Inquiring Into Security Requirements of Remote Code Execution for IoT Devices

Do you have something to say about this article?
Visit the *Journal* pages of the ISACA web site *(www.isaca.org/journal)*,find the article and click on the Comments link to share your thoughts.

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."[1] 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.[2, 3] 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.

Farbod Hosseyndoust Foomany, Ph.D.
Is a senior application security researcher (technical lead) at SD Elements/Security Compass. Foomany has been involved in various academic research and industry projects in the area of secure software development, secure design for enterprise applications, signal processing and evaluation of biometric verification systems. Foomany is currently involved in a project that aims to investigate and formulate the security requirements of the IoT systems.

Ehsan Foroughi, CISM, CISSP
Is the vice president of the SD Elements division at Security Compass. Foroughi is an application security expert with more than 10 years of management and technical experience in security research and an extensive product management, development and reverse-engineering background. Prior to joining Security Compass, he managed the vulnerability research subscription service for TELUS Security Labs (previously Assurent).

Rohit Sethi
Is a specialist in software security requirements. He has helped improve software security at some of the world's most security-sensitive organizations in financial services, software, e-commerce, health care, telecommunications and other industries. In his current role, Sethi manages the SD Elements team at Security Compass. Sethi has appeared as a security expert on television outlets such as Bloomberg, CNBC, FoxNews, CBC, CTV and BNN. Sethi has spoken at numerous industry conferences.

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).[4]
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.[5]

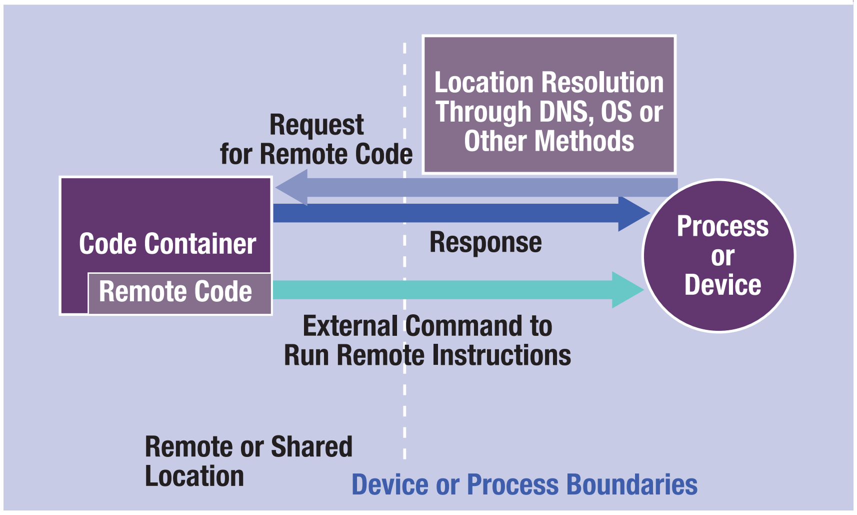
3. **Object serialization.**[6] Also known as marshaling, object serialization involves turning the object (structure, functions and attributes) into a new format (e.g., a byte stream) that could be easily transmitted and stored. Serialization and deserialization (sometimes called unserialization) is implemented in many languages such as Java[7] and C#.[8] 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)[9] 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).[10] Calling web services such as Simple Object Access Protocol (SOAP) and Representational State Transfer (REST) web services[11] 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).[12] 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.[13, 14]
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[15] and NIST 800-53[16] compliance regulations.

“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.”

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.[17] 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

Figure 1—Flow Diagram of Typical Forms of Remote Code Execution


Location Resolution Through DNS, OS or Other Methods
Request for Remote Code
Code Container
Remote Code
Response
Process or Device
External Command to Run Remote Instructions
Remote or Shared Location
Device or Process Boundaries

Source: Farbod H. Foomany, Ehsan Foroughi and Rohit Sethi. Reprinted with permission.

service is utilized. For example, in the case of files in shared locations, the operating system can handle the requests and send them to the right resource. For Internet access, domain name servers translate the resource's address to an Internet Protocol (IP) address.

One important idea displayed in **figure 1** is that there are two conceivable flow directions. In some cases, the host/device initiates the request for the remote code. In others, the device receives the commands even though it has not necessarily initiated the request. For example, a printer may have a channel for receiving remote commands for performing various jobs.

Using spoofing identity, tampering with data, repudiation, information disclosure, denial-of-service (DoS), and the elevation of privilege (STRIDE) threat modeling technique, the security threats of remote code execution can be classified and summarized as follows:[18]

- **Spoofing identity**—Domain name system (DNS) spoofing can cause requests for one resource to be sent to another.[19] Other types of man-in-the-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.[20]
- **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.

