máquinas podem ter diferentes itens de hardware, sistemas operacionais, versões de software e configurações. Algumas delas são críticas pois podem disponibilizar serviços especializados como servir e-*mails* e arquivos. Além disso, a administração destas máquinas deve respeitar políticas locais, que normalizam periodicidade de backups, controle de acesso e segurança. Mesmo pequenos ambientes de rede possue diferentes políticas para baseadas em regras próprias.

Geralmente as configurações são classificadas em grupos básicos com base no propósito de cada máquina—servidores, estações de trabalho etc. Em ambientes com uso de *Linux*, podem ser usadas diferentes distribuições dependendo do objetivo do sistema, algumas distribuições oferecem melhores condições de usabilidade—ideal para estações. Já outras podem oferecer melhores recursos e eficiência para máquinas servidoras.

É necessário que a ferramenta de segurança trabalhe com este ambiente complexo, ela precisa ser flexível para suportar configurações diferentes e até únicas, além disso deve possuir mecanismos que facilitem seu uso em qualquer sistema operacional onde vai ser instalada.

5.1.2 Relatórios

Para esta proposta em especial, a ferramenta deve informar o estados de arquivos que foram mudados, adicionados ou alterados. Essa questão pode ser crítica pois muitos arquivos podem estar em constante mudança: *logs* são atualizados, programas são atualizados, preferências de usuário são trocadas etc. Tipicamente essas informações não tem interesse para administradores de sistemas. Uma ferramenta que reportar qualquer mudança certamente força o usuário a analisar grande quantidade de informação, podendo dificultar o reconhecimento de ameaças genuínas.

De fato disponibilizar métodos de filtragem para a saída das análises é fundamental. Essa funcionalidade deve ser feita com extremo cuidado, de forma a reduzir a quantidade de informação inútil sem haver descarte de partes onde há grande interesse.

5.1.3 Questões de Segurança

A ferramenta proposta irá armazenar uma significativa quantidade de dados de controle e eles necessitam ser protegidos. Se um atacante conseguir alterar os dados de controle ele será capaz de comprometer todo o funcionamento da ferramenta. Armazenar os dados de

controle em uma mídia com acesso somente para leitura é a melhor abordagem, previne alterações e concede livre acesso ao sistema.

Após um arquivo ser adicionado ou excluído e essa ação ser considerada correta, os dados de controle precisam ser atualizados para refletir a mudança. Isso previne de essa alteração de aparecer futuramente em relatórios de verificação. Além disso, pela natureza da ferramenta, quando um arquivo é alterado corretamente ela deve atualizar o seu histórico e conteúdo no repositório. Atualizar os dados de controle que estão em uma partição montada com acesso somente para leitura trará procedimentos burocráticos inevitáveis, porém que tem um considerável nível de segurança.

Sistemas de arquivos são dinâmicos por natureza, isto significa que os dados de controle devem ser frequentemente atualizados. Regerar todos os dados de controle é sem dúvida uma tarefa tediosa, portanto a ferramenta deve suportar atualizações sem regerar toda a base de dados novamente.

5.1.4 Principais causas de Violação de Integridade

Violação de integridade pode ser causada por mal funcionamento de *hardware* ou de *software*, atividade maliciosas ou erros de usuário. Na maioria dos sistema que não há verificação de integridade, essas violações não serão detectadas, causando futuros problemas de software ou mais danos aos dados do sistema. A seguir serão descritas as principais causas de violação de integridade[49]:

Erros de Hardware e Software

Os dados de um dispositivo de armazenamento podem ser transmitidos através da rede em resposta a um pedido de armazenamento, podendo ser corrompido por mal funcionamento de *hardware* ou *software*. Um erro de *hardware* pode passar desapercebido pelo *software* e causar grandes danos aos dados armazenados. Se houver por exemplo, um erro de um *bit* de hardware no número do *inode* onde a informação será armazenada, importantes dados podem ser sobrescritos.

Erros de hardware não são incomuns, por exemplo, um disco defeituoso pode fazer que dados sejam escritos erroneamente em porções aleatórias do disco[5]. A maioria dos *softwares* de armazenamento abstraem esta possibilidade por acreditar que o hardware pare em caso de falha.

Defeitos no software também podem causar modificações inesperadas. *Drivers* defeituosos podem corromper dados do sistema de arquivos. Sistemas de arquivos podem apresentar problemas corrompendo partições e seus dados, a maioria dos sistemas de arquivos assumem a escrita assíncrona, quando o sistema é desligado abruptamente um processo de escrita pode parar antes de ele acabar, resultando na inconsistência dos dados.

Sistemas de armazenamento distribuído necessitam passar seus dados através de redes não confiáveis, podendo corromper os dados ao longo do caminho. A menos que use protocolos que possuam técnicas de verificação e correção esses erros de comunicação podem ser transformados em dados inconsistentes.

Modificação Maliciosa

Gerenciamento de dados confiável é um desafio para engenheiros de *software* e *hardware*. Enquanto que cada vez mais informações confidenciais são armazenadas eletronicamente, necessitando de acesso através de diferentes interfaces, novas vulnerabilidades surgem. Em um sistema de armazenamento distribuído, por exemplo, os dados podem ser acessados remotamente através de redes não confiáveis. Se os dados através da rede não forem cifrados, certamente alguém que tiver acesso a rede pode capturá-los. Eventualmente danos aos dados confidenciais podem causar mais problemas do que a divulgação dos dados. A modificação dos dados ou até sua perda, podem causar grandes paradas ao sistema, podendo parar em corporações inteiras. Além disso, dados modificados podem garantir acesso permanente atacante, com isso ele pode usar da infraestrutura atacada para realizar novos ataques.

Descuidos de Usuário

Erros de usuário podem comprometer a integridade dos dados na camada de aplicação. Por exemplo, um usuário pode excluir o arquivo de configuração de um sistema de gerenciamento de banco de dados. O mal funcionamento do SGBD pode causar corrompimento dos dados do banco. Em geral, se usuários invalidam a metodologia de funcionamento de algum aplicativo, danos podem ser causados aos dados que eles manipulam.

5.1.5 Detecção de Alteração

Um método simples para detectar alteração em um arquivo é compará-lo a uma cópia feita anteriormente. A vantagem dessa abordagem é extrair exatamente que mudança foi feita no arquivo. Entretanto este método é extremamente ineficiente em tempo e recursos de armazenamento, potencialmente dobra o espaço usado para o sistema de arquivos armazenar o arquivo em questão.

Um eficiente método para assegurar a integridade de dados é o uso de funções de *hash*, este método também é conhecido como *file hashing*[32]. Atualmente funções *hash* tornaram-se um padrão de assinaturas para aplicações e protocolos de comunicação por rede. Funções de *hash* criptográfico mapeiam strings de diferentes tamanhos para *strings* pequenas de tamanho pré-fixado. Essas funções são geralmente resistente a colisões, o

que significa que achar duas *strings* que tem o mesmo valor de *hash* é impraticável[38]. Além disso, funções como SHA1 e MD5 oferecem resultados imprevisíveis[59].

5.2 Objetivos da Arquitetura

Para suportar eficientemente as questões abordados acima, as seguintes metas devem ser observadas:

1. **Plenitude**: O sistema deve coletar informação necessária para identificar ataques sobre os arquivos que monitora e , além disso para estar apto a recuperar completamente o estado da aplicação a qual o arquivo infectado faz parte. É desejado que o sistema seja capaz de detectar:

   - Substituição ou modificação maliciosa de arquivos vitais como os do diretório `/bin`. Usualmente atacantes podem substituir comandos básicos como `ls` e `ps` com cavalos-de-tróia. Esse tipo de ataque deve ser descoberto pela ferramenta através de assinaturas previamente tiradas de arquivos executáveis.
   - Corrompimento de dados por erros de hardware. Os dados de discos rígidos podem ser comprometidos por alguma interferência magnética. Nem sempre o sistema de arquivos detecta este tipo de problema.

2. **Autenticidade**: Tanto os dados sobre o controle de integridade quanto o funcionamento da ferramenta devem ser seguros, além disso a comunicação entre esses dois dois componentes deve ser fortemente autenticada.

3. **Análise**: O sistema deve ser capaz de informar quando e quais arquivos foram comprometidos, para arquivos texto deve ser capaz de reproduzir a diferença entre o arquivo original e o infectado.

4. **Eficiência**: A quantidade de dados armazenados deve ser mínima, prevenir que a redundância seja apenas para arquivos essenciais e o comportamento do sistema não seja igual a métodos de *backup*.

5. **Solidez**: Reparar somente os arquivos que sofrerem alterações, não todo o conjunto de arquivos sob controle de integridade. Quando vários arquivos forem alterados a sua restauração deve ser consistente, ou seja, qualquer estado de operação entre eles deve ser mantido.

6. **Disponibilidade**: Evitar indisponibilidade quando um erro for encontrado, a menos que manter operante o serviço em questão possa propagar o erro.

5.3 Arquitetura Proposta

Esta seção descreve a estrutura da solução proposta. A Figura 5.1 exibe o diagrama de fluxo geral de operação. A figura mostra duas entradas, o sistema de arquivos e um repositório. Inicialmente o repositório é gerado com informações coletadas dos sistema de arquivos quando este é considerado íntegro, no repositório também serão guardados os arquivos de configuração encontrados—arquivos de texto.

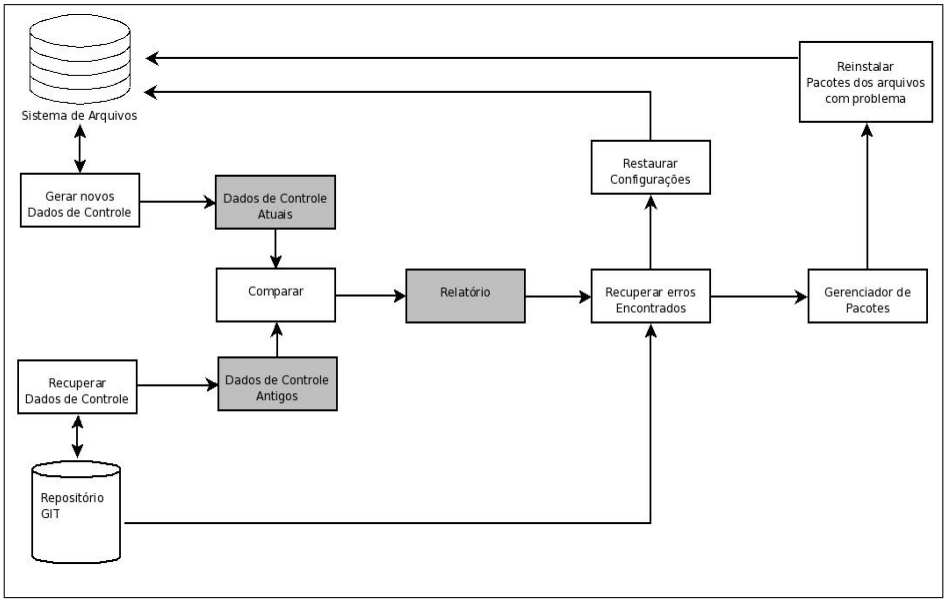


Figura 5.1: Diagrama de Fluxo Geral de Operação.

