Curso de Ciência da Computação
Universidade Estadual de Mato Grosso do Sul

MONITORAMENTO DE PROCESSOS COM SYSTEMTAP

Felipe da Silva Lima

Prof. Dr. Fabrício Sérgio de Paula(Orientador)

Dourados - MS
2018

MONITORAMENTO DE PROCESSOS COM SYSTEMTAP

Felipe da Silva Lima

Este exemplar corresponde à redação final da monografia da disciplina Projeto Final de Curso devidamente corrigida e defendida por Felipe da Silva Lima e aprovada pela Banca Examinadora, como parte dos requisitos para a obtenção do título de Bacharel em Ciência da Computação.

Dourados, 23 de Novembro de 2018.

Prof. Dr. Fabrício Sérgio de Paula
(Orientador)

Curso de Ciência da Computação
Universidade Estadual de Mato Grosso do Sul

MONITORAMENTO DE PROCESSOS COM SYSTEMTAP

Felipe da Silva Lima

Novembro de 2018

Banca Examinadora:

- Prof. Dr. Fabrício Sérgio de Paula (Orientador)
- Prof. Dr. Cleber Valgas Gomes Mira
  Área de Computação - UEMS
- Prof. Dr. Nilton Cézar de Paula
  Área de Computação - UEMS

Agradecimentos

Agradecimentos vão a Deus, ao meu Pai (*in memorian*), Mãe, Irmão e demais familiares.
A todos os Professores do Curso de Ciência da Computação da UEMS - Dourados/MS por dedicarem suas vidas a nos passar conhecimento.
Ao Professor Orientador deste projeto por sua excelentíssima orientação, paciência e por seu depósito de confiança.
À UEMS.
A todos meus amigos.

Resumo

A *SystemTap* é uma ferramenta para monitoramento e rastreamento do *Kernel* Linux que utiliza uma API (Interface de Programação de Aplicações) chamada Kprobes. Através de simples *scripts* a ferramenta permite rastrear e obter dados de qualquer chamada ao sistema, processo ou função do código do *kernel*.

Este trabalho propõe melhorar o desenvolvimento de um IDS (Sistema de Detecção de Intrusão) que usa a SystemTap para obter os dados sobre os processos/Linux previamente descritos em uma política de monitoramento e verifica sua execução de acordo com o que foi definido na política para cada programa desenvolvido anteriormente pelos mesmos autores desse trabalho. Para cada programa é especificado o que é permitido como por exemplo, acesso de leitura e/ou escrita em diretório, ou arquivo, programas que podem ser executados, portas de rede que podem ser usadas e criação de processos-filhos.

Palavras-chave: Sistema de detecção de intrusão, IDS, *SystemTap*

Conteúdo

Capítulo 1

Introdução

Aplicações computacionais se tornaram tão populares a ponto de estarem em todos os lugares e situações. Desde instituições bancárias e grandes indústrias ao comércio em geral, muitas operações que antes eram feitas de maneira manual ou mecânica, hoje são realizadas de forma eletrônica.

Com essa gigantesca quantidade de informação sendo processada, um ponto importante é a segurança da informação. Para isso, diversos sistemas de segurança são utilizados, como anti-vírus, *firewalls* e IDS's (Sistemas de Detecção de Intrusões), por exemplo. Segundo (Bishop 2002), um IDS pode ser definido como um sistema que automatiza a detecção de uma grande variedade de intrusões. O tema deste trabalho engloba o desenvolvimento e avaliação da eficiência de um IDS baseado em monitoramento de processos utilizando *scripts SystemTap*.

1.1 Objetivo geral

O objetivo geral deste projeto é utilizar a ferramenta *SystemTap* para construir um protótipo de um IDS baseado no monitoramento de processos do sistema operacional Linux.

1.1.1 Objetivos Específicos

Os objetivos específicos são:

- Estudar os conceitos que estão relacionados ao tema do trabalho: o que é um IDS, aspectos básicos de segurança da informação e estudo da ferramenta SystemTap.
- Verificar a viabilidade de utilizar a ferramenta *SystemTap* na construção de um IDS com monitoramento de processos.

O Capítulo 2 apresenta a literatura consultada neste projeto. No Capítulo 3 é apresentado o sistema detecção proposto. No Capítulo 4 são apresentados os resultados alcançados no desenvolvimento e o Capítulo 5 conclui este trabalho.

Capítulo 2

Revisão da Literatura

2.1 Segurança computacional

Esta seção apresenta conceitos e definições acerca da segurança da informação de acordo com (Bishop 2002). que define a segurança computacional com base em três componentes básicos: confidencialidade, integridade e disponibilidade.

2.1.1 Confidencialidade

Confidencialidade é o sigilo absoluto da informação ou dado que está sendo tratado. Isso se faz necessário quando tratamos por exemplo, de dados de agentes como governo, instituições militares e indústrias. Por exemplo, geralmente o acesso aos dados de instituições militares é permitido para aqueles que precisam dessa informação.

Um controle de acesso, por exemplo, é um mecanismo que permite a implementação de confidencialidade. O uso de criptografia, que a um grosso modo seria embaralhar os dados para que sejam incompreensíveis, pode ser citado como uma maneira de preservar a confidencialidade da informação em questão.

Outra abordagem para impedir que os dados sejam acessados ilegalmente é o uso de mecanismos dependentes do sistema, algo como um *software* que gerencia o acesso a informação. Em contra-partida, caso esse mecanismo venha a falhar, os dados ficam visíveis.

Por trás dos mecanismos de confidencialidade temos então suposição e confiança: suposição de que os serviços de segurança podem confiar tanto no *Kernel* quanto em outros agentes para fornecer os dados corretamente.

2.1.2 Integridade

A integridade remete à fidedignidade dos dados ou recursos, garantindo que esses dados e/ou recursos não sejam alterados inapropriadamente. A integridade é dividida em:

- Integridade dos dados: garante a integridade dos dados propriamente ditos;
- Integridade da origem: garante a origem propriamente dita dos dados.

A integridade possui também duas classes de mecanismos:

- Prevenção: busca manter os dados íntegros através do bloqueio da alteração não autorizada dos dados ou alteração de forma não autorizada. Essas duas maneiras se divergem em sua síntese. A primeira ocorre por exemplo, quando um usuário tenta alterar dados os quais não têm permissão. O segundo ocorre quando o usuário tem permissão para alterar os dados de uma determinada forma, mas faz de outra maneira.
- Detecção: os mecanismos detectores não impedem violações, apenas relatam que os dados não são confiáveis.

Tratar a integridade diverge de tratar a confidencialidade. A confidencialidade é comprometida com apenas um acesso indevido. A integridade envolve a confiabilidade da informação e/ou recurso protegido.

2.1.3 Disponibilidade

A disponibilidade diz respeito também a capacidade de fazer uso das informações ou dos recursos desejados. A disponibilidade está ligada a confiabilidade, pois se o sistema não está disponível, este é tão ruim quanto não existir sistema algum. A disponibilidade se torna importante para a segurança no contexto de que alguém pode propositalmente tornar o serviço ou o acesso aos dados indisponível.

