Security Challenges in CPS and IoT: from End-Node to the System

Kelvin Ly and Yier Jin
Department of Electrical and Computer Engineering, University of Central Florida
rangertime@knights.ucf.edu, yier.jin@eecs.ucf.edu

***Abstract*—The rising of cyber-physical systems (CPS) and Internet of Things (IoT) has significantly increase the industrial productivities and customer convenience. However, the widely distributed CPS and IoT systems also breeds new challenges among which security is a major concern. In order to help build secure and resilient CPS/IoT systems, systematic analyses are performed on CPS and IoT from individual smart devices which serve as the end nodes to the entire system. The analysis results will help identify the vulnerabilities of CPS/IoT systems and/or provide guidance on how to build security into these systems. Security mitigation methods will also be discussed which can help balance security with other criteria such as safety and performance.**

I. Introduction

As it stands now, the Internet of Things is suffering from many widespread security flaws. Most if not all systems suffer from security vulnerabilities, ranging the full gamut of faults, from an attacker being able to leak private information to attaining complete remote control over the system. A 2015 HP report determined that 8 out of the 10 devices they tested had privacy concerns [1]. Along with high impact exploits that are discovered on a roughly yearly basis, such as the recent exploits that allowed complete remote control of various vehicle models, there are many reports released monthly or even weekly revealing flaws in new IoT or CPS systems [2]–[4].

Cyber physical systems have been shown to suffer similar flaws, with even greater consequences because of infrastructure's growing reliance on cyber physical systems. CPSs traditionally have their network separate from the Internet to ensure their security. In their original context, CPS devices would be physically secured by simply being out of reach of any attackers, and could rely on this physical isolation for protection, resulting in standards that were design with this sort of scenario in mind. More recent CPSs have been deployed in places where they rely on direct connection to the Internet, or in places where an attacker could potentially have physical access to the device, vastly changing the threat model and undermining existing CPS standards that were not designed with these possibilities in mind. For instance, Modbus, a common CPS protocol standard, by itself does not include any authentication in its protocol, and relies on plaintext communication across all connections [5].

A lack of awareness of security during the design of the devices is one of the main causes of these vulnerabilities, as IoT/CPS devices tend to have a very large surface area that can be attacked, and consequently suffer many security flaws if security was not a primary concern during their development. Many flaws in security may be resolved through patches sent after its release, but some, such as inadequate hardware security, cannot. Similarly, a lack of understanding and knowledge of security and proper security practices is considered a potential cause for these flaws, as many of the problems found were caused by simple design mistakes, such as not validating firmware at booting process, exposing debugging ports on deployed devices, and communicating across plaintext channels.

IoT/CPS systems will continue to grow in complexity and ubiquity, as they become smarter and find more applications in industry. Consequently, if no work is done to improve their security, the problems facing IoT/CPS systems now will only grow in severity. Thus, a systematic approach towards examining IoT/CPS devices is needed, and one potential system for doing this is outlined in the rest of the paper. This systematic approach was developed in two steps, the first of which was gathering data on existing IoT/CPS security vulnerabilities into a database for better analysis, and the second of which was a case study of developing a IoT/CPS system (which will serve as a testbed for future endeavours) and recording how security fit into the development process.

II. Related Work

Babar et al provided a taxonomy to classify threats to IoT systems [6]. Their model divides threats into six categories, storage management, embedded security, physical threat, dynamic binding, communication threat, and identity management. They try to model security in IoT devices as a cube, where security, privacy, and trust forming the three dimensions of the cube. While this model may be useful for potentially scoring the quality and thoroughness of security in IoT devices, it does not provide an intuitive system or methodology for examining security during the development and design of IoT/CPS devices or systems.

A security framework for IoT is then developed in [7]. The authors first described another categorization of IoT vulnerabilities, dividing threats into physical attacks, side channel attacks, environmental attacks, cryptanalysis attacks, software attacks, and network attacks. From these attacks a set of requirements were created to ensure security of an IoT device, and from these requirements the framework was developed. Their framework outlines the main components that need to be present in an IoT device to ensure its security. The framework details the different parts of an IoT system, and

978-1-4673-9039-2/16 $31.00 © 2016 IEEE
DOI 10.1109/ISVLSI.2016.109

how each one contributes to the overall security of the device. However, their framework is very high level and does not actually provide recommendation on the kinds of technologies that should be used. This framework could potentially be used as the foundation for a more detailed framework, but as it is described here it cannot be applied in practice during development to improve security.