Após esta inicialização, a cada verificação serão calculados novos dados de controle do sistema de arquivos, esse dados serão comparados com os armazenados no repositório. Dessa forma, podem ser encontradas evidências de alteração. O relatório conterá todas essas informações de forma clara para o usuário, e servirá de base para o recuperador tomar decisões sobre a correção do sistema de arquivos.

O gerenciador de pacotes será responsável por reinstalar os arquivos binários possivelmente comprometidos, após essa reinstalação serão restauradas as configurações e o sistema poderá voltar a sua normalidade. Além disso, o gerenciador de pacotes também permite uma rápida maneira de criação de arquivos de controle para um determinado pacote. Desta maneira o usuário pode manter sob controle um aplicativo apenas sabendo qual pacote ele pertence, garantindo sua integridade completa.

A portabilidade referente ao uso de diferentes controladores de versão envolve apenas um módulo, que é o grande responsável por todo acesso a arquivo e pela comunicação externa. A manipulação dos descritores de integridade também é feita por este módulo,

pois os mesmos serão salvos no repositório. Este módulo é responsável pelo fornecimento dos descritores a todos os outros módulos.

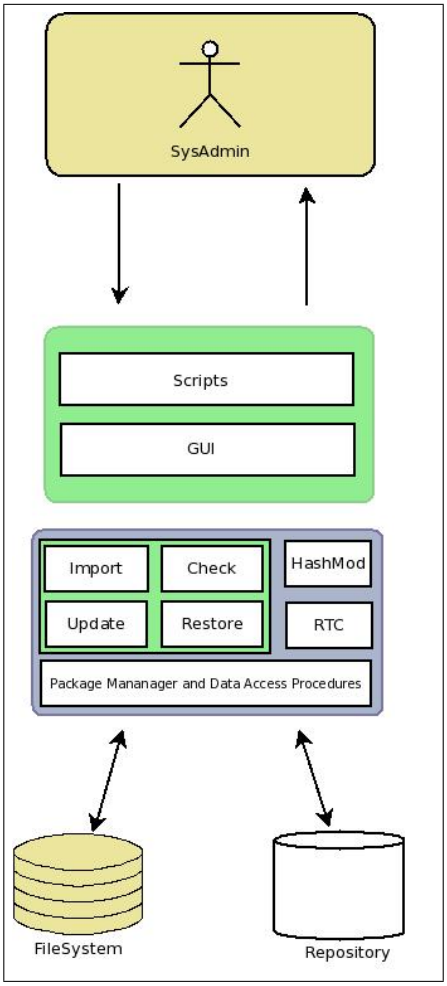


Figura 5.2: Arquitetura do sistema resiliente proposto.

A arquitetura apresentada na Figura 5.2 define um módulo responsável pela comunicação com o sistema de arquivos, com o repositório e com o gerenciador de pacotes. Logo, portar a ferramenta para várias distribuições envolve apenas codificação neste módulo.

Na segunda camada da arquitetura encontram-se duas possibilidades distintas de comunicação com os módulos implementados. Os *scripts* seriam ferramentas de linha de comando, úteis em ambientes de servidor onde não há recursividade gráfica. Outra implementação voltada ao usuário comum seria uma interface gráfica, almejando maior nível de usabilidade.

Na arquitetura encontramos:

- **Import**: Módulo responsável pela inicialização dos arquivos;

- **Check**: Responsável por verificar a integridade do sistema, realizará a leitura dos dados de controle no repositório para posteriormente compará-los com dados calculados a partir no sistema de arquivos;
- **Restore**: Módulo que interpreta a saída da verificação para calcular os passos de restauração automatizada;
- **Update**: Módulo responsável pela atualização dos dados de controle de acordo com alterações íntegras declaradas pelo Admnistrador do sistema;
- **RTC – Runtime Tree Calculator**: Responsável por criar a árvore de dependências de arquivos binários. Este módulo também é responsável por classificar os arquivos para o módulo *import* tomar decisões para cada arquivo.
- **HashMod**: Responsável pelos cáculos de Hash, também implementa mecanismos de cálculo de hash e cópia de blocos do arquivo para o repositório simultaneamente;
- **Package Manager and Data Access Procedures**: Módulo que contém todas as chamadas externas do programa, fornecendo aos outros módulos comunicação com o gerenciador de pacotes e, além disso, acesso os dados do repositório e do sistema de arquivos;
- **Repository**: O repositório corresponde ao local de armazenamento da cópias das configurações, os descritores de arquivo e os dados do Controle de Versão;
- **Descritores de arquivo**: São entidades que estarão armazenadas dentro do repositório, cada arquivo sob observação terá um descritor de integridade resposável pelo armazenamento de seus dados de controle, que serão usados em futuras verificações;

5.3.1 Descritor de Integridade

O descritor de integridade terá todas as informações possíveis a respeito de um arquivo ou diretório. Posteriormente informações retiradas de arquivos serão comparadas com as armazenadas nos descritores. As informações armazenadas serão:

- *timestamps* – modificação e criação;
- Tamanho em *bytes*;
- Número de blocos;
- *UID* e *GID*;

- Permissões;
- Inode;
- Número de Links;
- Duplo *hash* – composto por um hash MD5 e outro SHA256;

Os seis primeiros itens correspondem aos metadados presente em todos os arquivos e diretórios. Não é possível calcular *hashes* dos diretórios, porém pode-se armazenar outras informações que podem identificar evidências de alteração, como: o número de arquivos e o nome dos arquivos contidos no diretório. Armazenar todos os nomes dos arquivos para cada diretório é caro em termos de armazenamento, para contornar este problema será armazenado um *hash* da lista dos nomes dos arquivos contidos na pasta.

5.3.2 Módulo de Comunicação com o Gerenciador de Pacotes

Antigamente os usuários Linux necessitavam baixar e compilar um código fonte para instalar um aplicativo em suas máquinas. Atualmente ainda podem ser feitas instalações dessa forma. Outra alternativa é instalar um pacote. Um pacote contém o código fonte pré-compilado e empacotado como um arquivo binário de instalação—executável. Nele podem estar ícones, bibliotecas, arquivos de configuração, binários, manuais, atalhos, fontes etc. Além disso, um pacote pode conter metadados, como informações sobre versão, mantenedor do pacote, autor do software, informações de contato, licenciamento, alterações e o site do projeto e do código fonte.

Cada formato de pacote tem sua estrutura de arquivos, mas geralemente todos contém dos dados de instalação compactados. Quando o pacote é executado, seus dados são descompactados e copiados para o sistema de arquivos do sistema operacional, criando links simbólicos onde for necessário, atalhos de menu e, às vezes, oferecendo opções de configuração ao usuário. Os pacotes são criados para uma versão específica de uma determinada distribuição, pois as dependências podem variar entre distribuições e entre versões de uma distribuição.

Um gerenciador de pacotes instala, remove e atualiza pacotes. Essa é uma definição simples, um gerenciador de pacotes moderno pode fazer bem mais do que isso. Ele pode se conectar automaticamente a um repositório, baixar um programa, verificar e resolver suas dependências, listar pacotes, listar dependências, fazer buscas na lista de pacotes, ordenar a lista e adicionar e remover repositórios de pacotes. Pode ainda especificar um repositório para um pacote específico e bloquear atualizações de outros pacotes, verificar os hashes e as assinaturas digitais para garantir a integridade dos pacotes, fazer atualizações automáticas e remover dependências ao desinstalar programas.

Existem várias alternativas de gerenciamento de pacotes, geralmente cada distribuição adota uma para manter repositórios oficiais. Todas as ferramentas possuem particularidades, porém em um nível superficial, todas executam as mesmas operações. Para esta proposta em especial, o gerenciador deve ser capaz de: listar os arquivos de um pacote; descobrir de qual pacote é um arquivo e instalar e desinstalar pacotes.

Visando portabilidade e facilidade de uso, um módulo de comunicação com os gerenciadores de pacotes foi previsto no projeto. Desta forma, o sistema pode funcionar com em diferentes distribuições apenas com modificações no módulo de comunicação.

5.3.3 Runtime Tree Calculator

A integridade das ações de um aplicativo só é garantida com uso bibliotecas e arquivos de configuração íntegros. Um executável pode possuir diferentes tipos de dependência: arquivos de configuração, arquivos temporários, arquivos de *log*, executáveis auxiliares, bibliotecas e módulos carregáveis do kernel.

Além disso, executáveis também podem depender de recursos presentes em módulos de *kernel* específicos. Os módulos carregáveis do Linux – LKM - *Loadable Kernel Modules* – permitem que o *kernel* seja escalável e dinâmico. Estes módulos podem ser dinamicamente carregados e integrados a imagem do kernel que foi carregada no início do sistema. Estes módulos são armazenados no sistema de arquivos, podendo ser lidos, alterados e substituídos da mesma maneira que arquivos comuns. Desta forma, tornam-se arquivos normais que também irão passar pela verificação de integridade.

Este módulo é responsável por construir uma árvore de dependências de um arquivo executável, essa árvore servirá de base para o cálculo de dados de controle. Dessa forma todos os arquivos que forem dependência dos executáveis de um certo pacote também serão considerados parte do pacote. Sempre que uma verificação sobre um determinado arquivo for realizada, toda a sua árvore de dependência será analisada. A Figura 5.3 representa parte da árvore de dependências do gerenciador de máquinas virtuais VirtualBox.

Na maioria das vezes um gerenciador de pacotes não instala módulos do *kernel*, que por sua vez precisam ser compilados com intervenção manual. Assim uma alteração de módulo pode ser uma evidência crítica, ainda mais se este módulo estiver carregado na memória como parte do *kernel*.

Outra tarefa deste módulo é a classificação dos tipos de arquivos, que podem ser quatro: binário, texto, módulo e *log*. Esta classificação especifica aos módulos de *import*, *check* e quais são as ações tomadas para cada arquivo. Ao acessar esta classificação, o módulo *import* sabe quais arquivos serão copiados na íntegra para o repositório e quais metadados serão armazenados. Por sua vez, o módulo de *check* toma por base esta classificação para saber quais testes de integridade serão feitos para cada arquivo, em um

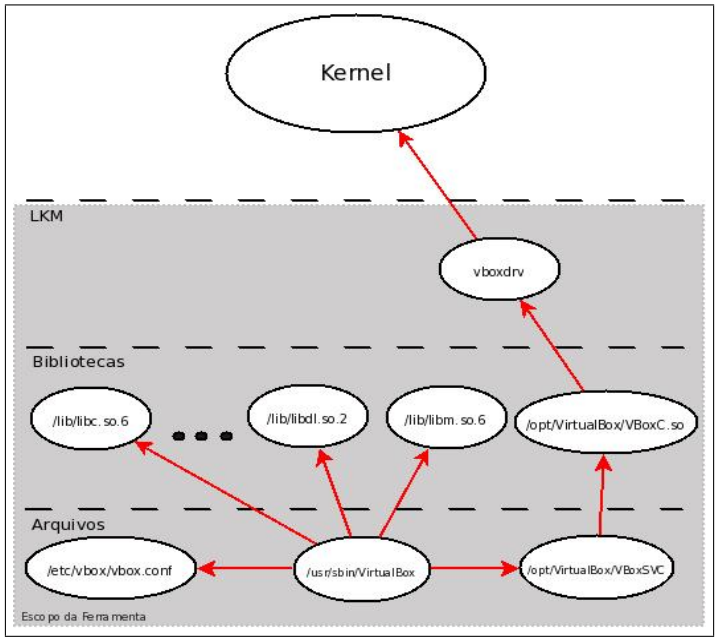


Figura 5.3: Exemplo de Árvore de execução do aplicativo VirtualBox.

arquivo de *log* por exemplo, não são comparados os *hashes* e é verificado se seu tamanho aumentou.

