The Internet of Things (IoT) is an evolving concept and is described in various ways, one of the most common being “an infrastructure of interconnected objects, people, systems and information resources.” It is obvious to practitioners, however, that IoT is not a new concept. It is a new paradigm that is created and realized through the use of old concepts, methods and tools that have been around for many years in the world of IT and computing. Some of these concepts include remote function call and remote code execution.

From a security perspective, however, IoT exhibits new features and characteristics, such as the need to share additional types of data and operations. In contrast with older systems, IoT devices receive various types of inputs from other devices in the form of data and remote commands. IoT devices (e.g., smart locks or printers) are required to run a set of commands that are sent to them by remote entities, such as phones, on the same network or fetch utility libraries that are placed on scattered servers (e.g., JavaScript libraries). It is common for IoT devices to receive a set of machine instructions or commands for updates to the software that controls the physical device (e.g., firmware) or instructions to tell the device what exactly needs to be done. In technical terms, the devices can use well-known methods of remote procedure call, remote method invocation, dynamic class loading, and download of shared libraries and objects.

This article focuses on the security requirements around remote code execution, which means receiving and running code/commands from another system on the same network. In the case of IoT, this amounts to a device (the source of instructions) being able to control a connected “thing” from anywhere in the world. Used maliciously, remote code execution is a serious threat. It is sought after by hackers: being able to control a machine to do anything. Think, for example, of a malicious person being able to remotely control connected cars, medical devices or power plant control systems.

The article investigates security requirements of traditional remote code execution techniques in light of threat modeling results and expounds on the sections of security compliance regulations that stipulate those requirements.

Types and Scenarios of Remote Code Execution

Remote code execution is an umbrella term used for various types of code sharing in which an entity requests or receives some code and runs the code in its own environment. These are the common scenarios in which remote code execution occurs:

1. Use of common utility libraries placed on a remote server (e.g., JavaScript libraries). The functions are fetched from the server, but run on the client (e.g., the browser).
2. Dynamic loading of (compiled) classes. An example is Java dynamic class loading, which involves loading the binary form of a class (from a file or network location) that has been previously compiled from the source code.
Remote code execution is an umbrella term used for various types of code sharing in which an entity requests or receives some code and runs the code in its own environment.

3. Object serialization. Also known as marshaling, object serialization involves turning the object (structure, functions and attributes) into a new format (e.g., a byte stream) that could be easily transmitted and stored. Serialization and deserialization (sometimes called unserialization) is implemented in many languages such as Java and C#. JavaScript Object Notation (JSON) is built on the same concept; however, the goal of JSON is primarily data transfer rather than running remote code. Note that in this scenario, there is an instance of the class (an object with a set of properties) being transmitted. It is different from dynamic class loading in which the class (the binary) is loaded (usually only the structure, code and constants, and not a particular instance).

4. Remote procedure calls (RPC) or remote method invocation (RMI). There are numerous RPC protocols from older methods based on Common Object Request Broker Architecture (CORBA) and Open Software Foundation (OSF) RPC to newer models of Java application programming interfaces (API) for Extensible Markup Language (XML)-based RPC (JAX-RPC) and JAX-WS (Java API for XML-based web services). Calling web services such as Simple Object Access Protocol (SOAP) and Representational State Transfer (REST) web services could also be considered a special case of RPC. However, note that if the code runs on the host (e.g., server) and only the result is passed to the requesting device, the process will not qualify as RPC.

5. Device-specific operational commands. This includes commands sent to a device or an embedded system to carry out a sequence of tasks. One example is commands in the form of HP Printer Job Language (PJL). It is foreseeable that these types of proprietary and standard protocols will emerge and become widespread for numerous devices and applications as IoT matures.

6. Device-specific control commands (including firmware update commands). Firmware and basic input/output system (BIOS) update commands are very common for IoT devices, and the code may be received on the same channel as the device-specific commands (mentioned in scenario 5). There are also other standard and proprietary control commands that could be sent to devices according to IoT protocols.

7. Executable code embedded in files. Examples include code in the form of Postscript, ActiveX and Macros and embedded in files such as Microsoft Word, Microsoft Excel, PDF and Adobe Flash. The code is transmitted as part of the file and is executed at the destination. This concept is explained under the title of “mobile code” in American National Standards Institute (ANSI)/International Society of Automation (ISA) 62443 and NIST 800-53 compliance regulations.

