INSTITUTO POLITÉCNICO DE BEJA

Escola Superior de Tecnologia e Gestão

Mestrado em Engenharia de Segurança Informática

Sistema Distribuído de Deteção de Intrusões

David António Barrocas do Nascimento Palma

Beja

2014

INSTITUTO POLITÉCNICO DE BEJA

Escola Superior de Tecnologia e Gestão

Mestrado em Engenharia de Segurança Informática

Sistema Distribuído de Deteção de Intrusões

Relatório de projeto de dissertação de mestrado apresentado na Escola Superior de Tecnologia e Gestão do Instituto Politécnico de Beja

Elaborado por:

David António Barrocas do Nascimento Palma

Orientado por:

Doutor José Jasnau Caeiro

Beja

2014

Resumo

A segurança da informação é uma preocupação prioritária para as organizações. As intrusões informáticas causam consideráveis prejuízos financeiros. Este problema é particularmente relevante em redes sem fios distribuídas devido ao recente aumento do número de dispositivos móveis nas redes organizacionais. Os sistemas de deteção de intrusões (IDS) tradicionais não possuem características de escalabilidade adequadas a redes distribuídas em larga escala.

Nesta dissertação desenvolveu-se um IDS distribuído constituído por uma rede de sensores e por um servidor. O sistema foi desenvolvido na linguagem de programação Python e implementa *sockets* ØMQ que permitem a expansão dinâmica da rede. As comunicações são cifradas com algoritmos de criptografia de chaves públicas baseados em curvas elípticas (ECC). A deteção de intrusões é realizada pelo IDS Suricata, otimizado para sistemas de processamento paralelo. O utilizador controla o funcionamento do sistema em tempo real através de uma interface gráfica baseada no servidor web Tornado. Utiliza-se uma base de dados MongoDB para obter a persistência dos dados no servidor.

Foi construído um protótipo experimental do sistema com dispositivos físicos. Os testes de interação com o utilizador e com a ferramenta automatizada de teste Pytbull comprovam a viabilidade do sistema e incentivam a continuação do seu desenvolvimento no futuro.

Palavras-chave: Sistema Deteção Intrusões, IDS, Sistema Distribuído, Criptografia Curvas Elípticas, ECC

Abstract

Information security is a primary concern for organizations. Computer intrusions cause considerable financial losses. This problem is particularly relevant in wireless distributed networks due to the recent increase in the number of mobile devices in organizational networks. The tradicional intrusion detection systems (IDS) do not have the adequate scalability characteristics for large-scale distributed networks.

In this thesis a distributed IDS was developed consisting of a sensor network and a server. The system was developed in the Python programming language and implements ØMQ *sockets* that allow a dynamic network expansion. Communications are encrypted with public-key cryptography algorithms based on elliptic curves (ECC). The intrusion detection is performed by the Suricata IDS, optimized for parallel processing systems. The user controls the operation of the system in real time through a graphical user interface based on the Tornado web server. A MongoDB database is used for the storage of persistent data on the server.

An experimental prototype system was built with physical devices. Interaction tests with the user and the automated test tool Pytbull prove the feasibility of the system and encourage its continued development in the future.

Keywords: Intrusion Detection System, IDS, Distributed System, Elliptic Curve Cryptography, ECC

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

As organizações públicas e privadas mantêm registos relativos a assuntos internos e a colaboradores externos. Durante séculos o direito à privacidade da informação foi garantido por simples meios de segurança físicos. Com a crescente disponibilização de serviços através da Internet, as organizações enfrentam o desafio de permitir o acesso à informação de forma segura através de um meio inseguro.

A informação que é transmitida através de redes informáticas está sujeita a uma variedade de ameaças. Os atacantes são em cada vez maior número e são movidos pelas mais variadas razões, desde motivações financeiras, políticas ou o chamado "hactivismo". A segurança de sistemas de informação é um problema complexo. Trata-se de uma luta desigual. O atacante tem que explorar apenas um vetor de ataque, ao passo que o defensor tem de garantir a eliminação de todas as fraquezas.

O profissional de segurança da informação é altamente especializado nas ferramentas de segurança que utiliza. A escassez de profissionais com a profundidade de conhecimentos necessários e o grande custo dos sistemas de segurança, leva a que muitas organizações fiquem expostas a ataques.

Uma possível solução é o desenvolvimento de sistemas de segurança mais distribuídos e com interfaces mais acessíveis.

Este capítulo inicia-se com um enquadramento da segurança da informação para organizações e indivíduos. Introduzem-se os conceitos mais utilizados neste campo e apresenta-se uma perspetiva da dimensão da indústria. Introduz-se o conceito de intrusão em sistemas de informação e descrevem-se as principais ameaças existentes. No final descreve-se a estrutura da dissertação.

1.1 Enquadramento

Além do seu património físico, muitas organizações possuem valiosos espólios de informação. A garantia da segurança da informação relativa a clientes e colaboradores é uma das preocupações primordiais de qualquer organização com presença no mundo eletrónico. Historicamente, são as próprias organizações que definem o nível de segurança aplicável à informação que mantêm. A variabilidade dos critérios utilizados entre organizações resulta numa inconsistência que pode levar à diminuição do nível segurança da informação [6]. Isto leva a que as entidades governantes introduzam legislação que regulamenta estes critérios, estabelecendo um nível mínimo de segurança[1].

A **Segurança da Informação**, como definida pela literatura vigente [21], identifica cinco princípios fundamentais:

Confidencialidade: acesso restrito a entidades devidamente autorizadas.

Integridade: impossibilidade de modificação indetetável durante o ciclo de vida.

Disponibilidade acesso quando requisitado. Redundância e resistência a falhas.

Autenticidade: veracidade da mensagem e da identidade dos intervenientes.

Não repúdio: impossibilidade de negação do envio e da receção.

1.1.1 Segurança de Sistemas de Informação

Embora a informação tenha um carácter intangível, as propriedades anteriores não podem ser aplicadas num vácuo tecnológico. Na prática, são sistemas informáticos que fornecem suporte físico à informação e permitem a transação da mesma entre entidades comunicantes. Estes sistemas são complexos na sua conceção e manutenção e como qualquer elemento tecnológico estão sujeitos a falhas [21].

Vulnerabilidade: falha no sistema de informação.

Ameaça: entidade maliciosa interessada em explorar vulnerabilidades.

Ataque exploração da vulnerabilidade por parte da ameaça (atacante).

Risco: potencial prejuízo para a organização em caso de ataque.

[1] Em Portugal esta matéria está principalmente prevista na Lei 67/ 98 – Lei da proteção de Dados Pessoais.

1.1.2 Controlos de Segurança

A gestão da segurança da informação é um processo contínuo. A todo o momento surgem novas ameaças que requerem uma constante adaptação dos meios de defesa. Os responsáveis pela segurança da informação realizam ciclicamente as fases de **planeamento**, **execução** e **avaliação** dos controlos de segurança que implementam nas organizações [23].

Os controlos de segurança surgem como extensão lógica das políticas de segurança da organização. São regras práticas com caráter **preventivo**, **detetivo** ou **corretivo**.

Várias entidades mantêm listas atualizadas dos controlos de segurança mais comuns[2]. Estes critérios são desenvolvidos para serem independentes da tecnologia e incluem-se geralmente nas seguintes categorias genéricas:

Físicos: infraestrutura física da rede e equipamentos eletrónicos.

Técnicos: aplicações e sistemas lógicos de segurança.

Procedimentais: gestão da segurança e treino dos colaboradores. (atacante).

Legais: adequação das políticas de segurança à legislação vigente.

1.1.3 Tecnologias de Segurança da Informação

A implementação prática dos controlos de segurança deu origem a uma indústria mundial avaliada em €50 mil milhões em 2013, com uma perspetiva de crescimento para €63 mil milhões em 2016[3].

As soluções de segurança das empresas líderes de mercado apresentam tipicamente custos bastante elevados e requerem igualmente um grande nível de familiaridade com tecnologias proprietárias de cada fabricante. Organizações mais pequenas e cidadãos individuais encontram-se em desvantagem no acesso a certos recursos tecnológicos e tornam-se alvos fáceis para atacantes oportunistas.

Num momento em que se assiste à crescente democratização do acesso à Internet e onde todos os cidadãos possuem presença eletrónica, surge como um grande desafio o desenvolvimento de mecanismos de segurança baseados em tecnologias de livre acesso e tendencialmente gratuitas.

[2]ISO/IEC 27001:2013 e NIST SP 800-53 Rev. 4.
[3]Gartner Security & Risk Management Summit 2013.

1.2 Intrusões em Sistemas Informáticos

O conceito de **Intrusão em Sistema Informático** aplica-se a um ataque bem sucedido, após o qual, o atacante obtém acesso ao sistema alvo. A intrusão pode resultar da exploração de uma vulnerabilidade conhecida e documentada, mas para a qual o sistema alvo não possui os mecanismos de defesa adequados. Por outro lado, se a intrusão explorar uma vulnerabilidade ainda não conhecida designa-se por *zero-day* [7]. Este tipo de intrusões são particularmente perigosas, pois não existem ainda defesas contra as mesmas. Os atacantes têm ao seu dispor uma extensa variedade de métodos no processo de intrusão. As subsecções seguintes descrevem alguns dos métodos mais relevantes no contexto do presente documento.

1.2.1 *Malware*

A forma mais direta de ataque a sistemas e redes informáticas resulta da utilização de programas informáticos maliciosos, conhecidos como *malware*[4] . Existem vários tipos de *malware*, nomeadamente: vírus, *worms*, *trojans* e *spyware*.

Vírus: programa informático que tem a capacidade de se auto-replicar sem a permissão ou conhecimento do utilizador e atuar furtivamente sobre o sistema. Um vírus pode causar o funcionamento incorreto de programas ou corromper a memória do sistema. Geralmente a infeção ocorre ao copiar dados de outros sistemas infetados ou como resultado de uma ação inadvertida do utilizador.

***Worm*:** difere de um vírus no sentido em que se trata de *malware* que se propaga automaticamente pelas redes de computadores e procura ativamente vítimas. Um *worm* pode incluir um conjunto de instruções que são executadas imediatamente após a infeção, ou quando forem satisfeitos critérios específicos, por exemplo. datas predefinidas.

***Trojan*:** denominado "cavalo de Tróia"devido à forma como atua. Ao contrário de *virus* e *worms*, não possui capacidades de auto-replicação. Consiste num documento ou programa com uma função aparentemente benigna e inofensiva, mas que possuí código que cria uma ligação para o atacante. Após a infeção inicial, requer a ação do atacante que controla remotamente o sistema alvo.

[4] *Malicious software.*

***Spyware*:** a sua finalidade é "espiar"o sistema alvo, recolhendo informação sobre o próprio sistema e os seus utilizadores. A informação recolhida é enviada para o atacante. Tal como um *trojan*, é incapaz de se auto-replicar, no entanto, não permite o controlo do sistema alvo.

1.2.2 Ataques na *web*

Embora a utilização de *malware* seja uma forma eficaz de atacar sistemas em rede, requer geralmente ações por parte do utilizador que podem indiciar ao mesmo a iminência de um ataque. Com a vulgarização de acesso a serviços através da Internet surgiu uma classe de ataques que exploram vulnerabilidades nos protocolos baseados na *web*. Referem-se de seguida dois exemplos de ataques mais comuns: *SQL-injection* e *Cross-site scripting*.

***SQL-injection*:** explora vulnerabilidades de injeção de comandos. Consiste na inclusão de dados e comandos maliciosos em formulários de aplicações que efetuam pesquisas com base em informação proveniente do utilizador. Ocorre em sistemas que não efetuam a validação segura do conteúdo dos comandos antes de executar pesquisas nas bases de dados.

***Cross-site-scripting*:** explora vulnerabilidades existentes em aplicações que utilizam os protocolos *web*, tal como *web browsers*. Analogamente ao *SQL-Injection*, ocorre quando não existe uma validação e sanitização rigorosa de informação proveniente do utilizador e quando não são utilizadas formas de autenticação apropriadas.

1.2.3 *Botnets*

Da mesma forma que os sistemas informáticos se tornam mais distribuídos, também os métodos de ataque evoluíram no sentido de formar redes organizadas que permitem realizar ataques em massa. Uma *botnet* (*robot network*) é formada por inúmeros *bots* (*robots*) que são coordenados pelo sistema do atacante (*master*). Um sistema alvo torna-se um *bot* após ser infetado por *malware*. As *botnets* são utilizadas em ataques de *DDoS*[5], *spam*, *phishing* e roubo de identidade.

[5] *Distributed Denial of Service.*

1.3 Estrutura da Dissertação

Neste primeiro capítulo efetua-se um enquadramento da segurança da informação para organizações e indivíduos. Introduzem-se os conceitos mais utilizados neste campo e dá-se uma perspetiva da dimensão da indústria. Apresenta-se o conceito de intrusão em sistemas de informação e descrevem-se as principais ameaças existentes.

No capítulo 2 introduzem-se os conceitos fundamentais de um sistema de deteção de intrusões. É apresentada uma perspetiva histórica e referem-se alguns dos sistemas mais importantes. Enumeram-se as características fundamentais e descrevem-se os principais métodos de deteção e as suas tipologias.

No capítulo 3 introduz-se o problema da segurança em redes sem fios. Enumeram-se as principais ameaças neste âmbito, nomeadamente em dispositivos móveis. Descrevem-se os principais sistemas comerciais e apresenta-se a proposta de um novo sistema distribuído de deteção de intrusões em redes sem fios.

No capítulo 4 apresenta-se a arquitetura geral do sensor e enumeram-se os vários módulos e as respetivas funcionalidades. Descreve-se o processamento em modo de linha de comandos e os ficheiros de configuração. Introduz-se a comunicação com *sockets* ØMQ e *sockets* UNIX no contexto do IDS.

No capítulo 5 apresenta-se a arquitetura geral do sensor e enumeram-se os vários módulos e as respetivas funcionalidades. Introduz-se a base de dados MongoDB e o seu papel no funcionamento do servidor.

No capítulo 6 descreve-se a realização de um protótipo experimental do sistema desenvolvido. Apresenta-se a topologia da rede de teste e enumeram-se os vários componentes da mesma. No final realiza-se um conjunto de testes que confirmam o correto funcionamento do sistema desenvolvido.

A dissertação termina com o capitulo 7, que realiza uma reflexão conclusiva da dissertação e sugere várias hipóteses de trabalho futuro.

Capítulo 2

Deteção e Prevenção de Intrusões

A necessidade de mecanismos de deteção de intrusões em sistemas informáticos surge no momento em que estes passam a suportar simultaneamente processos de diferentes utilizadores. O crescimento em dimensão e complexidade de redes e sistemas informáticos torna inevitável a criação de um modelo de segurança da informação de indivíduos e organizações.

Na área da segurança informática, a implementação de um sistema de deteção de intrusões concretiza-se através de um dispositivo físico ou de um programa informático, cujo objetivo primordial consiste na monitorização de eventos associados a sistemas informáticos ou redes de computadores. A análise dos eventos observados pode revelar a existência de ameaças relacionadas com a exploração de vulnerabilidades tecnológicas dos sistemas monitorizados ou de violações das políticas de segurança de uma organização.

Ao longo das últimas décadas foram propostos sistemas de deteção de intrusões que variam na classificação e tipologia consoante a funcionalidade e âmbito da sua utilização. A área da investigação e desenvolvimento de sistemas de deteção de intrusões é hoje uma das mais ativas no campo das ciências informáticas e originou uma indústria com um volume de negócios considerável.

Neste capítulo introduzem-se os sistemas de deteção de intrusões. É apresentada uma perspetiva histórica e referem-se alguns dos sistemas mais relevantes. Enumeram-se as características fundamentais e descrevem-se os principais métodos de deteção e tipologias. Apresenta-se uma arquitetura típica e descrevem-se os elementos que a compõem. Finalmente introduzem-se os sistemas de utilização livre mais utilizados atualmente.

2.1 Estado da Arte

A doutrina da segurança informática tem origem na década de 60 com os primeiros sistemas informáticos multiutilizador. Os mecanismos de segurança existentes nestes sistemas são extremamente rudimentares, tendo como único objetivo tentar evitar que dados de um processo corrompam acidentalmente dados de outros processos [15]. Numa época em que as redes de computadores são inexistentes ou limitadas, surge uma nova preocupação fundamentada no risco que utilizadores legítimos dos sistemas contornem as medidas de segurança de sistemas informáticos e acedam a recursos para os quais não possuem a devida autorização [9].

Originalmente, o processo de controlo de segurança consiste simplesmente na monitorização, por parte do administrador do sistema, de uma consola central que apresenta em tempo real as atividades dos utilizadores [13].

Génese

O primeiro trabalho na área da segurança informática que introduz o conceito de monitorização de eventos de forma sistemática é publicado por James Anderson em 1972 [1] [2]. Este relatório técnico descreve um sistema de monitorização que deteta eventos relacionados com a segurança do sistema, regista a informação referente a estes eventos e gera relatórios da atividade observada.

Durante a década de 70 surgem várias iniciativas que implementam e expandem o conceito sugerido por Anderson. Devido ao facto dos sistemas propostos apenas efetuarem o registo dos eventos observados, toda a tarefa de análise continua a recair sobre o administrador do sistema, implicando normalmente a inspeção manual de pilhas de papel contínuo, um processo tedioso e de reduzida eficiência [13]. O grande desafio durante esta fase da evolução dos sistemas de segurança centra-se na decisão de quais os controlos de segurança a monitorizar. Torna-se imperativo definir um conjunto de critérios que permitam alcançar um equilíbrio entre a recolha de demasiada informação e de informação insuficiente para uma análise conclusiva em tempo útil [15].

Em 1980, Anderson apresenta um novo trabalho, no qual introduz a terminologia ainda hoje utilizada extensivamente na área de segurança de informação, definindo termos como: **Ameaça**; **Risco**; **Vulnerabilidade**; **Ataque**; **Penetração** [3]. Neste trabalho, Anderson classifica os vários tipos de ameaça que os utilizadores

representam consoante o nível de autorização que possuem para utilizar os sistemas (internos ou externos) e a autorização para aceder a recursos específicos desses sistemas (legítimos ou ilegítimos). Desta forma torna-se possível a criação de controlos de segurança com maior granularidade, reduzindo assim a quantidade de informação a analisar pelos responsáveis pela segurança dos sistemas.

O Primeiro IDS

O primeiro sistema de deteção de intrusões (IDS[1]) foi desenvolvido entre 1984 e 1986 por Dorothy Denning e Peter Neumann e designado por IDES[2]. Em 1987 Denning publica um artigo que descreve o modelo utilizado pelo IDES [5]. Este modelo constitui uma evolução dos trabalhos originais de Anderson e baseia-se na criação de perfis que representam estatisticamente a utilização normal dos recursos do sistema pelos utilizadores. Estes perfis são posteriormente comparados com os eventos observados pelo IDES. Para além das representações estatísticas, o IDES possui um outro mecanismo inovador baseado na utilização de assinaturas que correspondem a padrões predefinidos de ações ilegítimas bem conhecidas [14]. As características anteriores representam a génese dos sistemas de intrusão baseados em anomalias e baseados em assinaturas, respetivamente.

IDS em Rede

Durante a primeira metade da década de 1980, a propagação de vírus informáticos ocorre devido à partilha de disquetes infetadas entre sistemas, pelo que até então todos os IDS são baseados no próprio dispositivo. Em meados dos anos 80, com a evolução da ARPANET[3] surgem os primeiros ataques a computadores ligados em rede. A maioria das intrusões deve-se à negligência de fabricantes e administradores de equipamento informático, nomeadamente a pré-configuração de contas de utilizadores privilegiadas e a não alteração de palavras-passe predefinidas. Este novo tipo de ameaça dá origem ao conceito de IDS baseado na rede, particularmente o NSM[4] [10]. Este tipo de IDS captura e analisa o tráfego de rede, permitindo a monitorização de dispositivos com arquiteturas diferentes ligados na mesma rede local.

[1] *Intrusion Detection System.*
[2] *Intrusion Detection Expert System.*
[3] *Advanced Research Projects Agency Network.* - rede precursora da Internet atual
[4] *Network Security Monitor.*

IDS Distribuído

O primeiro ataque em larga escala através da Internet ocorre a 2 de Novembro de 1988, causado pelo *worm* Morris. Em resposta às graves consequências deste ataque, é criado o CERT[5] original em Carnegie Mellon. A expansão do âmbito geográfico dos ataques motivou uma nova evolução dos IDS. Em 1991 surge o primeiro IDS distribuído (DIDS) [22]. Este tipo de sistema é implementado através de um conjunto de sensores distribuídos pelos sistemas e redes a monitorizar. O DIDS é o primeiro IDS a disponibilizar uma consola de controlo e monitorização de eventos com a identificação de utilizadores e recursos abrangendo todas as redes e sistemas de uma organização.

2.2 Características Gerais

Atualmente existe uma grande variedade de IDS, entre sistemas académicos, governamentais e comerciais. Estes sistemas possuem características específicas que os diferenciam entre si. Identificam-se, no entanto, um conjunto de aplicações e funcionalidades comuns a todos os IDS. O documento SP800-94 do NIST mantém um registo atualizado de toda a informação sobre IDS. Este documento define igualmente a taxonomia padrão a utilizar na escrita de literatura técnica. Na área da segurança informática os sistemas que efetuam a deteção de intrusões categorizam-se de acordo com as suas capacidades de deteção e resposta a incidentes [20].

Sistema de Deteção de Intrusões (IDS): efetua a deteção de intrusões de forma passiva, ficando a resposta a possíveis incidentes totalmente a cargo do administrador.

Sistema de Deteção e Prevenção de Intrusões (IPS ou IDPS): possui todas as capacidades de um IDS, além de dispor de mecanismos adicionais que respondem automaticamente aos incidentes detetados.

A localização deste tipo de sistemas numa rede de computadores varia consoante a categoria a que pertencem (Fig. 2.1). Geralmente, um IDS é instalado de forma passiva, capturando as comunicações num determinado segmento de rede, sem interagir diretamente com o tráfego de rede. Um IPS deve ser instalado em linha antes do

[5] *Computer Emergency Response Team.*

segmento de rede que pretende monitorizar, de forma a poder bloquear ativamente a comunicação em caso de deteção de intrusão.

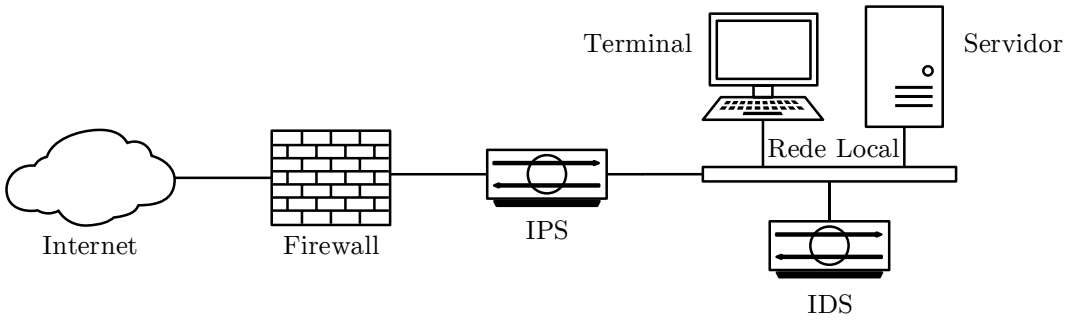

FIGURA 2.1: Diferenças na localização entre IDS e IPS.

2.2.1 Aplicações

A principal aplicação de um IDS consiste na identificação de incidentes originados pela exploração de vulnerabilidades ou violação de políticas de segurança. Todavia, um IDS é uma ferramenta de segurança versátil, utilizada por administradores de sistemas com diversas finalidades. Um IDS é igualmente utilizado com os seguintes objetivos:

Deteção de tráfego de rede suspeito através da utilização de regras semelhantes a uma *firewall*. É assim possível detetar irregularidades nos protocolos TCP/IP, bem como a transferência de ficheiros com conteúdo confidencial.

Deteção de atividades de reconhecimento tais como propagação de *worms* e varrimento de portas TCP e UDP que indicam a iminência de um possível ataque.

Comunicação de incidentes aos administradores responsáveis através de relatórios e ficheiros com registo de eventos.

Identificação de problemas em políticas de segurança em que o IDS é utilizado como salvaguarda em situação de falha de dispositivos de segurança de primeira linha, tais como *firewalls*.

Documentação de ameaças à organização através da caracterização dos ataques e da sua frequência. Útil no planeamento e implementação de medidas de segurança.

Dissuasão de atacantes devido à simples existência de mecanismos de segurança que desencorajam a exploração de possíveis vulnerabilidades.

2.2.2 Funcionalidades

De forma a realizar as aplicações descritas anteriormente, cada IDS deve suportar uma ou mais funcionalidades. A implementação das funcionalidades diverge consoante o objetivo do IDS seja apenas efetuar a deteção ou a prevenção de intrusões.

As funcionalidades que se descrevem de seguida são comuns a IDS e IPS:

Registo de informação relativa a eventos observados utilizando meios de armazenamento local ou de forma centralizada por um Sistema de Gestão de Eventos de Segurança de Informação (SIEM[6]).

Notificação de administradores de eventos observados através do envio de alertas com informação resumida dos eventos por vários meios de comunicação (*email*, SNMP, *syslog*).

Produção de relatórios nos quais é efetuada a apresentação de informação sobre eventos estruturada de acordo com vários critérios.

Adaptação de acordo com as ameaças detetadas que consiste na alteração no perfil de segurança ativo e na prioridade dos alertas correspondentes aos vários tipos de ameaças.

As seguintes funcionalidades aplicam-se apenas a sistemas com capacidades preventivas (IPS):

Bloqueio direto de ataques através da terminação da ligação de rede ou sessão do utilizador atacante, bloqueio do acesso ao alvo a partir de IP ou utilizador atacante e bloqueio de todo o acesso a um dispositivo, serviço ou aplicação.

[6] *Security Information and Event Management.*

Bloqueio indireto de ataques através da alteração da configuração de outro dispositivo de rede (*firewall*, *router*, *switch*), controlo de mecanismos de segurança no dispositivo alvo ou aplicação de atualizações de segurança no dispositivo alvo.