Técnicas de *boot* seguro

A árvore de dependências possui um limite máximo de ação, o *kernel*. O *kernel* é carregado no processo de *boot* do sistema, que por sua vez é carregado através de pequenos códigos localizados em uma parte específica do disco, a MBR—*master boot record*—também conhecida como setor de *boot*. A MBR contém informações sobre a tabela de partições do disco e o código de arranque do sistema operacional.

Alguns tipos de vírus podem contaminar a MBR e serem carregados em memória juntamente com o *kernel* [1][24], estes são chamados vírus de *boot*. Os vírus de *boot* mais recentes contém sofisticados mecanismos que conseguem driblar as verificações de integridade de código contidas no processo de carregamento do *kernel*, mantendo-se em memória após a subida do sistema operacional.

Este tipo de ataque não pode ser detectado pela solução proposta, porém eficientes técnicas de *boot* seguro podem garantir a integridade do sistema no processo de carga inicial[6] [17]

LKR - Linux Kernel Rootkits

Qualquer aplicação é controlada pelo *kernel*, e qualquer acesso de sistema é realizado através do *kernel*. A aplicação irá realizar uma chamada de sistema—*kernel syscall*—e o *kernel* fará o trabalho e retornará o resultado para a aplicação. Sob o ponto de vista do usuário, estas chamadas de sistema representam o mais baixo nível de acesso ao sistema de arquivos, conexões de rede e outros benefícios.

Algums *rootkits* de kernel conseguem modificar as chamadas de sistema, desse modo conseguem esconder arquivos e diretórios, processos e conexões de rede. Dessa forma, qualquer aplicação em espaço de usuário pode ser "enganada", pois utiliza uma chamada de sistema forjada pelo *rootkit*. A ferramenta proposta pode não ser capaz de detectar alguns *rootkits* de *kernel*, pois é suscetível ao uso de chamadas de sistema forjadas. Técnicas de detecção de *rootkits* de *kernel* encontradas em [29] devem ser usadas em conjunto com a ferramenta proposta para determinar políticas de segurança mais rígidas, ou seja, que cubram os dois casos.

5.3.4 Módulo Import

Este módulo é responsável por copiar os arquivos texto para um repositório, deverá também salvar os descritores de integridade para cada arquivo no repositório. Os descritores de integridade ficarão no repositório assim como os arquivos de texto, pois este apresenta um rígido controle de integridade sobre os dados versionados.

O módulo deve aceitar três argumentos, a localização do repositório, um conjunto de arquivos e um nome para este conjunto. Eventualmente o nome do conjunto de arquivos será o nome de um pacote. A localização do repositório será verificada por este módulo, em caso de inexistência ele o criará antes de importar os dados.

5.3.5 Módulo Update

O módulo de atualização será acionado toda vez que o administrador decidir guardar um novo estado dos pacotes instalados no sistema. Esta atualização pode ser realizada por vários motivos: atualização de aplicativos; instalação de novos aplicativos; mudança nas configurações; inclusão de novos usuários ou grupos de acesso etc.

Este módulo irá verificar primeiramente os *timestamps* dos arquivos para criar uma lista dos arquivos que necessitam ser atualizados. Após isso são adicionadas as novas versões dos arquivos texto e dos descritores de integridade no repositório.

5.3.6 Módulo Check

A verificação será feita a partir de comparações com base nas informações armazenadas no repositório:

1. Verificar a existência dos arquivos;
2. Comparar metadados;
3. Verificar o número e o nome dos arquivos de um diretório;
4. Comparar o *hash* dos arquivos.

A verificação terá três níveis de profundidade, criados a partir da combinação das operações enumeradas acima. O primeiro checa apenas a existência do arquivo ou diretório, o segundo compara os metadados e finalmente a terceira compara o *hashes* dos arquivos as informações específicas de diretórios.

Para efeitos de análise forense, um quarto nível de verificação pode ser considerado, a auditoria de arquivos de configuração. Com base nos arquivos texto armazenados no repositório podem ser identificadas as modificações que levaram o sistema a ter sua integridade afetada. Além disso, ordenando os *timestamps*—criação, modificação e acesso—dos arquivos onde foram reportados problemas pode-se criar a sequência de ações do atacante.

5.3.7 Módulo Restore

A detecção de intrusão e seu diagnóstico fazem parte de uma boa estratégia de segurança, a recuperação, entretanto é uma necessidade. Manter o sistema atualizado diminui a ocorrência de intrusões, mas elas inevitavelmente podem acorrer, portanto é crítico que a recuperação seja eficiente e completa.

A restauração conterá dois passos principais, a reinstalação de pacotes e a restauração das configurações, que serão executados dependendo do problema que for encontrado no sistema de arquivos. Se um arquivo binário for comprometido, é necessária a reinstalação do pacote do qual ele faz parte. Após isso devem ser restauradas todas as configurações previamente armazenadas no repositório.

Quando o problema é encontrado em um arquivo texto a recuperação torna-se trivial, apenas restaurar o arquivo que foi alterado. Salvo se esta configuração fizer parte de um serviço esteja rodando, neste caso cabe ao administrador parar o serviço para descarregar da memória possíveis configurações comprometidas que foram carregadas no arranque do serviço.

5.4 Protótipo

Um protótipo foi construído para validar a arquitetura proposta neste trabalho, visando amadurecer o projeto para uma solução completa de resiliência para sistemas *Linux*. Em busca de portabilidade, foram criados módulos independentes com mecanismos de comunicação bem definidos, facilitando a implementação futura de uma interface gráfica destinada a usuários comuns ou de um aplicativo de linha de comando para administradores de grandes servidores.

5.4.1 Detalhes da Implementação

O protótipo criado consiste em um utilitário de linha de comando *Linux*—como os tradicionais `ls`, `cd`, `grep` etc. Em geral, aplicações do modo texto são mais rápidas do que as que utilizam ambientes gráficos, pois estes por sua vez requerem maiores recursos de processamento . Por esta mesma razão, as aplicações de modo texto usam mais eficientemente os recursos de memória. Em contrapartida, ferramentas de modo texto apresentam interfaces menos amigáveis, principalmente para usuários iniciantes. Desta forma, a metodologia adotada é ideal para o uso em servidores, onde exige-se ótimo aproveitamento de recursos e administradores experientes gerenciam o ambiente.

Todos os módulos descritos na arquitetura, foram implementados contendo uma interface de comunicação bem definida, podendo ser facilmente integrado com qualquer *framework* de programação gráfica. O programa principal assim como todos os módulos foram implementados na linguagem *Python*, respeitando as normas de boas práticas de ferramentas de linha de comando para *Linux*. A Figura 5.4 mostra uma exemplificação de uso da ferramenta e seu batismo oficial, RE*squared*—*REsilient Recovery*, será chamada assim a partir deste ponto.

```
kid / # ./resquared --help
Usage: RepPath -[i|c|u|r] options FILES

Options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -i, --import          Import action
  -c CHECK, --check=CHECK
                        Check action
  -r RESTORE, --restore=RESTORE
                        Restore action
  -u UPDATE, --update=UPDATE
                        Update action
  -p PACKAGE, --package=PACKAGE
                        Update action
  -f FILELIST, --filelist=FILELIST
                        Update action
  -b BIN, --binary=BIN  Update action
kid / #
```

Figura 5.4: Exemplo de uso da ferramenta proposta.

A manipulação dos dados do repositório é feita internamente pela ferramenta através de um módulo Python específico, porém o repositório inicial deve ser criado pela ferramenta de Controle de Versão devidamente instalada na máquina, neste caso o GIT.

Módulo *Import*

A ferramenta possui três formas de import, por pacote—a partir do gerenciador de pacotes, por uma lista de arquivos ou a partir de um arquivo somente. Em qualquer um dos três meios de entrada, a ferramenta cria um registro que conterá o conjunto de arquivos e todas as operações futuras serão realizadas com base no nome que o usuário der para este conjunto.

Quando o usuário utilizar a entrada por nome de pacote, a ferramente buscará automaticamenta no gerenciador de pacotes quais são seus arquivos. Após obter a lista de arquivos, o módulo de cálculo de dependências classificará os arquivos—texto, executável, *log* ou diretório—e mapeará o conjunto de dependências deste pacote para posteriormente mandar a lista de arquivos para o módulo *import*. Ao utilizar um arquivo como entrada a ferramenta descobrirá automaticamente o pacote ao qual o arquivo pertence e repetirá o processo da entrada por pacote.

Para realizar o *hash* é necessário ler todo o conteúdo do arquivo, para evitar desperdício de recursos, o módulo de *import* realiza as cópias dos arquivos texto para o repositório paralelamente ao cálculo do *o hash*, dessa forma o arquivo deve ser acessado somente uma vez.

Módulo de comunicação com o Gerenciador de Pacotes

A implementação considerou o gerenciador de pacotes de distribuição Gentoo, o portage. E somente utilizou as operações básicas de qualquer gerenciador: listar pacotes, listar arquivos de um determinado pacote, dizer de que pacote é um arquivo e reinstalação de pacotes.

Segurança da Ferramenta

A segurança proposta pela ferramenta depende diretamente da integridade de seus componentes e dados de controle. Além disso, deve-se garantir que apenas operações genuínas alterem os dados de controle do repositório.

Para o uso da ferramenta é necessário a configuração de uma estrutura de segurança básica. Todo o repositório e o código dos módulos e da interface principal estarão em uma partição separada, cifrada e montada em modo leitura. Para realizar as operações de *import* e *update* a partição precisa ser montada em modo escrita, o que requer senha e intervenção do administrador.

Logo após a operação a partição deve ser novamente montada em modo leitura, para evitar que alterações não permitidas sejam feitas na ferramenta e em seus dados de controle. As operações de *check* e *restore* podem ser executadas normalmente sem intervenção de montagem, pois não utilizam recursos de escrita nos dados da ferramenta.chroot

Módulo de Queda

O sistema pode ser acessado por qualquer *LiveCD* que contenha o gerenciador de pacotes usado pela ferramenta e Python instalado. A estratégia de recuperação neste caso consiste em:

1. Rodar o sistema *Live* na máquina desejada;
2. Montar no sistema *Live* o sistema de arquivos que contém a ferramenta;
3. Fazer um "chroot" para o sistema de arquivos montado;
4. Recuperar os arquivos;
5. Reiniciar a máquina no sistema antigo.

