Towards Securing Software of Embedded Linux Devices

by

Rijnard van Tonder

Thesis presented in partial fulfilment of the requirements for the degree of Master Engineering in Electronic Engineering in the Faculty of Engineering at Stellenbosch University

Department of Electronic Engineering,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.

Supervisor: Dr. H. A. Engelbrecht

December 2014

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: 2014/09/01

Abstract

Towards Securing Software of Embedded Linux Devices

R. van Tonder

Department of Electronic Engineering,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.

Thesis: MEng (Electronic)

September 2014

As Embedded devices continue to proliferate, there is a rising concern surrounding the security that these increasingly complex and capable devices provide. Software development processes are successfully employed to address security in desktop operating systems and applications, yet there is no widely accepted security process for embedded systems. In this thesis, we demonstrate how security of embedded Linux devices may be improved by considering 12 well-chosen case studies that exemplify methods advocated by established secure software development processes. Specifically, we derive high-level methods from a comparative study of two well-known security processes: The Microsoft Security Development Lifecycle (SDL) and the OWASP Comprehensive Lightweight Application Security Process (CLASP), and use these to evaluate embedded Linux devices. These methods, namely, attack surface analysis, threat modeling, and security testing, drive the assessment techniques that enable vulnerability discovery and analysis covered in our case studies. We apply and investigate these techniques in terms of attacks that pertain to three common elements of a typical embedded Linux device, that is, operating system, network, and Universal Serial Bus (USB) attacks. During assessment, a number of new security vulnerabilities are discovered in these attack surfaces, demonstrating the effectiveness of our approach. Moreover, we develop a novel, publicly available USB fuzz testing framework for discovering USB vulnerabilities. Our final contribution culminates in six concrete, actionable recommendations based on our case studies for improving embedded security. Interestingly, our recommendations correlate with those advocated by security expert Gary McGraw, but with the added benefit of being substantiated by concrete case study analyses in the embedded space.

Uittreksel

R. van Tonder
Departement Elektroniese Ingenieurswese,
Universiteit van Stellenbosch,
Privaatsak X1, Matieland 7602, Suid Afrika.

Tesis: M.Eng (Electroniese)
September 2014

Soos toegewyde toestelle voortgaan om te vermenigvuldig, is daar 'n toenemende kommer rondom die sekuriteit wat hulle bied. Al word sagteware-ontwikkeling prosesse suksesvol toegepas op gewone rekenaars en programme, bestaan daar nie 'n aanvaarde sekuriteitsproses vir toegewyde stelsels nie. In hierdie tesis wys ons hoe die sekuriteits aspekte van toegewyde Linux stelsels verbeter kan word deur middel van 12 gevallestudies, waarin ons gevestigde sagteware-ontwikkeling proses metodes toepas. Ons begin deur twee bekende sekuriteit prosesse te vergelyk: die Microsoft Security Development Lifecycle (SDL) en die OWASP Comprehensive Lightweight Application Security Process (CLASP). Hiermee kies ons metodes wat van toepassing is om die sekuriteit van toegewyde Linux toestelle te evalueer. Die metodes, naamlik aanval oppervlak analise, bedreigingsmodellering, en toegepaste veiligheids-evalueering word gebruik om sekuriteits foute te ontdek en te analiseer in ons gevallestudies. Verder neem ons drie elemente in ag van toegewyde Linux toestalle wat tipies aangeval word, naamlik, die bedryfstelsel, netwerk, en USB oppervlaktes. Gedurende assessering is 'n aantal nuwe sekuriteit probleme ontdek in hierdie aanval oppervlaktes, wat die doeltreffendheid toon van ons benadering. Verder ontwikkel ons 'n nuwe USB toetsraamwerk om sekuriteits foute te ontdek, wat boonop aan die publiek beskikbaar gemaak is. Ons finale bydrae is ses konkrete aanbevelings vir die verbetering van sekuriteit in toegewyde stelsels, wat ontwikkel is op grond van ons gevallestudies. Interessant genoeg, ons aanbevelings stem ooreen met dié bepleit deur sekuriteit deskundige Gary McGraw, maar met die addisionele voordeel dat dit gebaseer is op konkrete gevallestudies in die veld van toegewyde stelsels.

Acknowledgements

My thanks go to Naspers, the MIH Media Lab, and my supervisor Dr. Engelbrecht for making their support and resources available to me during this work. I also thank my family for their support, encouragement, and love during this time.

I thank my God, who has taught me to trust in Him with all my heart, and not to lean on my own understanding, to acknowledge Him in all my ways, that He may direct my paths. *Prov. 3:5-6.*

Contents

List of Figures

List of Tables

Glossary

ASLR Address Space Layout Randomization is a defensive computer security technique which attempts to thwart exploitation of vulnerabilities by using non-deterministic addresses for the memory contents of a process.

Black-hat An entity with expertise in computer security who typically engages in illegal hacking activities with malicious intent.

CIA Triad Confidentiality, Integrity, and Availability form a triad of three core security goals that are widely accepted in the field of Information Security.

CLASP The Comprehensive Lightweight Application Security Process is a project maintained by OWASP that incorporate security concerns into the software development lifecycle.

CVE Common Vulnerabilities and Exposures is a collection of identifiers that reference publicly known security vulnerabilities.

NIST The National Institute of Standards and Technology is a federal agency that promotes standards and measurements in a number of fields, including computer security.

OWASP The Open Web Application Security Project is a not-for-profit organization that focuses on improving the security of software.

SDL Microsoft's Security Development Lifecycle is a software development process that specifically addresses security concerns of products.

White-hat An entity with expertise in computer security, sometimes referred to as an ethical hacker, that performs security evaluation of products within a legal framework.

Chapter 1

Introduction

1.1 Introduction

Overview

Recent years have seen a dramatic increase in the number of embedded devices and consumer electronics that individuals interact with on a daily basis. With a projected number of 15 billion connected devices by 2015 [18], embedded devices are set to increase in ubiquity for the foreseeable future.

This growth is met with an increased concern for ensuring security on embedded devices. Moreover, as embedded devices become more powerful, they are able to support functionality and software of increasing complexity, thereby demanding greater effort towards security practices. More powerful embedded devices have also resulted in the uptake of running complete operating systems which support common libraries, hardware, and services by default. At the forefront is Linux, an open-source solution which many embedded device manufacturers adopt. Due to its extensive support and open-source nature, Linux is the most common operating system of new embedded devices [51], and runs on a diverse set of devices, including mobile phones, tablets, routers, switches, smart TVs, set-top boxes, consoles, and many more.

Securing Embedded Devices

In broad terms, this thesis aims to address the need to secure software of such devices through actionable recommendations. By assessing embedded Linux devices, we derive the benefit of covering a broad range of devices, while also providing a concrete platform on which we can base our security recommendations. Specifically, we are interested in securing various properties of embedded Linux devices from vulnerabilities, and to find methods that are effective at doing so. The need for recommendations toward improving embedded security is evident, especially in the light of the numerous vulnerabilities disclosed on a daily basis.

Security Properties In the interest of securing embedded devices, we must first ask what the core security goals of an embedded device is. In this thesis, we are interested foremostly in assessing a device's security in terms of an *attacker-centric* threat. The notion of an adversary implies that the device's security properties can be undermined specifically in terms of its

- Confidentiality,
- Integrity, and
- Availability.

These properties are known as the CIA triad [33], and refer collectively to the security of information (confidentiality), accuracy and validity of data (integrity), and resource availability. Throughout a system's lifetime, these properties are to be preserved. The violation of one or more of these properties constitute a *threat* to the core security goals of a given device; a successful attack would invariably result in such a violation. Note, however, that this principle is applicable to all systems, and not only to embedded devices.

Approach In order to address the security threats[1] particular to *embedded* devices, we must consider where they are attacked, and how. In answering the former, we consult accepted methods of software development processes to adopt *attack surface* analysis. The attack surface represents the potential vectors through which an attack can be launched against a device. This typically implies that an attacker can specify input to a device's software which is subsequently processed. In terms of embedded software, we identify in this thesis that all embedded Linux devices have an attack surface that includes one or more of the following:

- Operating System
- Network services
- USB functionality

In addressing the latter, we employ security testing techniques, vulnerability classification, and vulnerability analysis in a series of 12 case studies, four per attack surface. By performing vulnerability analysis, we answer how an attacker might proceed to discover vulnerabilities, compromise an embedded device, and how the threat of doing so differs from traditional computers. We use the term "case study" to refer to the act of demonstrating particular instances where security assessment techniques are applied and the subsequent results are analyzed, illustrating the need for addressing the concerns presented

[1]That which we have now established to be a violation of one of the CIA properties.

by embedded devices.[2] Whereas the term "use case" might seem suitable, we do not feel that this term encompasses the analysis component of the examples provided in this thesis.

In summary, we demonstrate how security of embedded Linux Devices may be improved by considering 12 well-chosen case studies that exemplify methods advocated by established secure software development processes. Specifically, we derive high-level methods from a comparative study of two well-known security processes: The Microsoft Security Development Lifecycle (SDL) [39] and the OWASP[3] Comprehensive Lightweight Application Security Process (CLASP) [63], and use these to evaluate the devices. These methods, namely, attack surface analysis, threat modeling, and security testing, drive the assessment techniques that enable vulnerability discovery and analysis covered in our case studies. We apply and investigate these techniques in terms of attacks that pertain to three common properties of a typical Embedded Linux device, that is, operating system, network, and USB attacks. Particular emphasis is placed on the USB attack vector, where an increased number of vulnerabilities have been discovered due to the recent advance of hardware-based USB emulation techniques.

The case studies of this thesis include vulnerabilities that have been disclosed in the past, as well as newly discovered vulnerabilities as a result of this research. The devices of these case studies include mobile phones, tablets, switches, routers, smart TVs, and set-top boxes. While these do not present an exhaustive list of potential embedded attacks, the case studies are representative of the unanticipated challenges that embedded devices face. They convey the importance of incorporating security methods into embedded software development, and deliver insight into a number of common pitfalls. Based on the insights from our findings, we recommend secure practices that assist in preventing many of the attack classes presented throughout this thesis.

Objectives

The objectives of this work is to:

- Derive appropriate techniques from secure software development processes to evaluate embedded Linux devices
- Investigate and apply vulnerability assessment techniques over system, network, and USB attack surfaces of embedded Linux devices

[2]Although somewhat related, case study should not be taken to mean consideration of a particular person, group, or situation over a period of time, as it is commonly understood in the field of social sciences.