Threat Modeling of Remote Code Execution

Figure 1 displays a simple data flow diagram as recommended by the Open Web Application Security Project (OWASP) application threat modeling method. The diagram shows the common elements of the described scenarios. The source of remote code is either a shared location (e.g., world-writeable locations on Android devices when dynamic class loading is used) or remote locations (e.g., a server on the Internet when a JavaScript library is loaded). A process or device (e.g., an IoT-embedded device) will eventually host and run the remote code. To determine the place of remote code and fetch data, a location resolution protocol is required.
middle attacks can also facilitate misrepresentation of spoof code as original code. These threats are relevant to all seven types and scenarios of remote code execution described in the previous section.

- **Tampering with data**—Any form of data tampering in transit or at rest (e.g., tampering with data through man-in-the-middle attacks) can fall under this category. A specific form of this vulnerability occurs when the code is loaded from a shared or world-accessible location (e.g., universal serial bus [USB] storage connected to a PC or a world-writeable location on an Android SD card). Tampered data, if handled by typical remote code execution libraries (such as the deserialization libraries outlined in scenario 3 described earlier) without additional protection measures, can lead to malicious code execution similar to those reported for Apache Commons libraries.

- **Information disclosure**—Any confidential data that are transmitted as part of an object (e.g., properties of a C# serialized object that constitute a person’s health record) are vulnerable to unauthorized disclosure (especially for scenarios 3 and 4). Some of the serialization/deserialization or RPC steps are delegated to the libraries that do not use encrypted channels. Developers may be unaware of the underlying mechanisms used by those libraries (e.g., if a particular library uses an encrypted channel for remote procedure calls).

- **Denial-of-service**—The availability of a system that executes remote code can be threatened by malicious code. A simple form of attack may involve creating huge payloads and sending them to the system as code. This can occur in all seven scenarios. Even if the system carries out integrity checks, a large amount of data can hinder normal operation of the system and can eventually lead to denial of service. Additional threats to availability are overreliance on a remote resource and lacking fail-safe procedures when that resource is unavailable. Another major vulnerability emerges from the use of third-party libraries that lack DoS protection.
• **Elevation of privilege**—There are numerous situations in which insecure remote code execution can lead to elevation of privileges. For example, Android applications can dynamically load Java classes (scenario 2). The application that loads the classes passes all of its permissions to the class that it is running. The loaded class receives the application’s permissions and privileges since the code is running in a new environment. Another example is if a device does not discriminate between various channels from which it receives commands (e.g., it does not separate its firmware update channel from the channel dedicated to its normal job), there is a risk of using permissions of one channel to perform unauthorized activities (scenarios 5 and 6).\(^{21}\) Third-party libraries may also be a vulnerability.

• **Repudiation**—Any other vulnerabilities can create opportunity for repudiation.

**Figure 2** depicts threats under various categories and also shows their relation to the security triad of confidentiality, integrity and availability. Based on all the identified threats and vulnerabilities, this article provides eight rules of remote code execution that mitigate these areas of security risk.

### A Prescriptive Approach to Securing Remote Code Execution

This section outlines a set of security requirements that mitigate the risk and threats relating to low-complexity IoT devices.

1. **Encrypt fields, obfuscate classes and use encrypted channels.** This requirement stems from the goal of confidentiality and the possibility of information disclosure. There are several ways to maintain the confidentiality of the data transmitted as part of objects or procedures by:
   - Encrypting individual fields (e.g., properties of the objects). Secure key management and distribution, especially for stand-alone devices, is an important undertaking in this case.
   - Obscuring or obfuscating code and objects. Binaries of compiled classes are easy to reverse engineer. By using decompilers, hackers can obtain the original code and any constants in the code. Obfuscation is not a panacea, but it adds a layer of defense, i.e., it should not be treated as the sole security measure. More information on this can be found in documents relating to the OWASP project on code reverse engineering.\(^{22}\)
   - Communicating through an encrypted channel (e.g., Secure Sockets Layer [SSL]/Transport Layer Security [TLS] channels). It is important to keep an eye on the studies of SSL/TLS vulnerabilities and apply the result of those studies. There are several guidelines on the types of encrypted channels to use and what to avoid.\(^{23}\) For example, SSL v2.0 and 3.0 are not secure, and SSL libraries need constant updates due to various vulnerabilities that are regularly discovered (e.g., Heartbleed, Browser Exploit Against SSL/TLS [BEAST], Factoring RSA Export Keys [FREAK] and Compression Ratio Info-leak Made Easy [CRIME] attack vector). An IoT device with no update capability will become insecure in no time. Implementing SSL/TLS on low-complexity devices can help mitigate these risks.

