Security Pattern Catalog Schema. Common description.

(schema/SecurityPatternCatalogNaiveSchema.owl)
brazhuk@grsu.by

The Security Pattern Catalog Schema ontology is a model of security patterns. It contains concepts and individuals, used for creation and maintenance of security pattern catalogs.

The key points are:

-Using the ontology it is possible to represent different types of patterns (security patterns, as well as misuse patterns and common threats). Also, relations between patterns can be depicted (for complex patterns it shows included patterns, related patterns, and you can build hierarchies of patterns).

-It allows to put a pattern into a context, i.e. add labels that support answering the decision questions like “Is a pattern suitable for a system design?”, and “Does the pattern solves a security challenge?”. It is useful for security analysis of a design of a computer system.

-The pattern contextualization is based on a decomposition of system design to components and functions. For a particular domain (i.e Cloud Computing or Internet of Things) it requires a model that describes domain-specific components and functions (further research - creation models for different domains).

-For the security contextualization it is proposed to use security concerns and threats. Security concerns are considered as security metrics in terms of software requirements (useful for developers) rather the perspective of an attack (further research – creation of an adequate model of security concerns). And a traditional approach for security experts is involved with threats as possibilities to implement some attack.

Property	*Range and predefined values*	*Description*	*Example*
Class: **Pattern**			*pattern_SecureDistributedPublishSubscribe IoT*
Metadata: The metadata section contains properties, used to identify a particular pattern, its authors and an idea. Also, it holds links to an original documents. The most valuable thing is an original pattern's document, so it should be a way to get access to it, locally or remotely. And in some cases it is impossible to put a full description of the pattern because of copyright and trademark, so the "textreview*" fields retell a content of the pattern in own words.			
textName	**xsd:String**	Pattern’s primary name	*Secure Distributed Publish/Subscribe (P/S) pattern for IoT*
textAKAName	**xsd:String**	Pattern’s alternative name(s)	
textAuthor	**xsd:String**	Pattern’s author(s)	*Eduardo B. Fernandez* *Nobukazu Yoshioka* *Hironori Washizaki*
textURL	**xsd:String**	URL(s) of webpage that describes a pattern	*https://www.researchgate.net/publication/339103887_Secure_Distributed_PublishSubscribe_PS_pattern_for_IoT*

textPDF	**xsd:String**	Downloadable URL(s) of pattern's PDF	
textReference	**xsd:String**	A bibliographic description of a primary paper, describing a pattern	*E.B.Fernandez, N. Yoshioka, H. Washizaki, "Secure Distributed Publish/Subscribe (P/S) for IoT, 2020. Procs. Asian PLoP'20, March 4 6, Taipei, Taiwan. 9 pages.*
textIntent	**xsd:String**	Pattern's Intent (full text)	*In an IoT system, decouple the publishers of events from those interested in the events (subscribers). Subscription and publication are performed securely.*
textReviewContext	**xsd:String**	A brief text description of pattern's context	Something like that: *Information exchange between IoT/IIoT devices (e.g. smart thermostats or sprinkler systems, different sensors) with minimal security control and cloud/fog applications.*
textReviewProblem	**xsd:String**	A brief text description of pattern's problem	Something like that: *Subscribers (S) register and receive messages of their interest sent by a publisher (P). The main concerns are how to organize the interactions between them securely, avoiding rogue participants, insecure communications, unwanted P/S operations.*
textReviewSolution	**xsd:String**	A brief text description of pattern's solution	*In addition of the standard P/S functions it is possible to use secure channels for protected communications, access control for restricting the actions of publishers and subscribers, security logging, and digital signatures.*
Organizational and scope aspects: This section describes organizational aspects of a pattern (type, base template) and its scope (i.e. relations to other patterns - "hasGroup", "usesPattern", "relatesTo", "isChildOf). Considered pattern should be described carefully without worrying about dependent patterns, because the automatic reasoning procedures allow to create a full "network" of pattern relations, thanks to the inverse and symmetric properties.			
hasType	**Type** Predefined items:	Type of a pattern, like security pattern, misuse pattern, or threat pattern.	The first option is to tell it is an instance of the *Pattern* class and:

	type_SecurityPattern **type_ThreatPattern** **type_MisusePattern** Class: SecurityPattern (Defined class) Class: ThreatPattern (Defined class) Class: MisusePattern (Defined class)	It is possible to define this with the class assertion, e.g. <pattern> is an instance the SecurityPattern class.	*hasType value type_SecurityPattern* The second option is to tell it is an instance of the *SecurityPattern* class
hasTemplate	**Template** Predefined items: **template_POSA** **template_GOF** template_ESP ???	Template, used to describe a pattern. It can be POSA or GOF. POSA stands from “Pattern Oriented Software Architecture”. GOF stands from “Gang of Four”.	*hasTemplate value template_POSA*
hasGroup (inverse: isGroupOf)	**Group**	Tells to which group a pattern belongs to. It can be possible to use the class assertion here, e.g. create a hierarchy with abstract patterns on the top and concrete ones at the bottom.	*hasGroup value* *patterngroup_SecureMiddleware*
usesPattern (inverse: isUsedBy)	**Pattern**	Enumerates patterns that are used by this one. For the POSA template it should be taken from "Description" and "Class Diagram".	*pattern_RoleBasedAccessControl* *pattern_Authenticator* *pattern_SecurityLoggerAuditor* *pattern_SecureChannel*
relatesTo (symmetric)	**Pattern**	Enumerates patterns that are related to this one. For the POSA template it should be taken from “Related patterns”.	*pattern_SecurePS* *pattern_Broker* *pattern_SecureChannel* *pattern_EnterpriseServiceBus* *pattern_Authorizer* *pattern_IoTSegmentation*
isChildOf (inverse: isParentOf)	**Pattern**	For a concrete pattern shows from which abstract pattern it has been made. It can be possible to use the class assertion here, e.g. create a hierarchy with abstract patterns on the top and concrete ones at the	*IsChildOf value* *pattern_SecurePublishSubscirbe*

		bottom.	
Common characteristics: This section contains common labels, used to characterize a pattern [VanHilst, 2009], [Guan, 2016], [Vale, 2019] .			
hasDomain	**Domain** Predefined items: **domain_FogComputing** **domain_EdgeComputing** **domain_InternetOfThings** **domain_SCADA** **domain_Military** **domain_ECommerce** **domain_GridComputing** Class: CloudComputingDomain ***domain_CloudComputing*** **domain_IaaS** **domain_PaaS** **domain_SaaS** **domain_NFV**	Tells to which domain(s) a pattern belongs to. Domain is a large functional field of Information Technologies (IT), like Cloud Computing, Internet of Things. It might be less gigantic, like IaaS or NVF.	*hasDomain value domain_InternetOfThings*
hasArchitecturalLayer	**ArchitecturalLayer** Predefined items: Class: ApplicationArchitecturalLayer **al_ClientLayer** **al_LogicLayer** **al_DataLayer** Class: PlatformAndOperatingSystemLayer **al_PlatformAndOperatingSystem** Class: CommunicationArchitecturalLayer **al_DistributionLayer** **al_TransportLayer** **al_NetworkLayer**	Shows a common architectural domain, to which a pattern relates, like Applications, Platform and Operating systems, also Communications. Instances are taken from [VanHilst, 2009]	*hasArchitecturalLayer value al_ClientLayer* *hasArchitecturalLayer value al_DistributionLayer*
hasConstraintLevel	**ConstraintLevel** Predefined instances: **cl_RegulatoryLevel** **cl_OrganizationalLevel** **cl_HumanLevel**	Refers to four levels of constraint: mechanism, human (operator or developer), organizational, and regulatory(Leveson, 2004). Instances are taken from [VanHilst, 2009]	*hasConstrainLevel value cl_MechanismLevel*

	cl_MechanismLevel		
hasResponseType	**ResponseType** Predefined instances: **rt_Avoidance** **rt_Deterrence** **rt_Prevention** **rt_Detection** **rt_Mitigation** **rt_Recovery** **rt_Forensics**	"The response axis based on whether or not and attack happens and the extent, from not happening at all (avoidance), to completely happened and in the past (forensics)." [VanHilst, 2009] Instances are taken from [VanHilst, 2009]	*hasResponseType value rt_Avoidance*
hasLifecycleStage	**LifecycleStage** Predefined instances: **lc_ArchitectureAndDesign** **lc_BuildAndCompilation** **lc_Implementation** **lc_Installation** **lc_Operations** **lc_Requirements** **lc_SystemConfiguration** **lc_Deployment**	Tells which which system's lifecycle stage a pattern is applicable. Frankly, most of the patterns are applicable on the Design (Architecture) stage, but it might be possible to have a few exceptions. Instances are taken from [CAPEC].	*hasLifecycleStage value lc_ArchitectureAndDesign*
hasSecurityLevel	**SecurityLevel** Predefined instances: **sl_PhysicalSecurity** **sl_PersonnelSecurity** **sl_CommunicationAndDataSecurity** **sl_OperationalSecurity**	Tells to which field of security a pattern belongs to. To review. https://en.wikipedia.org/wiki/Physical_security https://en.wikipedia.org/wiki/Secure_communication https://en.wikipedia.org/wiki/Communications_security https://en.wikipedia.org/wiki/Operations_security	*hasSecurityLevel value sl_CommunicationAndDataSecurity*
Context characteristics:			