[3]The Open Web Application Security Project

- Classify vulnerability discoveries and demonstrate the risk posed to Embedded devices
- Recommend secure practices for software design in embedded Linux devices based on findings

Contributions

This thesis presents:

- A case study for applying secure software development processes (Microsoft SDL, OWASP CLASP) to embedded Linux devices
- Exposure and disclosure of new vulnerabilities on embedded devices
- The Transparent Two-Way Emulation framework, a novel testing framework for the USB attack vector
- Concrete recommendations based on case studies for improving device security during software development

Structure

The structure of this thesis is as follows. In Chapter 2 we cover research relevant to secure software processes and embedded device security. These software processes assist in providing secure development methods without being platform specific. Thereafter follows a discussion of advances in the domain of embedded device security, including methods for finding vulnerabilities, executing attacks, and mitigating these attacks. We also consider the contribution that these advances have made toward recommending secure practices for embedded devices. In Chapter 3 we break up the secure software processes into their constituent parts, from which we derive appropriate techniques that can be used to evaluate security attributes of embedded devices in terms of system, network, and USB functionality. We also define the scope of assessment, and assessment techniques.

Chapter 4 gives a technical overview of system, network, and USB services that are typically found on embedded devices. Here, we provide an understanding of an embedded device's design and software implementation from a developer's perspective. These elements affect the security of the final device, and the points covered in this chapter serve as an additional reference to the subsequent case studies. The case study chapters, namely, Chapter 5, Chapter 6, and Chapter 7 contain the application of security assessment over a variety of devices, so as to demonstrate attack vectors, vulnerability classification, and associated threats. To this end, we show how the vulnerabilities could have been averted; these insights provide the basis for recommendations which are delivered in Chapter 8.

Chapter 2

Literature Survey

2.1 Overview

This literature survey covers existing work in embedded security as it pertains to the objectives of §1.1. We consider research on secure software development processes, embedded software security, and existing recommendations found in literature.

2.2 Secure Software Development Processes

As the need for ensuring privacy and security rises, there is an increasing need for appropriate secure software development processes. Gregoire et al. perform a comparative study [35] between Microsoft's Secure Development Lifecycle (SDL) [39] and OWASP CLASP [63]. A follow-up study [31] discusses high-level similarities and differences in more depth. However, these studies do not demonstrate the difference in how the secure development processes are applied, but rather give an indication of the value and potential shortcomings of each approach. In this thesis, we identify methods in these processes that are relevant to the secure development of embedded devices. We apply these methods in practical case studies, delivering clear guidelines on how the methods can be applied in practice. Howard, a key contributor to the SDL, supports the notion that "numerous examples of correct and incorrect design and coding behavior" provides the necessary insight for developers to "design and build more secure software" [38]. Enck et al. acknowledge the role of security processes including SDL and CLASP for defining security requirements in the context of mobile phones [34]. Ukil et al. addresses a broader range of devices found in the "Internet of Things" [59] , but like Enck et al., emphasis is placed on evaluating tools that address platform or hardware-specific requirements. Hence, while valuable, these studies do not comprehensively address the application of secure practices on a broad range of devices.

Practical application of security assessments are covered in many studies and books. Well-known resources include works by Dowd [33], McGraw [64], NIST [53], and Howard [44]. These works establish specific and thorough methods for performing software security assessments, and do not prescribe a structured, high-level process such as the SDL. Furthermore, they are not device and application specific, but consider software implementation concerns in operating systems, programs, and the environments in which they execute. While these resources provide the technical depth for evaluating a system's security, they do not address some of the more subtle considerations that should be taken into account specifically for embedded devices.

2.3 Embedded Software Security

The rising ubiquity of embedded devices, and security concerns associated with it, are well addressed by Koopman as early as 2004 [43]. There is a recognition that embedded security is different from conventional computers; tight development budgets cause developers to overlook or ignore security factors. In a case study of internet-connected thermostats, Koopman identifies the danger of allowing these devices to be accessible over a network—behind this danger is the idea that the attack surface for embedded devices have increased. Koopman envisions a variety of attacks, and suggests that those carried out on the thermostat could be very subtle due to its embedded nature. Schneier echoes Koopman's questions on our ability to meet the needs of embedded security, stating that "embedded computers are riddled with vulnerabilities, and there's no good way to patch them" [54]. These sentiments ring true, but are delivered on the basis of thoughtful speculation and personal knowledge. This thesis substantiates these sentiments with numerous thorough practical case studies.

In 2013, Cui et al. demonstrated the impact of vulnerabilities on printers, which were discovered by analyzing device firmware [26]. It was shown that vulnerabilities in the implementation of `zlib` and `openSSL` could allow an attacker full control over the device. A further example includes Cisco Internet Protocol (IP) phone devices, which are vulnerable to compromise through an exploit of the operating system kernel [25]. Cui advocates hardening techniques where "unused code. . . is autotomically removed in order to reduce the potential vulnerable attack surface of the overall system". Cui admits that this is helpful, but does not stop attacks that may succeed against "necessary features that cannot be removed". To overcome this, Cui advocates the use of intrusion detection code that is resident on the embedded device during its operation [27]. While this solution may prove effective, we advocate that the discovery and analysis of embedded device vulnerabilities from a secure process perspective also alleviates threats.

Yang et al. performed a case study of concurrency attacks on various platforms, including Linux and Cisco Internetwork Operating System (IOS), an embedded operating system based on Linux [67]. Although the study is specific to concurrency attacks, including race conditions and time-to-check-to-time-of-use attacks, its empirical approach is similar to the one advocated by this thesis. That is, vulnerabilities are classified, attacks are demonstrated, and the impact is analyzed. At a high-level, the paper concerns vulnerabilities that compromise memory safety, not unlike many of the vulnerabilities considered in our case studies. We address a broader range of attacks as well as devices in the interest of embedded security.

Ravi et al. have identified hardware and software attacks that make embedded systems design challenging [52]. Software attacks are classified as "logical attacks", examples of which include "buffer overflow exploits" and "subverted device code updates". These attacks are very broad, and we elaborate on their specific properties in §3.5.2. Multivarious solutions have been constructed to protect embedded devices from these attacks, including memory protection through software attestation [55], SELinux access control [56], and secure bootstrapping that guards against firmware modification [20]. Application isolation and sandboxing techniques further promote the security of embedded devices. Ravi et al. briefly consider one such solution, namely, the ARM TrustZone [19] technology that "provides an architecture-level security solution" and separates device code into "trusted" and "untrusted" portions, such that untrusted code cannot indiscriminately modify system memory and resources.

Kocher et al. describe tamper-resistant software solutions, and point out the threat of a "hardware virus" that can enter via kernel attack vectors in software and persist throughout a embedded device's lifetime in hardware protected memory [42]. Although not based on an OS kernel, it was demonstrated in 2013 that firmware residing on harddisk controller chips could be reprogrammed by an attacker, persisting in flash even when the harddisk is reformatted [32]. This demonstration proved to be an insightful exercise, but does not address further classes of attacks.

Bratus et al. state that "securing a system requires preventing attackers from exploiting... inevitable vulnerabilities...," referring particularly to embedded systems which support USB functionality [23]. Although USB security has been evaluated in terms of virus propagation [24] [57], it's potential impact and threat to embedded devices at a driver and system level is not yet well understood. Aiding in this pursuit is the recent development of USB testing tools that enable bug finding in USB-related software [1]. However, USB testing tools for security purposes lack in functionality and prove expensive [28]. We present a solution for testing USB software that improves on existing solutions. Furthermore, we infer security concerns and recommendations from recent research in the area of USB security, and address them with four case studies in the embedded space.

2.4 Recommendations

Given the need for secure practices during embedded design and implementation, current literature delivers a number of recommendations. An important recommendation is the automatic removal of superfluous code and disabling of unnecessary features [26]. Yang et al. recommend runtime software checks, setting non-executable memory in hardware, and enabling address space randomization for countering concurrency attacks [67]. Ravi et al. [52] consider the following as valid countermeasures for software attacks on embedded devices:

- Preserving privacy and integrity in the execution environment
- Validating that a given program is safe to execute
- Identifying and removing of software bugs that make the device vulnerable

Ravi et al. give a number of examples in which these countermeasures may be achieved. The first is the inclusion of dedicated hardware and software which enforce access control to system resources. This may include secure bootstrapping when the device is initially started, or hardware to protect sensitive memory locations. Code can be considered "safe" by performing validation checks (such as code-signing), and providing sandbox execution environments. "Finding security flaws in trusted software. . . and their implementations. . . " are deemed important, but not elaborated upon.

2.5 Conclusion

Although there are many software *development processes* that aid in the design, verification, and implementation of systems, they do not define concrete analysis techniques, nor are their methods particular to a specific platform. They are primarily valuable in guiding the sequence of high-level testing methods, and in this way guard against obvious blind spots during security assessment. Concrete and reliable techniques for security assessment are readily available in a number of resources, which ought to be consulted during the various phases of a development process. We mentioned that these techniques are not platform-specific, but tailored to the system being tested. We assert therefore that methods and testing techniques need to be tailored to embedded device security assessment; there is no predefined, infallible process specific to embedded devices for security testing.

Research on various aspects of *embedded security* were discussed. There is a clear indication that embedded security present new challenges, and that defensive techniques have not yet matured to the extent of traditional computers. Some research identify problems that affect both embedded devices and traditional computers, such as concurrency attacks. Other research draw upon a single case study or platform in order to justify a defensive technique, such as put forth by Cui. Yet further research, such as the design challenges identified by Ravi, describe numerous valid techniques; unfortunately these techniques are not accompanied by thorough case studies. This thesis differentiates itself from these works by considering multiple case studies, taking into consideration a wide range of attacks, and referencing these with appropriate testing techniques and design processes.

Recent research dedicated to USB security present novel approaches to testing, and warrant particular attention. Bratus et al. refer to the potential threat that USB functionality poses to embedded devices; in this thesis we present scenario where an attack against the USB vector can have unanticipated consequences for embedded devices.

The recommendations covered in this literature survey are valid for embedded devices: memory protection and removal of unused code are sensible objectives for improving device security. Given the challenging task of delivering appropriate recommendations for the wide number of embedded devices, it is understandable that advice is somewhat general. The research in this thesis, being limited to the Linux operating system, enables us to deliver precise recommendations for improving embedded device security. Moreover, these recommendations are based on methods and techniques for software assessment, and not, for example, dependent on a specific tool or runtime solution.