Alteração do conteúdo do ataque modificando partes das mensagens geradas pelo atacante, removendo ou substituindo conteúdo malicioso por conteúdo inofensivo.

2.2.3 Eficácia

Uma característica sempre presente num sistema de classificação de padrões, como é um IDS, é a impossibilidade de obter resultados absolutamente corretos. Uma percentagem dos resultados é considerada "falsa". Um falso positivo representa a identificação de atividade inofensiva como sendo maliciosa. Um falso negativo significa que o sistema falhou na deteção de atividade maliciosa. Apesar de ser impossível eliminar completamente falsos positivos e falsos negativos, o IDS deve ser configurado de forma a reduzir falsos resultados através de um processo de afinação. As taxas de erro caracterizam estes sistemas e os limiares de aceitação/rejeição devem ser escolhidos adequadamente. Geralmente é preferível reduzir o número de falsos negativos, aumentando os falsos positivos.

Um fator adicional que afeta adversamente a eficácia de IDS consiste na utilização, por parte dos atacantes, de técnicas evasivas que alteram o formato ou a temporização das suas mensagens para que estas aparentem ser inofensivas. Alguns IDS possuem métodos que detetam técnicas evasivas, realizando o mesmo processamento das mensagens que o sistema alvo.

2.3 Métodos de Deteção

A grande diversidade de ameaças implica a existência de vários métodos na deteção de intrusões, nomeadamente métodos baseados em assinaturas, anomalias, ou na análise de estados de protocolos. A eficiência de cada um dos métodos varia com o tipo de ameaça. Um IDS utiliza vários métodos de forma isolada ou em conjunto com o intuito de ampliar o âmbito e a eficácia da deteção.

2.3.1 Baseado em Assinaturas

No contexto de um IDS, uma assinatura ou regra, representa um padrão que corresponde a uma ameaça conhecida. Este método realiza a comparação das assinaturas com eventos observados de forma a identificar possíveis incidentes.

A análise de assinaturas é a técnica de deteção mais simples, sendo particularmente eficaz contra ameaças conhecidas e bem caracterizadas. No entanto, mostra-se ineficaz contra ameaças previamente desconhecidas ou na análise de mensagens que apresentam variações em relação aos modelos predefinidos. É igualmente suscetível a ataques dissimulados com recurso a técnicas evasivas. O lançamento de novas versões das aplicações monitorizadas obriga à publicação de novas assinaturas. Isto significa que os sistemas monitorizados encontram-se vulneráveis até receberem atualizações do fabricante do IDS.

2.3.2 Baseado em Anomalias

Este método utiliza perfis que representam o comportamento normal dos utilizadores, dispositivos, comunicações de rede e aplicações monitorizadas. Os eventos observados que apresentam desvios em relação aos perfis ativos são considerados anómalos e classificados como incidentes.

Um perfil é gerado a partir da monitorização e registo do comportamento típico de sistemas durante um período de aprendizagem que decorre ao longo de vários dias ou semanas. Os perfis gerados podem ser estáticos ou dinâmicos. Perfis estáticos não são alterados durante o funcionamento do sistema e devem ser reconstruídos periodicamente. Perfis dinâmicos podem ser alterados em tempo real, em função dos eventos observados. Este método apresenta maior eficácia contra novas ameaças, previamente desconhecidas. Apresenta algumas desvantagens, nomeadamente a dificuldade da geração de perfis com alta fidelidade, pois o funcionamento típico da rede pode ser complexo e variável. Existe igualmente o risco de inclusão inadvertida de atividade maliciosa durante o processo de criação dos perfis. Um IDS baseado em anomalias produz uma grande quantidade de falsos positivos devido à deteção de atividade inofensiva que apresenta desvios em relação ao perfil ativo. Comparativamente com IDS baseados em assinaturas é mais difícil efetuar a correspondência direta entre o evento observado e a causa da geração do incidente.

2.3.3 Baseado em Análise de Estados de Protocolos

Esta metodologia utiliza modelos que representam o funcionamento correto de protocolos conhecidos. Os eventos observados que não correspondem às definições formais dos protocolos são considerados incidentes. Durante o processo de monitorização, o IDS mantém o registo do estado das ligações estabelecidas, podendo assim identificar sequências incorretas ou inesperadas de comandos. É igualmente possível identificar utilizadores autenticados nas várias sessões e definir perfis de atividade aceitável individualmente ou em grupos.

Esta técnica de deteção apresenta maior eficácia na monitorização de sistemas e aplicações que implementam de forma estrita os protocolos definidos por fabricantes e entidades reguladoras. A eficácia é reduzida quando os modelos padrão utilizados estão incompletos, originando variações nas implementações de produtos comerciais. Muitos fabricantes não publicam as especificações completas de protocolos proprietários, reduzindo a fidelidade dos modelos utilizados. Alterações e atualizações de protocolos e de produtos que os implementam obrigam à atualização dos modelos de IDS. Uma desvantagem desta metodologia é a elevada utilização de recursos de sistema devido à complexidade de análise dos protocolos e controlo de sessões ativas de utilizadores. Em comparação com as outras metodologias, não deteta ataques que não violem o funcionamento aceitável dos protocolos.

2.4 Tipologias

Os IDS são classificados em vários tipos, dependendo da aplicação a que se destinam e das funcionalidades pretendidas. Cada um dos tipos de IDS descritos de seguida utiliza um ou mais dos métodos de deteção descritos anteriormente.

Baseado na Rede

Monitorização das comunicações em segmentos de rede ou em dispositivos específicos. Análise de atividade observada dos protocolos nos níveis de rede, transporte e aplicação. Geralmente instalado em dispositivos na fronteira entre segmentos de rede, tais como *routers*, *firewalls*, servidores VPN, servidores de acesso remoto e redes sem fios.

Redes Sem Fios

Utilizado na monitorização de atividades em redes sem fios. Neste tipo de IDS não é realizada a análise do tráfego contido nas comunicações, mas apenas do funcionamento de protocolos específicos de tecnologias de redes sem fios. Geralmente instalado junto a dispositivos de redes sem fios e também em locais onde seja interdita a existência deste tipo de redes.

Análise Comportamental da Rede

Análise de tráfego de rede de forma a identificar ameaças que geram fluxos de tráfego que correspondem a padrões de tráfego suspeito. Geralmente instalado em redes internas ou em pontos de confluência de fluxos de tráfego interno e externo.

Baseado no Dispositivo

Este tipo de IDS realiza a monitorização de um único dispositivo efetuando o controlo de segurança a vários níveis, tais como, tráfego de rede, ficheiros de registo de eventos do sistema, processos em execução, atividade de aplicações, acesso a ficheiros, alteração de configurações. Geralmente instalado em dispositivos com importância crítica, tais como servidores acessíveis a partir do exterior ou que contêm informações confidenciais.

2.5 Arquitetura

Um IDS é constituído por um conjunto de componentes tecnológicos que funcionam de forma concertada. Desde os elementos que capturam os eventos, passando pelo armazenamento de dados e metadados, até à interface que apresenta os resultados das análises dos eventos e gera os alertas destinados aos administradores. A arquitetura específica de cada IDS depende do sistema monitorizado que lhe está subjacente. Existem, no entanto, alguns componentes comuns a todos os tipos de IDS.

De seguida descrevem-se os vários componentes elementares de um IDS. A Fig. 2.2 representa uma arquitetura típica de aplicação de IDS baseado em redes e em dispositivos.

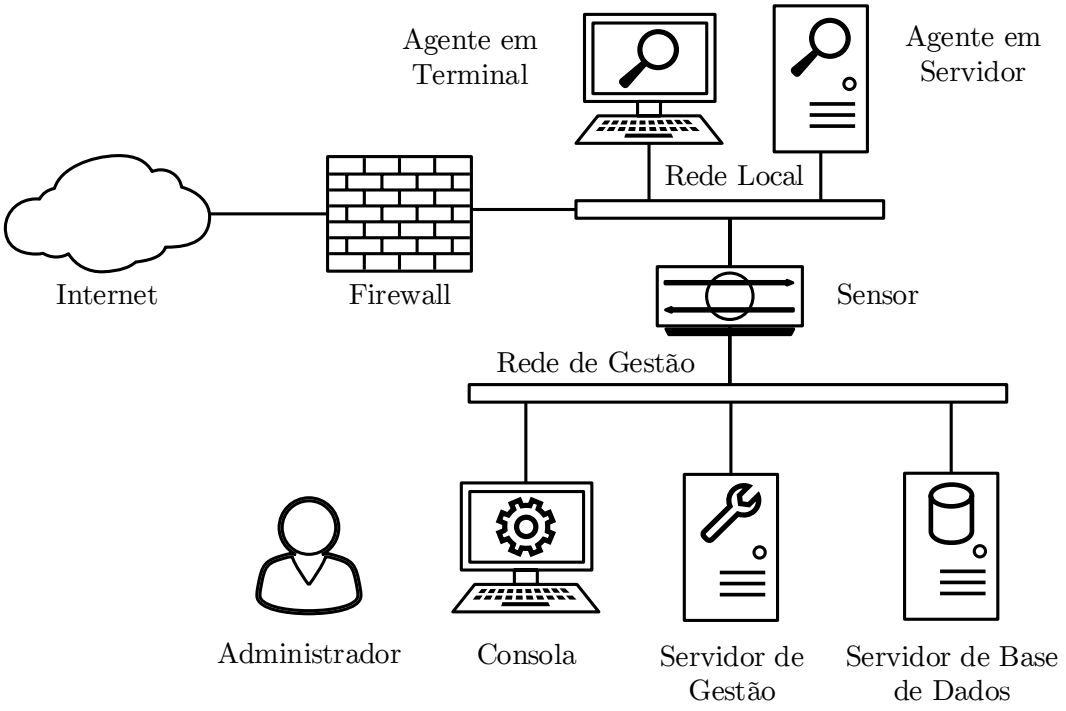

FIGURA 2.2: Arquitetura típica de um IDS baseado na rede e em dispositivos.

Sensor/Agente: monitoriza e analisa a atividade em redes e sistemas. O termo **sensor** é utilizado em IDS baseado em redes e pode ser implementado como um dispositivo ou um programa de computador. O termo **agente** aplica-se em IDS baseado em dispositivos e consiste num programa de computador.

Servidor de Gestão: dispositivo que recebe informação proveniente dos sensores e agentes e gere o funcionamento dos mesmos. Realiza uma análise mais detalhada da informação recolhida pelos sensores e efetua a correlação de eventos registados por vários sensores. Dependendo da dimensão da rede monitorizada, podem existir vários servidores de gestão, organizados hierarquicamente em dois níveis de controlo.

Servidor de Base de Dados: armazena a informação gerada pelos sensores. Pode armazenar apenas informação relativa aos eventos, bem como os próprios dados capturados durante a monitorização. pode ser implementado de forma integrada no mesmo dispositivo que o servidor de gestão, ou num dispositivo dedicado.

Consola: programa informático que fornece uma interface de controlo do IDS aos utilizadores e administradores. A consola é utilizada para a monitorização e/ou configuração de sensores e agentes, bem como os vários servidores de gestão e bases de dados do sistema.

Rede de Gestão: a gestão dos componentes de um IDS pode ser realizada através das redes de dados da organização ou de uma rede dedicada. Obtém-se assim um isolamento entre a rede de gestão do IDS e as redes de dados da organização, ocultando dos atacantes a existência dos componentes e das suas características técnicas.

2.6 Sistemas de Deteção e Prevenção de Intrusões

Atualmente os IDS mais comuns utilizam a deteção baseada em assinaturas a partir de tráfego capturado por sensores/agentes. Este método é particularmente eficiente quando o número de assinaturas a analisar é reduzido [19]. Devido ao aumento exponencial de ameaças nas últimas décadas, um IDS eficiente deve suportar a constante atualização de assinaturas e permitir a modificação da lista de assinaturas ativas em cada momento.

A escolha entre um IDS comercial ou de utilização livre não deve depender apenas do custo de aquisição. É igualmente importante considerar o nível de qualificação dos operadores do sistema e a capacidade de adaptação do mesmo às necessidades especificas da organização.

Sistemas Comerciais Proprietários

Os sistemas comerciais estão geralmente associados à aquisição de equipamento especializado e à subscrição de contratos de serviços dispendiosos. Apresentam vantagens na existência de suporte técnico contínuo e atualização de assinaturas para as ameaças mais recentes. A utilização de tecnologias proprietárias dificulta a integração com outros sistemas e impossibilita geralmente a modificação da arquitetura interna do sistema.

Sistemas de Utilização Livre

Alternativamente, os sistemas de utilização livre não obrigam a aquisição de equipamento especifico. Estes sistemas não possuem suporte técnico nem disponibilização de assinaturas com a mesma rapidez que sistemas comerciais. A eficiência do sistema depende diretamente da qualificação dos técnicos responsáveis pela instalação, configuração e manutenção do sistema.

A arquitetura aberta dos sistemas de utilização livre torna-os igualmente mais adequados para aplicação em contexto académico. Dos vários sistemas disponíveis atualmente, destacam-se o Snort e o Suricata, que se descrevem de seguida.

2.6.1 Snort

Inicialmente desenvolvido por Martin Roesch em 1998, o Snort é o IDS mais utilizado em todo o mundo [18]. Fruto do grande sucesso do sistema, o seu criador fundou a empresa Sourcefire em 2001 (adquirida em 2013 pela Cisco Systems). A Sourcefire produz e comercializa equipamentos de segurança que integram uma versão comercial do Snort. Continua no entanto a existir uma versão gratuita disponível ao público em geral[7].

O Snort pode funcionar num de três modos: *sniffer*, *packet logger* ou IDS. Os dois primeiros modos efetuam apenas a captura de tráfego. No modo IDS os pacotes capturados são descodificados e analisados sequencialmente de acordo com uma lista de assinaturas predefinidas. As assinaturas podem ser construídas manualmente ou obtidas de fontes externas em versões pagas ou gratuitas.

O Snort possui uma arquitetura modular que permite a expansão da funcionalidade do sistema (Fig. 2.3). Na sua forma mais simples apenas é produzida informação visualizável na consola de controlo. A produção de alertas em formatos estruturados depende da instalação de módulos externos. Existem igualmente vários módulos que interpretam os alertas gerados e apresentam a informação graficamente.

[7]Sourcefire *Open Source* - `http://www.sourcefire.com/products/open-source`

2.6.2 Suricata

O Suricata é um IDS gratuito e de utilização livre criado em 2010. O desenvolvimento do sistema é controlado pela OISF[8], uma fundação sem fins lucrativos patrocinada pelo DHS[9]. O Suricata é desenvolvido por uma equipa multinacional de especialistas na indústria da segurança informática que contam com o apoio técnico de um consórcio das empresas líderes na área da cibersegurança. Este projeto é completamente acessível ao público em geral e os seus autores incentivam a comunidade a contribuir com críticas e sugestões.

Arquitetura

O Suricata possui uma arquitetura semelhante ao Snort. Ambos os sistemas efetuam inicialmente a captura do tráfego, realizando de seguida a descodificação dos pacotes e o pré-processamento da informação nos cabeçalhos, de seguida ocorre a inspeção do conteúdo das comunicações e aplicação das assinaturas correspondentes a ameaças conhecidas. Finalmente, na eventualidade de incidentes são emitidos alertas (Fig. 2.4).

Desempenho

O Suricata apresenta uma grande escalabilidade devido a suportar múltiplas *threads* em qualquer uma das fases do processo de deteção de intrusões. Isto permite explorar plenamente as capacidades de processadores com múltiplos núcleos. São igualmente suportadas arquiteturas existentes em placas gráficas com processamento vetorial tais como NVIDIA CUDA[10] e Tilera que permitem a implementação de paradigmas de programação paralela em larga escala.

Identificação de Protocolos

Os protocolos de rede mais comuns são automaticamente reconhecidos pelo Suricata, independentemente das portas TCP/UDP utilizadas. Desta forma, apenas se aplicam as assinaturas ao tipo de tráfego correspondente. É possível definir palavras-chave correspondentes a campos nos cabeçalhos dos protocolos da família TCP/IP,

[8] Open Information Security Foundation
[9] Department of Homeland Security.
[10] *Compute Unified Device Architecture.*

bem como SSL/TLS, aumentando a granularidade da inspeção de tráfego e consequentemente a eficiência do IDS.

Identificação de Ficheiros

Os ficheiros transferidos na rede podem ser identificados através de palavras-chave correspondentes a campos dos cabeçalhos, ou do valor da *hashfile*[11] MD5. Os ficheiros que correspondem a estes critérios podem ser bloqueados e armazenados juntamente com informação relevante ao evento que originou a sua captura.

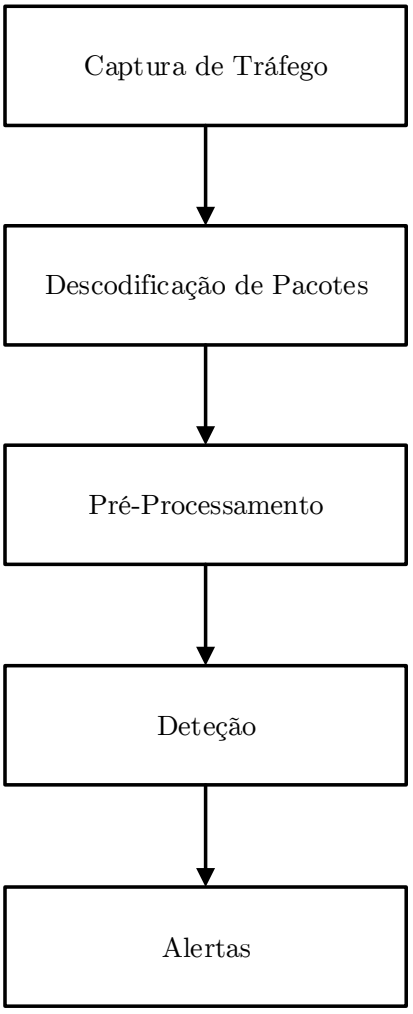

FIGURA 2.3: Arquitetura Snort.

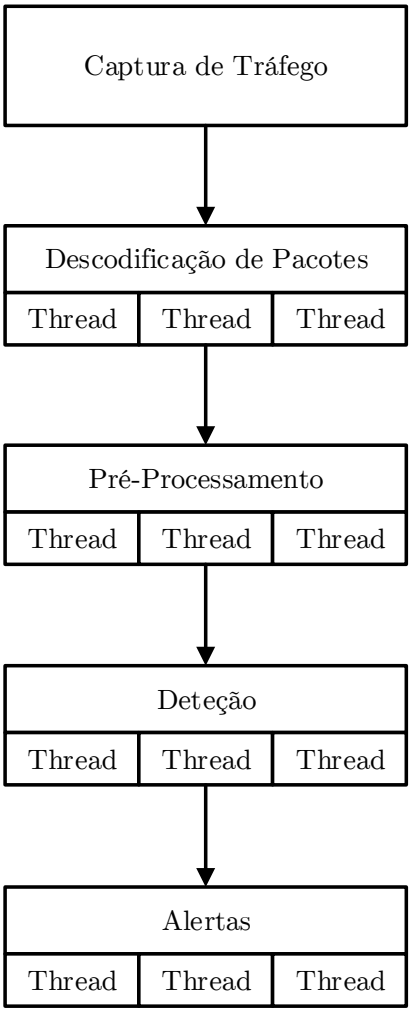

FIGURA 2.4: Arquitetura Suricata.

[11] Função matemática que efetua a soma de verificação do ficheiro.

Alertas

O Suricata suporta vários formatos de alertas. Para além da simples visualização de eventos na consola, estes podem ser registados em ficheiros estruturados de acordo com padrões utilizados na indústria (json, unified2, prelude, syslog, pcap). A interpretação dos ficheiros de alertas e a sua transmissão na rede dependem da utilização de módulos externos, geralmente compatíveis com o Snort.

Interfaces Gráficas

Os alertas gerados pelo Suricata consistem em simples ficheiros de texto cujo conteúdo se encontra estruturado de forma muito compacta. Para facilitar a análise dos eventos detetados existem vários módulos e aplicações que proporcionam uma interface gráfica mais intuitiva. A escolha da interface a utilizar depende do formato de alertas previamente configurado. Soluções como o OSSIM da Alien Vault podem interpretar alertas de dispositivos IDS de vários fabricantes e em vários formatos.

Snorby: `https://snorby.org`

Sguil: `https://bammv.github.io/sguil`

SqueRT: `http://www.squertproject.org`

BASE: `http://base.professionallyevil.com`

OSSIM: `http://www.alienvault.com/solutions/siem-log-management`

Alternativamente, podem ser utilizadas aplicações de monitorização de eventos genéricos.

Splunk: `http://www.splunk.com`

Logstash: `http://logstash.net`

ELSA: `https://code.google.com/p/enterprise-log-search-and-archive`

Esta solução não produz relatórios com o mesmo nível de detalhe que soluções dedicadas, todavia pode ser útil no cruzamento de informação proveniente de dispositivos IDS e de outros equipamentos de rede e servidores da organização.

Capítulo 3

Sistema Distribuído de Deteção de Intrusões em Redes Sem Fios

A segurança em redes sem fios encontra-se suscetível a um número crescente de ameaças, agravadas pela popularidade de dispositivos móveis de comunicação pessoal que criam novas vias de acesso à rede interna das organizações.

Os mecanismos de segurança em sistemas distribuídos, contrariamente aos sistemas tradicionais monolíticos, estão dispersos por toda a organização. Este requisito apresenta novos desafios tecnológicos e logísticos.

Os principais fabricantes de equipamentos de segurança em redes apresentam continuamente soluções para este problema. Não obstante a sua eficácia, estes sistemas são geralmente complexos e dispendiosos.

Alternativamente, propõe-se um sistema baseado em tecnologias de utilização livre que exploram as potencialidades de novas plataformas multi-processador, com uma arquitetura de rede com grande escalabilidade e que apresenta uma interface central de controlo simples, garantindo simultaneamente a segurança das comunicações do sistema.

Neste capítulo introduz-se o problema da segurança em redes sem fios. Enumeram-se as principais ameaças neste âmbito, nomeadamente em dispositivos móveis. Caracterizam-se os vários tipos de sistemas de segurança em redes distribuídas e descrevem-se-se exemplos de sistemas comerciais. Finalmente apresenta-se a proposta de um novo sistema distribuído de deteção de intrusões em redes sem fios e descrevem-se as suas características fundamentais.

3.1 Segurança em Redes Sem Fios

As redes com fios tradicionais possuem interfaces físicas bem definidas que separam a rede interna de uma organização de redes externas (tipicamente a Internet). Em torno de cada uma destas interfaces é instalado um **perímetro de segurança**, cuja função é policiar o tráfego em ambos os sentidos, bloqueando as comunicações que violem as políticas de segurança da organização [4].

O sucesso do modelo de segurança baseado em controlos no perímetro baseia-se nas seguintes premissas:

- todas as ameaças externas são bloqueadas;
- não existem caminhos alternativos de ataque;
- não existem ameaças internas.

O surgimento das redes locais sem fios baseadas na norma 802.11 motiva uma mudança de paradigma. A utilização de dispositivos móveis veio atenuar os limites tradicionais entre redes internas e externas. Ocorre uma mudança de prioridade do perímetro físico para as redes locais internas e para os próprios dispositivos.

A proliferação de dispositivos móveis com capacidade de ligação a redes sem fios apresenta novos desafios à segurança das redes e da informação organizacional. Tornou-se comum que colaboradores utilizem os seus dispositivos pessoais para aceder à rede interna da organização (BYOD[1]).

Este nível de mobilidade representa uma dilema para os responsáveis pela segurança das redes. O acesso à informação da organização deve ser facilitado de forma a aumentar a produtividade dos colaboradores, por outro lado devem ser implementadas políticas de segurança sucessivamente mais restritivas.

3.1.1 Ameaças em Redes Sem Fios

O facto das radiações eletromagnéticas permearem os limites físicos de instalações e poderem ser livremente capturadas por entidades externas cria novos problemas que não podem ser resolvidos de forma tradicional [17].

[1] *Bring Your Own Device.*

Pontos de acesso mal configurados: configurações predefinidas inseguras. Falhas na atualização das configurações dos equipamentos após alteração de políticas de segurança.

Pontos de acesso não autorizados: podem ser criados de forma negligente pelos próprios utilizadores legítimos, ou por atacantes com objetivos maliciosos.

Denial of Service: atacantes utilizam sinais eletromagnéticos para criar interferências e perturbar o funcionamento normal da rede ou levar ao bloqueio total das comunicações.

Captura de comunicações: atacantes capturam tráfego de redes que utilizem protocolos inseguros. Este ataque é realizado de forma passiva e completamente indetetável.

3.1.2 Ameaças em Dispositivos Móveis

Os próprios dispositivos móveis representam um novo vetor de ataque. A confluência de utilização pessoal e organizacional destes dispositivos impede a aplicação plena das políticas de segurança das organizações.

De entre os vários sistemas operativos que constituem o ecossistema dos dispositivos móveis, os sistemas *Android* representam 97% dos alvos dos atacantes. Este facto deve-se principalmente à falta de regulação da origem das aplicações[2].

Além das técnicas clássicas de ataques, identificam-se adicionalmente três novos tipos especificamente direcionados à plataforma *Android* [12]:

***Repackaging*:** o atacante descarrega uma aplicação legitima, adiciona conteúdo malicioso e volta a disponibilizar a aplicação com o mesmo nome no *Android marketplace*.

***Update attack*:** o atacante disponibiliza atualizações com conteúdo malicioso para aplicações legitimas.

***Drive-by download*:** redirecionamento de utilizadores para descarregarem inadvertidamente conteúdos maliciosos através de ligações dissimuladas em publicidades ou códigos QR[3].

[2] *Blue Coat Systems 2013 Mobile Malware Report.*
[3] *Quick Response Code.*

3.2 Segurança em Sistemas Distribuídos

A grande diversidade dos tipos de ameaças existentes numa rede sem fios leva a que sejam necessários controlos de segurança a vários níveis [8].

