Representando comportamento de usuários para detecção de ameaças internas

Felipe Tiberio Maciel Barbosa[1], Matheus Allan de Oliveira[1]

[1]Instituto Metrópole Digital – Universidade Federal do Rio Grande do Norte (UFRN)

Natal – RN – Brasil

felipetiberio@yahoo.com.br, matheusrpd@ufrn.edu.br

Resumo. *Projeto desenvolvido durante a disciplina IMD0040 - Linguagem de Programação II, como última atividade avaliativa . O projeto consiste de um sistema orientado a objetos em Java para a representação de usuário e seus comportamentos em uma empresa a partir de arquivos de log. Pois, uma vez que se tem esses dados é possível identificar comportamentos suspeitos que podem acarretar ameaças a organização.*

1. Introdução.

O Projeto "Insider Threat" em linhas gerais tem como objetivo, utilizar os conceitos discutidos durante a disciplina IMD0040 da Universidade Federal do Rio Grande do Norte. Nele abordamos quase todos os conceitos discutidos em aula e mais alguns técnicas necessárias para resolução do problema e desenvolvimento da aplicação. Procuramos sempre buscar a melhor maneira de resolver cada uma das etapas do projeto, algumas vezes, por dificuldades técnicas, essas resoluções não necessariamente são as mais eficientes, mas sim as mais práticas. Os problemas encontrados durante a implementação serão discutidos na sua respectiva seção.

No estado atual do programa, é fornecido ao sistema quatro arquivos de logs com algumas informações de uma determinada corporação; LDAP.csv, device.csv, logon.csv e http.csv em que, respectivamente, corresponde a dados dos funcionários, periféricos utilizados, login e logoff em máquinas e sites visitados.

O sistema é carregado , inicialmente com os dados de todos os usuários , e para cada usuário é gerado uma estrutura de dados de tipo árvore que irá representar o perfil deste usuário. Um vez que, o administrador já possui todos os usuário cadastrados, é possível cadastrar todas as atividades dos funcionários em suas respectivas árvores, ou apenas em um pequeno intervalo de tempo em que se deseja fazer a verificação de atividades suspeitas.

Quando o intervalo de comportamento dos usuários já estiver carregado no sistema o administrador poderá gerar um histograma de cada usuário com a quantidade de atividades desenvolvidas por hora de cada funcionário. Uma vez com esses dados, é possível gerar um perfil médio de cada um papel da organização, ou seja, será criado para cada cargo um perfil histograma que quantifica o que é esperado de cada funcionário de um determinado papel.

Qualquer perfil de usuário que esteja com valores de distância muito distantes daquilo que é espera do seu função, perfil médio, será identificado como possível ameaça, para posterior verificação.

2. Abordagem de resolução do problema.

Para solucionarmos o problema geral do projeto, que é a criação da floresta inicial, tendo todas as árvores criadas com as devidas atividades preenchidas na árvore, sabiamos que teriamos que tratas os dados dos arquivos da melhor maneira possível. Com isso, temos na classe LogAnalyzer métodos que são responsáveis por essa tarefa, onde pegamos o arquivo e transformamos em uma list de array de String, onde cada array de String da list é uma linha e cada posição do array é um dado presente na linha do arquivo. Assim, facilitou a criação dos objetos User, Logon, Devices e Http, já que essa organização nos dados facilitou o manuseio das informações. A classe BildPerfil é a responsável pela criação desses objetos.

Tendo os objetos criados, a classe BildPerfil retorna eles em LinkedList que podemos agora trabalhar nesses objetos, criando as árvores com os usuários ou adicionando na árvore do usuário as atividades realizadas por ele. Para isso, utilizamos a KeepTrees para gerenciar essa inclusão através da ProfileTree, que como já foi dito anteriormente controla as árvores específicas. A KeepTrees é responsável por armazenar essas árvores, como também os NodeUser que representam os perfis médios. Através de métodos da KeepTrees já criamos os perfis médios automaticamente a cada vez que uma nova atividade é adicionado a uma árvore, para que ao final da inclusão das atividades já tenhamos tudo pronto para a análise do usuários.

Partindo do momento que temos essa floresta completa, partimos para analisar o comportamento do usuário. Para isso, temos a classe AnomalyAnalysis, ela através dos seus métodos calcula o valor da distância Euclidiana dos usuários com os perfis médios, isso utilizando os valores dos histogramas. Tendo esse valor calculado, agora já podemos verficar se o usuário é uma anomalia perante os usuários de mesmo papel, como também criar um ranking com esse valor para usuários de papel igual. Caso queira maiores detalhes desses métodos, no tópico “3.2. Algoritmos utilizados” possui mais informações e que critério utilizamos para considerar um usuário uma anomalia.

