

# W3C Web of Things Security Testing Plan

Michael McCool and Elena Reshetova

January 30, 2019

# Draft W3C Note

- See
  - <https://github.com/w3c/wot-security-testing-plan>
- Rendered version:
  - <https://cdn.staticaly.com/gh/w3c/wot-security-testing-plan/master/index.html>
- To be submitted for publication at the same time as the CR transition
- The WoT charter only requires a security testing *plan*
  - Not actual test results
- Some "demonstration" security testing might be useful, however
  - There is not really a single test suite that will work for all implementations

# Document Outline

1. [Introduction](#)
2. [Best Practices](#)
3. [Functional Security Testing](#)
4. [Adversarial Security Testing](#)
  1. [Static Vulnerability Discovery](#)
    1. [Static Code Analysis](#)
    2. [Known vulnerability checking](#)
  2. [Runtime \(Dynamic\) Vulnerability Discovery](#)
    1. [Fuzz Testing](#)
    2. [Protocol Analysis](#)
    3. [Vulnerability Scanners](#)
  3. [Exploitation](#)
5. [Security Testing Plan Framework](#)
6. [Suggested Testing Frequency](#)
7. [Terminology](#)
8. [Summary](#)
- A. [References](#)
  1. [Normative references](#)

# Functional Security Testing

- Related to "Behavioral" assertions for security in spec:
  - Security mechanisms supported by a Thing should be consistent with its description in its Thing Description
  - Generally, we cannot constrain a server, which may be pre-existing device...
    - TD cannot "correct" insecure behavior by a described Thing
    - A TD should describe exactly what the Thing does and needs, no more or less
    - If the TD does not correctly describe a Thing, then the **TD** is incorrect, not the Thing
- "Functional Testing" just checks for correct implementation of described security mechanisms:
  - Accesses with proper authentication should be allowed
  - Accessed without proper authentication should be denied
  - Correct encryption and authentication mechanisms should be used

# Adversarial Security Testing

## ***Mimics what an attacker would try to do***

- We can re-use tools intended for attacking web services, although we should extend these with tools for IoT protocols (CoAP, MQTT, etc).
- Three stages:
  1. Information gathering
  2. Vulnerability discovery
  3. Exploitation
- Adversarial testing focuses on *Vulnerability Discovery*
  - We assume the attacker has FULL knowledge of the target, potentially including source code, and of course the information in the TD
  - Knowing how or whether a vulnerability is exploitable is important to testing only to prioritize which vulnerabilities should be fixed first (since we are not actually trying to break into the system)

# Discovering Vulnerabilities (1)

## **Static Code Analysis**

- Tools with access to source code can find coding practices likely to lead to vulnerabilities, such as missing boundary checks on arrays accesses, lack of input validation, etc.
  - Example tools: Klocwork, Coverity, Checkmarx.
- With access to the binary or library/package list, known vulnerabilities in any libraries used can be checked.
  - Example tools: Protecode, OWASP's Dependency-Check, Snyk, npm audit.

# Discovering Vulnerabilities (2)

## Runtime/Dynamic Code Analysis

- Does not require access to source, just external (network) interfaces
- **Fuzz Testing:** Send randomly-generated inputs to interfaces, designed to bypass early checks (eg for protocol, JSON syntax checks, etc.) but trigger errors (eg buffer overruns, unexpected parameters, etc) in backend code.
  - Example HTTP tools: Burp Suite, Wfuzz, Wapiti.
  - Example CoAP tools: FuzzCoAP, CoAP Peach Pit.
  - Example MQTT tools: ?
- **Protocol Analysis:** Look at data on wire, look for misconfigured headers, etc. Often manual, but some scripted tools available.
  - Example tools: Wireshark, tcpdump.

# Discovering Vulnerabilities (3)

## **Runtime/Dynamic Code Analysis**

- Does not require access to source, just external (network) interfaces
- **Protocol Analysis:** Look at data on wire, looking for misconfigured headers, insecure encryption options or combination of options, etc. Often manual, but some scripted tools available.
  - Example tools: Wireshark, tcpdump.



# Discovering Vulnerabilities (4)

## **Runtime/Dynamic Code Analysis**

- Does not require access to source, just external (network) interfaces
- **Vulnerability Scanner:** Attempt to run a known set of exploits or vulnerability trigger inputs against the target and report the outcomes.
  - Example tools: w3af, Burp vulnerability scanner, Nikto, WATOBO.

# Security Testing Plan Framework

- **Static Vulnerability Discovery**:
  - **Static Code Analysis**: If source code is available, a static code checker should be used to check for vulnerabilities in new purpose-written code.
  - **Known Vulnerability Checking**: If binaries and/or a list of package dependencies are available, a vulnerability checker should be used to check for known vulnerabilities in libraries used.
- **Runtime Vulnerability 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.
- **Exploitation Analysis**: This can be used to prioritize which vulnerabilities should be addressed first or whether a given vulnerable system can even be used in a given environment.

# Suggested Testing Frequency

- **Static Vulnerability 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 vulnerabilities can be discovered in the libraries that a program depends on. New vulnerabilities are discovered in existing libraries almost daily.
- **Runtime Vulnerability Discovery**: Should be also done on a regular basis, for example for each major release or development milestone.
- **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.

# Summary

- Have created a first draft of a testing plan; *please review*
- Provides a framework and a set of tool categories
- Lists example tools in each category
- Does not provide a "universal testing plan" for all possible WoT systems; this is not feasible
- We should, however, do some "demonstration" adversarial testing against WoT systems
  - We did not do this at this TestFest, unfortunately, but should do at least some basic testing against specific systems as soon as possible.

## **Limitations:**

- Few tools for CoAP, MQTT; plan does not consider physical attacks.