Chroot é um comando Unix que é usado para mudar o diretório raiz do sistema. Quando o sistema *Live* tem esta nova raiz definida ele não pode acessar os arquivos que estão fora do ditetório passado como parâmetro, este processo é conhecido como "prisão *chroot*"— *chroot jail*. É extremamente comum admnistradores de sistema utilizarem este método para acessarem sistemas comprometidos.

5.5 Conclusão

Administradores atuais confrontam-se com ambientes que possuem centenas de máquinas. Além de cuidar de todos os serviços oferecidos por este ambiente ele precisa se preocupar com ameaças de segurança que podem ocorrer. Violações nos arquivos do ambiente podem ocorrer de várias maneiras, e, de fato o administrador necessita de auxílio computacional para identificá-las. Diversos mecanismos de verificação de integridade realizam esta tarefa, porém não fornecem mecanismos especializados de recuperação apenas sob arquivos violados.

Este capítulo apresentou uma arquitetura modular que permite a implementação de aplicativos de verificação de integridade e recuperação tanto para usuários leigos quanto para administradores de grandes servidores. O modelo possui ligação com o gerenciador de pacotes, comum em todas as distribuições atuais, para que possam ser feitas reinstalações sem a necessidade de *backup* de grandes arquivos. Apenas são guardadas as configurações dos aplicativos para que seja possível a restauração do comportamento de cada um deles.

Capítulo 6

Validação do Estudo e Resultados

A validação do estudo é composta por duas partes. Primeiro foi realizada uma análise da sobrecarga imposta pelo método de Controle de Versão em conjunto com a verificação de integridade. A sobrecarga foi calculada pela diferença do tempo médio de operação de alguns verificadores de integridade pelo tempo médio de operação da ferramenta proposta com todos os módulos ativados.

Para segunda parte de validação foram criados cenários de intrusão e erros de administração que tipicamente ocorrem em ambientes de produção. A ferramenta proposta foi colocada em prática em cada um deles para ser testado seu poder de detecção e recuperação.

6.1 Sobrecarga do Modelo Proposto

Esta seção contém uma avaliação de algumas ferramentas de verificação de integridade e da ferramenta proposta. Para isto uma outra versão da ferramenta foi criada, com os módulos de mapeamento da árvore de dependência e de Controle de Versão desativados, ou seja, a ferramenta teve duas funcionalidades desativadas para ser limitada a operar como um verificador de integridade.

As ferramentas foram avaliadas levando em consideração dois conjuntos de arquivos diferentes, a primeira base de dados possuia em sua maioria arquivos de texto, a segunda, por sua vez possuia mais da metade composta de arquivos binários. Cada teste foi executado 30 vezes, foram descartados os limites inferior e superior. Além disso, antes de cada teste os *caches* de disco foram excluídos.

Por convêniencia, nesta seção será referenciada por *Resquared* a ferramenta com capacidades limitadas e por *Resquared+GIT* a ferramenta com todas as funcionalidades ativadas.

6.1.1 Base de dados composta de arquivos texto

Para esta avaliação foram considerados os arquivos do diretório `/etc`, contendo 3428 arquivos, sendo 623 binários, 904 diretórios e 1901 arquivos texto, com um tamanho total de 104 Mb. Foram realizadas operações de inicialização e verificação em todas as ferramentas, a cada segundo foram capturadas informações dos processos, que são representadas nos gráficos abaixo.

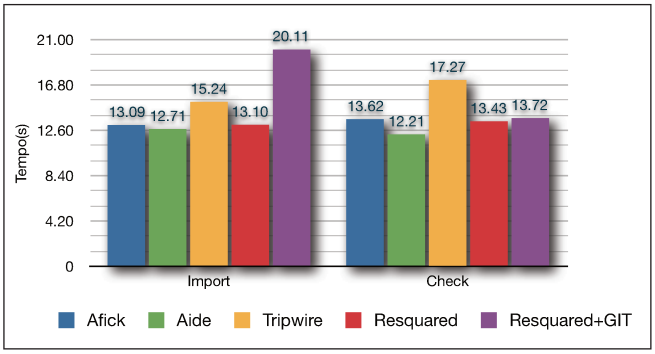


Figura 6.1: Tempo de execução sobre o diretório `/etc`

Na Figura 6.1 podemos verificar que o tempo de execução de *Resquared* é bem similar ao dos outros verificadores de integridade, tanto para operação de inicialização—*import*—quanto para a operação de verificação—*check*. A sobrecarga imposta pelas funcionalidades de mapeamento de dependências e recuperação impactaram somente na execução de *import*, o que era esperado, pois junto a essa operação é criado o repositório e são copiados os arquivos texto. Na verificação apenas são considerados os dados de controle de *Resquared+GIT*, portanto a operação é bem semelhante a de *Resquared*.

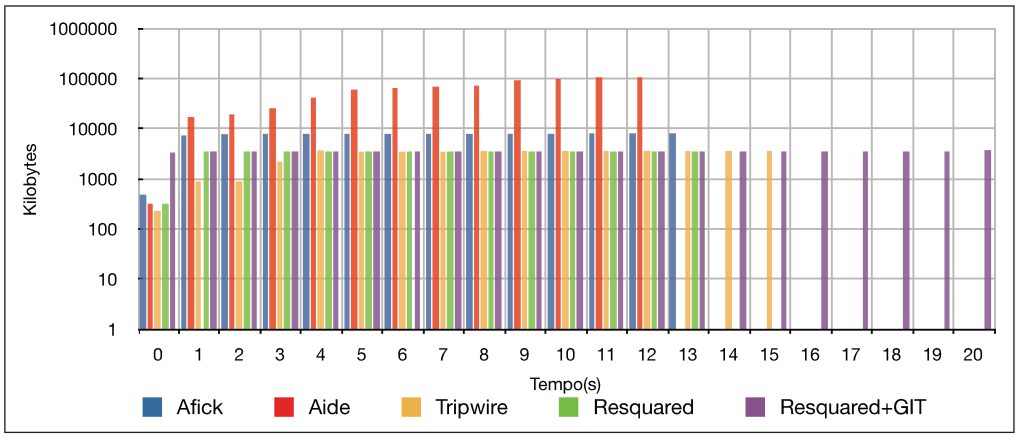


Figura 6.2: Consumo de memória da operação de inicialização com diretório `/etc`

A Figura 6.2 representa o gráfico de alocação de memória em escala logarítmica para

a operanção de *import* e, demonstra que o consumo de memória apresentado por *Aide* foi cerca de 10 vezes maior do que as outras ferramentas. *Resquared* , *Resquared+GIT* e *Tripwire* gastaram recursos com moderação semelhante. Já *Afick* gastou 2 vezes mais que as três. A análise das chamadas de sistema de *Aide* revelam que esta ferramenta utiliza mapeamento de arquivos em memória e faz uma operação de escrita em disco somente no final da execução, o que sugere um possível armazenamento de dados de controle em memória, sem realizar sequenciais transferências dos dados da memória para o disco.

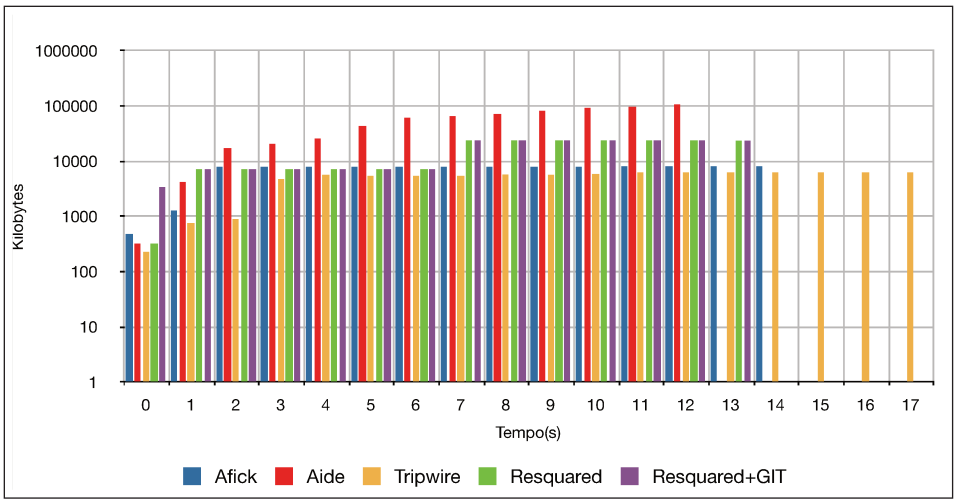


Figura 6.3: Consumo de memória da operação de verificação com diretório `/etc`.

O gráfico de consumo de memória da operação de *check*—Figura 6.3—mostra que *Resquared* e *Resquared+GIT* tiveram aumento considerável no consumo de memória, porém ainda distante do consumo apresentado por *Aide*, que foi 5 vezes maior.

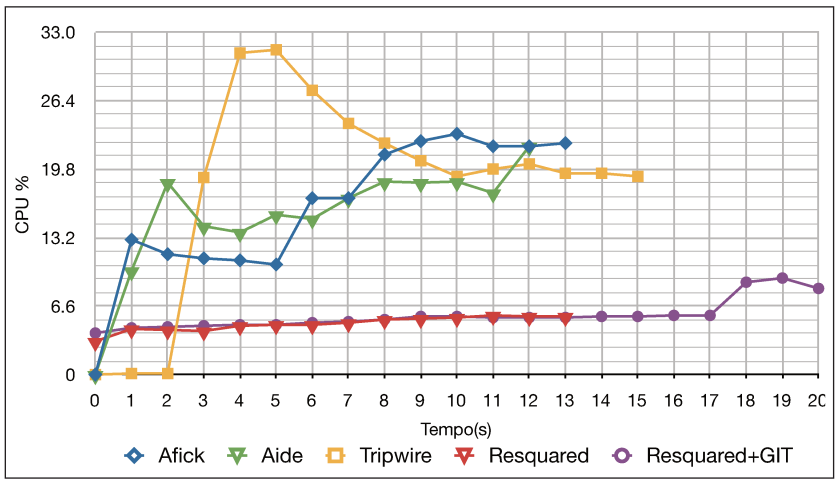


Figura 6.4: Porcentagem de CPU operação de inicialização com diretório `/etc`.

A análise do consumo de CPU—Figura 6.4—revelou uma grande vantagem tanto de *Resquared* quanto de *Resquared+GIT* sobre as outras ferramentas, girando em torno de 6%, foi cerca de 3 vezes menor do que a média das outras ferramentas. Pode ser notado um pequeno pico no final da execução de *Resquared+GIT* que representa o momento de adição de versionamento nos arquivos texto importados.