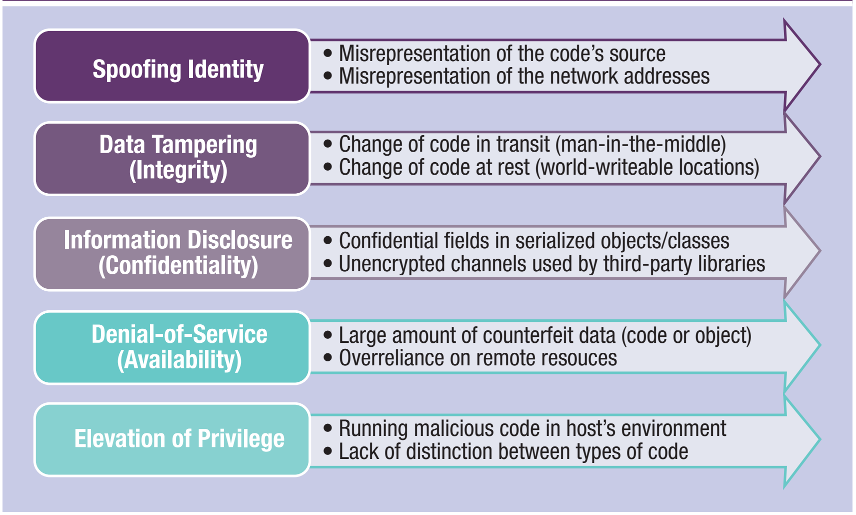
Figure 2 depicts threats under various categories and also shows their relation to the security triad of confidentiality, integrity and availability. Based on all the identified threats and vulnerabilities, this article provides eight rules of remote code execution that mitigate these areas of security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

Figure 2—Security Threats of Remote Code Execution



Spoofing Identity
• Misrepresentation of the code's source
• Misrepresentation of the network addresses
Data Tampering (Integrity)
• Change of code in transit (man-in-the-middle)
• Change of code at rest (world-writeable locations)
Information Disclosure (Confidentiality)
• Confidential fields in serialized objects/classes
• Unencrypted channels used by third-party libraries
Denial-of-Service (Availability)
• Large amount of counterfeit data (code or object)
• Overreliance on remote resouces
Elevation of Privilege
• Running malicious code in host's environment
• Lack of distinction between types of code

Source: Farbod H. Foomany, Ehsan Foroughi and Rohit Sethi. Reprinted with permission.

devices is a challenge that may cause reliance on solutions a or b mentioned earlier instead of encrypting the entire stream of data, which is required by SSL/TLS.

2. **Check the size of payload.** Before anything else—even before checking the code signature—check the payload size and avoid dealing with large counterfeit lumps of data that are sent as part of a DoS attack.
3. **Sign the code or use protocol-specific authentication methods.** Signing the code and avoiding running any unsigned code is the single most important security measure. If using encrypted channels (e.g., TLS), validate the certificate and chain of trust. Signed code is not obviously secure code, but signature, at a minimum, manifests the integrity of code.[24]
4. **Do not run any part of the code before checking size and signature.** No constructor or overridden methods should be executed by the code or any third-party library before all security checks are performed. For example, a library contains a set class that handles serialized objects. The set class should not receive the external inputs before size/signature checking. It also should not run any part of the classes (e.g., constructors or overridden *readObject*() methods) before the object is validated.
5. **Sandbox the remote code execution process and memory.** Do not let the code run in a shared memory or storage space to which other processes have access and *vice versa* (especially the update commands). Sandboxing (direct access to other applications' storage and memory) does not protect against any of the vulnerabilities mentioned so far. However, since a lack of sandboxing can void other security measures (such as signature verification), sandboxing contributes to strengthening other defense mechanisms. In the case of a BIOS update, for example, researchers have shown that a buffer overflow can enable executing the unsigned portion of the update package.[25]
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

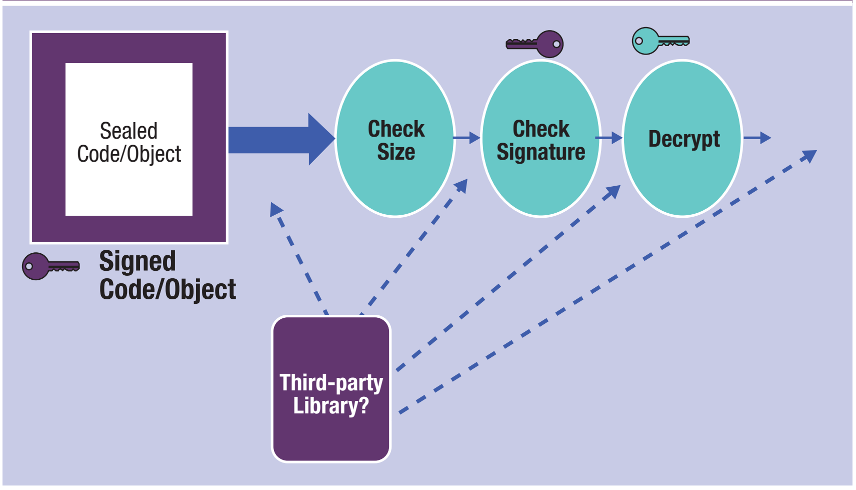
ANSI/ISA 62443, under security requirement (SR) 2.4 (mobile code), instructs control systems to enforce usage restrictions on mobile code technologies that include: preventing the execution of mobile code, requiring proper authentication/authorization for origin of the code, restricting mobile code transfer and monitoring the use of mobile code.[26]