Chapter 3

Security Assessment Methods

3.1 Overview

This chapter establishes the baseline methods that underlie the investigation of Embedded Linux security throughout this thesis. These are used foremostly to identify the scope of security concerns that are applicable to embedded devices, and to direct the manner in which vulnerability discovery and analysis is performed. In doing so, it is also demonstrated how these processes translate into *practical* steps for Embedded device development.

Software security requirements are often vast, and security processes have been developed in order to compartmentalise these requirements. The methods chosen for assessing embedded security are taken from elements shared by two high-profile security processes: The Microsoft Security Development Lifecycle (SDL) [39] and the OWASP Comprehensive Lightweight Application Security Process (CLASP) [63]. These processes are chosen on the basis of being mature and well-understood in industry. The methods dictate the *assessment scope* such that it maintains relevancy to embedded devices. The *application* of these methods for vulnerability analysis and classification are largely driven by practices in definitive literature of security assessment: Dowd's The Art of Software Security Assessment [33] and NIST's Technical Guide to Information Security Testing and Assessment [53].

3.2 Secure Software Development Processes

In this section an overview is given of the SDL and CLASP processes. We derive methods from common elements identified, and limit these to where the design and implementation of software is concerned. We assert that *attack surface analysis*, *threat modeling*, and *security testing* affords the primary methods by which to conduct software security assessment of embedded devices.

3.2.1 Microsoft SDL

The Microsoft Security Development Lifecycle consists of 17 Practices contained in 7 phases as depicted in Figure 3.1. Highlighted are the three phases that directly influence the security of code which runs on the device: design, implementation, and verification. Practice 15 in the release phase addresses a final security review which could arguably be included, however, it makes use of the same methods as the three phases mentioned. Next we consider the practices of particular interest for the three phases.

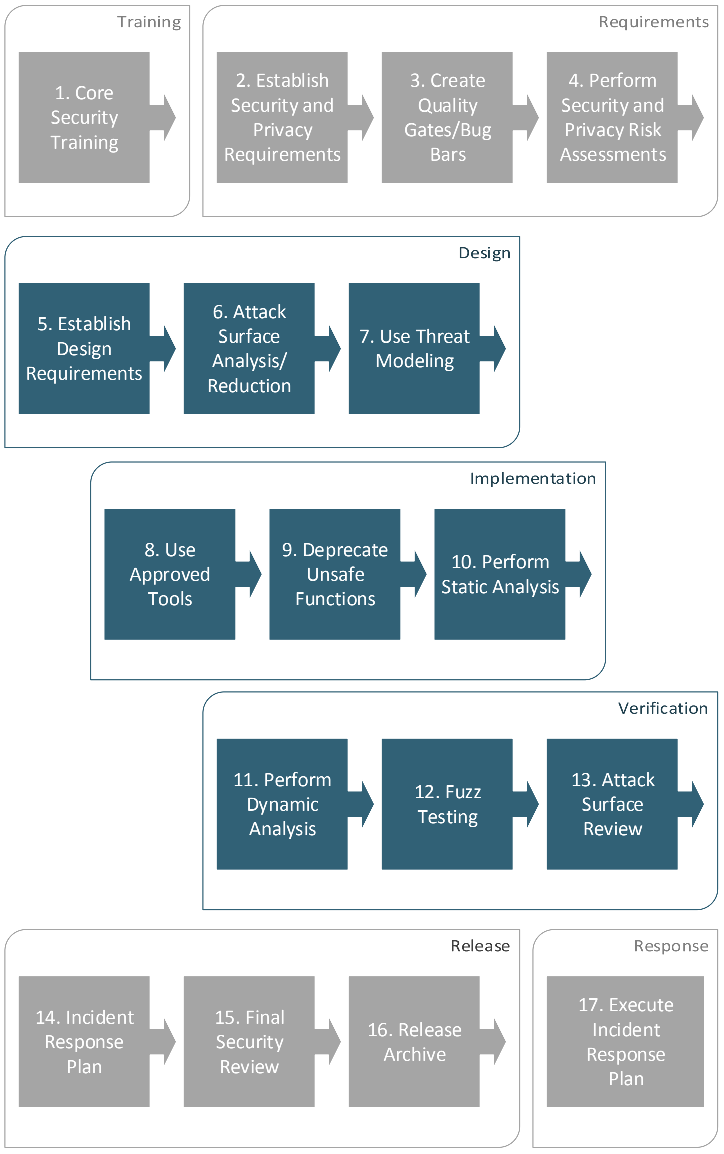
Training
1. Core Security Training
Requirements
2. Establish Security and Privacy Requirements
3. Create Quality Gates/Bug Bars
4. Perform Security and Privacy Risk Assessments
Design
5. Establish Design Requirements
6. Attack Surface Analysis/ Reduction
7. Use Threat Modeling
Implementation
8. Use Approved Tools
9. Deprecate Unsafe Functions
10. Perform Static Analysis
Verification
11. Perform Dynamic Analysis
12. Fuzz Testing
13. Attack Surface Review
Release
14. Incident Response Plan
15. Final Security Review
16. Release Archive
Response
17. Execute Incident Response Plan

Figure 3.1: Microsoft Software Development Lifecycle

6 & 13. Perform Attack Surface Analysis, Reduction, and Review Attack Surface Analysis is the process of determining the potential vectors through which an attack can be launched against a device's software. For example, an attack vector may exist in system software, network services, or applications, and implies that the attacker is able to send input which is handled by this software. The final Attack Surface Review determines the state in which the device software will be released, possibly into the hands of an attacker. Mapping the potential avenues of attack with the attack surface analysis method establishes where to perform vulnerability anlaysis.

Attack Surface Reduction (or Mitigation) is the process of reducing potential attack vectors, and can also be described as *software hardening*. Software hardening can be performed by reducing the amount of running code and available entry points, restricting access to system services, or eliminating unnecessary services. The act of reducing the attack surface prevents possible vulnerabilities from existing on a system, but does not lessen the effects of vulnerabilities that remain present after software hardening.

7. Threat Modeling Threat modeling within the SDL is described as a "structured approach to threat scenarios during design" that helps "identify security vulnerabilities, determine risks from those threats, and establish appropriate mitigations. [39]" Threat Modeling commonly involves identifying assets that reside on a device that should be protected, and can provide means of prioritizing where focus is placed for finding vulnerabilities.

For example, a mobile phone may contain a user's contact list—an asset that should be protected. Consider two scenarios that unauthorized access to the user's contacts could be obtained: one which requires physical access to the phone, and another which simply requires the phone to have an internet connection. While both scenarios should be addressed, the latter scenario presents a bigger threat. Hence, more resources should be devoted to ensure that this threat is mitigated. In this manner, threat modeling assists in prioritizing where software security is ensured.

8-12. Software Security Testing Software security is ensured in the SDL with security-checking tools (8), removing unsafe functions (9), performing static and dynamic analysis (10 & 11), and fuzz testing (12). More broadly, these methods all serve to prevent vulnerabilities from existing in the software after implementation. Consider that an attacker would employ the same methods when performing *vulnerability discovery and analysis*: the use of security tools, searching for unsafe functions, performing static, dynamic, and fuzz testing where possible.

3.2.2 OWASP CLASP

The CLASP process consists of five-high level methods contained in "Views", as shown in Figure 3.2. CLASP places emphasis on activites that are performed based on participant roles. Roles such as project managers, developers, or security auditors participate in activities where software security requirements are assessed and software is implemented. The Vulnerability View governs these activities and provides methods for identifying, classifying, and remediating software flaws.

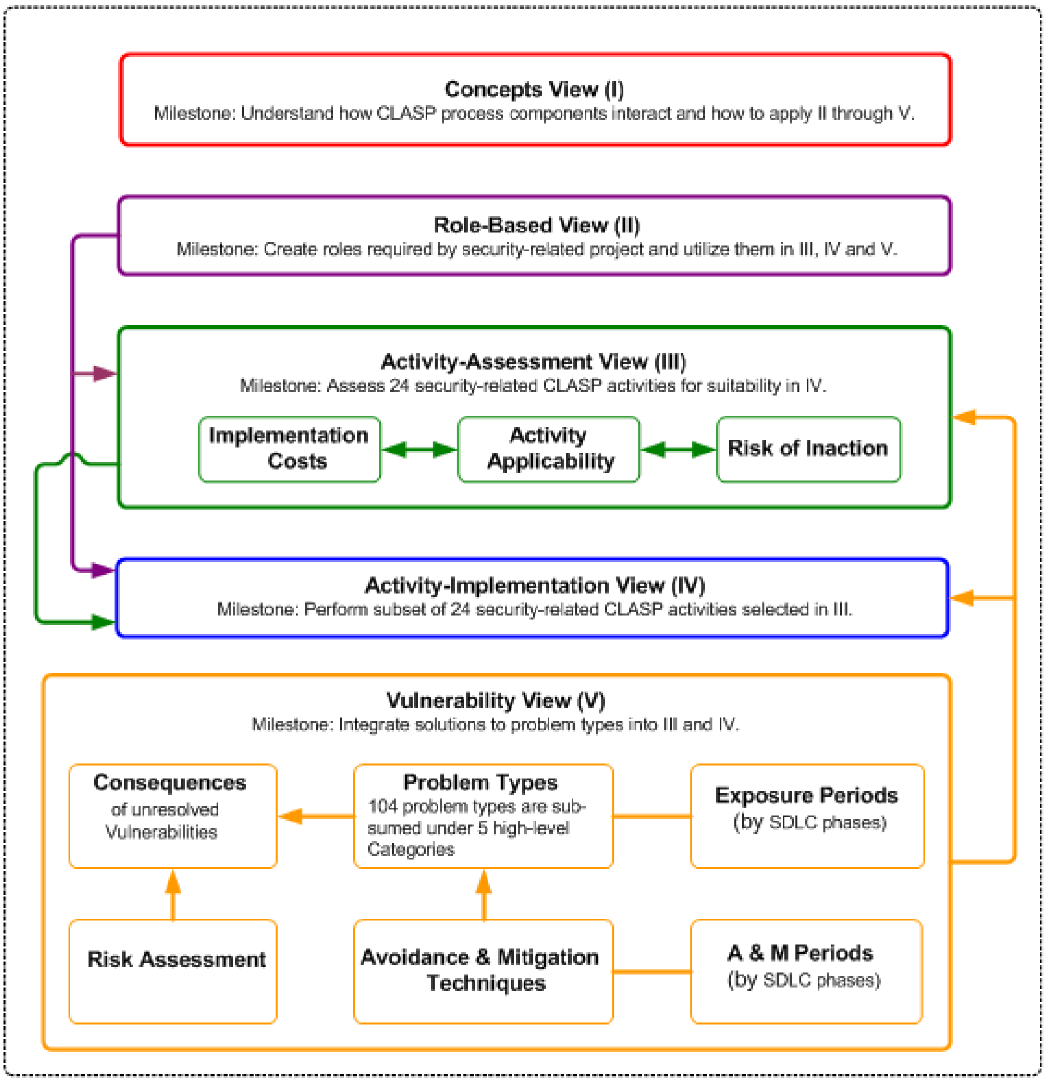
Concepts View (I)
Milestone: Understand how CLASP process components interact and how to apply II through V.
Role-Based View (II)
Milestone: Create roles required by security-related project and utilize them in III, IV and V.
Activity-Assessment View (III)
Milestone: Assess 24 security-related CLASP activities for suitability in IV.
Implementation Costs
Activity Applicability
Risk of Inaction
Activity-Implementation View (IV)
Milestone: Perform subset of 24 security-related CLASP activities selected in III.
Vulnerability View (V)
Milestone: Integrate solutions to problem types into III and IV.
Consequences of unresolved Vulnerabilities
Problem Types 104 problem types are subsumed under 5 high-level Categories
Exposure Periods (by SDLC phases)
Risk Assessment
Avoidance & Mitigation Techniques
A & M Periods (by SDLC phases)