Meio de Comunicação: proteger os equipamentos da rede contra interferências e outros ataques disruptivos ao nível físico e de ligação de dados.

Protocolos de Rede: analisar os protocolos e o conteúdo das comunicações e detetar ataques aos níveis da rede e de transporte.

Aplicações no Dispositivo: monitorizar diretamente a atividade no próprio dispositivo. Detetar tentativas de exploração de vulnerabilidades do sistema operativo e das aplicações.

O caráter inerentemente distribuído de uma rede local sem fios significa que os vários controlos de segurança devem ser implementados localmente.

A conjugação de diferentes métodos de deteção de intrusões origina arquiteturas de rede que são classificadas em função da distribuição geográfica e da organização hierárquica do equipamento de rede. A escolha da estrutura mais correta em cada situação depende principalmente da capacidade de processamento da infraestrutura de rede sem fios [16].

3.2.1 Distribuição Geográfica

Ao desenvolver a arquitetura das redes sem fios de uma organização, existem duas opções relativamente à utilização de dispositivos de deteção de intrusões. Estes podem funcionar numa rede sobreposta ou integrados na rede existente.

Rede Sobreposta

Consiste num conjunto de dispositivos IDS que formam uma rede completamente independente da infraestrutura de rede sem fios existente.

Devido à utilização de dispositivos dedicados, são otimizados os recursos de monitorização sem afetar o desempenho da rede. Acrescenta resistência a ataques DoS[4] e cada IDS pode monitorizar vários pontos de acesso próximos.

[4] *Denial of Service.*

Apresenta, no entanto, um custo superior e um encargo maior para os responsáveis técnicos e permite apenas monitorização aos níveis físico e de ligação de dados.

Rede Integrada

A deteção de intrusões é realizada de forma total ou parcial pelos próprios dispositivos da infraestrutura de rede.

Utiliza a capacidade de processamento e os meios de comunicação da rede para analisar diretamente o tráfego nos vários níveis da rede. Reduz a latência na deteção de ataques e não requer a aquisição de equipamento adicional.

Esta opção apenas é viável se os pontos de acesso sem fios possuírem capacidade de processamento suficiente. Caso contrário, pode reduzir o desempenho da rede. Não apresenta redundância e um ataque bem sucedido à rede pode igualmente desativar o IDS.

3.2.2 Estrutura Hierárquica

O processo de deteção de intrusões contempla várias fases que devem ocorrer sequencialmente, nomeadamente a captura de tráfego, análise, deteção e emissão de alertas. As fases seguintes à captura de tráfego podem ser realizadas por um dispositivo central ou pelos próprios equipamentos de rede.

Sistema Centralizado

Neste modelo hierárquico, os pontos de acesso à rede sem fios realizam apenas a captura do tráfego, reencaminhando de seguida o mesmo para dispositivos especializados que efetuam a análise e deteção de intrusões e emissão de possíveis alertas (Fig. 3.1).

Esta opção adequa-se a equipamentos de rede com funcionalidades de captura de tráfego, mas sem capacidade de processamento para efetuar a análise e deteção de intrusões.

Requer a aquisição de equipamentos de deteção de intrusões dispendiosos com grande capacidade de processamento e armazenamento. O reencaminhamento integral do tráfego de cada uma das redes locais utiliza recursos da infraestrutura da rede de dados e aumenta as possibilidades de interceção das comunicações.

Sistema Distribuído

Num sistema completamente distribuído, todas as fases do processo de deteção de intrusões são realizadas pelos próprios equipamentos de rede sem fios (Fig. 3.2).

Este modelo elimina a necessidade de reencaminhar dados para redes externas, realizando o isolamento dos dados correspondentes a cada rede local. Cada ponto de acesso a redes sem fios tem apenas que analisar as comunicações dos dispositivos que lhe estão diretamente ligados. O número potencialmente reduzido de sistemas e protocolos permite a aplicação de listas de assinaturas mais reduzidas, aumentando a eficiência do IDS e reduzindo a capacidade de processamento necessária.

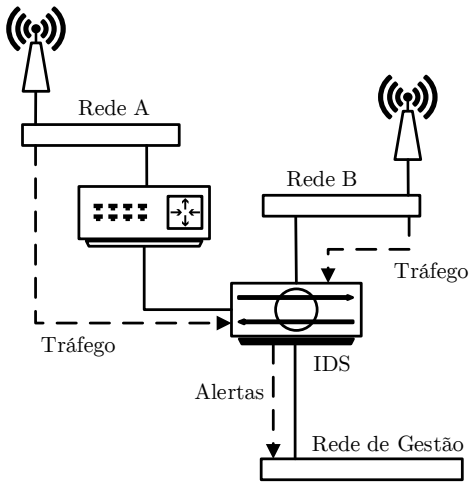

FIGURA 3.1: Sistema centralizado.

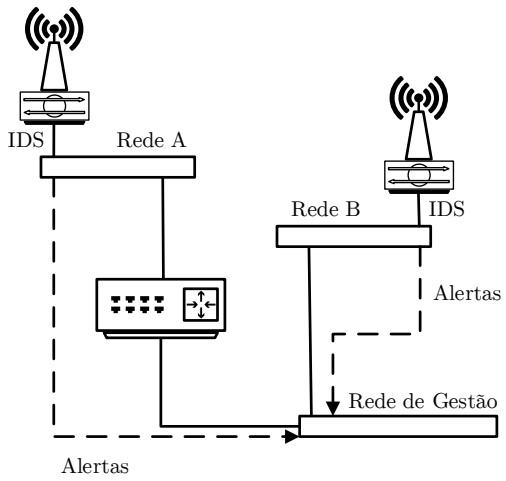

FIGURA 3.2: Sistema distribuído.

Uma arquitetura distribuída requer equipamentos de rede que efetuem simultaneamente a função de ponto de acesso à rede sem fios e de IDS. Estes equipamentos possuem recursos inferiores a um IDS dedicado, no entanto são muito menos dispendiosos. Um desafio particular deste tipo de sistemas é obter uma visão global dos vários equipamentos na rede e efetuar a gestão das configurações de cada IDS remotamente.

3.3 Sistemas Comerciais

Os principais fabricantes abordam o problema da segurança em redes sem fios como uma extensão da infraestrutura da rede com fios. O mais recente estudo de mercado

da Gartner sobre equipamento de segurança[5] identifica uma dezena de fabricantes considerados líderes tecnológicos no setor . Após a análise do portfólio de cada fabricante, conclui-se que estes equipamentos são geralmente integrados na rede de dados e estruturados em torno de um dispositivo UTM[6] centralizado que realiza simultaneamente as tarefas de *firewall* e IDS. Apenas dois fabricantes oferecem sistemas completamente distribuídos. A Tab. 3.1 apresenta um resumo das características das soluções de cada fabricante, indicando os níveis de rede analisados e a designação comercial dos equipamentos.

Fabricante	Modelo	IDS (níveis)		Estrutura	
		1-2	3-7	Centralizada	Distribuída
AirTight	WIPS	X	-	X	-
Check Point	1100 UTM-1 Edge N	X	X	-	X
Cisco	Aironet IPS/ASA	X	X	X	-
Dell	SonicPoint-N SonicWall	X	X	X	-
Fluke Networks	AirMagnet Enterprise	X	-	X	-
Fortinet	FortiAP FortiGate	X	X	X	-
	FortiWiFi	X	X	-	X
Juniper	WLA SRX	X	X	X	-
Motorola	AP AirDefense	X	X	X	-
Sophos	AP UTM	X	X	X	-
Watchguard	AP	X	-	X	-

Tabela 3.1: Sistemas comerciais.

[5] *Gartner Magic Quadrant UTM 2013.*
[6] *Unified Threat Management.*

3.4 Sistema Proposto

O presente trabalho apresenta um novo Sistema Distribuído de Deteção de Intrusões em Redes Sem Fios. Este sistema possui as principais funcionalidades dos sistemas comerciais disponíveis no mercado. O sistema cumpre os seguintes requisitos:

- arquitetura de rede com alta escalabilidade;
- plataformas tecnológicas mais recentes e de múltiplo processamento;
- tecnologias e ferramentas gratuitas e de utilização livre;
- comunicações seguras entre os elementos do sistema;
- gestão centralizada de configurações.

Arquitetura de Rede

A maioria dos sistemas comerciais apresentam uma arquitetura de rede em que os sensores estão integrados nos equipamentos da rede de dados e onde o tráfego é capturado integralmente e reencaminhado para dispositivos IDS centralizados. Este modelo de rede apresenta baixa escalabilidade, visto que a expansão da rede implica a aquisição de novos e dispendiosos equipamentos.

A visão tradicional de pontos de acesso a redes sem fios diretamente ligados a uma infraestrutura física bem definida está a dar lugar a um paradigma de rede distribuída em malha cuja estrutura e dimensão variam dinamicamente. Recentemente, os líderes mundiais na área das tecnologias da informação: Intel[7], Cisco[8] e Google[9] apresentaram iniciativas para a implementação de redes sem fios distribuídas em larga escala e com grande densidade de dispositivos clientes. A utilização de sistemas IDS convencionais em redes desta dimensão implica o aumento da quantidade de tráfego capturado e transmitido na rede de dados, provocando a saturação da largura de banda da rede física e possíveis quebras no funcionamento da rede da organização.

[7] *Intel - Internet of Things* - `http://www.intel.com/content/www/us/en/internet-of-things`.

[8] *Cisco - Tomorrow Starts Here* - `http://www.cisco.com/web/tomorrow-starts-here`.

[9] *Google - Project Loon* - `http://www.google.com/loon`.

Alguns fabricantes disponibilizam sistemas completamente distribuídos, resolvendo assim o problema da escalabilidade. A deteção de intrusões no próprio equipamento de rede garante o isolamento local da informação, no entanto as comunicações relativas à gestão do próprio IDS continuam a partilhar o mesmo meio de comunicação da rede de dados. Isto significa que um ataque com sucesso à infraestrutura de rede de dados coloca igualmente em risco os sistemas de segurança.

A arquitetura de rede do sistema proposto neste trabalho é constituída por sensores integrados nos equipamentos de rede sem fios e utiliza meios de comunicação dedicados para a rede de dados e para a rede de gestão do IDS. Esta solução possui simultaneamente as vantagens do isolamento em redes sobrepostas e escalabilidade em redes integradas.

Plataforma Tecnológica

Os pontos de acesso à rede sem fios tradicionais baseiam-se em plataformas tecnológicas extremamente limitadas ao nível da capacidade de processamento e memória. Estes sistemas utilizam normalmente arquiteturas MIPS ou ARM mais rudimentares e com ênfase no baixo consumo energético.

As soluções comerciais de IDS em redes sem fios que utilizam este tipo de equipamento mais simples implementam apenas sistemas com redes centralizadas. A baixa capacidade de processamento é um dos fatores limitativos na implementação de sistemas IDS distribuídos. Ambos os fabricantes que oferecem soluções realmente distribuídas recorrem a tecnologias ASIC[10] especialmente desenvolvidas para a análise de tráfego de rede. Este tipo de plataforma utiliza tecnologias proprietárias do fabricante, limitando potencialmente o nível de adaptação às necessidades da organização e reduzindo a compatibilidade com outras tecnologias e ferramentas.

Recentemente assistiu-se a um avanço tecnológico sem precedentes na indústria de microprocessadores, fruto do desenvolvimento de dispositivos móveis com capacidades multimédia. A empresa ARM anunciou recentemente a familia Cortex A-50[11] com 16 núcleos de processamento. A AMD disponibiliza várias gamas de processadores com arquitetura ARM[12] destinados ao mercado de servidores e comunicações, com capacidade de processamento paralelo em larga escala. A Intel apresentou a

[10] *Application Specific Integrated Circuit.*
[11] ARM Cortex-A50 - `http://www.arm.com/products/processors/cortex-a50`.
[12] AMD ARM - `http://www.amd.com/en-us/solutions/embedded/communications`.

iniciativa NUC[13], que permite obter níveis de processamento equivalentes a estações de trabalho profissionais, baseado no novo formato UCFF[14] de apenas 10x10 cm.

O sistema proposto neste trabalho foi projetado de forma a explorar as potencialidades das novas plataformas tecnológicas existentes no mercado. As ferramentas e os programas desenvolvidos implementam técnicas de programação concorrente que distribuem as tarefas a executar pelos vários núcleos de processamento disponíveis. A adoção de plataformas padrão permite aos profissionais de segurança da informação adaptarem livremente as capacidades do sistema às necessidades da organização.

Tecnologias de utilização livre

As aplicações de gestão e os protocolos de comunicação dos sistemas comerciais utilizam tecnologias proprietárias de cada fabricante. Ao optar por um determinado fabricante, o cliente limita as possibilidades de integração com sistemas de outros fabricantes ou com sistemas e protocolos de utilização livre.

Um dos objetivos na idealização e desenvolvimento deste projeto foi a utilização de ferramentas que permitem o desenvolvimento modular e incremental das funcionalidades do sistema com recurso a tecnologias inovadoras e completamente livres. O sistema proposto utiliza tecnologias de utilização livre em todas as fases do seu desenvolvimento, nomeadamente:

Sistema operativo: Debian Linux.

Linguagens de programação: Python e Javascript.

Ambiente integrado de desenvolvimento: Eclipse + PyDev.

Comunicação em rede: sockets ØMQ e UNIX BSD.

Base de dados: MongoDB + Motor.

Servidor web: Tornado.

[13]Intel NUC - `http://www.intel.com/content/www/us/en/nuc/overview.html`.
[14]Ultra Compact Form Factor.

Comunicações Seguras

Uma característica fundamental de um sistema de segurança da informação é a capacidade de garantir a segurança das suas próprias comunicações internas. Os sistemas tradicionais baseiam-se numa arquitetura integrada que prevê a existência de uma infraestrutura física dedicada para a rede de gestão. Esta arquitetura permite um maior controlo do fluxo do tráfego, no entanto é menos flexível e oferece uma escalabilidade inferior.

Um sistema com uma rede de sensores completamente distribuída não requer a ligação direta dos sensores a uma rede física. Desta forma, as comunicações da rede de gestão realizam-se através de ligações de rede sem fios. Além dos meios de segurança existentes nos níveis de rede inferiores, tais como o WPA2, devem ser implementados mecanismos de segurança mais abrangentes. As mensagens são cifradas com recurso a criptografia de chaves públicas garantir a segurança em comunicações ponto-a-ponto em redes dinâmicas.

Desde a sua invenção em 1977, o algoritmo RSA tornou-se o mais utilizado em sistemas criptográficos de chaves públicas. A segurança do algoritmo RSA baseia-se na dificuldade de fatorização de números primos. Com a evolução dos processadores e o surgimento de novos métodos de criptanálise, a dimensão aconselhada da chave a utilizar na cifra RSA aumentou consideravelmente, sendo atualmente de 2048 bits. A necessidade de maiores recursos de processamento e memória tornam indesejável a utilização deste algoritmo em dispositivos móveis e em redes distribuídas.

A criptografia de curvas elípticas (ECC[15]) surgiu como uma alternativa mais eficiente para sistemas criptográficos de chaves públicas. A segurança numa cifra ECC baseia-se na dificuldade de resolução do problema do logaritmo discreto, mais complexo do que a fatorização de números primos e para o qual ainda não é conhecido nenhum método computacionalmente eficiente. Isto significa que as cifras baseadas em ECC são mais adequadas para dispositivos com reduzidos recursos de processamento e memória. Estima-se que uma cifra ECC com chave de 224 bits oferece um nível de segurança superior a uma cifra RSA com chave de 2048 bits[16].

O sistema proposto neste trabalho utiliza o algoritmo CurveCP[17] apresentado por Daniel Bernstein em 2010. O CurveCP utiliza a curva 25519 obtida pela equação

[15] *Elliptic Curve Cryptography.*
[16] NSA ECC - `http://www.nsa.gov/business/programs/elliptic_curve.shtml`.
[17] CurveCP - `http://curvecp.org`.

$y^2 = x^3 + 486662x^2 + x$, com o número primo $2^{255} - 19$ e com o ponto base $x = 9$. A curva 25519 é utilizada em várias aplicações, por exemplo: rede Tor; OpenSSH 6.5; Apple iOS; DNSCurve. Todavia, as bibliotecas criptográficas baseadas no CurveCP encontram-se ainda em fase desenvolvimento e aconselha-se a sua utilização apenas em situações experimentais.

Gestão Centralizada

A natureza distribuída de um sistema deste género gera um novo desafio relacionado com a gestão das configurações dos vários elementos que o compõem. Os sistemas comerciais disponibilizam interfaces que permitem o controlo das configurações e a monitorização dos alertas em tempo real. Estas interfaces são geralmente complexas, embora sejam acompanhadas de manuais de utilização extensivos.

Os sistemas de utilização livre existentes não foram desenvolvidos expressamente para uma aplicação em redes distribuídas. Estes sistemas dispõem de vários meios de monitorização de alertas, no entanto continuam a permitir apenas uma configuração local através da edição manual de ficheiros de texto.

O sistema proposto neste trabalho dispõe de uma interface gráfica simples baseada na web que permite a gestão centralizada de todos os elementos do sistema, nomeadamente a criação e remoção de sensores, bem como a modificação de todos os parâmetros de configuração dos sensores e do próprio servidor. A interface permite igualmente a monitorização em tempo real do estado de funcionamento dos sensores.

3.4.1 Arquitetura de Rede

O sistema proposto é composto principalmente por dois tipos de elementos: servidor e sensores. Contrariamente a um sistema servidor-cliente tradicional, as comunicações não ocorrem necessariamente de forma direta entre o servidor e cada um dos sensores. No sistema proposto, cada sensor funciona igualmente como um intermediário que encaminha mensagens de outros sensores de e para o servidor. Cada sensor possui duas interfaces de rede, sendo uma dedicada à rede de dados e outra à rede de gestão do IDS (Fig. 3.3).

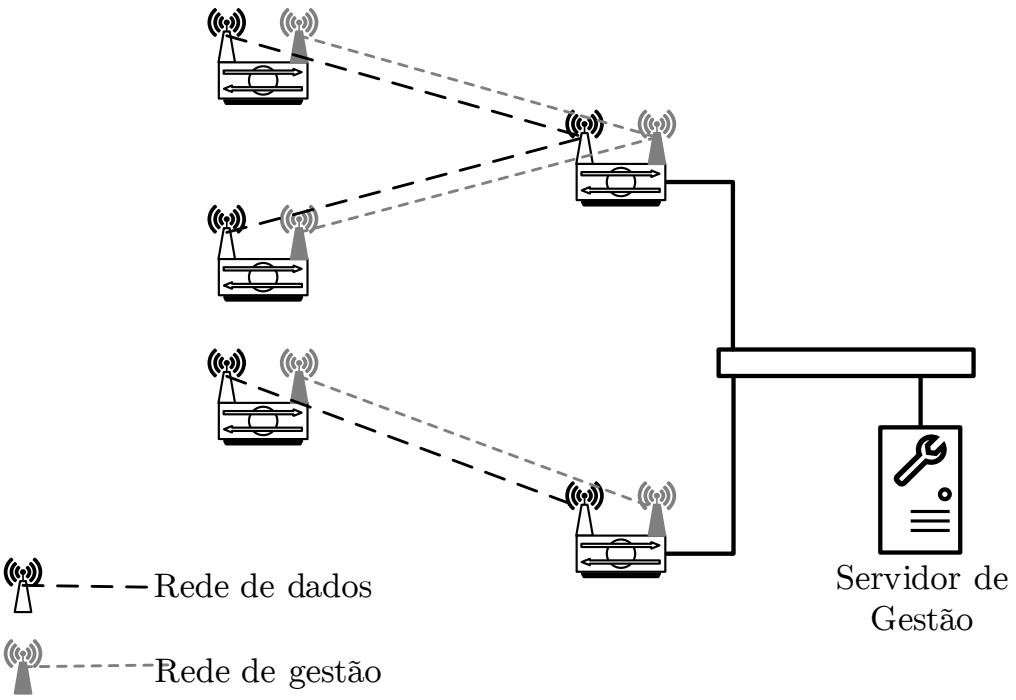

FIGURA 3.3: Arquitetura do sistema proposto.

Sockets ØMQ

Os *sockets* BSD tradicionais apresentam apenas dois modos básicos de funcionamento: servidor - recebe ligações de clientes; cliente - liga-se a um servidor ou a outro cliente, numa ligação ponto-a-ponto. A implementação de servidores que suportem a ligação de vários clientes implica a utilização de mecanismos de programação concorrente, normalmente com múltiplos processos ou *threads*. As aplicações cliente requerem igualmente a criação de um *socket* por cada ligação estabelecida. Estas limitações resultam em maior complexidade no desenvolvimento e funcionamento de aplicações distribuídas.

Os *sockets* ØMQ (ZeroMQ)[18] apresentam uma solução inovadora na implementação de sistemas altamente distríbuidos. Este novo tipo de *socket* apresenta uma API semelhante aos *sockets* BSD e está disponível para utilização livre em vinte e oito linguagens de programação. Implementam o conceito de *message queue* e internamente utilizam *threads* dedicadas para a leitura e escrita de mensagens, apresentando assim

[18] *Sockets* ØMQ - `http://zeromq.org`.

um funcionamento intrinsecamente assíncrono completamente transparente ao utilizador. A programação concorrente com *sockets* ØMQ (ZeroMQ) não requer técnicas de controlo tradicionais, como semáforos, *mutexes* ou *locks* [11].

Os *sockets* ØMQ possuem vários modos distintos de funcionamento:

REQ: envia pedidos e recebe respostas sequencialmente.

REP: recebe pedidos e envia respostas sequencialmente.

PUB: publica mensagens em modo *multicast*.

SUB: subscreve serviços de publicação de mensagens.

PUSH: distribui mensagens por um conjunto de clientes.

PULL: recebe mensagens de um distribuidor.

ROUTER: recebe pedidos de vários clientes.

DEALER: envia respostas para vários clientes.

PAIR: comunica exclusivamente com outro *socket* PAIR.

As combinações válidas dos vários modos de funcionamento permitem a implementação dos seguintes padrões de uso:

- **REQ-REP**
- **PUB-SUB**
- **REQ-ROUTER**
- **DEALER-REP**
- **DEALER-ROUTER**
- **DEALER-DEALER**
- **ROUTER-ROUTER**
- **PUSH-PULL**
- **PAIR-PAIR**

No desenvolvimento do sistema proposto utiliza-se o padrão DEALER-ROUTER .

3.4.2 Protocolo CurveZMQ

O protocolo CurveZMQ[19] permite a realização de comunicações seguras com *sockets* ØMQ. Este protocolo baseia-se no algoritmo CurveCP de criptografia de curvas elípticas, nomeadamente a curva 25519 desenvolvida por Daniel Bernstein.

O protocolo CurveZMQ utiliza o modelo de comunicação cliente/servidor, em que cada interveniente possui um par de chaves públicas/privadas permanentes e para cada ligação são gerados pares adicionais de chaves de curta duração, partilhadas de forma segura. A eficácia da criptografia de curvas elípticas depende da introdução de um número único em cada operação de cifra. Este número, designado de *nonce*[20] é escolhido pelo emissor da mensagem. O protocolo CurveZMQ utiliza dois tipos diferentes de *nonce*: um valor de 16 *bytes* criado por um gerador de números pseudo-aleatórios para proteger as chaves permanentes e um valor sequencial de 8 *bytes* para as chaves temporárias.

O funcionamento geral do CurveZMQ engloba o estabelecimento da ligação inicial (*handshake*), seguido de comunicação assíncrona em ambas as direções. Cada mensagem transmitida corresponde a um dos seguintes tipos de operação: HELLO; WELCOME; INITIATE; READY; MESSAGE e ERROR. O processo de comunicação segura ocorre da seguinte forma:

Início da ligação: A comunicação é sempre iniciada pelo cliente, que envia uma mensagem HELLO para o servidor. O servidor responde com uma mensagem WELCOME. Após esta troca inicial de mensagens, ambos os lados possuem a chave pública do lado oposto.

Autenticação: O cliente envia uma mensagem INITIATE e o servidor responde com uma mensagem READY. Após esta troca de mensagens, cada elemento realizou a autenticação do lado oposto. A partir deste momento pode ocorrer a troca livre de mensagens do tipo MESSAGE em qualquer ordem.

Terminação: Qualquer das partes pode terminar a ligação deixando de enviar mensagens até que se atinja um limite de tempo predefinido. O servidor pode recusar a mensagem de um cliente respondendo com uma mensagem ERROR.

[19] CurveZMQ - `http://curvezmq.org`.
[20] *Nonce - number used once.*

3.4.3 Protocolo de Encaminhamento

Os *sockets* ØMQ do tipo ROUTER guardam uma lista com as identidades e os endereços dos *sockets* aos quais estão diretamente ligados, para que desta forma possam manter múltiplas ligações simultâneas. O problema do encaminhamento surge devido ao facto dos sensores do sistema proposto funcionarem como encaminhadores para as mensagens de outros sensores.

Utiliza-se um protocolo de encaminhamento simplista que prevê apenas a existência de ligações com *sockets* DEALER-ROUTER, resultando numa rede aberta com uma topologia em árvore. O protocolo proposto requer apenas a implementação de uma tabela de encaminhamento em cada elemento do sistema e a transmissão de mensagens com campos que designam o sensor originário/destinatário da mensagem (`dst`) e o último/próximo sensor a transmitir a mensagem (`nxt`), consoante a direção da mesma. O protocolo resume-se na aplicação das seguintes regras:

Servidor recebe mensagem: adiciona originário como `dst` e último como `nxt`;

Servidor envia mensagem: coloca `dst` como destinatário e envia para `nxt`;

Sensor recebe mensagem no *socket* ROUTER: adiciona originário como `dst` e último como `nxt` e envia a mensagem pelo socket DEALER;

Sensor recebe mensagem no *socket* DEALER: coloca `dst` como destinatário e envia a mensagem para o `nxt` correspondente pelo *socket* ROUTER .

A Fig. 3.4 apresenta um exemplo de aplicação do protocolo.