IoT security has even been the topic of an IETF draft [8]. This divided device operation into three main phases: 1) bootstrapping where the security parameters are added to all the devices in the system through a trusted authority; 2) operation where the device is operating normally; and 3) maintenance where the device, and potentially its security parameters, are updated, or the device is removed from the system. This draft divides threats along the layers of the network on which the device is operating. It also describes the constraints that IoT devices face, with respect to their limited computational resources, and provides recommendations on the types of communication protocols that should be used in the the system during each devices' different phases, as well as listing the security requirements for operation during each phase. The authors provide a very comprehensive overview of many of the problems faced in IoT security from a network perspective, with each decision and idea carefully backed with technical details that allow this work to stand as a good starting point for designing security in IoT architectures. They also provide different security profiles that may be used in different situations depending on the importance of the system, from home use to advanced industrial use.

There are a few minor flaws in this work with respect to acting as a guide for developing IoT, however, due to its focus mainly on the communication side of IoT/CPS security. The draft fails to acknowledge the importance of securing the hardware and software in the device, and more importantly does not acknowledge that these are also important aspects to secure in an IoT device. Thus, using only this draft as a reference, an IoT designer may fail to notice the importance of securing other aspects of their devices, such as securing the bootloader, signing firmware updates, and using appropriately secure web interfaces.

One thing to note about all the works listed here is that they approach the problem of IoT primarily from the perspective of a defender rather than an attacker. This paper will discuss security from an offensive point of view, allowing the design to be more resilient against attacks by requiring the designer to attack their own design in the same ways an attacker might.

III. IoT and CPS Security Database

A database has been developed that has been aggregating data on IoT/CPS device vulnerabilities as they are disclosed. The database summarizes existing IoT/CPS device security and vulnerabilities from three perspectives: IoT/CPS device vulnerabilities, case studies, and design rules for trusted IoT/CPS construction.

Device vulnerabilities are exploits that have been discovered and applied to existing COTS IoT devices. These form the bulk of the data collected by the database, and will continue to grow as more exploits are found and added. The vulnerabilities can be statistically analyzed to determine what the most commonly found flaws in IoT devices are, and, given more time to collect data, perhaps trends in exploits over time may be found.

Design rules are then developed from the classes of vulnerabilities discovered in the last step. These are not amended as often as vulnerabilities are added because a single design rule can rule out an entire class of vulnerabilities. New design rules could potentially be developed when there are exploits that fail to be patched through the application of the design rules, but this has yet to happen.

Case studies attempt to study how difficult it is to apply the design rules developed in the preceding step to actual IoT device development, and places where adhering to the design rules proved to be difficult can be ameliorated with the modification of the development framework to make secure development in those areas easier. There is also room here to experiment with automating the verification of design rules during the design process. Currently there has been a single design study, done during the development of a robotic arm CPS security testing platform, that will be discussed in greater detail below.

A. Security Vulnerabilities and Case Studies

This database categorizes attacks into different categories, related to the different stages of device operation that could have vulnerabilities [9]. This is visualized in Figure 1.

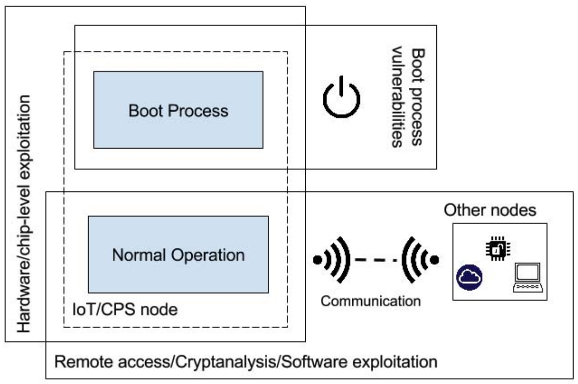
Boot Process
Boot process vulnerabilities
Hardware/chip-level exploitation
Normal Operation
IoT/CPS node
Communication
Other nodes
Remote access/Cryptanalysis/Software exploitation

Fig. 1: The threat taxonomy used in the IoT database.