Figure 3.2: OWASP CLASP Views (https://www.owasp.org/index.php/CLASP_Concepts)

CLASP Activities CLASP defines 24 security-related activities that may be undertaken during a software development process. Some activities address specific needs that may not be applicable in all cases, for instance, the activity to specify database security configuration. Activities that should be integrated into the security process are determined during the Activity-Assessment View, and each activity has an associated role that performs the activity. The following list documents key activities that correlate with Microsoft's SDL practices. These activities distinguish themselves in that the associated role is that of the security-auditor.

- Identify attack surface
- Perform security analysis of system requirements and design (threat modeling)
- Perform source-level security review
- Integrate security analysis into source management

The Activity-Implementation View provides specific guidelines on how the activities are performed. Attack surface analysis involves identifying all system entry points, and mapping resources that are accessible from entry points. Threat modeling involves identifying threats on assets and capabilities, and determining the level of risk poised by threats. In the specification [63], identifying threats is accompanied by the question "If I were an attacker, how could I possibly try to exploit this security service?" The source-level security review is applicable when software source code is available. The purpose of this activity is to find security vulnerabilities introduced into the implementation by manual code review, automated analysis tools, or both. Furthermore, this activity specifically prescribes identifying vulnerabilities as set forth in the Vulnerability View. Integration security analysis is an appropriate activity for performing automated dynamic or static analysis on a code base, and may happen in conjunction with the source-level security review.

Vulnerability View The Vulnerability View provides a taxonomy for guiding CLASP activities. The Vulnerability View places strong emphasis on the "Vulnerabilty Lexicon" which classifies vulnerabilities according to 5 categories of problem types. These problem types are:

1. Range and Type Errors
2. Environmental Problems
3. Synchronization & Timing Errors
4. Protocol Errors

5. General Logic Errors

The categories identify the types of problems that should be detected during security activities, for example: unsafe functions that lead to memory corruption, system environment settings that may lead to a denial-of-service (DoS), or time-of-check-to-time-of-use (TOCTTOU) race conditions. All of these problem types imply one or more violations of the core security goals for system software:

- Confidentiality
- Integrity
- Availability
- Authenticity
- Non-repudiation

From an attacker's standpoint, the most commonly violated goals are the Confidentiality, Integrity, and Availability properties of a system. This is commonly known as the CIA triad [33], which is covered further in §3.4.

CLASP recognizes the importance of identifying and classifying vulnerabilities in a software environment, where a vulnerability "allows an attacker to assume privileges within a user's system, utilize and regulate its operation, compromise the data it contains, and/or assume trust not granted to the attacker." This definition also enforces the notion that vulnerability analysis is an attacker-oriented activity.

3.3 SDL and CLASP Compared

It is evident from §3.2.1 and §3.2.2 that the two software processes share common methods. This section documents the common methods, which are used in §3.4 for inclusion into our embedded device assessment.

3.3.1 Attack surface analysis

Both processes recognize the importance of attack surface analysis—in CLASP it is a dedicated activity, and in the SDL it comprises phase 6. A comparative study of CLASP and the SDL by Gregoire et al. [35] identifies this intersection as well, and points out that this method is "aimed at preventing an attacker from taking advantage of potentially insecure code".

3.3.2 Threat modeling

Threat modeling is a common method to both processes—in CLASP it is a dedicated activity, and in the SDL it comprises phase 7. Gregoire et al. [35] also discusses the intersection of threat modeling for both processes, and notes that both methods of threat modeling focus on "technical weaknesses rather than business-level threats"; an appropriate designation for secure software processes.

3.3.3 Security testing

Security testing of software is critical to both processes. The SDL specifically identifies removal of unsafe functions with code scanning tools (Phase 8), static analysis (Phase 9), and manual code review. The SDL states that "analysis tools can do much of the work of finding and flagging vulnerabilities", but that they are not perfect—manual code review should be used for critical components of an application. During the integration of security analysis into source management in CLASP, both static and dynamic analysis tools are used. Manual code review is applied during source-level security review. Often, static analysis and manual code review can be viewed as complementary methods. For instance, a static code analyzer may be used to find unsafe functions, which may further prompt manual code inspection.

The SDL gives more depth to the role of dynamic testing (Phase 11), for example, detection of memory corruption, user privilege issues, and handling malformed data. Fuzz testing (Phase 12) incorporates this last example, and is a special case of dynamic testing—malformed data is intentionally sent as input to the program in order to induce a crash. Though CLASP does not specifically discuss fuzz testing, it references the importance of performing input validation on malformed data. De Win et al. [31] point out that both processes stress the importance of security testing, with the SDL taking a "predominant black-hat approach", whereas CLASP is more white-hat oriented. Refer to table 3.1 for a summary of the above discussion.

Security Test	SDL	CLASP
Static Analysis	Phase 9 & 10	Integration of security analysis into source management
Manual Code Review	Phase 10 (optional)	Source level security review
Dynamic Analysis	Phase 11	Integration of security analysis into source management
Fuzzing	Phase 12	Referenced in “Basic Principles in Application Security"

Table 3.1: SDL/CLASP Comparison Table

3.4 Assessment Scope

The comparison of SDL and CLASP in §3.3 demonstrates the importance of three methods common to both processes that aid in discovery of software flaws and vulnerabilities:

- Attack surface analysis
- Threat modeling
- Security testing

These three methods are chosen to underlay the application of security assessment in the context of embedded Linux systems within this thesis. However, embedded devices vary in hardware, functionality, and software, thus requiring further effort in determining the scope of assessment. For example, the *attack surface* of an entire device, such as a smartphone, may include attack vectors over Wi-Fi, cellular networks, or USB connections, whereas a printer may include attack vectors only over an ethernet connection. All embedded Linux devices, however, have an attack surface that include attack vectors in one or more of the following:

- Operating System
- Network services
- USB functionality

In this thesis, the scope of assessment and associated case studies are applicable to these three categories.

As with the attack surface method, it is similarly challenging to perform *threat modeling* for varying embedded devices. For instance, it may be desirable for a privileged user to have access to a router device so that they can alter configuration according to their needs. On the other hand, any type of access to the internal operation of a set-top box may not be desired by the manfucaturer, and could constitute a threat.

Rather than attempting to model all the threats for a number of different devices, we seek to identify the capabilities of an attacker that would constitute a threat for a given device. In this manner, we're able to use an attacker-centric approach to threat-modeling in order to identify whether the discovery of a security vulnerability is of legitimate concern.

In this thesis, we consider a security vulnerability to be of legitimate concern, that is, a threat to a device, if it violates one of the following security properties:

- Confidentiality
- Integrity
- Availability

These properties were alluded to in §3.2.2 and are commonly known as the CIA triad. Dowd et al. [33] specifically describes these properties as those which help "to determine which issues they[1] consider to be security violations". Here is a description of each property:

Confidentiality is the property that ensures the privacy of information on a device. For example, personal files on a mobile device containing data such as accounts and passwords should be kept secret by means of access control or encryption. Confidentiality may also be required for an entire device, for example, it should not be possible to access the internals of a set-top box from external ports. In this case, any disclosure of the software running on the device constitutes a breach in the confidentiality property.

Integrity concerns the accuracy and validity of data. If data can be modified, altered, or tampered with in an unanticipated way, the integrity of the data is at stake. Data modification may occur while it travels over a network, or while it resides on a device. The ability of an attacker to modify data poses threats of varying degrees, and in the worst case could mean an attacker having full control over a device.

[1]The security auditor or assessor.

Availability specifies that the resources and services of a device should be operational when needed. For example, when a Denial-of-Service (DoS) attack succeeds against a device or service, it will be unable to service legitimate requests.

3.5 Application of Embedded Software Security Assessment

In §3.4 we established that the type of software assessed shall pertain to system, network, and USB functionality of devices. We further established that we will identify threats to the security of a device by assessing whether a given vulnerability violates one of the CIA principles. With this approach, the influence of the attack surface analysis and threat modeling methods are evident. Another contribution of these two methods is their property of being *attacker-centric* in their application–the approach which this thesis advocates.

Attack surface analysis is a natural exercise for an attacker, who prioritizes which aspects of a device or piece of software to attack. Attacking the operating system is of interest to the attacker, who may seek elevated privileges, control over the device, and access to restricted resources. Likewise, network services open up the ability for an attacker to exploit vulnerabilities remotely. Because networking may make a device remotely accessible, it is often prioritized by an attacker. Network attacks may vary greatly, due to the characteristic of being layered by various software; in fact, the network stack itself presents an entire attack surface on its own account. This thesis does not attempt to be exhaustive in this respect, but provides insight into a variety of attack vectors at the network level in Chapter 6. The USB attack vector is distinguished from the other attack vectors considered. USB functionality is often supported by drivers in the operating system, and could be considered part of the operating system. However, the USB protocol mandates a separate hardware port, and drivers are often supplied by third-parties. Moreover, while embedded devices commonly support additional hardware capabilities (e.g. wireless networking), this may not always be the case. In contrast, USB support on embedded devices is ubiquitous. These unique hardware and software properties imply further security considerations, and provide enough incentive for separate consideration.

When considering an attacker-centric threat modeling approach, we, like the attacker, are interested in the properties that could be exploited as described in §3.2.2, such as "Range and Type Errors". This thinking translates into the application of vulnerability assessment techniques, and we discuss it's impact in terms of the CIA model.