Ao projetar um sistema, usualmente é feito um modelo estatístico do padrão de uso esperado e há mecanismos que garantem a disponibilidade quando o modelo estatístico é válido. Se alguém manipula parâmetros de uso como o tráfego de rede, o modelo estatístico pode não ser mais válido.

Ataques de negação de serviço, quando os recursos de uma sistema se tornam indisponíveis, por exemplo, são difíceis de se detectar se os padrões de acesso incomuns podem ser manipulados por atacantes, modificando as próprias variações medidas nos eventos considerados válidos.

2.1.4 Políticas e Mecanismos

Um ponto importante no estudo da segurança computacional é a diferença entre política e mecanismo de segurança. Uma política de segurança é uma declaração do que é e o que não é permitido.(Bishop 2002)

Um mecanismo de segurança é um método, ferramenta ou procedimento para impor uma diretiva de segurança. Mecanismos podem tanto ser técnicos como não ser. Ao ser técnico, pode exigir uma autenticação de identidade para por exemplo, alterar uma senha.(Bishop 2002)

As políticas podem ser apresentadas de uma maneira formal, como uma lista de estados permitidos (seguros) e não permitidos (não seguros). Na prática, raramente as políticas serão precisas, e normalmente descrevem em linguagem humana o que usuário pode ou não fazer. Por essa característica, a descrição pode se tornar ambígua e chegar à estados que não são permitidos.

Tendo a política de segurança com as ações consideradas seguras e as inseguras, mecanismos de segurança podem então impedir, detectar ou recuperar-se de um ataque.

Prevenir significa que o ataque não será efetuado. Os mecanismos preventivos podem interferir no uso normal do sistema, mas alguns mecanismos mais simples, como senhas, se tornaram aceitos sem maiores problemas.

A detecção se torna útil quando o ataque já não pode ser evitado. O mecanismos de detecção aceitam que um ataque ocorra. Seu objetivo é saber que um ataque está em andamento ou se houve um ataque e então denunciá-lo. Um monitoramento do ataque pode ser feito para obter dados da natureza, gravidade e os resultados do ataque. Um exemplo desse mecanismo é um aviso dado a um usuário quando este digita uma senha errada três vezes.

A recuperação possui duas maneiras de ser executada. A primeira é parar um ataque e avaliar e reparar todo e qualquer dano que esse ataque possa ter causado. Com isso, a recuperação se torna muito mais complexa devido a natureza singular de cada ataque. Além disso, a correção das vulnerabilidades exploradas também está envolvida na recuperação. A segunda maneira de recuperação é o sistema possuir tolerância à falhas. Ou seja, mesmo com um ataque em andamento, o sistema funciona normalmente. Nessa forma de recuperação, o sistema nunca opera incorretamente, mas devido à complexidade dos sistemas, a implementação desse tipo de recuperação é muito difícil.

2.1.5 Mecanismos para a segurança computacional

Esta seção apresenta os seguintes exemplos mecanismos para segurança computacional, de acordo com (Lehtinen 2006): criptografia, *firewalls*, anti-vírus e treinamento pessoal.

2.1.6 Criptografia

A criptografia é o ato de transformar informações ou dados originais, através de algoritmos baseados em premissas matemáticas, em uma cifra, que geralmente possui a aparência de dados aleatórios e ininteligíveis. Sua eficiência se baseia na dificuldade em se reverter a criptografia sem que se possua a chave para decifrar a informação.

Com a criptografia é possível atingir dois dos três componentes básicos da segurança: confidencialidade e integridade.

A criptografia se torna uma maneira muito boa de manter os dados confidenciais, pois mesmo que o dado criptografado seja exposto, será difícil decifrar a informação contida no dado.

Para garantir a integridade são usados tipos de algoritmos de criptografia que inibem a adulteração. Aplicável em *softwares* de instituições financeiras para garantir por exemplo, que um ponto decimal não esteja faltando. Em redes, garantir que a comunicação não tenha sido afetada. Na integridade também é possível garantir especificamente a integridade da origem que também é referenciada por autenticidade. Nesse ponto a criptografia garante que as informações são verdadeiras, e vêm de onde realmente deveriam vir.

2.1.7 *Firewalls*

Um *firewall* protege um sistema examinando cada pacote de dados ou informações que trafegam pela rede. O propósito de um pacote pode ser descoberto a partir do endereço de destino desse pacote. Uma lista de destinos e funções permitidas e não permitidas é mantida pelo *firewall*. Se um pacote vem de ou vai para um endereço não permitido, o *firewall* se encarrega de descartá-lo. *Firewalls* mais avançados também podem controlar os pacotes que saem ou chegam e os permitem apenas se esse pacote era esperado em um determinado momento da comunicação.

2.1.8 Anti-vírus

Um vírus no âmbito de segurança computacional pode ser definido como um trecho de código que copia a si mesmo para um determinado programa. Logo, um vírus é um programa que depende da execução de um programa hospedeiro para ser executado. Ao ser executado, ele se replica em outros programas, além de executar o código malicioso.

Dessa forma, *softwares* que trabalham protegendo um sistema, utilizando técnicas para detectar vírus, são os chamados anti-vírus. Para isso, possuem duas técnicas principais: detecção por assinatura e por heurística, que são definidas a seguir.

- Assinaturas: padrões de código do vírus. Quando o anti-vírus detecta um padrão que se alinha com alguma assinatura armazenada, ele gera um alerta e faz um isolamento

desse código, podendo ser feita através de uma varredura periódica ou em tempo real. Porém, essa abordagem possui problemas. Anti-vírus baseados em assinatura necessitam um fluxo constante de novas assinaturas para responder à evolução dos ataques. Isso então se torna um problema à medida que o número de vírus aumenta e o banco de assinaturas fica maior (um problema particular de dispositivos com pouca memória). Outro problema pode ser de a vítima ser atingida por um vírus do qual ela ainda não recebeu a assinatura. Um terceiro problema é da capacidade de mutação do vírus o que altera a sua assinatura.

- Detecção heurística: uma heurística pode ser definida como uma regra ou comportamento. Quando o vírus exibe esse comportamento, o anti-vírus tenta pará-lo. Por exemplo, um trecho de código que acessa uma região crítica do disco. Outros comportamentos podem ser alterações sem explicação no tamanho ou alterações de data, em especial em arquivos do sistema.

2.1.9 Treinamento pessoal

A falha de segurança mais complexa de ser resolvida talvez seja a falha a partir da intervenção humana. Seja por que um usuário autorizado a usar o sistema tenha passado suas credenciais para se autenticar no sistema para outra pessoa, permitindo assim que um usuário não cadastrado possa acessar o sistema com um menor possibilidade de ser detectado. Ou seja por conta de um funcionário descontente que faz então o uso indevido de seus privilégios.