Boot Process Vulnerabilities. The boot sequence is a common component to target in attacks. This is primarily because the boot process is the root of trust as well as the starting point of the device's operation, and consequently any attacks that compromise this component are capable of controlling anything that occurs in a subsequent stage of operation. The compromise of the Google Nest Thermostat is one recent example of this type of vulnerability [10], [11]. In this compromise the attacker modified the first stage boot loader (`x-loader`), allowing them to replace the second stage boot-loader (`u-boot`) that would let them set custom parameters for the kernel. This attack is trivially extended to allow for arbitrary payloads using the custom second stage boot loader [11]. Mitigation methods to defend against this were discussed in [12], [13].

Hardware Exploitation. Hardware level exploitation is critical as it is often overlooked by designers, who focus more on security at the software or firmware levels. These attacks rely on the hardware itself, often including tactics such as looking for debugging ports the designer has left exposed, modifying external Flash memories, glitching address lines, etc. For instance, the Xbox 360 was exploited by using a timing attack a bypass a security check that would prevent downgrading the kernel to an older one with known vulnerabilities [14].

Chip-Level Exploitation. Chip-level exploitation involves semi-invasive and invasive attacks on the chip itself. These are a serious threat to smart devices, which rely on trust boot sequences which rely on hardware chip assets for their security. Previously security information such as encryption and decryption keys, along with other sensitive information, was considered secure if it was stored on-chip. However, newly developed methods can extract this information, and consequently disrupt the security claims based on the assumption of on-chip security. For example, researchers were able to extract a stored AES key from the internal memory of an Actel ProASIC3 FPGA by “bumping” the internal memory [15].

Encryption, Hash Function and Authentication Implementations. Similarly, many attacks today result from weak authentication mechanisms. While system designs will impose strong authentication mechanisms, e.g. x.509 certificate based TLS [16], unless the credentials (e.g., keys) are securely stored they can be subject to attack. As IoT devices are now exposed in open and public spaces, the ability for any attacker to recover such credentials becomes a trivial attack; once the keys are recovered, those identities are then compromised obviating the security properties afforded by any encryption mechanism. For example, the Sony PlayStation 3 firmware was downgraded due to a series of vulnerabilities in weak cryptographic applications [17], [18].

Backdoors in Remote Access Channels. For the sake of convenience, smart devices now commonly come equipped with channels that allow for remote communication and debugging after manufacture. These channels are often used for over-the-air (OTA) firmware upgrades. Any insecurities in the protocol used for the OTA firmware upgrade would allow an attacker control over the firmware of a device, and consequently complete control over the device. Additionally, manufacturers may leave in APIs used during development that would allow arbitrary command execution, or they may not properly secure some communication channels. One recent example of this type of vulnerability is in the Summer Baby Zoom WiFi camera, which used hardcoded credentials for securing administrator access [19]. These types of attacks can be mitigated through requiring the user to change the default credentials, sanitizing input strings to avoid code injection, etc.

Software Exploitation. As with traditional general purpose computing platforms, smart devices are also vulnerable to software-level vulnerabilities in much the same way. Code from general purpose computing software is often reused in smart device software stacks, passing along any vulnerabilities in the process. Similarly, software patches must be applied to prevent these attacks. Recent examples of this type of attack include stack overflow attacks in glibc and elsewhere in the code base that affected multiple smart house devices [20], [21]. The approaches to prevent software exploitation in traditional computing platforms can be applied to smart devices, but many of these solutions are impractical due to the resource constraints [11], [22], [23].

B. Design Rules for IoT/CPS Security

An IoT/CPS device is then vulnerable to attacks through the vectors described above. Consequently, defending against an attack can be accomplished by simply ensuring that the above vectors are defended. For example, to defend against boot process vulnerabilities, the boot process must be made secure through the following countermeasures.

- Ensuring the integrity of the bootloader (i.e., through the use of a checksum or preferably a signature);
- Protecting the bootloading chain through verifying the bootloader at boot and using encrypted data streams during firmware updates;
- Disabling the device’s debug mode so that the attacker cannot modify the bootloader after the device has been shipped.

These rules match directly with the properties of the boot loader that can be exploited, and thus all the vulnerabilities are directly countered with protection methods. This avoids adding security features that do not match existing and common security threats, avoiding adding too much overhead while providing security where it is needed.

Similar design rules were developed for all the other categories described above.