3.5.1 Vulnerability Assessment

Security testing as put forth by SDL and CLASP are performed in the form of vulnerability discovery and assessment for each attack vector. This analysis forms the basis for the recommendations and insights delivered in Chapter 8.

A variety of techniques for performing vulnerability assessment have been discussed: manual code review, static, dynamic, and fuzz testing. In the case studies of this thesis, some of these techniques were often more appropriate than others, depending on the availability of source code, the size of the source code, or complexity of input. While one or more of these techniques were used to find the vulnerabilities considered in Chapters 5-7, we do not evaluate their effectiveness beyond this property. Instead, this thesis purposes to qualify a vulnerability by the threat it poses in an embedded system, provide an analysis and classification thereof, and to consequently provide a recommended secure practice which mitigates this threat. This is in line with NIST's approach to technical security testing and assessment [53] which states that "no one technique can provide a complete picture of the security of a system or network" and that "appropriate techniques to ensure robust security assessments should be combined". Furthermore, the publication emphasizes "*how* these different technical techniques can be performed and does not specify *which* techniques should be used", that is, which are most effective or necessary.

3.5.2 Technical Assessment

At this point, we address the technical considerations necessary for vulnerability discovery and classification for each of the case studies. The technical details of vulnerability assessment in this thesis draw on Dowd's `The Art of Software Security Assessment` [33] and NIST's `Technical Guide to Information Security Testing and Assessment` [53]. As discussed below, a close correlation can be observed between these technical details and the methods of SDL and CLASP.

As Dowd [33] points out, "there isn't a single, clean taxonomy for grouping vulnerabilities into accurate, nonoverlapping classes". It is more valuable to break away from the rigidity of a formal taxonomy, and evaluate the properties of a vulnerability based on vulnerability classes. Dowd recognizes three such classes, and correlates them with phases in the SDL which address these classes. In addition, NIST's guide contains specific Target Vulnerability Validation Techniques [53], providing further details of vulnerabilities that we correlate with the three classes put forth by Dowd. Case study vulnerabilities will therefore be identified and analyzed in accordance with the following distinctions:

- Design Vulnerabilities

- Operational Vulnerabilities
- Implementation Vulnerabilities

3.5.2.1 Design Vulnerabilities

Design vulnerabilities occur when a flaw exists due to an oversight in the requirements and specification of software. SDL phases 1, 2, and 3, attempt to guard against these vulnerabilities in a preventative manner by placing emphasis on identifying the right requirements and specification at the software's initial conception.

For example, consider a Linux-based smart TV which supports a mass storage USB device. The Linux distribution may also support a host of additional USB peripheral devices by means of third party drivers, including keyboards, imaging devices, or printers. Though it may never have been intended for the smart TV to support such devices *by design*, and though it may have no application-level support for such devices, the kernel will still load the appropriate driver when any of these peripherals are inserted. A bug in this code could pose a security threat. We consider such a case in Chapter 7.

Moreover, note that design vulnerabilities may open up the possibilities of implementation and operational vulnerabilities.

3.5.2.2 Operational Vulnerabilities

The existence of operational vulnerabilities depend on the environment or context in which software operates. These vulnerabilities are not due to bugs in a software's source code, but rather depend on how the software is deployed and configured. A misconfiguration or insecure default setting arises once software is deployed in its final state. Detecting such flaws therefore requires a holistic approach, which is why SDL phase 6—attack surface analysis—is appropriate at this point. Phase 6 marks the boundary between the *preventative* and *attacker-centric* methods of vulnerability detection in the SDL.

The following NIST categories [53] could qualify as operational vulnerabilities:

- Misconfiguration
- Incorrect File & Directory Permission
- Symbolic File Link Manipulation
- Race Condition

Note that `Symbolic File Link` and `Race Condition` vulnerabilities are also listed in §3.5.2.3, as their classification depends on the system context. Again, these share the same properties of CLASP's `Synchronization & Timing Errors`.

3.5.2.3 Implementation Vulnerabilities

Implementation vulnerabilities apply to bugs that result from technical programming errors. Again, SDL takes preventative measures in Phases 4 and 5 (refer to Figure 3.1) to guard against implementation vulnerabilities. These vulnerabilities include popularly understood flaws such as buffer overflows and other memory corruption vulnerabilities. In terms of NIST's penetration testing phase, the following are commonly attacked implementation vulnerabilities:

- Kernel Code Flaws
- Buffer Overflows
- Insufficient Input Validation
- Race Conditions
- File Descriptor Attacks
- Symbolic File Link

Note that the `Buffer Overflows` and `Insufficient Input Validiation` vulnerability designations would fall under CLASP's `Range and Type Errors` problem type. Similarly, `Race Condition` vulnerabilities share the same properties of CLASP's `Synchronization & Timing Errors`.

3.5.3 Summary

It is evident that there remains overlapping gray areas of vulnerability classification; depending on the context, the same vulnerability can be classified as one resulting from a *design* decision, and only exhibited in a certain *operational* context. However, in most circumstances, it is possible to unify the classification of a vulnerability across these definitions given enough information and understanding of a vulnerability's cause and impact—this thesis advocates precisely such an approach, and demonstrates such reasoning throughout the vulnerability case studies. Finally, we correlate each vulnerability with an associated method in the SDL and CLASP processes, which in turn accompanies our secure recommendations.

Chapter 4

Embedded System Design

4.1 Overview

In this chapter we consider aspects of embedded system development, and look specifically at operating system, network, and USB functionality. The purpose of this chapter is to provide the necessary technical background information in preparation for the discussion of our case studies. The content of this chapter is largely based on contemporary design of devices as found in Building Embedded Linux Systems by Yaghmour et al. [66]. We note that embedded devices make use of programs and services that are optimized for performance and storage due to having limited hardware resources in contrast to general desktop computers. These characteristics drive design decisions that impact the security of devices, of which we give examples in this chapter. On the other hand, many Linux-based embedded devices share core Linux operating system features and device drivers, which also impact the security of an embedded device, and thus merits likewise discussion.

4.2 Operating System

Aspects of the embedded Linux operating system include the kernel, libraries and utilities, and filesystem implementations. The following sections provide descriptions of the purpose and role of these software components during system design.

4.2.1 Kernel

The kernel interacts directly with a device's hardware, and manages resources such as the CPU, memory, and device peripherals. The kernel supports kernel *modules*; software that is loaded and run within the kernel for added hardware support or additional features. For example, SELinux is a dedicated kernel security module that enforces strong access control policies. Developers may specify the use of SELinux when building the kernel, although doing so will require knowledge of the device's intended use, and how the SELinux model will fit in with other applications on the device. Kernel configuration options include hardening techniques such as Address Space Layout Randomization (ASLR), which loads programs into randomized offsets in memory. Kernel modules and options are configured prior to compilation. Thereafter the kernel is compiled according to a platform architecture. Common embedded flavours include ARM, PowerPC (PPC), and Microprocessor without Interlocked Pipeline Stages (MIPS).

Drivers The Linux kernel contains a plethora of device drivers which support peripherals such as printers, keyboards, graphics cards, and mass storage devices, to name but a few. One of the biggest advantages for using Linux in embedded system is its out-of-the-box support for these devices–portability that no other operating system readily provides. Many of these drivers support devices that operate on the USB protocol stack, and by default all drivers are included.

Virtual Filesystems Linux makes use of virtual filesystems to expose information about hardware, devices, and drivers to user space applications. Virtual filesystems are an abstraction that exist on top of concrete filesystem implementations such as `ext`, and are represented by a hierarchical directory structure. The Device filesystem, `devfs`, provides an abstraction for representing peripheral devices as files, as well as an interface to interact with them. Custom peripherals, for example, will export a specific interface through the `devfs`. On the other hand, interfaces for common peripheral buses, such as USB, are also contained in `sysfs`. The `sysfs` filesystem also contains system information of the kernel, device debugging information, and module options. `sysfs` is the successor of `procfs`; `procfs` is still present on the majority of Linux distributions, and provides access to kernel options, running processes, and memory mappings.

4.2.2 Libraries and Utilities

Libraries Embedded Linux application binaries most commonly use the rich and robust GNU C library (`glibc`). Due to its full features, `glibc` is substantially bigger than alternative C libraries such as `uClibc` and `diet libc`. C libraries are typically statically linked with application binaries on embedded systems, meaning that binary sizes will be affected by the C library chosen. For example, the `BusyBox` binary (discussed below) is roughly 1MB in size when statically linked against `glibc`, and 500KB when linked against `uClibc`. Therefore, on devices that are severely constrained in terms of flash storage resources, an alternative to `glibc` might be considered.

In terms of security, the libraries can be built with a number of options. `uClibc`, for example, includes a menu with a number of security options. These options include compiling Position Independent Executables (PIE) which can harden executables against memory exploitation techniques. However, turning on these options usually impacts runtime performance, and for this reason are subsequently disabled by embedded developers.

BusyBox BusyBox is perhaps the most popular application binary included in embedded Linux systems. BusyBox implements most Unix utility commands and programs in a single executable. It's primary advantage over desktop counterparts (such as `GNU Coreutils`) is its ease of installation and small size. BusyBox includes additional functionalities out-of-the-box, which, depending on relevance, may be included in an embedded device. Such programs include a DHCP server and client, web server, text editor, and package manager.

4.2.3 Filesystems

Filesystems on embedded devices are important for optimizing storage performance and partitioning data according to their attributes. Embedded devices will typically contain a persistent and read-only filesystem, a persistent read-write filesystem, and a temporary read-write filesystem. Popular read-only filesystems include `Cramfs` and `Squashfs`. Writable filesystems include `ext3`, `jffs2`, and `ubifs`. Temporary filesystems include `Tmpfs` and `Ramfs`. The filesystems have varying tradeoffs, including file name length, maximum file size, compression, and so forth. Filesystems do not directly introduce security vulnerabilities into an embedded device, but an error in designing the appropriate residence and permissions of data on a storage medium could.

4.3 Network Functionality

Network functionality is pervasive among embedded devices. These include obvious candidates such as mobile phones, routers, and printers, but also extend to devices such as smart TVs, set-top boxes, and gaming consoles. Some devices support physical ethernet connectivity, while others use wireless technologies, or a combination of both. Other network functionality, such as GSM (Global System Mobile for Communications), is popular on mobile phones, but not ubiquitous among embedded devices.

Embedded devices are designed to support a number of protocols in the network stack (see Table 4.1).