Para gerenciar todas essas funcionalidades, a classe Controller cumpre esse papel. Ela possui as classes BildPerfil, KeepTrees e AnomalyAnalysis como atributos, e assim é capaz de controlar tudo o que é necessário para que as funcionalidades ocorram de maneira coesa. Para finalizar temos a classe Debug com o método main com um menu para que o operador escolha a funcionalidade que deseja realizar na aplicação. Segue abaixo, o diagrama de classes UML demonstrando tudo o que foi descrito.

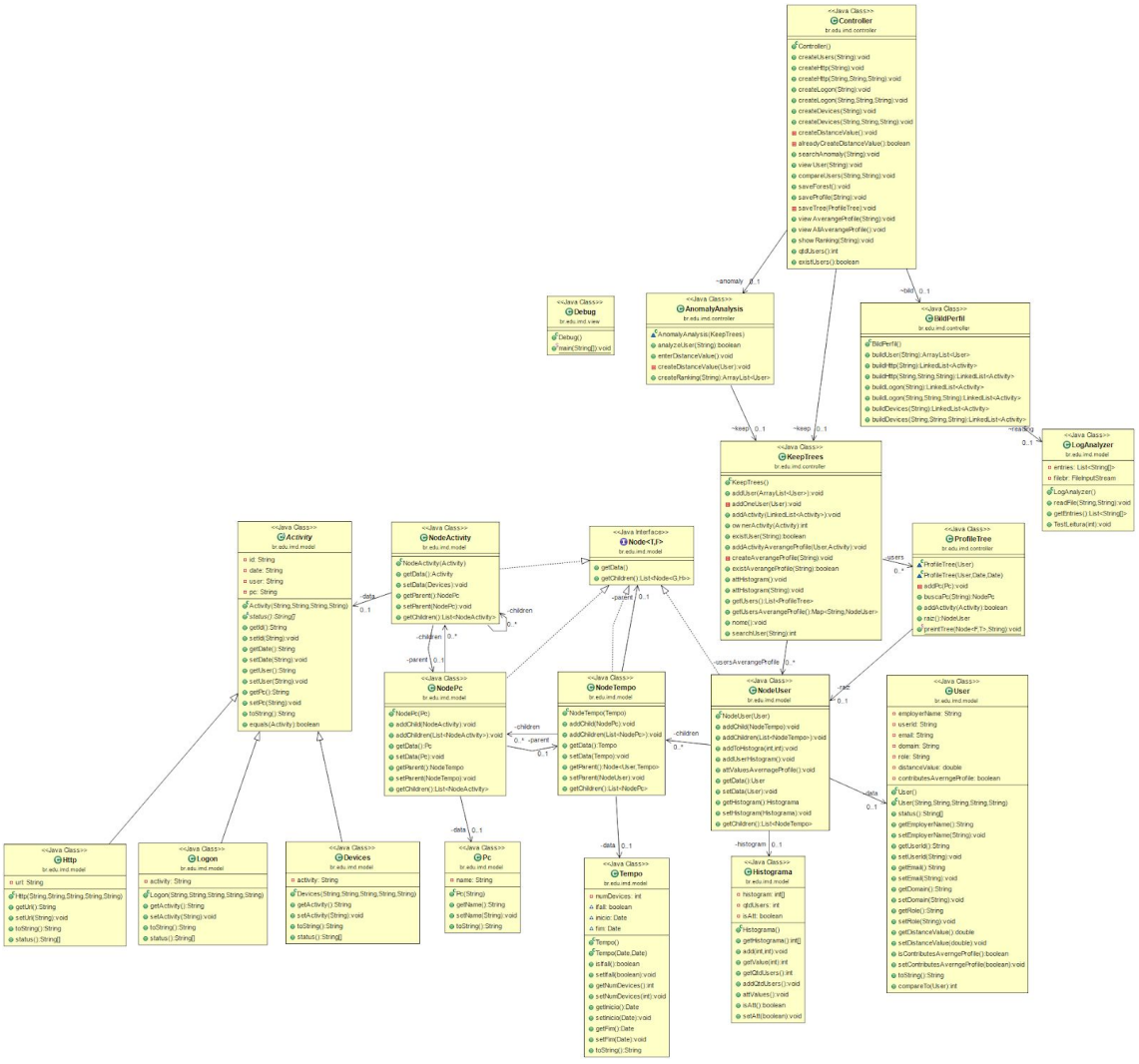

Imagem 1: Diagrama de classes UML do projeto

3. Algoritmos utilizados e estruturas de dados utilizadas.

3.1. Estrutura de dados utilizadas.

A principal estrutura de dados utilizada é Profile Tree, que é a árvore que representa um perfil de cada usuário, ele consiste em nós interligados de tipos diferentes, cada um desses nós implementa um interface Node.