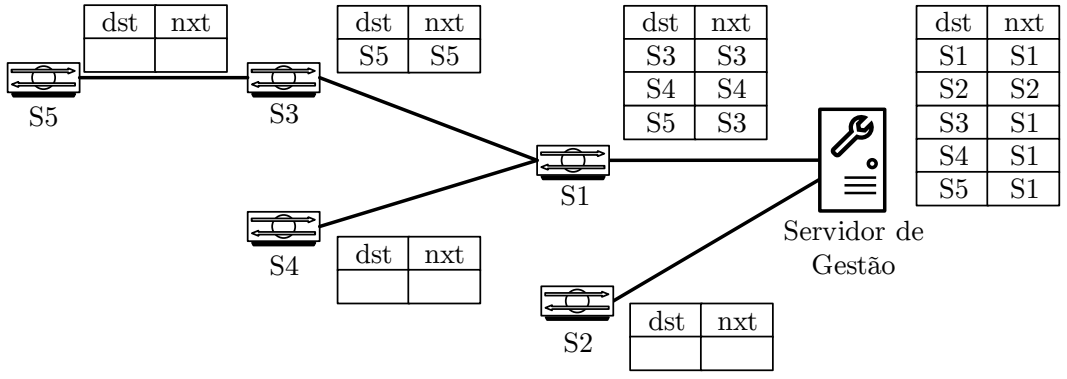

FIGURA 3.4: Protocolo de encaminhamento.

3.4.4 Protocolo de Comunicação

O funcionamento do sistema é definido por um protocolo de comunicação em rede que estabelece um conjunto de regras gerais que são posteriormente implementadas na aplicação. O protocolo de comunicação respeita os seguintes critérios:

Início da comunicação: o sensor deve iniciar sempre a comunicação com o servidor e nunca o inverso. Desta forma reduz-se a quantidade de mensagens na rede e evitam-se situações de impasse devido à simultaneidade de envio de mensagens.

Sincronização da configuração: após o estabelecimento da ligação inicial e antes de iniciar o processo de IDS, o sensor deve requisitar ao servidor a versão mais recente das configurações.

Atualização da configuração: após a modificação da configuração do sensor por parte do utilizador, o servidor deve enviar uma mensagem ao sensor para atualização das configurações e reinicio do processo de IDS.

Monitorização em tempo real: o sensor deve enviar periodicamente para o servidor mensagens com informação do estado atual de funcionamento do processo de IDS.

Deteção de inoperacionalidade remota: o sensor e o servidor devem ambos manter um temporizador desde a ultima mensagem recebida com sucesso. Após a expiração do temporizador relativo a um sensor do lado do servidor, este deve considerar o sensor como inoperacional. A expiração do temporizador do lado do sensor deve originar uma alteração para o estado inicial e o envio de uma mensagem de estabelecimento de ligação.

Transições de estados

Os critérios anteriores são formalizados com um conjunto de estados que definem o comportamento do sistema. De seguida descrevem-se os estados do servidor e do sensor, descrevendo a reação do sistema a mensagens recebidas, quais as mensagens a enviar e as transições que devem ocorrer. Na Fig. 3.5 apresenta-se o diagrama de estados correspondente.

Transições de Estados do Servidor:

CLOSED: *socket* fechado, não existe comunicação; abertura do *socket* com a função `bind()`; passa para o estado LISTEN.

LISTEN: aguarda ligações de sensores: SYN; envia: SYN_ACK; passa para o estado FIL_WAIT.

FIL_WAIT: aguarda pedido de ficheiros de configuração: FIL; envia: FIL_ACK; passa para o estado INI_WAIT.

INI_WAIT: aguarda estado inicial do IDS: INI; envia: INI_ACK; passa para o estado ESTABLISHED.

ESTABLISHED: ligação estabelecida; se o utilizador reiniciar o sensor, envia: RST; passa para o estado FIN_WAIT; se não, passa para o estado UPD_WAIT.

UPD_WAIT: aguarda estado atual do sensor: UPD; envia: UPD_ACK; passa para o estado ESTABLISHED.

FIN_WAIT: aguarda terminação do sensor: FIN; passa para o estado CLOSED.

Transições de estados do sensor:

CLOSED: *socket* fechado, não existe comunicação; abertura do socket com a função `connect()`; passa para o estado CONNECT.

CONNECT: inicia ligação com servidor; envia: SYN; passa para o estado SYN_SENT.

SYN_SENT: recebe: SYN_ACK; envia pedido de ficheiros de configuração: FIL; passa para o estado FIL_SENT.

FIL_SENT: recebe: FIL_ACK; inicia o processo do IDS; envia estado inicial do IDS: INI; passa para o estado INI_SENT.

INI_SENT: recebe INI_ACK; passa para o estado ESTABLISHED.

ESTABLISHED: ligação estabelecida; se receber RST, termina o processo do IDS; passa para o estado SHUTDOWN; se não, envia estado atual do sensor: UPD; passa para o estado UPD_SENT;

UPD_SENT: recebe UPD_ACK; passa para o estado ESTABLISHED.

SHUTDOWN: envia terminação do sensor: FIN; passa para o estado CLOSED.

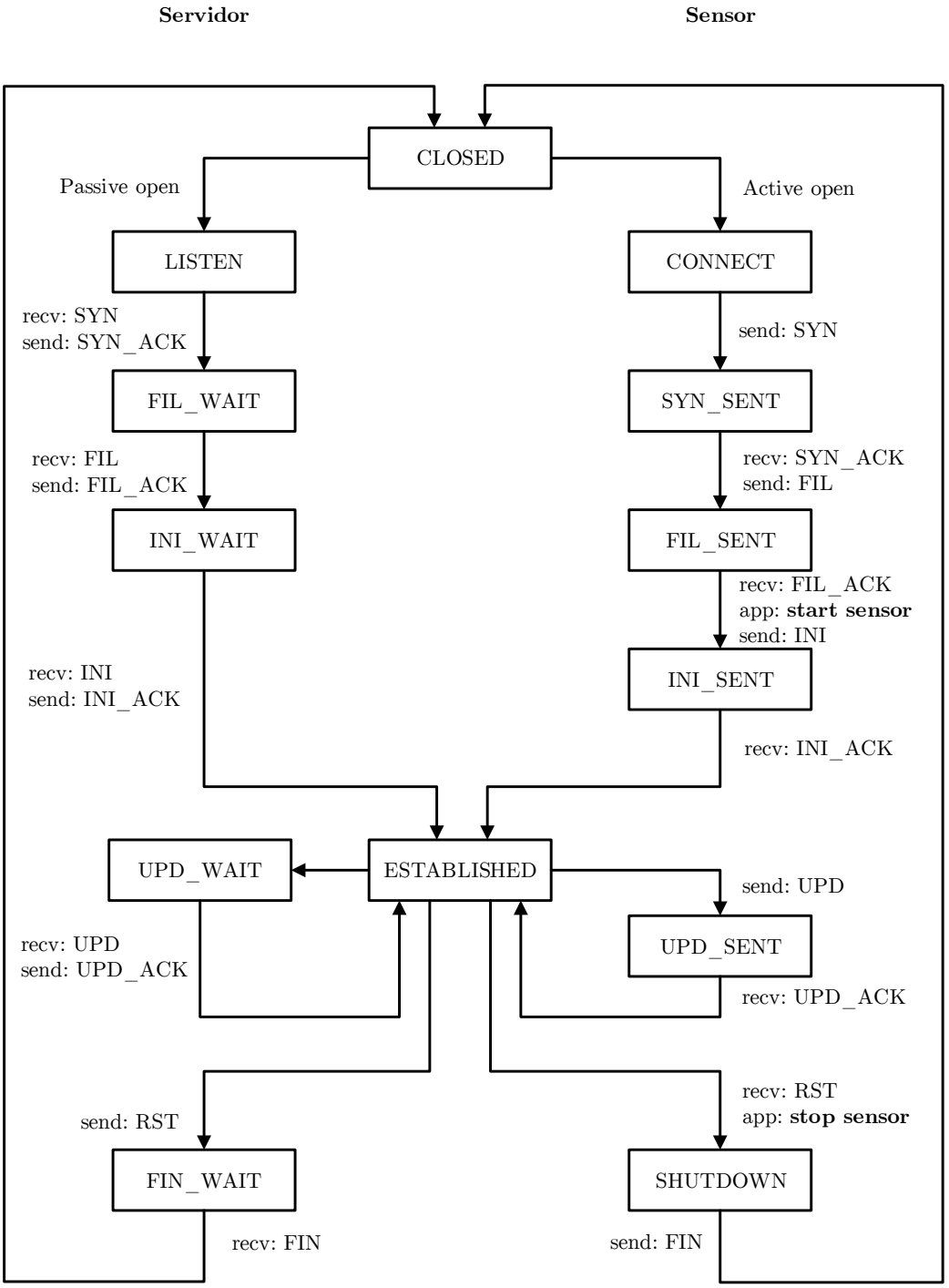

FIGURA 3.5: Protocolo de comunicação.

Capítulo 4

Sensor

A aplicação informática correspondente ao sensor apresenta uma arquitetura modular em que cada uma das suas partes constituintes possui uma função bem definida. No núcleo central do módulo principal encontra-se uma estrutura que gere de forma concorrente as comunicações entre os vários módulos e o exterior.

A evolução das linguagens de programação dinâmicas conduz ao abandono de antigos paradigmas de desenvolvimento de aplicações complexas, como é o caso dos sistemas distribuídos. As atuais ferramentas de desenvolvimento permitem criar rapidamente aplicações complexas no se funcionamento e simultaneamente simples e elegantes na sua estrutura, aproximando o resultado final do projeto idealizado pelo autor.

A linguagem de programação Python reúne as condições sintáticas para concretizar o sistema descrito anteriormente. Possui uma extensa biblioteca padrão e conta com uma comunidade extremamente ativa que apresenta diariamente novos projetos de grande utilidade e qualidade.

Neste capítulo apresenta-se a arquitetura geral do sensor e enumeram-se os vários módulos e as respetivas funcionalidades. Descreve-se o processamento em modo de linha de comandos e os ficheiros de configuração. Introduz-se a comunicação com *sockets* ØMQ e descrevem-se detalhadamente as várias fases de configuração e utilização dos mesmos. Refere-se igualmente o papel dos *sockets* UNIX no controlo do IDS. Descrevem-se todos os mecanismos utilizados na implementação prática do protocolo de comunicação definido anteriormente.

4.1 Arquitetura do Sensor

O sensor é constituído por uma aplicação desenvolvida na linguagem de programação Python e pelo IDS Suricata. A aplicação encontra-se estruturada em quatro módulos da linguagem Python, representados na Fig. 4.1. Cada módulo realiza um conjunto de tarefas específicas, descritas de seguida..

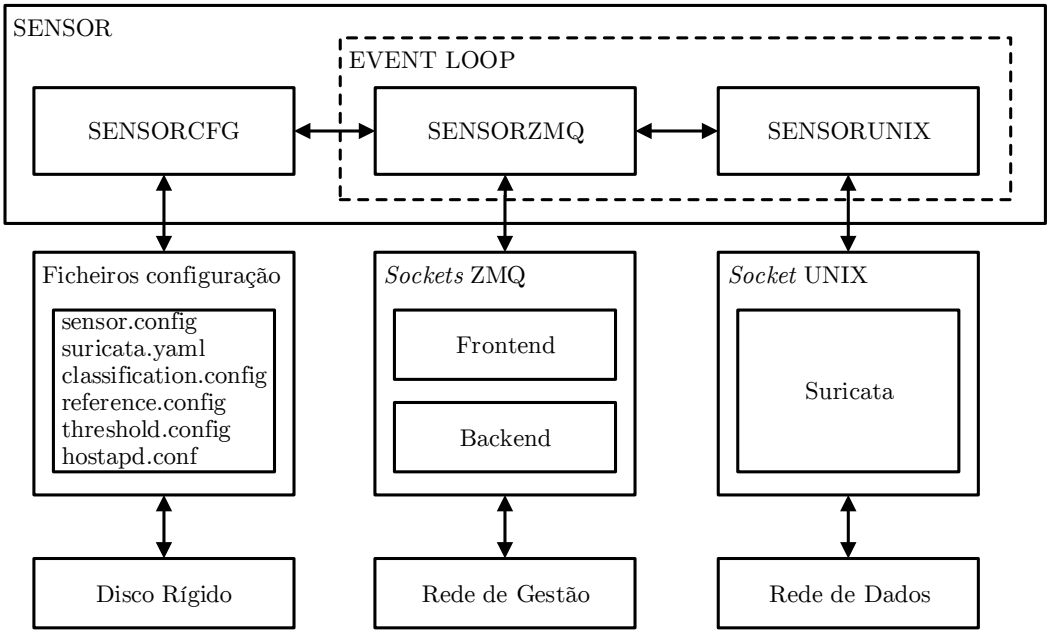

FIGURA 4.1: Arquitetura do sensor.

SENSOR: módulo principal da aplicação. Interpreta os argumentos da linha de comandos e inicia a execução do *event loop*[1].

SENSORCFG: processa os ficheiros de configuração armazenados no disco rígido e mantém uma cópia destes valores em memória.

SENSORZMQ: implementa o protocolo de comunicação do sensor. Utiliza *sockets* ØMQ (`frontend` e `backend`) para comunicar na rede de gestão.

SENSORUNIX: utiliza um *socket* UNIX para controlar o funcionamento do IDS Suricata que captura tráfego da rede de dados.

[1]Designa a estrutura programática que controla o fluxo de execução do programa em função de eventos internos ou de entrada/saída de dados.

Os módulos descritos anteriormente estão organizados em ficheiros com o nome correspondente, da seguinte forma:

- `sensor.py`;
- `sensorcfg.py`;
- `sensorzmq.py`;
- `sensorunix.py`.

Cada ficheiro contém funções, classes e métodos escritos na linguagem de programação Python.

Funcionamento Geral da Aplicação

Ao executar a aplicação do sensor, são realizadas as seguintes tarefas:

1. Início da aplicação:
    - interpretação da linha de comandos;
    - processamento do ficheiro de configuração.
2. Inicialização da comunicação em rede:
    - criação e inicialização do *event loop* e de *sockets* ØMQ;
3. Ligação ao servidor:
    - pedido de ligação;
    - pedido de ficheiros de configuração.
4. Início do Sensor IDS:
    - início do processo do Suricata;
    - envio do estado inicial do IDS.
5. Envio do estado atual do IDS.
6. Reinício do IDS (após receção de mensagem do servidor).

4.2 Início da Aplicação

O programa do sensor é executado a partir da linha de comandos de um sistema Linux. Existe apenas um argumento (`-c` ou `--config`) que especifica o caminho de sistema para o ficheiro de configuração do sensor. De seguida apresenta-se a sintaxe completa do programa:

```
$ ./sensor.py -c [configuration file]
```

A função `main()` do ficheiro `sensor.py` realiza as seguintes tarefas:

Interpretação dos Argumentos da Linha de Comandos

A interpretação dos argumentos é realizada com o módulo `argparse` da biblioteca padrão do Python 3. A variável `args` contém todos os argumentos existentes.

```
32 parser = argparse.ArgumentParser()
33 parser.add_argument('-c', '--config')
34 args = parser.parse_args()
```

Processamento do Ficheiro de Configuração

O ficheiro de configuração é processado pelo módulo `sensorcfg.py`. O argumento `args.config` contém o caminho de sistema para o ficheiro de configuração.

```
38 sensorcfg.parse_config(args.config)
```

Instanciação das Classes `SensorZMQ` e Início do *event loop*

O objeto `sensor_zmq` representa uma instância da classe `SensorZMQ`. O *event loop* é iniciado através do método `event_loop()`.

```
39 sensor_zmq = sensorzmq.SensorZMQ()

43 sensor_zmq.event_loop()
```

4.2.1 Processamento do Ficheiro de Configuração

O ficheiro de configuração é processado no módulo `sensorcfg.py`. Este módulo funciona como um ponto de armazenamento central para as variáveis de configuração e é acessível por todos os módulos que o importam. Este módulo contém apenas as variáveis e a função `parse_config(cfg_path)` que recebe como argumento o caminho de sistema para o ficheiro de configuração, efetuando a leitura e processamento do mesmo e mantendo um registo dos dados em variáveis globais.

Ficheiro de Configuração

O ficheiro de configuração principal do sensor contém toda a informação necessária para a inicialização do mesmo. Este ficheiro é criado e modificado na interface gráfica de gestão do sistema e define os seguintes parâmetros:

`sensor_name:` identificação do sensor; utilizado como identificador dos *sockets* ØMQ.

`server_addr:` endereço IP do servidor.

`server_port:` porto TCP do servidor.

`zmq_port:` porto do *socket* ØMQ.

`unix_suri:` endereço do *socket* UNIX para comunicar com o Suricata.

`suri_cmd:` comando para executar o Suricata.

`pvt_key:` chave privada do sensor.

`pub_key:` chave pública do servidor.

`paths:` caminhos de sistema para os ficheiros de configuração do Suricata.

O ficheiro de configuração está formatado na notação JSON[2]. Exemplo de um ficheiro de configuração de sensor:

```
{
    "sensor_name": "sensor",
    "server_addr": "192.168.0.110",
    "server_port": "12345",
    "zmq_port": "12346",
    "unix_suri": "/var/run/suricata/suricata-command.socket",
    "suri_cmd": "sudo suricata -c /etc/suricata/suricata.yaml -i wlan0",
    "pvt_key": "/home/user1/sensor/private_keys/sensor.key_secret",
    "pub_key": "/home/user1/sensor/public_keys/server.key",
    "paths": {
        "sen_cfg": "/home/user1/sensor/configs/sensor.config",
        "sur_cfg": "/etc/suricata/suricata.yaml",
        "cla_cfg": "/etc/suricata/classification.config",
        "ref_cfg": "/etc/suricata/reference.config",
        "thr_cfg": "/etc/suricata/threshold.config",
        "hos_cfg": "/etc/hostapd/hostapd.conf"
    }
}
```

Leitura do Ficheiro de Configuração

A leitura do ficheiro de configuração efetua-se com o módulo `json` da biblioteca padrão do Python 3. A função `load()` descodifica os dados formatados em notação JSON, que são registados na variável `data`.

```
44 with open(cfg_path, 'r') as f:
45     data = json.load(f)
```

Inicialização de Variáveis de Configuração

Todos os parâmetros existentes no ficheiro de configuração são representados por uma variável com nome idêntico neste módulo. Exemplo da declaração e inicialização da variável `sensor_name` correspondente ao nome do sensor:

```
12 sensor_name = None
```

```
34 global sensor_name
```

```
47 sensor_name = data['sensor_name']
```

[2]JSON - `http://www.json.org`.

4.3 Inicialização da Comunicação em Rede

A comunicação do sensor com o servidor ou outros sensores realiza-se através de *sockets* ØMQ e a comunicação com o processo Suricata realiza-se através de um *socket* UNIX. Todas estas comunicações ocorrem simultaneamente. Este paradigma de programação concorrente é implementado com recurso a funcionalidades dos módulos externos `asyncio` e `aiozmq`.

`asyncio`

O `asyncio`[3] é um novo módulo da biblioteca padrão do Python 3.4. Tem como principal objetivo simplificar o desenvolvimento de aplicações concorrentes apenas com uma *thread*. Implementa um *event loop* que permite a execução simultânea de várias tarefas. Contrariamente a outras bibliotecas de programação concorrente, evita a utilização de funções de *callback* para controlar o fluxo de execução do programa. O `asyncio` implementa o conceito de co-rotina para definir métodos e funções cuja execução pode ser suspensa em determinados pontos, assinalados com a sintaxe `yield from`. As co-rotinas distinguem-se através do decorador `@asyncio.coroutine`.

`aiozmq`

O módulo `asyncio` não suporta nativamente *sockets* ØMQ. Para obter esta funcionalidade utiliza-se o módulo externo `aiozmq`[4]. O desenvolvimento deste módulo foi iniciado em Março de 2014 e o seu estado encontra-se ainda numa fase muito experimental. A aplicação deste módulo consiste na criação de um *event loop* que é de seguida definido como *event loop* do módulo `asyncio`:

```
import asyncio
import aiozmq

loop = aiozmq.new_event_loop()
asyncio.set_event_loop(loop)
```

As comunicações com *sockets* ØMQ e com *sockets* UNIX estão implementadas nos módulos `sensorzmq` e `sensorunix` respetivamente.

[3] Python `asyncio` - `https://docs.python.org/3/library/asyncio.html`
[4] `aiozmq` - `http://aiozmq.readthedocs.org/`

4.3.1 Mensagens ØMQ

Os *sockets* ØMQ suportam a transmissão de mensagens em vários formatos: binário; *string*; pyobj (*pickle*); JSON. Neste sistema utiliza-se o formato JSON com as seguintes estruturas:

Mensagens enviadas do sensor para o servidor: `[ SRC | OP | DATA ]`

Mensagens enviadas do servidor para o sensor: `[ DST | OP | DATA ]`

Onde `SRC` e `DST` representam respetivamente a identificação do sensor que origina a mensagem e a identificação do sensor ao qual se destina a mensagem. `OP` contém a operação a realizar pela mensagem. A secção `DATA` de dimensão e estrutura variáveis contém os dados da mensagem formatados em notação JSON.

4.3.2 Classe `SensorZMQ`

A classe `SensorZMQ` inclui métodos que implementam um modelo de comunicação assíncrona com *sockets* ØMQ. Esta classe possui os seguintes atributos:

`sensor_name:` identificação do sensor;

`loop:` *event loop*;

`frontend:` *socket* ØMQ que comunica em direção ao servidor;

`backend:` *socket* ØMQ que comunica com outros sensores;

`routes:` tabela de encaminhamento com os sensores conhecidos e os próximos "*hops*";

`sensor_unix:` objeto da classe `SensorUnix` para gestão do processo Suricata;

`conn_state:` estado da ligação ao servidor;

`proc_state:` estado do processo Suricata;

`timeout:` temporizador para deteção de inoperacionalidade do servidor.

De seguida descrevem-se os procedimentos realizados pelo método construtor da classe `SensorZMQ` para inicializar os atributos descritos anteriormente.

Criação e Inicialização de *sockets* ØMQ

Antes de poderem ocorrer comunicações com *sockets* ØMQ é necessário criar um novo *event loop*. É criado um contexto `aiozmq` para os *sockets* ØMQ. O *socket* `frontend` do tipo `DEALER` liga diretamente ao servidor ou a uma cadeia de um ou mais sensores. O *socket* `backend` do tipo `ROUTER` permite a ligação de outros sensores.

```
99  self.loop = aiozmq.new_event_loop()
100 ctx = aiozmq.Context(loop=self.loop)
101 asyncio.set_event_loop(self.loop)
102 self.frontend = ctx.socket(zmq.DEALER)
103 self.backend = ctx.socket(zmq.ROUTER)
104 self.frontend.identity = self.sensor_name.encode('ascii')
105 self.backend.identity = self.sensor_name.encode('ascii')
```

Autenticação do Sensor

Após a inicialização dos *sockets* ØMQ, mas antes do estabelecimento da ligação, o sensor efetua a autenticação com o servidor com certificados CurveZMQ, utilizados na cifra de chaves públicas baseada em criptografia de curvas elípticas.

```
108 sensor_public, sensor_secret = auth.load_certificate(sensorcfg.pvt_key)
109 self.frontend.curve_secretkey = sensor_secret
110 self.frontend.curve_publickey = sensor_public
111 server_public, _ = auth.load_certificate(sensorcfg.pub_key)
112 self.frontend.curve_serverkey = server_public
```

Ligação dos *sockets*

Início da ligação do *socket* `frontend` ao servidor (`connect()`) e colocação do *socket* `backend` a aguardar novas ligações de sensores (`bind()`).

```
115 self.frontend.connect('tcp://{0}:{1}'.format(sensorcfg.server_addr,
116                       sensorcfg.server_port))
117 self.backend.bind('tcp://*:{0}'.format(sensorcfg.zmq_port))
```

Inicialização das Variáveis de Estado

Inicialmente a tabela de encaminhamento está vazia. A ligação ao servidor encontra-se no estado `CLOSED` e o processo do Suricata encontra-se no estado `OFF`.

```
121 self.routes = {}
122 self.conn_state = 'CLOSED'
123 self.proc_state = 'OFF'
```

4.3.3 Início do *event loop*

O event loop é iniciado no método `event_loop()`. São adicionadas ao *event loop* as tarefas concorrentes (`Task()`[5]) de monitorização de eventos em ambos os *sockets* `frontend` e `backend`. O *event loop* é colocado em execução até à conclusão destas tarefas (indefinidamente, pois ambas contêm ciclos infinitos).

```
132 tasks = [asyncio.Task(self.handle_frontend()),
133          asyncio.Task(self.handle_backend())]
134
135 self.loop.run_until_complete(asyncio.wait(tasks))
136 self.loop.close()
```

4.3.4 Tratamento de Eventos no *socket* `frontend`

Estas Mensagens são processadas no método `handle_frontend()`. O conteúdo do campo `OP` da mensagem determina o método a ser executado. A variável `ops` contém todas as operações válidas e os métodos correspondentes.

```
152 ops = {'SYN_ACK': self.send_fil,
153        'FIL_ACK': self.start_sensor,
154        'INI_ACK': self.send_upd,
155        'UPD_ACK': self.send_upd,
156        'RST': self.rst_proc}
```

Após a receção da mensagem, os campos `DST` e `OP` são descodificados e atribuídos às variáveis `dest` e `op`.

```
161 rep = yield from self.frontend.recv()
162 print(rep)
163 data = jsonapi.loads(rep)
164 dest = data['DST']
165 op = data['OP']
```

Se o destinatário for o próprio sensor, cancela o temporizador que monitoriza a atividade do servidor e invoca uma tarefa com o método correspondente à operação.

Caso contrário, encapsula a mensagem recebida numa nova mensagem cujo destinatário é o sensor imediatamente mais próximo em direção ao destinatário original (*next hop*). Envia a mensagem através do *socket* `backend`.

```
167 if dest == self.sensor_name:
168     self.timeout.cancel()
169     asyncio.Task(ops[op](data))
170 elif dest in self.routes:
171     ret = [self.routes[dest].encode(), rep]
172     yield from self.backend.send_multipart(ret)
```

[5] `https://docs.python.org/3/library/asyncio-task.html#task`.