IV. Case Study on Robotic Arm System

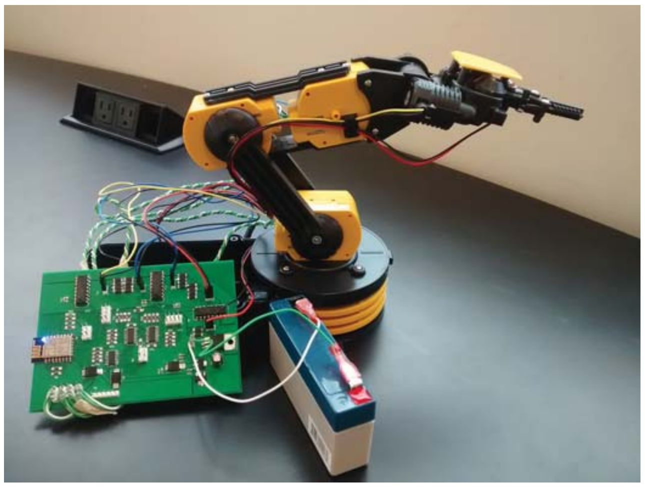
Fig. 2: Robotic arm designed for CPS testing platform.

A robotic arm platform is being developed that is designed to model a CPS/IoT system, displayed in Figure 2. It consists of two robot arms that can be accessed using a web based interface, and are also capable of machine-to-machine communication to allow for automated cooperation. This system tries to match features found in IoT/CPS systems, such as relying on wireless communication between two different units and a user, being powered by batteries, using a low-power, low-cost computation unit, requiring a real-time feedback loop between its sensors and actuators, and allowing for remote firmware updates.

Similar to how the platform itself is a model of a CPS/IoT system, the development of the platform can be used as a model of IoT development. It can be expected that the technical challenges and decisions made during development would be representative of those found and made by other IoT designers. Consequently, the analysis developed from the data from the IoT database can be applied here to see how well it would apply during the development of a real IoT system.

The development process can be divided into three major stages: 1) hardware design, where the custom printed circuit board that would control the robot arms was designed; 2) software design, where the major details of the system's communication were decided; and 3) software development, where the firmware that implements all of the desired features was developed.

A. Hardware Design

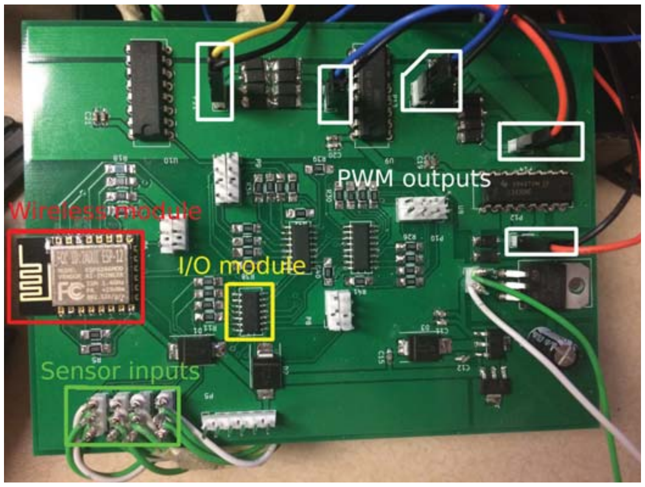

Fig. 3: Printed circuit board designed and fabricated for the robotic arm control.

The first important decision during this phase was determining the form of wireless communication would use. This decision would substantially affect the device's vulnerabilities to cryptanalysis and remote access. The choice here would greatly affect the design, as different chipsets would be needed for different communication channels. Many low power wireless communications options have become available, such as Zigbee, low-power Bluetooth, 6LoWPAN, but standard WiFi was chosen because it would be easy to work with, with respects to interacting with the IoT network, and it has a mature security scheme. Additionally, some of the low power options listed above were not designed with security in mind, resulting in vulnerabilities. For example, Bluetooth is known to be vulnerable to eavesdropping and man-in-the-middle attacks. Of course, additional security could be added at the application layer, but the resulting complexity was deemed not worth the effort, so the simpler choice of using an appropriately secured Wifi connection was chosen instead.

The choice of microcontroller is vital in ensuring the security of the entire system, because it affects many aspects of the device's overall design, from the availaility of GPIO to the limits of the device's computational abilities. During this step a lot of effort was made to try to protect against boot process attacks. It was decided that a low-performance, low-power system-on-a-chip would be used, because the device did not need to run a full OS to perform its duties, and using a single device for both communication and processing reduced the surface area for attacks. For the sake of easy development, there were two main choices in terms of low-power WiFi-enabled SoCs, the ESP8266 and the CC3200.