HTTP	Application
TCP	Transport
IP	Internet/Network
Ethernet	Data Link
IEEE 802.3u	Physical

Table 4.1: Simplified network stack

Protocols pertaining to the physical medium and data link layer are implemented in hardware, and do not make use of software implementations. For example, an embedded device can obtain internet connectivity via an ethernet cable, Wi-Fi, or GSM. On the other hand, the Application layer of the network stack is of particular relevance for embedded design. Much of an embedded device's usefulness is derived from its ability to communicate on the application layer using protocols implemented in software, such as HTTP and DHCP. These services protocols, and associated services, do not prescribe how protocols lower in the network stack should operate.

4.3.1 Services

Embedded device functionality is often improved by a number of network services. The following is a description of the most common network services found on embedded devices. These protocols run on the application layer, and most of them are supported by BusyBox. If a service is undesired, it may be disabled in the BusyBox settings.

Telnet The telnet protocol enables users to connect to a remote telnet host and obtain a command-line interface. The telnet protocol passes information across the network in plain-text. Developers frequently use telnet to configure and test devices. In some instances, as with network routers, the telnet service may provide a means for users to configure a device. However, the `SSH` protocol has become more popular for providing access to an embedded device due to enforcing encryption.

SSH The Secure Shell exists for the same purposes as Telnet, while securing the communication channel with encryption. `SSH` may be preferred when performing device changes in a more hostile environment (for example, over the internet). Notably, a `SSH` server/client application is not included in the BusyBox suite; dropbear [4] is a common open-source SSH implementation.

FTP The File Transfer Protocol is the most common protocol used for copying files from one device to another. `FTP` functionality is often exposed on the command-line, and transfers are performed in plain-text across a network. Encrypted variants exist, such as `SFTP` (`FTP` over `SSH`) and `FTPS` (`FTP` over `SSL`), but are not commonly supported on embedded devices. `FTP` is commonly used by developers to copy files and update firmware on embedded devices. In some instances, the `FTP` service may be enabled in production environments as well.

DHCP The Dynamic Host Configuration Protocol is responsible for configuring devices on a network. Specifically, a DHCP server allocates and leases IP addresses to devices that connect to a network. Embedded devices often contain a DHCP client so that they can communicate with other networked devices, either locally or over the internet. Embedded devices may operate either as a DHCP client (such as a smartphone), or a DHCP server (such as a router).

DNS The Domain Name System (DNS) operates a protocol whereby domain names, such as `example.com` are resolved to their physical IP addresses. A DNS server maintains records of the domain-to-IP mappings. Client devices on a network query DNS servers using the DNS protocol in order to resolve domain names. Embedded devices with network connectivity therefore commonly make use of the DNS protocol as clients. Select devices, such as routers, may relay DNS information between server and client.

4.4 USB Specification

4.4.1 Overview

The ubiquity of the USB port extends to the world of embedded systems. Often, the presence of a USB port device will add operational functionality to a device. Even in situations where this is not the case, developers will likely rely heavily on a USB connection to perform development tasks. Therefore, interaction with the USB port is almost unavoidable when building or using an embedded device.

USB ports are often included on an embedded device because of the flexibility that it affords. Common uses include supporting data transfers, firmware updates, and peripheral attachments. In order for this to realise, a variety of device classes specify protocols that communicate via USB. The USB Device Working Group actively maintains documentation relating to these protocols [36].

As a prerequisite to exploring the security implications of USB technology, it is necessary to develop an understanding of its modus operandi. What follows is a non-exhaustive description of the USB specification as it pertains to vulnerability analysis.

4.4.2 Operation

At the highest level, USB facilitates communication between two devices across a bus. One device is designated the role of a USB host, and the other a peripheral. These roles are analogous to a client-server model, as a host is tasked with keeping account of its peripheral's abilities and state. However, initial communication differs from this model, and follows a strict request-response pattern.

The host manages the bus communication flow, and performs requests to learn about the abilities of a connected device. The device issues appropriate responses to these requests. Hosts are further responsible for assigning appropriate drivers to a peripheral based on its responses, and mediating transfers between multiple devices. Comparatively, the host has a more complex task in orchestrating USB communication, and hardware comprises a dedicated host controller chip. Conversely, an 8-bit USB microcontroller (MCU) may suffice for simple peripherals such as keyboards.

A peripheral must, upon connection to the host, respond to incoming requests. Appropriate responses are placed on the bus subsequent to processing a request from the host. While a peripheral may not be able to process all of the eleven requests defined by the USB protocol specification, it must inform the host if it is unable to do so.

The culmination of initial communication (or enumeration) between host and peripheral is that the host places the peripheral in a *configured* state. In this state, the peripheral is ready to be used for its intended function.

4.4.3 Enumeration

During enumeration, the host sends *control* requests in *packets* on a dedicated pipe (hereon referred to as Endpoint 0), to which the device must respond. Importantly, devices must not assume that these requests will occur in a predefined order. Typical enumeration of a USB device consists of the following steps:

1. The device is attached to the USB port, and becomes powered
2. The host detects the device, and determines whether it is a low or full speed device
3. The host resets the device, and waits for it to exit the reset state
4. The host initiates communication by sending a Get_Descriptor request to learn of the device's maximum packet size
5. The host assigns a unique address for the device
6. The host learns about the device's abilities by sending multiple requests

7. The host assigns and loads an appropriate device driver
8. The host puts the device in the *configured* state

The bulk of host-controller software is dedicated to the tasks of steps 6 and 7. In terms of USB functionality, embedded software developers will spend most of their time programming the correct responses that a peripheral device gives in step 6. Therefore, we further focus our attention on this area of design.

4.4.4 USB Descriptors and Requests

Devices respond to requests with *descriptors*. Eleven standard descriptor types exist for USB devices. A full listing of these are given in Appendix A. Not all descriptors are required for a single USB device. Descriptors are sent in response to a `Get_Descriptor` request. Ten other standard requests exist for enumeration and control purposes, and may be viewed in Appendix A.1. We now draw attention to a short summary of common descriptors. For a full listing of descriptor fields, please refer to Appendix A.2

Device Descriptor The device descriptor is the first descriptor that the host is interested in. It contains basic information about the device, including the device class, vendor ID, product ID, and number of configurations. Importantly, the vendor and product IDs may be used by the host to determine an appropriate driver.

String Descriptor The string descriptor contains Unicode strings. Depending on the request, a string descriptor may contain, amongst other things, a device product string, manufacturer string, or serial number.

Configuration Descriptor After receiving a device descriptor, the host will typically ask for the configuration descriptor. This descriptor specifies the device's features and abilities, and is followed immediately by two *subordinate* descriptors: the interface and endpoint descriptors. Important fields include the number of interfaces that the device supports, and its power requirements.

Interface Descriptor The interface descriptor identifies the interface class, as well as the number of endpoints that the device supports. Consider that a device may support multiple uses, corresponding to multiple interfaces. In this context, the interface class is similar to the device class of the device descriptor, but specifically identifies the use of a single interface. Some possible classifications include "HID" (human interface device), "mass storage", and "printer".

Endpoint descriptor For each interface, specific communication pipes are set up, called endpoints. Endpoint addresses, directions, and attributes are specified in the endpoint descriptor. Attributes may include the type of USB communication used, i.e. bulk transfers for storage devices, and interrupt transfers for HID devices.

Additionally, class- or vendor-specific descriptors exist for USB devices. These descriptors further distinguish the differences of functionality between devices. For example, a keyboard maintains a HID class descriptor of type 21h. Going even further, a report descriptor of type 22h is to be expected in the HID class of a keyboard, which describes how key presses should be interpreted by the host.

Thus, we can expect that the host, upon learning that a device belongs to the HID class, would further request a HID descriptor and report descriptor. In this manner all the conceivable functions of USB devices are realised. We will consider three specific interface classes that commonly pertain to embedded devices (in either a peripheral or host role).

Human Interface Device This class includes keyboards, mice, and game controllers. An embedded device can support HID devices by reading and acting on human input encapsulated in *reports* as specified by a peripheral's report descriptor. The report descriptor structure describes the format and size of the reports sent by a HID device. Additional HID-specific requests and responses may also need to be implemented.

Mass Storage The mass storage class allows large data transfers to take place between a host and peripheral. Common devices include USB flash-drives, portable harddrives, DVD drives, and SD cards. A variety of media specifications exist for these devices, the most popular being the Small Computer System Interface, or SCSI. In order to support the SCSI command set, the USB specification implements the Command Block Wrapper (CBW) and Command Status Wrapper (CSW) wrapper structures. Implementing USB mass storage support relies in part on communication via these wrappers, class-specific requests, and a correct understanding of the SCSI command set.

Device Firmware Upgrade The DFU class establishes a protocol whereby a device may be upgraded by a host. For example, this may occur when a wireless dongle has new firmware pushed to it via a desktop computer. The process consists of four phases: enumeration, reconfiguration, transfer, and manifestation. By the end of the process, the device resumes normal operation with the new firmware. As with previous classes, the DFU class has class-specific requests and descriptors to achieve its goal. Developers need to be aware of these requirements in order to implement it successfully.

4.5 Conclusion

When an embedded system is designed, much effort is required to align the inclusion of software with device requirements. Developers need to consider the resource use of a device and its peripherals when building and configuring the kernel. Libraries and programs that run on the device will commonly be optimized to have a small resource footprint—this can translate into security options being turned off for performance purposes. One factor that eases the task of building the system is Linux's wide support for peripherals through device drivers.

Developers similarly have to elect the network services that are appropriate for the device. Enabling some services, such as `ftp`, may be useful during development, but dangerous after production. Depending on the complexity of the device's functions, a number of network protocols may need to be supported. Each network service, in turn, requires compilation and configuration for the embedded device platform.

USB support implies structured yet complex interactions between a host and peripheral. Much of the detail is abstracted away from the developer with embedded Linux, which supports many of the base classes by default. Nevertheless, custom USB functionality requires the developer to be knowledgeable of the specification.

Due to the complex process of constructing a device's software stack as per the system, network, and USB categories, it is conceivable that security issues exist as a result. At each step of a design choice, configuration, and compilation of software, there looms the danger of introducing a vulnerability into the system. Whereas some vulnerabilities may exist due to implementation bugs inherent to the kernel itself, others may be due to a compilation option or network configuration setting. Yet further bugs can be introduced by third-party USB driver code, or custom modules written by the developer. With this in mind, we proceed to identify underlying causes through vulnerability case studies, thereby assisting software processes to address the security concerns during development.

