Web of Things (WoT) Security and Privacy Guidelines

W3C Editor's Draft 06 April 2020

This version:
https://w3c.github.io/wot-security/

Latest published version:
https://www.w3.org/TR/wot-security/

Latest editor's draft:
https://w3c.github.io/wot-security/

Editors:
Elena Reshetova (Intel Corp.)
Michael McCool (Intel Corp.)

Contributors:
In the GitHub repository

Repository:
We are on GitHub
File a bug
Contribute

Abstract

This document provides non-normative guidance on Web of Things (WoT) security and privacy. The Web of Things is descriptive, not prescriptive, and so is generally designed to support the security models and mechanisms of the systems it describes, not introduce new ones. However, a WoT System also has its own unique assets, such as a Scripting API and Thing Descriptions, that need to be protected and also have security and privacy implications.

We define the general security requirements for a Web of Things (WoT) System using a threat model. The WoT threat model defines the main security stakeholders, security-relevant assets, possible attackers, attack surfaces, and finally threats for a WoT System. Using this generic WoT threat model for guidance, it should be possible to select a specific set of security objectives for a concrete WoT

System implementation. We provide several examples using common scenarios based on home, enterprise, and industrial use cases. We also provide more general suggestions for the design and deployment of a secure WoT System. All recommendations are linked to the corresponding WoT threats in the generic woT threat model in order to facilitate understanding of why they are needed.

It is not the intention of this document to limit the set of security mechanisms that can be used to secure a WoT System. In particular, while we provide examples and recommendations based on the best available practices in the industry, this document contains informative statements only. Specific normative security aspects of the WoT specification are defined in the corresponding normative WoT documents for each WoT building block.

EDITOR'S NOTE

This content should be consistent with the summaries in the Security sections of other WoT documents, in particular the [WoT-Architecture] and [WoT-Thing-Description] documents.

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at https://www.w3.org/TR/.

EDITOR'S NOTE: The W3C WoT WG is asking for feedback

Please contribute to this draft using the GitHub Issue feature of the WoT Security and Privacy Considerations repository.

This document was published by the Web of Things Working Group as an Editor's Draft.

Comments regarding this document are welcome. Please send them to public-wot-wg@w3.org (archives).

Table of Contents

1. Introduction §

Security of a WoT System can refer to the security of the Thing Description (TD) itself or the security of the Thing the Thing Description describes.

1.1 Related W3C Documents §

For a general discussion of best practices for developing secure WoT Systems, as well as a set of suggested combinations of authentication mechanisms and secured protocols, see the WoT Security Best Practices document.

For details on the Web of Things architecture, please refer to the following:

- the Interest Group web site
- the Working Group web site
- the [WoT-Architecture] document,
- the [WoT-Thing-Description] document,
- the [WoT-Binding-Templates] document, and
- the [WoT-Scripting-API] document.

2. Lifecycle of a WoT Device §

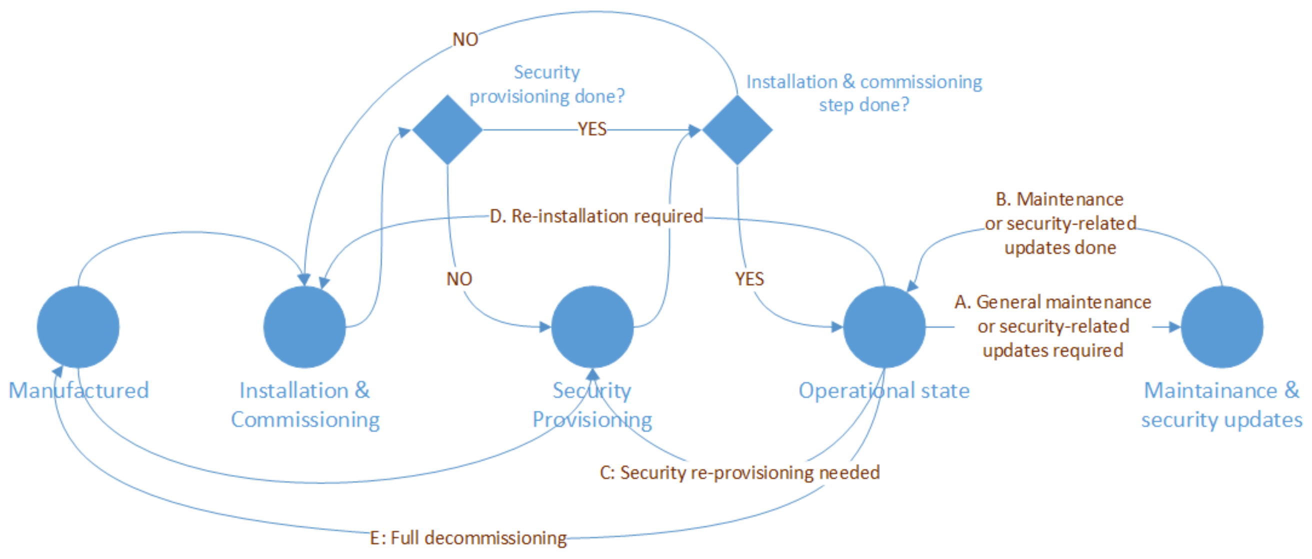

NO
Security provisioning done?
YES
Installation & commissioning step done?
D. Re-installation required
B. Maintenance or security-related updates done
NO
YES
A. General maintenance or security-related updates required
Manufactured
Installation & Commissioning
Security Provisioning
Operational state
Maintainance & security updates
C: Security re-provisioning needed
E: Full decommissioning

Figure 1 WoT Device Lifecycle

The lifecycle of a WoT Device is similar to the lifecycles of other IoT devices and services. It may in fact be identical if a WoT Thing Description is being used simply to describe an existing device implemented using another standard. However, for discussion purposes we will use the generic lifecycle shown in Figure 1 which includes the following states:

- **Manufactured:** The WoT Device has been manufactured by an OEM. It might have a WoT runtime present that enables single or multi-tenant users or it might only have network interfaces exposed present that are compatible with a WoT description. From the Manufactured state, based on the deployment scenario, the device can transition to either of two states: the Installation/Commissioning state or the Security Provisioning state.
- **Installation/Commissioning:** A WoT Device has been installed into its intended deployment environment by a system provider or system integrator and any necessary SW configuration has been performed. In practice this state may be combined with or reordered with the Security Provisioning state.
- **Security Provisioning:** A WoT Device is provisioned with the security metadata and credentials needed during its Operational state. The exact set of necessary credentials will vary based on the WoT Device's deployment model and choice of security mechanisms. The document *The State-of-the-Art and Challenges for the Internet of Things Security* [Garcia17] refers to this stage as "Bootstrapping".
- **Operational:** The default operational state of WoT Devices. Devices in this state are fully ready to communicate over WoT interfaces and satisfy their deployment's operational roles. From the Operational state a device can return to the Manufactured state if it is fully decommissioned and security de-provisioned (link E in Figure 1). It is also possible to perform only a partial reset of a device, if it only requires re-installation and recommissioning without security re-provisioning (link D in Figure 1). It is also possible that only security re-provisioning is needed without re-installation and re-commissioning (link C in Figure 1).
- **Maintenance and Security Update:** A state that is entered when the device's software or configuration needs to be updated. This is an optional state that can be activated remotely or locally (link A in Figure 1) on a WoT Device depending on the deployment model and overall setup. This state can be required to maintain security, for example to patch SW vulnerabilities or perform security-related configuration updates, such as key revocation or certificate refresh. After maintenance or updates are done, the device normally returns to the Operational state (link B in Figure 1).

The scope of this document only covers the Security Provisioning, Operational and Maintenance and Security Update States of a WoT Device.

3. Requirements §

3.1 General §

In general using the WoT approach should "do no harm": the security of any protocols should be maintained. The Web of Things does not introduce new security mechanisms, but should preserve the functionality and security of existing mechanisms. We also have the following requirements:

- **Exposing:** When exposing a TD, especially via the Scripting API, it should be possible to use best practices for security, including maintaining the integrity of the TD and delivering it only to authorized recipients.
- **Consuming:** A consumed TD should accurately reflect the actual security status of the device it describes.
- **Protocols:** The WoT can potentially apply to many protocols but to limit scope in this discussion we will focus on the needs of HTTP(S), CoAP(S), and MQTT(S).

3.2 Threat Model §

Due to the large diversity of devices, use cases, deployment scenarios and requirements for WoT Things, it is impossible to define a single WoT threat model that would fit every case. Instead we describe here an overall WoT threat model framework that can be adjusted by OEMs or other WoT System users or providers based on their concrete security requirements. This framework gives a set of possible threats that implementers should consider, but which may or may not apply to their WoT System.

When selecting suitable mitigations to the WoT threats described, it is important to consider various criteria that might increase the risk level for a particular WoT architecture, such as handing privacy-sensitive data, physical safety of individuals or the surrounding environment, high availability requirements, and so forth. In addition threat mitigations will always be limited by the capabilities of underlying devices, their list of supported security protocols, processes isolation capabilities, etc.

3.2.1 WoT Primary Stakeholders §

This section lists the primary WoT stakeholders.

Stakeholder	Description	Security goals	Edge cases

***Device Manufacturer* (OEM)**	Producer of the HW device. Implements the WoT runtime on the device, defines Things that the device provides, and generates TDs for these Things.	Don't want the devices they make to be used in any form of attack (due to loss of reputation and possible legal consequences in some countries). Want to satisfy the security requirements of System users and System providers to maximize HW sales.	
System Provider* or *System Integrator* or *System Installer	Uses devices from OEMs to build various WoT end solutions specific to particular WoT use case(s). An OEM might simultaneously be a System Provider, although System Providers might also define new TDs or modify TDs provided by other entities (OEMs). System Providers might use the WoT scripting API and runtime to implement their solution or might do a full reflash of the device to install their own software stack. A System Provider's scripts and their configuration data might require confidentiality and/or integrity. A System Integrator/Installer is similar to a System Provider but tends to reuse rather than create new technology.	Don't want their WoT System to be a target of any attacks (loss of reputation, possible legal consequences in some countries). Want to satisfy security requirements of System users for better sales. Don't want security to interfere with usability for better sales (usability vs. security). Want their WoT System to be robust and be resistant to DoS attacks. Want to hide their proprietary scripts and other relevant data from other parties.	There might be several independent system providers (tenants) on a single HW device. In that case isolation between them is required while (perhaps) sharing user preferences and identification.
System User	These might be physical users (e.g. John and his family in their smart home) or abstract users (e.g. the company running a factory). The primary characteristic of this	Don't want their data to be exposed to any other System User, System Provider or OEM unless intended (Data	Access to user data might be configurable based not

	stakeholder is the need to use the WoT System to obtain some functionality. System Users entrust WoT Systems with their data (video streams from home cameras or factory plans) or physical capabilities in the surrounding environment (put lights on in a room or start a machine on the factory line). They might differ in their access to the WoT System and transferred data (for example, they might only be able to configure certain parts of the network or only use it as it is).	Confidentiality). Don't want their data to be modified unless intended (Data Integrity).	only on "who" is the entity accessing the data (access control subject), but also his/her/its role, type of access, or time and context.
System Owner	An entity that provisions a Thing with the security root of trust and sets up the initial policies on who can provision/update/remove scripts to/from the WoT runtimes, if such model is supported. Any of the above stakeholders can be the System Owner depending on the concrete setup. A WoT System might have a number of independent System Owners, each with their own set of provisioned certificates and/or credentials. Underlying IoT protocol suites may also have their own notion of ownership.	Want to have a full control over their own provisioned security credentials, including their integrity and confidentiality.	
System Maintainer	This is an optional stakeholder. It administers a WoT System on behalf of a System Provider or System User	Needs to have enough access to the WoT System to perform its duties. Should have minimal (ideally none)	

		access to the data of other stakeholders.	

3.2.2 Assets §

This section lists all WoT Assets that are important from a security point of view.

Asset	Description	Who should have access (Trust Model)	Attack Points
***Thing Description* (TD)**	Access Control policies for TDs and the resources it describes are part of TDs. They are managed using a discretionary access control model (the owner of a TD sets the AC policy). Some contents of TDs might be privacy sensitive and therefore should not be shared without a specific need. The integrity of TDs is crucial for the correct operation of a WoT System.	TD owner: full access Others: read only for minimal required part.	Storage on thing itself, cloud storage, in-transfer (network) including TD updates.
System User Data	This includes all data that is produced or processed by the elements of a WoT System, i.e. sensor readings and their states, user configurations and preferences, actuator commands, video/audio streams, etc. Can be highly privacy sensitive and confidential.	Different System Users might have different levels of access to this data based on the use case. In some cases (like user preferences or settings) the partial access to this asset is also needed by a System Provider. Non-authorized access must be minimized since access even to a small amount of information (for example the last timestamp when a door lock API was	Storage on the device itself, remote solution provider storage (Cloud or other), in-transfer (network)

		used) might have severe privacy concerns (this example would allow an attacker to detect when the owner is away from his house). Security mechanisms should be flexible to configure and might also need to include RBAC. Others: should have no access unless specifically allowed	
System Provider Data	This includes both WoT scripts and any WoT configuration data produced and owned by a System Provider. It also includes Exposed Things, i.e. run-time representations of scripts running inside a WoT Runtime. System providers might have Intellectual Property (IP) in their scripts, making them highly confidential. Protocol Bindings might be part of System Provider Data, if a WoT Device has a single System Provider that installs and configures WoT Runtime and Protocol Bindings. However, these can be also provided and confgured by the OEM, and the same Protocol Bindings could be used by multiple System Providers on the same device.	System Provider: full access Others: no access	Storage on the device itself, runtime memory representation, in-transfer only for initial provisioning and script updates

WoT Thing Resources and WoT Infrastructure Resources	Resources should only be used for legitimate purposes and be available when required.	System User, System Provider	WoT Interface
WoT Controlled Environment	WoT Devices that have actuator capabilities are able to affect the physical environment around them. Such capability must only be used by authorized entities and for legitimate purposes. An actuator capability also implies a possible safety risk.	System User, System Provider and whoever is explicitly authorized by them to control the environment.	WoT Interface
WoT Behavior Metrics	Indirectly transmitted information (metadata) that represents measurements of a WoT System's behavior from which other information can be inferred, such as user presence. For example, the amount of communication between different things, the APIs used, distribution of communication over time, etc.	It should not be possible to collect/identify indirectly transferable information	In-transfer data and usage of WoT Interface
WoT Basic Security Metadata	All required security metadata (keys, certificates etc.) that might be provisioned to the WoT Device in order to be able to perform various security-related operations, such as authenticating another device or WoT Consumer in the WoT System, authenticating remote authorization service in case	Manager Thing (if present in WoT Runtime) - an entity that can implement script lifecycle management operations, WoT Runtime	On device storage, in-transfer including provisioning and updates of security metadata.

	the access control to WoT interfaces is done using tokens, etc.		

The term ***Thing Data*** can be used to refer to either System User Data or System Provider Data.

If a WoT System supports dynamic installation of scripts inside WoT runtime, the following assets are added:

Asset	Description	Who should have access? (Trust Model)	Attack Points
WoT Script Security Metadata	Defines who can provision, install, update, and remove scripts and other System Provider Data inside a WoT runtime.	System Provider, System Maintainer	WoT Script Management Interface

If a WoT System allows co-existence of different independent System Providers (tenants) on a single physical device, the following assets are added:

Asset	Description	Who should have access? (Trust Model)	Attack Points
WoT Runtime Isolation Policies	Access control policies for isolating co-existing system providers on the device. Not required if a simple baseline policy of "full isolation" is applied to different WoT Runtimes running scripts from different System providers. It might be required if some level of data sharing between them is needed.	System Provider, System Maintainer	Storage on the thing itself, remote storage (if they are backed up to a remote storage), in-transfer for initial provisioning

			and policy updates

3.2.3 Adversaries §

The following adversaries are in-scope for the WoT threat model:

Persona	Motivation	Attacker type
Network Attacker	Unauthorized access or modification of any stakeholder's asset over the network. Denial of service. Inferring of non-public or privacy sensitive information. Taking control of the network component of a WoT System. Reasons: money, fame etc.	Network attack: an attacker has network access to a WoT System, is able to use WoT Interface, perform Man-in-the-Middle (MitM) attacks, drop, delay, reorder, re-play, and inject messages. This attacker type also includes a passive network attacker, where an attacker is only able to passively observe the WoT System traffic.
Malicious Authorized User	Unauthorized access of private data of any other stakeholder (eavesdropping). Unauthorized modification of data of any stakeholder (modification of sensor data, or script modification). Obtaining higher privileges than originally intended by the System Owner (for example, a hotel guest trying to control overall hotel lighting).	Local physical access, WoT Consumer interface (smartphone, web interface, washing machine interface, etc.)
Malicious Unauthorized User	Similar to Malicious Authorized User, but with no prior access to the WoT System (examples: guests in a house, guests in a factory). Denial of service attacks on WoT System.	Limited local physical network access, can use proximity factor
Malware	Unauthorized access or modification	Unprivileged software adversary:

Developer	of any stakeholder's asset. Denial of service attacks on WoT System. Taking control of WoT System network components. Reasons: money, fame etc.	application intentionally gets installed or code injected into WoT Consumer (user's smartphone, browser, etc.) or in WoT runtime itself using available WoT Interfaces

The general term ***Malicious User*** may be used to refer to either a Malicious Authorized User or a Malicious Unauthorized User.

