Computer Standards & Interfaces 33 (2011) 24–34

Contents lists available at ScienceDirect

Computer Standards & Interfaces

journal homepage: www.elsevier.com/locate/csi

Aligning Semantic Web applications with network access controls

William M. Fitzgerald *, Simon N. Foley

Department of Computer Science, University College Cork, Ireland

ARTICLE INFO

Available online 13 October 2009

Keywords:
Network access control
Security
Semantic Web
Ontology

ABSTRACT

Access controls for Semantic Web applications are commonly considered at the level of the application-domain and do not necessarily consider the security controls of the underlying infrastructure to any great extent. Low-level network access controls such as firewalls and proxies are considered part of providing a generic network infrastructure that hosts a variety of Semantic Web applications and is independent of the application-level access control services. For example, it is unusual to include firewall policy rules in an application policy that constrain the kinds of application information different principals may access. As a consequence, an improperly configured infrastructure may unintentionally hinder the normal operation of a Semantic Web application. Simply opening a firewall for HTTP and HTTPS services does not necessarily result in a proper configuration. Taking an ontology-based approach, this paper considers how a firewall configuration should be analyzed with respect to the Semantic Web application(s) that it hosts.

1. Introduction

The Semantic Web [2] builds on application-domain ontologies in order to provide a framework for web-resource reasoning and inter-operation. Semantic Web applications are typically modeled at the application-domain (knowledge) level [4,21] and tend not to consider the underlying infrastructure: it is assumed that this infrastructure is suitably configured to support the application and its web-resources. However, there are situations where the infrastructure configuration may work against the normal operation of the Semantic Web and it becomes necessary to consider some knowledge about the infrastructure and how it relates to the application knowledge.

A reality of any practical system—regardless of the application it supports—is that a network access control (*NAC*) policy is applied to incoming and outgoing traffic. *NAC* configurations, in particular firewall configurations may run to many thousands of rules and are typically maintained on an ad-hoc basis [1,11,23]. New rules are added with little regard to existing rules and may result in an overly-restrictive and/or overly-permissive configuration.

While the Semantic Web may provide applications with security services that are domain-knowledge aware [15,17], it is argued that traditional *NAC*s, such as firewalls and proxies, still have a role to play in securing the low-level infrastructure. Not only do *NAC*s protect services that do not provide built-in application-level security, it is considered best practice to rely on—'belt and braces'—multiple layers of security [25]. A Semantic Web application may span many systems and, as a consequence, its proper operation is dependent on the *NAC* configuration at each system. An overly-restrictive configuration may prevent normal interaction of web-resources resulting in application failure. An overly-permissive configuration, while permitting normal operation of the application, may leave the system vulnerable to attack, for example, across open ports.

In practice, deploying a *NAC* firewall for a web-server or web-client is not simply about opening port 80 on the server for all traffic; one may wish to deny certain nodes (IP addresses, etc.), only accept HTTP traffic from some nodes, require other nodes to use HTTPS and also deal with HTTP traffic that is tunneled through proxies available on other ports. Furthermore, web services do not necessarily communicate on port 80. In addition, *NAC* content sanitation (application layer) provides fine-grained access control that may cut across the host-based access controls; for example, content containing the term 'sex' may be permitted for a medical service but blocked on other services. The ideal firewall configuration is one that is *aligned* with the application supported by the system, that is, it permits valid application traffic, and, preferably, no more and no less.

The contribution of this paper is an approach to the alignment of *NAC* configurations with Semantic Web applications. Semantic Web application knowledge is represented in terms of an ontology and we develop ontologies for Linux Netfilter filtering capabilities and TCP-Wrapper which are used to represent and reason about NAC configurations. By bridging these ontologies with the application ontology, it becomes possible to reason about how knowledge within the application may affect a *NAC* configuration, and vice-versa. In addition to analyzing whether an existing NAC configuration is aligned with an application (ontology), the approach can be used to automatically generate suitable NAC configurations aligned with the application.

This paper is a revised and extended version of [8] and is organized as follows. Section 2 provides an overview of Description Logic (*DL*), which is used to construct the ontologies in this paper. *DL* representations of

* Corresponding author.
E-mail addresses: info@williamfitzgerald.net (W.M. Fitzgerald), s.foley@cs.ucc.ie (S.N. Foley).

0920-5489/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.csi.2009.10.002

the Netfilter firewall and TCP-Wrapper proxy ontologies are given in Section 3. Section 5 describes how a further ontology is used to bridge the *NAC* ontology with the Semantic Web application E-Tourism case study outlined in Section 4. How the resulting ontology can be reasoned about is discussed in Section 6.

2. Description logic and ontologies

An ontology is an explicit specification of a conceptualisation using an agreed vocabulary and provides a rich set of constructs to build a more meaningful level of knowledge. An important characteristic of the Semantic Web is that its information is specified using a formal language in order to enable automated reasoning and the derivation of new knowledge from existing knowledge.

Description Logic (*DL*) is a family of logic-based formalisms that forms part of the W3C recommendation for the Semantic Web [3]. *DL* uses classes (concepts) to represent sets of individuals (instances) and properties (roles) to represent binary relations applied to individuals. For example, the *DL* assertion:

$$ServerNode \sqsubseteq Node \sqcap \exists hasSemanticService.Service \sqcap \exists hasFirewall.Firewall$$

specifies that a server node (class) hosts services (class) and has (property) a firewall (class) protecting them. Note that properties are conventionally prefixed by "*has*"; for instance, *hasSemanticService*, is the property over the individuals of the class *ServerNode* (domain) that host Semantic Web services (range). The Appendix provides an overview of Description Logic and we will also explain its constructors when they are first used in the subsequent sections of the paper.

The Semantic Web Rule Language, (*SWRL*), complements *DL* providing the ability to infer additional information in *DL* ontologies, but at the expense of decidability. *SWRL* rules are Horn-clause like rules written in terms of *DL* concepts, properties and individuals. A *SWRL* rule is composed of an antecedent part and a consequent part, both of which consist of positive conjunctions of atoms [18]. For example, the SWRL rule: *servers hosting open-vpn based semantic services protected by a firewall require that firewall to open port 1194* is represented as:

$$ServerNode(?n) \wedge hasSemanticService(?n,?s) \wedge hasPort(?s, portVPN) \wedge hasFirewall(?n,?f) \rightarrow hasPortOpen(?f,\text{portVPN})$$

3. NAC ontology knowledge base

3.1. Netfilter ontology

Netfilter [11] is a framework that enables packet filtering, network address translation (NAT) and packet mangling. As a firewall, it is both a stateful and stateless packet filter that is characterised by a sequence of firewall rules against which all packets traversing the firewall are filtered. Each firewall rule takes the form of a series of conditions representing packet attributes that must be met in order for that rule to be applicable, with a consequent action for the matching packet (accept, drop, log and so forth).

Our research focuses on the firewalling aspects of Netfilter and our current model only incorporates the `filter` table attributes. The model can be extended to include additional features of Netfilter such as `NAT` and `mangle` tables. Netfilter requires the specification of a *table* (`filter`, `NAT` or `mangle`), a *chain*, the accompanying rule *condition* details and an associated *target* outcome. A table (`filter` in our model) is a set of chains and it defines the global context, while chains define the local context within a table. A chain is a set of firewall rules and those rules in a chain are applied to the context defined both by the chain itself and the particular table. By default there is no need to specify the `filter` table (using `-t` option) when defining firewall rules. A Netfilter rule has the following components.

```
[ Table][ Chain Type][ Filter Conditions][ Target Decision] .
```

For example, a rule that states that HTTP traffic along the `FORWARD` chain outward bound from the trusted internal interface `eth0` is permitted is:

```
iptables -t filter -A FORWARD -o eth0
                   -p tcp --dport 80 -j ACCEPT
```

3.1.1. Firewall rule components

The Netfilter `filter` table (the overall firewall configuration policy) is composed of a number of sub-governing local policies controlled by each of its in-built chains (*Chain* class) to which various protective packet condition filters (ConditionFilter class) and their respective verdict permissions (*Target* class) are applied. Netfilter provides a mechanism of three separate firewall or filtering chains to police various kinds of network traffic (Fig. 1). These chains filter traffic being routed to, from and beyond the firewall device itself [20].