Além desses problemas, a falta de treinamento também é uma ameaça à segurança de um sistema. Usar senhas fáceis de adivinhar, por exemplo, é uma prática que tornam sistemas inseguros.

Esse treinamento não é focado apenas na área técnica. Ataques bem-sucedidos podem ser feitos usando a engenharia social. Por exemplo, fazer uma ligação telefônica para algum funcionário que pode mudar senhas via telefonema e se passar por algum outro funcionário, por exemplo alguém com cargo mais alto como um diretor-geral. Se essa senha for alterada, o ataque de engenharia social foi bem sucedido e o atacante pode então acessar o sistema com a nova senha.

2.2 Sistemas de detecção de intrusão

Esta seção apresenta conceitos de sistemas de detecção de intrusão que estão de acordo com (Bishop 2002).

Quando um sistema de computador está sob um ataque, algumas características podem ser percebidas. As principais são:

- Os processos e as ações executadas pelos usuários geralmente seguem um padrão previsível estatisticamente.
- Usuários e processos não executam ações para desfazer a política de segurança do sistema. Na teoria, qualquer sequência desse tipo é impedida. Na prática, somente as conhecidas podem ser detectadas.
- Um conjunto de especificações descrevem as ações que os processos podem ou não fazer.

De acordo com (Bishop 2002), um sistema de detecção de intrusão(IDS: *Intrusion Detection Systems*) deve executar quatro ações:

1. Detectar uma grande variedade de intrusões. Tanto intrusões internas quanto externas são relevantes. Ataques conhecidos e não conhecidos devem ser detectados. Isso então sugere a capacidade para se adaptar aos ataques.
2. Detectar intrusões em tempo hábil. O suficiente é descobrir uma intrusão em um curto período de tempo.
3. Mostrar o resultado da análise de uma forma fácil de entender. Por exemplo, uma luz verde para nenhuma intrusão e uma luz vermelha quando um ataque é detectado.
4. Ser preciso. Isso se refere à confiança do IDS em produzir falsos positivos, isto é, quando é gerado um alerta de ataque mas não existem ataques em andamento. Falsos negativos ocorrem quando o sistema não detecta a intrusão que está ocorrendo. O objetivo então é minimizar os dois tipos de erros.

2.3 Sistemas de detecção de intrusão usando *SystemTap*

2.3.1 Conceitos de sistemas operacionais

Esta seção apresenta conceitos relacionados a chamadas ao sistema operacional, com enfoque no sistema operacional Linux, relevante para este Projeto Final de Curso. Esses conceitos são apresentados de acordo com (Silberschatz 2010).

Sistema Operacional (SO) é um programa que gerencia o *hardware* (parte física) do computador e também dá o alicerce para os aplicativos, agindo como um mediador entre

o usuário e o *hardware* do computador. Os sistemas operacionais podem assumir diferentes abordagens ao cumprir essas tarefas. Os sistemas operacionais de *mainframe*, por exemplo, são projetados basicamente para otimizar a utilização do *hardware*. Já os sistemas operacionais dos computadores pessoais devem suportar aplicações diversificadas, tais como para edição de textos e planilhas, multimídia, aplicações comerciais e, inclusive, jogos. Há também os sistemas operacionais para dispositivos móveis (ex.: celulares e *tablets*), que são projetados a oferecer um ambiente no qual o usuário possa interagir facilmente com o dispositivo. Dessa forma, há sistemas operacionais projetados para serem convenientes, outros para serem eficientes e outros para ambos os aspectos

Do ponto de vista do computador, o sistema operacional é o programa mais ligado ao *hardware*. Nesse contexto, pode-se considerar um sistema operacional como um alocador de recursos. Um sistema de computação tem muitos recursos que podem ser necessários à resolução de um problema: tempo de CPU, espaço de memória, espaço de armazenamento em disco, acesso a dispositivos de E/S, etc. O sistema operacional atua, então, como o gerenciador desses recursos. Ao lidar com solicitações de recursos numerosas e possivelmente concorrentes, o sistema operacional precisa decidir como alocá-los a programas e usuários específicos para poder operar o sistema de computador de maneira eficiente e justa.

Sistemas operacionais existem porque eles representam uma maneira razoável de resolver o problema de criar um sistema de computação utilizável. O objetivo fundamental dos sistemas de computação é executar os programas dos usuários e facilitar a resolução dos seus problemas. É com esse objetivo que o *hardware* do computador é construído. Os programas aplicativos são desenvolvidos tendo em vista que o *hardware* puro não é particularmente fácil de ser utilizado. Esses programas requerem determinadas operações comuns, como as que controlam os dispositivos de E/S. As funções comuns de controle e alocação de recursos são então reunidas em um *software*: o sistema operacional.

Sistemas de computação modernos são geralmente compostos por uma ou mais CPUs ou núcleos de processamento, e vários controladores de dispositivos conectados através de um barramento comum que proporciona acesso a memória compartilhada. Para que um computador comece a operar — por exemplo, quando é ligado ou reiniciado — precisa dispor de um programa inicial para executar. Esse programa inicial, ou programa *bootstrap*, tende a ser simples. Para alcançar esse objetivo, o programa deve alocar e carregar na memória o *kernel* (núcleo) do sistema operacional. O sistema operacional, então, começa a executar o primeiro processo, e aguarda que algum evento ocorra. Geralmente o acontecimento de um evento é indicado por uma interrupção proveniente tanto do *hardware* como do *software*.

O *hardware* pode provocar uma interrupção a qualquer momento enviando um sinal à CPU, normalmente através do barramento do sistema. O *software* pode provocar uma

interrupção executando uma operação especial denominada chamada de sistema.

Chamadas de sistema

As chamadas de sistema fornecem uma interface com os serviços disponibilizados por um sistema operacional. Geralmente essas chamadas estão disponíveis como rotinas escritas em C e C++, embora certas tarefas de baixo nível (por exemplo, tarefas em que o *hardware* deve ser acessado diretamente) possam ter de ser escritas com o uso de instruções em linguagem de montagem.

As chamadas do sistema podem ser basicamente agrupadas em seis grandes categorias:

- Controle de processo: finalizar e abortar um programa, carga e execução de aplicações, criação e término de processos, obtenção e alteração de atributos de processos, espera por eventos ou sinais, alocação e liberação memória.
- Manipulação de arquivo: criar e apagar, abrir e fechar, ler, escrever e reposicionar arquivos, obter e atribuir atributos à arquivos.
- Manipulação de dispositivo: requisitar e liberar um dispositivo, ler, escrever e reposicionar, dar e receber atributos de dispositivos.
- Manutenção de informação: obter ou atribuir a hora ou a data, obter e pôr dados no sistema.
- Comunicações: criar e apagar conexões de comunicação, enviar e receber mensagens, transferir informações e montar e desmontar dispositivos.
- Proteção: mudar políticas de segurança de arquivos, diretórios, usuários, grupos etc.