If a WoT System allows the co-existence of different independent System providers (tenants) on a single physical device, the following adversaries are added:

Persona	**Motivation**	**Attacker Type**
Malicious System Provider	Tries to get access or modify other System Provider Data, tries to access or modify System User Data from another solution provider.	Unprivileged software adversary: System provider scripts running on the same HW device, but inside a different WoT runtime

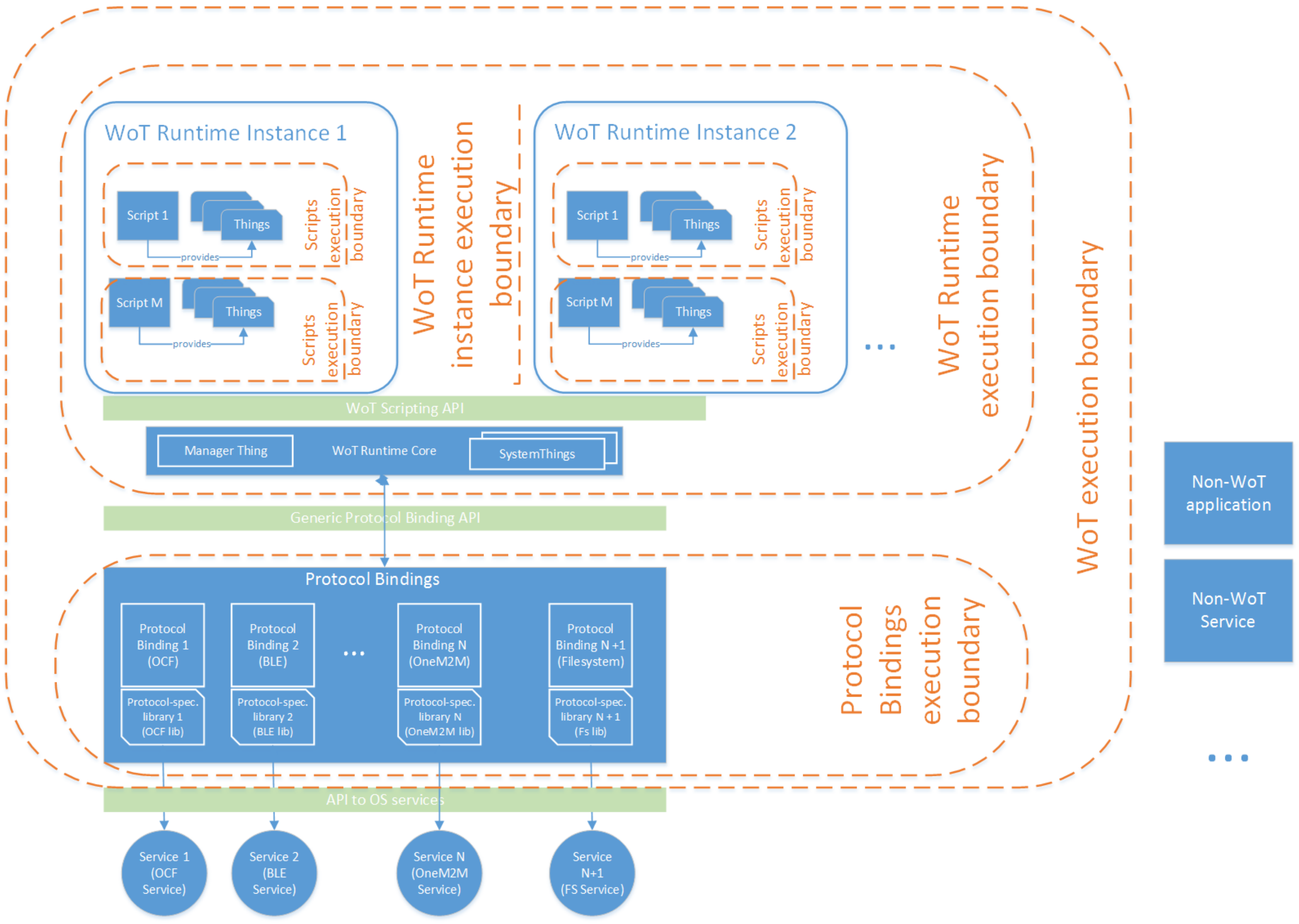
The following adversaries are out of scope of the WoT threat model:

Persona	**Motivation**	**Attacker Type**
Malicious OEM	Intentionally installs HW, firmware or other lower SW level backdoors, rootkits etc. in order to get unauthorized access to WoT assets or affect WoT System in any form	Local physical access, privileged access
Careless OEM	In order to reduce production costs substitutes security validated original HW parts with low quality unvalidated replacement parts potentially impacting security.	Local physical access, privileged access
Careless System Provider	In order to reduce solution deployment costs utilizes low quality SW, performs poor testing, misconfigures hardware or software, TDs and/or access control policies.	Local physical access, privileged access

Non-WoT End Point Attacker	Tries to get authorized access/modify any asset stored at non-WoT end points (Cloud, non-WoT Devices etc.) Reasons: monetary, fame etc.	Remote or local attack

3.2.4 Attack Surfaces §

In order to correctly define WoT attack surfaces one needs to determine the system's trust model as well as execution context boundaries. A high-level view of the WoT architecture from the security point of view is presented in Figure 2, showing all possible execution boundaries. However, in practice certain execution boundaries might not be present as they are dependent on the actual device implementation.



WoT Runtime Instance 1
Script 1
Things
provides
Scripts execution boundary
Script M
Things
provides
Scripts execution boundary
WoT Runtime instance execution boundary
WoT Runtime Instance 2
Script 1
Things
provides
Scripts execution boundary
Script M
Things
provides
Scripts execution boundary
...
WoT Runtime execution boundary
WoT execution boundary
WoT Scripting API
Manager Thing
WoT Runtime Core
SystemThings
Generic Protocol Binding API
Protocol Bindings
Protocol Binding 1 (OCF)
Protocol-spec. library 1 (OCF lib)
Protocol Binding 2 (BLE)
Protocol-spec. library 2 (BLE lib)
...
Protocol Binding N (OneM2M)
Protocol-spec. library N (OneM2M lib)
Protocol Binding N +1 (Filesystem)
Protocol-spec. library N + 1 (Fs lib)
Protocol Bindings execution boundary
API to OS services
Service 1 (OCF Service)
Service 2 (BLE Service)
Service N (OneM2M Service)
Service N+1 (FS Service)
Non-WoT application
Non-WoT Service
...

Figure 2 WoT High-Level Architecture Security View

The main execution boundaries for WoT System are:

Boundary	Description	Notes
Script Execution Boundary	Separates execution contexts between different Exposed Things. If present, this boundary should be implemented either as having different script runtime contexts (with WoT runtime isolation means) or by having separate process execution contexts (with OS process isolation means)	This boundary is required if we assume that WoT Runtime can run untrusted scripts or if some scripts have a higher chance of being compromised remotely
WoT Runtime Instance Execution Boundary	Separates execution contexts between different WoT Runtime instances. If present, this boundary should be implemented using at least with OS process isolation, but in addition can be implemented using any stronger security measures available on the device (Mandatory Access Control, OS-kernel level virtualization, Full Virtualization etc.)	This boundary is required if we assume that we have different System Providers running on the same device in different WoT Runtime Instances.
WoT Execution Boundary	Separates execution contexts between WoT runtime (including WoT Protocol Bindings) and the rest of the system. If present, this boundary should be implemented at least using OS process isolation, but in addition can be implemented using any stronger security measures available on the device (Mandatory Access Control, OS-kernel level virtualization, Full Virtualization etc.)	This boundary is required if we assume that WoT Runtime might run untrusted scripts or there might be untrusted Protocol Bindings present. It is also required if a device also runs a non-WoT SW stack and elements of that stack can potentially be compromised
WoT Runtime and Protocol Bindings Execution Boundaries	Separate execution contexts between WoT runtime and Protocol Bindings. If present, these boundaries should be implemented at least using OS process isolation.	These boundaries might be needed if we assume that a WoT Runtime can run untrusted scripts and would like to implement additional access control measures at the Protocol Binding level.

Out of above execution boundaries, the following ones become in-scope attack surfaces for the WoT security model only if we assume a WoT runtime can run untrusted scripts or we assume co-existence of different System Providers (tenants) on a single physical device:

System Element	Compromise Type(s)	Assets Exposed	Attack Method
Script Execution Boundary	Modification or unauthorized access to any asset available to other scripts. Denial of Service	TDs, System User Data, WoT Thing Resources, WoT Infrastructure Resources, WoT Controlled Environment	Local attack by an untrusted WoT script using WoT Scripting API or any native OS local method.
WoT Runtime Instance Execution Boundary	Modification or unauthorized access to any asset available to another WoT Runtime running on the same HW device. Denial of Service	TDs, System User Data, WoT Thing Resources, WoT Infrastructure Resources, WoT Controlled Environment	Local attack by an untrusted WoT runtime using any native OS local method.

While the WoT Execution Boundary is the main boundary between the WoT layer and the rest of the software on a given physical device, we don't consider it as in-scope attack surface for this threat model, because we are not able to define security measures that prevent lower layers of device's software stack to access and compromise the WoT runtime and Protocol Bindings. Such measures would greatly depend on underlying device implementation and should be considered by an OEM that produces physical device and enables it for WoT usage. Similar the WoT Runtime and Protocol Bindings Execution Boundariesare out of scope, because their presence and implementation depends on concrete architecture of WoT runtime.

In addition to attack surfaces related to the execution boundaries within a physical device, the following network-facing interfaces are in-scope attack surfaces for this threat model:

System Element	Compromise Type(s)	Assets Exposed	Attack Method
WoT Interface	Modification or unauthorized access to any affected asset, Unauthorized execution of exposed WoT Interfaces, Denial of Service	TDs, System User Data, WoT Thing Resources, WoT Infrastructure Resources, WoT Controlled Environment, WoT Behavior Metrics	Network attack
WoT	Modification or unauthorized	System User Data, WoT Thing	Network

Protocol Bindings	access to any affected asset, Unauthorized execution of exposed WoT Protocol Bindings, Denial of Service	Resources, WoT Infrastructure Resources, WoT Controlled Environment, WoT Behavior Metrics	attack

If a WoT System allows dynamic installation of scripts inside WoT runtime, the following attack surfaces are added:

System Element	Compromise Type(s)	Assets exposed	Attack Method
WoT Script Management Interface. This interface, likely in practice implemented by a manager Thing (or Thing Manager), allows dynamic installation, removal and update of WoT scripts.	Compromise of WoT scripts, including installing an older version of legitimate scripts (rollback)	System Provider Scripts (part of System Provider Data) and their execution environment	Network attack

The following attack surfaces are currently out-of-scope for the WoT Security model:

System Element	Compromise Type(s)	Assets Exposed	Attack Method	Reasoning
Thing Directory	Modification or unauthorized access to hosted TDs, Denial of Service by preventing legitimate users from obtaining TDs, Inference of non-public or privacy sensitive information based on the observation of the protocol between the WoT	TDs, history of interaction with Thing Directory	Any remote or local attack method on a Thing Directory service that results in attacker obtaining unauthorized access or the ability to modify TDs hosted by this service or to cause any type of DoS attack on the hosting service itself. In addition public hosting of TDs by a Thing Directory can have strong privacy implications and risks, as well as obtaining information on	This attack surface is out of scope because WoT specifications do not define the internal structure of Thing Directory services or their interaction protocol.

	endpoints and Things Directory.		interactions between the Thing Directory and a given WoT System setup or user	
Non-WoT endpoints	Component compromise. This includes legacy devices, remote clouds and other infrastructures, user interfaces such as browsers or applications on a smartphone, etc.)	Any type of WoT asset	Any remote or local attack method on a non-WoT endpoint that results in an attacker obtaining unauthorized access or the ability to modify any WoT asset stored at the non-WoT endpoint or to cause any type of DoS attack on WoT Thing Resources or WoT Infrastructure Resources	This attack surface is out of scope because WoT specifications do not cover non-WoT endpoints.
Native device compromise	Device compromise on all SW or HW levels below WoT Runtime.	Any type of the WoT assets	Any remote or local attack method on WoT Devices on levels below WoT runtime that results in attacker obtaining unauthorized access or the ability to modify any WoT asset stored at the WoT Runtime or to cause any type of DoS attack on WoT Thing Resources or WoT Infrastructure Resources	This attack surface is out of scope because WoT specifications do not cover levels below the WoT Runtime.

3.2.5 Threats §

This section lists all basic WoT threats with the exception of threats on the attack surfaces that we have put out of scope in the previous section.

The WoT threat model assumes that the WoT Consumer (which might reside on a non-WoT Device, such as a smartphone, a PC browser, etc.) communicates with a WoT System using standard WoT

Interfaces and WoT Protocol Bindings. In order to minimize the attack surface, it is strongly recommended that no other separate communication means is provided for this purpose. This threat model makes the assumption that this recommendation is followed. It is also assumed that WoT Protocol Bindings are trusted and cannot be installed by untrusted parties.

Threat	Adversary	Asset	Attack method and pre-conditions
WoT Protocol Binding Threat	Network Attacker, Malware Developer, Malicious Users	All WoT assets within the same execution boundary	**Pre-conditions:** General preconditions for a network attacker with access to the WoT Protocol Bindings and/or WoT Interface. In case of Malicious Authorized Solution User or Malware Developer also access to solution user credentials available on the configuration interface (smartphone, tablet etc.) **Attack method:** Any remote attack method using WoT Interface or directly using protocol binding interfaces with the purpose of compromising protocol binding component and access to all available WoT assets
WoT Interface Threat - Exposed Thing Compromise	Network Attacker, Malware Developer, Malicious Users	All available WoT assets in the same execution boundary	**Pre-conditions:** Same as for WoT Protocol Binding Threat **Attack method**: Same as for WoT Protocol Binding Threat with the purpose of compromising an Exposed Thing.
WoT Interface Threat - Unauthorized WoT Interface Access	Network Attacker, Malware Developer, Malicious Unauthorized Users, Malicious System Provider	An asset exposed via a WoT Interface: WoT controlled environment, System user data, non-	**Pre-conditions:** An attacker with an access to a WoT Interface **Attack method:** Any remote attack method using a WoT Interface with the purpose of obtaining unauthorized access to a WoT asset

		public parts of TDs, etc.	
WoT DoS Threat	Network Attacker, Malware Developer, Malicious Users	WoT Thing Resources, WoT Infrastructure Resources, Infrastructure operability	**Pre-conditions:** General preconditions for an attacker with access to a WoT Interface **Attack method:** Any attack method using a WoT Interface in order to cause Denial-of-Service (DoS) to an end thing, device, service, or infrastructure (calling WoT Interface methods that require excessive processing, sending too many messages etc.) or to disturb the service operation by intentional dropping of all or selective WoT messages
WoT Communication Threat - TD Authenticity	Network Attacker, Malware Developer, Malicious Users, Malicious System Provider	TDs authenticity in-transfer	**Pre-conditions:** General preconditions for network attacker with access to a WoT Interface **Attack method:** Any attack method on a TD in-transfer with the purpose of its modification, including rolling back to an older version of that TD
WoT Communication Threat - TD Confidentiality and Privacy	Network Attacker, Malware Developer, Malicious Users, Malicious System Provider	Confidential and privacy-sensitive parts of TDs in-transfer	**Pre-conditions:** General preconditions for network attacker with access to a WoT Interface **Attack method:** Any attack method on TDs in-transfer with the purpose of obtaining unauthorized access to non-public parts of TDs or passively observing public parts of TDs in-transfer
WoT Communication Threat - System	Network Attacker, Malware developer,	System User Data authenticity in-transfer	**Pre-conditions:** General preconditions for network attacker with access to a WoT Interface **Attack method:** Any attack method on

User Data Authenticity	Malicious Users, Malicious System Provider		solution data in-transfer with the purpose of its modification, including sequence of received data, its freshness and illegitimate replay
WoT Communication Threat - System User Data Confidentiality and Privacy	Network Attacker, Malware Developer, Malicious Users, Malicious System Provider	System User Data confidentiality in-transfer	**Pre-conditions:** General preconditions for network attacker with access to a WoT Interface **Attack method:** Any attack method on solution data in-transfer with the purpose of obtaining unauthorized access
WoT Communication Threat - Side Channels	Network attacker, Malicious users, Malicious System Provider	WoT Behavior Metrics	**Pre-conditions:** General preconditions for network attacker with access to a WoT Interface **Attack method:** Passive observation of WoT System and messages with the intention to learn behavioral and operational patterns

EDITOR'S NOTE: Policy management threats to be added

We may need to add threats related to credentials/policy management depending on how they are exposed/stored within WoT runtime via scripting. However, currently WoT scripting is (intentionally) not able to access or manage credentials; instead they must be installed using a separate management interface during the runtime's configuration.

If a WoT System allows dynamic installation of scripts inside a WoT runtime, the following threats are added:

Threat	Adversary	Asset	Attack method and pre-conditions
WoT Script	Network	Execution inside WoT	**Pre-conditions:** General

Management Interface Threat - Script installation	Attacker, Malware Developer, Malicious Users	Runtime (stepping stone to other further local attacks) and all assets within WoT Runtime	preconditions for network attacker with access to a WoT Script Management API **Attack method:** Attacker installs its own script into WoT runtime using a WoT Script Management Interface
WoT Script Management Interface Threat - WoT Runtime Compromise	Network Attacker, Malware Developer, Malicious Users	Compromise of WoT Runtime itself (stepping stone to other further local attacks) and all assets within WoT Runtime	**Pre-conditions:** General preconditions for network attacker with access to the WoT Script Management API **Attack method:** Attacker compromises WoT Runtime itself using a WoT Script Management Interface
WoT Script Management Interface Threat - DoS	Network Attacker, Malware Developer, Malicious Users	WoT Thing Resources, WoT Infrastructure Resources	**Pre-conditions:** General preconditions for network attacker with access to the WoT Script Management API **Attack method:** Attacker misuses WoT Script Management Interface to cause DoS on things or infrastructure, including deleting or corrupting WoT scripts

The general term ***WoT Script Management Interface Threat*** may be used to refer to any or all of WoT Script Management Interface Threat - Script installation, WoT Script Management Interface Threat - WoT Runtime Compromise, or WoT Script Management Interface Threat - DoS.

If a WoT System allows co-existence of different independent System Providers (tenants) on a single physical device or if WoT scripts are assumed to be untrusted, the following new threats appear in addition to the above ones:

Threat	Adversary	Asset	Attack method and pre-conditions
WoT Untrusted	Malicious System Provider,	Component compromise and	**Pre-conditions:** An attacker has full control over a script running inside a

Script Threat - Script Compromise	Malware Developer, Malicious Users (if they can install their own scripts)	all WoT assets within the same execution boundary	WoT runtime either legitimately (System Provider Script) or by compromising a System Provider Script or by installing their own script by other means. **Attack method:** Any local attack method on another script running in the same WoT runtime that results in an attacker compromising that script and getting controls over it
WoT Untrusted Script Threat - Runtime Compromise	Malicious System Provider, Malware Developer, Malicious Users (if they can install their own scripts)	Component compromise and all WoT assets within the same execution boundary	**Pre-conditions:** Same as for WoT Untrusted Script Threat - Script Compromise **Attack method:** Any local attack method on WoT runtime itself with the intent of elevating its privileges to WoT runtime level
WoT Untrusted Script Threat - Confidentiality	Malicious System Provider, Malware Developer, Malicious Users (if they can install their own scripts)	System provider data (scripts and configuration) confidentiality	**Pre-conditions:** Same as for WoT Untrusted Script Threat - Script Compromise **Attack method:** Any local attack method on another script running in the same WoT runtime that results in an attacker obtaining knowledge about a script or its configuration
WoT Untrusted Script Threat - Authenticity	Malicious System Provider, Malware Developer, Malicious Users (if they can install their own scripts)	System provider data (scripts and configuration) authenticity	**Pre-conditions:** Same as for WoT Untrusted Script Threat - Script Compromise **Attack method:** Any local attack method on another script running in the same WoT runtime that results in allowing an attacker to modifying that script or its configuration, including rolling back to an older version of the script

WoT Untrusted Script Threat - TD Confidentiality and Privacy	Malicious System Provider, Malware Developer, Malicious Users (if they can install their own scripts)	TDs confidentiality and privacy as stored on a thing	**Pre-conditions:** Same as for WoT Untrusted Script Threat - Script Compromise **Attack method:** Any local attack method on another script running in the same WoT runtime that results in an attacker obtaining unauthorized access to non-public parts of TDs or privacy sensitive parts of TDs.
WoT Untrusted Script Threat - TD Authenticity	Malicious System Provider, Malware Developer, Malicious Users (if they can install their own scripts)	TD authenticity as stored on a thing	**Pre-conditions:** Same as for WoT Untrusted Script Threat - Script Compromise **Attack method:** Any local attack method on another script running in the same WoT runtime that results in an attacker acheiving unauthorized modification of a TD, including rolling back to an older version of that TD
WoT Untrusted Script Threat - System User Data Authenticity	Malicious System Provider, Malware Developer, Malicious Users (if they can install their own scripts)	System User data authenticity as stored/processed on a thing	**Pre-conditions:** Same as for WoT Untrusted Script Threat - Script Compromise **Attack method:** Any local attack method on another script running in the same WoT runtime that results in an attacker in being able to modify System User Data, including rolling back to an older version
WoT Untrusted Script Threat - System User Data Confidentiality and Privacy	Malicious System Provider, Malware Developer, Malicious Users (if they can install their own scripts)	System User Data confidentiality as stored/processed on a thing	**Pre-conditions:** Same as for WoT Untrusted Script Threat - Script Compromise **Attack method:** Any local attack method on another script running in the same WoT runtime that results in an attacker getting unauthorized access to System User Data

WoT Untrusted Script Threat - DoS	Malicious System Provider Malware Developer, Malicious Users (if they can install their own scripts)	WoT Thing Resources,WoT Infrastructure Resourcesof another system provider	**Pre-conditions:** Same as for WoT Untrusted Script Threat - Script Compromise **Attack method:** Any local attack method with the goal of consuming things or infrastructure resources that disturb legitimate operation of co-existing System Providers.