Classes in *DL* represent concepts within the domain of interest and in our formal ontology model, various domain specific classes provide knowledge of the key features of Netfilter's filtering capabilities. Individual objects that belong to a class are referred to as instances of that class. We have developed a *DL*-based ontology for Netfilter. Fig. 2 depicts a fragment of the class taxonomy for this model. The taxonomy provides the classes, subclasses and individuals that are inferred from the *DL* specification of the ontology. In the following subsections, we briefly illustrate some abstract examples to provide the reader with enough information as to how we codified the salient features of Netfilter in *DL*. Both disjoint and covering axioms are used extensively throughout our formal model. However, for reasons of space, we omit these axioms when presenting *DL* definitions in this paper. The reader should assume that all sibling classes defined in the paper are disjoint unless stated otherwise. The Appendix provides further information on *DL*.

3.1.1.1. Chain types and chain decision policy. Each Netfilter firewall chain must be assigned a default decision policy of 'accept' or 'drop', such that, if packets that have been correlated against the complete set of firewall rules within that chain have not met any of the filter conditions then the default decision policy is executed to decide the fate of those packets. The following is a detailed description of the *Chain* class definition; it is also used to explain to the reader some of the *DL* language.

The *Chain* class, subsumed by the *FirewallDomain* class, is defined as a *complete* class ($\equiv$) and as a domain concept with restrictions applied to the *hasChainDecision* property that binds individuals of the *Chain* class to individuals of another class within the firewall domain called *ChainPolicy*. The *hasChainDecision* property is a binary relation that is constrained with an existential ($\exists_{=1}$) restriction across that property with a cardinality value of 1. It states that there exists a single relationship between individuals of the *Chain* class and individuals of the *ChainPolicy* class. There is also a closure axiom applied to the *hasChainDecision* property via the universal ($\forall$) restriction. It states that individuals of the

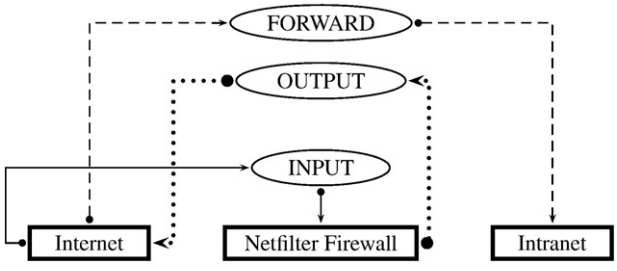


Fig. 1. Linux Netfilter (filter table) packet traversal.

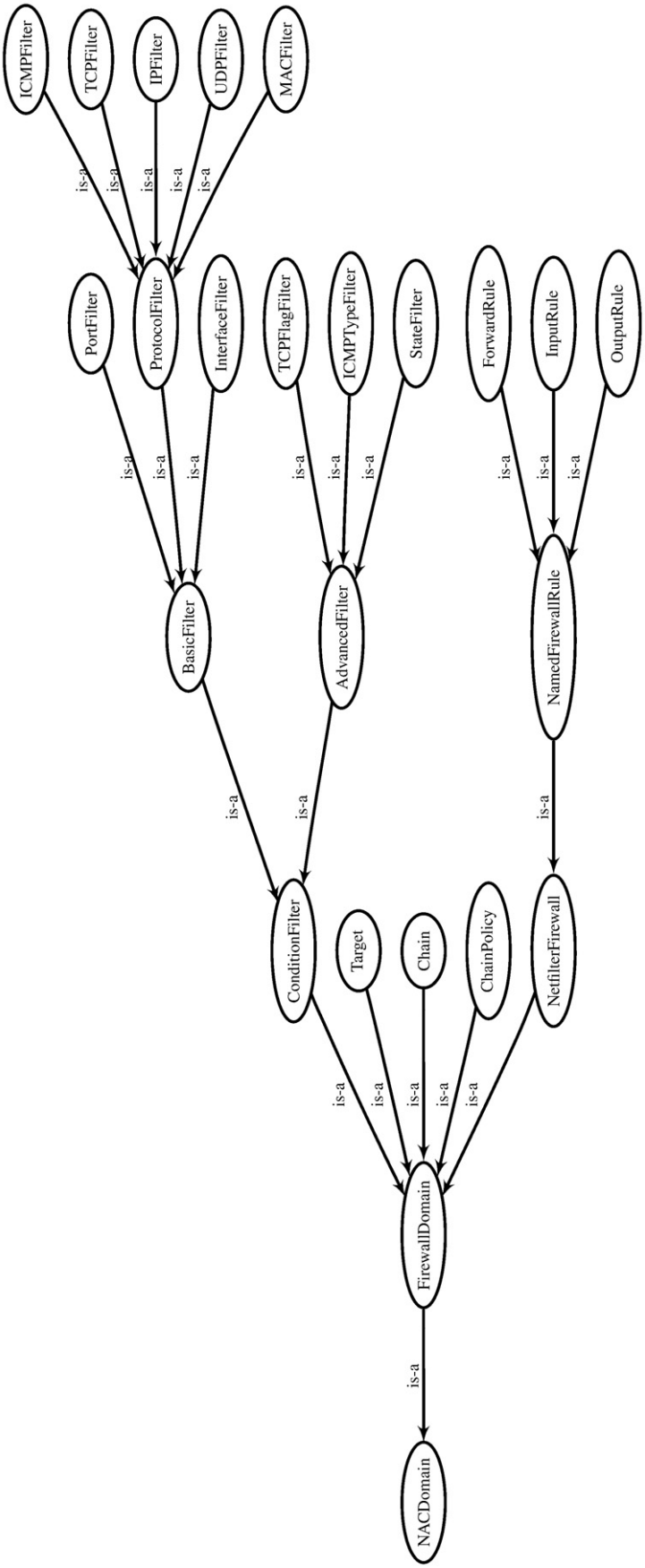


Fig. 2. Netfilter class taxonomy (generated from protégé).

Chain class can only ever have *hasChainDecision* relationships to individuals of the *ChainPolicy* class using that property. Atomic individuals are written in a San Serif font.

$$\begin{aligned} Chain \equiv\ & FirewallDomain \sqcap \\ & \exists_{=1} hasChainDecision.ChainPolicy \sqcap \\ & \forall hasChainDecision.ChainPolicy \sqcap \\ & \{\mathsf{input}, \mathsf{output}, \mathsf{forward}\} \end{aligned}$$

The class *ChainPolicy* defines the vocabulary for a default policy decision across a firewall chain.

$$\begin{aligned} ChainPolicy \equiv\ & FirewallDomain \sqcap \\ & \{\mathsf{decisionDeny}, \mathsf{decisionAccept}\} \end{aligned}$$

Within Netfilter there are two approaches to filtering: the first is a *deny everything by default*, thereafter explicitly permitting selected packets; the second approach, is to *accept everything by default*, thereafter explicitly denying selected packets. The first approach is usually implemented as security best practice [25]. The *ChainPolicy* class is composed of two distinct individuals: decisionDeny and decisionAccept to represent the default policy decisions that can be applied to a chain. Chains have a default policy decision in order to govern how it processes unmatched traffic. Netfilter provides a mechanism for traffic to be inspected and analyzed, depending on various pre-firewall routing rules (Fig. 1). Thus, the *Chain* class provides a finite/enumeration set of three individuals namely, input, output and forward. For example, the input individual (instance of class *Chain*) is a chain to which packet condition filters and corresponding target decisions can be applied to, in order to analyze traffic incoming to local services hosted on the actual firewall itself.

3.1.1.2. Filter conditions. Netfilter firewall chains `INPUT`, `OUTPUT` and `FORWARD` govern routed traffic and can (and normally do) contain a set of packet filter conditions. Hence, chains act as containers for firewall rules. The class *ConditionFilter* represents the kinds of filters defined in our model:

$$ConditionFilter \sqsubseteq FirewallDomain$$

Filtering criteria that can inspect individual packets can be further subdivided into two more specialised categories: basic filtering techniques (Class *BasicFilter*) and advanced filtering techniques (Class *AdvancedFilter*). Basic filtering has the ability to inspect each packet. It can be applied in various levels of granularity: to a particular interface or set of interfaces, to various protocols (MAC address, IP address or range of IP addresses, TCP, UDP and ICMP), or to a particular port or set of ports. Netfilter's advanced filtering techniques involve deep packet header inspection (TCP flags and ICMP types) and it can filter based on stateful connection state (previous packet streaming context).

$$BasicFilter, AdvancedFilter \sqsubseteq ConditionFilter$$

In the following examples, we demonstrate how condition filters can be modelled in more depth for port, TCP flag and state filtering.