A seguir, alguns exemplos de chamadas de sistema comumente presentes no Linux.
Controle de processo:

- *fork()*: Cria um processo filho.
- *exit()*: Fecha um processo em execução.

Manipulação de arquivos:

- *open()*: Abre um arquivo
- *read()*: Lê bytes de um arquivo.
- *write()*: Escreve bytes em um arquivo.

Manipulação de dispositivos:

- *ioctl()*: Chamada ao sistema especifica para dispositivos de i/o.
- *read()*:Lê bytes de um arquivo (o sistema entende qualquer dispositivo E/S como um arquivo).
- *write()*: Escreve bytes em um arquivo (o sistema entende qualquer dispositivo de E/S como um arquivo).

Manutenção de informações:

- *getpid()*: Retorna o PID do processo de chamada
- *alarm()*: Ajusta um alarme para a entrega de algum sinal.

Comunicações:

- *pipe()*: Cria um pipe, um canal de dados para comunicação entre processos.
- *mmap2()*: mapeia arquivos ou dispositivos na memória.

Proteção:

- *chmod()*: Muda as permissões de um arquivo.
- *umask()*: Determina uma máscara de criação de modo de arquivo.

2.3.2 *SystemTap*

De acordo com (Domingo 2013), a *SystemTap* é um sistema de rastreamento que permite um rico estudo sobre o sistema operacional e em particular o *kernel*. Possui sua saída semelhante à *strace*, ferramenta do Linux que permite rastrear as chamadas ao sistema e seus respectivos argumentos de programas/processos do sistema operacional. Mas a *SystemTap* possui ainda opções de filtragem e análise das informações coletadas.

Diferente da *strace*, a *SystemTap* permite que esse monitoramento seja feito através de simples *scripts*, permitindo que chamadas de sistema e outros eventos que ocorrem no *kernel* sejam rastreados e ainda, sem a necessidade de modificações manuais do *kernel* alvo e/ou reinicializações do sistema. Dessa forma, a *SystemTap* deixa de ser uma ferramenta e se mostra um sistema que dá a possibilidade do usuário desenvolver suas ferramentas de monitoramento.

A Figura 2.1 a seguir, ilustra o processo *SystemTap*.

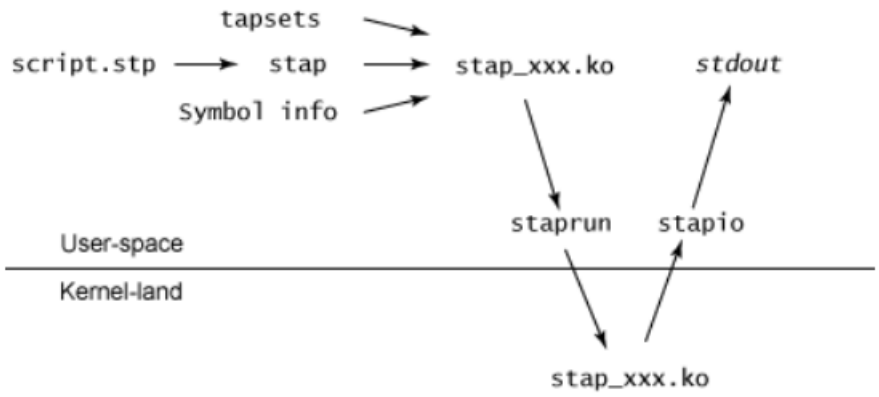

Figura 2.1: Processo *SystemTap* do ponto de vista de *kernel* e espaço de usuário

Fonte: (Jones 2009)

Ao ser iniciado a *SystemTap*, o *script* (*script.stp*) é compilado para um módulo de *kernel* (*stap_xxx.ko*). O utilitário *staprun* executa esse módulo no espaço de *kernel* e o utilitário *stapio* encaminha as informações para a saída padrão (*stdout*). Segundo (IBM 2009), *Tapsets* é um conjunto de funções úteis e comuns e *probes* pré-definidos que são correntemente utilizados. *Stap* é a implementação da *SystemTap*, que recebe um arquivo de entrada (*script.stp*) no qual estão definidos os *probes*. *Symbol info*, é uma tabela de símbolos usada para traduzir o *script* para linguagem C.

De acordo com (Leitão 2010), a ideia original do *SystemTap* é baseada nos conceitos de eventos e *handlers* (manipuladores). Um evento ou um *breakpoint*, permite que um usuário possa definir que um evento seja gerado no sistema. Sabendo disso, no momento que o usuário definir um evento, ao passar por esse evento, o *kernel* executará um *handler* (manipulador). Esses eventos são os chamados pontos de *probe* (*probe points*), e existem vários tipos de eventos que vão desde o básico como a execução ou o retorno de uma função, ou o fim de um *timer*.

Na *SystemTap*, esses *handlers* são escritos em uma linguagem própria com uma sintaxe muito próxima a da linguagem C, sendo capaz de permitir uma boa paridade com o que a linguagem C permite. Uma ressalva deve ser feita nesse ponto, pois embora tenha muitas funções derivadas da linguagem C e até permita que códigos em C sejam embarcados no *script*, a *SystemTap* impõe algumas restrições de alocação de memória e tamanho de pilha de execução.

Um ponto crucial para a *SystemTap* são os recursos de *kernel Kprobes*. *Kprobes* é uma API do *kernel* que permite que uma determinada posição do código do *kernel* (*probe points*), seja alterado para que o *kernel* chame uma outra função e volte depois à sua

execução normal.

Exemplo de *Kprobe*: `probe syscall.open { print(argstr)}`. Este *probe* de *syscall* imprime na saída padrão cada vez que a chamada ao sistema *open* chamada, os seus respectivos argumentos.

Capítulo 3

Sistema de Detecção com Monitoramento de Processos

Este capítulo descreve o sistema que foi desenvolvido e como foi desenvolvido. Explica o funcionamento de cada módulo do sistema, da política de segurança, mostra alguns detalhes da implementação e a maneira utilizada para que as partes do sistema se comuniquem.

Este trabalho foi a tentativa de melhorar um projeto anterior que possuía o mesmo propósito: construir um IDS usando a *SystemTap*. Este projeto anterior, porém, não teve bom desempenho e principalmente, teve o seu desenvolvimento muito complicado devido a abordagem escolhida, que foi inserir códigos C dentro da própria *SystemTap*. Embora a *SystemTap* permita isso, ela impõe algumas restrições como por exemplo, de alocação de memória.

Neste projeto a principal mudança foi a modularização do sistema, como é possível ver a seguir:

A Figura 3.1 ilustra o funcionamento do sistema de detecção com o monitoramento dos processos via *SystemTap* desenvolvido neste projeto. A entrada de todo o processo é o arquivo de políticas de segurança *policies.conf*. Um *script* de *shell* chamado *geraStap.sh* lê esse arquivo de políticas e gera o *script SystemTap* chamado *fonte.stp*. Esse *script SystemTap* é utilizado na coleta de chamadas ao sistema. As chamadas ao sistema capturadas são repassadas ao módulo de monitoramento que, por sua vez, emite alertas quando processos monitorados violam as políticas definidas no arquivo *policies.conf*.