3.3 Security Scenarios §

Here we describe some typical high-level WoT scenarios and give practical examples of the above threats for each scenario.

3.3.1 Scenario 1 - Home Environment §

In this scenario we assume a standard home environment with a WoT System running behind a firewall NAT that separates it from the rest of the Internet. However the WoT System is shared with the standard user home network that contains other non-WoT Devices that have high chances of being compromised. This results in viewing these non-WoT Devices as possible network attackers with access to the WoT System and its APIs/Protocol Bindings. Other assumptions: WoT scripts and protocol bindings are considered trusted; a single System Provider exists on physical WoT Devices; and no dynamic installation of WoT scripts is possible.

In this scenario the following **WoT Threats** are possible:

Threat name	Example(s)
WoT Protocol Binding Threat	A compromised application on a user smartphone connected to the same internal network as WoT Devices sends a malformed request to a WoT Device targeting the Protocol Bindings interface directly. This malformed request causes the respective Protocol Binding to be compromised (e.g. by exploiting a buffer-overflow bug) and the attacker is able to run code with the privileges of the Protocol Binding on the WoT Device.
WoT Interface Threat -	A compromised application on a user smartphone connected to the same internal network as WoT Devices sends a malformed request to a WoT Device

Exposed Thing Compromise	using the WoT interface. This malformed request causes the respective Exposed Thing to be compromised and attacker is able to run code with the privileges of the Exposed Thing on the WoT Device.
WoT Interface Threat - Unauthorized WoT Interface Access	A user's party guest with network access to the same internal network as WoT Devices uses his device to send a WoT interface request to a user's WoT Device for a certain resource access, for example, to get a video stream from user's video camera footage or permission to unlock some rooms in the house. Due to lack of proper authentication, he is able to access this information or execute the required action (e.g. opening the targeted door).
WoT Communication Threat - TD Authenticity	A compromised application on a user smartphone connected to the same internal network as WoT Devices listens to the network and intercepts a legitimate TD sent by one of the WoT Devices. Then it modifies the TD to state different authentication method or authority and forwards it to the intended device. The device follow the instructions in the modified TD and sends authentication request to the attacker's specified location potentially revealing its credentials, such as username and password. Similarly instead of modifying the legitimate TD, an attack might save it and use iit later on, for example when it would be updated to a newer version. Then, an attacker substitutes a new version of the TD with an older version to cause DoS to a device trying to get an access to a resource. An additional reason for using an older TD might be exposing a previously available vulnerable interface that got hidden when the TD was updated to a newer version. This can be a stepping stone to conducting other attacks on the WoT System.
WoT Communication Threat - TD Confidentiality and Privacy	A user's party guest with network access to the same internal network as WoT Devices using his device listens to the network and intercepts a legitimate TD send by one of the WoT Devices. While inspecting the TD he/she learns privacy-sensitive information about the host, such as presence of medical tracking or assistant equipment, name of healthcare provider company etc. An example of privacy threat: a compromised application on a user smartphone connected to the same internal network as WoT Devices listens for publicly broadcast TDs and sends this data to a remote attacker to build a profile of devices installed in the home and their purpose.
WoT Communication Threat - System	A user's party guest with network access to the same internal network as WoT Devices using his device listens to the network and intercepts a legitimate WoT interface request to set some settings on some actuator, for example the time

User Data Authenticity	when house doors get locked and unlocked. He then modifies the specified value to the desirable one and then forwards the request to the destination WoT Device. The destination WoT Device accepts incorrect settings. Another attack example is a replay of a legitimate WoT interface request to set a setting, for example, increase temperature of the house by a number of degrees. When repeated many times, it might not only make living environment unusable, but also might damage the heating equipment. Another attack involves replaying an old legitimate WoT interface request that attacker intercepts while visiting the user's home, such as commands to unlock doors, stop camera recordings etc. from a different time when the user wants to get authorized access to the house.
WoT Communication Threat - System User Data Confidentiality and Privacy	A user's party guest with network access to the same internal network as WoT Devices using his device listens to the network and intercepts WoT interface exchange between legitimate entities. From that exchange the guest learns privacy-sensitive information about the host, such as a data stream from his medical tracking equipment, information about user's preferences in home environment, video/audio camera stream etc.
WoT DoS Threat	A compromised application on a user smartphone connected to the same internal network as WoT Devices sends huge amount of requests to either a single WoT Device or all device available in the WoT System using directly Protocol Bindings interface or WoT interface. These requests take all the processing bandwidth of a WoT Device or WoT System and make it impossible for legitimate users to communicate with WoT Devices or devices to communicate with each other. Depending on the implementation it can also lead to the case that alarm or authentication systems might be disabled and thieves get into user's house (however systems should always implement "safe defaults" principle and in such cases keeps doors closed despite of inability to authenticate).
WoT Communication Threat - Side Channels	A compromised application on a user smartphone connected to the same internal network as WoT Devices listens to the WoT System traffic. By this passive listening, he is able to infer various forms of privacy-related information, such as WoT System configuration, WoT Devices that are present, network active and passive phase timing, etc.

3.3.2 Scenario 2 - Business/Corporate Environment §

In this scenario we assume an office building environment shared between a number of independent companies (tenants) with a shared WoT System running that controls temperature, lights, video surveillance, air etc. The companies sharing the premises do not trust each other and can be viewed as potential attackers. However, we assume that there is a trusted non-compromised System Provider that sets the WoT System through the whole building and handles the on-boarding and termination process for all building tenants (making sure that after company leaves the building, all its data is not present in the WoT System anymore). Compared to Scenario 1 above, the main challenge here is to securely share the WoT System between these companies and make sure their WoT System User Data (that can include highly confidential company information) is not exposed, tampered with, etc. Additionally one has to take into account the fact that a set of building tenants might change from time to time, including employee turnover in each company. Therefore the WoT setup must be flexible enough to start and terminate access to the WoT System promptly to both companies (entire sets of users) and individuals. Privacy is also important here, because companies need to make sure that the data about their employees or clients is well protected, including such information as employee presence. Since we assume the presence of a trusted System Provider, we can consider the WoT scripts and protocol bindings to be trusted also. Similar to Scenario 1, there is only a single System Provider present on physical WoT Devices, and no dynamic installation of WoT scripts are possible for simplicity.

In this scenario the following **WoT Threats** are possible:

Threat name	Example(s)
WoT Protocol Binding Threat	This attack is similar to the one described in Scenario 1 where a WoT System shares connectivity with other non-WoT Devices (smartphones, PCs, etc.). In this case a company's employee can send a malformed request to a WoT Device targeting a Protocol Binding interface directly. This malformed request causes the respective Protocol Binding to be compromised and attacker is able to run the code with the privileges of the Protocol Binding on the WoT Device. As a result the attacker can get any WoT asset from that device, including System User Data from another company.
WoT Interface Threat - Exposed Thing Compromise	Similar to Scenario 1, but the attack can be done by an employee with network access to the same internal network as WoT Devices.
WoT Interface Threat - Unauthorized	An employee with network access to the same internal network as WoT Devices using his device sends a WoT interface request to obtain the data from a WoT video camera device located in a meeting room of another company or

WoT Interface Access	to unlock the door to another company's office wing. Due to lack of proper authentication, he is able to access this information or execute the targeted action (e.g. opening the door and getting physical access to another company's premises).
WoT Communication Threat - TD Authenticity	Similar to Scenario 1, but the attack can be done by an employee with network access to the same internal network as WoT Devices.
WoT Communication Threat - TD Confidentiality and Privacy	If we assume that all Things and their TDs are standard to the building, then unlike Scenario 1 we can (perhaps) assume that all info in them is public (since they relate to System provider and not system users). However, if companies or employees can add their own devices to the WoT System (for instance, wearables or desk lamps) than the system may have to protect non-public TDs.
WoT Communication Threat - System User Data Authenticity	Similar to Scenario 1, but the attack can be done by an employee with network access to the same internal network as WoT Devices. In case this if the attack succeeds, an attacker may be able to control other's company WoT Devices using actuator capabilities (including unlocking doors), changing important settings (such as heating controls, disrupting a company's business activities) etc.
WoT Communication Threat - System User Data Confidentiality and Privacy	Similar to Scenario 1, but the attack can be done by an employee with network access to the same internal network as WoT Devices. In case this if an attack succeeds, an attacker may learn other's company confidential information, and get access to the privacy-sensitive information of other's company clients and employees (including, for example, employee presence information).
WoT DoS Threat	An employee with network access to the same internal network as WoT Devices using his device sends a huge number of requests to either a single WoT Device or all devices available in the WoT System using either the Protocol Bindings interface directly or the WoT interface. These requests consume all the processing or bandwidth of a WoT Device or WoT System and make it impossible for employees of other company to communicate with WoT Devices or devices to communicate with each other. Depending on the implementation it might also lead to alarm or authentication systems being disabled and allowing attackers to get into other company's premises (however

	systems should always implement "safe defaults" principle and in such cases keeps doors closed despite of inability to authenticate; but this causes the opposite problem of denying authorized users access and disrupting legitimate business).
WoT Communication Threat - Side Channels	Similar to Scenario 1, but the attack can be done by an employee with network access to the same internal network as WoT Devices.

3.3.3 Scenario 3 - Industrial/Critical Infrastructure Environment §

In this scenario we assume an industrial factory or infrastructure (power plant, water distribution system, etc.) environment that is using WoT System to monitor or perform certain automation tasks in its Operational Technology (OT) network. Compared to other scenarios above, the main challenge here is to guarantee safety and availability of the critical infrastructure. Therefore, for example, Denial-Of-Service attacks must be mitigated as well as possible. On the other hand, privacy is usually less important for this scenario.

Similar to the previous scenarios, we assume that there is a trusted non-compromised System Provider that sets up the WoT System, so we can consider all WoT scripts and protocol bindings to be trusted also. We also assume there is only a single System Provider present on physical WoT Devices and a single tenant on all system, and assume no dynamic installation of WoT scripts is possible for simplicity.

Due to the high safety and availability requirements in the industrial environment, typically the WoT System (part of a bigger OT network) won't be shared with other tenants of the same building and normally won't be shared with the general IT network of the same company. The IT network is where all other company operations are happening, for example, accounting. We also don't expect factory employees to browse the internet in the coffee break using the OT network (and ideally not the IT network; in some companies, a third network is even provided for employee personal use or for less-trusted personal devices.) However, usually some bridging must be present between the OT and IT networking in order to support monitoring requirements, so these networks are not fully isolated. In this scenario, the risk of compromised devices and applications interacting with the WoT System is limited but nonetheless present. Therefore the usual WoT threats, described in the previous scenarios, still apply, if a malicious application or device gets into the industrial OT WoT System.

Also, a factory employee with authorized access to the OT WoT System can be also viewed as a potential attacker. The additional security challenge in this case is role management, i.e. distribution of authorized accesses between the actors (factory employees, devices with actuators, etc.) in such a way that a single misbehaving actor does not have enough authorization to endanger the safety and availability of the whole infrastructure. Similar to the case in Scenario 2, the ability to promptly terminate an employees' access rights to the WoT System when necessary, and without disrupting other activities, is essential.

4. Privacy Considerations §

In this section we focus specifically on the privacy aspects that are important to keep in mind while developing and deploying WoT solutions. The threat model, described in § 3.2 Threat Model, already takes into account numerous privacy-related threats and aspects. However, this section summarizes them and gives guidelines and recommendations specifically aimed at minimizing privacy-related risks.

The following WoT privacy threats can potentially affect the privacy of WoT system users:

Privacy threat name	Details	Mitigation
Disclosing WoT Thing/Device configuration	Thing Descriptions (TDs) might leak privacy-sensitive information, such as detailed configuration of a single WoT Thing.	WoT TDs should minimize the amount of information that is publicly available about a WoT Thing. There has to be a way (if a TD is privacy-sensitive) to limit clients who can obtain a certain TD (or its privacy-sensitive parts) via authentication and authorization methods. It is also possible to expose a number of TD variants for the same WoT Thing based on the level of client's access. This recommendation corresponds to the data minimization and security (unauthorized usage, peer entity authentication and confidentiality) treat mitigation strategies outlined in [Coo13] - *Privacy Considerations for Internet Protocols*.
Disclosing	Privacy-sensitive information might	The communication between WoT

WoT System configuration	be inferred via observing the communication between a WoT endpoint (Client, Servient or Device) and a Thing Directory. For example one can learn information about the configuration of a target WoT System by observing the TDs that the target WoT System downloads from a Thing Directory. In addition, loading JSON-LD context files might also leak WoT System configuration. For example, context files may be used to allow TDs to be extended with new protocols and security schemes; loading these context files then implies the TDs uses these protocols or schemes.	endpoints and a TD distribution service (i.e. Thing Directories) should be protected against eavesdroppers using well-established security protocols. It also should not be possible for a casual observer (for example, a guest visiting a person's home with access to a private wireless access point) to query available WoT Devices (there has to be a way to limit WoT Devices' ability to respond to different broadcast events) and as a result obtaining WoT System configuration. Caching on gateways can help with the JSON-LD context files loading problem. This recommendation corresponds to the data minimization and security (unauthorized usage, peer entity authentication and confidentiality) treat mitigation strategies outlined in [Coo13] - *Privacy Considerations for Internet Protocols*.
Leaking WoT System User Data	Depending on the network topology, WoT System User Data can be transferred between the WoT Consumer and WoT Thing via many intermediate nodes, such as an Intermediary (WoT Servient) or non-WoT nodes. If these nodes can access or process the WoT system user data, they can unintentionally leak information impacting the privacy of WoT System Users. In addition, if WoT interfaces are publicly accessible (for example allowing anyone to query the state of WoT Thing), it can also leak WoT System User Data.	WoT Things should minimize publicly accessible WoT interfaces. If WoT System User Data is privacy-sensitive and travels via intermediate nodes, end-to-end encryption methods and object level security should be preferred. This recommendation corresponds to the data minimization and security (inappropriate usage, unauthorized usage and confidentiality) treat mitigation strategies outlined in [Coo13] - *Privacy Considerations for Internet Protocols*.

Tracking WoT System User	According to the WoT specification, each TD includes a unique identifier (id), but it is not specified how such identifier should be generated and the duration of its validity. Typically such identifiers are generated using various methods outlined in [Lea05] - *A Universally Unique IDentifier (UUID) URN Namespace*. If this identifier is a persistent unique identifier that is never changed during device lifecycle (immutable) or changed very rarely (for example only when a WoT System is installed and bootstrapped), then a WoT Device and a WoT System User might be uniquely identified and its communications tracked. Similar tracking can be done even without identifiers by using fingerprinting techniques based on a WoT System configuration information (which is likely to be unique).	A TD should avoid exposing publically any persistent unique identifiers (especially immutable ones) that can be used to track it. Ideally, a TD's identifier (id) should be changed periodically (potentially even every time the WoT Exposed Thing is created, if it does not infer the normal operation mode of a WoT Thing). This however conflicts with the needs of Linked Data and the registration of Things in directories and their use by other Things. A compromise needs to be struck. Mechanisms to notify legitimate users of a Thing when a Thing's identifier has changed may be considered if frequent updates are necessary. It should be at least possible to change a Thing's identifier by re-provisioning the device if necessary. Using methods to limit disclosure of WoT System configuration and TDs also helps to prevent tracking of WoT System users.

In addition to the mitigations described above, it is also important to keep System Users informed about the information that can be exposed and/or collected about them, as well as to have a way for System Users to control the degree of such exposure. This recommendation corresponds to the user participation in treat mitigation strategy outlined in [Coo13] - *Privacy Considerations for Internet Protocols*.

EDITOR'S NOTE: Links

We need to mention privacy risks associated with links. In particular, dereferencing links carries similar inferencing risks as context file dereferencing. This has been mentioned in the Security and Privacy sections of [WoT-Architecture] and [WoT-Thing-Description] but needs to be expanded upon here.

5. Security Provisioning(Onboarding) and Maintenance §

5.1 Goals and requirements §

Following the lifecycle diagram Figure 1 , the purpose of the Security provisioning is that upon its completion a WoT Device is equipped with the security metadata and credentials it needs during its Operational state. The exact set of necessary credentials will vary based on the WoT Device's deployment model and choice of security mechanisms. The document The State-of-the-Art and Challenges for the Internet of Things Security [Garcia17] refers to this stage as "Bootstrapping".

The purpose of the Security maintenance is maintaining security level set during the security provisioning phase, i.e. security-related configuration updates, such as key rotation or certificate refresh. This also includes the actions to be taken in case of credential compromise, i.e. key revocation.

Security provisioning usually consists of three main stages outlined below. The WoT adds WoT-specific stage at the end:

- **Initial enablement.** During manufacturing a device is provisioned with the necessary credentials that enable secure mutual authentication and remote attestation in the next step.
- **Mutual trust establishment.** Establishing mutual trust between the device to be provisioned and the provisioning entity. Note that the provisioning entity is not necessary (and quite commonly not) a manufacturer, but some service provider or service domain owner. This includes mutual authentication and authorization for both device and provisioning entity and potentially remote attestation of device.
- **Runtime credential provisioning.** The actual provisioning of credentials for the operational phase in a secure fashion between the provisioning entity and the device.
- **WoT specific provisioning.** Installing the provisioned secrets and made them available and ready to use on the WoT level (within the WoT Runtime)

EDITOR'S NOTE: Question1

Do we try to describe only the bootstrapping of the credentials into a WoT run time (after which it can be transparently used during operational phase) or do we go into describing of how credentials will end up in the physical device at the first place, and then how they can be securely retrieved and embedded into WoT run time?

EDITOR'S NOTE: Question2

Do we limit ourselves to the case of a single tenant (single security domain device joins and accepts credentials from a single provisioning entity) or we also take multiple tenants in the scope from beginning?

5.2 Existing approaches §

EDITOR'S NOTE: Question3

I guess we dont want to consider old style approaches as "ask user to input manually smth" or "connect to manufacturer to fetch credentials via manufacturer service"?