3.1.1.3. Basic filter by port type. Port condition filtering occurs at Layer 4 of the OSI stack. Ports (for example, individuals like portSSH) in our model are defined as a class of individuals that should be constrained to protocols TCP or UDP. Hence, we define a closure axiom that states: should a port be associated to a protocol along the *hasProtocol* property relationship then it must only be to *TCPFilter* or *UDPFilter*.

$$\begin{aligned} PortFilter \sqsubseteq\ & BasicFilter \sqcap \\ & \forall hasProtocol(TCPFilter \sqcup UDPFilter) \end{aligned}$$

3.1.1.4. Advanced filter by TCP flag. The model defines three advanced filtering techniques (TCP flags, ICMP types and statefulness) that the Netfilter framework is capable of providing. It is sometimes useful to permit TCP connections in a single direction, but not in the opposite direction. For example, an administrator might want to permit the initiation of connections to an external web-server, but not an initiation connection from that server to the internal network. The solution is to deny only the web-server traffic used to request a connection, that is the syn, to the internal network. The TCP protocol has six such extensions that can be adopted in a firewall rule and are represented in the following *DL* definition:

$$\begin{aligned} TCPFlagFilter \equiv\ & AdvancedFilter \sqcap \\ & \exists_{=1} hasProtocol.TCPFilter \sqcap \\ & \{\mathsf{syn}, \mathsf{ack}, \mathsf{rsh}, \mathsf{psh}, \mathsf{fin}, \mathsf{urg}\} \end{aligned}$$

3.1.1.5. Advanced filter by stateful operands. The Netfilter firewall framework has stateful capabilities (*StateFilter*) that can filter at Layer 4 (TCP, UDP) and Layer 3 (ICMP) of the OSI model to filter packets based on the context of the traffic's current stream. Packets can be filtered basedon the following attributes: *NEW* (equivalent to the TCP SYN request or initial UDP packet), *ESTABLISHED* (equivalent to ongoing TCP ACK traffic after connection has been established), *RELATED* (ICMP error messages or FTP secondary connections etc) and *INVALID* operations. The stateful capabilities augment the stateless, static packet filter protection. State information is recorded when a TCP connection or UDP exchange is initiated and subsequent packets are examined not only based on stateless tuple rules but also on the context of the ongoing connection [23]. Stateful filtering does not apply to MAC addresses or IP addresses so we define a complement along the universal restrict ion of *MACFilter* or *IPFilter*.

$$\begin{aligned} StateFilter \equiv\ & AdvancedFilter \sqcap \\ & \neg(\forall hasProtocol.(MACFilter \sqcup IPFilter)) \sqcap \\ & \{\mathsf{new}, \mathsf{established}, \mathsf{related}, \mathsf{invalid}\} \end{aligned}$$

3.1.1.6. Target permissions. When a filter condition matches a packet traversing a particular chain, a firewall target option specifies the fate of that packet (for example, *DROP* or *ACCEPT* the packet). Netfilter provides a mechanism of packet authorisations (class individuals) represented by the *Target* class in our model for this purpose. The *Target* class is defined as a *complete* class (≡) that details the *necessary & sufficient* conditions of class membership. The reader is referred to [3] for an introduction to *DL*.

$$\begin{aligned} Target \equiv\ & FirewallDomain \sqcap \\ & \{\mathsf{return}, \mathsf{reject}, \mathsf{ulog}, \mathsf{log}, \mathsf{drop}, \mathsf{accept}\} \end{aligned}$$

3.1.2. Firewall rule composition

In this section, we illustrate some examples of firewall rule constraints that are constructed in terms of the model vocabulary proposed above. Various kinds of firewall rules can be defined as subclasses of the *NamedFirewallRule* class in our model. This class defines the *necessary & sufficient* conditions for the composition of a firewall rule. A Netfilter firewall rule is composed of exactly one chain, one or more condition filters and a single permission target.

$$\begin{aligned} NamedFirewallRule \equiv\ & NetfilterFirewall \sqcap \\ & \exists_{=1} hasChain.Chain \sqcap \\ & \exists_{\geq 1} hasCondition.ConditionFilter \sqcap \\ & \exists_{=1} hasTarget.Target \end{aligned}$$

To instantiate `FORWARD` rules, for example, it is first necessary to define the membership constraints of class *ForwardRule*. This class is a more specialised *NamedFirewallRule* class whereby `FORWARD` rules must have a has-value (∈) relationship to a specific *Chain* class individual, forward, along the *hasChain* property:

$$\begin{aligned} ForwardRule \equiv\ & NamedFirewallRule \sqcap \\ & \in hasChain.\mathsf{forward} \end{aligned}$$

Note, additional firewall rule components such as rule order (*hasRuleOrder*) are beyond the scope of this paper. While the order of rules plays an important role in the internal structuring of a firewall

policy configuration, for the purposes of this paper it is assumed that the individual firewall rules have the correct sequence. The significance of rule ordering is considered is in [7].

3.1.3. Firewall configuration

The previous sections describe an ontology for firewall configuration. A firewall rule instance, defined in terms of specific ports, protocols, etc., is represented as an instance of the ontology defined in terms of class individuals. For example, the firewall rule

```
iptables -A FORWARD -i eth1
                    -s 4.3.2.1 -d 1.2.3.4
                    -p tcp - -dport 1194 -j ACCEPT
```

is represented by an individual vpnAccess in the ontology whereby the following *ForwardRule*(vpnAccess) holds; intuitively, we can think of vpnAccess as an individual of class *ForwardRule*. This individual is inferred across the ontology whereby,

$$\begin{aligned} \textit{ForwardRule}(\text{vpnAccess}) \leftarrow\ & \textit{hasChain}(\text{vpnAccess, forward}) \sqcap \\ & \textit{hasExternalInterface}(\text{vpnAccess, eth1}) \sqcap \\ & \textit{hasSrcIP}(\text{vpnAccess, ip4.3.2.1}) \sqcap \\ & \textit{hasDstIP}(\text{vpnAccess, ip1.2.3.4}) \sqcap \\ & \textit{hasProtocol}(\text{vpnAccess, tcp}) \sqcap \\ & \textit{hasDstPort}(\text{vpnAccess, portVPN}) \sqcap \\ & \textit{hasTarget}(\text{vpnAccess, accept}) \end{aligned}$$

Filtering for example Denial of Service (*DoS*) attacks to a web-server through the use of a syn-threshold rule-set that: a) limits the number of TCP connections to the web-server to 1 per second after 4 connections have been observed and b) offending packets that exceed the limit are dropped, is expressed as follows:

```
iptables -A FORWARD  -d WebServerIP -p tcp --syn
                     -m limit --limit 1/s --limit-burst 4
                     -j ACCEPT
iptables -A FORWARD  -d WebServerIP -p tcp --syn
                     -j DROP
```

The following are *DL* fragments that are representative of the above Netfilter rules:

$$\begin{aligned} \textit{ForwardRule}(\text{synDoSLimit}) \leftarrow\ & \textit{hasChain}(\text{synDoSLimit, forward}) \sqcap \\ & \textit{hasDstIP}(\text{synDoSLimit,webServerIP}) \sqcap \\ & \textit{hasProtocol}(\text{synDoSLimit, tcp}) \sqcap \\ & \textit{hasTCPFlag}(\text{synDoSLimit, syn}) \sqcap \\ & \textit{hasLimit}(\text{synDoSLimit, 1}) \sqcap \\ & \textit{hasLimitBurst}(\text{synDoSLimit, 4}) \sqcap \\ & \textit{hasTarget}(\text{synDoSLimit, accept}) \end{aligned}$$

$$\begin{aligned} \textit{ForwardRule}(\text{synDoSDrop}) \leftarrow\ & \textit{hasChain}(\text{synDoSDrop, forward}) \sqcap \\ & \textit{hasDstIP}(\text{synDoSDrop,webServerIP}) \sqcap \\ & \textit{hasProtocol}(\text{synDoSDrop, tcp}) \sqcap \\ & \textit{hasTCPFlag}(\text{synDoSDrop, syn}) \sqcap \\ & \textit{hasTarget}(\text{synDoSDrop, drop}) \end{aligned}$$

Note that the low-level facts of a firewall configuration are presented as individuals rather than classes on the basis that they are atomic and will not be further decomposed. Using instances (rather than subclasses) allows subsequent reasoning of collections of firewall rules using *SWRL* [14].