NIST 800-53r4 in the system and communications protection section (SC-18, mobile code), recommends execution of remote code in a confined environment.[27] 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]

Figure 3—A Secure Three-part Procedure for Object Serialization



Source: Farbod H. Foomany, Ehsan Foroughi and Rohit Sethi. Reprinted with permission.

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

2 Athreya, A. P.; B. DeBruhl; P. Tague; "Designing for Self-configuration and Self-adaptation in the Internet of Things," 9th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing, CollaborateCom, 2013

3 Klauck, R.; M. Kirsche; "Chatty Things—Making the Internet of Things Readily Usable for the Masses With XMPP," 8th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing, CollaborateCom, 2012

4 Flanagan, D.; *JavaScript: The Definitive Guide: Activate Your Web Pages*, O'Reilly Media Inc., USA, 2011

5 Gosling, J.; *et al*.; "The Java Language Specification–Java SE 8 Edition," Oracle America, 2014

6 Deitel, P.; H. M. Deitel; *Java for Programmers, Second Edition*, Prentice Hall Professional, USA, 2011

7 *Ibid*.

8 Hericko, M.; *et al*.; "Object Serialization Analysis and Comparison in Java and .NET," *ACM Sigplan Notices*, vol. 38, iss. 8, August 2003, p. 44-54

9 Ben-Natan, R.; *Corba: A Guide to Common Object Request Broker Architecture*, McGraw-Hill Inc., USA, 1995

10 Fisher, M.; *et al.; Java EE and .NET Interoperability: Integration Strategies, Patterns, and Best Practices*, Prentice Hall Professional, USA, 2006

11 Richardson, L.; S. Ruby; *RESTful Web Services*, O'Reilly Media Inc., USA, 2007

12 Hewlett-Packard, Printer Job Language Technical Reference Manual, 2003, *www.hp.com*

13 *Op cit*, Athreya

14 *Op cit*, Klauck

15 ANSI/ISA, *Security for Industrial Automation and Control Systems Part 3-3: System Security Requirements and Security Levels*, USA, 2013

16 National Institute of Standards and Technology, *Security and Privacy Controls for Federal Information Systems and Organizations*, NIST Special Publication 800-53r4, USA, 2013

17 Open Web Application Security Project (OWASP), "Application Threat Modeling," *https://www.owasp.org/index.php/Application_Threat_Modeling*

18 Shostack, A.; *Threat Modeling: Designing for Security*, John Wiley & Sons, USA, 2014

19 Shinder, D. L.; M. Cross; *Scene of the Cybercrime*, Syngress, USA, 2008

20 The Apache Software Foundation Blog, "Apache Commons Statement to Widespread Java Object De-serialisation Vulnerability," 10 November 2015, *https://blogs.apache.org/foundation/entry/apache_commons_statement_to_widespread*

21 Cui, A.; M. Costello; S. Stolfo; "When Firmware Modifications Attack: A Case Study of Embedded Exploitation," Presented at the NDSS symposium, 2013

22 See OWASP's Reverse Engineering and Code Modification Prevention Project, *https://www.owasp.org/index.php/OWASP_Reverse_Engineering_and_Code_Modification_Prevention_Project.*

23 Ristic, I.; "SSL/TLS Deployment Best Practices," 2013, *https://www.ssllabs.com/downloads/SSL_TLS_Deployment_Best_Practices.pdf*

24 *Op cit,* Cui

25 Wojtczuk, R.; A. Tereshkin; "Attacking Intel BIOS," BlackHat, Las Vegas, Nevada, USA, 30 July 2009

26 *Op cit*, ANSI/ISA

27 *Op cit*, NIST

28 Common Weakness Enumeration, "2011 CWE/SANS Top 25 Most Dangerous Software Errors," 2011, *http://cwe.mitre.org/top25/*

29 European Banking Authority, Final guidelines on the security of internet payments, 19 December 2014, *https://www.eba.europa.eu/documents/10180/934179/EBA-GL-2014-12+(Guidelines+on+the+security+of+internet+payments)_Rev1*

30 HIMSS/NEMA, Manufacturer Disclosure Statement for Medical Device Security, 2013, *www.nema.org/Standards/Pages/Manufacturer-Disclosure-Statement-for-Medical-Device-Security.aspx*