5.2.1 Bootstrapping Remote Secure Key Infrastructures (BRSKI) §

The draft IETF standard [Pri19]

5.2.2 Intel Secure Device Onboard (SDO) §

Intel technology overview [SDO]

6. References to Existing Security Best Practices §

Best practices in security are constantly evolving, and it is important to keep up to date. At the same time, IoT is new enough and is itself evolving rapidly enough that best practices are just now emerging and are likely to require rapid updating. Recently attempts have been made to document and categorize useful approaches to security in IoT. The following are some of the more useful points of reference:

- [Garcia17] - *State-of-the-Art and Challenges for the Internet of Things Security*:
  This IETF document, a product of the IETF T2TRG, is still in draft form but is now a candidate for ratification. It covers a range of recommendations for securing IoT systems. Readers should look for the latest version using the IETF RFC tracker.

- [IicSF16] - *The Industrial Internet of Things Volume G4: Security Framework*:
  Focuses on industrial IoT systems and use cases, and so as discussed above emphasizes safety over privacy considerations. However, this document includes many practices that are also relevant to other IoT use cases.
- [IETFACE] - *IETF Authentication and Authorization for Constrained Environments (ACE)*:
  Discusses the use of existing IETF standards for constrained environments (such as CoAP and DTLS) of relevance to IoT. This working group defines building blocks for IoT security esp. addressing authentication and authorization.
- [ISF17] - *IoT Security Foundation Best Practice Guidelines*:
  A general set of recommendations for managing the security of IoT systems, including lifecycle management.

Here are some additional general references for threat modeling and security architecture planning. These frameworks can be helpful when designing the security architecture of a specific IoT system or standard. OWASP in particular is useful for Things using the HTTP protocol for their network interface.

- [Mic17] - *STRIDE - Internet of Things security architecture*
- [Nis15] - *NIST Guide to Industrial Control Systems (ICS) Security*
- [Owa17] - *OWASP Threat Risk Modeling*

The following documents define the security and privacy considerations that should be included in internet standards. These references helped define the topics covered in this document.

- [Coo13] - *Privacy Considerations for Internet Protocols*
- [Res03] - *IETF Guidelines for Writing RFC Text on Security Considerations*

EDITOR'S NOTE: Elaborate on additional references and/or cull

The references below are relevant, but the text should explain them more, categorize them, and put them in context. Also, we don't necessarily need all of these, and may need others not listed.

Additional references:

- [CoRE-RD] - *CoRE Resource Directory*
- [Bel89] - *Security Problems in the TCP-IP Protocol Suite*
- [Bel13] - *Web Security in the Real World*

- [Ber14] - *Authentication Protocols, Web UX and Web API*
- [Bor14] - *Terminology for Constrained-Node Networks*
- [Bru14] - *Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations*
- [Dur13] - *Analysis of the HTTPS Certificate Ecosystem*
- [Ell00] - *Ten Risks of PKI: What You're not Being Told about Public Key Infrastructure*
- [Fu01] - *Dos and Don'ts of Client Authentication on the Web*
- [Geo12] - *The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software*
- [Gol03] - *Cryptography and Cryptographic Protocols*
- [Gre14] - *How do you know if an RNG is working?*
- [Gut02] - *PKI: It's Not Dead, Just Resting*
- [Hea13] - *An update on our war against account hijackers*
- [Iic15] - *Industrial Internet Reference Architecture*
- [IicRA17] - *The Industrial Internet of Things Volume G1: Reference Architecture*
- [Jon14] - *A JSON-Based Identity Protocol Suite*
- [Ken03] - *Who Goes There? Authentication Through the Lens of Privacy*
- [Lam04] - *Computer Security in the Real World*
- [Loc05] - *Demystifying SAML*
- [Mel15] - *Securing the Industrial Internet of Things*
- [Moo02] - *A critical review of End-to-end arguments in system design*
- [Oos10] - *Provisioning scenarios in identity federations*
- [Sch14] - *The Internet of Things Is Wildly Insecure — And Often Unpatchable*
- [Sch99] - *Breaking Up Is Hard To Do: Modeling Security Threats for Smart Cards*
- [She14] - *The Constrained Application Protocol (CoAP)*
- [Vol00] - *AAA Authorization Framework*
- [Yeg11] - *Stevey's Google Platforms Rant*

EDITOR'S NOTE: Use global database for RFC references

Should use references from global ReSpec bibliography whenever possible, rather than local bibliographic entries.

7. Best Security Practices and Mitigations for WoT Systems §

Based on the main WoT assets, privacy and security threats listed in § 3.2.2 Assets, § 3.2.5 Threats and § 4. Privacy Considerations, we provide here a set of recommended practices for enhancing security and privacy when designing a WoT System. Depending on the usage scenario, these may not be adequate for securing any given WoT System, however. The full threat model should be considered to identify and mitigate any additional threats for a specific WoT System.

7.1 Secure Practices for designing a Thing Description §

Secure Delivery and Storage of Thing Description.

When a TD is transferred between WoT endpoints, to or from the TD hosting service, it is important to use secure protocols guaranteeing data authenticity, freshness and in many cases confidentiality (depending on whenever a TD contains any confidential or privacy-sensitive information). TDs should not be provided without first authenticating the requester and checking that they have authorization to access the requested TDs. If end-to-end TD authenticity or confidentiality is required, see § 7.4 Secure Practices for end-to-end security for more specific guidance. When a TD is stored at the end device, in a gateway (e.g. in a cache or directory service) or in remote storage (e.g. an archive), its authenticity and in some cases confidentiality should also be protected using the best available local methods.

This recommendation helps prevent the following threats: WoT Communication Threat - TD Authenticity, WoT Communication Threat - TD Confidentiality and Privacy, Disclosing WoT Thing/Device configuration, Disclosing WoT System configuration, Tracking WoT System User.

Avoid Exposing Persistent Unique Identifiers

When defining fields exposed by a TD immutable information should be avoided, especially if that information can be tied to a particular person or is associated with personally identifiable information. It is specifically strongly recommended to avoid exposing any immutable hardware identifiers. Instead it is recommended to use "soft identifiers", i.e. identifiers that can be changed at some point during the device's lifecycle. Given the requirements of Linked Data, it may not be possible to change the identifier during the Operational state of a device without additional infrastructure to register consumers of a TD and notify them of updates, but it should at least be possible during (re-)provisioning.

This recommendation helps prevent the following threats: WoT Communication Threat - TD Confidentiality and Privacy, Disclosing WoT System configuration, Tracking WoT System User.

Minimize Publicly Exposed Information in TD

WoT TDs should minimize the amount of information that is publicly available about a WoT Thing. For example, the TD should generally not identify the version of the software or operating system a device is running, since this information can be used to identify vulnerable systems. Instead the TD should capture all information needed to support interoperability without reference to particular implementations.

This recommendation helps prevent the following threats: WoT Communication Threat - TD Confidentiality and Privacy, Disclosing WoT Thing/Device configuration, Disclosing WoT System configuration, Tracking WoT System User, WoT DoS Threat.

Limit TDs exposure

Limit clients who can obtain a certain TD (or its privacy-sensitive parts) via authentication and authorization methods. This can be done by exposing a number of TD variants for the same WoT Thing based on the level of client’s access. For example, versions of a TD for "public access" may be generated with a subset of information in the full TD, omitting (for example) support or version metadata.

This recommendation helps prevent the following threats: WoT Communication Threat - TD Confidentiality and Privacy, Disclosing WoT Thing/Device configuration, Tracking WoT System User, WoT DoS Threat.

7.2 Secure Practices for protecting System User Data §

Use Secure Transports

When defining protocols for APIs exposed by a TD, it is often important to use secure protocols guaranteeing System User Data authenticity and confidentiality. Specifically, HTTP-over-TLS (HTTPS), CoAP-over-DTLS (COAPS), and MQTT-over-TLS (MQTTS) should be used if any kind of authentication or secure access is required. If data travels over many intermediate nodes (potentially malicious or not privacy-preserving), end-to-end encryption methods should be preferred.

This recommendation helps prevent the following threats: WoT Communication Threat - System User Data Authenticity, WoT Communication Threat - System User Data Confidentiality and Privacy, Leaking WoT System User Data.

User Consent

Keep System Users informed about the information that can be exposed or collected about them. Provide a way for System Users to control the degree of such exposure.

This recommendation helps prevent the following threats: WoT Communication Threat - System User Data Confidentiality and Privacy, Leaking WoT System User Data.

7.3 Secure Practices for designing WoT Interfaces §

Use Appropriate Access Control Schemes

Use Security Metadata options to configure appropriate authentication and authorization methods for WoT Interfaces exposed by a WoT Thing. Minimize the amount of WoT Interfaces without any access control defined (including emitting public events). Consider different levels of access for different users.

This recommendation helps prevent the following threats: WoT Interface Threat - Unauthorized WoT Interface Access, WoT Interface Threat - Exposed Thing Compromise, WoT Communication Threat - System User Data Authenticity, WoT Communication Threat - System User Data Confidentiality and Privacy, Leaking WoT System User Data, Disclosing WoT Thing/Device configuration, WoT DoS Threat.

Avoid Heavy Functional Processing without Authentication

When defining WoT Interfaces exposed by a TD, it is important to avoid any heavy functional processing before the successful authentication of a WoT Consumer. Any publicly exposed network interface should avoid heavy processing altogether.

This recommendation helps prevent the following threats: WoT DoS Threat.

Minimize Network Interface Functionality and Complexity

Network interfaces exposed by a TD (WoT Interfaces) should only provide the minimal necessary functionality, which helps to minimize implementation errors, possibilities for exposing potentially sensitive data, DoS attack possibilities etc. Devices should be strongly encapsulated, meaning the network interfaces should not expose implementation details (for example, the use of particular software frameworks).

This recommendation helps prevent the following threats: WoT Interface Threat - Exposed Thing Compromise, WoT Interface Threat - Unauthorized WoT Interface Access, WoT DoS Threat.

Strong Validation and Fuzz Testing on All WoT Interfaces

All WoT Interfaces (and especially public ones) should be well-tested (including fuzz testing) and validated since they are a primary attack surface of a WoT Thing.

This recommendation helps prevent the following threats: WoT Interface Threat - Exposed Thing Compromise, WoT Interface Threat - Unauthorized WoT Interface Access, WoT DoS Threat.

7.4 Secure Practices for end-to-end security §

The notion 'end-to-end' security is often encountered when security is considered. This notion is often misleading: it is a matter of risk assessment to determine the "ends" that are to be protected (identifying the domain-of-protection) and then select one or more security technologies to fulfill the identified demand (rather than vice versa). In that sense, there is no single, universal notion of "end-to-end" in WoT security. There is also no single security technology that allows to deliver "end-to-end security" for any definition of the ends. The perception of WoT security is to support users of WoT security in implementing security architectures according their perception of the ends (which can vary across users, deployments etc.) For instance: MACSec provides E2E security between endpoints in one (local) network, IPSec provides E2E security between endpoints in an IP-based internetwork, (D)TLS provides E2E security between client and server processes, CMS as well as XML/JSON/CBOR security can provide E2E security on level of application data

The following text addresses E2E security in the sense of protecting application data exchanged between WoT system components: If end-to-end authenticity is desired, it is possible to sign WoT objects (TDs, System User Data) with either digital signatures (using asymmetric cryptographic primitives) or Message Authentication Codes (MACs, using symmetric cryptographic primitives). Such digital signatures or MACs are created by the producers/issuers of the objects and validated by consumers of the objects (which should reject signed the objects whose signatures/MACs are invalid). For data expressed in JSON, RFC 7515 (IETF JSON Web Signature) provides guidelines for computation and validation of digital signatures or MACs using JSON-based data structures. Similarly, if end-to-end confidentiality is desired, it is possible to encrypt objects using available cryptographic encryption primitives. For the objects expressed in JSON, RFC 7516 (IETF JSON Web Encryption) provides guidelines for encryption and decryption of JSON-based data structures.

This recommendation helps prevent the following threats: WoT Communication Threat - System User Data Authenticity, WoT Communication Threat - System User Data Confidentiality and Privacy, Leaking WoT System User Data, WoT Communication Threat - TD Authenticity, WoT Communication Threat - TD Confidentiality and Privacy.

8. Examples of WoT Security Configurations §

The WoT is intended to be used in variety of use cases and deployment configurations. While the examples in this section do not cover all possible deployment variations, they cover many common cases, shows how security mechanisms can be configured for these cases, and highlights some details relevant to security.

8.1 Basic Interaction between WoT Thing and WoT Consumer §

Figure 3 shows the basic WoT Consumer (which can be a browser or an application on a user's smartphone) used to directly operate a WoT Thing. The WoT Thing exposes a WoT Interface and also directly provides a Thing Description. The Thing Description describes the interactions that can be performed on the WoT Thing.

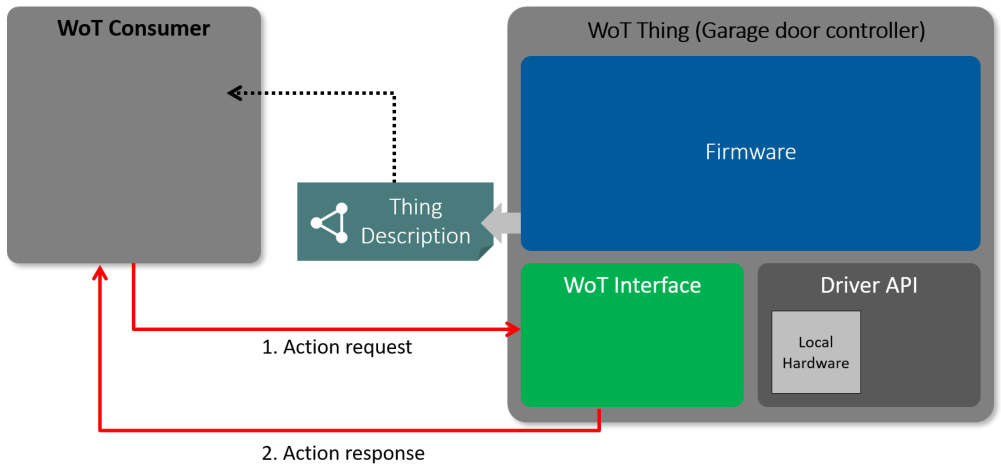


Figure 3 Basic WoT Thing and WoT Consumer

From a security point of view two aspects are important to consider in this scenario. First, the WoT Consumer should be certain that it is talking to the correct WoT Thing and not to some other device exposed on the same network. Users intending to control a specific device, for example opening a garage door, want to make sure they are talking to their own garage door device. In security terms this means that WoT Consumer must have a way to authenticate the device exposing the WoT Thing.

Second, upon receiving requests from a WoT Consumer, the WoT Thing must verify the the WoT Consumer is authorized to perform such requests. For example, a garage door must only process "open" requests from devices associated with authorized users or service providers and not from arbitrary passerby.

These requirements can both be fulfilled using a variety of different security mechanisms. The choice of concrete security mechanisms depends on the deployment scenario, as well as capabilities of the devices. Either or both the WoT Consumer and the WoT Thing might have some resource, network connectivity, or user interface constraints that may limit the choice of security mechanisms.

For example, suppose the WoT Thing is actually an OCF device that has been provided with a Thing Description. This is a reasonable use case since WoT metadata can be used to describe OCF devices, which have RESTful CoAP and/or HTTP network interfaces. Since the WoT only describes devices during operational phase, and does not describe onboarding or provisioning processes, we have to refer to an external standard (such as OCF) that does specify these processes. However, it should be clear that much of what is described in this example is applicable to other standards or even to custom solutions.

The WoT Consumer and the OCF device providing the network interface acting as the WoT Thing can use one of the following methods recommended by the OCF Security Specification [Ocf17] (see Section 10) to mutually authenticate each other given that there is a way to supply credential information required for authentication during provisioning:

- **Symmetric key credentials.** This method assumes that both WoT Consumer and the device exposing WoT Thing have pre-shared symmetric key credentials. Such keys can be established during device provisioning/on-boarding phase. In this case, since this is actually an OCF device, it can use one of the methods recommended in Section 7 [Ocf17] or can use an ad-hoc security protocol, such as EC Diffie-Hellmani, based on other existing pre-shared credentials. See Section 9.2.1 [Ocf17] for more details on such credential types.
- **Raw asymmetric public key credentials.** This method assumes that a WoT Consumer has been provisioned with the public key of the device exposing the WoT Thing (and vice versa) so the devices are able to mutually authenticate each other. Provisioning of public keys can take place during a device provisioning/on-boarding phase. In the case of OCF devices, this can take place using one of the methods recommended in Section 7 [Ocf17]. Section 9.2.3 [Ocf17] also provides more details on this credential type as well as guidance on its secure provisioning.
- **Certificates.** This method assumes that the WoT Consumer has been provisioned with the certificate of the device exposing the WoT Thing (and visa versa) so the devices are able to mutually authenticate each other. The provisioning of the certificates can take place during device

provisioning/on-boarding phase using one of the methods recommended in Section 7 [Ocf17]. Section 9.2.5 [Ocf17] also provides more details on this credential type as well as Section 9.3.

In case it is not possible to pre-provision any of the types of credentials described above during the network setup phase or if the WoT Thing wants to use a more fine-grained access control policy on the WoT Interfaces it is exposing (for example, different controls might require different levels of authorization), the following methods can be used instead:

- **OAuth 2.0-based access tokens.** These tokens follow the JSON Web Token (JWT) format defined in RFC 7519 [JWT15] and should used according to the suggestions by the IETF Authentication and Authorization for Constrained Environments (ACE) specification draft [IETFACE]. These are recommended for devices that are able to use the HTTP protocol and do not have significant resource constraints.
- **Proof-of-possession (PoP) tokens.** These are extensions of the OAuth 2.0-based access tokens that follow the CBOR web token (CWT) format [CBOR17]. The IETF Authentication and Authorization for Constrained Environments (ACE) specification draft [IETFACE] recommends such tokens for resource-constrained devices using the COAP protocol instead of HTTP. For details of using such tokens please see [IETFACE].

Instead of assigning different access tokens to different WoT Interfaces, it is also possible to group related WoT Interfaces under some group or capability name (i.e. "Light controls", "House access" etc.) and assign one token per group to guard any such interfaces. This can provide more balanced access control granularity in many circumstances.

Suitable choices of the above security measures allows mitigation of the WoT Interface Threat - Unauthorized WoT Interface Access threat.

In addition to the above measures, authenticity, confidentiality and replay protection of transferred solution data and of TDs between the WoT Consumer and the WoT Thing is strongly recommended for scenarios where an attacker can observe or/and modify the traffic in the WoT System. While TD transfer might only require authenticity protection, the solution data itself usually requires protection of all its aspects. This helps to mitigate the following threats: WoT Communication Threat - TD Authenticity, WoT Communication Threat - TD Confidentiality and Privacy, WoT Communication Threat - System User Data Authenticity, WoT Communication Threat - System User Data Confidentiality and Privacy.