3.2. TCP-Wrapper ontology

The Linux/Unix-based TCP-Wrapper [24] service is a host-based transport layer proxy that provisions network access control to local daemons spawned by the Internet services daemon (*inetd*). Under typical circumstances Linux environments use a super server (*inetd*) to invoke TCP/IP based network services, for example the *open-vpn* service. Instead of invoking the *open-vpn* service (`openvpnD` daemon) directly the *inetd* daemon will invoke the TCP-Wrapper daemon. The TCP-Wrapper proxy will permit or deny access to the requested service daemons it protects based on the requesting client (for e.g. an IP address) as ascertained from the *inetd* network connection. If a rule has concluded with a permit action then the TCP-Wrapper proxy invokes the appropriate service daemon.

A TCP-Wrapper policy — a set of access control rules to protect host-based services — is specified in terms of access control files *hosts.allow* and *hosts.deny*. Recent versions allow the access controls to be specified in a single file. Like firewall rules, the ordering of rules is vital, thus once a rule has been matched no further rules are processed. A TCP-Wrapper rule has the following components.

```
[ Daemon List][ Client List][ Options][ Target Action]
```

For example, a rule to deny access to a vpn server from requestors of a particular subnet whereby attempts by IP addresses of that subnet to access the protected service are logged and time-stamped (using `spawn`) can be written as follows.

```
open-vpnD : 192.168.1. : spawn /bin/echo '/bin/date'
                         %h ≫ /var/log/vpn.log : deny
```

3.2.1. Proxy rule components

We have developed a *DL*-based ontology for TCP-Wrapper. Fig. 3 depicts a fragment of the class taxonomy for this model. The taxonomy provides the classes, subclasses and individuals that are inferred from the *DL* specification of the ontology.

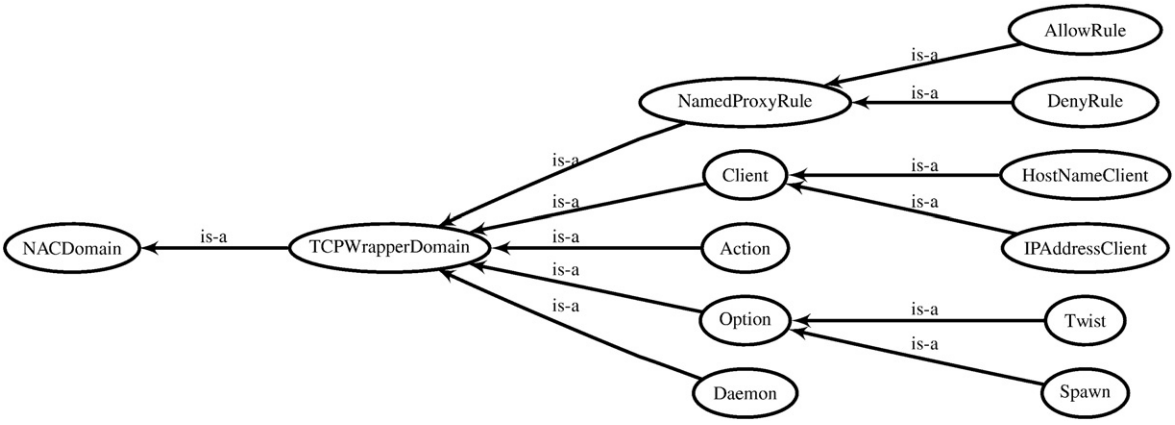


Fig. 3. TCP-Wrapper class taxonomy (generated from protégé).

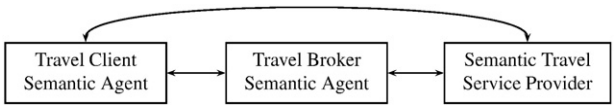


Fig. 4. E-Tourism abstract service interaction.

3.2.1.1. Protected services. Class *Daemon* defines an enumerated or finite set of server-side services. Typical examples are ssh, ftp, email and so forth.

$Daemon \equiv TCPWrapperDomain \sqcap \{ssdD, ftpD, imapD, \ldots\}$

3.2.1.2. Client access. TCP-Wrapper can filter access to individuals of the *Daemon* class in a number of ways. It can filter based on a DNS name or at an IP address level.

$HostNameClient, IPAddressClient \sqsubseteq Client$

3.2.1.3. Optional extensions. There are various optional features that can be configured within a rule that provide additional actions when a rule has been matched. For example, log offending denied clients, notify an administrator via email and so forth. There are two primary options called *spawn* — launches a shell command as a child process which can be used to log data and *twist* — replaces the requested service with the specified command and can be used as a honey-pot. The TCP-Wrapper options are defined by the following DL expression:

$Spawn, Twist \sqsubseteq Option$

3.2.1.4. Target actions. Like Netfilter, when a TCP-Wrapper filter condition matches a packet destined for a locally protected daemon or service the target action of that rule specifies the fate of that packet (for example, allow or deny the packet). The *Action* class is defined as an enumerated *complete* class that details the *necessary & sufficient* conditions of class membership.

$Action \equiv TCPWrapperDomain \sqcap \{allow, deny\}$

3.2.2. Proxy rule composition

In this section, we illustrate some examples of wrapper rule constraints that are constructed in terms of the model vocabulary proposed above. A TCP-Wrapper rule at its simplest level can be described as having the following components; one or more service daemons, one or more requesting clients, zero or more options and exactly one permission target action and is represented by the assertion:

$$\begin{aligned} NamedProxyRule \equiv\ & TCPWrapperDomain \sqcap \\ & \exists_{\geq 1} hasDaemon.Daemon \sqcap \\ & \exists_{\geq 1} hasClient.Client \sqcap \\ & \forall_{\geq 0} hasOption.Option \sqcap \\ & \exists_{=1} hasAction.Action \end{aligned}$$

3.2.3. Proxy configuration

A TCP-Wrapper rule prule, (an individual of *NamedProxyRule*) that states a trusted client IP address is permitted access to the protected open-vpnD daemon is represented by the following:

$$\begin{aligned} NamedProxyRule(\text{prule}) \leftarrow\ & hasDaemon.(\text{prule}, \text{open} - \text{vpnD}) \sqcap \\ & hasClient(\text{prule}, \text{ip4.3.2.1}) \sqcap \\ & hasAction(\text{prule}, \text{allow}) \end{aligned}$$

The corresponding TCP-Wrapper syntax detailing the access control of the previous knowledge base individual prule is written as:

```
open-vpnD: 4.3.2.1 : allow
```

4. E-Tourism application

Based on the W3C use-case scenario defined in [28], a semantics-aware travel broker service (Fig. 4) provides (intelligent) travel clients with an ability to query and purchase complete vacation/business packages based on user preferences. Typically, the travel client will interface with one or more travel broker service providers with which they have a subscription. The travel broker service provider interacts with various service providers (transport, accommodation, activities and so forth) on behalf of client requests. Direct interaction to a specific service provider by the travel client is also possible.

The *ETourismDomain* ontology defines the knowledge within the E-Tourism application. For example, *ETourismDomain* is sub-classed to describe the kinds of classes that are transport related:

$TransportService \sqsubseteq ETourismDomain$

A simple taxonomy of the overall E-Tourism ontology of this application is provided in Fig. 5.

In this example (Fig. 6), we focus on the operation of the airline service that provides three different levels of service (gold, silver and bronze) based on client contract. These service types are modeled as individuals of the FlightService class:

$FlightService \sqsubseteq TransportService \sqcap \{gold, silver, bronze\}$

The gold service provides unlimited access to the airline booking and auction system which is offered to the airline's sales branch itself along with premium business partners (for example, travel brokers). The silver grade service provides a more constrained service offering but at a cheaper rate, while the bronze grade is offered to all public users.

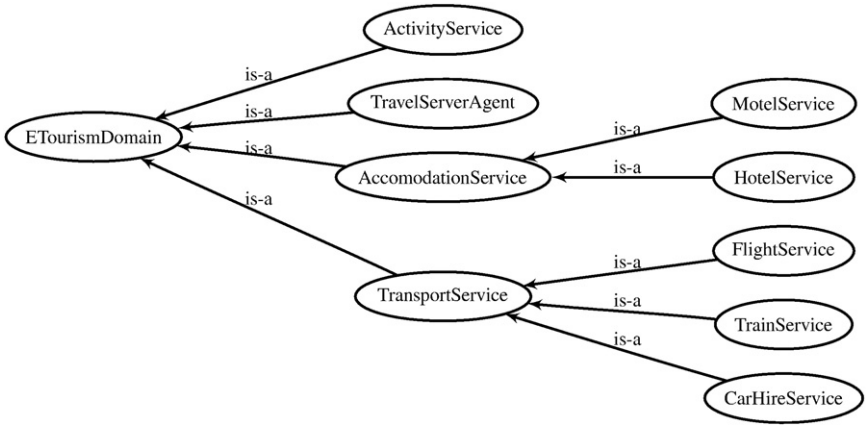


Fig. 5. E-Tourism taxonomy (generated from protégé).