Both of these store the program data in an external Flash chip, which means in both cases the firmware cannot be made unwriteable with physical access. Both additionally have some functions stored in ROM, which helps to reduce the surface areas for attacks, as less of the code can potentially be modified. The ESP8266, however, comes packaged as module that has the ESP8266 chip and a Flash module contained within an RF enclosure that would at least discourage an attacker from attempting to gain access to the Flash module's SPI interface. The ESP8266's bootloader is, however, stored in the Flash, which means if it were programmed to verify the firmware during boot, it could potentially be modified to bypass the verification check.

The ESP8266 was chosen in the end because it had a more mature community and documentation, and the additional firmware security was an added benefit. This demonstrates one facet of design where security was not the deciding factor in a design decision, and also that there is a lack of popular, wireless SoCs that meet all the requirements to prevent attacks on the hardware level.

To ensure low latency and fast response in the device overall, a second, smaller microcontroller was added to handle the PWM and pulse counting used to control the motors in the arm and sense the movement of the arms. This microcontroller was chosen to be as small as possible, so an attacker gaining control of this chip would have more difficulty in gaining control of the system through it. An Atmel ATtiny44A was used here, because it provided enough GPIO to drive the actuators and read data from all the sensors, and had the additional benefit of having the program memory stored on chip. The chip has fuses that can be blown to prevent reprogramming, a very important security feature that could be used in deployed chips to reduce attack surface area.

SPI is used to communicate between the two microcontrollers. This communication is unencrypted because the ATtiny44A would be unable to decrypt and then process the request in a sufficiently fast matter. However, to improve security, this trace could in theory be buried in a multilayer PCB design to make it considerably harder to intercept and modify. This was not done in this case to, as might be the case in industry, avoid the extra cost of an additional PCB layer in a relatively simple design.

There were two sets of programming ports that were left exposed on the final PCB design, one for each microcontroller. This may seem like a major design flaw, because this means both the microcontrollers can be easily reprogrammed by an attacker. However, the one for the Atmel chip may be nonfunctional when deployed if the flash writing fuse on the chip is blown after it is programmed during manufacture, leaving the programming port for the ESP8266. The barrier to reprogramming the ESP8266 can be increased using all sorts of design tricks, such as tying the boot mode pins to pins on the Atmel chip. The Atmel chip could be programmed to set the boot mode pins to the correct levels for programming at the factory, and then after the ESP8266 has been programmed, the final program for the Atmel could keep the boot mode pins at the levels that would only allow for normal operation. However, neither blowing the fuses on the Atmel or using this rather complicated programming mode were used on the robotic arm due to the expectation that both will need to be reprogrammed fairly often as development continues. The finalized design can be seen in Figure 3.

B. Software Design

The use of an ESP8266 ruled out running a full Linux environment, which had multiple effects on the software design. In terms of security, it greatly improved security as there were now no services that could accidentally be left open, and no root shell that could be reached through the mis-configured services. Given that about forty percent of the vulnerabilities in the database involved reaching a root shell, this potentially is a major improvement in security. However, it also meant that most of the code would be written by hand at a very low level, and would not receive the thoroughly auditing and testing the code in the Linux environment has been put through.

The development environment around the ESP8266 API is very mature, and consequently there were many frameworks available to develop on. These include the proprietary SDK Espressif offers, an open sourced version of the SDK, nodemcu, which essentially runs a Lua interpreter on user-provided code, and an implementation of the Arduino API on the ESP8266. The open source version of the SDK was chosen, as the design required access to many of the more advanced features of the device which were not included the last two frameworks. However, using the SDK requires using C as the main development language rather than a newer, more memory-safe language. This does reduce the potential area of attack, however, as using the SDK natively does reduce the overall amount of code being run. Thus, here some security was sacrificed for performance.

It was decided that the device would support HTTP to interact with the user, and MQTT for machine to machine communication. These are both popular and commonly used, making development fairly easy. In terms of security, both of these can be used on across a TLS connection (which the ESP8266 supports natively) to provide confidentiality, and there are also well-known methods for adding authentication to both protocols. TLS has been cryptanalyzed thoroughly, so it should be sufficiently secure from most threats when it is properly configured [24]. Additionally, in both of these cases it is practical to write all the network code without authentication and then add in the appropriate authentication cases after the unauthenticated code has been adequately tested. Therefore, security here was both convenient and easy to add to the design, as it composes easily with the existing, unsecured protocols.