Authenticity, confidentiality and replay protection can be guaranteed by usage of secure transport protocols to exchange data between the WoT Consumer and the WoT Thing, such as TLS/DTLS. If the underlying protocol does not provide the required security (such as plain CoAP run over a non-(D)TLS protected channel), then authenticity and confidentiality of the transferred data can be implemented separately on the application layer. The recommended method for doing this in resource constrained environments is to use the Object Security of CoAP (OSCoAP) method described in the IETF Object Security of CoAP (OSCoAP) specification draft [OSCOAP17].

The scenario shown in Figure 4 is similar but with the important difference that a Thing description is not stored on the WoT Thing device nor returned by it directly. The TD is instead provided to the WoT Consumer from a Thing Directory residing on some other system, such as a cloud server. This mode may be useful for "retro-fitting" existing devices with Thing Descriptions, for example, or when more secure access to the TD is required than the device itself can support.

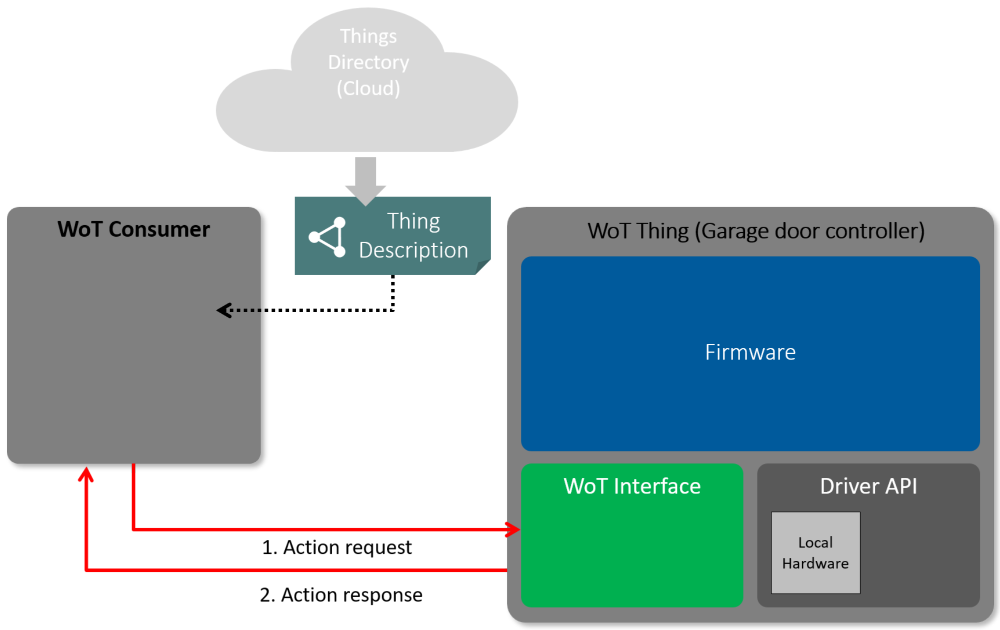


Figure 4 Basic WoT Thing and WoT Consumer with TD provided from a Thing Directory

The primary additional security consideration of this scenario is the need for the secure transfer of the TD between the WoT Consumer and the Thing Directory. The Thing Directory can reside in a Intermediary WoT Servient but could also be supported by a service running on a remote cloud or other remote location. Similarly to the methods described above such transfer should be done using secure transport protocols. In the case that the WoT Consumer is not resource constrained, the usage of TLS/DTLS is the recommended method. In case the remote cloud does not guarantee the secure storage and delivery of Thing Descriptions to the WoT Consumer or in case the remote cloud is not a trusted entity, the authenticity of the Thing Description should be verified using other methods on the application layer, such as wrapping the Thing Description into a protected CBOR Object Encryption and Signing (COSE) object [COSE17]. This can be done by the provider of the Thing Description and verified by the WoT Consumer after the download from the remote cloud.

8.2 Interaction between WoT Thing and WoT Consumer via an Intermediary WoT Servient §

A Figure 5, as in the Figure 3 configuration, also allows a WoT Consumer to connect to and operate or monitor a WoT Thing. However in contrast to the basic (direct connection) case, the interaction between the WoT Consumer and the WoT Thing is now mediated by an Intermediary WoT Servient. The Intermediary WoT Servient exposes a WoT interface and provides a Thing Description that describes interactions that that apply to the Thing. In general, the TD provided by the Intermediary WoT Servient is structurally identical to the one that would have been provided by the Thing directly, except for any necessary modifications to URLs, protocols used, and (perhaps) a modified security configuration.

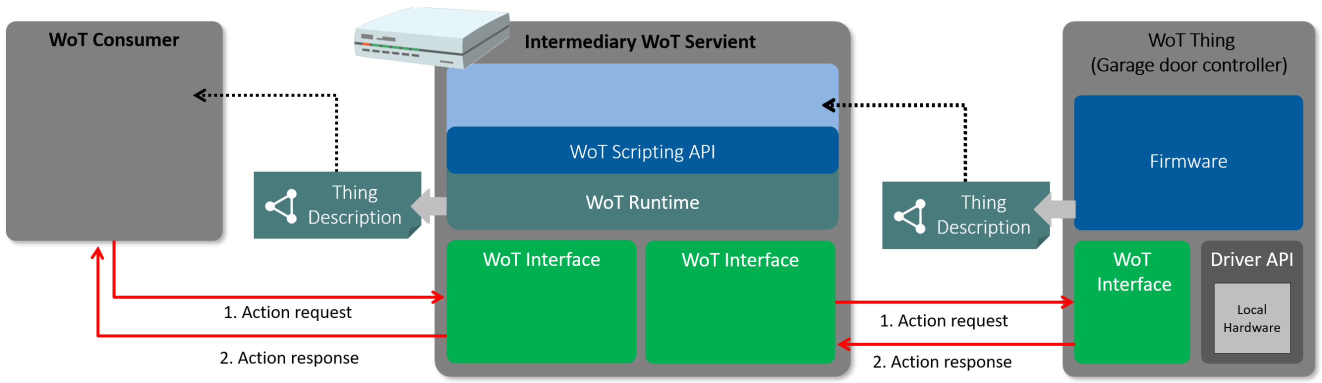


Figure 5 WoT Thing and WoT Consumer via Intermediary WoT Servient

From a security standpoint this case is quite similar to the one described in Section § 8.1 Basic Interaction between WoT Thing and WoT Consumer. The main difference is that mutual authentication should be established between all directly communicating parties, i.e. between the WoT Consumer and the Intermediary WoT Servient, as well as between the Intermediary WoT Servient and the device exposing the WoT Thing network interface. If there is no direct communication between the Intermediary WoT Servient and the device providing the WoT Thing however the mutual authentication between them is redundant. The authentication mechanisms described in Section § 8.1 Basic Interaction between WoT Thing and WoT Consumer are also applicable for this case.

The Thing Description that the Intermediary WoT Servient device provides to the WoT Consumer can be the same as it gets from the device exposing the WoT Thing or it can be modified to better suit the deployment use case or expose additional functionality via the Intermediary WoT Servient. It is also possible that the end device behind the Intermediary WoT Servient is a non-WoT Device and does not provide any Thing Description on its own, and may not support any interaction via a WoT-compatible network Interface. In this case the Intermediary WoT Servient has to build the Thing Description by itself i and expose it to the WoT Consumer. For all typical use cases, the Intermediary WoT Servient should be considered a trusted entity and in this case it can freely modify or build Thing Descriptions it exposes as well as set up fine-grained access controls on different exposed WoT Interfaces. The Intermediary WoT Servient can do this using the same methods described in Section § 8.1 Basic Interaction between WoT Thing and WoT Consumer, i and therefore acts fully on behalf of the end device.

Figure 6 is similar to the previous case but with an important difference: a Thing Description is not provided by the the Intermediary WoT Servient, but can be fetched by the WoT Consumer from a Thing Directory residing on a remote system. Similar to the previous case, this Thing Description can either be the original Thing Description supplied by the end device or modified by the Intermediary WoT Servient. Regardless of the actual setup, the transfer of a Thing Description between any two endpoints should be done using underlying secure protocols. Currently the use of TLS or DTLS is recommended. Similarly to Section § 8.1 Basic Interaction between WoT Thing and WoT Consumer, in case the remote system does not guarantee the secure storage and delivery of Thing Descriptions to the WoT Consumer or in case the remote system is not a trusted entity, the authenticity of the Thing Description should be verified using other methods in the application layer.

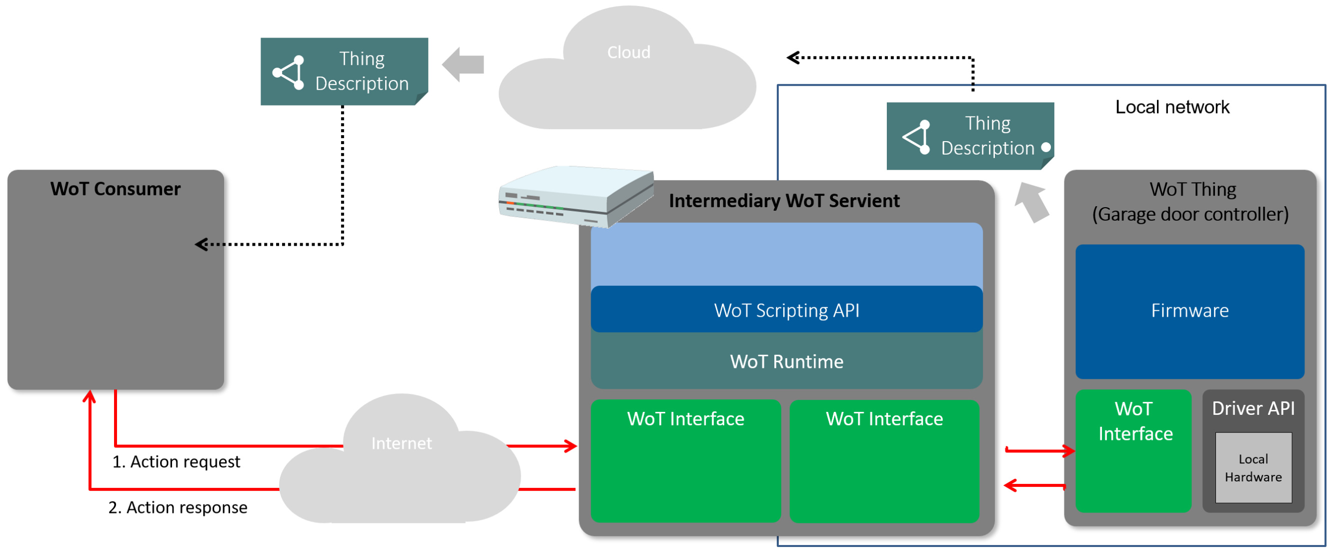


Figure 6 WoT Thing and WoT Consumer via Intermediary WoT Servient with Remote Cloud

8.3 Basic Interaction between WoT Thing and WoT Consumer via a Split Proxy §

Figure 7 shows another common situation where the WoT Thing resides in a local protected network, e.g. behind a NAT/Firewall. In this case the WoT Consumer cannot contact the WoT Thing directly. In the configuration shown here the communication between the WoT Consumer and the WoT Thing is handled via a pair of Intermediary WoT Servients: one residing in the local protected network, known as the Local Intermediary WoT Servient, and another running in the cloud, known as the Remote

Intermediary WoT Servient. We refer to this configuration as a "Split Proxy" because the combination of the Local and Remote Proxy together act like a single proxy service. The Local and Remote Intermediary WoT Servients are connected with a secure channel (which may be non-WoT, eg it does not have to use a protocol known to WoT) as the Local Intermediary WoT Servient only communicates over this channel with the Remote Intermediary WoT Servient. The WoT Remote Intermediary WoT Servient should use the mechanisms described in Section § 8.1 Basic Interaction between WoT Thing and WoT Consumer in order to authenticate and authorize the WoT Consumer before processing any of its requests. Similar to the configurations described in Section § 8.1 Basic Interaction between WoT Thing and WoT Consumer, secure protocols should be used to protect authenticity and confidentiality and provide replay protection when data is exchanged between the WoT Consumer and the Remote Intermediary WoT Servient. If the WoT Consumer is authorized, then the Remote Intermediary WoT Servient can package the WoT request and transfer it over the secure channel to the Local Intermediary WoT Servient, which in turn can issue requests to the WoT Thing on the local network. The local network channel between the Local Intermediary WoT Servient and the WoT Thing can be left unprotected (relying purely on the closed nature of the local network), but it is strongly recommended to employ suitable additional security mechanisms as described in Section § 8.1 Basic Interaction between WoT Thing and WoT Consumer to perform authentication and authorization of involved parties (WoT Thing and Local Intermediary WoT Servient), as well as to provide confidentiality and integrity of transferred solution data and the TD.

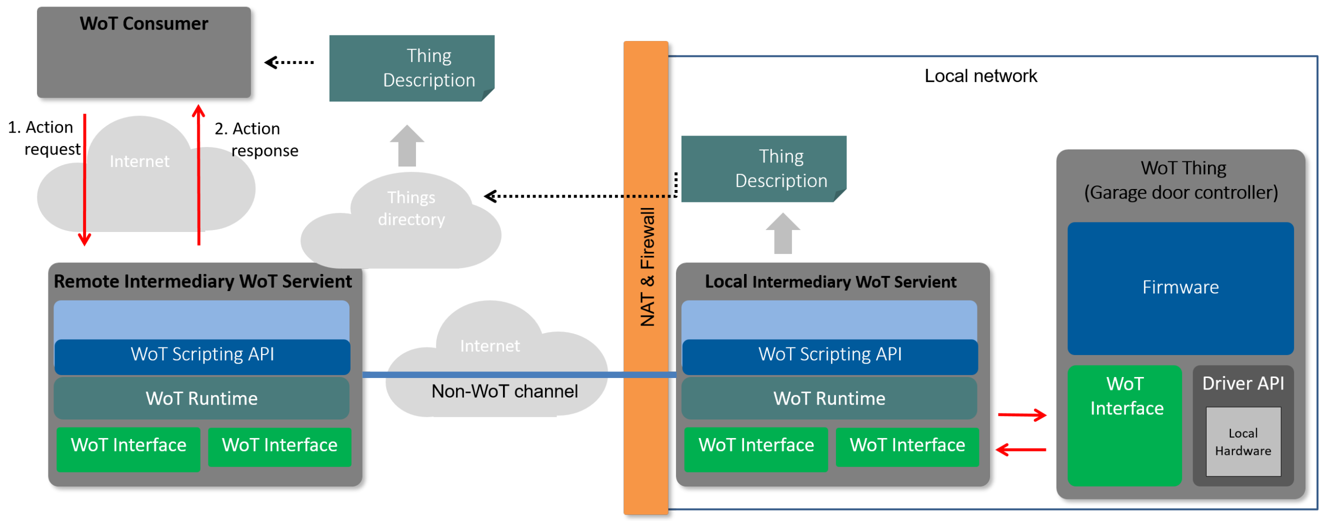


Figure 7 WoT Thing and WoT Consumer via Split Proxy

An alternative security setup for this scenario can use end-to-end security authentication and authorization. In this case the WoT Consumer would not only need to authenticate and authorize itself to the Remote Intermediary WoT Servient, but also to the WoT Thing. In this case it would need a separate set of security credentials, required to successfully authenticate with each party. The exposed TD should indicate the mechanisms to obtain these credentials, e.g. via OAuth2.

8.4 Basic Interaction between WoT Thing and WoT Consumer via a Tunnel §

As an alternative to the use of a split proxy, which requires a trusted remote proxy to accept interaction requests on behalf of the WoT Thing, a secure tunnel (such as an SSH tunnel) can be set up between the WoT Thing and an endpoint in the cloud. In order to traverse the NAT the WoT Thing would have to initiate the connection, so in practice this would be a reverse tunnel set up by the "server". However, then a port made available by the Thing would be made available directly on the cloud server supporting the tunnel endpoint, possibly mapped to a different port. The use of a tunnel for NAT traversal is shown in Figure 8.

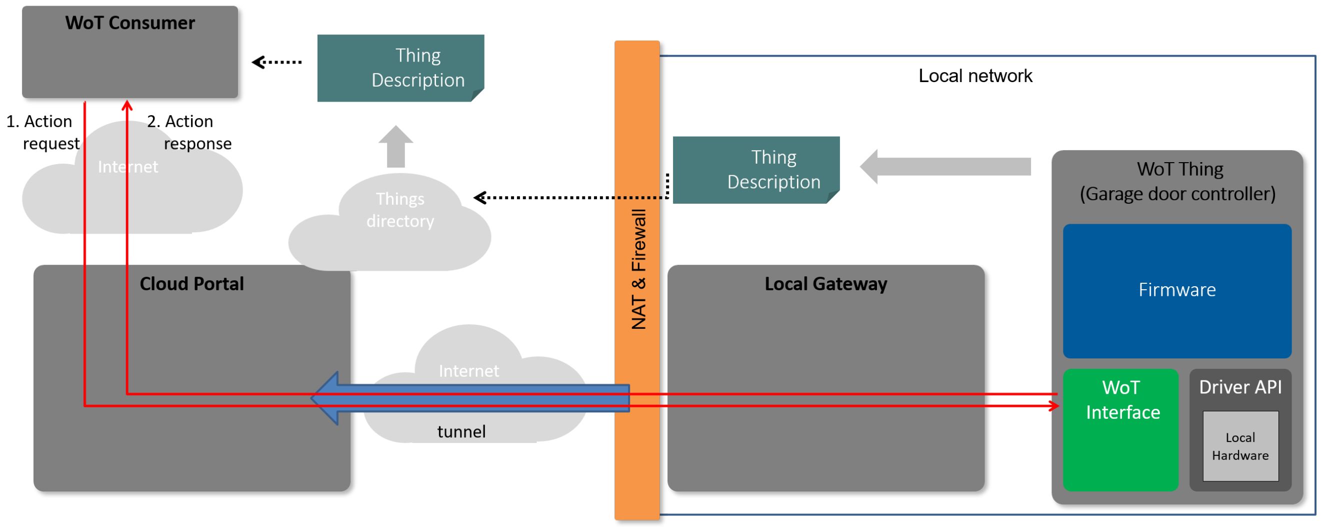


Figure 8 WoT Thing and WoT Consumer via Tunnel

In this configuration the WoT Thing must be responsible for all security, including (if necessary) certificates appropriate for the tunnel endpoint in the cloud. i This implies that the WoT Thing needs to be relatively powerful, as constrained devices may not be able to support security mechanisms suitable for direct exposure on the Internet. The WoT Thing would also have to be actively maintained and patched. If the WoT Thing provides a TD, it should use URLs that refer to the tunnel endpoint. This generally means that the port used for the local side of the tunnel should not be used to provide local connections as the certificates will be incorrect for local connections. Instead a separate port (and possibly a separate TD with local URLs) should be provided.

One advantage of this arrangement is that communication is never available in plaintext on either the gateway or the client. Specifically, the private keys for end-to-end communication between the WoT

Thing and the WoT Consumer, assuming a secure protocol stack such as HTTP-over-TLS is used, never need to be made available to either the local gateway or to the cloud portal. In other words, this configuration supports end-to-end encrypted communication, and does not rely on trusting either the gateway or the cloud portal. Because of this, it is not technically necessary for the tunnel to be over SSH; a regular IP tunnel would suffice.