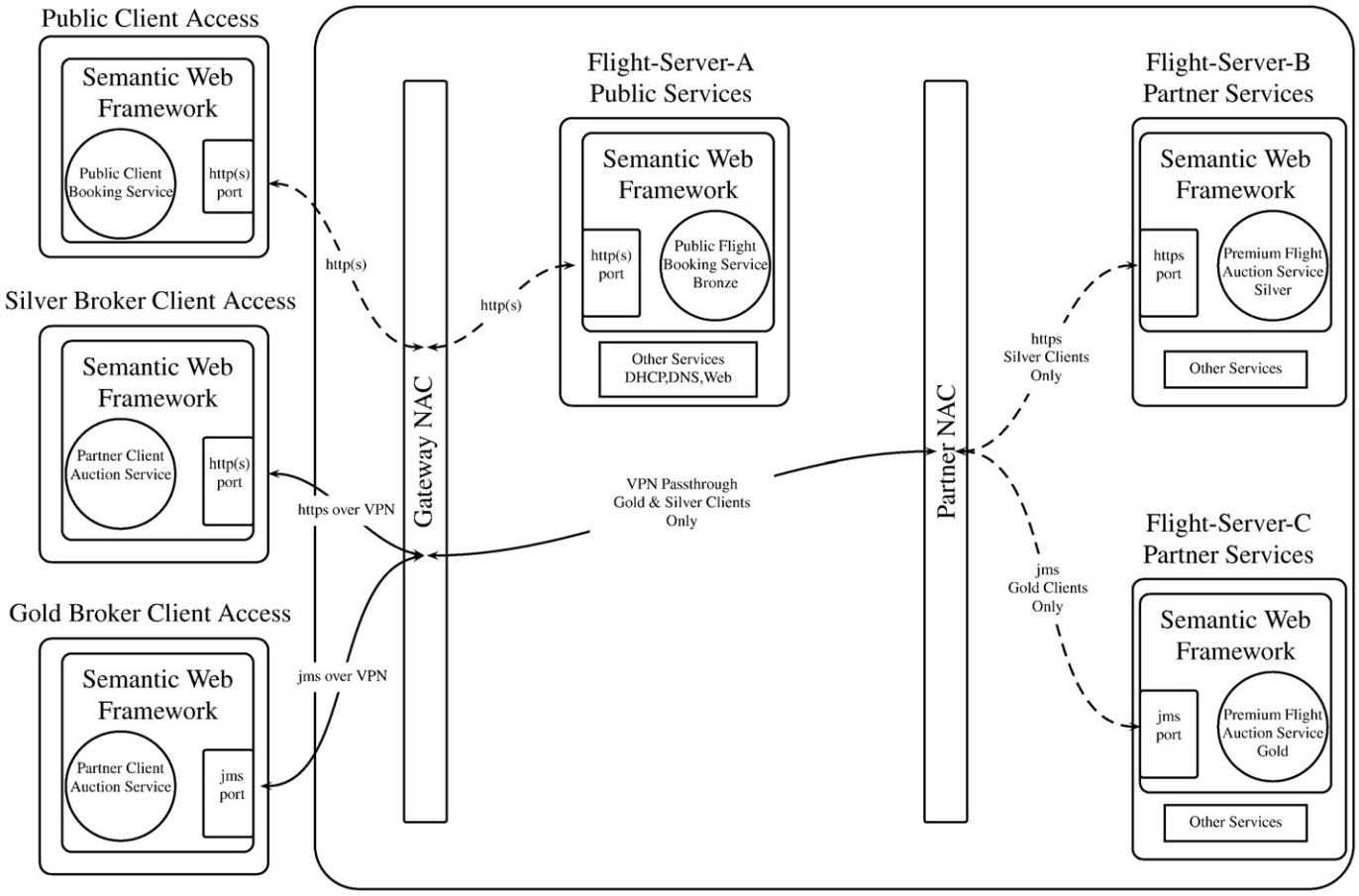


Fig. 6. E-Tourism abstract example architecture.

5. Bridging application and infrastructure

5.1. Mapping services to infrastructure

The E-Tourism application requires, among other things, network resources and associated business partners.

$$ETourismDomain \sqsubseteq \exists_{\geq 1} hasNetRequirement.NetDomain \sqcap \forall hasPartner.IPAddress$$

The kinds of network resources (class *NetDomain*) required by Semantic Web applications include (Fig. 7): hosting nodes modeled as IP addresses (class *IPAddress*), communication protocols (class *Protocol*) and listening ports (class *Port*). Service individuals and their corresponding relationships to network resources can be instantiated from the ontology model. For example, a gold service individual may have a requirement to be hosted on a particular node (ip1.2.3.4) opening a tcp communication channel listening on port 1194, (open-vpn), to provide secure access for premium business partners (example, IP address ip4.3.2.1).

$$FlightService(\text{gold}) \leftarrow hasNetRequirement(\text{gold}, \text{ip1.2.3.4}) \sqcap hasNetRequirement(\text{gold}, \text{open} - \text{vpn}) \sqcap hasNetRequirement(\text{gold}, \text{tcp}) \sqcap hasPartner(\text{gold}, \text{ip4.3.2.1})$$

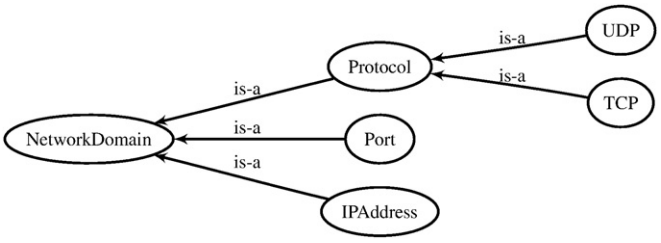


Fig. 7. Service network taxonomy (generated from protégé).

5.2. Relating services to NAC configuration

A business domain ontology is used to relate service access requirements and *NAC* rules. Fig. 8 outlines the taxonomy of the *BusinessPolicyDomain* ontology. Within this ontology, we define a class *ServicePolicy* to represent how we map semantic services to network access control:

$$ServicePolicy \sqsubseteq BusinessPolicyDomain \sqcap \exists_{\geq 1} hasManagedService.ETourismDomain \sqcap \exists_{\geq 1} hasNACrule.(NamedFirewallRule \sqcup NamedProxyRule)$$

This *ServicePolicy* class is further subdivided into three disjoint classes:

IntranetAccess, *ExtranetAccess* and *InternetAccess*. The *hasNACrule* property is further decomposed into two sub-properties: *hasFirewallRule* and *hasProxyRule* having ranges of *NamedFirewallRule* and *NamedProxyRule* respectively. An individual gep (gold Extranet policy) of the *ExtranetAccess* class that associates a gold service to an appropriate (forward) Netfilter firewall rule vpnAccess is described as follows:

$$ExtranetAccess(\text{gep}) \leftarrow hasManagedService(\text{gep}, \text{gold}) \sqcap hasFirewallRule(\text{gep}, \text{vpnAccess})$$

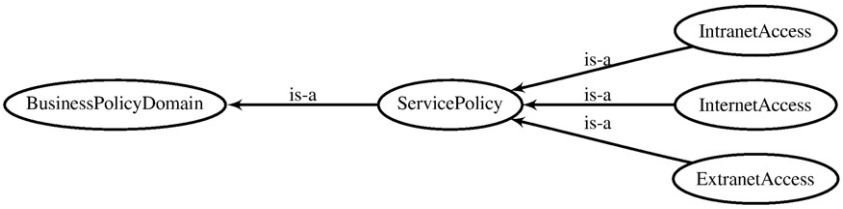


Fig. 8. Business policy taxonomy (generated from protégé).

While the *NetDomain* and *BusinessPolicyDomain* ontologies bridge firewall knowledge and application knowledge, the concrete examples (bridge configuration of individuals vpnAccess, gold and gep) are manually constructed, presumably extracted from configuration data. The following section considers how a suitable *NAC* configuration can be automatically selected from an existing knowledge base and based on an E-Tourism configuration.