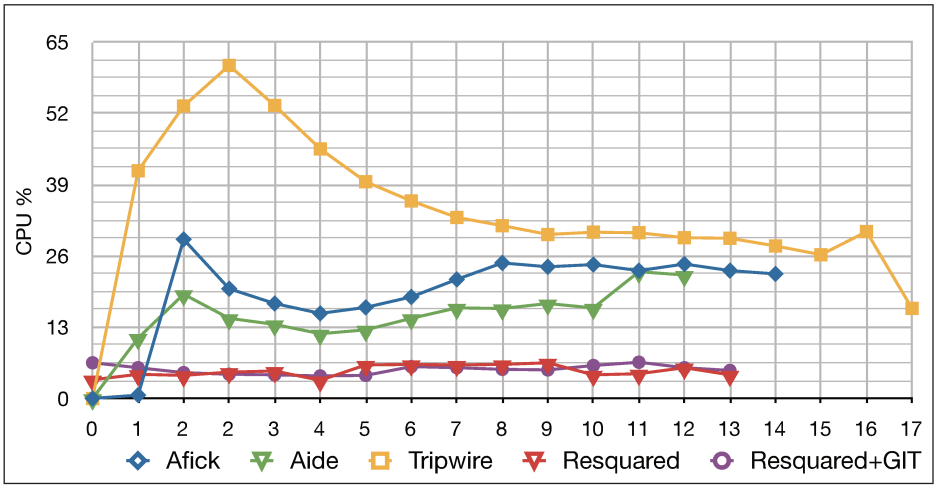


Figura 6.5: Porcentagem de CPU operação de verificação com diretório `/etc`.

Na operação de verificação o consumo de CPU de *Resquared* e de *Resquared+GIT* foi bem semelhante ao da operação de inicialização girando em torno de 6%—Figura 6.5. Todas as outras apresentaram aumento no consumo, liderado por *Tripwire*. Dessa vez o consumo de *Resquared* e *Resquared+GIT* foi em média 4 vezes menor do que a média das outras ferramentas.

6.1.2 Base de dados composta de arquivos binários

A base de dados foi montada com os arquivos do diretório `/usr/bin`, contendo um total de 2890 arquivos, sendo 2203 binários, 687 arquivos texto, com um tamanho total de 214 Mb. Foram realizadas operações de inicialização e verificação em todas as ferramentas. Conforme foi feito na análise anterior a cada segundo foram coletadas informações do processo.

O tempo de inicialização de *Resquared+GIT* foi aproximadamente 62% maior do que a média das outras ferramentas, além da sobrecarga atribuída à cópia dos arquivos texto, *Resquared+GIT* também sofre este aumento devido ao cálculo da compressão delta e dos arquivos de controle do GIT—ver Seção 3.1. O tempo de verificação de *Resquared+GIT* ficou próximo da média, apenas o carregamento dos descritores de arquivos é feita durante a verificação, portanto nenhuma operação específica do repositório é necessária.

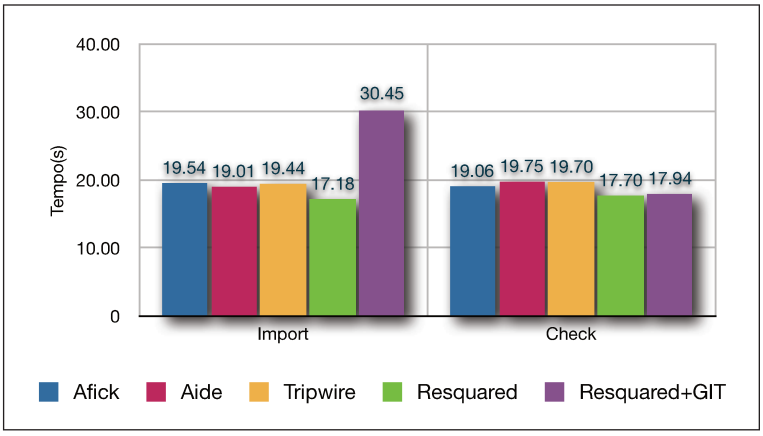


Figura 6.6: Tempo de execução sobre o diretório `/usr/bin`

A Figura 6.7 mostra que o consumo de memória quando trabalha-se com binários foi semelhante ao de *Tripwire*, mesmo trabalhando com arquivos grandes—binários—*Resquared+GIT* realiza leituras sequenciais de blocos com um tamanho fixo pré-estabelecido, após o cálculo do *hash* para cada bloco a memória alocada durante cada leitura é descarregada. Durante a inicialização dos dados são feitas sucessivas descargas dos dados calulados que estão na memória para o disco, reduzindo o consumo de memória. Um pico de consumo no final da inicialização pode ser observado, exatamente no momento em que os dados de controle do repositório são calculados.

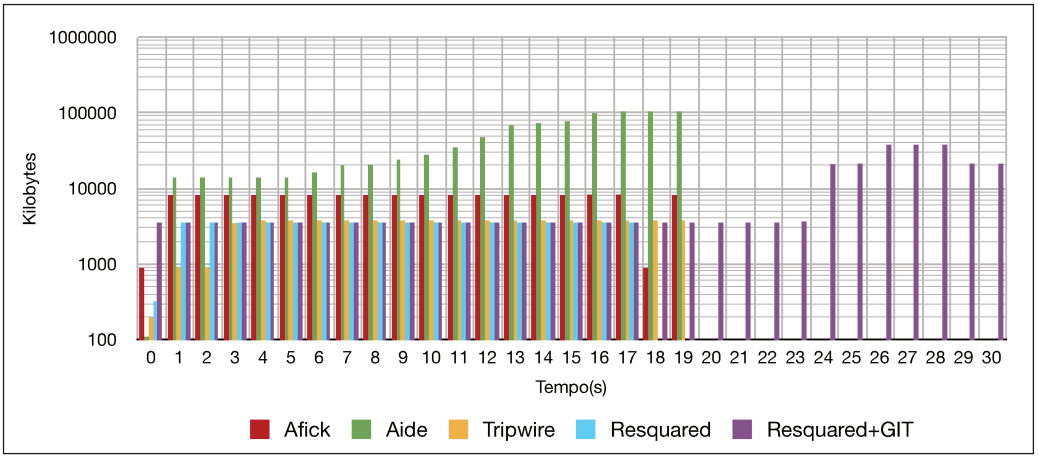


Figura 6.7: Consumo de memória da inicialização com o diretório `/usr/bin`.

Como pode ser observado na Figura 6.8 houve um pequeno crescimento do consumo de memória por parte de *Resquared* e *Resquared+GIT*, porém o consumo foi duas vezes maior do que a melhor média—*Tripwire*—e cerca de três vezes menor do que a pior média—*Aide*.

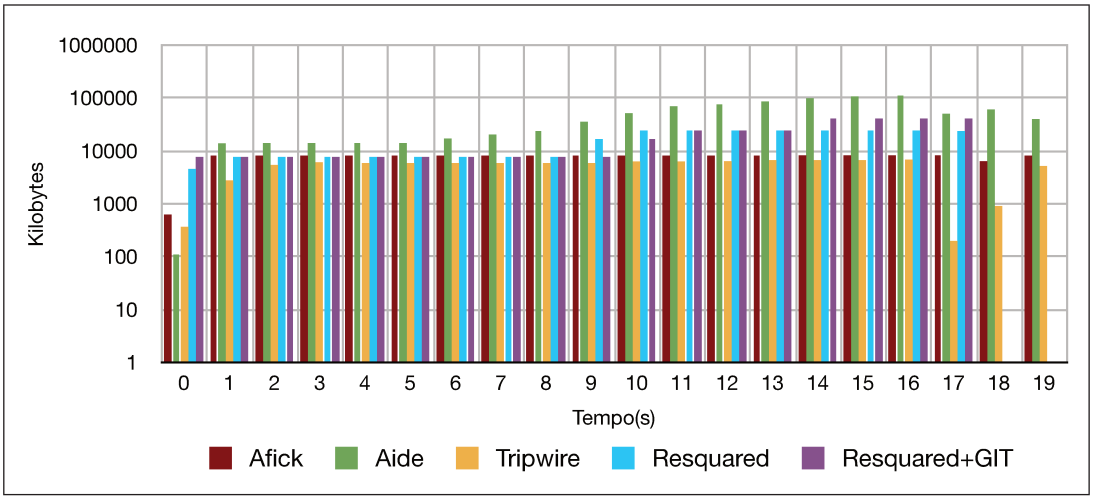


Figura 6.8: Consumo de memória da operação de verificação com o diretório `/usr/bin`.

Tanto *Resquared* quanto *Resquared+GIT* mostraram-se novamente mais eficientes no uso de CPU do que todas as outras ferramentas, tendo consumo médio em torno de 5,2%. A Figura 6.9 revela consumo médio em volta de 6 vezes maior de *Afick* e *Tripwire*. *Aide* também teve consumo elevado, em média um consumo 4 vezes maior.

A *performance* de *Resquared* e *Resquared+GIT* na verificação seguiu o padrão da inicialização, girando em torno de 5%—Figura 6.10 . Porém duas ferramentas tiveram um salto no consumo médio, *Afick* teve média 38,51%, enquanto que *Tripwire* obteve média de 52,92%. *Aide* teve comportamento semelhante ao da inicialização tendo média de consumo de 22,35%.

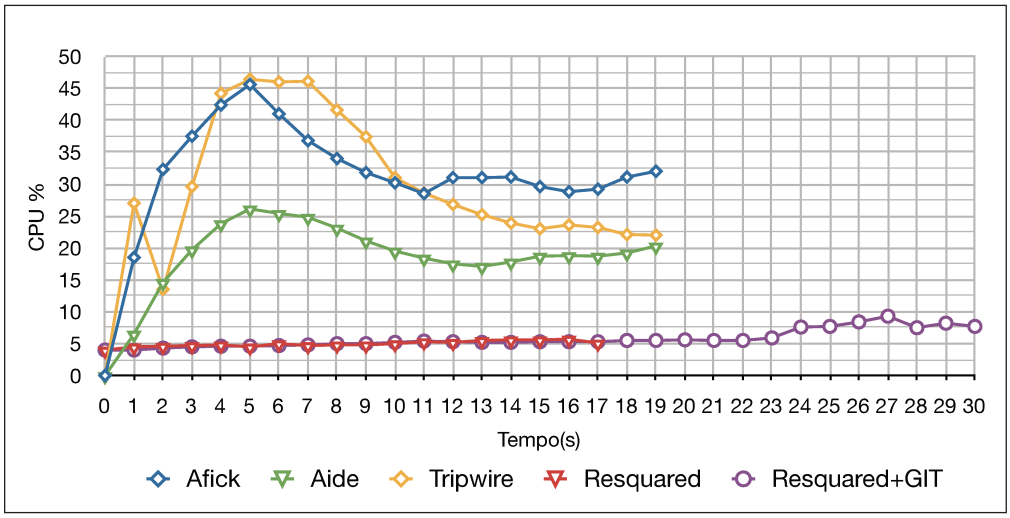


Figura 6.9: Porcentagem de CPU operação de inicialização com diretório `/usr/bin`.

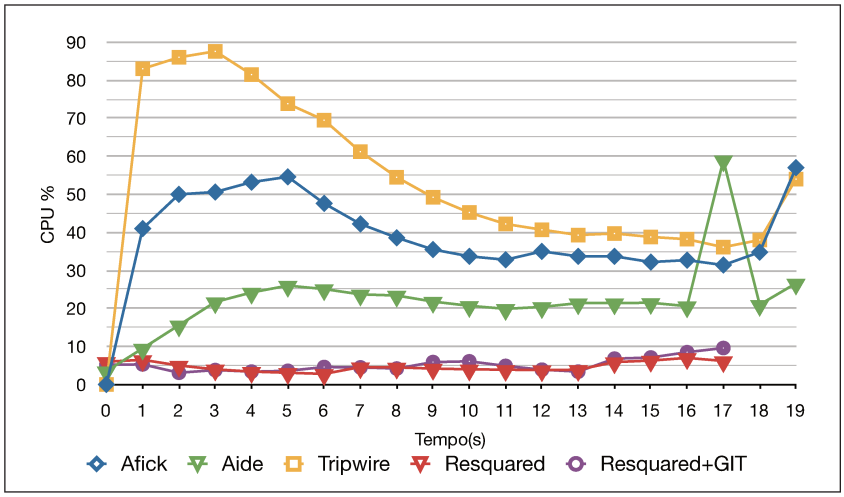


Figura 6.10: Porcentagem de CPU operação de verificação com diretório `/usr/bin`.