4.3.5 Envio de Mensagens no *socket* `frontend`

Devido à inclusão da função do temporizador e a necessidade da sua ativação no envio de mensagens no *socket* `frontend`, para evitar repetição de código, foi criado o método `send_zmq()` que simplesmente envolve o método `send_json()` e ativa o temporizador. Ao fim de trinta segundos de inatividade, a ligação ao servidor é reiniciada automaticamente pelo método `rst_conn()`.

```
208 yield from self.frontend.send_json(msg)
209 self.timeout = self.loop.call_later(30, self.rst_conn)
```

O método `rst_conn()` apenas é realizado se não tiver sido enviado o pedido de ligação inicial. Coloca a ligação no estado inativo. Finalmente é enviada uma mensagem com o `OP: SYN`

```
334 if self.conn_state is not 'SYN_SENT':
335     self.conn_state = 'CLOSED'
336     asyncio.Task(self.send_syn())
```

4.3.6 Tratamento de Eventos no *socket* `backend`

O *socket* `backend` recebe mensagens de outros sensores, que são processadas no método `handle_backend()`. Após receber uma mensagem com formatação JSON, descodifica-a e extrai o campo `SRC`. Verifica a existência do sensor originário da mensagem na tabela de encaminhamento, adicionando-o se for necessário. Finalmente encaminha a mensagem para o *socket* `frontend` em direção ao servidor.

```
187 while True:
188     last = yield from self.backend.recv()
189     msg = yield from self.backend.recv()
190     data = jsonapi.loads(msg)
191     src = data['SRC']
192 
193     if src not in self.routes:
194         self.routes[src] = last.decode()
195 
196     yield from self.frontend.send_json(data)
```

4.4 Ligação ao Servidor

Após a inicialização dos *sockets* ØMQ e da conclusão com sucesso da autenticação com o servidor, o sensor inicia o protocolo de comunicação descrito em 3.4.3.

4.4.1 Pedido de Ligação

Implementado no método `send_syn()`. A ligação ao servidor realiza-se apenas se previamente a ligação estiver inativa. É enviada uma mensagem com o `OP: SYN`. A ligação passa para o estado `SYN_SENT`.

```
218 if self.conn_state is 'CLOSED':
219     msg = {'SRC': self.sensor_name, 'OP': 'SYN'}
220     yield from self.send_zmq(msg)
221     self.conn_state = 'SYN_SENT'
```

4.4.2 Pedido de Ficheiros de Configuração

Implementado no método `send_fil()`. Realiza-se apenas se já tiver sido enviado o pedido de ligação inicial. É construída uma mensagem com `OP: FIL`. São calculadas as hashes dos ficheiros de configuração.

```
232 if self.conn_state is 'SYN_SENT':
233     msg = {'SRC': self.sensor_name, 'OP': 'FIL'}
234
235     hashes = get_hashes()
```

As *hashes* são adicionadas à mensagem e esta é enviada. A ligação passa para o estado `FIL_SENT`.

```
239     msg.update(hashes)
240     yield from self.send_zmq(msg)
241     self.conn_state = 'FIL_SENT'
```

As *hashes* SHA1 são calculadas pela função auxiliar `get_hashes()`, que utiliza a função `sha1()` do módulo `hashlib`[6] da biblioteca padrão do Python 3:

```
34 hashes = {}
35
36 for k, v in sensorcfg.paths.items():
37     with open(v, 'r') as f:
38         data = f.read()
39         hashes[k] = sha1(data.encode()).hexdigest()
40
41 return hashes
```

[6]Python `hashlib` - `https://docs.python.org/3/library/hashlib.html`

4.5 Início do Sensor IDS

Realizado no método `start_sensor()`. O sensor apenas é iniciado se anteriormente tiver sido enviado o pedido de ficheiros de configuração e o IDS estiver inativo. São atualizados os ficheiros de configuração com os dados recebidos do servidor.

```
258 if self.conn_state is 'FIL_SENT' and self.proc_state is 'OFF':
259
260     update_files(data)
```

Os ficheiros de configuração são atualizados pela função auxiliar `update_files()`:

```
51 exclude = ['OP', 'DST']
52
53 for k, v in data.items():
54     if (k not in exclude) and (v is not None):
55         with open(sensorcfg.paths[k], 'w') as f:
56             f.write(v)
```

Após a atualização dos ficheiros de configuração é criado o objeto `sensor_unix` e executado o método que inicia o processo do Suricata. O estado do processo passa para `ON`. É enviada uma mensagem com o estado inicial do IDS.

```
264     self.sensor_unix = sensorunix.SensorUnix()
265     yield from self.sensor_unix.start_proc()
266     self.proc_state = 'ON'
267
268 yield from self.send_ini()
```

4.5.1 Mensagens UNIX

A interação com o *socket* UNIX do Suricata está definida no protocolo de comunicação criado pela OISF[7]. As mensagens são formatadas na notação JSON.

O cliente inicia a ligação com a mensagem: {`"version":"0.1"`}. O servidor responde positivamente: {`"return":"OK"`}.

A partir deste momento, o cliente pode enviar comandos, por exemplo:

Envia: {`"command":"version"`}.

Recebe: {`"message":{"version":"2.0 RELEASE"}`}.

A lista de comandos válidos é a seguinte: `shutdown`, `command-list`, `help`, `version`, `uptime`, `running-mode`, `capture-mode`, `conf-get`, `dump-counters`, `iface-stat`, `iface-list`, `quit`.

[7] `https://redmine.openinfosecfoundation.org/projects/suricata/wiki/Interacting_via_Unix_Socket`

4.5.2 Classe `SensorUnix`

A classe `SensorUnix` é responsável pelo início e paragem do processo Suricata. Efetua a leitura e escrita de mensagens no *socket* UNIX. possui os seguintes atributos:

`loop:` *event loop*;

`reader:` *stream* de leitura no *socket* UNIX;

`writer:` *stream* de escrita no *socket* UNIX;

`suri_proc:` processo do IDS Suricata.

Inicialização do *event loop*

Não é necessária a criação de um novo *event loop* para o *socket* UNIX. Basta obter a instância global e os eventos do *socket* UNIX são monitorizados juntamente com os *sockets* ØMQ.

```
40 self.loop = asyncio.get_event_loop()
```

4.5.3 Início do Processo do Suricata

O módulo `asyncio` permite a execução de programas externos através da criação de subprocessos[8]. Este módulo disponibiliza igualmente um conjunto de funções para controlar a execução do processo e comunicar com o mesmo através de *pipes* ou o envio de sinais de sistema. O método `start_proc()` lança o processo do suricata assincronamente numa nova *shell*. O comando utilizado com o caminho de sistema do ficheiro executável é obtido da variável `suri_cmd` do módulo `sensorcfg`.

```
52 self.suri_proc = yield from asyncio.create_subprocess_shell(sensorcfg.suri_cmd)
```

[8] `asyncio-subprocess - https://docs.python.org/3/library/asyncio-subprocess.html`.

4.5.4 Leitura de Dados no *socket* UNIX

A leitura de dados é realizada em blocos de `SIZE` *bytes*. As mensagens do servidor podem ser enviadas em várias partes. São realizadas cinco tentativas de leitura, devido a eventuais situações de leitura incompleta. O método `unix_recv()` é uma versão assíncrona do modelo de referência disponibilizado pela OISF[9].

```
87  data = ''
88
89  for _ in range(5):
90      buf = yield from self.reader.read(SIZE)
91      data += buf.decode()
92      try:
93          ret = json.loads(data)
94      except ValueError:
95          yield from asyncio.sleep(0.1)
96      else:
97          break
98
99  return ret
```

4.5.5 Envio de Comandos no *socket* UNIX

O método `write_recv()` recebe uma lista de comandos que envia sequencialmente para o servidor. As respostas de cada comando são adicionadas à variável `ret`, devolvida no final.

```
119 ret = {}
120
121 for comm in comm_list:
122     self.reader, self.writer = yield from asyncio.open_unix_connection(
123                                             sensorcfg.unix_suri, loop=self.loop)
124     self.writer.write(json.dumps({'version': VERSION}).encode())
125     rep = yield from self.unix_recv()
126
127     if rep['return'] == 'OK':
128         self.writer.write(json.dumps({'command': comm}).encode())
129
130     rep = yield from self.unix_recv()
131     ret[comm] = rep['message']
132     self.writer.close()
133
134 return ret
```

[9] https://redmine.openinfosecfoundation.org/projects/suricata/repository/changes/scripts/suricatasc/src/suricatasc.py

4.5.6 Envio do Estado Inicial do IDS

Implementado no método `send_ini()`. Esta mensagem apenas é enviada se já tiver sido enviado o pedido de ficheiros de configuração e o IDS estiver atualmente ativo. Obtém os parâmetros iniciais do processo do Suricata. É enviada uma mensagem com o `OP: INI`. A ligação passa para o estado `INI_SENT`.

```
281 if self.conn_state is 'FIL_SENT' and self.proc_state is 'ON':
282     msg = {'SRC': self.sensor_name, 'OP': 'INI'}
283     ini = yield from self.sensor_unix.get_ini()
284     msg.update(ini)
285     yield from self.send_zmq(msg)
286     self.conn_state = 'INI_SENT'
```

Os parâmetros de funcionamento do IDS que se mantêm inalteráveis durante a sessão são obtidos com a função `get_ini()`:

```
146 comm_list = ['version', 'running-mode', 'capture-mode']
147 ret = yield from self.write_recv(comm_list)
148 return ret
```

4.6 Envio do Estado Atual do IDS

Realizado pelo método `send_upd()`. Enviado apenas se o processo do IDS estiver ativo. A mensagem é enviada a cada cinco segundos. Obtém os parâmetros atuais do processo do Suricata. É enviada uma mensagem com o `OP: UPD`. A ligação passa para o estado `UPD_SENT`.

```
299 yield from asyncio.sleep(5)
300
301 if self.proc_state is 'ON':
302     msg = {'SRC': self.sensor_name, 'OP': 'UPD'}
303     upd = yield from self.sensor_unix.get_upd()
304     msg.update(upd)
305     yield from self.send_zmq(msg)
306     self.conn_state = 'UPD_SENT'
```

Os parâmetros de funcionamento do IDS que se variam ao longo da sessão são obtidos com a função `get_upd()`:

```
160 comm_list = ['uptime', 'dump-counters']
161 ret = yield from self.write_recv(comm_list)
162 return ret
```

4.7 Reinício do IDS

Esta ação é iniciada após a receção de uma mensagem do servidor para reinicio do sensor. Implementado no método `rst_proc()`. Realizado apenas se o processo estiver ativo. Coloca o processo no estado `SHUTDOWN`. Executa o método que desativa o processo do Suricata e aguarda a sua finalização. É enviada uma mensagem com o `OP: FIN`. Coloca o processo e a ligação no estado inativo. Finalmente é enviada uma mensagem com o `OP: SYN` para indicar ao servidor o novo estado de prontidão funcional.

```
318 if self.proc_state is 'ON':
319     self.proc_state = 'SHUTDOWN'
320     yield from self.sensor_unix.stop_proc()
321     msg = {'SRC': self.sensor_name, 'OP': 'FIN'}
322     yield from self.send_zmq(msg)
323     self.proc_state = 'OFF'
324     self.conn_state = 'CLOSED'
325     asyncio.Task(self.send_syn())
```

A terminação do processo é efetuada pelo método `stop_proc()`. É enviado o comando `shutdown` para o socket UNIX do Suricata. É enviado o sinal de sistema `SIGKILL` para o processo e aguardada a terminação do mesmo.

```
71 yield from self.write_recv(['shutdown'])
72 self.suri_proc.kill()
73 yield from self.suri_proc.wait()
```

Capítulo 5

Servidor

A parte do sistema designada por servidor é uma solução para o problema da gestão das configurações de uma rede de sensores distribuídos. Embora a deteção de intrusões esteja dispersa por toda a organização, o controlo e monitorização do seu funcionamento está concentrado num ponto central.

O servidor tem duas interfaces de comunicação com tecnologias e protocolos distintos: uma interface que comunica com os sensores e outra que efetua a interação com o utilizador, o que lhe permite cumprir os requisitos previamente enunciados. O servidor inclui uma interface gráfica para acesso através de *browsers* de Internet.

A persistência dos dados entre sessões é obtida com o recurso a uma base de dados NoSQL. Este tipo de bases de dados distingue-se das tradicionais bases de dados SQL ao permitir o armazenamento e manipulação de dados dinâmicos com estruturas variáveis. Esta característica facilita a transmissão das mensagens entre os sensores e o servidor que é efetuada sempre com o formato JSON.

A aplicação informática que implementa o servidor possui uma estrutura modular semelhante à utilizada no sensor. Ambas as aplicações têm em comum o desenvolvimento realizado na linguagem de programação Python e a utilização de *sockets* ØMQ. Adicionalmente, esta aplicação lança um servidor web baseado na biblioteca Tornado, suportado por uma base de dados MongoDB.

Neste capítulo apresenta-se a arquitetura geral do sensor e enumeram-se os vários módulos e as respetivas funcionalidades. O processamento em modo de linha de comandos e os ficheiros de configuração é exibido e comentado. Descreve-se a base de dados MongoDB e o seu papel no funcionamento do servidor. A implementação da comunicação com *sockets* ØMQ e do servidor web é descrita em detalhe.

5.1 Arquitetura do Servidor

O servidor é constituído por uma aplicação desenvolvida na linguagem de programação Python, pelo servidor HTTP Tornado e pela base de dados MongoDB. A aplicação encontra-se estruturada em cinco módulos da linguagem Python, representados na Fig. 5.1 e descritos imediatamente de seguida..

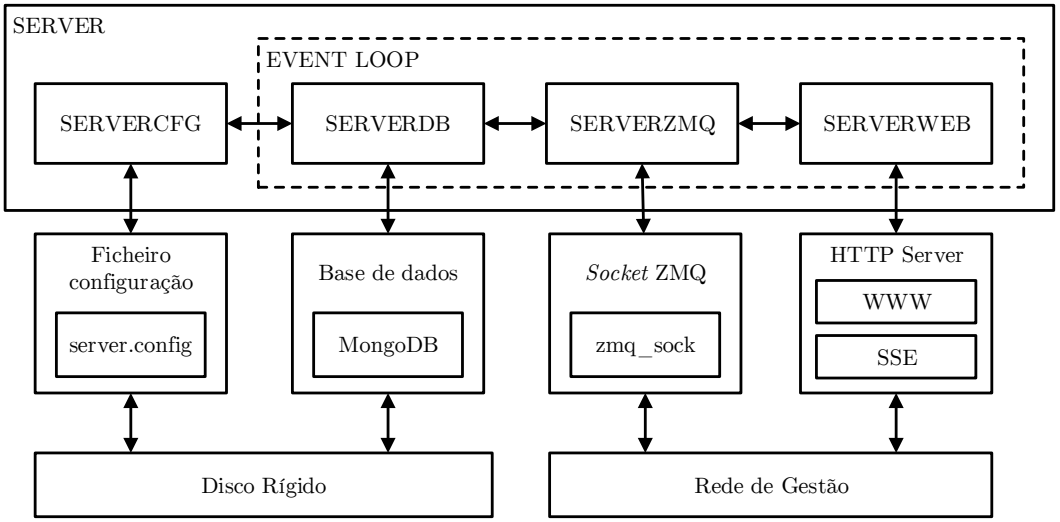

FIGURA 5.1: Arquitetura do servidor.

SERVER: módulo principal da aplicação. Interpreta os argumentos da linha de comandos e inicia a execução do *event loop*.

SERVERCFG: processa o ficheiro de configuração armazenado no disco rígido e mantém uma cópia destes valores em memória.

SERVERDB: suporta o acesso à base de dados MongoDB com os dados dos sensores e mantém uma variável global com o estado do sistema.

SERVERZMQ: implementa o protocolo de comunicação do servidor. Utiliza um *socket* ØMQ (`zmq_sock`) para comunicar na rede de gestão.

SERVERWEB: implementa um servidor HTTP `tornado`. O código correspondente ao sítio web de controlo do sistema e um serviço SSE[1] para comunicação com o *browser* encontra-se neste módulo.

[1]Server Sent Events - `http://www.w3.org/TR/eventsource/`.

Os módulos descritos anteriormente estão organizados em ficheiros com o nome correspondente, da seguinte forma:

- `server.py`;
- `servercfg.py`;
- `serverdb.py`;
- `serverzmq.py`;
- `serverweb.py`.

Cada ficheiro contém funções, classes e métodos escritos na linguagem de programação Python.

Funcionamento Geral da Aplicação

Ao executar a aplicação do sensor, são realizadas as seguintes tarefas:

1. Início da aplicação:
    - interpretação da linha de comandos;
    - processamento do ficheiro de configuração.
2. Inicialização da base de dados:
    - criação da ligação à base de dados e inicialização da variável de estado;
3. Inicialização da comunicação em rede:
    - criação e inicialização do *event loop* e do *socket* ØMQ;
4. Ligação aos sensores:
    - receção de pedidos dos sensores;
    - envio de respostas aos pedidos.
5. Início do servidor web:
    - interação com o utilizador;
    - envio do estado atual do IDS através do serviço SSE;
    - atualização de sensores com ficheiros de configuração modificados.

5.2 Início da Aplicação

O programa do servidor é executado a partir da linha de comandos de um sistema Linux. Existe apenas um argumento (`-c` ou `--config`) que especifica o caminho de sistema para o ficheiro de configuração do servidor. De seguida apresenta-se a sintaxe completa do programa:

```
$ ./server.py -c [configuration file]
```

A função `main()` do ficheiro `server.py` realiza as seguintes tarefas:

Interpretação dos Argumentos da Linha de Comandos

A interpretação dos argumentos é realizada com o módulo `argparse` da biblioteca padrão do Python 3. A variável `args` contém todos os argumentos existentes.

```
44 parser = argparse.ArgumentParser()
45 parser.add_argument('-c', '--config')
46 args = parser.parse_args()
```

Processamento do Ficheiro de Configuração

O ficheiro de configuração é processado pelo módulo `servercfg.py`. O argumento `args.config` contém o caminho de sistema para o ficheiro de configuração.

```
50 servercfg.parse_config(args.config)
```

Instanciação das Classes `ServerZMQ` e `HTTPServer`

O `ioloop` do `tornado` é definido como a instância de *event loop* principal. São criados os objetos `server_zmq` e `http_server`. A função `init_state()` do módulo `serverdb` é executada na primeira iteração do *event loop*, que é iniciado na linha seguinte.

```
55 loop = ioloop.IOLoop.instance()
56 server_zmq = serverzmq.ServerZMQ()
57 http_server = httpserver.HTTPServer(serverweb.ServerWeb(server_zmq))
58 http_server.listen(options.port)
59 loop.add_callback(serverdb.init_state)
60 loop.start()
```

5.2.1 Processamento do Ficheiro de Configuração

O ficheiro de configuração é processado no módulo `servercfg.py`. Este módulo funciona como um ponto de armazenamento central para as variáveis de configuração e é acessível por todos os módulos que o importam. Este módulo contém apenas as variáveis e a função `parse_config(cfg_path)` que recebe como argumento o caminho de sistema para o ficheiro de configuração, efetuando a leitura e processamento do mesmo e mantendo um registo dos dados em variáveis globais.

Ficheiro de Configuração

O ficheiro de configuração principal do servidor contém toda a informação necessária para a inicialização do mesmo. Este ficheiro é criado e modificado na interface gráfica de controlo e define os seguintes parâmetros:

`web_port:` porto do servidor web;

`zmq_port:` porto do *socket* ØMQ;

`pvt_key:` chave privada do servidor;

`paths:` caminhos de sistema para os ficheiros de configuração predefinidos para o sensor;

O ficheiro de configuração está formatado na notação JSON[2]. Exemplo de um ficheiro de configuração de servidor:

```
{
    "pvt_key": "/home/user1/workspace/p01/server/private_keys/server.key_secret",
    "web_port": "8888",
    "zmq_port": "12345",
    "paths": {
        "cla_cfg": "/home/user1/workspace/p01/server/configs/classification.config",
        "hos_cfg": "/home/user1/workspace/p01/server/configs/hostapd.conf",
        "ref_cfg": "/home/user1/workspace/p01/server/configs/reference.config",
        "sen_cfg": "/home/user1/workspace/p01/server/configs/sensor.config",
        "sur_cfg": "/home/user1/workspace/p01/server/configs/suricata.yaml",
        "thr_cfg": "/home/user1/workspace/p01/server/configs/threshold.config"
    }
}
```

[2]JSON - `http://www.json.org`.

Leitura do Ficheiro de Configuração

A leitura do ficheiro de configuração efetua-se com o módulo `json` da biblioteca padrão do Python 3. A função `load()` descodifica os dados formatados em notação JSON, que são registados na variável `opts`.

```
37 with open(cfg_path, 'r') as f:
38     opts = json.load(f)
```

Inicialização de Variáveis de Configuração

Todos os parâmetros existentes no ficheiro de configuração são representados por uma variável com nome idêntico neste módulo. Exemplo da declaração e inicialização da variável `web_port` correspondente ao porto do servidor web:

```
13 web_port = None
```

```
30 global web_port
```

```
40 web_port = opts['web_port']
```

5.3 Inicialização da Base de Dados

O servidor utiliza uma base de dados MongoDB[3]. Este tipo de base dados NoSQL não requer a formalização de um *schema*[4] estático. Os dados armazenados são agrupados em coleções de objetos dinâmicos que podem variar em dimensão e estrutura, sendo inseridos e manipulados diretamente em notação JSON. A equivalência direta entre os tipos de dados do padrão JSON e Python permite uma simplicidade no desenvolvimento do acesso à base de dados que seria impossível com bases de dados SQL tradicionais[5].

Na base de dados é armazenada informação que deve persistir entre sessões do sistema, por exemplo: registo de utilizadores; sensores e configurações. As funções que realizam operações de leitura/escrita na base de dados são co-rotinas integradas no `event loop` do tornado e são executadas assincronamente.

O módulo contém uma variável que guarda informações relativas ao estado de funcionamento do sistema num determinado instante e que não necessitam de persistir entre sessões, por exemplo: tempo de funcionamento e contadores estatísticos.

[3] MongoDB - `http://www.mongodb.org`.
[4] Designa a organização lógica de objetos numa base de dados.
[5] MongoDB *Data Modeling* - `http://docs.mongodb.org/manual/data-modeling`.

Acesso à Base de Dados

O acesso à base de dados é realiza-se através de uma instância da classe `MotorClient` do módulo `motor`[6]. Este módulo é um adaptador assíncrono da base de dados MongoDB, para utilização com o `tornado`. O objeto `MotorClient` nunca bloqueia o *event loop* do `tornado` ao efetuar operações de leitura/escrita na base de dados.

Coleções de objetos

A base de dados contém as seguintes coleções de objetos:

`users:` utilizadores com permissões para administrar o sistema;

`sensors:` sensores registados no sistema.

Inicialização de Variáveis

Declaração e inicialização da variável de estado do sistema.

```
20 state = {}
```

A variável `db` contém a referência para uma instância do cliente da base dados do módulo `motor`.

```
23 db = motor.MotorClient().test
```

5.4 Inicialização da Comunicação em Rede

A comunicação do servidor com os sensores realiza-se através de um *socket* ØMQ e a comunicação com o utilizador realiza-se através do servidor web. Todas estas comunicações ocorrem simultaneamente. Este paradigma de programação concorrente é implementado com recurso a funcionalidades dos módulos externos `tornado` e `pyzmq`.

[6]Motor - `http://motor.readthedocs.org`.

`tornado`

O `tornado`[7] é uma biblioteca de programação web desenvolvida na linguagem Python que contém um servidor HTTP de alto desempenho e um conjunto de ferramentas que permitem desenvolver sítios web complexos. Utiliza um modelo de comunicação assíncrona compatível com o módulo `asyncio`, com o qual partilha várias características tecnológicas e sintáticas, tais como a utilização de um *event loop* e do decorador `@gen.coroutine` para indicar co-rotinas.

`pyzmq`

O módulo `pyzmq`[8] contém a implementação de *sockets* ØMQ na linguagem de programação Python. Este módulo suporta diretamente a integração no *event loop* do `tornado`. Permite igualmente a integração de um servidor de autenticação CurveZMQ no *event loop* do `tornado`.

As comunicações com *sockets* ØMQ e o servidor web estão implementadas nos módulos `serverzmq` e `serverweb` respetivamente.

O servidor partilha com o sensor o formato de mensagens ØMQ definido em 4.3.1.

5.4.1 Classe `ServerZMQ`

A classe `ServerZMQ` inclui métodos que implementam um modelo de comunicação assíncrona com *sockets* ØMQ. Esta classe possui os seguintes atributos:

`loop:` *event loop*;

`stream:` descritores de ficheiro para leitura/escrita no *socket* ØMQ;

`routes:` tabela de encaminhamento com os sensores conhecidos e os próximos "*hops*";

`timeout:` temporizador para deteção de inoperacionalidade dos sensores.

De seguida descrevem-se os procedimentos realizados pelo método construtor da classe `ServerZMQ` para inicializar os atributos descritos anteriormente.

[7] `tornado - http://www.tornadoweb.org`.
[8] `pyzmq - http://zeromq.github.io/pyzmq`.