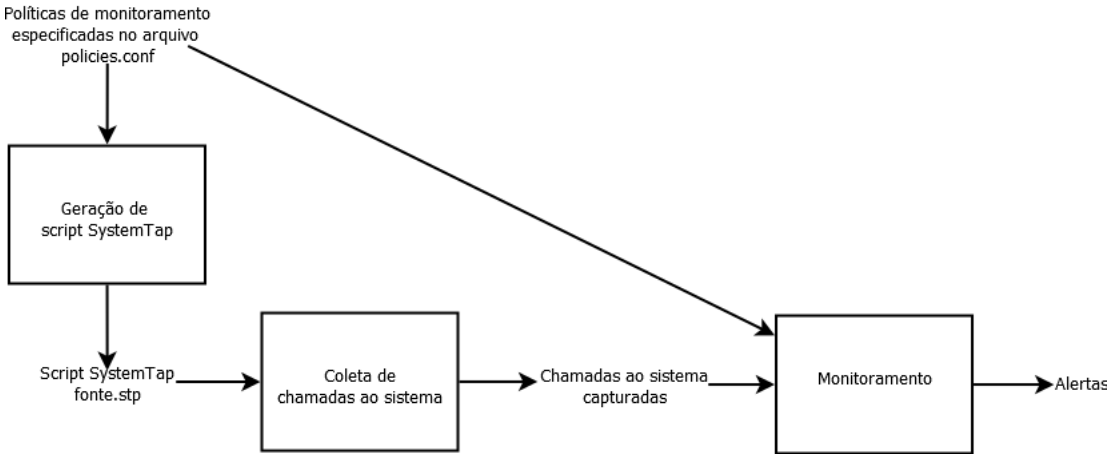

Figura 3.1: Descrição do sistema de detecção.

3.1 Mecanismo para especificação de políticas de monitoramento

A política de segurança é baseada na política de segurança utilizada na tese de doutorado defendida pelo orientador desse projeto, como definido a seguir. (Paula 2004).

O protótipo para detecção de anomalias em processos usa um arquivo de configuração, *policies.conf*, que modela o comportamento esperado desses processos. Cada política de acesso especifica o comportamento que determinados processos podem exibir durante a execução. O comportamento é verificado observando os seguintes eventos:

- Arquivos e diretórios: abertura, criação e remoção.
- Processos: criação e execução;
- Comunicação: estabelecimento e aceitação de conexões TCP, e envio e recebimento de tráfego UDP.

A seguir é apresentada a estrutura de um arquivo de políticas de monitoramento de processos de acordo com (Paula 2004).

```
# Comentários são feitos dessa forma
Politica1[/home/felipe/prog1]
{
    fs_access # acesso ao sistema de arquivo
    {
```

```
        / r,
        /home/felipe w
    }
    can_exec # quais programas ele pode executar  {  }
    max_children = 5 # número máximo de processos filhos
    can_send_signal = no
    connect_using_tcp = no
    send_using_udp = yes
    # portas que podem ser usadas para aceitar conexões
    # TCP ou aceitar tráfego UDP
    accept_conn_on_ports {7}
}
```

Nesse exemplo foi definida uma política de nome "Politica1" para monitorar processos que executam o programa /home/felipe/prog1.

De acordo com a seção *fs_access* dessa política, um processo só pode escrever em arquivos/diretórios localizados a partir de */home/felipe*. Para o restante do sistema de arquivos só é permitida a leitura. A política, através da seção *can_exec*, não permite executar outros programas a partir desse através de chamadas ao sistema do tipo *exec()*. O número máximo de processos/threads criados são, no máximo, 5 (seção *max_children*). O processo não pode enviar sinais a outros processos (seção *can_send_signal*). Não é permitido iniciar conexões TCP mas é permitido enviar dados via UDP (seções *connect_using_tcp* e *send_using_udp*). É permitido aceitar transmissão de dados via UDP através da porta 7.

3.2 Módulo de captura

Para a captura dos dados dos programas monitorados foi utilizada a ferramenta *SystemTap*. Um único *script* é criado para capturar os dados de todos os programas definidos na política de segurança.

O *script* possui os seguintes *probes* para executar o monitoramento dos programas:

- *Probes process point*: cada programa da política possui um *probe* de processo. Sendo assim, se existem dez programas sendo monitorados, existem dez *probes* de processo. Esse tipo de *probe* utiliza um *path* (caminho) para algum determinado programa executável. Ele permite que no momento do disparo[1] possa ser capturada informações importantes da *syscall* (chamada ao sistema) que o programa está executando na-

[1]termo usado pela *SystemTap* para o momento da execução de algum evento que esteja sendo monitorado por ela, ou seja a execução do programa em si

quele momento, como por exemplo: qual a *syscall* está sendo executada; argumentos da *syscall*; *pid* (identificador do processo) do programa que está executando.

- Um *Probe syscall*: esse *probe* é disparado em qualquer chamada ao sistema que é feita no Linux. Dessa forma, esse *probe* estará sempre sendo disparado. Esse *probe* também possui informações de *pid* e argumentos da *syscall*. A diferença está na maneira como esse *probe* entrega os argumentos da *syscall*. Enquanto o *probe* de processo entrega os argumentos separados em seis váriaveis, o *probe* de *syscall* disponibiliza as mesmas informações em uma única variável do tipo *string* separados por vírgulas, o que facilita o processamento e o envio dessas informações para o monitor.

Com esses dois tipos de *probe*, no momento que um processo que está sendo monitorado executa alguma chamada ao sistema, o *pid* e o número da *syscall* que está sendo executada são capturados através do *probe process* e os argumentos da chamada ao sistema é obtido pelo *probe* de processo.

A *SystemTap* identifica a *syscall* através de números não negativos. Com esses dados, o número da *syscall* que está sendo executada e o *pid* do processo é então utilizado um vetor associativo, estrutura de dados disponibilizada pela ferramenta de forma nativa na *SystemTap*. Com esse vetor, no *probe* de processo de cada programa, é associado o valor do *pid* do programa com o valor da *syscall* que está sendo executada. Isso serve para que no *probe* de *syscall*, que é disparado com qualquer chamada ao sistema, possa ser identificado qual chamada ao sistema é de um programa/processo monitorado ou não. Dessa forma, através do *pid* o *probe* de *syscall* ignora os processos que não estão sendo de fato monitorados.

Após ser identificado que um processo é um programa a ser monitorado, os seguintes dados são reunidos em uma função na *SystemTap* que é responsável por empacotar os dados para enviar para o monitor. Os dados que são enviados ao monitor são: o *pid* do programa, o número da chamada ao sistema que foi executada, o caminho do executável do programa e os argumentos que a chamada ao sistema possui. O caminho do executável e os argumentos são enviados em formato de *string*. Dessa forma também são enviados os tamanhos respectivos dessas *strings*.