6.2 Efetividade de detecção e recuperação

Nesta seção serão apresentados alguns estudos de caso que demonstram a detecção de intrusão e efetiva recuperação da ferramenta. Para cada cenário foi criada uma máquina virtual *Gentoo Linux* 32 bits simulando serviços normais providos por estações de laboratório.

A maioria dos arquivos modificados por *rootkits* são utilitários simples do sistema Linux. Eles são modificados para ocultar evidências e atividades de intrusão. Mudanças comuns são[40]:

- No executável `ps`—para ocultar processos do atacante;
- Em `ls`—para não mostrar arquivos do atacante;
- No comando `netstat`—para esconder conexões e portas abertas pelo atacante.

Para cada cenário estavam sendo monitorados os arquivos recomendados por [40][15] e arquivos relacionados na configuração base de *Tripwire*, que é considerado o primeiro verificador de integridade existente. Fundamentalmente são monitorados diretórios como:

- `/bin`
- `/etc`
- `/usr/bin`
- `/usr/sbin/`

- `/sbin`
- `/root`

Apenas um dos cenários exigiu uma configuração diferente, onde seriam considerados arquivos específicos do *homedir* um usuário, visando a restauração de suas configurações de ambiente gráfico. Mais informações a respeito dos arquivos observados podem ser vistos no Apêndice A.

6.2.1 Uso ilegal de disco

Cenário

Um usuário logado no *host* roda um *exploit* para escalar privilégios—*proto_ ops exploit*[5]—e consegue acesso de super-usuário. O atacante cria uma nova conta de super-usuário denominada *root100*, para conseguir isso faz escrita direta em dois arquivos, `/etc/passwd` e `/etc/shadow`. Logo após, cria um novo diretório e copia 100 arquivos protegidos por leis de *copyright*. Com a intenção de esconder seu armazenamento ilegal instalou uma variante do *t0rn rootkit*. Este *rootkit* substitui arquivos binários do sistema responsáveis por mostrar ao usuário informações do sistema de arquivos e dos processos do sistema. Desta forma o atacante conseguiria esconder seus arquivos do usuário e além disso esconder os processos do usuário *root100*.

Ações corretas de recuperação

Remover todos os arquivos ilegais, os binários falsos e o *homedir* do usuário root100 criado pelo atacante. Além disso é necessário recuperar versão legítima dos arquivos `/etc/passwd` e `/etc/shadow`.

Ponto de detecção

Verificação dos arquivos importados na ferramenta: arquivos alterados em `/etc` e substituição de binários.

Resultados

A ferramenta proposta acusou alteração em 6 binários—`/bin/netstat`,`/bin/ls`, `/bin/ps`, `/usr/bin/top`, `/usr/bin/pstree`, `/sbin/ifconfig`—e nos três diretórios onde eles se

[5]Todo *socket* no *kernel* do *Linux* possui um *struct* chamado *proto_ ops* que contém ponteiros para funções padrões de *socket*, um bug deixava alguns ponteiros sem inicialização dentro do *proto_ ops*, o que permitia execução de código malicioso. Mais em: http://lwn.net/Articles/347006/

encontram. Além de reportar corretamente as alterações nos arquivos `/etc/passwd` e `/etc/shadow`.

Por intermédio da funcionalidade que mostra a diferença entre os arquivos texto infectados e os que estão sobre Controle de Versão—seção 5.3.6, pode ser constatado o superusuário criado pelo atacante. O que indicou uma posterior busca por arquivos "ocultos" criados por este usuário.

Após serem identificados os arquivos alterados, o módulo de comunicação com o gerenciador de pacotes permitiu a identificação de quatro pacotes que necessitariam de reinstalação. Porém um problema surgiu durante a instalação, o atacante havia colocado atributos nos arquivos que não permitiam sua remoção ou substituição. Os atributos de arquivos associam metadados não interpretados pelo sistema de arquivos[56], esses atributos podem representar políticas de escrita, registro de *timestamps*, informações sobre autoria do arquivo entre outros.

Deste modo o gerenciador não era capaz terminar a reinstalação, ao tentar sobescrever os arquivos a instalação parava. O que pode ser evitado se forem armazenados todos os atributos de arquivo no descritor no momento em que os arquivos são importados para o repositório. Ao executar uma restauração, a ferramenta precisa setar os atributos e permissões corretamente—mesmo que no arquivo infectado—para evitar problemas de instalação.

6.2.2 Resiliência das preferências de ambiente gráfico

Cenário

Um usuário típico faz modificações para melhorar seu ambiente gráfico, quando se dá conta de que estragou alguns atalhos gostaria de voltar ao ambiente antigo.

Ações corretas de recuperação

Em ambientes Linux, arquivos de configuração ficam em diretórios ocultos dentro do *homedir* de cada usuário. A maioria das configurações de aplicativos obedece uma hierarquia de configuração, primeiramente são interpretados os arquivos de configuração do usuário, após isso são lidas as configurações padrões do sistema instalado—que o administrador definiu—e por último as configurações padrões do aplicativo. Se forem recuperadas as configurações do usuário pode-se garantir que as configurações padrões do sistema e do aplicativo não foram modificadas, pois o usuário comum não tem permissão para isso.

Ponto de detecção

Usuário percebe que seu ambiente gráfico está desfigurado e pede ao administrador que inicie o processo de recuperação.

Resultados

Primeiramente foram adicionados na ferramenta todos os arquivos ocultos dentro do *homedir* do usuário, quando sua interface gráfica estava com o aspecto representado na Figura 6.11. O tempo de inicialização foi de 5,79 segundos e a quantidade de arquivos no repositório foi de 2,6Mb.

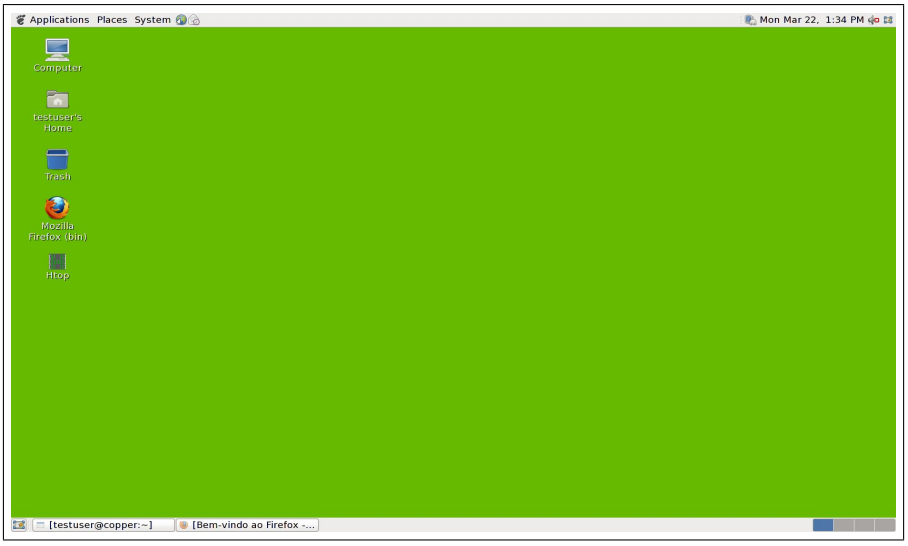


Figura 6.11: Ambiente gráfico no momento de inicialização

Após isso foi simulado o uso do ambiente de forma normal, abrindo páginas na internet, criação de documentos e planilhas e uso de comunicadores instantâneos. Também foram trocadas várias configurações de ambiente e aplicativos como mostra a Figura 6.12.

A recuperação envolveu um total de 69 arquivos, 51 foram restaurados para a versão do repositório e 18 arquivos foram excluídos. A recuperação restaurou todos os menus do gerenciador de ambiente gráfico—*gnome 2.24.1*—e aplicativos como navegador *web*, aplicativos de escritório e comunicadores instantâneos.

Vale ressaltar que a inicialização do repositório levou em consideração apenas os arquivos ocultos dentro do *homedir* do usuário, estes arquivos normalmente são criados por aplicativos e pelo gerenciador do ambiente para guardar as preferências do usuário. O que significa exclusão de arquivos do usuário na restauração, porém em ambientes de produção os administradores geralmente realizam *backups* dos *homedirs* que não consideram

Figura 6.12: Ambiente gráfico no momento da verificação

os arquivos de configuração. Esta prática comum pode ser usada em combinação com o modelo proposto para uma cobertura completa deste estudo de caso.

6.2.3 Infecção de *daemon* de conexão remota

Cenário

Um usuário local consegue acesso privilegiado através de um *exploit* no módulo de *kernel* PulseAudio[6], após isso instala um *backdoor* sobre o OpenSSH[7] para garantir sua reentrada.

Ações corretas de recuperação

Reinstalação do OpenSSH e restauração das configurações deste serviço na máquina vítima.

Ponto de detecção

Verificação de rotina na máquina.

[6]Servidor de som em rede instalado por padrão em várias distribuições. Mais detalhes do *exploit* em: http://www.securityfocus.com/bid/35721/

[7]Coleção de aplicativos que permitem sessões de comunicações cifradas em uma rede de computadores usando o protocolo SSH. http://www.openssh.com/

Resultados

Flea é um *rootkit* para todas as distribuições que contém um utilitário de instalação escrito em `C`. Flea modifica configurações e aplicativos do OpenSSH deixando uma porta aberta para a reentrada do atacante ao sistema infectado. Além disso, sobescreve alguns executáveis comuns da linha de comando com o objetivo de esconder seus processos que estão realizando escuta da rede para sua reentrada ou para obedecer comandos remotos.

Resquared+GIT foi capaz de detectar alterações em 5 binários—/bin/ps; /bin/netstat; /usr/bin/pstree; /usr/bin/du; /usr/bin/slocate—além de alteração do *daemon* do OpenSSH e em suas configurações. A restauração envolveu a reinstalação de 6 pacotes e a restauração de 4 arquivos de configuração referentes ao OpenSSH. Foi necessária a intervenção do administrador apenas para reiniciar o *daemon* sem alterações.

6.3 Conclusão

Neste capítulo foi constatado que a sobrecarga do modelo de recuperação proposto neste trabalho foi de 54% para o processo de inicialização dos dados em comparação com ferramentas comuns de verificação de integridade. Por sua vez, o processo de verificação—que é mais utilizado—ficou próximo ao tempo médio das outras ferramentas. A memória consumida durante a operação da ferramenta não foi afetada pelos métodos adicionais para tornar viável a recuperação. Além disso, o consumo de CPU pela ferramenta proposta foi em média 5 vezes menor do que as outras ferramentas que fazem apenas a verificação de integridade. O baixo consumo de recurso é item crucial para o uso de ferramentas de segurança em servidores, que devem evitar a qualquer custo, problemas ou métodos que possam afetar a disponibilidade de seus serviços.