A variation of this is shown in Figure 9. A reverse tunnel is still used here, but terminates in a local port "inside" the cloud portal, that is, the local port is not made visible outside the cloud portal's firewall. A local proxy service then runs inside the cloud portal and provides a bridge to a secure external network interface. For example, the protocol from the WoT Thing might be HTTP, which the proxy service could bridge to an external HTTP-over-TLS (HTTPS) network interface. In this case, the tunnel should be encrypted (for example, using SSH) to prevent eavesdropping; a regular IP tunnel is not sufficient. Depending on the configuration, the unencrypted traffic (eg HTTP) may also be visible on the local network. In this case, the only protection would be from that provided by the local network, eg WPA. If such a relatively unprotected local interface is *not* desirable, another SSH tunnel can be used between the WoT Thing and the Intermediary WoT Servient, or the WoT Thing can act as its own gateway (with the HTTP traffic protected behind a firewall).

To support local SSH tunnels, of course credentials need to be provisioned between the gateway and the cloud portal. However, this only needs to be done once, when the gateway is installed.

It is worth emphasizing that the security in this configuration is weaker than in the end-to-end tunnel configuration. In particular, unencrypted traffic is available both locally (on the local network and/or inside the gateway) and in the cloud portal. Therefore both the gateway and the cloud portal need to be trusted. This can be mitigated somewhat by using application-level end-to-end object security [COSE17].

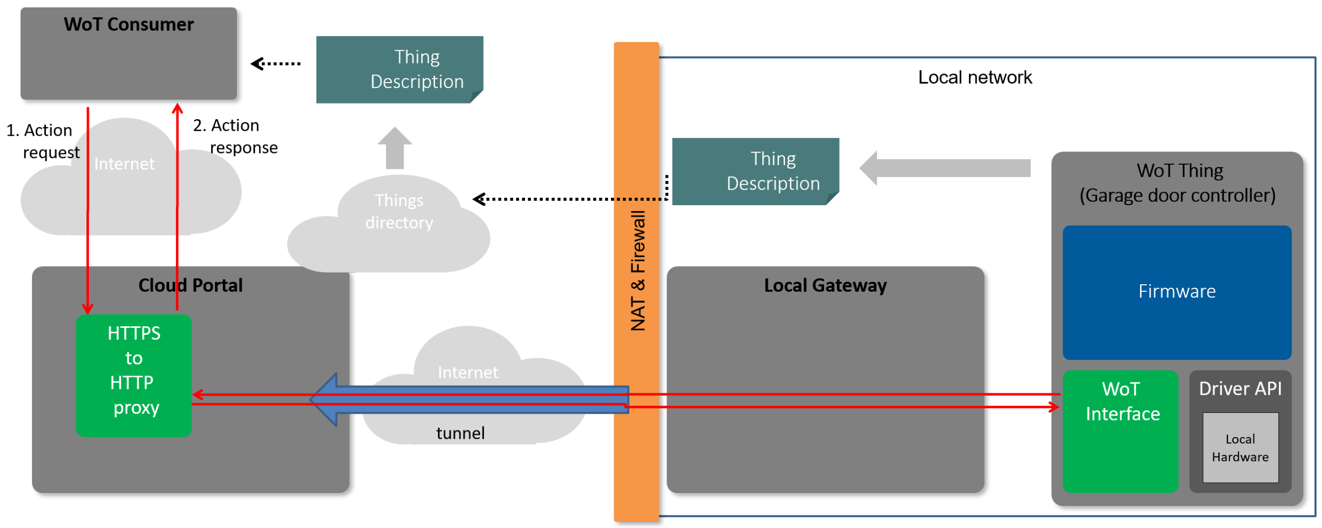


Figure 9 WoT Thing and WoT Consumer via HTTP-over-TLS (HTTPS)/HTTP Proxy

Yet another variant of this approach is shown in Figure 9. In this variant, the proxy is extended to support caching. This means that the proxy may not always forward requests to the WoT Thing, but may instead respond on its behalf using stored state.

There are two sub-variants of this approach. First, a simple HTTP Proxy can be used that caches responses from the WoT Thing. For example, property reads might return an exact copy of an earlier cached payload if only a small amount of time (for some configurable value of "small") has elapsed since the last request for that property. However, this approach still requires that the WoT Thing be "always on" and able to respond as a server in case cached content is not available or has expired. It also requires some care on the part of the WoT Thing to mark payloads as cacheable or uncacheable and configure appropriate time-to-live values.

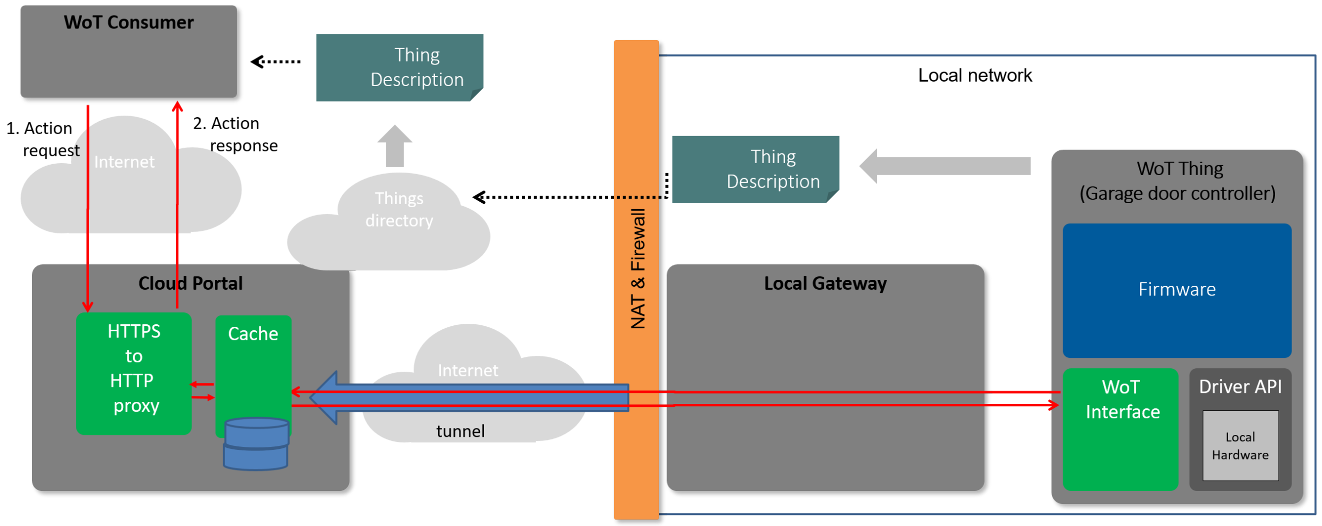


Figure 10 WoT Thing and WoT Consumer via HTTP-over-TLS (HTTPS)/HTTP Proxy

The second sub-variant, shown in Figure 10, takes advantage of the metadata provided by the Thing Description. Specifically, the Thing Description indicates which interactions are "properties" that can be "observed". The proxy service can "observe" all such properties and maintain copies of their state in the cloud server (it is also possible to do this in the gateway, a sub-variant of this pattern). Then when a request comes in to read a property, it can be read from the stored state. However, the "observe" pattern allows the WoT Thing to send updates on its own schedule, leading to greater power efficiency. In addition, the proxy service can see from the Thing Description which network interface methods are "actions" and need to be sent directly to the WoT Thing. This reduces the need for the WoT Thing to mark interactions as being cacheable or not.

In both of these sub-patterns, object security (that is, encryption and authentication of payloads at the application level) can be used to preserve confidentiality and integrity if necessary [COSE17]. A caching proxy can simply store and return copies of the encrypted data representing the state of "private" properties.

A caching proxy, being a variant of the proxy configuration discussed above, has similar security caveats.

8.5 WoT Servient Single-Tenant §

In previous examples we have considered the Wot Servient as a black box exposing a set of WoT Interfaces and Thing Descriptions. Figure 11 shows the WoT Servient (for example consider again a garage door controller device or an Intermediary WoT Servient) in more detail with Scripting API support and a couple of scripts running inside a WoT runtime. For this case we consider all scripts to be trusted and provisioned to the device by a single trusted entity (for example by the manufacturer during factory installation).

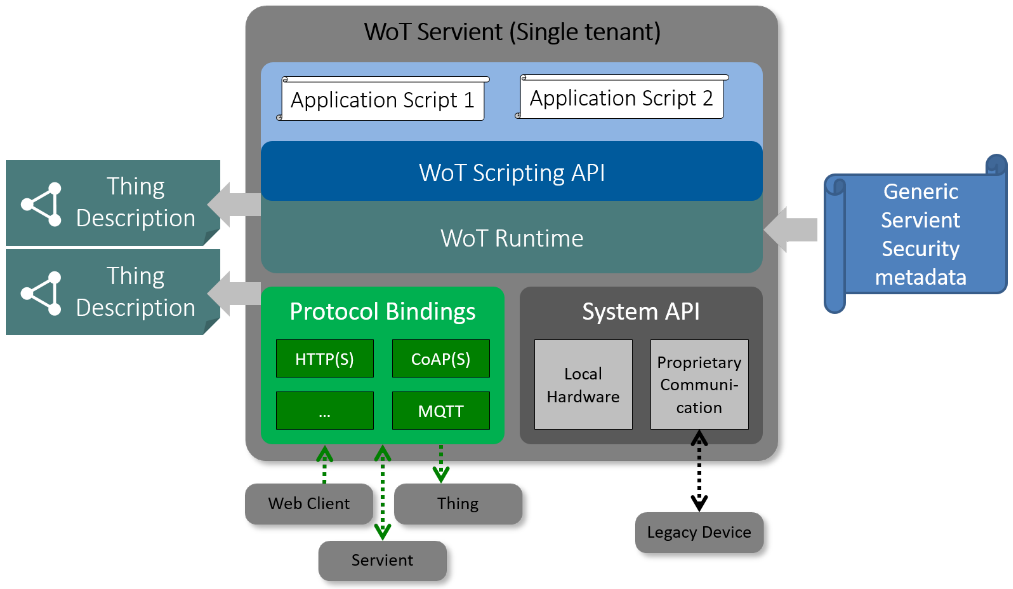


Figure 11 WoT Servient Single-Tenant

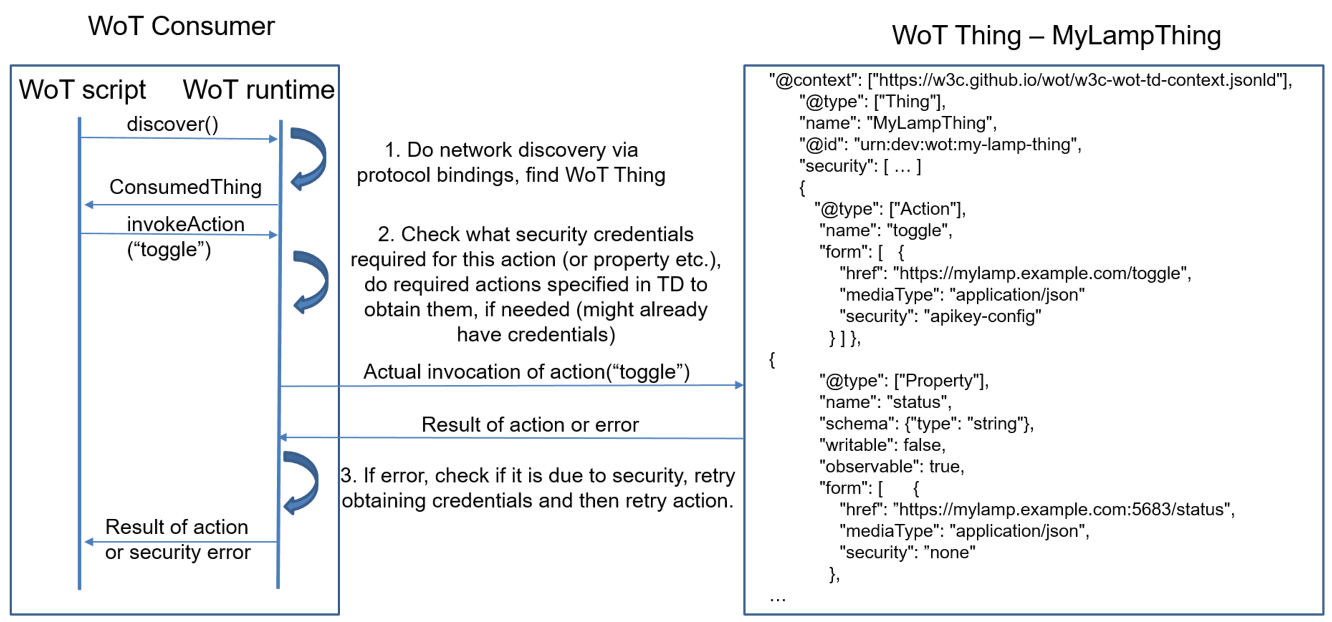
Since all scripts running inside the WoT runtime are considered trusted for this scenario, there is no strong need to perform strict isolation between each running script instance. However, depending on device capabilities and deployment use case scenario risk level it might be desirable to do so. For example, if one script handles sensitive privacy-related data and well-audited, it might be desirable to separate it from the rest of the script instances to minimize the risk of data exposure in case some other script inside WoT gets compromised during the runtime. Such isolation can be performed within the WoT Runtime using platform security mechanisms available on the device.

Depending on the device capabilities, the following isolation mechanisms might be available:

- **WoT Runtime enforced isolation.** This is the basic isolation provided by the WoT Runtime itself and is based on restricting the API exposed to the scripts running within the WoT Runtime. The exact set of guarantees depends on the concrete WoT Runtime implementation and might vary.
- **Native OS process-based isolation.** This method relies on the underlying native OS to provide basic isolation guarantees based on native process isolation. The exact set of guarantees depends on capabilities of native OS, but it usually includes at least memory and execution content isolation. Some other mechanisms like Discretionary Access Controls (DAC) might be also available.
- **Native OS advanced isolation.** An OS might also provide a stronger set of measures that can be deployed for process-based isolation. Examples of such measures are Mandatory access Control (MACs in all major OSes), OS-level virtualization (i.e. namespaces and containers in Linux), Cryptographic methods etc.

The mechanisms are provided in the order from weakest to the strongest isolation method and can be used in combination. In order to choose the appropriate mechanism one should study the capabilities of the underlying device, as well as evaluate the risk level of particular deployment scenario. It is also important to note that all isolation mechanisms usually affect performance and might require non-trivial policy setup and management.

Now if we consider the basic communication scenario between the WoT Consumer and the device exposing the WoT Thing described in Section § 8.1 Basic Interaction between WoT Thing and WoT Consumer from a perspective of script instances running inside WoT Runtime, each such instance should be able to have a way to perform mutual authentication between itself and a remote WoT Consumer or support a token-based authentication method. However, there are no methods in the WoT Scripting API to perform any such tasks. Instead the underlying WoT Runtime and protocol bindings handle the required security functionality transparently for the WoT scripts. Figure 12 shows how it is done, when a WoT Script discovers a new WoT Thing, MyLampThing, using the discover() scripting API method, and invokes an action "toggle".



1. Do network discovery via protocol bindings, find WoT Thing

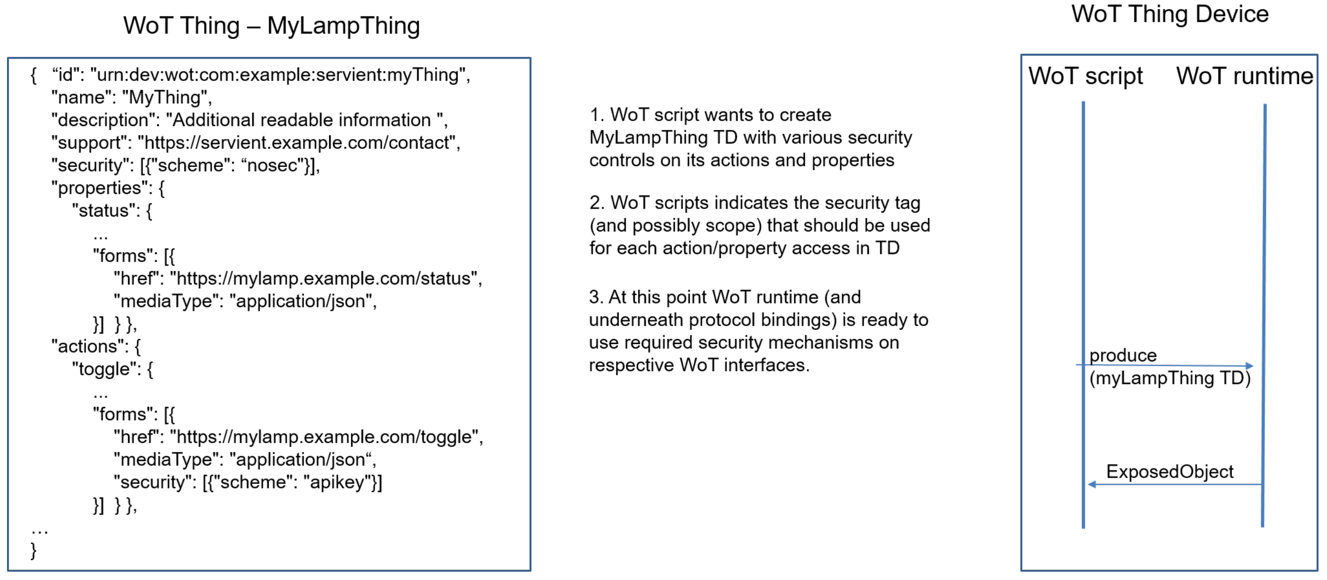
2. Check what security credentials required for this action (or property etc.), do required actions specified in TD to obtain them, if needed (might already have credentials)

3. If error, check if it is due to security, retry obtaining credentials and then retry action.

```
"@context": ["https://w3c.github.io/wot/w3c-wot-td-context.jsonld"],
    "@type": ["Thing"],
    "name": "MyLampThing",
    "@id": "urn:dev:wot:my-lamp-thing",
    "security": [ … ]
    {
      "@type": ["Action"],
      "name": "toggle",
      "form": [  {
         "href": "https://mylamp.example.com/toggle",
         "mediaType": "application/json"
         "security": "apikey-config"
       } ] },
{
      "@type": ["Property"],
      "name": "status",
      "schema": {"type": "string"},
      "writable": false,
      "observable": true,
      "form": [     {
         "href": "https://mylamp.example.com:5683/status",
         "mediaType": "application/json",
         "security": "none"
       },
…
```

Figure 12 Transparent handling of security during WoT Thing discovery and use

Figure 13 shows handling of security configuration during the exposure of a new WoT Thing, MyLampThing. Since the security configurations (types of authentication and authorization methods and corresponding credentials) are not exposed to the WoT Scripts, but handled by the WoT Runtime and underlying protocol bindings, WoT Scripts must either rely on the WoT Runtime to use the default credentials provisioned to it, or indicate the choice of security methods and credentials using a name/id tag known to the WoT Runtime.



```
{  "id": "urn:dev:wot:com:example:servient:myThing",
   "name": "MyThing",
   "description": "Additional readable information ",
   "support": "https://servient.example.com/contact",
   "security": [{"scheme": "nosec"}],
   "properties": {
      "status": {
         …
         "forms": [{
            "href": "https://mylamp.example.com/status",
            "mediaType": "application/json",
         }]  } },
   "actions": {
      "toggle": {
         …
         "forms": [{
            "href": "https://mylamp.example.com/toggle",
            "mediaType": "application/json",
            "security": [{"scheme": "apikey"}]
         }]  } },
 …
}
```

1. WoT script wants to create MyLampThing TD with various security controls on its actions and properties

2. WoT scripts indicates the security tag (and possibly scope) that should be used for each action/property access in TD

3. At this point WoT runtime (and underneath protocol bindings) is ready to use required security mechanisms on respective WoT interfaces.