Os tamanhos das variáveis, o *pid* e o número da chamada ao sistema são valores inteiros, mas são transformados em caractere. O *pid* é enviado como uma sequência de dois caracteres, já o número da chamada e os tamanhos das variáveis *string* são enviados cada um como um caractere apenas.

Dessa forma, o *script* possui uma função que envia os dados para o monitor, um *probe* de processo para cada programa que está descrito na política e um *probe* de *syscall*. Assim, os *probes* de processo possuem o mesmo código, mudando apenas o nome do processo que

será monitorado, permitindo que o processo de criação do *script SystemtTap* possa ser automatizado pelo *script* de *shell* (*script* que usa comandos de *bash* do Linux).

Para gerar o *script SystemTap* é utilizado o seguinte comando: `./geraScript.sh policies.conf`, onde *geraScript.sh* é o *script* de *shell* que gera o *script SystemTap* e *policies.conf* é o arquivo com as políticas de segurança.

O módulo *kernel fonte.ko* é gerado através do comando:
`stap -g -v -p 4 -D MAXMAPENTRIES=1048576 -suppress-time-limits fonte.stp -m fonte`

Onde:

- *fonte.stp* é o *script SystemTap*;
- -g é o *guru mode* e indica que há código em C embarcado no *script*;
- -v é o *verbose mode* que imprime os passos, que vão de um a cinco, executados pela *SystemTap*;
- -p para parar em um dos cinco passos. É escolhido parar no passo quatro devido ao último passo ser a execução. A execução do módulo é feita posteriormente;
- -D significa que serão modificados parâmetros de limite que são pré-definidos pela *SystemTap*. Nesse caso o parâmetro *MAXMAPENTRIES* é o tamanho máximo que um vetor pode ter;
- *–suppress-time-limits* indica que deve ser removido os limites de tempo que a *SystemTap* impõe;
- -m indica qual o nome o módulo de *kernel* terá. No caso é escolhido *fonte*.

3.3 Envio dos dados para o monitor

Após os dados serem empacotados pela *SystemTap*, eles são impressos na saída padrão do terminal de execução e um *pipe* faz a conexão entre o *script SystemTap* e o monitor. Um *pipe* é definido por (Silberschatz 2010) como um canal que permite que dois processos se comuniquem. Nesse trabalho foi o utilizado o *pipe* do *bash* do Linux, que utiliza o caractere '|'. Dessa forma o monitor lê os dados que são enviados para a saída padrão pela módulo *fonte.ko*.

O comando utilizado no terminal para a execução do módulo de *kernel* enviar os dados para o monitor foi: `staprun fonte.ko | ./monitor` ou seja, o processo *staprun* tem a sua saída de dados padrão redirecionada para a entrada de dados padrão do processo *monitor*.

3.4 Módulo de monitoramento dos dados recebidos

O monitor, além de receber os dados da *SystemTap* e monitorar os programas, possui um importante pré-processamento que é obter os dados da política de segurança. Um *parser* simples foi desenvolvido para que possa pegar os dados que estão no arquivo de política. Então, cada programa da política possui seus dados, que foram descritos anteriormente, salvos em uma estrutura do tipo *dado*. Ao final da execução do *parser*, temos então um vetor dessa estrutura com as informações que especificam o comportamento esperado para cada programa.

A definição da estrutura *dado* é a seguinte:

```
struct dado
{
    char programa[250];
    struct Trie* f_paths;
    can_exec e_paths[512];
    int max_child;
    int can_send_signal:2;
    int connect_using_tcp:2;
    int send_using_udp:2;
    int port_list[512];
    int quantE_paths;
    int quantPortList;
};
```

A estrutura *dado* utiliza ainda uma segunda estrutura do tipo can_exec, definida a seguir:

```
typedef struct pode_executar
{
    char path[512];
}can_exec;
```

Os campos dessa estrutura são descrito na Seção 3.1. O campo *f_paths*, do tipo ponteiro para *Trie*, é uma árvore de prefixo que será descrita a seguir.

Um dado que possui a busca recorrente durante a execução do monitoramento são os caminhos descritos na seção *fs_access* e por esse motivo dentro da *struct* que une os dados da política, a estrutura de dado utilizada para armazenar os diretórios descritos em *fs_access* foi uma árvore de prefixo, também referenciada por *trie*.

Uma *trie*, de acordo com (Szwarcfiter 2010) é definida como uma árvore m-ária, não vazia, que segue as seguintes premissas:

1. Se um nó v é o j-ésimo filho de seu pai, então v corresponde ao dígito d_j do alfabeto S, sendo $1 \leq j \leq m$.

2. Para cada nó v, a sequência de dígitos definida pelo caminho desde a raiz de T até v correspondente a um prefixo de alguma chave de S.

Com essa estrutura, é possível efetuar buscas por um diretório de uma maneira mais rápida, tendo complexidade *O(k)*, com k igual ao tamanho da *string* que está sendo buscada.

A Figura 3.2 ilustra a *trie* para um alfabeto R de tamanho 26.

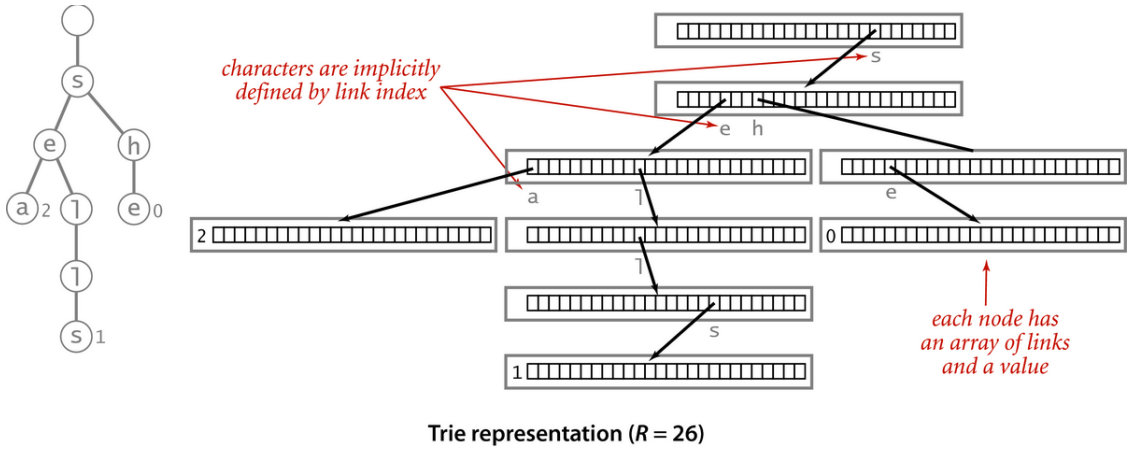

Figura 3.2: Exemplo de uma *trie* para um alfabeto R de 26 caracteres. Fonte: (Feofiloff 2018).