6. Synthesis and analysis of NAC configuration

Configuration management of *NAC* has, to date, been complex, error-prone, costly and inefficient for many large networked organizations [27]. This challenge is further complicated by having to align the configuration with Semantic Web application access requirements.

In practice, *NAC* configuration involves, either the use of a command-line interface with esoteric command syntax, or the use of a graphical management console; either way there can be a conceptual disconnect between the application and its (NAC) environment. Our approach is to represent both the knowledge about the *NAC* configuration and the Semantic Web application in terms of a uniform ontology. In doing this, it becomes possible to reason about configurations; in particular,

- (*Analysis*) determine whether a *NAC* is aligned to the application or,
- (*Synthesis*) search for a *NAC* configuration that is aligned with the application access requirements. This search is done over a predefined catalog of best-practice *NAC* rules, rather than over the space of all possible *NAC* configurations. While this approach has the disadvantage of not discovering 'new' forms of *NAC* policy rules, it has the advantage of allowing a complete modeling of *NAC*, rather than a high-level abstraction as is used in approaches such as [6,12,13,16].

Table 1 illustrates an extract of *NAC* catalog rules of best practice. For the purposes of this paper, sample configurations were tested based on a catalog with twenty rule sets. We are currently building more extensive catalogs of best practice, for instance, a catalog of firewall rules reflecting PCI-DSS [5] best practice.

6.1. Synthesizing NAC configurations

Reasoning provides the ability to synthesize a suitable *NAC* configuration, given application and bridging configuration. While *DL*-based reasoners support inferences about classes within an ontology, *SWRL* supports reasoning about relationships between individuals. In this paper, *SWRL* is used to provide synthesis of new knowledge about configuration and its integration back into the ontology.

In this section, we consider firewall configuration synthesis. Firewall synthesis relies on the existence of a knowledge base of candidate firewall rules. These could, for example, represent considered best practice for systems that host Semantic Web applications. The synthesis process searches for firewall rules that are consistent with the configuration of the services (and bridge).

Table 1
Catalog extract of Netfilter NAC rules.

Best practice goal Sample Netfilter rule-set configuration	Description

"Drop all inward that has not been established by trusted outward traffic"
`iptables -P INPUT DROP`
`iptables -P FORWARD DROP`
`iptables -P OUTPUT ACCEPT`
`iptables -A FORWARD -i eth1 -m state --state ESTABLISHED,RELATED -j ACCEPT`
	Dynamically open ports to services on the Internet initiated by internal nodes.

"DoS rate limiting"
`iptables -A FORWARD -i eth1 -m state --state NEW -p tcp -m tcp --syn -m recent -name synflood --set`
`iptables -A FORWARD -i eth1 -m state --state NEW -p tcp -m tcp --syn -m recent -name synflood --update --seconds 1 --hitcount 60 -j DROP`
	Rate limit possible SYN based denial of service attacks.

"Log inward traffic to the firewall itself" `iptables -A INPUT -p tcp -j LOG --log-prefix "INPUT packets"`	Log direct access attempts to the firewall itself thus providing an audit trail in case of zero-day attacks and/or so that additional controls such as IPS can make use of the data.

"Drop inward traffic from blacklisted IP addresses"
`iptables -N BLACKLIST`
`iptables -A FORWARD -i eth1 BLACKLIST -j DROP`
	Deny all known malicious IP addresses. There are public repositories where these lists can be retrieved.

"Drop all traffic outward on known trojan dial-out ports"
`iptables -N TrojanDialOutPorts`
`iptables -A FORWARD -o eth0 -p udp --dport TrojanDialOutPorts -j DROP`
	Trojans often create a reverse proxy from the Intranet to the Internet as a result of most firewalls blocking inward traffic but not restricting outward traffic. Trojan port repositoryneeds to established in the user-table TrojanDialOutPorts.

For example, the following *SWRL* rule selects a (FORWARD) firewall rule given a particular Extranet service access policy that currently manages an application service.

ExtranetAccess(?*ex*) ∧ *FlightService*(?*fs*) ∧ *ForwardRule*(?*r*) ∧
hasManagedService(?*ex*, ?*fs*) ∧ *hasNetRequirement*(?*fs*, ?*reqip*) ∧
hasNetRequirement(?*fs*, ?*reqport*) ∧ *hasPort*(?*r*, ?*dstport*) ∧
hasSrcIPStartRange(?*r*, ?*sip*) ∧ *hasSrcIPEndRange*(?*r*, ?*eip*) ∧
ipAddress(?sip, ?x) ∧ ipAddress(?eip, ?y) ∧
hasTarget(?*r*, *accept*) ∧ … ∧
swrlb : *greaterThanOrEqual*(?*reqip*, ?*x*) ∧
swrlb : *lessThanOrEqual*(?*reqip*, ?*y*)
→ *hasFirewallRule*(?*ex*, ?*r*)

The *SWRL* variable ?*ex* represents a service policy that currently manages a service, (?*fs*), but has no firewall rule applied that provides it with the required network access controls. Should the firewall rule be deemed appropriate, this new knowledge can be asserted back into the ontology by setting the *hasFirewallRule* property of the *ExtranentAccess* individual ?*ex* to reflect its relationship to the firewall rule ?*r*. In practice, the above rule should contain additional filtering information to consider the protocol, interface, etc. Note the use of accept rather than a *SWRL* variable in the previous example. This is because we are interested in selecting only firewall rules that permit or accept traffic to a particular service. In practice, *SWRL* can be used to synthesize a variety of configuration scenarios. For example, *SWRL* rules can be provided, which given a specific firewall configuration can determine which services of an application can be reliably supported. This is useful when a network topology requires specific controls for systems in specific locations.

6.2. Analysing NAC configurations

SWRL also provides a SQL-like notation that provides the ability to analyze (query) a configuration in order to discover conflicts. These may be conflicts between firewall rules themselves, such as shadowing [1,7], that are independent of the application, or conflicts between the firewall rules and the access requirements of the service. In this paper, we consider the latter; the reader is referred to [7] for a discussion on using SWRL to analyze conflicts between the rules of a firewall policy.

An example of analyzing conflicts between firewall rules and application access requirements is: *Does the gold service have a firewall rule or set of rules permitting access to Extranet business partners?* This can be represented by the simple query:

ExtranetAccess(?ex) ∧ *FlightService*(gold) ∧ *ForwardRule*(?r) ∧
hasManagedService(?*ex*, *gold*) ∧ *hasFirewallRule*(?*ex*, ?*r*) ∧
hasTarget(?*r*, accept)
→ *sqwrl* : *select*(gold, ?r)

This query should return a list of tuples of the form gold ↦ ?*r*, where gold is the application service of concern and ?*r* the firewall rule that provisions network access control (as already inferred from Section 6.1). For example, it would be expected to return the tuple gold ↦ vpnAccess. The firewall rule vpnAccess was defined in Section 3.1.3. Should the above *SWRL* query return a *null* answer then the current *NAC* configuration is denying appropriate access to the gold service and thus correct alignment needs to be synthesized.

One may also want to query for possible unintended clients who can access restricted services. That is, are the firewall rules overly-promiscuous. For example, *Are the firewall rules protecting the gold service also permitting bronze grade clients?*, can be represented by the simple query:

ExtranetAccess(gep) ∧ *InternetAccess*(bip) ∧ *FlightService*(gold) ∧
FlightService(bronze) ∧ *ForwardRule*(?*ger*) ∧ *ForwardRule*(?*bir*) ∧
hasFirewallRule(gep, ?*ger*) ∧ *hasFirewallRule*(bip, ?*bir*) ∧
hasTarget(?*ger*, accept) ∧ *hasTarget*(?*bir*, accept) ∧
sameAs(?*ger*, ?*bir*)
→ *sqwrl* : *select*(?*ger*)

Should the gold service governed by the gep business policy and the bronze service governed by the bip business policy be protected by the same firewall rule or rule-set then highlight offending permissive rules.

6.3. Synthesis and analysis in practice

The Semantic Web builds upon machine interpretable ontologies in order to provide a framework for web-resource reasoning and inter-operation. However, it is important to move beyond representational issues of domain knowledge, and consider the practical applications of the Semantic Web in the context of network access control. One area of practical application is an ontology-driven *NAC* toolbox that provides *NAC* administrators with a management tool for configuring network access control. Administrators contribute new knowledge to the existing catalogs and refine analysis and synthesis based on events and experience.