Criação e Inicialização do *socket* ØMQ

O *event loop* utilizado na comunicação com *sockets* ØMQ é a mesma instância global do `tornado ioloop` iniciado no módulo principal e partilhado com as instâncias do servidor HTTP e da base de dados. A aplicação do servidor possui apenas o socket `zmq_sock` do tipo `ROUTER` que permite a ligação de múltiplos sensores.

```
87 self.loop = ioloop.IOLoop.instance()
88 ctx = zmq.Context()

93 zmq_sock = ctx.socket(zmq.ROUTER)
94 zmq_sock.identity = 'server'.encode('ascii')
```

Servidor de Autenticação CurveZMQ

O servidor de autenticação é implementado através de uma instância da classe `IOLoopAuthenticator()` integrada no `IOLoop` do `tornado`. Utiliza-se uma configuração básica que não restringe gamas de IPs nem domínios específicos.

```
89 auth_loop = IOLoopAuthenticator()
90 auth_loop.start()
91 auth_loop.allow('*')
92 auth_loop.configure_curve(domain='*', location=auth.CURVE_ALLOW_ANY)

95 server_public, server_secret = auth.load_certificate(servercfg.pvt_key)
96 zmq_sock.curve_secretkey = server_secret
97 zmq_sock.curve_publickey = server_public
98 zmq_sock.curve_server = True
```

Ligação do *socket*

O *socket* `zmq_sock` é colocado à escuta de novas ligações de sensores. É criada uma instância da classe `ZMQStream` associada ao *socket* `zmq_sock` e ao *event loop*. Todas as mensagens recebidas são processadas pelo método `handle_recv()`.

```
99  zmq_sock.bind('tcp://*:{0}'.format(servercfg.zmq_port))
100 self.stream = ZMQStream(zmq_sock, self.loop)
101 self.stream.on_recv(self.handle_recv)
```

Inicialização das Variáveis de Estado

Inicialmente a tabela de encaminhamento e a lista de temporizadores para cada sensor encontram-se vazias.

```
102 self.routes = {}
103 self.timeouts = {}
```

5.4.2 Tratamento de Eventos no *socket* `zmq_sock`

As mensagens recebidas no *socket* `zmq_sock` são descodificadas e interpretadas. O conteúdo do campo `OP` da mensagem determina o método a ser executado. A variável `ops` contém todas as operações válidas e os métodos correspondentes.

```
118 ops = {'SYN': self.send_syn_ack,
119        'FIL': self.send_fil_ack,
120        'INI': self.send_ini_ack,
121        'UPD': self.send_upd_ack,
122        'FIN': self.rst_state}
```

Após a receção da mensagem, os campos `SRC` e `OP` são descodificados e atribuídos às variáveis `source` e `op`.

```
124 last = msg[0].decode()
125 data = jsonapi.loads(msg[1].decode())
126 source = data['SRC']
127 op = data['OP']
```

Se o sensor que originou a mensagem não estiver ainda na tabela de encaminhamento, é adicionado a esta. O último sensor que encaminhou a mensagem até ao servidor é incluído como *last hop*.

```
130 if source not in self.routes:
131     self.routes[source] = last
```

Se o sensor que originou a mensagem estiver na lista de temporizadores, cancela a função `timeout()` no *event loop* correspondente a este sensor e remove-o da lista.

```
133 if source in self.timeouts:
134     self.loop.remove_timeout(self.timeouts[source])
135     del self.timeouts[source]
```

Caso contrário, adiciona um novo temporizador para este sensor. O temporizador está associado ao método `sensor_timeout()`, que deve ser executado caso não seja recebida nenhuma mensagem deste sensor durante trinta segundos.

```
137 timeout_cb = functools.partial(self.sensor_timeout, source)
138 self.timeouts[source] = self.loop.add_timeout(
139                                         datetime.timedelta(seconds=30), timeout_cb)
```

O método `sensor_timeout()` coloca o sensor no estado `FIN_WAIT` e executa o método que reinicializa o sensor na base de dados.

```
268 serverdb.state[source]['state'] = 'FIN_WAIT'
269 self.rst_state(source)
```

Finalmente, executa o método correspondente à mensagem recebida.

```
140 yield ops[op](source, last, data)
```

5.4.3 Aceitar Pedido de Ligação

Ao receber uma mensagem de pedido de ligação de um sensor, executa o método `send_syn_ack()`. O servidor reinicia o estado do sensor com o método `reset_state()`. É enviada uma mensagem com `OP: SYN_ACK` e o estado relativo ao sensor passa para `FIL_WAIT`.

```
154 serverdb.reset_state(source)
155 rep = {'DST': source, 'OP': 'SYN_ACK'}
156 ret = [last.encode(), jsonapi.dumps(rep)]
157 self.stream.send_multipart(ret)
158 serverdb.state[source]['state'] = 'FIL_WAIT'
```

O método `reset_state()` reinicializa o estado de um sensor na variável de estado global.

```
144 state.update({name: {'state': 'CLOSED', 'version': '', 'uptime': 0,
145                      'running-mode': '', 'capture-mode': '', 'stats': '',
146                      'counters': ''}})
```

5.4.4 Responder a Pedido de Ficheiros de Configuração

O método `send_fil()` é executado após receber uma mensagem de pedido de ficheiro de configuração de um sensor. O pedido apenas é processado se o estado relativo ao sensor for `FIL_WAIT`. Obtém o objeto da base de dados correspondente ao sensor com o método `get_sensor()`. Se este sensor existir na base dados, compara os valores das *hashes* dos seus ficheiros de configuração com os valores recebidos do sensor. Envia a mensagem com os ficheiros de configuração cujas são *hashes* diferentes e `OP: FIL_ACK`. O estado relativo ao sensor passa para `INI_WAIT`.

```
177 if serverdb.state[source]['state'] is 'FIL_WAIT':
178     sensor = yield serverdb.get_sensor(source)
179
180     if sensor:
181         rep = {'DST': source, 'OP': 'FIL_ACK'}
182         hashes = compare_hashes(data, sensor)
183         rep.update(hashes)
184         ret = [last.encode(), jsonapi.dumps(rep)]
185         self.stream.send_multipart(ret)
186     serverdb.state[source]['state'] = 'INI_WAIT'
```

O método `get_sensor()` devolve o objeto da coleção `sensors` da base de dados, cujo atributo `name` corresponde ao recebido por argumento.

```
70 sensor = yield db.sensors.find_one({'name': name})
```

Os valores das *hashes* dos ficheiros de configuração são comparados pela função `compare_hashes()`. esta função devolve apenas os ficheiros de configuração cujo valor da *hash* SHA1 não coincide com o valor recebido do sensor, indicando que a versão existente localmente no sensor está corrompida ou desatualizada e deve ser substituída pela versão existente na base de dados do servidor.

```
43 ret = {}
44 exclude = ['OP', 'SRC']
45
46 for k, v in data.items():
47     if (k not in exclude):
48         if v != hashlib.sha1(sensor[k].encode()).hexdigest():
49             ret[k] = sensor[k]
50         else:
51             ret[k] = None
52
53 return ret
```

5.4.5 Responder a Mensagem de Inicialização do IDS

O método `send_ini_ack()` é executado após receber a mensagem com o estado inicial do IDS de um sensor. A mensagem apenas é processada se o estado relativo ao sensor for `INI_WAIT`. O estado do sensor na base de dados é inicializado com o método `set_state()`. Envia uma mensagem com `OP: FIL_ACK`. O estado relativo ao sensor passa para `UPD_WAIT`.

```
202 if serverdb.state[source]['state'] is 'INI_WAIT':
203     serverdb.set_state(source, data)
204     rep = {'DST': source, 'OP': 'INI_ACK'}
205     ret = [last.encode(), jsonapi.dumps(rep)]
206     self.stream.send_multipart(ret)
207     serverdb.state[source]['state'] = 'UPD_WAIT'
```

O método `set_state()` altera o estado de sensor na variável de estado global com os dados de funcionamento inicial recebidos do sensor e que permanecem inalterados durante a sessão.

```
184 state[name].update({'version': data['version'],
185                     'running-mode': data['running-mode'],
186                     'capture-mode': data['capture-mode']})
```

5.4.6 Receber Mensagem de Atualização do Estado do IDS

O método `send_ini_ack()` é executado após receber a mensagem com o estado atual do IDS de um sensor. A mensagem apenas é processada se o estado relativo ao sensor for `UPD_WAIT`. O estado do sensor na base de dados é atualizado com o método `update_state()`. Envia uma mensagem com `OP: UPD_ACK`. Não existe alteração no estado relativo ao sensor.

```
223 if serverdb.state[source]['state'] is 'UPD_WAIT':
224     serverdb.update_state(source, data)
225     rep = {'DST': data['SRC'], 'OP': 'UPD_ACK'}
226     ret = [last.encode(), jsonapi.dumps(rep)]
227     self.stream.send_multipart(ret)
```

O método `update_state()` atualiza o estado do sensor na variável de estado global com os valores de tempo de funcionamento ativo e dos contadores estatísticos do sensor que variam durante a sessão.

```
199 state[name].update({'uptime': data['uptime'], 'counters': data['dump-counters']})
```

5.4.7 Enviar Mensagem de Reinicialização do IDS

Esta mensagem é originada pelo utilizador através da interface gráfica de gestão, após a alteração dos ficheiros de configuração do sensor. É realizada pelo método `send_rst()` que envia uma mensagem com `OP: RST`. O estado relativo ao sensor passa para `FIN_WAIT`.

```
240 last = self.routes[source]
241 rep = {'DST': source, 'OP': 'RST'}
242 ret = [last.encode(), jsonapi.dumps(rep)]
243 self.stream.send_multipart(ret)
244 serverdb.state[source]['state'] = 'FIN_WAIT'
```

5.4.8 Desativar o Sensor na Base de Dados

Este método é executado após receber a mensagem de reinicialização do sensor. Implementado no método `rst_state()`. A entrada na tabela de encaminhamento relativa ao sensor é removida. É executado o método `shutdown_state()` que desativa o estado do sensor.

```
256 if serverdb.state[source]['state'] is 'FIN_WAIT':
257     del self.routes[source]
258     serverdb.shutdown_state(source)
```

O método `shutdown_state()` altera o estado de um sensor para `SHUTDOWN` na variável de estado global.

```
158 state.update({name: {'state': 'SHUTDOWN', 'version': '', 'uptime': 0,
159                      'running-mode': '', 'capture-mode': '', 'stats': '',
160                      'counters': ''}})
```

5.5 Servidor HTTP

O servidor HTTP proporciona o meio de interação entre o utilizador que administra o sistema e as várias componentes que o constituem. O sítio web é desenvolvido com o módulo `tornado.web` que contém um sistema de *templates* que permite integrar código Python com html e *javascript*. Os ficheiros html encontram-se no apêndice 3. O *design* gráfico foi desenvolvido com a plataforma web *bootstrap*[9], proporcionando uma experiência de utilização dinâmica e com elementos que se adaptam automaticamente a dispositivos com ecrãs de diferentes resoluções.

Mapa do Sítio Web

O sítio web possui um sistema de autenticação de utilizadores e permite realizar as seguintes funcionalidades: adicionar/remover sensores; editar as configurações do servidor e dos sensores e monitorizar o funcionamento dos sensores em tempo real. Apresenta-se a estrutura do sítio web na Fig. 5.2.

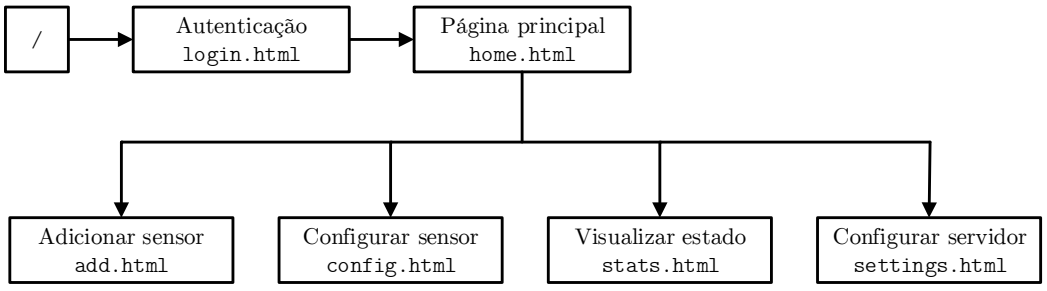

FIGURA 5.2: Arquitetura do servidor.

[9] *Bootstrap* - `http://getbootstrap.com`.

5.5.1 Classe `ServerWeb`

Classe principal do servidor web. Possui um método construtor onde são definidas as opções de encaminhamento para as várias páginas e as classes correspondentes. Os métodos que requerem autenticação do utilizador possuem o decorador `@web.authenticated`. As classes que realizam leitura/escrita assíncrona possuem o decorador `@gen.coroutine`.

```
356 handlers = [(r'/', HomeHandler),
357             (r'/login', LoginHandler),
358             (r'/logout', LogoutHandler),
359             (r'/add', AddHandler),
360             (r'/remove', RemoveHandler),
361             (r'/config', ConfigHandler),
362             (r'/stats', StatsHandler),
363             (r'/reset', ResetHandler),
364             (r'/settings', SettingsHandler),
365             (r'/sse', SSEHandler)]
```

No final, a aplicação web é lançada executando explicitamente o método construtor.

```
378 web.Application.__init__(self, handlers, **settings)
```

Autenticação do Utilizador

O acesso a qualquer das funcionalidades do sítio web requer a prévia autenticação do utilizador. A autenticação no sistema é implementada na classe `LoginHandler`. O processo de autenticação é realizado através da introdução de utilizador/palavra-passe na página `login.html`.

A verificação de permissões é efetuada pelo método `check_permission()` que verifica simplesmente se o par `username/password` existe na base de dados.

```
51 user = yield get_user(username)
52
53 if user:
54     if user['username'] == username and user['password'] == password:
55         return True
56
57 return False
```

A função `get_user()` devolve o objeto da coleção `users` da base de dados, cujo atributo `username` corresponde ao recebido por argumento.

```
37 user = yield db.users.find_one({'username': username})
38 return user
```

Após terminar o processo de autenticação com sucesso, o utilizador é redirecionado para a página principal.

5.5.2 Página Principal

A página principal funciona como um portal para todas as funcionalidades da aplicação web. Esta página apresenta o número de sensores no sistema e a lista dos sensores (Fig. 5.3) e permite as seguintes funcionalidades:

- adicionar novo sensor ao sistema;
- remover um sensor do sistema.
- reiniciar um sensor;
- editar as configurações de um sensor;
- visualizar as estatísticas de um sensor;
- editar as configurações do servidor;

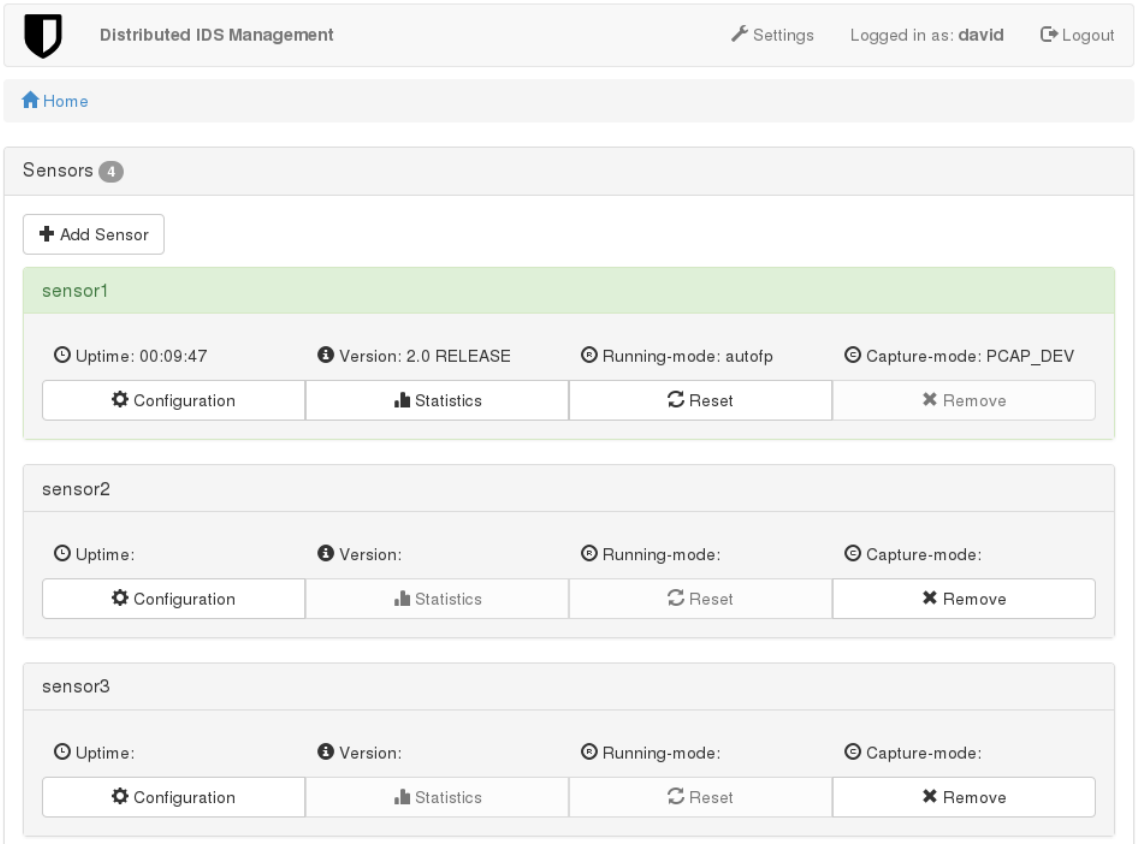

FIGURA 5.3: Página principal.

A página principal está implementada na classe `HomeHandler`.

Classe `HomeHandler`

A classe `HomeHandler` possui apenas um método do tipo `GET`. Utiliza as funções `get_sensors()` e `count_sensors()` para obter respetivamente a lista de e o número de sensores. No final gera a visualização da página `home.html`.

```
64 sensors = yield serverdb.get_sensors()
65 num_sensors = yield serverdb.count_sensors()
66 self.render('home.html', sensors=sensors, num_sensors=num_sensors)
```

O método `get_sensors()` obtém um objeto iterável do tipo `cursor` que contém todos os elementos da coleção de sensores ordenados alfabeticamente pelo atributo `name`. Copia os elementos para a variável sensores e devolve este objeto.

```
116 sensors = []
117
118 cursor = db.sensors.find().sort([('name', +1)])
119
120 while (yield cursor.fetch_next):
121     sensor = cursor.next_object()
122     sensors.append(sensor)
123
124 return sensors
```

A função `count_sensors()` obtém o número de sensores no sistema utilizando a função `count()` que devolve o numero de elementos de uma coleção da base de dados.

```
134 num_sensors = yield db.sensors.count()
```

5.5.3 Adicionar Sensor

A interface que permite adicionar um novo sensor ao sistema está implementada na classe `AddHandler`. O método `GET` gera a página `add.html` (Fig. 5.4). Esta página contém um formulário com um campo para introdução do nome do sensor.

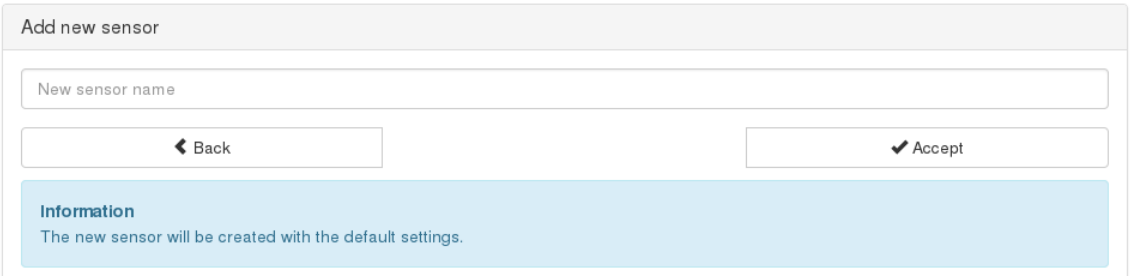

FIGURA 5.4: Adicionar sensor.

Após o envio do formulário, o método do tipo `POST` extrai o argumento `name`, utilizado pela função `add_sensor()` para inserir o sensor na base de dados.

```
177 name = self.get_argument('name', '')
178 yield serverdb.add_sensor(name)
179 self.redirect('/')
```

A função `add_sensor()` cria um novo objeto de sensor. Inicializa o nome e o conteúdo das configurações a partir dos ficheiros de configuração predefinidos existentes no servidor. Insere o objeto na coleção de sensores da base de dados e inicializa o estado correspondente ao sensor na variável de estado global.

```
83 sensor = {}
84
85 sensor['name'] = name
86
87 for k, v in servercfg.paths.items():
88     with open(v, 'r') as f:
89         read_data = f.read()
90         sensor[k] = read_data
91
92 yield db.sensors.insert(sensor)
93 reset_state(name)
```

5.5.4 Remover Sensor

A remoção de um sensor do sistema está implementada na classe `RemoveHandler`. Esta classe possui apenas um método do tipo `GET` que extrai o argumento com o nome do sensor e executa a função `remove_sensor()`.

```
167 name = self.get_argument('name', '')
168 yield serverdb.remove_sensor(name)
169 self.redirect('/')
```

A função `remove_sensor()` remove o objeto de sensor com o nome recebido por argumento da coleção de sensores da base de dados.

```
103 yield db.sensors.remove({'name': name})
```

5.5.5 Editar Configurações de Sensor

O sítio web possui a possibilidade de editar os vários ficheiros de configuração do sensor e do IDS Suricata (Fig. 5.5). Esta funcionalidade está implementada na classe `ConfigHandler()`.

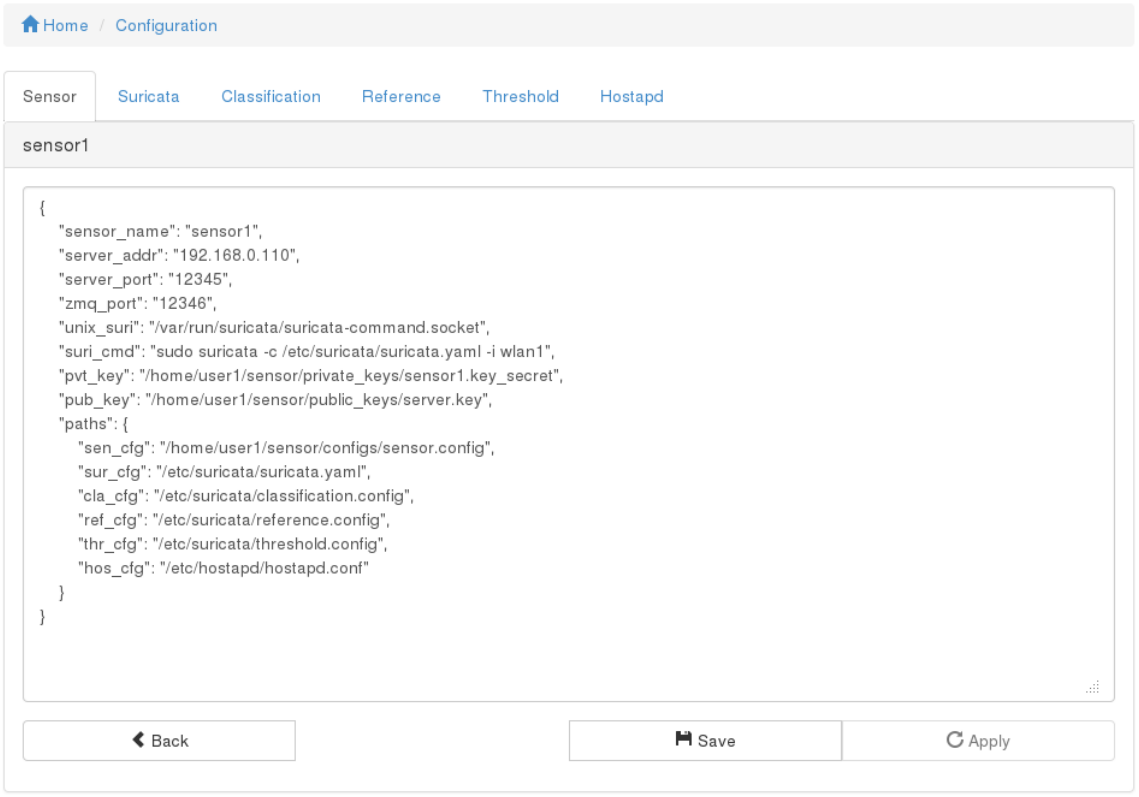

FIGURA 5.5: Editar configuração de sensor.

Inicialmente é gerada a página `config.html` onde o utilizador selecciona o ficheiro a editar.

```
83  name = self.get_argument('name', '')
84  file = self.get_argument('file', '')
85  saved = self.get_argument('saved', '')
86  sensor = yield serverdb.get_sensor(name)
87  conf_files = ['sen_cfg', 'sur_cfg', 'cla_cfg', 'ref_cfg', 'thr_cfg', 'hos_cfg']
88
89  self.render('config.html', sensor=sensor, file=file, saved=saved)
```

Após a modificação do ficheiro é ativado o botão que permite aplicar a nova configuração ao sensor através da função `update_config()`.

```
114 yield serverdb.update_config(name, file, cfg)
115 self.redirect('/config?name=' + name + '&file=' + file + '&saved=' + 'yes')
```

A função `update_config()` obtém a coleção de sensores da base de dados e altera a configuração do sensor com o atributo `name` pretendido, com uma configuração recebida como argumento.

```
1  sensors = db.sensors
2  yield sensors.update({'name': name}, {'$set': {file + '_cfg': cfg}})
```

5.5.6 Visualizar Estatísticas de Funcionamento do Sensor

O IDS Suricata produz continuamente um conjunto de valores que indicam o estado de funcionamento atual (Fig. 5.6).

FIGURA 5.6: Visualização de estatísticas de sensor (parcial).