Chapter 5

Embedded Operating System Attacks

5.1 Chapter Overview

This chapter discusses operating system attacks in the context of embedded systems. The operating system is a critical piece of software that handles hardware resources such as the memory, CPU, and input devices, and provides the necessary services and interfaces for higher level applications and programs. Operating systems also manage the filesystem and have access control mechanisms to preserve file permissions. Attacks on the operating system level try to circumvent such protections in order to gain access to sensitive files or to allow arbitrary modification of system behavior, such as running new programs, or modifying existing programs.

Attacks on the operating system of a personal computer may leave the user at risk, but the user is typically allowed complete control over their computer. However, system attacks on embedded devices is great cause for concern for manufacturers, who typically do not want users to have complete control over device behavior. This is the case with popular consumer electronics, such as game consoles, routers, and set-top boxes. Manufacturers of such devices typically restrict the user to a ‘sandbox’ environment, where they are only able to perform a limited set of operations on the system. Alternatively, manufacturers may desire to disallow access completely.

Access to the operating system level of a device is a powerful position, and can often reveal sensitive information in a variety of ways. We consider attacks that can be performed out of this position on an embedded device which runs a Linux operating system. Vulnerabilities are classified broadly as resulting from design, operational, or implementation errors, according to the taxonomy discussed in §3.5.2. As mentioned, classifications for a given vulnerability may overlap, yet one of these three classifications is usually dominant. Where relevant, we describe the specifics of the vulnerability and attack at a technical level, corresponding to the descriptions in §3.5.2. Next, we analyze the vulnerability and potential attack, specifically in terms of the CIA properties introduced in §3.4. From this standpoint, we motivate that vulnerabilities warrant specific consideration in software development processes for embedded systems, building on those established by the SDL and CLASP. The insights gained from this investigative technique form the basis of the recommendations in Chapter 8, specifically in terms of attack surface analysis, threat modeling, or security testing as put forth in §3.4.

5.2 Design Vulnerabilities

Design decisions of both programs and physical devices have a bearing on how secure the product will be. An oversight in this area can introduce vulnerabilities that may allow a device to be compromised, or more commonly, introduce weaknesses that make it easier to exploit operational and implementation vulnerabilities. As such, errors in the design of software do not typically constitute a vulnerability on their own, but introduce further risk, making the device more susceptible to attacks. One aspect that further distinguishes design vulnerabilities from others is that these issues may be addressed at the design stage of software development; usually, it is not necessary to perform a code or configuration audit to identify avenues of attack. When security considerations are taken in account at the design stage, it translates into a system that will be hardened against exploitation attempts.

5.2.1 Case Study 1: Address Space Layout Radomization

Overview In modern operating systems, defense mechanisms are implemented to harden programs against memory exploitation techniques. The front-line defense to mitigating the risk of a memory vulnerability is Address Space Layout Randomization (ASLR). ASLR is performed by the operating system kernel, which is achieved by loading program code and data into memory at random offsets. Without ASLR, program code and data is loaded into memory at deterministic, predictable positions, allowing an attacker to easily execute existing code in memory; a standard technique used to exploit memory vulnerabilities. ASLR may curb the ability to exploit a vulnerability, and at the very least forces the attacker to pursue more advanced exploitation techniques, which normally includes discovering a memory leak that reveals information about the memory layout [41].

There are two prerequisites for enabling ASLR on a device. Namely, the kernel must support loading programs into memory at random offsets, and program binaries must be built with the appropriate compile options. While all desktop operating systems support ASLR, embedded Linux devices typically fail to perform ASLR due to one of these two reasons. This case study considers the state of ASLR on a set-top box.

Technical Description Linux kernels as of version `2.6.12` support ASLR of binaries, given that the option is set in `/proc/sys/kernel/randomize_va_space` and that the binaries are compiled with the appropriate options. In §4.2.2, we alluded to compile options that help to protect embedded libraries from exploitation. Indeed, the `-fPIE -pie` options, enabling Position Independent Executables, are required for ASLR of code for programs in Linux. We found that a set-top box exhibited partial ASLR for program binaries: some were built with the `PIE` options, and anothers not. To evaluate whether ASLR was enabled, the output of `proc/self/maps` was compared before and after a cold reboot, initiated with a power cycle. Comparing two Listings 5.1 and 5.2 we see that the memory maps of the stack and `uClibc` differ due to ASLR. However, the heap address, virtual dynamic shared object (VDSO) and `busybox` mappings do not.

Listing 5.1: ASLR Output 1

```
00400000-00474000 r-xp 00000000 1f:00 200        /bin/busybox
00484000-00485000 rw-p 00074000 1f:00 200        /bin/busybox
00485000-00486000 rwxp 00000000 00:00 0          [heap]
77ef9000-77f9c000 r-xp 00000000 1f:00 277        /lib/libuClibc-0.9.32.1.so
77f9c000-77fab000 ---p 00000000 00:00 0
77fab000-77fac000 r--p 000a2000 1f:00 277        /lib/libuClibc-0.9.32.1.so
77fac000-77fad000 rw-p 000a3000 1f:00 277        /lib/libuClibc-0.9.32.1.so
77fad000-77fb3000 rw-p 00000000 00:00 0
77fb3000-77fb6000 r-xp 00000000 1f:00 267        /lib/libcrypt-0.9.32.1.so
77fb6000-77fc5000 ---p 00000000 00:00 0
77fc5000-77fc6000 rw-p 00002000 1f:00 267        /lib/libcrypt-0.9.32.1.so
77fc6000-77fd7000 rw-p 00000000 00:00 0
77fd7000-77fde000 r-xp 00000000 1f:00 264        /lib/ld-uClibc-0.9.32.1.so
77feb000-77fed000 rw-p 00000000 00:00 0
77fed000-77fee000 r--p 00006000 1f:00 264        /lib/ld-uClibc-0.9.32.1.so
77fee000-77fef000 rw-p 00007000 1f:00 264        /lib/ld-uClibc-0.9.32.1.so
7fcc0000-7fce1000 rwxp 00000000 00:00 0          [stack]
7fff7000-7fff8000 r-xp 00000000 00:00 0          [vdso]
```

Listing 5.2: ASLR Output 2

```
00400000-00474000 r-xp 00000000 1f:00 200        /bin/busybox
00484000-00485000 rw-p 00074000 1f:00 200        /bin/busybox
00485000-00486000 rwxp 00000000 00:00 0          [heap]
77a8a000-77b2d000 r-xp 00000000 1f:00 277        /lib/libuClibc-0.9.32.1.so
77b2d000-77b3c000 ---p 00000000 00:00 0
77b3c000-77b3d000 r--p 000a2000 1f:00 277        /lib/libuClibc-0.9.32.1.so
77b3d000-77b3e000 rw-p 000a3000 1f:00 277        /lib/libuClibc-0.9.32.1.so
77b3e000-77b44000 rw-p 00000000 00:00 0
77b44000-77b47000 r-xp 00000000 1f:00 267        /lib/libcrypt-0.9.32.1.so
77b47000-77b56000 ---p 00000000 00:00 0
77b56000-77b57000 rw-p 00002000 1f:00 267        /lib/libcrypt-0.9.32.1.so
77b57000-77b68000 rw-p 00000000 00:00 0
77b68000-77b6f000 r-xp 00000000 1f:00 264        /lib/ld-uClibc-0.9.32.1.so
77b7c000-77b7e000 rw-p 00000000 00:00 0
77b7e000-77b7f000 r--p 00006000 1f:00 264        /lib/ld-uClibc-0.9.32.1.so
77b7f000-77b80000 rw-p 00007000 1f:00 264        /lib/ld-uClibc-0.9.32.1.so
7f8b1000-7f8d2000 rwxp 00000000 00:00 0          [stack]
```

```
7fff7000-7fff8000 r-xp 00000000 00:00 0          [vdso]
```

In this case, an attacker may be able to leverage the known addresses of the heap or busybox executable to craft a successful exploit.

Analysis Ideally, all executables, libraries, and memory regions should be subject to ASLR. While this is not necessarily true even of desktop environments, adoption of ASLR enabled kernels and programs are highly encouraged in desktop environments. On the other hand, ASLR has been largely dismissed for embedded devices, principally due to the performance overhead that it incurs. By default, PIE compilation options are disabled or actively discouraged, as is the case with `busybox` and `uclibc` [66]. Furthermore, Android only partially supported ASLR in version 4.0 (2011), and fully in version 4.1 (2012) [2]. Evidently, ASLR support for embedded devices is becoming increasingly important.

Miller has demonstrated that the lack of ASLR has made exploitation of embedded devices such as Android and iOS mobile phones easy [47]. Miller states that "As long as there's anything that's not randomized, then it (ASLR) doesn't work, because as long as the attacker knows something is in the same spot, they can use that to break out of everything else..." [50]. As arbitrary code execution is undesired on the set-top box, lack of full ASLR means that predictive memory addresses will be useful to an attacker if a memory corruption vulnerability exists in the set-top box software. Therefore, in terms of the CIA triad, ASLR support affects the confidentiality and integrity properties, in that these are typically violated by exploited vulnerabilities.

Summary As embedded devices become increasingly powerful, the performance impact on ASLR becomes negligible. A more substantial challenge lies in ensuring full ASLR for all components on a given platform. The benefit derived from partial ASLR is questionable in light of Miller's statement. As evidenced by Bojinov et al. [22], ASLR faces many implementation challenges; yet, the pursuit of full ASLR is desirous, as reflected by advances in Android development.

5.3 Operational Vulnerabilities

Some attacks are possible due to *operational* vulnerabilities. This means that there is no bug or inherent vulnerability in the concerned software, but the manner of interaction between the software and the system constitute a vulnerability. These vulnerabilities relate to the attack surface of an embedded device, as each additional software component increases potential misconfiguration or unanticipated default behavior. Detection of operational vulnerabilities should thus happen as a result of considering the interaction between software, such as with the reviews in SDL Phase 6 and 13. Operational vulnerabilities differ from design flaws in that secure behavior can often be achieved by configuring the appropriate settings, or by defining the correct behavior for the existing software.

5.3.1 Case Study 2: Service Misconfiguration

Overview Services run on a system either with their default settings, settings configured by the developer, or both. Misconfiguration can result from any of these actions, and can result in a vulnerability if it violates one of the CIA properties. Interaction between software components is usually required to trigger the problem; a misconfiguration on its own with no means of interaction will remain undiscovered. This case study considers a misconfiguration of the `pppd` service that we discovered on a Linux-based Huawei router, leading to undesired information disclosure when paired with the interaction of a user-supplied system command.