Observou-se que o uso da ferramenta proposta não é limitado ao sistema operacional instalado, usuários podem usá-la para implantar resiliência no seu ambiente de trabalho através de verificação e restauração dos arquivos de configuração. A ferramenta provê de mecanismos de recuperação automática que identificam os arquivos infectados e realizam uma nova instalação de uma fonte confiável sem necessidade de *backup*. Através da cópia das configurações o comportamento de cada aplicativo configurado pelo admnistrador pode ser restaurado.

Capítulo 7

Considerações Finais

Novas técnicas de intrusão surgem a cada dia, administradores gerenciam centenas de máquinas, usuários não se dão conta de que existem ameaças em *links* acessados e *e-mails* lidos. Essa é a realidade atual dos ambientes de produção, que além de tudo isso está suscetível a problemas que exigem recuperação imediata.

Estudos revelam que uma das mais árduas tarefas de uma administrador de sistema é a recuperação, pois envolve procedimentos manuais e tedisos e que, além disso, consomem muito tempo[58]. Atualmente os custos de recursos humanos superam os custos de recursos de tecnologia, o que sugere o desenvolvimento de soluções de análise e recuperação de sistemas automatizada.

A trajetória de solução de problemas de sistemas frequentemente é acompanhada da afirmação: "funcionava ontem e não está funcionando hoje". O que indica que algo mudou no sistema para ele deixar de funcionar. Encontrar e restaurar o que foi alterado para um estado anterior é, de fato, o princípio da solução do problema. Normalmente problemas deste tipo ocorrem no desenvolvimento de programas, onde um programador sem inteção pode fazer com que o código fonte não compile ou não se comporte como o esperado. Para solucionar este tipo de problema, o programador recorre ao controlador de versão, que é capaz de produzir revisões de código e ter como saída vários estados do código fonte.

Uma vasta quantidade de metodologias é aplicada na detecção de intrusão e ou alteração de sistemas, podendo até haver uma combinação delas trabalhando com o mesmo ideal. Verificadores de integridade são usados para capturar mudanças indesejadas no sistema de arquivos em busca de alterações maliciosas.

Foi descrito e implementado neste trabalho um versátil verificador de integridade de sistema de arquivos chamado *Resquared*, que, traz como principal diferencial, a possibilidade de recuperação do sistema sem necessidade de *backup* dos aplicativos. Através das funcionalidades presentes em controladores de versão, é possível que sejam restaurados estados anteriores das configurações sistema. A recuperação dos aplicativos é confiada

através de comunicação com o gerenciador de pacotes, item comum a todas as distribuições atuais. *Resquared* prevê portabilidade em diferentes distribuições através do módulo de comunicação com o gerenciador de pacotes, que pode ser desenvolvido de acordo com a distribuição onde a ferramenta será utilizada.

Por meio de estudos comparativos constatou-se que *Resquared* teve sobrecarga de aproximadamente 52% na operação de inicialização dos dados de controle—que serão usados em futuras verificações do sistema. O tempo médio da operação de verificação e de consumo de memória foi semelhante entre todas as ferrramentas, porém *Resquared* teve consumo médio de CPU consideravelmente menor. O que leva a conclusão que *Resquared* é mais apropriado ao uso de servidores, que dependem de ferramentas de baixo consumo para não afetarem sua disponibilidade. Três estudos de caso avaliaram o poder de detecção, diagnóstico e recuperação automatizada da ferramenta, que mostrou-se capaz de auxiliar administradores de sistema em tarefas que normalmente seriam exaustivas.

7.1 Trabalhos Futuros

Ao longo do desenvolvimento e testes da ferramenta foram identificados aprimoramentos desejáveis ao funcionamento de *Resquared*. Estas serão descritos a seguir.

7.1.1 Centralização do Repositório entre várias máquinas

A centralização dos dados de controle adicionam uma camada de segurança sobre os dados de controle. Isso pode ser adquirido através dos protocolos de comunicação dos controladores de versão distribuídos, onde cada host terá uma parte do repositório localmente e o repositório contendo os dados de todos os hosts da rede seria centralizado em uma só máquina. Verificações próprias do controle de versão seriam realizadas para garantir a integridade do repositório local.

7.1.2 Interface para controle remoto

Em busca de total centralização da solução, uma interface de controle remoto faz-se necessária para disparar operações da ferramenta por um controlador único. Além disso, um centralizador de relatórios pode ser formado para encontrar relações de ataque em máquinas distintas.

7.1.3 *Daemon* de baixo consumo

Resquared provou ser superior no consumo médio de CPU dentre as outras ferramentas, porém ainda consome uma fatia considerável de CPU para servidores que fornecem grande

quantidade de serviços. Além disso, todas as ferramentas apresentaram uso massivo de leitura e escrita em disco—conforme dito na seção 4.1.1—típico neste tipo de ferramenta. Normalmente as verificações são agendadas para horários em que a máquina é pouco utilizada, de modo a evitar impacto nos serviços oferecidos.

Cgroup [8] fornece mecanismos para criação grupos de processos que obedecerão a regras de limitação do uso de recursos, que podem ser estudados para criação de daemons de verificação com recursos limitados no âmbito de detecção de ataques em "tempo real" e de evitar impacto aos outros processos.

[8] *Control Groups* fornece pequenos módulos ao kernel responsáveis pela agregação/particionamento de grupos hieráquicos de processos e seus futuros processos filhos. O objetivo geral é a implantação de regras de comportamento para estes grupos.

Apêndice A

Arquivos importantes do sistema Linux

Este apêndice comenta sobre alguns dos diretórios mais importantes do sistema *Linux*, além de mostrar alguns arquivos críticos para mostrar ao leitor como um atacante pode se aproveitar de alterações em arquivos e configurações do sistema. A listagem dos arquivos não significa que somente eles devem ser observados para evitar ou detectar ataques.

Arquivos do `/bin`

Contém comandos acessíveis tanto pelo administrador quanto pelo usuário. Também existem neste diretório comandos que são usados indiretamente por *scripts* do sistema;

- `/bin/chmod`—utilitário para mudar as permissões de um arquivo;
- `/bin/df`—utilitário para mostrar o uso de espaço em disco;
- `/bin/dmesg`—imprimir e controlar as mensagens do *kernel* ;
- `/bin/su`—utilitário para mudar o *id* do usuário;
- `/bin/ps`—mostra estados de todos os processos.

Existem ainda muitos outros comandos para manipular diretórios, trocar o dono de um arquivo, mover arquivos, imprimir o conteúdo de um arquivo etc, que podem ser modificados para esconder informação do usuário, ou ainda realizar operações sem que o usuário perceba.

Configurações e outros arquivos do `/etc`

O diretório `/etc` possui vários arquivos de configuração, que são usados para controlar a operação de um determinado aplicativo.

- `/etc/exports`—lista de controle de acesso para o NFS—Network FileSystem;
- `/etc/fstab`—define pontos de montagem para dispositivos de armazenamento;
- `/etc/inittab`—configurações que definem o comportamento do utilitário `init`, que é usado para carregar todo o ambiente durante o *boot*;
- `/etc/passwd`—arquivos contendo as senhas de todos os usuários;
- `/etc/ftpusers`—lista de controle de acesso ao *daemon* FTP.

Alterações neste diretório podem fazer com que o comportamento de um aplicativo esconda informações do Administrador, ou ainda, permitir que o atacante possa acessar o sistema sem explorar nenhuma vulnerabilidade.

Binários do sistema, diretório `/sbin`

Utilitários usados para administração do sistema—e outros comandos acessíveis apenas pelo root—são armazenados em `/sbin`. Este diretório contém binários essenciais para o processo de *boot*, recuperação e reparo do sistema.

- `/sbin/ifconfig`—mostra informações e configura interfaces de rede da máquina;
- `/sbin/route`—mostra informações sobre a tabela de rotas da máquina.

Mais uma vez a modificação de binários desta pasta pode esconder informações importantes ao administrador.

Arquivos críticos para boot

- `/boot/`—diretório que contém a maioria dos arquivos usados no *boot* da máquina;
- `/boot/kernel.image`—o nome do arquivo pode variar—imagem do *kernel* compilado e comprimido, será carregado na subida do sistema;
- `/boot/grub`—também conhecido como *bootloader*—executável responsável por carregar o *kernel*;
- `/sbin/init`—após ser completamente carregado o *kernel* desvia o processo de *boot* para este executável, que é responsável por configurar o restante do ambiente.

Alterações nestes e em outros arquivos presentes no diretório `/boot` podem representar ameaças no processo de *boot* dos sistema, onde módulos podem ser carregados sem permissão. Além disso, código não autorizado pode ser inserido na imagem do kernel.

Arquivos do `/dev`

O diretório `/dev` contém arquivos especiais para todos os dispositivos da máquina. Estes arquivos são criados durante a instalação, posteriormente podem ser atualizados por meio de um *script* localizado em `/dev/MAKEDEV`. Dentre os arquivos deste diretório podem ser observados:

- `/dev/kmem`—basicamente dá acesso a região de memória ocupada pelo *kernel* em execução;
- `/dev/mem`—dá acesso a memória física da máquina;
- `/dev/null`;
- `/dev/zero`;

Arquivos do `/dev` podem ser sobescritos com intuito de capturar a saída de processos do sistema.

Arquivos do `/proc`

O *kernel* Linux tem duas principais funções: controlar o acesso aos dispositivos do computador e escalonar o acesso dos processo à esses dispositivos. O diretório /proc/—também chamado proc *filesystem*—contém um conjunto de arquivos especiais que representam o estado corrente do *kernel*. Dentro do diretório /proc podem ser encontrados detalhes de hardware e de qualquer processo em execução.

A seguir veremos alguns arquivos que podem ser observados dentro do diretório /proc e a função de cada um:

- `/proc/devices`—contém a lista de dispositivos do sistema;
- `/proc/net`—diretório que contém vários parâmetros de rede da máquina;
- `/proc/sys/`—informações váriadas do sistema, controle de cache, sistemas de arquivos, *RAID* etc;
- `/proc/cpuinfo`—identifica o processador usado pela máquina;
- `/proc/modules`—contém a lista de módulos carregados;
- `/proc/mounts`—uma lista contendo todos sistemas de arquivos montados;
- `/proc/filesystems`—lista contendo os tipos de sistemas de arquivos suportados pelo *kernel*;

- `/proc/pci`—lista de todos os dispositivos PCI da máquina;

Alguns arquivos dentro do diretório `/proc` podem acusar ataques físicos, onde atacantes podem instalar dispositivos nas máquinas.

Referências Bibliográficas

[1] *SubVirt: Implementing malware with virtual machines*, Washington, DC, USA, 2006. IEEE Computer Society.

[2] S. Auvray. Distributed version control systems: A not-so-quick guide through. *InfoQ Articles*, 2008.