A funcionalidade que permite ao utilizador visualizar estes valores está implementada na classe `StatsHandler` que gera a página `stats.html` de cada um dos sensores .

```
167 name = self.get_argument('name', '')
168 self.render('stats.html', name=name)
```

Classe `SSEHandler`

Os dados correspondentes às estatísticas são enviados pelo serviço SSE, implementado na classe `SSEHandler()`. O método do tipo `GET` desta classe obtém os dados com a função `get_sse()`.

```
314 msg = serverdb.get_sse()
315 self.emit(msg)
```

O método `initialize()` desta classe constrói os cabeçalhos das mensagens a enviar.

```
322 self.set_header('Content-Type', 'text/event-stream')
323 self.set_header('Cache-Control', 'no-cache')
```

A função `get_sse()` obtém os valores relativos ao tempo de funcionamento ativo e dos contadores estatísticos dos sensores que estiverem em funcionamento. Os sensores que estiverem no estado `SHUTDOWN` são reinicializados.

```
248 state_list = []
249
250 for sensor in state.items():
251
252     (name, data) = sensor
253
254     if data['state'] is not 'CLOSED':
255         state_list.append({'name': name, 'version': data['version'],
256                            'uptime': data['uptime'],
257                            'running_mode': data['running-mode'],
258                            'capture_mode': data['capture-mode'],
259                            'counters': data['counters']})
260         if data['state'] is 'SHUTDOWN':
261             reset_state(name)
262
263 return state_list
```

No final, o método `emit()` formata os dados a enviar em notação JSON e envia a mensagem para o *browser*, onde uma função em linguagem javascript descodifica a mensagem e atribui os valores aos campos corretos da página.

```
322 data = json.dumps(data)
323 msg = 'data: ' + data.strip() + '\n\n'
324 self.write(msg)
325 self.flush()
```

5.5.7 Reiniciar Sensor

A funcionalidade que permite ao utilizador reiniciar um sensor está implementada na classe `ResetHandler`. Esta classe utiliza o método `send_rst()` do módulo `server_zmq`, descrito em 5.4.7.

```
151 name = self.get_argument('name', '')
152 self.server_zmq.send_rst(name)
153 self.redirect('/')
```

5.5.8 Editar Configurações do Servidor

O sítio web possui a possibilidade de editar o ficheiro de configuração do servidor que contém os parâmetros de funcionamento do servidor e os caminhos para os ficheiros de configuração dos sensores predefinidos (Fig. 5.7).

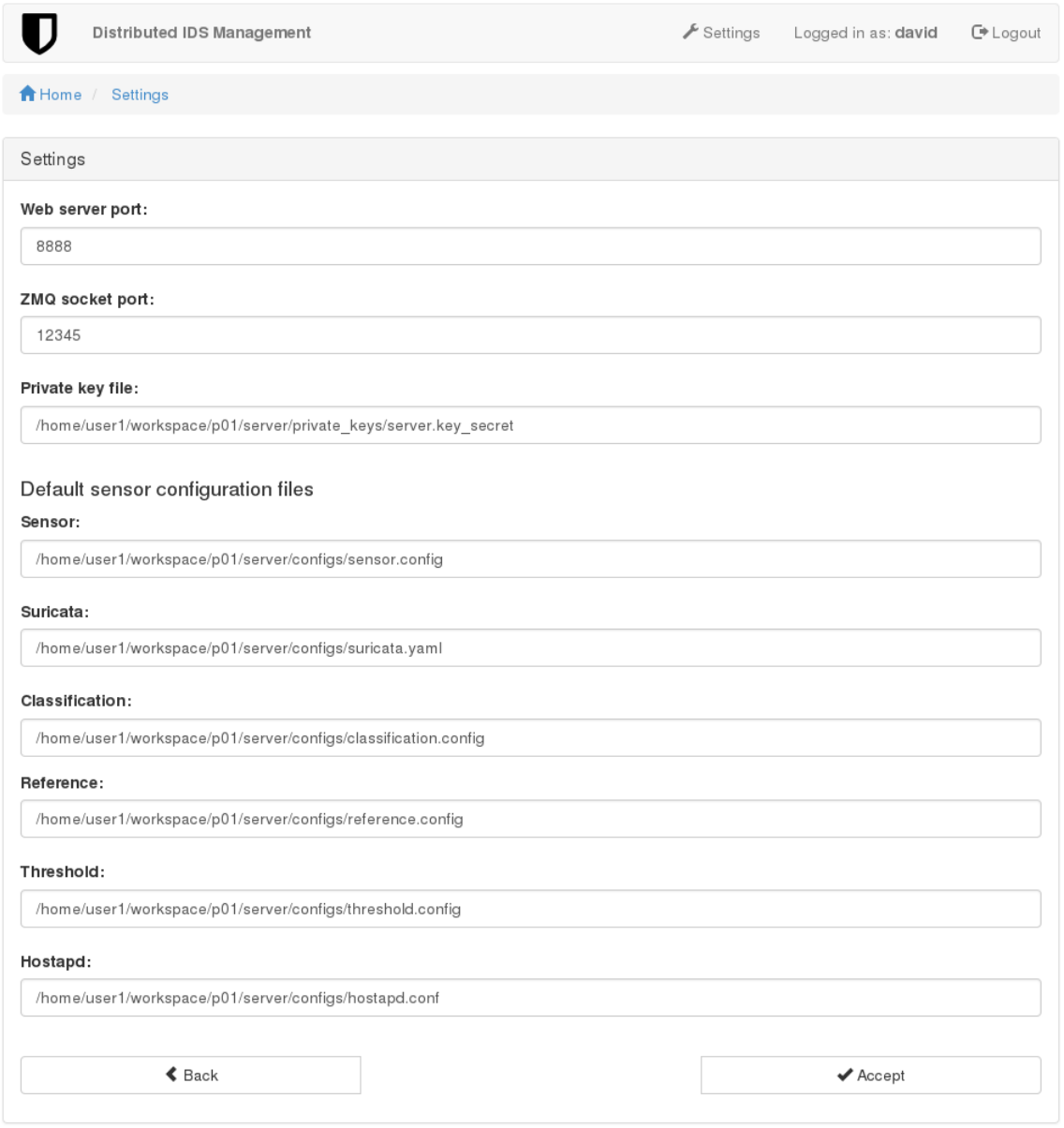

FIGURA 5.7: Editar configuração do servidor.

Esta funcionalidade está implementada na classe `ConfigHandler()`. O método `GET` desta classe gera a página `settings.html` contendo as configurações obtidas com a função `load_settings()`.

```
213 settings = serverdb.load_settings()
214 self.render('settings.html', settings=settings)
```

A função `load_settings()` devolve o conteúdo do ficheiro `server.config`.

```
211 with open(servercfg.conf_path, 'r') as f:
212     settings = json.load(f)
213 return settings
```

O método `POST` desta classe extrai as configurações modificadas e guarda os novos valores com a função `save_settings()`.

```
223 settings = {}
224 settings['paths'] = {}
225 settings['web_port'] = self.get_argument('web_port', '')
226 settings['zmq_port'] = self.get_argument('zmq_port', '')
227 settings['pvt_key'] = self.get_argument('pvt_key', '')
228 path_list = ['sen_cfg', 'sur_cfg', 'cla_cfg', 'ref_cfg', 'thr_cfg',
229              'hos_cfg']
230
231 for path in path_list:
232     settings['paths'][path] = self.get_argument(path, '')
233
234 serverdb.save_settings(settings)
235 self.redirect('/')
```

A função `save_settings()` atualiza o ficheiro server.config com o novo conteúdo.

```
227 servercfg.web_port = settings['web_port']
228 servercfg.zmq_port = settings['zmq_port']
229 servercfg.pvt_key = settings['pvt_key']
230
231 for k, v in settings['paths'].items():
232     servercfg.paths[k] = v
233
234 with open(servercfg.conf_path, 'w') as f:
235     json.dump(settings, f, sort_keys=True, indent=4)
```

Capítulo 6

Protótipo Experimental

O desenvolvimento de aplicações informáticas em ambiente virtualizado apresenta vantagens na rapidez de deteção e resolução de erros programáticos. A implementação de sistemas em dispositivos físicos está suscetível a contrariedades imprevisíveis durante o desenvolvimento em ambiente controlado.

A criação de um protótipo experimental é um passo importante no sentido de produzir um sistema completamente funcional. Os testes realizados nesta fase permitem detetar falhas na comunicação física entre os vários elementos do sistema e o utilizador. Estas falhas podem assim ser colmatadas em próximas iterações do desenvolvimento do projeto.

Na área dos dispositivos de segurança da informação é importante testar o seu funcionamento com equipamentos reais. A complexidade do sistema aumenta devido à interação de elementos de diferentes fabricantes. A incapacidade na gestão da compatibilidade entre os vários componentes pode levar à inviabilidade do projeto.

Neste capítulo descreve-se a realização de um protótipo experimental do sistema desenvolvido. Apresenta-se a topologia da rede de teste e enumeram-se os vários componentes da mesma. Descreve-se detalhadamente a configuração dos elementos tecnológicos utilizados. No final realiza-se um conjunto de testes que confirmam o correto funcionamento do sistema desenvolvido.

6.1 Objetivos

O objetivo da construção do protótipo experimental é provar que o sistema desenvolvido realiza corretamente os requisitos enunciados em 3.4. Nomeadamente, pretende-se demonstrar que o sistema possui as seguintes características:

- otimização de plataformas tecnológicas de múltiplo processamento;
- arquitetura de rede com alta escalabilidade;
- gestão centralizada das configurações dos sensores.

Os critérios anteriores são testados com a realização de um conjunto de experiências práticas subjacentes a uma metodologia previamente definida.

6.2 Protocolo

Os testes realizados com o protótipo experimental seguem um protocolo que especifica de forma sistemática os parâmetros de controlo e os resultados esperados.

Devido à complexidade do sistema, foram criados três testes que restringem o âmbito de cada experiência a uma parte específica do sistema.

6.2.1 Testes Experimentais

Teste de desempenho do sensor: validar o funcionamento do Suricata numa plataforma com vários núcleos de processamento. Utilizar a ferramenta de teste Pytbull para simular ataques informáticos e verificar a captura dos pacotes e a geração de alertas na interface gráfica de controlo.

Teste de comunicação em rede: efetuar a ligação de dois sensores em série e comprovar que as comunicações do sensor mais remoto são recebidas corretamente no servidor.

Alteração da configuração do sensor: alterar o ficheiro de configuração através da interface do servidor HTTP. Verificar que o sensor recebe a nova configuração e reinicia corretamente apresentando as alterações efetuadas.

6.2.2 Topologia Experimental

A topologia da Fig. 6.1 representa uma rede simples que permite realizar os testes descritos anteriormente.

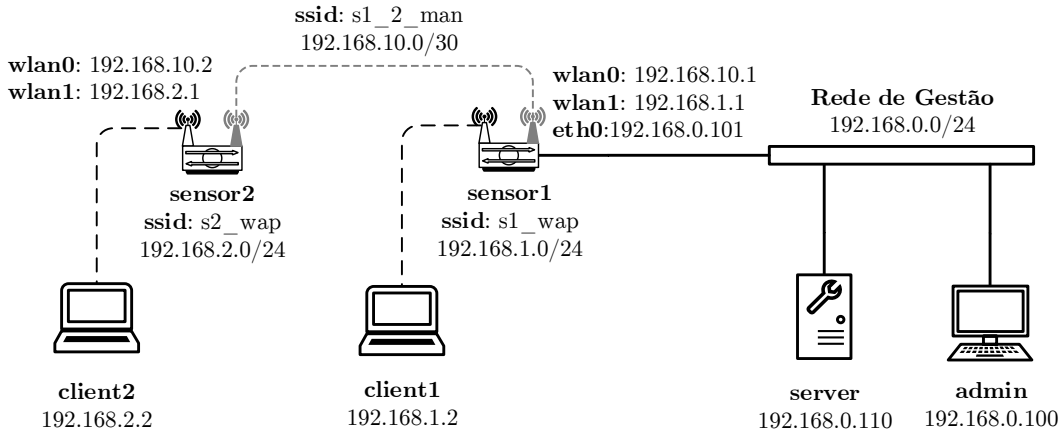

FIGURA 6.1: Topologia experimental.

Esta rede é constituída por seis computadores, um *switch* de rede *ethernet* e seis adaptadores de rede sem fios. De seguida descrevem-se brevemente os principais componentes da rede.

Servidor (`server`): o servidor de teste possui funcionalidade completa. Contém a aplicação Python do servidor, a base de dados MongoDB e o servidor Tornado.

Administrador (`admin`): representa a estação de trabalho do administrador do sistema na sua versão mais simples. Contém um browser web que acede ao servidor.

Sensores (`sensor1/2`): simulam equipamentos de acesso a rede sem fios com IDS integrado. Cada sensor contém a aplicação Python do sensor e a aplicação de acesso a redes sem fios `hostapd`.

Clientes (`client1/2`): simulam clientes maliciosos nas redes sem fios de cada um dos sensores. Cada cliente contém a aplicação Pytbull de teste de IDS.

Todos os computadores possuem o sistema operativo Linux Debian[1] 7.4 e uma instalação da linguagem de programação Python versão 3.4.

[1]Debian Linux - `https://www.debian.org`.

De seguida descrevem-se os detalhes de configuração adicionais para cada um dos elementos do sistema.

6.2.3 Servidor

O servidor possui apenas uma interface de rede *ethernet* com fios para comunicar com a rede de gestão. Contém instalações dos seguintes módulos e aplicações:

- módulo `pyzmq` 14.1.1;
- servidor web Tornado 3.2;
- base de dados MongoDB 2.6.0;
- adaptador da base de dados `motor` 0.2.

Interface de Rede

A interface de rede `eth0` do servidor que liga à rede de gestão está configurada no ficheiro `/etc/network/interfaces` da seguinte forma:

```
auto eth0
iface eth0 inet static
    address 192.168.0.100
    netmask 255.255.255.0
    network 192.168.0.0
    broadcast 192.168.0.255
    gateway 192.168.0.1
    dns-nameservers 192.168.0.1
```

Criação do Utilizador na Base de Dados

No sentido de melhorar a segurança do sistema, a aplicação do servidor não possui mecanismos para criar novos utilizadores dentro da própria aplicação. Estes devem ser inseridos manualmente na base de dados.

A base de dados é acedida através do cliente de linha de comandos `mongo`, e são executados os seguintes comandos:

```
use test
db.users.insert( { username : "david" , password : "pass1234" } )
```

6.2.4 Sensores

Ambos os sensores possuem duas interfaces de rede sem fios. O `sensor1` possui adicionalmente uma interface de rede com fios para ligar ao servidor. Os seguintes módulos e aplicações estão instalados em cada sensor:

- módulo `pyzmq` 14.1.1;
- módulo `aiozmq` 0.0.2;
- ponto de acesso sem fios `hostapd` 1.1.0;
- IDS Suricata 2.0;
- adaptador da base de dados `motor` 0.2.

Interfaces de Rede

Os sensores possuem instalados adaptadores de rede sem fios com o *chipset* Atheros AR9271. Estes dispositivos suportam simultaneamente os modos de funcionamento "AP"e "Monitor", necessários ao `hostapd` e Suricata respetivamente. Apresenta-se o exemplo da configuração para o `sensor1`:

```
auto eth0
iface eth0 inet static
    address 192.168.0.101
    netmask 255.255.255.0
    network 192.168.0.0
    broadcast 192.168.0.255
    gateway 192.168.0.1
    dns-nameservers 192.168.0.1

auto wlan0
iface wlan0 inet static
    address 192.168.10.1
    netmask 255.255.255.252
    wireless-channel 1
    wireless-essid s1_2_man
    wireless-mode ad-hoc

auto wlan1
iface wlan1 inet static
    address 192.168.1.1
    netmask 255.255.255.0
```

A configuração aplicada no `sensor2` varia apenas nos endereços IP.

hostapd

A aplicação `hostapd`[2] implementa o servidor de um ponto de acesso sem fios, incluindo suporte para autenticação WPA. Esta aplicação está configurada no ficheiro `/etc/hostapd/hostapd.conf` da seguinte forma:

```
interface=wlan0
driver=nl80211
ssid=s1_wap
channel=1
hw_mode=g
auth_algs=1
wpa=3
wpa_passphrase=pass1234
wpa_key_mgmt=WPA-PSK
wpa_pairwise=TKIP CCMP
rsn_pairwise=CCMP
macaddr_acl=0
ctrl_interface=/var/run/hostapd
ctrl_interface_group=0
```

Suricata

O Suricata utiliza os ficheiros de configuração predefinidos. O ficheiro principal de configuração do IDS possui um conteúdo idêntico ao apresentado em 4.2.1, variando apenas nos seguintes parâmetros:

Para o `sensor1`:

```
"sensor_name": "sensor1",
"server_addr": "192.168.0.100",
"server_port": "12345",
"zmq_port": "12346",
```

Para o `sensor2`:

```
"sensor_name": "sensor2",
"server_addr": "192.168.10.1",
"server_port": "12346",
"zmq_port": "12347",
```

Esta alteração deve-se ao facto do `sensor1` funcionar como servidor para o `sensor2`, ligando-se este diretamente à interface com o socket `backend` do primeiro.

[2] `Hostapd - http://hostap.epitest.fi/hostapd`

6.2.5 Clientes

Cada cliente possui uma interface de rede sem fios. Os sistemas cliente contêm uma instalação da aplicação Pytbull, para teste do funcionamento do IDS Suricata.

Interfaces de Rede

Os clientes possuem instalados adaptadores de rede sem fios com o *chipset* Realtek RTL8187 devido à sua capacidade de injeção de pacotes na rede. Apresenta-se um exemplo de configuração da interface de rede no `sensor1`:

```
auto wlan0
iface wlan0 inet static
    address 192.168.1.2
    netmask 255.255.255.0
    wpa-ssid s1_wap
    wpa-psk pass1234
```

Pytbull

O Pytbull[3] é uma ferramenta de teste de IDS. É composto por um sistema cliente/-servidor. Realiza uma bateria de 300 testes, agrupados em 11 categorias:

badTraffic: pacotes de rede que não respeitam a formatação correta;

bruteForce: ataques de força bruta (p. ex: passwords `ftp`);

clientSideAttacks: envio de ficheiros com conteúdo malicioso;

denialOfService: simula ataques DoS;

evasionTechniques: técnicas de evasão a IDS;

fragmentedPackets: ataques com conteúdo malicioso disperso por vários ficheiros;

ipReputation: comunicação com servidores com má reputação;

normalUsage: pacotes que simulam a utilização normal da rede;

pcapReplay: reprodução de ficheiros *pcap*;

shellCodes: envio de vários *shellcodes* para o porto 21/tcp;

testRules: teste das regras predefinidas nos IDS Snort/Suricata.

[3]Pytbull - `http://pytbull.sourceforge.net`

O Pytbull requer a instalação dos seguintes pacotes adicionais:

No servidor:

- `apache2`;
- `vsftpd`;

No cliente:

- `ncrack`;
- `nikto`;

Após a instalação dos pacotes referidos anteriormente, a execução da aplicação realiza-se da seguinte forma:

Servidor:

```
$ ./pytbull-server.py -p 12345
```

Cliente (exemplo para `client1`):

```
$ ./pytbull --offline -t 192.168.1.1
```

6.3 Resultados

Após a instalação e configuração de todos os elementos do protótipo experimental, foram realizados os testes e obtiveram-se os seguintes resultados.

Teste de desempenho do sensor

Na primeira parte deste teste constata-se que a execução da aplicação do sensor cria um processo do Suricata. Este processo inicial lança adicionalmente 7 *threads* de processamento de pacotes e 3 *threads* de gestão (Fig. 6.2).

A segunda parte do teste consistiu em executar a ferramenta de teste de IDS Pytbull e registar o resultado nos contadores estatísticos do sensor. O estado atual do IDS é monitorizado em tempo real pelo administrador do sistema. Após a realização da bateria de testes do Pytbull, confirma-se o correto funcionamento do IDS Suricata e a efetividade da transmissão da informação até ao servidor. A análise da Fig. 6.3 indica o processamento de 124156 pacotes capturados na interface `wlan1`, correspondentes à geração de 2856 alertas pelo motor de deteção do Suricata.

```
1  [||                                  1.9%]     Tas[|                                 2/144]
2  [||||||||                           24.1%]    Load average: 0.44 0.35 0.26
Mem[||||||                         421/3967MB]    Upt[                           00:21:23]
Swp[                                 0/8189MB]

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
 2980 root       20   0  4300   608   512 S  0.0  0.0  0:00.00 /bin/sh -c sudo su
 2981 root       20   0 33620  1660  1332 S  0.0  0.0  0:00.00 sudo suricata -c /
 2982 root       20   0  854M  342M  2412 S  1.0  8.6  0:14.33 suricata -c /etc/s
 2988 root       20   0  854M  342M  2412 S  1.0  8.6  0:03.56 suricata -c /etc/s
 2987 root       20   0  854M  342M  2412 S  0.0  8.6  0:00.06 suricata -c /etc/s
 2983 root       20   0  854M  342M  2412 S  0.0  8.6  0:00.00 suricata -c /etc/s
 2984 root       20   0  854M  342M  2412 S  0.0  8.6  0:00.00 suricata -c /etc/s
 2985 root       20   0  854M  342M  2412 S  0.0  8.6  0:00.00 suricata -c /etc/s
 2986 root       20   0  854M  342M  2412 S  0.0  8.6  0:00.00 suricata -c /etc/s
 2989 root       20   0  854M  342M  2412 S  0.0  8.6  0:00.00 suricata -c /etc/s
 2990 root       20   0  854M  342M  2412 S  0.0  8.6  0:00.00 suricata -c /etc/s
```

FIGURA 6.2: Teste de desempenho do sensor.

sensor1 - Performance counters

```
FlowManagerThread:
        flow_mgr.est_pruned: 0
        flow_mgr.new_pruned: 60568
        flow.emerg_mode_entered: 0
        flow.spare: 10000
        flow.emerg_mode_over: 0
        flow.memuse: 7346896
        flow_mgr.closed_pruned: 3064

Detect:
        tcp.segment_memcap_drop: 0
        detect.alert: 2856
        dns.memcap_global: 0
        tcp.invalid_checksum: 746
        tcp.synack: 0
        tcp.reused_ssn: 12
        tcp.reassembly_gap: 0
        tcp.ssn_memcap_drop: 0
        tcp.memuse: 8900544
        tcp.rst: 4190
        tcp.sessions: 63540
        tcp.stream_depth_reached: 0
        tcp.pseudo: 0
        tcp.syn: 65269
        dns.memcap_state: 0
        http.memcap: 0
        tcp.no_flow: 0
        http.memuse: 0
        dns.memuse: 0
        tcp.reassembly_memuse: 0

RxPcapwlan11:
        defrag.ipv6.timeouts: 0
        dns.memcap_global: 0
        decoder.udp: 71
        decoder.ethernet: 124156
        decoder.pkts: 124156
        decoder.icmpv4: 191
        capture.kernel_drops: 0
        decoder.vlan: 0
        decoder.invalid: 1
        decoder.ipv4_in_ipv6: 0
        defrag.ipv6.fragments: 0
        defrag.ipv4.reassembled: 1079
        defrag.ipv4.timeouts: 0
        decoder.avg_pkt_size: 431
        dns.memcap_state: 0
        decoder.raw: 0
        decoder.max_pkt_size: 1514
        decoder.bytes: 53595418
        decoder.ipv6: 6
        decoder.ppp: 0
        decoder.ipv4: 125190
        defrag.max_frag_hits: 0
        decoder.tcp: 121054
        decoder.sll: 0
        capture.kernel_packets: 123642
        decoder.teredo: 0
        capture.kernel_ifdrops: 0
        defrag.ipv4.fragments: 3873
        decoder.vlan_qinq: 0
        defrag.ipv6.reassembled: 0
        decoder.gre: 0
        decoder.pppoe: 0
        decoder.sctp: 0
        decoder.ipv6_in_ipv6: 0
        decoder.icmpv6: 6
        dns.memuse: 0
```

FIGURA 6.3: Estatísticas após testes Pytbull.

Teste da Comunicação em Rede

A ligação é iniciada pelo `sensor2` e reencaminhada através do `sensor1` com sucesso. O estado de ambos os sensores é visualizado na interface web e ocorrem atualizações a cada 5 segundos (Fig. 6.4).

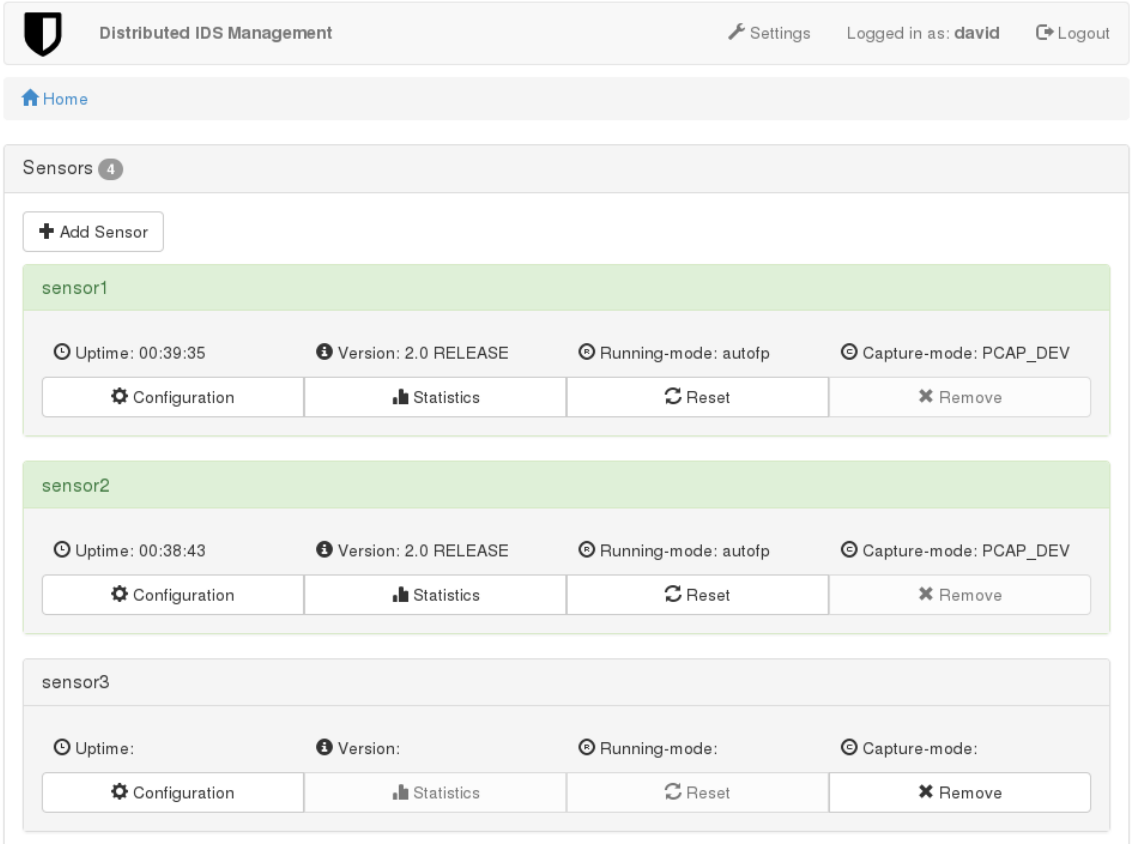

FIGURA 6.4: Teste da comunicação em rede.