An open source alternative bootloader was used, as this provided more control over the bootloader that would allow signature checking on boot, and additionally improved transparency and allowed auditing of the bootloading code for potential vulnerabilities.

C. Software Development

Software vulnerabilities comprise the majority of the vulnerabilities found in the IoT database. Consequently, during attention was paid during development to reduce the likelihood of adding bugs and vulnerabilities to the code, specifically with respects to the most common sources of exploitable bugs.

There were existing open source projects that could have provided code for the HTTP server and client, but it was deemed easier to write by hand to avoid the request of adding much code to the codebase. Adding all that code would have used a large amount of the limited Flash on the device for features that would not be used, and would expand the potential area for software faults such as buffer overflows to occur. Code written tailored for this purpose would be much smaller, at the expense of being less thoroughly tested. Preferably, a more memory-safe language would be used, but none is available in this area yet. The continued prevalence of C as the only language to program on embedded platforms seems to be a potential source of security vulnerabilities, as it requires much developer effort to ensure memory safety compared to other languages available on more conventional computing platforms.

Some notable issues were the difficulty of using tools to improve security. For example, it was thought that using automatically generated code to produce the finite state machines for HTTP request/response parsing would reduce potential bugs by offloading the tedious state tracking to a tool, but it proved too difficult to debug, and consequently a less secure hand-written finite state machine implementation was used instead. Additionally, adding unit testing to the code was not performed mainly because much of the code relies on accessing specific hardware registers that would be difficult to emulate.

As the case study demonstrates, maintaining an understanding of vulnerabilities during systematically analyze the device during design can improve security noticeably. This is especially important during the hardware part of the design, as the choice of components can affect the overall security of the system substantially. Although security was not the main driving force behind the design process, it still remained in a sufficiently important role so that the most common vulnerabilities were circumvented.

There were also some clear phases of the design process where security could be made easier to implement to reduce the possibility of vulnerabilities. Software tools and frameworks on at least the ESP8266 still have not matured to where they will by default produce secure implementations of web servers, and the tooling still relies on primarily on developers' mindfully choosing to write their firmware in a secure way. There was also a lack of a clear choice for a low-cost wireless SoC with hardware security features to allow for secure boot and on-chip programming, although the current choices do include hardware encryption modules.

V. Conclusion

Vulnerabilities from a large sample of IoT devices were aggregated into database. The development of the database produced a taxonomy for threats to IoT devices, and this taxonomy was used to create a set of design rules that would avoid the major mistakes in device design. Additionally, the database provides a better understanding of the severity and importance of the different potential vulnerabilities, further supplementing the design rules with a more intuitive understanding of the severity of the vulnerabilities. This information was used to augment the design process for a robotic arm CPS testbed, and the information gathered from that experience was recorded in this paper. The IoT database will continue to grow, and as more information from existing large IoT/CPS systems is gathered, a new set of design rules to improve security at a system level will be developed from there. A similar case study will be performed to examine the benefits of system-level design rules after they have been developed.

References

[1] "Internet of things research study: 2015 report," Hewlett Packard Enterprise Development LP, Tech. Rep., 2015. [Online]. Available: http://www8.hp.com/h20195/V2/GetPDF.aspx/4AA5-4759ENW.pdf

[2] C. Miller and C. Valasek, "Remote exploitation of an unaltered passenger vehicle," *Black Hat USA*, 2015.

[3] N. Kobie, "Hacking the internet of things: from smart cars to toilets," *Alphr — A fresh take on technology*, Jul 22 2014. [Online]. Available: http://www.alphr.com/features/389920/hacking-the-internet-of-things-from-smart-cars-to-toilets

[4] mjg59. (2016, Mar.) I stayed in a hotel with android lightswitches and it was just as bad as you'd imagine. [Online]. Available: http://mjg59.dreamwidth.org/40505.html

[5] "Modbus messaging on tcp/ip implementation guide v1.0b," Modbus-IDA, Standard, Oct. 2006. [Online]. Available: http://www.modbus.org/docs/Modbus_Messaging_Implementation_Guide_V1_0b.pdf

[6] S. Babar, P. Mahalle, A. Stango, N. Prasad, and R. Prasad, *Recent Trends in Network Security and Applications: Third International Conference, CNSA 2010, Chennai, India, July 23-25, 2010. Proceedings*. Springer, 2010, ch. Proposed Security Model and Threat Taxonomy for the Internet of Things (IoT), pp. 420–429.