2. **Spoofing Identity**—Misrepresentation of the code’s source and Misrepresentation of the network addresses.

3. **Data Tampering (Integrity)**—Change of code in transit (man-in-the-middle) and Change of code at rest (world-writeable locations).

4. **Information Disclosure (Confidentiality)**—Confidential fields in serialized objects/classes and Unencrypted channels used by third-party libraries.

5. **Denial-of-Service (Availability)**—Large amount of counterfeit data (code or object) and Overreliance on remote resources.

6. **Elevation of Privilege**—Running malicious code in host’s environment and Lack of distinction between types of code.

**Figure 2**—Security Threats of Remote Code Execution

6. **Separate the channels of code transfer.** Make sure data on ordinary channels of data transfer (e.g., operational commands for a printer) cannot be used to carry out malicious remote code execution. Restrictions of update commands (e.g., signature requirements) should be different from the ones for ordinary commands.

7. **Verify that third-party libraries comply with the previous requirements.** Do not feed the libraries user data unless all the other checks have been carried out. For example, if the library is used before size-checking, an organization may make itself vulnerable to DoS attacks.

8. **Avoid overreliance on remote resource and have a fail-safe plan.** Devise an alternate plan for the situations that the remote resources become unavailable. If continuing the process may become impossible due to unavailability of those resources, design a fail-safe plan.

**Figure 3** displays a best practice for object serialization, in which the transmitted object is sealed (encrypted), then signed and then transferred. On the receiver side, the object is first size checked, then the signature is verified and finally decrypted.

**Relation to Major Security Compliance Regulations**

ANSI/ISA 62443, under security requirement (SR) 2.4 (mobile code), instructs control systems to enforce usage restrictions on mobile code technologies that include: preventing the execution of mobile code, requiring proper authentication/authorization for origin of the code, restricting mobile code transfer and monitoring the use of mobile code.

NIST 800-53r4 in the system and communications protection section (SC-18, mobile code), recommends execution of remote code in a confined environment. In the section on system and information integrity, SI-7 (15), the standard stipulates code signing and verification.
The vulnerabilities described in this article are among the Common Weakness Enumeration (CWE)/SANS top 25 listed vulnerabilities: Download of code without integrity check (CWE-494) and inclusion of functionality from untrusted control sphere (CWE-829).28

The European Banking Authority’s final guidelines on the security of Internet payments state that software delivered via the Internet needs to be digitally signed by the payment service provider.29 In the Manufacturer Disclosure Statement for Medical Device Security (MDS2), the manufacturers are required to declare if the device protects transmission integrity (TXIG) and if there are any mechanisms to ensure that the installed code or update is manufacturer authorized (15-2).30

Conclusion

Sending remote code in various forms to “things” and asking for instructions by those things, especially for device and firmware updates, is common and will become a more common practice in the IoT ecosystem. Since IoT devices have interaction with the physical world and, in many cases, those interactions are remotely controllable (whether in a thermostat or in the collision-prevention system of a connected car), the consequences of bypassing security controls are immense. Unsafe execution of remote code can lead to a bypass of safety controls and can cause physical harm to consumers of IoT products. Therefore, all security measures and relevant compliance regulation sections should be considered before any attempt to design security for IoT solutions.

Endnotes

1 ISO/IEC, SWG 5 agreed on this definition of IoT in 2014: “An infrastructure of interconnected objects, people, systems and information resources together with intelligent services to allow them to process information of the physical and the virtual world and react.” ISO/IEC JTC 1, “Internet of Things (IoT) Preliminary Report,” 2014


7 Ibid.
11 Richardson, L.; S. Ruby; RESTful Web Services, O’Reilly Media Inc., USA, 2007
13 Op cit, Athreya
14 Op cit, Klauck
19 Shinder, D. L.; M. Cross; Scene of the Cybercrime, Syngress, USA, 2008
24 Op cit, Cui
26 Op cit, ANSI/ISA
27 Op cit, NIST