Capítulo 4

Resultados

4.1 Ambiente de teste

O ambiente de teste foi preparado em computador com sistema Linux Fedora 25 instalado com o *kernel* na versão 4.8.6.300.x86-64 e a ferramenta *SystemTap* instalada na versão 3.2.

O computador utilizado possui as seguintes especificações: processador Intel Core i5 7200U, 8GB de memória RAM DDR4, SSD de 240GB SATA 3.

Os testes se resumem em capturar o tempo de execução dos programas com e sem o funcionamento do sistema desenvolvido a fim de comparar os tempos e avaliar o impacto gerado. O tempo foi obtido usando o comando *time* do *bash*, que mede o tempo desde a execução até o término do processo.

Durante os testes as saídas geradas pelo programa *monitor* e pelos programas testados, foram redirecionadas para o diretório */dev/null* para que o tempo que é gasto fazendo a impressão no *bash* não afete os testes.

4.2 Casos de teste

Para os testes foram utilizados 4 programas: o *ls*, que lista diretórios e arquivos do sistema; o *cat*, que imprime o conteúdo de arquivos na saída padrão; o *progfork*, um programa que cria processos filhos através da chamada *fork()* que foi desenvolvido para os testes; e um programa *clientServ* que cria *sockets* e executa conexões em um servidor local que foi desenvolvido para o propósito de testes apenas.

A seguir o código fonte dos programas *progFork* e *clientServ* respectivamente:

```
int main(void)
{
```

```
 unsigned int i;
 for(i=0;i<10000;i++)
     if ( fork() > 0 )
         continue;
     else
         break;
}
```

Função principal do programa *clientServ*.

```
#define TAM 10000

int clienteSockfd[TAM];

int setup()
{
    int i;
    struct sockaddr_in serv_addr;
    socklen_t addrlen = sizeof (serv_addr);

    for (i = 0; i < TAM; ++i)
    {
        clienteSockfd[i] = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
        if (clienteSockfd[i] < 0)
        {
            printf("Erro no Socket: %s %d\n",strerror(errno),clienteSockfd[i]);
            exit(1);
        }
    }
    bzero((char *) & serv_addr, sizeof (serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
    serv_addr.sin_port = htons(9999);

    for (i = 0; i < TAM; ++i)
    {
       if(connect(clienteSockfd[i],(struct sockaddr *)&serv_addr, addrlen) < 0)
        {
```

```
            printf("Erro no Socket\n");
            exit(1);
        }
    }
}
```

O programa *ls* foi levado mais além, sendo testado de duas maneiras: fazendo a listagem comum e uma listagem recursiva de todos os subdiretórios a partir do diretório raiz do Linux (/).

Nos programas *ls* (sem a listagem recursiva) e *cat*, para que houvesse um tempo significativo para comparação, foi aferido o tempo de execução de dez mil execuções consecutivas do mesmo comando. Essa repetição foi feita usando um *script* de *shell*.

O programa *progfork*, apenas cria dez mil processos filhos e encerra.

O programa *clienteServ* foi desenvolvido de forma que seja criado e executado dez mil *sockets* e conexões respectivamente.

Cada teste executado teve seu o tempo de execução medido dez vezes.

A seguir, são apresentados as políticas de segurança que foram usadas nos testes dos programas anteriormente descritos.

Política do programa *ls*:

```
politicals[/usr/bin/ls]
{
    fs_access
    {
        / n,
        /usr/bin rw,
        /home r,
        /home/felipe w,
        /dev/null w,
    }
    can_exec    { }
    max_child = 0
    can_send_signal = yes
    connect_using_tcp = no
    send_using_udp = no
    port_list{ }
}
```

Essa política descreve que o programa *ls* com o caminho da instalação em */usr/bin/ls*, pela seção *fs_access* é permitido executar leitura e escrita a partir do diretório */usr/bin*,

e apenas escrita a partir dos diretórios */home/felipe* e */dev/null* e apenas leitura a partir dos diretórios */home*. Para os demais diretórios a partir da raiz / não pode ser executada nenhuma operação.

Pela seção *can_exec*, o programa não possui permissão para executar outros programas, pela seção *max_child*, não é permitido criar filhos. O programa possui permissão para enviar sinais pela secão *can_send_signal* e não possui permissão para fazer conexões via TCP ou enviar datagramas via UDP, pelas seções *onnect_using_tcp send_using_udp*, respectivamente. Nenhuma porta foi especificada na seção *port_list*.

Política do programa *cat*:

```
politicacat[/usr/bin/cat]
{
    fs_access
    {
        / n,
        /home r,
        /dev/null w,
    }
    can_exec    { }
    max_child = 0
    can_send_signal = no
    connect_using_tcp = no
    send_using_udp = no
    port_list{ }
}
```

A política do programa *cat*, que está em */usr/bin/cat*, pela seção *fs_access* é permitido executar apenas leitura a partir do diretório */home* e apenas escrita a partir do diretório */dev/null* e para os demais diretórios a partir da raiz / não pode ser executada nenhuma operação.

Pela seção *can_exec*, o programa não possui permissão para executar outros programas e através da seção *max_child*, não é permitido criar filhos. O programa não possui permissão para enviar sinais, fazer conexões via TCP ou enviar datagramas via UDP, pelas seções *can_send_signal*, *connect_using_tcp* e *send_using_udp*, respectivamente. Nenhuma porta foi especificada na seção *port_list*.

Política do programa *progfork*:

```
politicaprogfork[/usr/bin/progfork]
{
    fs_access
    {
```

```
        / n,
        /home r,
        /dev/null w,
    }
    can_exec    { }
    max_child = 0
    can_send_signal = no
    connect_using_tcp = no
    send_using_udp = no
    port_list{ }
}
```

A política do programa *progfork*, com caminho de instalação */usr/bin/progfork*, pela seção *fs_access* possui permissão para executar apenas leitura a partir do diretório */home* e apenas escrita a partir do diretório */dev/null*. Para os demais diretórios a partir da raiz (/) não pode ser executada nenhuma operação.

Pela seção *can_exec*, o programa não possui permissão para executar outros programas, pela seção *max_child*, não é permitido criar filhos. O programa não possui permissão para enviar sinais, fazer conexões via TCP ou enviar datagramas via UDP, pelas seções *can_send_signal*, *connect_using_tcp* e *send_using_udp*, respectivamente. Nenhuma porta foi especificada na seção *port_list*.

Política do programa *clienteServ*:

```
politicaclienteServ[/usr/bin/clienteServ]
{
    fs_access
    {
        / n,
        /home r,
        /dev/null w,
    }
    can_exec    { }
    max_child = 0
    can_send_signal = no
    connect_using_tcp = yes
    send_using_udp = no
    port_list{ 1, 53, 80}
}
```

A política do programa *clienteServ*, com caminho de instalação */usr/bin/clienteServ*, pela seção *fs_access* possui permissão para executar apenas leitura a partir do diretório

/home e apenas escrita a partir do diretório */dev/null*. Para os demais diretórios a partir da raiz (/) não pode ser executada nenhuma operação.