The “context characteristics” and “security characteristics” (see below) sections include a set of labels that allow to put a pattern to a context.
Here, “context” means a possibility to use the pattern as a part of an architecture of a particular computer system.
In particular, this set of labels can be used to support a decision towards a list of patterns and a system design, like “Is the pattern suitable for the system design or not?”.

The idea of the context approach is that each computer system can be describe in two ways: with components and functions (features). Here different approaches are possible:

A) Base assertions are: ***functions*** are unique features that build a coherent model of a system, and ***components*** are common items, used by all systems.
For example, what functions make a hypervisor (IaaS component) unique? An answer might include “Management of VMs”, “VM migration”, “Virtual networking” etc. Which common components does it consist of? It might be “Hardware server”, “Operating system”, “System service”, “Network service”, “CLI interface”, “API interface”.
In many cases security problems of common components are known and described well, and security problems of functions are in focus of research of a new type of computer system. So, considering abstract (common) patterns, it is better **to describe more components and less functions**, and considering domain specific patterns, it is better **to describe more functions and less components**.
They wish an ideal model was to have well-formed hierarchy of components, and interpret the domain specific systems as combinations of these base components. It would be possible to consider only functions for domain specific systems with such model.

B) Base assertion is: function is a part of component that does not have subcomponents (i.e. functions describe non-decomposable parts of components).
For example, there is "Hypervisor", and we do not want to decompose it (e.g. to say it includes virtual machine manager, virtual network manager and virtual storage machine manager). Instead it can be said, Hypervisor is an atomic component, and it provides the functions of management of virtual machines, management of virtual networks and storages.

C) It is quite possible to use either components or functions.

hasAffectedFunction	**Function** see ***schema_functions_components.pdf***	Tells which system function(s) a pattern affects.	*hasAffectedFunction value function_DistributeEventInformation* *hasAffectedFunction value function_DistributeSensorData*
hasAffectedComponent	**Component** see ***schema_functions_components.pdf***	Tells which common component(s) a pattern affects.	*hasAffectedComponent value component_IoTApplication* *hasAffectedComponent value component_IIoTApplication* *hasAffectedComponent value component_CloudApplication*

			hasAffectedComponent value component_FogApplication
Security characteristics: After suitable patterns have been found for a system design (see above), the next step is to correlate them to security challenges. A relevant question, which this set of characteristics allows to answer, is "Does this security pattern solve a particular security problem, valuable for its context?" Sure, a final decision is the responsibility of a system architect, but the context security characteristics allow to reduce number of options and show only relevant security solutions. For the security contextualization it is proposed to use security concerns and threats. Security concerns are considered as security metrics in terms of software requirements (useful for developers) rather the perspective of an attack (further research – creation of an adequate model of security concerns). Considering a pattern it is possible to figure out a set of threats, related to this pattern (or which ones a pattern touches). The current version of the ontology uses a CAPEC based model of threats. The CAPEC approach is considered as a common accepted approach. Used here model enables automatic reasoning of extra information, like threat impact, security objectives and STRIDE labels (for further research – to build different attack and vulnerability models, based on the threat assertions).			
hasSecurityConcern	**SecurityConcern** Predefined instances: **concern_AccessControl** **concern_AttackDetection** **concern_AttackPrevention** **concern_Audit** **concern_Authentication** **concern_Authorization** **concern_Containment** **concern_Coordination** **concern_EventLogging** **concern_Identification** **concern_InformationHiding** **concern_KeyDistribution** **concern_MessageAuthentication** **concern_MessageAuthenticity** **concern_MessageIntegrity** **concern_Monitoring** **concern_ProcessIsolation** **concern_Sandboxing**	Tells which security concern(s) a pattern touches. “A security concern represents some security feature(s)” [Guan, 2016]. Instances are taken from [VanHilst, 2009], [Vale, 2019]. To review.	*hasSecurityConcern value concern_AccessControl* *hasSecurityConcern value concern_EventLogging* *hasSecurityConcern value concern_MessageIntegrity* *hasSecurityConcern value concern_SecureCommunictations*