Another application focuses on the use of automated semantic agents that perform various tasks on behalf of the human user [26]. These semantic agents are autonomous and possess intelligent behaviour that allow them inter-operate with one another. As a result, application-level agents can share knowledge with infrastructure level agents and vice-versa. For example, in a dynamic service deployment scenario, *NAC* agents managing (re-)configuration need to communicate with application-level agents to ensure that services that require access are permitted and service access that becomes redundant possibly due to redeployment of that service to other network tiers are prevented.

7. Tool support

The *NAC* ontologies discussed in this paper where implemented in OWL-DL, a language subset of OWL which is a W3C standard that includes *DL* reasoning semantics [22]. *Protégé* is a plug-and-play knowledge acquisition framework that provides a graphical ontology editor [10]. Protégé interfaces with a *DL*-based reasoner called *Pellet* providing model classification and consistency [19]. In conjunction to *DL* reasoning support, the *SWRL* Protégé plug-in (SWRLTab), allows for the creation of horn-like logic rules that interfaces with an expert system called *Jess* [9,18]. In practice, a domain expert using such tools can avoid or at least limit having to become an expert in *DL* and/or *OWL* notation, as these semantic editors and underlying reasoning tools hide much of the underlying complexity.

8. Related work

A risk-driven approach to the inter-operation of multiple *NAC*'s with respect to both intra- and inter-policy conflict detection was considered in [7]. As part of the *NAC* conflict analysis provided in [7], incorrect rule-set sequences are examined when searching for rule shadowing, redundancy and so forth independent of the Semantic Web application. However, the contribution of this journal paper is an approach to the alignment of *NAC* configurations with Semantic Web applications and does not consider the kind of multiple *NAC* analysis addressed in [7].

Ontology methodologies are increasingly being applied to the security domain as a whole [29,30]. However, existing applications of ontologies in the security domain have focused primarily on security classifications and tended to go no further than providing abstract taxonomy's [31,32]. In conjunction with providing an explicit specification and standard *DL* reasoning from a taxonomy perspective, our research also provides inferences at a lower level of granularity. The low-level facts of the knowledge base are presented as individuals rather than classes on the basis that they are atomic and will not be further decomposed. Using instances (rather than subclasses) allows subsequent *SWRL* reasoning, that extends the expressive capabilities

of *DL*, whereby a richer analysis of *NAC* configuration and Semantic Web application policy alignment can be conducted.

9. Discussion and conclusion

A novel approach to using a *DL* constrained ontology to represent: *NAC* (Netfilter and TCP-Wrapper), semantic-aware E-Tourism and business policy configurations were outlined. The low-level infrastructure *NAC* ontologies are used to analyse the alignment consistency of their policy configuration and to synthesize policy recommendations with respect to a high-level Semantic Web application policy.

The *NAC* ontology reflects the semantic knowledge that a network administrator should 'keep in their head' when writing and/or updating *NAC* rules based on the semantic application it offers protection to. Section 6.3 outlined practical applications whereby both a human guided and an automated intelligent agent guided response to configuring the underlying infrastructure to coincide with business demands. While not the focus on this paper, in the case of intelligent agents further constraints need to be defined within the ontology that ensure the administrator (human user) maintains control over the network. For example, it would not be desirable to have a spurious port opened by an intelligent agent just on the basis of an application demand alone. An infected network with a Remote Access Trojan (*RAT*) could exploit outgoing *NAC* rule constraints simply by demanding they be relaxed. Thus additional constraints would need to be factored into the ontology thatdo not override human defined polices. One approach is to build upon the risk-driven approach outlined in [7] where administrator policy priorities can be adhered to. Another solution might perhaps involve a semantic trust management framework between intelligent agents making network related decisions [30].

We provided an E-Tourism case study to illustrate how Semantic Web applications cannot operate in a standalone manner when incorporating security features because there are situations where the infrastructure configuration (firewalls, Web proxies and so forth) may work against the normal operation of the semantic application. The alignment of Semantic Web applications with network access controls was demonstrated by bridging application requirements to network infrastructure in conjunction with business service access policy constraints so that various NAC configuration recommendations could be inferred. The business policy ontology, through the use of *SWRL* inferencing, facilitates access control alignment between a catalog of appropriate *NAC* (firewall & proxy) rules and various service grades within the E-Tourism application.

The advantage of taking a formal ontological approach is that it provides a basis for extendability, interoperability and complex composition of other security domains of interest based on the principles of Open World Assumption (*OWA*). By developing new ontologies, for example, Intrusion Detection System, *IDS*, one can then model and reason about configurations that involve NAC configurations and IDS configurations.

Acknowledgements

This research was funded by the Science Foundation Ireland under the following grants, AMCNS:04/IN3/I404C and FAME:08/SRC/11403.

Appendix A. Description logic

DL belongs to a family of logic that represents a decidable portion of first-order logic. The logic is characterised by a set of constructors (Table 2) that allows the construction of complex concepts and roles from atomic concepts or roles. Classes (concepts) represent sets of individuals and properties (roles) represent binary relations applied to individuals. Tables 2 and 3 illustrate parts of *DL* that are used in this paper. The reader is referred to [3] for further information.

Table 2
DL constructors.

Constructor	DL syntax	Example
Intersection Of	$C_1 \sqcap \ldots \sqcap C_n$	*Protocol* ⊓ *Port*
Union Of	$C_1 \sqcup \ldots \sqcup C_n$	*PrivilegedPort* ⊔ *UnprivilegedPort*
Complement Of	$\neg C$	¬*PrivilegedPort*
Universal Quantifier	$\forall P.C$	∀*hasPort.PrivilegedPort*
Existential Quantifier	$\exists P.C$	∃*hasPort.PrivilegedPort*
Max Cardinality	$\leq_n P$	$\leq_{16}$ *hasPort*
Min Cardinality	$\geq_n P$	$\geq_1$ *hasPort*
Exact Cardinality	$=_n$ P	$=_1$ *hasPort*

Classes and properties

Classes are interpreted as sets of individuals and can be organized into a superclass–subclass hierarchy. For example, *Protocol* is a class that represents the set of all individual protocols and its subclasses include *TCP* and *UDP* classes. Subsumption represents the superclass–subclass hierarchy, for example, *TCP* ⊑ *Protocol* indicates that *TCP* is a subclass of *Protocol*.

Properties are used to construct binary relationships between classes. They are used when making statements about classes. For example, the following defines the concept of "ports are either privileged or unprivileged but not both":

$$Port \equiv PrivilegedPort \sqcup UnPrivilegedPort$$
$$PrivilegedPort \sqsubseteq \neg UnPrivilegedPort$$

Like classes, sub-properties specialise their superproperties. For example, the property *hasSrcPort* specialises the property *hasPort*. This states that if two classes are related by the *hasSrcPort* property then an attributed source port is a more specific relationship than the general case of having a port relationship. Properties in our model are prefixed with the word "*has*". *DL* has the following properties: functional, inverse functional, transitive and symmetric.

Property restrictions

Object property restrictions are used to create constraints on individuals that belong to a particular class. Restrictions fall into three categories: *Quantifier*, *Cardinality* and *has-Value* restrictions. An existential (∃) restriction requires *at least one* relationship for a given property to an individual that is a member of a specific class. A universal (∀) restriction mandates that the *only* relationships for the given property that can exist must be to individuals that are members of the specified class. A property restriction effectively describes an anonymous or unnamed class that contains all the individuals that satisfy the restriction. When restrictions are used to describe classes they specify anonymous superclasses of the class being described.

Partial and complete classes

A *partial* class definition is specified with *necessary conditions* and is of the form *Class* ⊑ *SuperClass* ⊓ *PropertyConditions*. This states that if an individual is a member of the defined class it must satisfy the conditions that it characterises. However, it cannot be said that any (random) individual that satisfies these conditions must be a member of this class. When a class is defined with *necessary and sufficient conditions* (≡), like

Table 3
DL axioms.

Axiom	DL syntax	Example
SubClass Of	$C_1 \sqsubseteq C_2$	*TCPParam* ⊑ *TCPFlag* ⊓ *Port*
Equivalent Class	$C_1 \equiv C_2$	*Port* ≡ *PrivilegedPort* ⊔ *UnPrivilegedPort*
Disjoint With	$C_1 \sqsubseteq \neg C_2$	*PrivilegedPort* ⊑ ¬*UnPrivilegedPort*
Sub Property OF	$P_1 \sqsubseteq P_2$	hasSrcPort ⊑ hasPort

partial classes, then if an individual is a member of the class then it must then satisfy those conditions. However, with the *sufficient condition* included, then any (random) individual that satisfies these conditions must be a member of this class. Classes that have at least one set of *necessary and sufficient conditions* are known as *complete* classes.