Alteração da configuração do sensor

As configurações do `sensor1` são modificadas pelo utilizador. O modo de execução do Suricata (*running-mode*) é alterado de "autofp"para "single", criando apenas uma *thread* de deteção. Após a aplicação das novas configurações o `sensor1` é reiniciado. Ao iniciar novamente a ligação com o servidor, observa-se que o modo de execução foi alterado com sucesso (Fig. 6.5).

FIGURA 6.5: Alteração da configuração do sensor.

6.4 Análise de Resultados

Os resultados obtidos nos três testes anteriores confirmam o funcionamento do sistema de acordo com os critérios definidos: o sensor lança várias *threads* de forma a otimizar as possibilidades de processamento disponíveis; os sensores que se encontram mais afastados do servidor enviam e recebem com sucesso mensagens através de sensores intermediários e o processo de alteração da configuração remota de sensores funciona corretamente.

Estes testes limitam-se, no entanto a garantir a aplicabilidade do sistema numa rede em pequena escala. A simulação do funcionamento do sistema em redes maiores requer a realização de experiências em redes de teste com centenas ou milhares de sensores. Devido às limitações do equipamento disponível, indica-se esta área de investigação como possibilidade de trabalho futuro.

Capítulo 7

Conclusão e Trabalho Futuro

A segurança da informação é um problema cada vez mais relevante para as organizações. A ocorrência de intrusões informáticas pode causar consideráveis prejuízos financeiros e humanos. Este problema é particularmente complexo em redes sem fios. Os sistemas de deteção de intrusões constituem um dos métodos de defesa mais eficazes contra uma variedade de ameaças. No entanto, os sistemas comerciais disponíveis são demasiado complexos, dispendiosos e incapazes de acompanhar o ritmo de expansão das redes distribuídas atuais.

Neste trabalho apresentou-se um Sistema Distribuído de Deteção de Intrusões cujo conceito integra noções de sistemas tradicionais reinterpretados na perspetiva dos recentes avanços tecnológicos nas áreas das comunicações em rede, processamento distribuído e segurança de dados.

O sistema desenvolvido encontra-se estruturado numa rede de sensores que comunicam através dos novos *sockets* ØMQ, permitindo a formação de topologias dinâmicas e com grande escalabilidade, em que os próprios sensores efetuam o encaminhamento de mensagens entre si, sem recurso a equipamento de rede adicional.

A utilização de motores de deteção de intrusões que implementam técnicas de programação concorrente permite otimizar as capacidades das novas plataformas de processamento paralelo de dimensão reduzida, substituindo os pontos de acesso sem fios tradicionais por estes dispositivos e efetuar a deteção de intrusões no próprio dispositivo, reduzindo os custos em equipamentos dispendiosos e largura de banda.

A comunicação entre dispositivos é assegurada pelos mais recentes algoritmos de criptografia de curvas elípticas, aumentando o nível de segurança e reduzindo o tempo de processamento necessário, resultando num processo mais eficiente.

O desenvolvimento do projeto realizou-se exclusivamente com recurso a ferramentas gratuitas de utilização livre. Desde o sistema operativo e ambiente de desenvolvimento, passando pelas linguagens de programação e bibliotecas de comunicação em rede até aos servidores web e de bases de dados, todos estes elementos permitem a modificação livre e adaptação do sistema a necessidades e situações específicas. Estas características tecnológicas e uma interface gráfica intuitiva formam a base de um sistema de gestão de configurações centralizado de uma rede distribuída. A construção de um protótipo experimental com equipamento físico e um gerador de tráfego realista permitiu comprovar o correto funcionamento do sistema.

O sistema distingue dois elementos principais: sensor e servidor. Cada um com funcionalidades distintas. Existe, porém um aspeto tecnológico comum a ambos: o *event loop*. Este mecanismo de controlo de fluxo de execução permite a simplificação do desenvolvimento das aplicações na linguagem de programação Python.

Para o sensor adotou-se uma arquitetura com dois *sockets* ØMQ: `frontend` que comunica com o servidor e `backend` que permite a expansão autónoma da rede. Um *socket* UNIX adicional controla o funcionamento do processo do IDS Suricata e possibilita a alteração da sua configuração e o reinício do processo a quilómetros de distância do dispositivo físico.

O servidor apresenta igualmente uma arquitetura modular multifacetada com um *socket* ØMQ que comunica com os sensores por um lado e um servidor web Tornado que permite a interação com o utilizador por outro lado. A persistência da informação no servidor é assegurada por uma base de dados MongoDB. Estas características tecnológicas e uma interface gráfica apelativa formam a base de um sistema de gestão de configurações centralizado de uma rede distribuída.

Aceitando-se que um projeto tecnológico nunca está realmente terminado, sugerem-se algumas melhorias a efetuar em futuras evoluções do sistema:

- funcionamento em modo de prevenção de intrusões;
- protótipos nas arquiteturas NUC e ARM;
- comunicação em rede em malha;
- aplicações de gestão para dispositivos móveis Android/iOS/WinPho;
- integração em SIEM (por exemplo: OSSIM);

- algoritmo para controlo automático de regras ativas em função dos clientes;
- rede colaborativa entre sensores de diferentes organizações.

O sistema desenvolvido neste trabalho apresenta uma alternativa ás soluções comerciais para o problema da gestão de configurações de sensores num sistema distribuído de deteção de intrusões. A solução alcançada demonstrou ser eficaz devido às tecnologias inovadoras que implementa. Simultaneamente constitui um mecanismo de defesa acessível e preparado para os futuros desafios da segurança da informação.

Referências Bibliográficas

[1] J. P. Anderson. Computer security technology planning study vol. I. Technical report, Air Force Systems Command, 1972.

[2] J. P. Anderson. Computer security technology planning study vol. II. Technical report, Air Force Systems Command, 1972.

[3] J. P. Anderson. Computer security threat monitoring and surveillance. Technical report, Air Force Systems Command, 1980.

[4] M. Bouchard. Securing internal networks. *Juniper Networks*, 2008.

[5] D. E. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, 13(2):222–232, 1987.

[6] D. L. Evans, P. J. Bond, e A. L. Bement. *Standards for Security Categorization of Federal Information and Information Systems (FIPS Pub 199).* National Institute of Standards and Technology, 2004.

[7] J. Graham, R. Howard, e R. Olson. *Cyber Security Essentials.* Auerbach Publications, 2010.

[8] S. Halder e A. Ghosal. Cross layer–based intrusion detection techniques in wireless networks: A survey. In *The State of the Art in Intrusion Prevention and Detection.* Auerbach Publications, 2014.

[9] J. L. M. Hart. An historical analysis of factors contributing to the emergence of the intrusion detection discipline and its role in information assurance. *Air Force Institute of Technology*, 2005.

[10] T. Heberlein, G. Dias, K. Levitt, B. Mukherjee, J. Wood, e D. Wolber. A network security monitor. *Proceedings of the 1990 IEEE Symposium on Security and Privacy*, págs. 296–304, 1990.

[11] P. Hintjens. *ZeroMQ: Messaging for Many Applications*. O'Reilly Media, 2013.

[12] X. Jiang e Y. Zhou. *Android Malware*. Springer, 2013.

[13] R. A. Kemmerer e G. Vigna. Intrusion detection: A brief history and overview. *IEEE Computer Society*, 35(4):27–30, 2002.

[14] T. F. Lunt e P. G. Neumann. A real-time intrusion-detection expert system (IDES). Technical report, SRI International, 1992.

[15] J. McHugh. Intrusion and intrusion detection. *International Journal of Information Security*, 1:14–35, 2001.

[16] J. Milliken. Introduction to wireless intrusion detection systems. In *The State of the Art in Intrusion Prevention and Detection*. Auerbach Publications, 2014.

[17] B. Potter. Wireless intrusion detection. *Network Security - Elsevier*, págs. 4–5, 2004.

[18] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. *Proceedings of LISA '99: 13th Systems Administration Conference*, págs. 229–238, 1999.

[19] C. Sanders e J. Smith. *Applied Network Security Monitoring: Collection, Detection, and Analysis*. Syngress, 2013.

[20] K. Scarfone e P. Mell. *Guide to Intrusion Detections and Prevention Systems (SP 800-94 Rev.1)*. National Institute of Standards and Technology, 2012.

[21] T. Shimeall e J. Spring. *Introduction to Information Security: A Strategic-Based Approach*. Syngress, 2013.

[22] S. Snapp, S. Smaha, D. Teal, e T. Grance. The DIDS (Distributed Intrusion Detection System) prototype. *Proceedings of the Summer USENIX Conference*, págs. 227–233, 1992.

[23] J. R. Vacca. *Computer and Information Security Handbook, Second Edition*. Morgan Kaufmann, 2013.

Apêndice A - Ficheiros html

A1. login.html

```
<!DOCTYPE html>
<html lang="en">
        <head>
        <meta charset="utf-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <meta name="description" content="Login page for Distributed IDS">
        <meta name="author" content="David Palma">
        <link rel="shortcut icon" href="../static/assets/ico/favicon.ico">
        <link href="../static/css/bootstrap.min.css" rel="stylesheet">
        <link href="../static/css/login.css" rel="stylesheet">
        <title>Login</title>
    </head>
    <body>
        <div class="container">
            <form class="form-login" role="form" action="/login" method="post">
                <h1 class="form-login-heading">Login</h1>
                <input type="text" class="form-control" name="username" placeholder
    ="Username" required autofocus>
                <input type="password" class="form-control" name="password"
    placeholder="Password" required>
                <button class="btn btn-lg btn-primary btn-block" type="submit">
    Login</button>
                {% if fail is "yes" %}
                            <div class="alert-danger"><strong>Login failed!</
    strong> Please try again.</div>
                {% end %}
            </form>
        </div>
    </body>
</html>
```

A2. base.html

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <meta name="description" content="Distributed IDS management system">
        <meta name="author" content="David Palma">
        <link rel="shortcut icon" href="../static/assets/ico/favicon.ico">
        <link href="../static/css/bootstrap.css" rel="stylesheet">
        {% block title %}<title>Base</title>{% end %}
    </head>
    <body>
        <div class="container">
            <nav class="navbar navbar-default" role="navigation">
            <div class="container-fluid">
                <div class="navbar-header">
                    <a class="navbar-brand" href="/"><img src="../static/assets/
    images/logo.png"></a>
                    <p class="navbar-text"><strong>Distributed IDS Management</
    strong></p>
                </div>
                <ul class="nav navbar-nav navbar-right">
                <li><a href="/settings"><span class="glyphicon glyphicon-wrench"></
    span> Settings</a></li>
                <li><p class="navbar-text">Logged in as:<strong> {{ current_user
    }}</strong></p></li>
                <li><a href="/logout" class="navbar-link"><span class="glyphicon
    glyphicon-log-out"></span> Logout</a></li>
                </ul>
            </div><!-- /.container-fluid -->
            </nav>
            {% block breadcrumb %}{% end %}
            {% block sensor_list %}{% end %}
            {% block section %}{% end %}
        </div> <!-- /container -->
        {% block scripts %}{% end %}
    </body>
</html>
```

A3. home.html

```
{% extends "base.html" %}
{% block title %}<title>Home</title>{% end %}
{% block breadcrumb %}
    <ol class="breadcrumb">
    <li class="active"><a href="/"><span class="glyphicon glyphicon-home"></span>
    Home</a></li>
```

```
    </ol>
{% end %}
{% block sensor_list %}
    <div class="panel panel-default">
        <div class="panel-heading">
            <h3 class="panel-title">Sensors <span class="badge">{{ num_sensors }}</
    span></h3>
        </div>
        <div class="panel-body">
            <a class="btn btn-default" role="button" href="/add">
            <span class="glyphicon glyphicon-plus"></span> Add Sensor</a>
            {% for sensor in sensors %}
                <div id ="panel-{{ sensor['name'] }}" class="panel panel-default">
                    <div class="panel-heading">
                        <h4 class="panel-title">{{ sensor['name'] }}</h4>
                    </div>
                    <div class="panel-info panel-body">
                        <div class="div-info col-sm-3">
                        <span class="glyphicon glyphicon-time"></span>
                            Uptime: <span id="time-{{ sensor['name'] }}"></span>
                        </div>
                        <div class="div-info col-sm-3">
                            <span class="glyphicon glyphicon-info-sign"></span>
                            Version: <span id="version-{{ sensor['name'] }}"></span
    >
                        </div>
                        <div class="div-info col-sm-3">
                            <span class="glyphicon glyphicon-registration-mark"></
    span>
                            Running-mode: <span id="running-mode-{{ sensor['name']
    }}"></span>
                        </div>
                        <div class="div-info col-sm-3">
                            <span class="glyphicon glyphicon-copyright-mark"></span
    >
                            Capture-mode: <span id="capture-mode-{{ sensor['name']
    }}"></span>
                        </div>
                        <div class="btn-group btn-group-justified">
                            <a href="/config?name={{ sensor['name'] }}&file=sen"
    class="btn btn-default">
                                                    <span class="glyphicon glyphicon-
    cog"></span> Configuration</a>
                                                    <a id="stats-{{ sensor['name'] }}"
    href="#" class="btn btn-default" disabled>
                                                            <span class="glyphicon
    glyphicon-stats"></span> Statistics</a>
                                                    <a id="reset-{{ sensor['name'] }}"
    href="#" class="btn btn-default" disabled>
                                                            <span class="glyphicon
    glyphicon-refresh"></span> Reset</a>
```

```
                                                    <a id="remove-{{ sensor['name'] }}"
     href="/remove?name={{ sensor['name'] }}" class="btn btn-default">
                                                                <span class="glyphicon
    glyphicon-remove"></span> Remove</a>
                        </div>
                    </div>
                </div>
            {% end %}
        </div>
    </div>
<div id="alert"></div>
{% end %}
{% block scripts %}
    <script type="text/javascript">
        function secondsToString(seconds)
        {
                var days = Math.floor(seconds / 86400);
                var hours = Math.floor((seconds % 86400) / 3600);
                var minutes = Math.floor(((seconds % 86400) % 3600) / 60);
                var seconds = ((seconds % 86400) % 3600) % 60;
                rep = "";
                if (days > 0){ rep += days + " days, ";}
                rep += ("0" + hours).slice(-2) + ":";
                rep += ("0" + minutes).slice(-2) + ":";
                return rep + ("0" + seconds).slice(-2);
        };
        function update_divs(upd)
        {
                if(upd.uptime > 0 )
                {
                        document.getElementById("panel-"+upd.name).className="panel
     panel-success";
                        document.getElementById("time-"+upd.name).innerHTML=
    secondsToString(upd.uptime);
                        document.getElementById("version-"+upd.name).innerHTML=upd.
    version;
                        document.getElementById("running-mode-"+upd.name).innerHTML
    =upd.running_mode;
                        document.getElementById("capture-mode-"+upd.name).innerHTML
    =upd.capture_mode;
                        document.getElementById("stats-"+upd.name).href="/stats?
    name="+upd.name;
                        document.getElementById("stats-"+upd.name).removeAttribute(
    "disabled");
                        document.getElementById("reset-"+upd.name).href="/reset?
    name="+upd.name;
                        document.getElementById("reset-"+upd.name).removeAttribute(
    "disabled");
                        document.getElementById("remove-"+upd.name).href="#";
                        document.getElementById("remove-"+upd.name).setAttribute("
    disabled", "disabled");
                }
```

```
                else
                {
                        document.getElementById("panel-"+upd.name).className="panel
     panel-default";
                        document.getElementById("time-"+upd.name).innerHTML="";
                        document.getElementById("version-"+upd.name).innerHTML="";
                        document.getElementById("running-mode-"+upd.name).innerHTML
    ="";
                        document.getElementById("capture-mode-"+upd.name).innerHTML
    ="";
                        document.getElementById("stats-"+upd.name).href="#";
                        document.getElementById("stats-"+upd.name).setAttribute("
    disabled", "disabled");
                        document.getElementById("reset-"+upd.name).href="#";
                        document.getElementById("reset-"+upd.name).setAttribute("
    disabled", "disabled");
                        document.getElementById("remove-"+upd.name).href="/remove?
    name="+upd.name;
                        document.getElementById("remove-"+upd.name).removeAttribute
    ("disabled");
                }
        };
        var source=new EventSource("/sse");
        source.onmessage=function(event)
        {
           json = JSON && JSON.parse(event.data);
                for(var i = 0; i < json.length; i++)
                {
                        var upd = json[i];
                        update_divs(upd);
                }
        };
        </script>
{% end %}
```

A4. add.html

```
{% extends "base.html" %}
{% block title %}<title>Add</title>{% end %}
{% block breadcrumb %}
    <ol class="breadcrumb">
    <li><a href="/"><span class="glyphicon glyphicon-home"></span> Home</a></li>
    <li class="active"><a href="/add"> Add Sensor</a></li>
        </ol>
{% end %}
{% block section %}
    <div class="panel panel-default">
        <div class="panel-heading">
            <h3 class="panel-title">Add new sensor</h3>
        </div>
```

```
        <div class="panel-body">
            <form role="form" action="/add" method="post">
                <div class="form-group">
                    <input type="text" name="name" class="form-control" placeholder
    ="New sensor name">
                </div>
                <div class="btn-group btn-group-justified">
                    <div class="btn-group form-group">
                        <a class="btn btn-default" role="button" href="/">
                        <span class="glyphicon glyphicon-chevron-left"></span> Back
    </a>
                    </div>
                    <div class="btn-group">
                    </div>
                                        <div class="btn-group">
                        <button type="submit" class="btn btn-default" name="submit"
     value="accept" >
                        <span class="glyphicon glyphicon-ok"></span> Accept</button
    >
                    </div>
                </div>
                <div class="alert alert-info">
                    <strong>Information</strong><br>
                    The new sensor will be created with the default settings.
                </div>
            </form>
        </div>
    </div>
{% end %}
```

A5. config.html

```
{% extends "base.html" %}
{% block title %}<title>Configuration</title>{% end %}
{% block breadcrumb %}
    <ol class="breadcrumb">
    <li><a href="/"><span class="glyphicon glyphicon-home"></span> Home</a></li>
    <li class="active"><a href="/config?name={{ sensor['name'] }}&file={{ file }}">
    Configuration</a></li>
    </ol>
{% end %}
{% block section %}
    <ul class="nav nav-tabs">
    <li id="tab-sen"><a href="/config?name={{ sensor['name'] }}&file=sen">Sensor</a
    ></li>
    <li id="tab-sur"><a href="/config?name={{ sensor['name'] }}&file=sur">Suricata
    </a></li>
    <li id="tab-cla"><a href="/config?name={{ sensor['name'] }}&file=cla">
    Classification</a></li>
    <li id="tab-ref"><a href="/config?name={{ sensor['name'] }}&file=ref">Reference
    </a></li>
```

```
    <li id="tab-thr"><a href="/config?name={{ sensor['name'] }}&file=thr">Threshold
    </a></li>
    <li id="tab-hos"><a href="/config?name={{ sensor['name'] }}&file=hos">Hostapd</
    a></li>
    </ul>
<div class="panel panel-default">
    <div class="panel-heading">
        <h3 class="panel-title">{{ sensor['name'] }}</h3>
    </div>
    <div class="panel-body">
        <form role="form" action="/config" method="post">
            <input name="name" value="{{ sensor['name'] }}" hidden>
            <input id="file_name" name="file" value="{{ file }}" hidden>
            <div class="form-group">
                <textarea id="txt" class="form-control" rows="20"></textarea>
            </div>
            <div class="btn-group btn-group-justified">
                <div class="btn-group form-group">
                    <a class="btn btn-default" role="button" href="/">
                    <span class="glyphicon glyphicon-chevron-left"></span> Back</a>
                </div>
                <div class="btn-group">
                </div>
                <div class="btn-group">
                    <button type="submit" class="btn btn-default" name="save_btn"
    value="save" >
                    <span class="glyphicon glyphicon-floppy-disk"></span> Save</
    button>
                </div>
                <div class="btn-group">
                    <button type="submit" class="btn btn-default" name="submit_btn"
     value="apply"
                    id = "submit_btn" disabled><span class="glyphicon glyphicon-
    repeat"></span> Apply</button>
                </div>
            </div>
        </form>
    </div>
</div>
{% end %}
{% block scripts %}
    <script type="text/javascript">
        function activateApply()
        {
            var files = {sen: "{{ sensor['sen_cfg'] }}",
                         sur: "{{ sensor['sur_cfg'] }}",
                         cla: "{{ sensor['cla_cfg'] }}",
                         ref: "{{ sensor['ref_cfg'] }}",
                         thr: "{{ sensor['thr_cfg'] }}",
                         hos: "{{ sensor['hos_cfg'] }}"
                         };
            var args = {}
```

```
            location.search.substr(1).split("&").forEach(function(item)
            {args[item.split("=")[0]] = item.split("=")[1]})
            var file = args['file'];
            document.getElementById("tab-"+file).setAttribute("class", "active");
            var data = files[file];
            document.getElementById("txt").setAttribute("name", file+"_cfg");
            document.getElementById("file_name").setAttribute("value", file);
            document.getElementById("txt").innerHTML=data;
                        if(args['saved'] == 'yes')
            {
                document.getElementById("submit_btn").removeAttribute("disabled");
            }
        };
        if(window.addEventListener)
        {
            window.addEventListener('load',activateApply,false);
        }
    </script>
{% end %}
```

A6. stats.html

```
{% extends "base.html" %}
{% block title %}<title>Statistics</title>{% end %}
{% block breadcrumb %}
        <ol class="breadcrumb">
                <li><a href="/"><span class="glyphicon glyphicon-home"></span> Home
    </a></li>
                <li class="active"><a href="/stats?name={{ name }}">Statistics</a
    ></li>
        </ol>
{% end %}
{% block section %}
        <div class="panel panel-default">
                <div class="panel-heading">
                        <h3 class="panel-title">{{ name }} - Performance counters</
    h3>
                </div>
                <div class="panel-body">
                        <pre id="counters"></pre>
                        <div class="btn-group btn-group-justified">
                                <a class="btn btn-default" role="button" href="/">
                                        <span class="glyphicon glyphicon-chevron-
    left"> Back</span></a>
                        </div>
                </div>
        </div>
{% end %}
{% block scripts %}
        <script type="text/javascript">
```

```
                var source=new EventSource("/sse");
                source.onmessage=function(event)
        {
                json = JSON && JSON.parse(event.data);

                for(var i = 0; i < json.length; i++)
                {
                                var upd = json[i];

                                if(upd.name == "{{ name }}")
                                {
                                        stats=JSON.stringify(upd.counters, null, '\
    t');
                                        document.getElementById("counters").
    innerHTML=stats.replace(/[{}'",]/g, '');
                                }
                        }
        };
        </script>
{% end %}
```

A7. settings.html

```
{% extends "base.html" %}
{% block title %}<title>Settings</title>{% end %}
{% block breadcrumb %}
        <ol class="breadcrumb">
        <li><a href="/"><span class="glyphicon glyphicon-home"></span> Home</a></li
    >
        <li class="active"><a href="/settings"> Settings</a></li>
        </ol>
{% end %}
{% block section %}
    <div class="panel panel-default">
        <div class="panel-heading">
                        <h3 class="panel-title">Settings</h3>
                </div>
                <div class="panel-body">
                        <form role="form" action="/settings" method="post">
                                <div class="form-group">
                                    <label for="web_port">Web server port:</label>
                    <input type="text" name="web_port" class="form-control" value
    ={{ settings['web_port'] }}>
                    <br/>
                    <label for="zmq_port">ZMQ socket port:</label>
                    <input type="text" name="zmq_port" class="form-control" value
    ={{ settings['zmq_port'] }}>
                    <br/>
                    <label for="pvt_key">Private key file:</label>
                    <input type="text" name="pvt_key" class="form-control" value={{
     settings['pvt_key'] }}>
```

```
                    <br/>
                    <h4>Default sensor configuration files</h4>
                    <label for="sen_cfg">Sensor:</label>
                    <input type="text" name="sen_cfg" class="form-control" value={{
    settings['paths']['sen_cfg'] }}>
                    <br/>
                    <label for="sur_cfg">Suricata:</label>
                    <input type="text" name="sur_cfg" class="form-control" value={{
    settings['paths']['sur_cfg'] }}>
                    <br/>
                    <label for="cla_cfg">Classification:</label>
                    <input type="text" name="cla_cfg" class="form-control" value={{
    settings['paths']['cla_cfg'] }}>
                    <br/>
                    <label for="ref_cfg">Reference:</label>
                    <input type="text" name="ref_cfg" class="form-control" value={{
    settings['paths']['ref_cfg'] }}>
                    <br/>
                    <label for="thr_cfg">Threshold:</label>
                    <input type="text" name="thr_cfg" class="form-control" value={{
    settings['paths']['thr_cfg'] }}>
                    <br/>
                    <label for="hos_cfg">Hostapd:</label>
                    <input type="text" name="hos_cfg" class="form-control" value={{
    settings['paths']['hos_cfg'] }}>
                    <br/>
                            </div>
                            <div class="btn-group btn-group-justified">
                                    <div class="btn-group form-group">
                                            <a class="btn btn-default" role="
    button" href="/">
                                            <span class="glyphicon glyphicon-
    chevron-left"></span> Back</a>
                                    </div>
                                    <div class="btn-group">
                                    </div>
                                    <div class="btn-group">
                                            <button type="submit" class="btn
    btn-default" name="submit" value="accept" >
                                            <span class="glyphicon glyphicon-ok
    "></span> Accept</button>
                                    </div>
                            </div>
                        </form>
                </div>
        </div>
{% end %}
```