Figure 13 Transparent handling of security during WoT Thing exposure

8.6 WoT Servient Multi-Tenant §

Figure 14 shows the WoT Servient with two different tenants running in two different instances of WoT Runtime. There is a no trust relation between different tenants and they can be untrusted. Each tenant can have multiple independent WoT scripts running inside their WoT Runtime. WoT Runtime core also has a Thing Manager entity that allows installation of scripts into the corresponding WoT Runtimes.

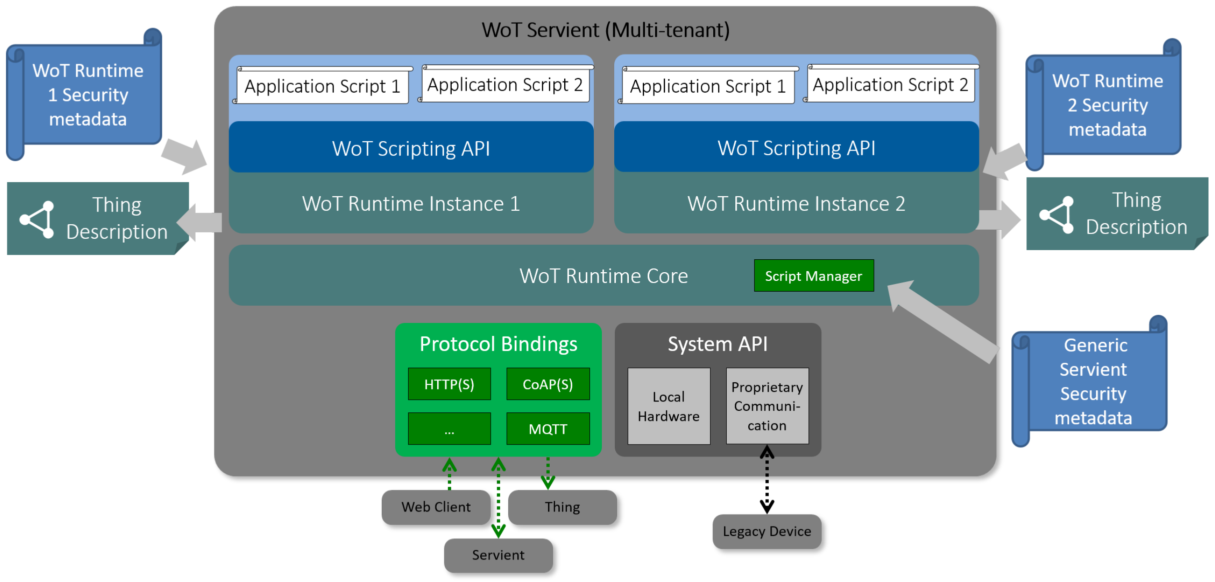


Figure 14 WoT Servient Multi-Tenant

The case of the multi-tenant servient with a possibility to install WoT Scripts remotely into WoT Runtimes brings an additional set of threats that must be taken into account: WoT Script Management Interface Threat, WoT Untrusted Script Threat - Script Compromise, WoT Untrusted Script Threat - Runtime Compromise, WoT Untrusted Script Threat - Authenticity, WoT Untrusted Script Threat - Confidentiality, WoT Untrusted Script Threat - TD Authenticity, WoT Untrusted Script Threat - TD Confidentiality and Privacy, WoT Untrusted Script Threat - System User Data Authenticity, WoT Untrusted Script Threat - System User Data Confidentiality and Privacy, and WoT Untrusted Script Threat - DoS.

In order to address these threats one should utilize the strongest isolation method available on the device to separate different WoT Runtimes. The isolation should cover execution context, runtime memory, local storage and resource allocation between different WoT Runtimes.

In addition the Thing Manager should have a reliable way to distinguish between different tenants in order to securely manage provisioning and updates of scripts into the corresponding WoT Runtime.

This can be achieved by authenticating the tenant using one of the authentication methods described in Section § 8.1 Basic Interaction between WoT Thing and WoT Consumer.

Also each tenant inside the WoT runtime requires a different set of security credentials to be provisioned and made accessible for its scripts. These credentials should be also strictly isolated between different tenants.

9. Security Validation §

This section provides guidance on how a WoT System implementation can be tested in order to evaluate its security and privacy. A WoT Security Best Practices document describes best practices for securing WoT implementations. This section focuses on systems that follow those best practice since systems that do not generally will have known vulnerabilities and testing to reveal them would be redundant.

WoT Systems can consist both of purpose-built components and pre-existing components. Purpose-built WoT components should be implemented using WoT security best practices. However, pre-existing systems may also have WoT Thing Descriptions written for them so that other WoT components can interface with them. Such "retrofitted" pre-existing components may also have pre-existing security weaknesses and vulnerabilities. Security testing may reveal these as well, but it should be understood that describing a network interface using a WoT Thing Description does not guarantee that it will be secure: in order to accommodate pre-existing devices, WoT Thing Descriptions need to be flexible enough to describe both secure *and* insecure interfaces. Security mitigations need to consider the entire system and should in particular take into account that not all components of the system may be equally secure.

9.1 Testing Goals §

The goal of WoT security testing is to find security vulnerabilities in the implementation of WoT components, so they can be mitigated. This can be broadened to finding weaknesses (potential vulnerabilities) since mitigating both actual and potential vulnerabilities is sufficient to secure a system, and in general it can be hard to determine if a weakness is exploitable. It is not possible in general to prove the converse, that a WoT component is secure (that is, that it has no vulnerabilities). However, thorough security testing can at least ensure that there are no obvious (easily discoverable) weaknesses or vulnerabilities in an implementation.

Furthermore, the goal of security testing for an implementor is to identify the existence of weaknesses, not to exploit them to break into or manipulate a system. For an implementor, the priority after discovery of a weakness is to implement a mitigation to prevent it from being exploited, not to actually find or create an exploit and use it. Not all weaknesses are vulnerabilities. Fixing actual (exploitable) vulnerabilities should have a higher priority than simple weaknesses, so an implementor may want to also determine which weaknesses are exploitable in order to prioritize mitigations.

In the following we provide some background on testing, including a description of several varieties of security testing and examples of suitable tools for each. While we provide some tool examples in the following, they are just examples; you are free to use any tool that fits the purpose. After introducing the categories of testing and examples of tools for each, we then present a general security testing plan for WoT Systems based on these categories.

9.2 Best Practices §

Before beginning security testing, it is useful to determine if the components of the system being tested have been implemented following good and up-to-date security practices. What we really want to do is identify systems that use known insecure practices, for which vulnerabilities are known. Many times such vulnerabilities will be obvious from the WoT Thing Description, and can be identified with simple tools.

As an example, suppose a WoT Thing combines "basic" authentication with plain (unencrypted) HTTP. In this case the username and password can be easily recovered from the header information of the unencrypted HTTP request by an eavesdropper with access to the network traffic, for example by code running in a compromised router or by someone who has broken WiFi encryption.

As another example, an older device may use an obsolete form of encryption for which an attack is known.

Hiding the Thing Descriptions of such systems is not really a solution: this is equivalent to security by obscurity which is widely considered to be ineffective. Generally, the "secret information" allowing access to a system should be in the form of easily modifiable keys, not engineering details which, once revealed, cannot be easily changed. However, certainly mitigating obvious vulnerabilities easily discoverable in the Thing Description should be a high priority. Ideally mitigation is accomplished by modifying the implementation but this is not possible in all cases. If the implementation cannot be modified it may be possible to achieve partial mitigation by securing the network environment, e.g. by using a VPN or encrypted tunnels to encapsulate the vulnerable network traffic.

Another point to note is that a Thing Description may not describe all the network interfaces of a Thing. A Thing may have a Thing Description that describes a network interface that follows best practices and may pass all the additional security tests described below on that interface, but may still have an insecure backdoor (additional network interface) not described in the Thing Description. It is the implementor's responsibility to secure *all* network interfaces, even if the Thing Description is subsetted. Note that there are valid reasons to subset a Thing Description, for example to provide different interfaces to users with different roles and access rights.

A WoT System may also have infrastructure components that do not have Thing Descriptions. There may, for example, be services such as proxies or directories. These should also be tested. If they are web services or otherwise have documented network interfaces, then much of the testing procedure described below should also apply to them.

9.3 Functional Security Testing §

Functional testing checks that a WoT Thing implements the interactions described in its Thing Description. This includes whether or not it responds to all URLs given and accepts and returns data as specified in the data schemas given in the Thing Description for each interaction.

Functional testing should also check that all security mechanisms described in the Thing Description are implemented properly. Does the Thing do what the Thing Description says it does in terms of security, and only that? Does the Thing Description accurately describe the security mechanisms used by the Thing? Do all interactions accept authorized accesses, and reject accesses that do not provide correct authentication and authorization information?

Again, there can be both purpose-built and pre-existing devices described by a Thing Description. A Thing Description needs to accurately describe the security behavior and requirements in both cases.

9.4 Adversarial Security Testing §

Adversarial security testing (also known as penetration testing or pentesting) includes various checks that can be done in order to find weaknesses in the target device or service and determine their severity and exploitability (that is, whether they are actual vulnerabilities). On a high level adversarial testing consists of three stages, which are often applied iteratively:

1. **Information gathering:** Required information is collected about the attack target.
2. **Weakness discovery:** An attack target is analyzed and weaknesses are found.
3. **Exploitation generation:** An attempt is made to convert weaknesses into vulnerabilities by discovering or designing exploits.

Exploits are designed to achieve a desired outcome, such as privilege escalation, authorization bypass, denial of service, and other goals. A successful exploit of one vulnerability may be used to expose other weaknesses and allow them in turn to be exploitable. Conversely, a weakness may be prevented from exploitation by a mitigation that prevents the class of exploit to which it is vulnerable.

Many resources and tools exist to help penetration and security testers to perform the above activities. Many documents and tutorials have also been written to help guide this activity. One example of such guide is the "Web Service Security Testing Cheat Sheet" [Owa18] written by the Open Web Application Security Project (OWASP). Generally speaking, WoT components can be treated as web services for the purposes of security testing, especially if they use HTTP. However, WoT components also have some additional considerations, for example local access while using HTTP-over-TLS (HTTPS), or use of non-HTTP protocols such as CoAP or MQTT.

For the purposes of WoT security testing we will focus on Stage 2, *Weakness Discovery*. Weakness discovery can be broken down into a combination of static weakness discovery and runtime weakness discovery.

Stage 1 is not really necessary in the case of white-box testing, which we will assume (given that, for example, a Thing Description already documents much of the information needed). Generally speaking, trying to achieve security through obscurity is not feasible or desirable, so for testing we should assume the attacker knows as much about the system as the implementor. Also, since our goal is to test the security of the components of a WoT System, not actually break into it, we also will not focus on Stage 3. However, an exploitable vulnerability has a higher priority for correction than a simple weakness for which no exploit is (yet) known, so this information is still useful to prioritize mitigation.

9.4.1 Static Weakness Discovery §

Static weakness discovery typically requires an access to the target's source code, and therefore is not always possible to perform for a black-box penetration tester. However, since a WoT Thing developer or WoT Service provider have the access, they are strongly encouraged to use the below methods to check for weaknesses (potential vulnerabilities) in their components as part of white-box testing. Static weakness discovery is very well supported and automated. Many tools are available, so after an initial setup phase, it should add little overhead to the WoT component development and release process.

9.4.1.1 Static Code Analysis §

The most common example of static weakness discovery is usage of a static code analyzer tool that is able to parse a program's code and highlight potential development mistakes and insecure practices, such usage of insecure functions or libraries, forgotten boundary checks when operating on arrays, and many other sources of security weakness. Usually the output of such a tool is in the form of a report that needs to be manually triaged to determine if each reported issue is a false positive or a real mistake. While static development tools are constantly improving and over time are able to offer better and better coverage, lower false positive rates, and discover more weaknesses, it is important to remember that these tools (as any others) do not guarantee finding all weaknesses in the scanned component. Other methods should be used to complement static analysis.

Examples of tools: Many commercial and open source tools exist that primarily differ on the source code languages that they can process as input. Our major suggestion would be to choose a well-established, mature tool that is actively supported and developed. Some examples include Klocwork [Klocwork], Coverity [Coverity], and Checkmarx [Checkmarx]. Also, while some of these tools are commercial and require a license to use, they often do provide a free scanning service for open source projects. One example of such service is the online Coverity service [CoverityOnline].

9.4.1.2 Known vulnerability checking §

In addition to checking for development mistakes in new code, it is also important to check that programs do not include vulnerable third-party libraries or utilities with known weaknesses or vulnerabilities. There are many tools developed for this purpose and on a high level they work by scanning executable binaries (or source code for non-compiled languages) or application packages in a search for vulnerable components, such as old versions of libraries that are vulnerable to known attacks. The information about weaknesses and vulnerabilities usually comes from the open NIST database which is updated regularly since new weaknesses and vulnerabilities are reported all the time. This also implies that the analysis should be repeated regularly, ideally daily and as part of an automated build process, using the latest version of the weakness and vulnerability database. As new weaknesses, vulnerabilities, and exploits are found, components may then have to be updated to mitigate any new weaknesses and vulnerabilities.

Examples of tools: Just as with static code analyzer software, our major suggestion would be to choose a well-established, mature tool that is actively supported and developed. Some examples include Protecode [Protecode], Dependency-check [OWASP-Dependency-Check], and Snyk [Snyk].

9.4.2 Runtime (Dynamic) Weakness Discovery §

In contrast to static weakness discovery, runtime weakness discovery does not require access to the source code or compiled binaries of a component, but merely an ability to access a component's exposed interfaces (especially network-facing interfaces) and/or an ability to observe and modify the protocol and the component's communication with other components.

The goal of runtime weakness discovery is to find an interface input or protocol modification that leads to unintended behavior, such as component deadlock or crash. Even if a crash is not the desired outcome for an attacker, a crash is also an indication of a weakness, such as a lack of input validation, that can be used in more sophisticated exploits. Of course finding inputs that cause such problems and mitigating them also increases the overall robustness and quality of the code.

Just as with static weakness discovery, many tools exist to help with this task. For web services, it is generally a bit harder to reach full automation (with the exception of fuzz testing) here, and the best results can usually only be achieved by combining both manual and automated tasks. However, the advent of API metadata such as OpenAPI and the WoT Thing Description may in the future lead to more sophisticated and automated tooling in this area.

9.4.2.1 Fuzz Testing §

Fuzz testing is one of the most well known and used runtime weakness discovery methods. It is well-automated, many tools exist for common protocols and payloads. It is a very powerful method for weakness discovery. Fuzz testing works by generating (randomly or pattern-based) highly varied input for a specific network exposed interface or protocol payload, sending this input to the tested component, and observing the result. If a "desired" outcome happens (such as tested component crashes), the corresponding input and details of the crash are recorded for the analyst to manually process.

The biggest challenge with fuzz testing is usually to be able to generate the randomized input in a form that is still "correct enough" that it does not get discarded by the lower layers of software or network stack. For example, if the fuzz testing goal is to test how a network-enabled thermostat processes numeric values on its temperature setting HTTP-exposed interface, it is important that fuzzed requests have valid HTTP headers, body structure, etc. This way the requests will get delivered all the way to the high-level logic inside the device runtime and will not discarded by lower-level HTTP message validation and processing components.

Examples of tools: There are numerous tools exists for fuzzing HTTP(S)-exposed interfaces. Examples include Burp Suite [Burp], Wfuzz [Wfuzz], and Wapiti [wapiti]. Newer protocols, like COAP(S) and MQTT(S), have considerably fewer tools available. However some do exist. Examples for CoAP(S) include fuzzcoap [FuzzCoAP] and CoAP Peach Pit (from Peach Fuzzer) [PeachPit-COAP]. Some of these may still in the experimental phase.

9.4.2.2 Protocol Analysis §

In addition to automated Fuzz testing, one might be able to discover weaknesses by manually analyzing protocol elements such as headers, payload, and attributes. For example, an analyst might look for the use of older encryption or authentication algorithms, or the use of insecure combinations of protocol options. While the actual protocol analysis is a manual process, many tools exist to capture the protocol traffic, parse it according to the protocol, and present it to the analyst in an organized fashion.

Examples of tools: Examples of network protocol analyzers include Wireshark [Wireshark] and tcpdump [tcpdump].

9.4.2.3 Vulnerability Scanners §

There are many tools available that attempt to perform runtime testing for known vulnerabilities. They usually operate by attempting to run a known set of exploits or weakness trigger inputs against the target and report the outcomes. Such tools should be used in combination with other testing tools in order to obtain a more complete runtime weakness and vulnerability discovery result.

Examples of tools: While many different vulnerability scanners exist, some specifically target web applications and web server service vulnerabilities and are therefore more focused. Examples include w3af [w3af], the Burp vulnerability scanner [Burp], Nikto [Nikto], and WATOBO [WATOBO].

9.4.3 Exploitation §

When a weakness or a set of weaknesses is found for a component, it is important to estimate the exploitation risk level for them. Exploitation risk level aims to determine if a particular weakness can be converted into a vulnerability and used by an attacker to achieve particular attack end goals, such as privilege escalation, authorization bypass, denial of service, and so on. This step can help to prioritize the order in which weaknesses must be mitigated. Developers should fix the most easily exploitable vulnerabilities first and then continuing with others in decreasing level of severity or probability of exploitation. Some tools mentioned in section § 9.4.2.2 Protocol Analysis can be used to test the exploitabilty of a particular weakness. Additionally, for known vulnerabilities, the Exploit-DB [exploit-db] database hosts many public exploits and allows to search by the CVE number of the associated (known) vulnerability. However, it is important to remember that it is generally very hard to reliably determine the exploitability of a given weakness: if an exploit cannot be found, it does not guarantee that the weakness is not exploitable. Moreover, for some weaknesses, they might not be exploitable on their own, or may only be weakly exploitable, but may still play an important role in the attacker exploitation chain as a stepping stone towards the end exploitation goal. Therefore, it is strongly encouraged to fix or mitigate all weaknesses discovered in analyzed components.

9.5 Security Testing Plan Framework §

Given the above background, we can now outline a testing plan framework for WoT Systems. This is just an outline and would have to be expanded and adapted for the specific circumstances and application area of the WoT System under test. In particular, some tool choices may depend on the programming language, build environment, network environment, and budget (for non-free tools). Also the testing procedure will be different for purpose-built devices and services vs. pre-existing devices and services.