Pela seção *can_exec*, o programa não possui permissão para executar outros programas, pela seção *max_child*, não é permitido criar filhos. O programa não possui permissão para enviar sinais ou enviar datagramas via UDP, pelas seções *can_send_signal*, e *send_using_udp*, respectivamente. É permitido fazer conexões via TCP pela seção *connect_using_tcp*. As portas 1,53 e 80 foram definidas como permitidas pela seção *port_list*.

Para o projeto anterior, as políticas utilizadas tinham a mesma configuração. A diferença foram os programas que foram utilizados. No projeto anterior, para os testes de criação de processos filhos e criação de *sockets*, foi utilizado o mesmo programa *progCliente*. Por conta disso, a Tabela 4.2 possui *progCliente1* e *progCliente2* que são respectivamente equivalentes ao *clienteServ* e *progfork*, presentes na Tabela 4.1.

4.3 Resultados obtidos

Como esse trabalho foi uma continuidade de um projeto anterior que possuía uma abordagem que se mostrou ineficiente principalmente no quesito de desenvolvimento do sistema, para efeito de comparação, os resultados obtidos anteriormente serão apresentados e comparados com o sistema desenvolvido que foi apresentado no Capítulo 3.

Os tempos apresentados nas tabelas são médias aritméticas.

A Tabela 4.1 apresenta os tempos obtidos com o sistema de detecção desenvolvido nesse projeto.

	Sem monitoramento	Com monitoramento	Impacto em %
ls	7,8003	13,0534	67,34 %
ls recursivo	5,8766	23,197	294,74 %
cat	5,4187	8,3603	54,29 %
clientServ	12,4611	20,9063	67,77 %
progfork	0,2388	0,3247	35,97 %

Tabela 4.1: Tempos obtidos com o sistema de detecção

A Tabela 4.2 apresenta os dados que foram obtidos nos testes com o projeto anterior

	Sem monitoramento	Com monitoramento	Impacto em %
ls	7,29	8,218	12,73 %
ls recursivo	1,637	3,863	135,98 %
progCliente1	0,080	0,0957	19,63 %
progCliente2	0,085	0,0130	52,94 %

Tabela 4.2: Tempos obtidos com o projeto anterior

4.4 Análise dos resultados

Pela Tabela 4.1, é possível ver que para o programa *ls* sem o parâmetro de recursividade, o impacto que o sistema de detecção gerou no tempo de execução foi em média, 67,34 % maior. O projeto anterior, pela Tabela 4.2, teve um impacto de 12,73 %.

Pela Tabela 4.1, o *ls* com o parâmetro de recursividade, teve um impacto expressivo de 294,74 % em quanto nos testes do projeto anterior, o impacto foi de 135,98 %.

O *cat* não foi testado no projeto anterior, mas nos testes feitos, com sistema de detecção desenvolvido, houve um impacto no tempo de execução médio de 54,29 %.

Pela Tabela 4.1, o programa *clienteServ* teve um tempo médio de execução, 67,77 % maior com o sistema de detecção ativo, se comparado com o teste feito sem o sistema de detecção. No projeto antigo, esse impacto foi de 19,63 % para o programa *progCliente1*.

Pela Tabela 4.1, o programa *progfork* teve um tempo médio de execução, 35,97 % maior com o sistema de detecção sendo executado. No projeto antigo, esse impacto foi de 52,94 % para o programa *progCliente2*.

Capítulo 5

Conclusão

O projeto anterior tinha como abordagem, ser todo desenvolvido com a *SystemTap*. Desde a captura dos dados e leitura do arquivo de políticas, até a detecção da anomalia em si. O principal problema encontrado no projeto anterior foi desenvolvê-lo. A ideia de embarcar códigos em linguagem C diretamente no *script SystemTap* aparentou ser boa mas na prática foi custosa. Já o atual trabalho modularizou o desenvolvimento, utilizando a *SystemTap* apenas para a captura das chamadas ao sistema e um processo de usuário para realizar o monitoramento propriamente dito.

Durante o estudo do sistema *SystemTap*, foi averiguado que a quantidade de informação capturada traz um enorme impacto no desempenho. O trabalho anterior capturava apenas as informações essenciais de cada chamada ao sistema utilizando *probes* de *syscall*. Já nesse trabalho, todas as informações disponíveis de cada *syscall* foram capturadas e enviadas para o módulo de monitoramento em função da necessidade da adoção de *probes* de *processos*.

Com o sistema dividido nos módulos descritos no Capítulo 3, o desenvolvimento foi ágil e rápido. Portanto, levando em consideração apenas a riqueza de informações obtidas e a agilidade do desenvolvimento, pode ser considerado que houve êxito no desenvolvimento do sistema, mas o ponto crucial para a avaliação do sistema seria o seu desempenho. Nesse ponto, a abordagem adotada neste trabalho mostrou-se bastante ineficiente, pesando também a necessidade de comunicação via *pipe* e monitoramento realizado em modo usuário.

Uma estratégia que poderia ser explorada em trabalhos futuros é a utilização de mecanismos de memória compartilhada para a comunicação entre os módulos do sistema de detecção em busca de um melhor desempenho.

Referências bibliográficas

[Bishop 2002]BISHOP, M. *Computer Security: Art and Science*. 1. ed. Boston, MA: Addison Wesley, 2002.

[Domingo 2013]DOMINGO, D. *SystemTap 3.0 Beginners Guide*. 3. ed. NC, EUA, 2013.

[Feofiloff 2018]FEOFILOFF, P. *Tries (árvores digitais)*. 2018. Disponível em: <https://www.ime.usp.br/ pf/estruturas-de-dados/aulas/tries.html>.

[IBM 2009]IBM. *Instrumenting the Linux Kernel for Analyzing Performance and Functional Problems*. [S.l.], 2009. Disponível em: <http://www.redbooks.ibm.com/redpapers/pdfs/redp4469.pdf>.

[Jones 2009]JONES, M. T. *Introspecção Linux e SystemTap*. 2009. Disponível em: <https://www.ibm.com/developerworks/br/linux/library/l-systemtap/index.html>.

[Lehtinen 2006]LEHTINEN, R. *Computer Security Basics, 2nd Edition*. 2. ed. Sebastopol, CA: O'Reilly, 2006.

[Leitão 2010]LEITãO, B. Tutorial depuração de kernel com systemtap. *Linux Magazine n. 67*, 2010.

[Paula 2004]PAULA, F. S. de. *Uma arquitetura de segurança computacional inspirada no sistema imunológico*. Tese (Doutorado), Campinas, SP, Brasil, 2004.

[Silberschatz 2010]SILBERSCHATZ, A. *Fundamentos de Sistemas Operacionais*. 8. ed. SP: LTC, 2010.

[Szwarcfiter 2010]SZWARCFITER, J. L. *Estruturas de Dados e Seus Algoritmos*. 3. ed. Rio de Janeiro, RJ, Brasil: LTC, 2010.