[7] S. Babar, A. Stango, N. Prasad, J. Sen, and R. Prasad, "Proposed embedded security framework for internet of things (iot)," in *Wireless Communication, Vehicular Technology, Information Theory and Aerospace Electronic Systems Technology (Wireless VITAE), 2011 2nd International Conference on*, 2011, pp. 1–5.

[8] O. Garcia-Morchon, S. Kumar, S. Keoh, R. Hummen, and R. Struik, "Security considerations in the ip-based internet of things," Working Draft, IETF Secretariat, Internet-Draft draft-garcia-core-security-06, September 2013, http://www.ietf.org/internet-drafts/draft-garcia-core-security-06.txt. [Online]. Available: http://www.ietf.org/internet-drafts/draft-garcia-core-security-06.txt

[9] N. Cam-Winget, A.-R. Sadeghi, and Y. Jin, "Can iot be secured: Emerging challenges in connecting the unconnected," in *IEEE/ACM Design Automation Conference (DAC'16)*, 2016, (to appear).

[10] G. Hernandez, O. Arias, D. Buentello, and Y. Jin, "Smart nest thermostat: A smart spy in your home," in *Black Hat USA*, 2014.

[11] O. Arias, J. Wurm, K. Hoang, and Y. Jin, "Privacy and security in internet of things and wearable devices," *IEEE Transactions on Multi-Scale Computing Systems*, vol. 1, no. 2, pp. 99–109, 2015.

[12] B. Parno, J. M. McCune, and A. Perrig, "Bootstrapping trust in commodity computers," in *Security and privacy (SP), 2010 IEEE symposium on*, 2010, pp. 414–429.

[13] A. Cui, J. Kataria, and S. Stolfo, "Killing the myth of cisco ios diversity: Recent advances in reliable shellcode design," in *USENIX Worshop on Offensive Technologies (WOOT)*, 2011.

[14] "Xbox 360 timing attack," 2007, [Online]. http://beta.ivc.no/wiki/index.php/Xbox_360_Timing_Attack.

[15] S. Skorobogatov, "Fault attacks on secure chips: from glitch to flash," in *Design and Security of Cryptographic Algorithms and Devices (ECRYPT II)*, 2011.

[16] S. Tuecke, V. Welch, D. Engert, L. Pearlman, and M. Thompson, "Internet x. 509 public key infrastructure (pki) proxy certificate profile," Tech. Rep., 2004.

[17] bushing, marcan, segher, and sven, "Console hacking 2010: Ps3 epic fail," in *27th Chaos Communication Congress*, 2010.

[18] R. Lemos, "Sony left passwords, code-signing keys virtually unprotected," *eWeek*, 2014, [Online]. http://www.eweek.com/security/sony-left-passwords-code-signing-keys-virtually-unprotected.html.

[19] B. Fowler, "Some top baby monitors lack basic security features, report finds," 2015, [Online]. http://www.nbcnewyork.com/news/local/Baby-Monitor-Security-Research-324169831.html.

[20] "Critical security flaw: glibc stack-based buffer overflow in getaddrinfo() (cve-2015-7547)," 2015, [Online]. https://access.redhat.com/articles/2161461.

[21] M. Smith, "Security holes in the 3 most popular smart home hubs and honeywell tuxedo touch," 2015, [Online]. http://www.networkworld.com/article/2952718/microsoft-subnet/security-holes-in-the-3-most-popular-smart-home-hubs-and-honeywell-tuxedo-touch.html.

[22] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks." in *Usenix Security*, vol. 98, 1998, pp. 63–78.

[23] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "Pointguard tm: protecting pointers from buffer overflow vulnerabilities," in *Proceedings of the 12th conference on USENIX Security Symposium*, vol. 12, 2003, pp. 91–104.

[24] H. Krawczyk, K. G. Paterson, and H. Wee, *Advances in Cryptology – CRYPTO 2013: 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*. Springer, 2013, ch. On the Security of the TLS Protocol: A Systematic Analysis, pp. 429–448.