	concern_Realibility **concern_ResourceManagement** **concern_RightsManagement** **concern_SecureDataStream** **concern_SecureCommunictations** **concern_SecureSystemIntegration** **concern_SecureSystemAdministration** **concern_SecurityArchitecture** **concern_SecurityPolicy** **concern_TrafficMonitoring** **concern_TrafficFiltration**		
hasThreat (inverse: isThreatOf)	**Threat** Predefined instances: see ***schema_threats.pdf*** Class: CommunicationsThreat **threat_ManInTheMiddle** **threat_Interception** **threat_Flooding** **threat_ContentSpoofing** **threat_IdentitySpoofing** **threat_Footprinting** **(=threat_InformationGathering)** **threat_ProtocolAnalysis** Class: SoftwareThreat **threat_SessionManipulation** **threat_AuthenticationBypass** **threat_PriviledgeEscalation** **threat_Excavation** **threat_CodeInjection** **threat_BufferManipulation** **threat_ExcessiveAllocation** **threat_ManipulationAPI** **threat_InputDataManipulation** **threat_EnvironmentManipulation** **threat_SharedDataManipulation** **threat_Malware**	Tells what threats a pattern describes with connection to component(s) and function(s), figured out on the previous stage. For security patterns defines the possible threats, met by a pattern. For attack pattern defines the possible threats, produced by an implementation of pattern. Instances are adopted from [CAPEC] (see ***schema_threats.pdf***)	“S1: An impostor impersonates a subscriber and subscribes to receive information that will be billed to somebody else or which will give her access to sensitive information.” so *hasThreat value* *threat_PriviledgeEscalation* "S2: The publisher is an impostor and collects information (and maybe money) from potential subscribers." *hasThreat value threat_IdentitySpoofing* “S3: The subscription messages are intercepted and read or modified by an attacker. The attacker may obtain in this way credit or other personal information from the subscriber.”, so *hasThreat value threat_Interception* "P2: A publisher publishes erroneous information. This action can inject data or commands to a device thus disturbing its operation;" *hasThreat value threat_ContentSpoofing* “P4. An attacker floods subscribers with fake messages thus stopping the subscribers

			from doing any useful work; this is a Denial of Service attack.", so *hasThreat value threat_Flooding*
Inferred characteristics: To do.			
hasPossibleAttack	**Attack**	Will be taken from [CAPEC] and other attacks' classifications.	
hasPossibleWeakness	**Weakness**	Will be taken from [CWE] and other weaknesses'/vulnerabilities' classifications.	
Class: **Threat** **Contains automatically assigned properties**. Automatic reasoning procedures will get them from the internal data scheme.			
hasThreatImpact (inverse: isThreatImpactOf)	**ThreatImpact** Predefined instances: see ***schema_threats.pdf*** ti_AlterExecutionLogic ti_BypassProtectionMechanism ti_ExecuteArbitraryCode ti_GainPrivileges ti_HideActivities ti_ModifyData ti_ReadData ti_ResourceConsumption ti_UnreliableExecution	Tells which negative impact(s) the threats, described by a pattern, have to component(s) and function(s). It obtains from the CAPEC attack descriptions (the hasThreat property here). It is not so far from the CAPEC/CWE approach, used to describe consequences of their attack patterns (but with some improvements). This allows to map attacks from CAPEC and weaknesses from CWE to the "Threat" items.	
hasSTRIDE (inverse: isSTRIDEof)	**STRIDE** Predefined instances: STRIDE_Spoofing STRIDE_Tampering STRIDE_Repudiation STRIDE_Information_Disclosure STRIDE_Denial_of_Service STRIDE_Elevation_of_Privilege	Tells which STRIDE item(s) a pattern touches. To do: map STRIDE & SO	
hasSecurityObjective	**SecurityObjective**	Tells which security objective(s) a pattern	

(inverse: isSecurityObjectiveOf)	Predefined instances: SO_AccessControl SO_Accountability SO_Authentication SO_Authorization SO_Availability SO_Confidentiality SO_Integrity SO_NonRepudiation	touches.	

References:

[Guan, 2016] H. Guan, H. Yang, and J. Wang, “An ontology-based approach to security pattern selection,” Int. J. Autom. Comput., vol. 13, pp. 168–182, Apr. 2016.

[Vale, 2019] A.P. Vale, E. B. Fernández, “An Ontology for Security Patterns”. Conference paper. 2019.

[VanHilst, 2009] VanHilst M. et al. A multi-dimensional classification for users of security patterns //Journal of Research and Practice in Information Technology. – 2009. – Т. 41. – №. 2. – С. 87.

[CAPEC] https://capec.mitre.org/

[CWE] https://cwe.mitre.org/