Open World Assumption

The Closed World Assumption (*CWA*) and the Open World Assumption (*OWA*) represent two different approaches of how to evaluate implicit knowledge in a knowledge base [3]. The CWA approach makes the presumption that what is not currently known to be true in a knowledge base is false, hence the interpretation of negation as failure. However, OWA assumes that its knowledge of the world is incomplete. If something cannot be proved to be true in the known knowledge base, then it does not automatically become false. A simple example of OWA would be to assume that we know that a firewall rule has been applied to a range of privileged ports and from this information using the OWA approach one cannot conclude that a firewall rule also has or has not some unprivileged ports assigned. Hence, if a class is to be confined to certain constraints, it must be explicitly stated that other unwanted constraints do not exist. In *DL*, OWA characteristics can be contained by stating exactly the components of a class using both *disjointness* and *covering axioms*.

References

[1] Ehab Al-Shaer, Hazem Hamed, Raouf Boutaba, Masum Hasan, Conflict classification and analysis of distributed firewall policies, IEEE Journal on Selected Areas in Communications, vol. 1-1, 2005, p. 1.

[2] H.Peter Alesso, Craig F. Smith, Thinking on the Web: Berners-Lee, Gödel and Turing, Wiley-Interscience, September 2006.

[3] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, Peter Patel-Schneider, The Description Logic Handbook: Theory, Implementation and Applications, Cambridge University Press, March 2003.

[4] Jorge Cardoso, Semantic Web Services: Theory, Tools and Applications, IGI Global, March 2007.

[5] PCI DSS, Payment Card Industry (PCI) Data Security Standard, PCI Security Standards Council, version 1.1, 2006.

[6] Pasi Eronen, Jukka Zitting, An Expert System for Analyzing Firewall Rules, 6th Nordic Workshop on Secure IT Systems, 2001, pp. 100–107.

[7] William M. Fitzgerald, Simon N. Foley, Mícheál Ó. Foghlú, Network Access Control Interoperation using Semantic Web Techniques, 6th International Workshop on Security In Information Systems (WOSIS 2008), Barcelona, Spain, June 12–13, 2008, 2008.

[8] Simon N. Foley, William M. Fitzgerald, Semantic Web and Firewall Alignment, 1st International Workshop on Secure Semantic Web (SSW'08), April 7–12, 2008, IEEE CS Press, Cancun, Mexico, 2008.

[9] Ernest J. Friedman-Hil, Jess the Rule Engine for the Java Platform, Version 7.0p1, 2006.

[10] John H. Gennari, Mark A. Musen, Ray W. Fergerson, William E. Grosso, Monica Crubézy, Henrik Eriksson, Natalya F. Noy, Samson W. Tu, The Evolution of Protege: An Environment for Knowledge-Based Systems Development, Proceedings of International Journal of Human-Computer Studies, vol. 58, 2003.

[11] Lucian Gheorghe, Designing and Implementing Linux Firewalls with QoS using netfilter, iproute2, NAT and l7-filter, PACKT Publishing, October 2006.

[12] Joshua D. Guttman, Filtering Postures: Local Enforcement for Global Security Policies, IEEE Symposium on Security and Privacy, Oakland, 1997.

[13] Scott Hazelhurst, A Proposal for Dynamic Access Lists for TCP/IP Packet Filtering, South African Computer Journal 33 (2004).

[14] Horrocks Ian, Peter F. Patel-Schneider, A Proposal for an OWL Rules Language, Proceedings of the 13th international conference on World Wide Web, 2004.

[15] Naren Kodali, Csilla Farkas, Duminda Wijesekera, Enforcing Semantics-Aware Security in Multimedia Surveillance, Journal on Data Semantics II 3360 (2005) 199–221.

[16] Robert Marmorstein, Phil Kearns, A Tool for Automated iptables Firewall Analysis, USENIX Annual Technical Conference, FREENIX Track, 2005.

[17] Srijith K. Nair, Bruno Crispo, Andrew S. Tanenbaum, Zodac—Towards a Secure Policy Enforcement Architecture, Fourteenth International Workshop on Security Protocols, March 27–29, 2006.

[18] Martin O'Connor, Holger Knublauch, Samson Tu, Benjamin Grosof, Mike Dean, William Grosso, Mark Musen. Supporting Rule System Interoperability on the Semantic Web with SWRL. Fourth International Semantic Web Conference (ISWC2005), Galway, Ireland, 2005.

[19] Bijan Parsia, Evren Sirin, Pellet: an OWL DL Reasoner, 3rd International Semantic Web Conference ISWC, 2004.

[20] Rusty Russell, Linux 2.4 Packet Filtering HOWTO, January 2002, www.netfilter.org.

[21] A.F. Salam, Jason R. Stevens, Semantic Web Technologies and E-Business: Toward the Integrated Virtual Organization and Business Process Automation, IGI Global, January 2007.

[22] Michael K. Smith, Chris Welty, Deborah L. McGuinness, OWL Web Ontology Language Guide, W3C Recommendation, Technical Report, 2004.

[23] Steve Suehring, Robert L. Ziegler, Linux Firewalls. Third Edition, Novell Publishing, 2006.

[24] Wietse Venema, TCP WRAPPER Network Monitoring, Access Control, and Booby traps, Mathematics and Computing Science, Eindhoven University of Technology, The Netherlands, 1992.

[25] John Wack, Ken Cutler, Jamie Pole, Guidelines on Firewalls and Firewall Policy: Recommendations of the National Institute of Standards and Technology, NIST, Special Publication 800-41, 2002.

[26] Christopher Walton, Agency and the Semantic Web, Oxford University Press, USA, December 2006.

[27] Wool Avishai, A Quantitative Study of Firewall Configuration Errors, COMPUTER, vol. 37, IEEE Computer Society Press, 2004, pp. 62–67.

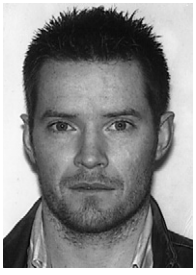
[28] He Hao, Haas Hugo, Orchard David, Web Services Architecture Usage Scenarios, W3C Working Group Note 11, 2004.

[29] Carlos Blanco, Joaquin Lasheras, Rafael Valencia-García, Eduardo Fernández-Medina, Ambrosio Toval, Mario Piattini, A Systematic Review and Comparison of Security Ontologies, Third International Conference on Availability, Reliability and Security, 2008.

[30] Thuraisingham Bhavani, Building Trustworthy Semantic Webs, Auerbach Publications, 2008, ISBN-13:978-0849350801.

[31] Kim Anya, Luo Jim, Kang Myong, Security Ontology for Annotating Resources, 4th International Conference on Ontologies, Databases, and Applications of Semantics, (ODBASE), Agia Napa, Cyprus, 2005.

[32] Almut Herzog, Nahid Shahmehri, Claudiu Duma, An Ontology of Information Security, International Journal of Information Security and Privacy (IJISP) 1 (4) (2007) 1–23.



William Fitzgerald has received a MSc by research (2002) and BSc honours degree (2000) from National University of Maynooth. He is currently employed as a research assistant at University College Cork (UCC) and is pursuing a PhD in systems security under the direction of Dr. Simon Foley. His core competency is in the area of network access control policy configuration using Ontology Engineering. William has completed a three year internship (Jan. 2006–Dec. 2008) in collaboration with UCC at the Telecommunications Software & Systems Group (TSSG), Waterford Institute of Technology. His fourth and final PhD year will be conducted at the Cork Constraint Computation Centre (4C). Prior to commencing a PhD, William was a researcher at TSSG (2004–2006) and while there he has collaborated on a number of European FP6 and FP7 IST projects, for example: SecurIST and Daidalos. He previously held a short-term research appointment at Ericsson, Ireland (2003). A detailed biography is available at: http://www.williamfitzgerald.net.

Simon Foley is a Statutory Lecturer in Computer Science at University College Cork where he teaches and directs research on computer security. He is on the editorial board of the Journal of Computer Security and has served as Program chair of the IEEE Computer Security Foundations Workshop and the ACM/ACSAC New Security Paradigms Workshop. His research interests include security modeling, distributed authorization and enterprise risk management.