1. **Static Weakness Discovery**
   1. **Static Code Analysis:** If source code is available, a static code checker should be used to check for weaknesses in new purpose-written code.
   2. **Known Vulnerability Checking:** If binaries and/or a list of package dependencies are available, a vulnerability checker should be used to check for known weaknesses and vulnerabilities in any libraries used.
2. **Runtime Weakness Discovery** appropriate tools such as Fuzz Testing, Protocol Analysis, and Vulnerability Scanners can be discover problems even when only the network interface is available, and are complementary to other kinds of checks.
3. **Exploitation Analysis** This can be used to prioritize which weaknesses and vulnerabilities should be addressed first or whether a given vulnerable system can even be used in a given environment.

Ideally, security testing should be integrated into the development process and all discovered weaknesses addressed as soon as possible. If this is not possible, weaknesses should be prioritized based on the threat model and potential for exploitation and addressed in priority order.

9.6 Suggested Testing Frequency §

The actual detailed test procedure depends on the type of activity, the WoT device setup and the actual test tool used. However, below are our general recommendations for the frequency of different testing activities:

1. **Static Weakness Discovery** should be done regularly, ideally integrated into the development work flow for a WoT component. This is important, since any even small code changes can introduce development mistakes, or alternatively new weaknesses can be discovered in the libraries that a program depends on. New weaknesses and vulnerabilities are discovered in existing libraries almost daily.
2. **Runtime Weakness Discovery** should be also done on a regular basis, for example for each major release or development milestone.

3. **Exploitation Analysis** should be done occasionally, whenever possible. It is the most time and resource consuming activity, often requires external resources with specialized knowledge, and cannot in general be automated.

9.7 Summary §

A security testing plan focused on weakness and vulnerability discovery has been presented that leverages existing tools and approaches for web services and emerging tools for new IoT-oriented network interfaces. A testing plan should be developed suitable for each system developed using the WoT Architecture but will generally consist of three stages: static weakness discovery, runtime weakness discovery, and Exploitation Analysis. A vulnerability is a weakness with a known exploit and should have priority for mitigation, but all weaknesses may potentially lead to vulnerabilities and ideally all weaknesses would be mitigated or corrected. In general, a WoT Thing Description can be used to describe both secure and insecure systems, and in fact this is necessary to support existing devices, which may or may not themselves be intrinsically secure. However, WoT Thing Descriptions provide new opportunities for understanding and planning security testing and mitigations.

10. Terminology §

Please refer to [WoT-Architecture] for definitions of the following terms used in this document: WoT Consumer, WoT Thing, WoT Servient, WoT Protocol Binding, WoT Binding Templates, WoT Intermediary, WoT Thing Directory, WoT Thing Description, Exposed Thing, WoT Interface, WoT Runtime, and WoT Scripting API.

Additionally we define the following terms:

WoT System:
A set of WoT-enabled devices and the communication links between them.

WoT Device:
A physical device that enables WoT layer on top of its software stack.

Within the Security Validation section we are using the security vocabulary defined by the ITU-T [ITUx1500], [ITUx1520], [ITUx1524] for the following terms:

Vulnerability:
Any weakness in software that could be exploited to violate a system or the information it contains [ITUx1500].

Weakness:

A shortcoming or imperfection in the software code, design, architecture, or deployment that, could, at some point become a vulnerability, or contribute to the introduction of other vulnerabilities [ITUx1500].

In short, a weakness is a *potential* vulnerability. A weakness only becomes an *actual* vulnerability once an exploit is known.

A. Change Log §

A.1 Changes from Second Release §

- Addition of references to published versions of the [WoT-Architecture], [WoT-Thing-Description], [WoT-Binding-Templates], and [WoT-Scripting-API] documents, as well as the following new references: [Burp], [Checkmarx], [Coverity], [CoverityOnline], [exploit-db], [FuzzCoAP], [ITUx1500], [ITUx1520], [ITUx1524], [Klocwork], [Nikto], [Owa18], [OWASP-Dependency-Check], [PeachPit-COAP], [Protecode], [Snyk], [tcpdump], [w3af], [wapiti], [WATOBO], [Wfuzz], and [Wireshark].
- The name was changed from *Web of Things (WoT) Security and Privacy Considerations* to *Web of Things (WoT) Security and Privacy Guidelines*. This change was made to avoid confusion with the "considerations" sections of individual WoT documents.
- Revision and clarification of the attack surfaces of a WoT System described in § 3.2.4 Attack Surfaces.
- Empty section for *Secure Practices for Thing Directory Interaction Protocols* was removed.
- Addition of § 9. Security Validation, which provides security testing guidance for WoT Systems.
- Revision of terminology usage to be consistent with the definitions in [WoT-Architecture].
- Revision of § 10. Terminology to state specific terms used from [WoT-Architecture], add definitions of terms used only in this document, and reference externally defined terms in [ITUx1500].

A.2 Changes from First Release §

- Replace inline description of W3C Patent Policy with link to external document.
- Added § 2. Lifecycle of a WoT Device describing the states and transitions of a device throughout its lifetime.

- Added MQTT as a focus protocol.
- Expanded and clarified role and threat definitions in § 3.2 Threat Model.
- Added WoT Thing Directory to items (currently) out of scope.
- Added Side Channel threat to § 3.3.1 Scenario 1 - Home Environment.
- Added § 3.3.2 Scenario 2 - Business/Corporate Environment describing threats in an office building environment.
- Added § 3.3.3 Scenario 3 - Industrial/Critical Infrastructure Environment describing threats in an industrial environment.
- Added § 4. Privacy Considerations.
- Revised § 7. Best Security Practices and Mitigations for WoT Systems, and added recommended mitigations.
- Revised § 8.1 Basic Interaction between WoT Thing and WoT Consumer to include OCF as an example, and describe use of CoAPS and DTLS in this case.
- Added § 8.3 Basic Interaction between WoT Thing and WoT Consumer via a Split Proxy and § 8.4 Basic Interaction between WoT Thing and WoT Consumer via a Tunnel to describe two possible strategies for traversing NATs.

B. References §

B.1 Informative references §

[Bel13]
Web Security in the Real World. S. Bellovin. Workshop on Improving Trust in the Online Marketplace, NIST. April 2013. URL: https://csrc.nist.gov/csrc/media/events/workshop-on-improving-trust-in-the-online-marketpl/documents/presentations/bellovin_ca-workshop2013.pdf

[Bel89]
Security Problems in the TCP-IP Protocol Suite. S. Bellovin. Computer Communication Review, Vol. 19, No. 2. April 1989. URL: https://cseweb.ucsd.edu/classes/sp99/cse227/ipext.pdf

[Ber14]
Authentication Protocols, Web UX and Web API. V. Bertocci. April 2014. URL: http://www.cloudidentity.com/blog/2014/04/22/authentication-protocols-web-ux-and-web-api/

[Bor14]
Terminology for Constrained-Node Networks. C. Bormann; et al.. IETF RFC 7228. May 2014. URL: https://tools.ietf.org/rfc/rfc7228.txt

[Bru14]
Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations. C. Brubaker; et al.. IEEE Security Privacy. 2014. URL: https://www.cs.utexas.edu/~shmat/shmat_oak14.pdf

[Burp]
Burp Suite Web vulnerability scanner. URL: https://portswigger.net/burp

[CBOR17]
CBOR Web Token (CWT). IETF. June 2017. Internet-Draft. URL: https://tools.ietf.org/pdf/draft-ietf-ace-cbor-web-token-07.pdf

[Checkmarx]
Checkmarx project page. URL: https://www.checkmarx.com/

[Coo13]
Privacy Considerations for Internet Protocols. A. Cooper; et al. IETF RFC 6973 (IAB Guideline). July 2013. URL: https://tools.ietf.org/html/rfc6973

[CoRE-RD]
CoRE Resource Directory. IETF. 03 July 2017. Internet-Draft. URL: https://tools.ietf.org/html/draft-ietf-core-resource-directory-11

[COSE17]
CBOR Object Signing and Encryption (COSE). IETF. May 2017. Internet-Draft. URL: https://tools.ietf.org/pdf/draft-ietf-cose-msg-24.pdf

[Coverity]
Coverity Tutorial: Introduction to Coverity. URL: https://community.synopsys.com/s/article/Coverity-Tutorial-Introduction-to-Coverity

[CoverityOnline]
Coverity online scanning service. URL: https://scan.coverity.com

[Dur13]
Analysis of the HTTPS Certificate Ecosystem. Z. Durumeric; et al.. Proc. of the 2013 conference on Internet measurement conference. October 2013. URL: https://conferences.sigcomm.org/imc/2013/papers/imc257-durumericAemb.pdf

[Ell00]
Ten Risks of PKI: What You're not Being Told about Public Key Infrastructure. C. Ellison; B. Schneier. Computer Security Journal, v 16, n 1,. 2000. URL: https://www.schneier.com/paper-pki.pdf

[exploit-db]
Exploit database project page. URL: https://www.exploit-db.com/

[Fu01]
Dos and Don'ts of Client Authentication on the Web. K. Fu; et al.. Proc. 10th USENIX Security Symposium. August 2001. URL: https://pdos.csail.mit.edu/papers/webauth:sec10.pdf

[FuzzCoAP]

FuzzCoAP - Fuzzing for Robustness and Security Testing of CoAP Servers. URL: https://github.com/bsmelo/fuzzcoap

[Garcia17]

State-of-the-Art and Challenges for the Internet of Things Security. O. Garcia-Morchon; S. Kumar; M. Sethi. URL: https://datatracker.ietf.org/doc/draft-irtf-t2trg-iot-seccons/

[Geo12]

The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software. M. Georgiev; et al.. Proc. of the 2012 ACM conference on Computer and communications security. 2012. URL: https://www.cs.utexas.edu/~shmat/shmat_ccs12.pdf

[Gol03]

Cryptography and Cryptographic Protocols. O. Goldreich. Distributed Computing, vol. 16. 2003. URL: http://www.wisdom.weizmann.ac.il/~oded/foc-sur01.html

[Gre14]

How do you know if an RNG is working?. M. Green. March 2014. URL: https://blog.cryptographyengineering.com/2014/03/19/how-do-you-know-if-rng-is-working/

[Gut02]

PKI: It's Not Dead, Just Resting. P. Gutman. IEEE Computer, vol. 35, no. 8. Aug. 2002. URL: https://www.cs.auckland.ac.nz/~pgut001/pubs/notdead.pdf

[Hea13]

An update on our war against account hijackers. M. Hearn. Feb 2013. URL: https://googleblog.blogspot.de/2013/02/an-update-on-our-war-against-account.html

[IETFACE]

IETF Authentication and Authorization for Constrained Environments (ACE). URL: https://tools.ietf.org/wg/ace/

[Iic15]

Industrial Internet Reference Architecture. Industrial Internet Consortium. June 2015. URL: http://www.iiconsortium.org/IIRA.htm

[IicRA17]

The Industrial Internet of Things Volume G1: Reference Architecture. Industrial Internet Consortium. IIC:PUB:G1:V1.80:20170131. Jan 2017. URL: https://www.iiconsortium.org/IIRA.htm

[IicSF16]

The Industrial Internet of Things Volume G4: Security Framework. Industrial Internet Consortium. IIC:PUB:G4:V1.0:PB:20160926. Sept 2016. URL: https://www.iiconsortium.org/IISF.htm

[ISF17]

IoT Security Foundation Best Practice Guidelines. IoT Security Foundation. May 2017. URL: https://iotsecurityfoundation.org/best-practice-guidelines/

[ITUx1500]
ITU-T Rec. X.1500 Overview of cybersecurity information exchange. ITU-T SG17. International Telecommunication Union. Apr 2011. URL: https://www.itu.int/rec/T-REC-X.1500

[ITUx1520]
ITU-T Rec. X.1520 Common vulnerabilities and exposures. ITU-T SG17. International Telecommunication Union. Jan 2014. URL: https://www.itu.int/rec/T-REC-X.1520

[ITUx1524]
ITU-T Rec. X.1524 Common weakness enumeration. ITU-T SG17. International Telecommunication Union. March 2012. URL: https://www.itu.int/rec/T-REC-X.1524

[Jon14]
A JSON-Based Identity Protocol Suite. M. Jones. Information Standards Quarterly, vol. 26, no. 3. 2014. URL: http://www.niso.org/sites/default/files/stories/2017-08/SP_Jones_JSON_isqv26no3.pdf

[JWT15]
JSON Web Token (JWT). IETF. May 2015. URL: https://tools.ietf.org/html/rfc7519

[Ken03]
Who Goes There? Authentication Through the Lens of Privacy. S. Kent; L. Millet. The National Academies Press, Washington D.C. 2003. URL: https://www.nap.edu/read/10656/chapter/1

[Klocwork]
Klocwork. Faster delivery of secure, reliable, and conformant code. URL: https://www.roguewave.com/products-services/klocwork

[Lam04]
Computer Security in the Real World. B. Lampson. IEEE Computer, vol. 37, no. 6. June 2004. URL: https://pdfs.semanticscholar.org/6fe5/ba7a096e391d985e7818fef9d0f0636210a0.pdf

[Lea05]
A Universally Unique IDentifier (UUID) URN Namespace. P. Leach; et al. IETF RFC 4122. July 2005. URL: https://tools.ietf.org/html/rfc4122

[Loc05]
Demystifying SAML. H. Lockhart. May 2005. URL: http://www.oracle.com/technetwork/testcontent/saml-084342.html

[Mel15]
Securing the Industrial Internet of Things. D. Melzer. June 2015. URL: https://c.ymcdn.com/sites/www.issa.org/resource/resmgr/journalpdfs/feature0615.pdf

[Mic17]

Internet of Things security architecture. Microsoft. STRIDE threat model for IoT. Jan 2017. URL: https://docs.microsoft.com/en-us/azure/iot-suite/iot-security-architecture

[Moo02]
A critical review of End-to-end arguments in system design. T. Moors. Proc. of the IEEE International Conference on Communications. 2002. URL: https://www.csd.uoc.gr/~hy435/material/moors.pdf

[Nikto]
Nikto web server scanner. URL: https://cirt.net/Nikto2

[Nis15]
Guide to Industrial Control Systems (ICS) Security. NIST. NIST Special Publication 800-82.

[Ocf17]
The OCF Security Specification, version 1.0.0. OCF. June 2017. URL: https://openconnectivity.org/specs/OCF_Security_Specification_v1.0.0.pdf

[Oos10]
Provisioning scenarios in identity federations. M. Oosdijk; et al.. Surfnet Research Paper. 2010. URL: https://tnc2011.terena.org/getfile/696

[OSCOAP17]
Object Security of CoAP (OSCOAP). IETF. July 2017. Internet-Draft. URL: https://tools.ietf.org/pdf/draft-ietf-core-object-security-04.pdf

[Owa17]
Threat Risk Modeling. OWASP. OWASP. Jan 2017. URL: https://www.owasp.org/index.php/Threat_Risk_Modeling

[Owa18]
Web Service Security Testing. OWASP. OWASP. Aug 2018. URL: https://www.owasp.org/index.php/Web_Service_Security_Testing_Cheat_Sheet

[OWASP-Dependency-Check]
OWASP Dependency Check. URL: https://www.owasp.org/index.php/OWASP_Dependency_Check

[PeachPit-COAP]
CoAP Peach Pit User Guide. URL: https://www.peach.tech/wp-content/uploads/CoAP_DataSheet.pdf

[Pri19]
Bootstrapping Remote Secure Key Infrastructures (BRSKI). M. Pritikin, M. Richardson, T. Eckert,M. Behringer, K. Watsen. IETF. Aug. 2019. URL: https://tools.ietf.org/html/draft-ietf-anima-bootstrapping-keyinfra-26

[Protecode]

Black Duck Software Composition Analysis. URL: https://www.synopsys.com/software-integrity/security-testing/software-composition-analysis.html

[Res03]
Guidelines for Writing RFC Text on Security Considerations. E. Rescorla; et al.. IETF RFC 3552 (IAB Guideline). 2003. URL: https://tools.ietf.org/html/rfc3552

[Sch14]
The Internet of Things Is Wildly Insecure — And Often Unpatchable. B. Schneier. Wired. Jan. 2014. URL: https://www.wired.com/2014/01/theres-no-good-way-to-patch-the-internet-of-things-and-thats-a-huge-problem/

[Sch99]
Breaking Up Is Hard To Do: Modeling Security Threats for Smart Cards. B. Scheier; A. Shostack. USENIX Workshop on Smart Card Technology, USENIX Press. 1999. URL: https://www.schneier.com/paper-smart-card-threats.pdf

[SDO]
Intel® Secure Device Onboard. Intel. 2017. URL: https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/intel-secure-device-onboard-product-brief.pdf

[She14]
The Constrained Application Protocol (CoAP). Z. Shelby; et al.. IETF RFC 7252. June 2014. URL: https://tools.ietf.org/rfc/rfc7252.txt

[Snyk]
A developer-first solution that automates finding and fixing vulnerabilities in your dependencies. URL: https://snyk.io/

[tcpdump]
tcpdump and libpcap project page. URL: https://www.tcpdump.org/

[Vol00]
AAA Authorization Framework. J. Vollbrecht; et al.. IETF RFC 2904. Aug. 2000. URL: https://tools.ietf.org/rfc/rfc2904.txt

[w3af]
Web Application Attack and Audit Framework. URL: http://w3af.org/

[wapiti]
Wapiti - The web-application vulnerability scanner. URL: http://wapiti.sourceforge.net/

[WATOBO]
WATOBO. The Web Application Toolbox. URL: http://watobo.sourceforge.net/index.html

[Wfuzz]
Wfuzz. The web application Bruteforcer. URL: http://www.edge-security.com/wfuzz.php

[Wireshark]

Wireshark project page. URL: https://www.wireshark.org/

[WoT-Architecture]
Web of Things (WoT) Architecture. Matthias Kovatsch; Ryuichi Matsukura; Michael Lagally; Toru Kawaguchi; Kunihiko Toumura; Kazuo Kajimoto. W3C. 9 April 2020. W3C Recommendation. URL: https://www.w3.org/TR/wot-architecture/

[WoT-Binding-Templates]
Web of Things (WoT) Binding Templates. Michael Koster; Ege Korkan. W3C. 30 January 2020. W3C Note. URL: https://www.w3.org/TR/wot-binding-templates/

[WoT-Scripting-API]
Web of Things (WoT) Scripting API. Zoltan Kis; Daniel Peintner; Johannes Hund; Kazuaki Nimura. W3C. 28 October 2019. W3C Working Draft. URL: https://www.w3.org/TR/wot-scripting-api/

[WoT-Thing-Description]
Web of Things (WoT) Thing Description. Sebastian Käbisch; Takuki Kamiya; Michael McCool; Victor Charpenay; Matthias Kovatsch. W3C. 9 April 2020. W3C Recommendation. URL: https://www.w3.org/TR/wot-thing-description/

[Yeg11]
Stevey's Google Platforms Rant. S. Yegge. Blog. Oct. 2011. URL: https://plus.google.com/+RipRowan/posts/eVeouesvaVX

↑