[3] A. Avizienis. Design of fault-tolerant systems. *AFIPS Conference Proceedings*, 31:733–743, 1967.

[4] Mohammad Banikazemi, Dan Poff, and Bulent Abali. Storage-based file system integrity checker. In *StorageSS '05: Proceedings of the 2005 ACM workshop on Storage security and survivability*, pages 57–63, New York, NY, USA, 2005. ACM.

[5] Wendy Bartlett and Lisa Spainhower. Commercial fault tolerance: A tale of two systems. *IEEE Transactions on Dependable and Secure Computing*, 1:87–96, 2004.

[6] Stefan Berger, Ramón Cáceres, Kenneth A. Goldman, Ronald Perez, Reiner Sailer, and Leendert van Doorn. vtpm: virtualizing the trusted platform module. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2006. USENIX Association.

[7] Brian Berliner, Prisma Inc, and Mark Dabling Blvd. Cvs ii: Parallelizing software development. 1990.

[8] Thomas Dale Bissett, Richard D. Fiorentino, Robert M. Glorioso, James D. Mccauley, Diane T. Mccollum, Glenn A. Tremblay, and Mario Troiani. Fault resilient/fault tolerant computing. United States Patent 6038685, May 2000.

[9] B. Collin-sussman, B. W. Fitzpatrick, and C. M. Pilato. Version control with subversion. online, 2007.

[10] Wesley D. Craig and Patrick M. McNeal. Radmind: The integration of filesystem integrity checking with filesystem management. In *LISA '03: Proceedings of the 17th*

USENIX conference on System administration, pages 1–6, Berkeley, CA, USA, 2003. USENIX Association.

[11] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, 2002.

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

[13] P Feiler. Configuration management models in commercial environments. Technical Report CMU/SEI-91-TR-7 ESD-9-TR-7, Software Engineering Institute. Carnegie Mellon, University. Pittsburgh, 1991.

[14] N. Gift and A. Shand. Introduction to distributed version control systems. IBM Technical library, April 2009.

[15] A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara. The taser intrusion recovery system. *SIGOPS Oper. Syst. Rev.*, 39(5):163–176, 2005.

[16] Ashvin Goel, Wu-chang Feng, David Maier, Wu-chi Feng, and Jonathan Walpole. Forensix: A robust, high-performance reconstruction system. In *ICDCSW '05: Proceedings of the Second International Workshop on Security in Distributed Computing Systems (SDCS) (ICDCSW'05)*, pages 155–162, Washington, DC, USA, 2005. IEEE Computer Society.

[17] Trusted Computing Group. Tcg tpm specification version 1.2. Trusted Computing Group Specification.

[18] L T Heberlein, B Mukherjee, and K N Levitt. Internet security monitor: An intrusion detection system for large-scale networks. In *In Proceedings of the 15th National Computer Security Conference*, 1992.

[19] Judith Hochberg, Kathleen Jackson, Cathy Stallings, J. F. McClary, David DuBois, and Josephine Ford. Nadir: an automated system for detecting network intrusion and misuse. *Comput. Secur.*, 12(3):235–248, 1993.

[20] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls, 1998.

[21] Francis Hsu, Hao Chen, Thomas Ristenpart, Jason Li, and Zhendong Su. Back to the future: A framework for automatic malware removal and system repair. In *ACSAC*

'06: Proceedings of the 22nd Annual Computer Security Applications Conference, pages 257–268, Washington, DC, USA, 2006. IEEE Computer Society.

[22] Pankaj Jalote. *Fault tolerance in distributed systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[23] Donald Johnson and Scott A. Vanstone. A resilient cryptographic scheme. World Intellectual Property Organization, Pub. No.: WO/2000/044129, 2000.

[24] Kimo Kasslin and Elia Florio. Your computer is stoned...(again!). the rise of mbr rootkits. In *Virus Bulletin Conference*, 2008.

[25] Gene H. Kim and Eugene H. Spafford. The design and implementation of tripwire: a file system integrity checker. In *CCS '94: Proceedings of the 2nd ACM Conference on Computer and communications security*, pages 18–29, New York, NY, USA, 1994. ACM.

[26] R. Baldwin Kuang. Rule based security checking. Technical report, Technical report, MIT Lab for Computer Science, Programming Systems Research Group, May 1994.

[27] Samhain Labs. Samhain: File system integrity checker. http://samhain.sourceforge.net.

[28] P. A. Lee and T. Anderson. *Fault Tolerance: Principles and Practice*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1990.

[29] John Levine, Julian Grizzard, and Henry Owen. A methodology to detect and characterize kernel level rootkit exploits involving redirection of the system call table. *Information Assurance, IEEE International Workshop on/Innovative Architecture for Future Generation High-Performance Processors and Systems, International Workshop on/Innovative Architecture, International Workshop on*, 0:107, 2004.

[30] J. Mantha. Bzr, git, and hg performance on the linux tree. May 2008.

[31] Robert McQuarrie, Maarit Palo, and Anne Suursalmi. Network resiliency. IBM Global Services, May 2002.

[32] Steve Mead. Unique file identification in the national software reference library. *Digital Investigation*, 3(3):138 – 150, 2006.

[33] R Miller. Confirguration management with subversion, yaml and perl template tollkit. *Proceedings. 5th System Administration and Network Engineering Conference*, 2006.

[34] E. T Nakamura and P. L. Geus. *Segurança de Redes em Ambientes Cooperativos*. 2007.

[35] J. V. Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. Automata Studies, (Annals of Math Studies No. 34), 1956.

[36] University NewCastle. Msc in computer security and resilience degree programme handbook. School of Computing Science at NewCastle University 2007.

[37] Osiris. Osiris: Host integrity management tool, www.osiris.com.

[38] Swapnil Patil, Anand Kashyap, Gopalan Sivathanu, and Erez Zadok. Ifs: An in-kernel integrity checker and intrusion detection file system. In *LISA '04: Proceedings of the 18th USENIX conference on System administration*, pages 67–78, Berkeley, CA, USA, 2004. USENIX Association.

[39] Fabrício S. Paula. *Uma arquitetura de segurança computacional inspirada no sistema imunológico*. Tese de doutourado, Instituto de Computação – UNICAMP, 2004.

[40] Adam G. Pennington, John D. Strunk, John Linwood Griffin, Craig A. N. Soules, Garth R. Goodson, and Gregory R. Ganger. Storage-based intrusion detection: watching storage activity for suspicious behavior. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 10–10, Berkeley, CA, USA, 2003. USENIX Association.

[41] Lids Project. Linux intrusion dectection system. www.lids.org.

[42] Laura L. Pullum. *Software fault tolerance techniques and implementation*. Artech House, Inc., Norwood, MA, USA, 2001.

[43] Marcus Ranum. Coverage in intrusion detection systems. Technical report, NFR Security, June 2001.

[44] J. Reed. File integrity with aide. www.iforkr.org/bri/presentations/aide, 2003.

[45] Marc J. Rochkind. The source code control system. *IEEE Trans. Software Eng.*, 1(4):364–370, 1975.

[46] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding when to forget in the elephant file system. *SIGOPS Oper. Syst. Rev.*, 33(5):110–123, 1999.

[47] Steve Schupp. Limitations of network intrusion detection. SANS Institute, December 2000.

[48] Baiju Shah. How to choose introduction detection solution. Whitepaper, SANS Institute, July 2001.

[49] Gopalan Sivathanu, Charles P. Wright, and Erez Zadok. Ensuring data integrity in storage: techniques and applications. In *StorageSS '05: Proceedings of the 2005 ACM workshop on Storage security and survivability*, pages 26–36, New York, NY, USA, 2005. ACM.

[50] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, and Gregory R. Ganger. Metadata efficiency in versioning file systems. In *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 43–58, Berkeley, CA, USA, 2003. USENIX Association.

[51] G. Stolf. Validação da técnica de tolerância a falhas de software programação n-cópia através da utilização de injeção de falhas em uma aplicação industrial. Monografia do Curso de Ciência da Computação da Universidade de Passo Fundo. Passo Fundo - RS, 2007.

[52] John D. Strunk, Garth R. Goodson, Adam G. Pennington, Soiles A. N. Craig, and Gregory R. Ganger. Intrusion detection, diagnosis, and recovery with self-securing storage. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh. CMU-CS-02-140, 2002.

[53] T. Suel and N. Menon. Algorithms for delta compression and remote file synchronization, 2002.

[54] Weiqing Sun, Zhenkai Liang, R. Sekar, and V. N. Venkatakrishnan. One-way isolation: An effective approach for realizing safe execution environments. In *In Proceedings of the Network and Distributed System Security Symposium*, pages 265–278, 2005.

[55] Walter F. Tichy. Rcs - a system for version control. *Softw., Pract. Exper.*, 15(7):637–654, 1985.

[56] Avishay Traeger, Erez Zadok, Nikolai Joukov, and Charles P. Wright. A nine year study of file system and storage benchmarking. *Trans. Storage*, 4(2):1–56, 2008.

[57] Leendert van Doorn, Van Doorn, Gerco Ballintijn, and William A. Arbaugh. Signed executables for linux. Tech. report cs-tr-4259, University of Maryland, 2001.

[58] Helen J. Wang, John Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. Peerpressure for automatic troubleshooting. *SIGMETRICS Perform. Eval. Rev.*, 32(1):398–399, 2004.

[59] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full sha-1. In *In Proceedings of Crypto*, pages 17–36. Springer, 2005.

[60] Yanxin Wang. *A hybrid intrusion detection system.* PhD thesis, Ames, IA, USA, 2004. Major Professor-Wong, Johnny S.

[61] Andy Watson, Paul Benn, and G. Alan Yoder. Multiprotocol data access: Nfs, cifs and http. Technical report, Technical Report TR3014 Network Appliance, 1999.

[62] T. S. Weber. Tolerância a falhas: conceitos e exemplos. 2001.

[63] B. Westerbaan. git's versus svn's storage efficiency. CodeYard Documents, July 2008.

[64] D. A. Wheeler. Free software (oss/fs) software configuration management (scm) / revision-control systems. Comments on Open Source Software, May 2008.

[65] J. WIEGLEY. Git from from the bottom up. E-book, 2008.

[66] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *Proceedings of the 11th USENIX Security Symposium*, pages 17–31, Berkeley, CA, USA, 2002. USENIX Association.

[67] Maria Angela Mattar Yunes and Heloísa Szymanski. *Resiliência: noção, conceitos afins e considerações.* TAVARES, J. (Org.). Resiliência e Educação., 2001.

[68] L. A. Zanardo, M. M. Haiduck, and W. Pavan. Estudo das técnicas bloco de recuperação e de programação n-versões. 1999.

[69] Youhui Zhang, Hongyi Wang, Yu Gu, and Dongsheng Wang. Idrs: Combining file-level intrusion detection with block-level data recovery based on iscsi. In *ARES '08: Proceedings of the 2008 Third International Conference on Availability, Reliability and Security*, pages 630–635, Washington, DC, USA, 2008. IEEE Computer Society.

[70] Ningning Zhu and Tzi-Cker Chiueh. Design, implementation, and evaluation of repairable file service. *Dependable Systems and Networks, International Conference on*, 0:217, 2003.