A raiz dessa árvore é um NodeUser, que guarda as informações de um funcionário, bem como, seu histograma de atividade e uma quantidade n-ária de nodes filhos do tipo NodeTempo, no nosso caso a raiz nunca tem mais de dois filhos. Ela terá no máximo um filho que contém um intervalo de tempo específico ou todas as atividades de um usuário.

Para o próximo nível, abaixo dos NodeTempo, há uma quantidade n-árias de filhos do tipo NodePc. Para cada estação de trabalho utilizado pelo usuário será criado

um nó desse tipo, que corresponde aos computadores usados pelo funcionário que corresponde a raiz da árvore.

Por último, abaixo dos NodePc está as folhas da árvore em que cada uma dessas folhas é uma atividade feita pelo usuário da ProfileTree correspondente no NodePc, pai correspondente .

3.2. Algoritmos utilizados.

Alguns métodos são especiais no desenvolvimento da aplicação. Como por exemplo, os métodos de montagem da floresta, que consta com as árvores de cada usuário do sistema. Os métodos de verificação se um determinado usuário é uma anomalia diante os usuários de mesmo papel que o seu, além dos métodos responsáveis para o cálculo da distância Euclidiana que servirá para a verificação da situação dita anteriormente.

Para a montagem da floresta, temos os métodos da ProfileTree que pega as informações com os user para criar a raiz e com isso, utiliza-se do método addActivity para adicionar as novas atividades realizadas por esse usuário. Além disso, nesse mesmo método já atualizamos as informações no histograma do usuário.

Para o cálculo da distância Euclidiana temos o método específico na classe AnomalyAnalysis, chamado createDistanceValue(User user), onde ele recebe como parâmetro o usuário que deve ser verificado. Com isso, faz uma busca pelo histograma do usuário, depois busca o histograma do perfil médio do papel deste usuário. Com esse dados disponíveis, fazemos o cálculo seguindo a equação da distância Euclidiana. Onde consta na raiz quadrada de um somatório de cada valor de uma posição do histograma do usuário mesmo o valor da posição do histograma do perfil médio elevado ao quadrado. Após calcular, armazenamos essa informação no atributo distanceValue do usuário.

Com os valores do distanceValue calculados para todos os usuários, agora temos o método responsável por verificar se um determinado usuário é anomalia, utilizando os valores de distanceValue para a verificação. O método pertence também a classe AnomalyAnalysis, chamado de analyzeUser(String name), recebemos o nome do usuário como parâmetro para verificarmos. Primeiro, buscamos o usuário com esse nome, caso exista, verificamos se já existe perfil médio com o papel deste usuário. Existindo, calculamos a média de todos os valores distanceValue de todos os usuários desse papel, sendo o valor desse usuário maior do que a média, ele considerado uma anomalia.

4. Conclusão

Desde o início do Inside Threat até a sua data de apresentação os desenvolvedores trabalharam de forma comunicativa para entender o problema e implementar a aplicação. Durante esse processo alguns problemas ocorreram, inicialmente, o primeiro desafio encontrado foi como desenvolver uma árvore n-ário que armazenaria em seus nós tipos diferentes de dados a partir de cada um de seus

níveis. Optamos, por uma abordagem mais prática em que existe um super classe genérica, e para cada nível da árvore implementamos especializações dessa superclasse.

Um outro problema que ocorreu durante a implementação foi a grande quantidade de recursos de memória que o Inside Threat consome quando estamos analisando uma grande quantidade de dados, sabemos que isso não ocorreria caso existisse um banco de dados onde pudéssemos armazenar essas informações, assim apenas tratamos uma pequena quantidade de dados por vez. Por isso para uma melhor utilização da aplicação recomenda-se analisar intervalos pequenos de tempo nos arquivos de log.

No geral, o projeto foi intuitivo de desenvolvermos. Apresentando desafios e dificuldades que imaginamos encontrar em projetos no mercado de trabalho. Fazendo com que fossemos pesquisar maneiras de utilizar os conteúdos da disciplina de maneira mais avançada e pensar em soluções práticas para o problema. Infelizmente, diante o tempo, não conseguimos desenvolver uma interface gráfica completa para o projeto, por isso, decidimos deixar sem, utilizando o console da IDE com um menu mostrando e as funcionalidades de maneira dinâmica.