Technical Description Consider Listing 5.3, which demonstrates the vulnerability discovered on the router. By displaying the processes running on the router with the `ps` system command, one is able to view login credentials for the `pppd` process.

Listing 5.3: Truncated output of the `ps` command

```
507 1 user S   113m242.9 0 0.0 clid
799 1 user S   106m228.4 0 0.0 upnpdmain !br+ br0 49652
369 1 user S 93468195.1 0 0.0 ssmp
370 1 user S 86476180.5 0 0.0 bbsp
372 1 user S 86184179.9 0 0.0 omci
375 1 user S 58280121.6 0 0.0 vspa_sip
371 1 user S 51148106.7 0 0.0 amp
373 1 user S 35032  73.1 0 0.0 igmp
867 1 user S 34636  72.3 0 0.0 web
677 1 user S 2808    5.8 0 0.0 pppd nic-wan1 user RTONDER02005 password D572896
444 1 user S 27200  56.7 0 0.0 procmonitor ssmp amp bbsp vspa_si
365 1 user S 10756  22.4 0 0.0 hw_ldsp_user
377 1 user S 10744  22.4 0 0.0 smp_usb
```

The `pppd` (Point-to-Point Protocol Daemon) process is responsible for establishing links with a broadband service provider and can negotiate parameters relating to IP addresses, transmission sizes, and name server addresses. In a scenario where an attacker has unprivileged access to the router (the default case), the username and password for the PPP connection is easily seized.

Analysis Consider that this is neither a flaw in the `ps` command (for it is not responsible for the command-line parameters that may be exposed by processes), nor is it a flaw in the `pppd` program. It is not a flaw in the `pppd` program since specific provision is made to enter this data in a `secrets` file instead of the command-line. This file could be secured in turn by appropriate permissions. The `man` page for `pppd` also specifically discourages the practice observed above:

```
password password-string
  Specifies the password to use for authenticating to the peer.
  Use of this option is discouraged, as the password is likely to
  be visible to other users on the system (for example, by using
  ps(1)).
```

As presented here, the vulnerability violates the confidentiality property by exposing private data which should not be available in the context of the terminal session. Specifically, the credentials could allow the attacker to spoof the identity of the victim's broadband account. This vulnerability is largely unnecessary, and present because due consideration was not given to the context in which the program would be used. The issue is remediated by using the `secrets` file for authentication.

Summary This case study makes a general case for sensitive data exposed via the command-line, as other variants are also possible. For example, if a user types a cleartext password on the command-line, it is likely being logged in a file which may be compromised by an attacker. Scenarios such as these should be envisioned when assessing operational vulnerabilities. This is most appropriate once the interaction between programs is understood, and the device is deemed production ready. The final review phase of the SDL thus affords a sound checkpoint for performing operational assessment.

5.4 Implementation Vulnerabilities

Implementation vulnerabilities constitute the most common type of vulnerability, and relate directly to bugs due to programming errors in software. Not all software bugs imply a security concern; it is only in the event that a bug poses a security threat that it is considered a vulnerability. A threat, therefore, is implied if the bug violates one of the CIA properties.

Implementation vulnerabilities on the Linux operating system can violate the CIA principle in a number of ways. Vulnerabilities may exist in the core Linux distribution, or may be introduced by application software or middleware of a manufacturer. Implementation vulnerabilities are often assigned a Common Vulnerabilities and Exposures identifier (CVE) if they merit public disclosure. Given the abundance and variety of implementation vulnerabilities, we consider two case studies in this section.

This section details vulnerabilities that threaten the security of system environments in an embedded Linux system, and expands on the methods that assist in discovering them. These vulnerabilities may manifest in a variety of ways depending on the device; they are most commonly classified by the characteristic of enabling attackers to escalate their privileges, or bypass file permissions. Identifying implementation vulnerabilities in software is critical to securing the device from undesired file access and program execution.

5.4.1 Case Study 3: Escaping Sandboxes

Overview Some system designs, like that of the Huawei router, allow users to be logged in as root, but place restrictions on the operations that they can perform within this context. The process of doing so is called "sandboxing" and may be performed by programs such as `chroot` [7] in order to to put a user in a 'jailed' environment.

We consider a Huawei router which implements sandboxing by intercepting shell commands, followed by a check against a whitelist of allowable commands. A user can obtain a shell on the router by logging in over telnet. The shell is made available for accessing diagnostic information, but prohibits most actions that change the operational state of the router. Standard executable binaries such as `cat` in the Linux `/bin` and `/sbin` directories cannot be executed, thereby disabling file reads and program execution.

Technical Description Though most commands are disallowed, a number of scripts are available to the user that wrap commands such as `echo` and `cat`. Consider Listing 5.4: the purpose of the `cat` wrapper script is to restrict the user from displaying any files except those contained in the `/tmp`, `/proc`, and `/var` directories.

Listing 5.4: `cat` wrapper script

```
#!/bin/sh

if [ 1 -ne $# ];
then
     echo "ERROR::input para number is not right!"
else
     case "$1" in
      *tmp* | *proc* | *var* )   cat $1;;
      * ) echo "can't read file in this directory!";;
     esac
fi
```

Observe what would happen if the script was executed as follows:

```
$cat_wrapper.sh /var/../etc/passwd
```

The argument would be matched by the first statement, since it matches the `*var*` pattern. The argument is passed to `cat`, which recognizes the traversal to the root directory, and subsequently prints the contents of `/etc/passwd` to screen. The wildcard filter in the script is insufficient, and the protection is bypassed. A similar situation exists in Listing 5.5 for the `echo` wrapper script on the router, which allows arbitrary writing to files.

Listing 5.5: `echo` wrapper script

```
#!/bin/sh

if [ 3 -ne $# ];
then
    echo "ERROR::input para number is not right!"
else
    case "$2" in
     *tmp/* | *var/* )
      if [ 1 = $3 ]; then
          echo $1>$2
      elif [ 2 = $3 ]; then
          echo $1>>$2
      else
          echo "ERROR::input para  is invalid!"
      fi ;;
     * ) echo "can't write file in this directory!";;
    esac
```

Analysis Lack of input sanitization is identified as a specific vulnerability class by NIST, with varying security impact. In the context of the Huawei router, this vulnerability class allows arbitrary reading and writing of files on the router. This breaks the confidentiality and integrity properties of the router, allowing an attacker to read configuration files containing plaintext passwords, and overwrite the router's public keys, for instance.

Moreover, consider that an attacker could append shell commands to an existing script in `/etc/init.d` that runs at startup, leading to arbitrary code execution. Provided that the filesystem is writable, this ability implies a risk more dire than considered originally.

Summary This case study demonstrates the importance of performing input sanitization, especially for bash commands that honour syntax of Linux directory paths. Since the scripts are so small in size, a code review of these would have afforded reliable means to revealing these flaws. This exercise, as emphasized by CLASP, was likely omitted during development. Input sanitization flaws can be remediated in a number of ways. One approach in this case may be to blacklist the "`..`" sequence of characters. A more sound approach may be to modify the scripts to match the command input appropriately.

5.4.2 Case Study 4: System Command Injection

Overview When input to a program contains characters that are meaningful to the execution context, the program may be susceptible to a command injection attack. Command injection vulnerabilities exist due to insufficient input validation, as previously shown in §5.4.1, but differs in impact and context. In embedded devices, there is often a high level of interaction between program execution environments, scripts, and named pipes. Linux scripts can often perform tasks with low-overhead processing, and are well understood by embedded system programmers. A common convention is to have the invoking process supply script arguments with environment variables. In embedded systems, named pipes are commonly used when data is passed between two running processes. Named pipes are preferred for their simplicity: Unix domain sockets offer an alternative to named pipes, but imply greater implementation effort.

Technical Description The fragmented interaction described can make it difficult to anticipate areas of attacks where command injection may be possible. One such discovered case on a set-top box demonstrates this threat. When a new USB storage device is plugged into the device, the kernel spawns a process which calls a script placed in the `/proc/sys/kernel/hotplug` path. This is a userland script which then executes subsequent commands. The script is supplied with environment variables `DEVPATH` and `ACTION` which contain the device's `sysfs` path, and whether it was added or removed respectively. The script then sends this information to a named pipe for processing by additional drivers. The addition of a new USB mass storage device would mean the script executes a line as follows:

```
/bin/echo "sdb1:1" > /tmp/named_pipe
```

The snippet in Listing 5.6 is a proof-of-concept rendition of the code that may be called, under certain conditions, when information is received on the `named_pipe`. In line 2 it performs a basic check to verify that the line starts with 'sd' followed by a non-numeric character. It then creates a new device in the `devfs` filesystem using the `mknod` function in line 15 if it does not already exist. Consider line 10 in Listing 5.6, where the `system` function executes the final shell command (a string) pointed to by `comd`.

Listing 5.6: Custom Linux device hotplug handler

```
1       ...
2       if ((l[0] != 's') || (l[1] != 'd') || (l[2] < 'a') || (l[2] > 'z')
            ) {
3           printf("Invalid!");
4           return 1;
5       }
6       ...
7
8       char comd[80];
9       snprintf(&comd[0], sizeof(comd), "/bin/mknod /dev/%s b 8 %d > /dev
            /null", l, dev);
10          if (system(comd) != 0) {
11              printf("mknod failed!"));
12          }
13      }
14      ...
```

If an attacker is able to manipulate the string to contain additional characters, command injection is possible. Observe that the `comd` string in line

9 is constructed by substituting the string pointed to by the `l` variable. The `l` variable, in fact, stores the string portion before the colon in which is sent to `/tmp/named_pipe` (`sdb` in this case). An attacker could then consider performing the following:

```
/bin/echo "sdb1:1; touch /tmp/hello" > /tmp/named_pipe
```

The intended result would be for `system` to be called on the string above. The semicolon is a special character which separates commands within the shell execution environment, in this case allowing the command `touch /tmp/hello` to be injected. The entire `comd` string would thus be:

```
"/bin/mknod /dev/sdb1:1; touch /tmp/hello b 8 1 > /dev/null"
```

The net effect is that the attacker is able to execute arbitrary commands in the shell. The `system` command in Listing 5.6 may fail, but that does not preclude the attacker's command from being executed successfully.

Analysis Like §5.4.1, this case study demonstrates the impact of insufficient sanitization, albeit in a different context. The attacker is able to directly execute arbitrary code in the context of the system user running Listing 5.6, which could potentially be the root user. This implies that the attacker could gain full control over the device, thereby breaking the confidentiality and integrity properties.

The threat of this attack is benign in this context, due to the set-top box being hardened against outside access to the command shell, which is disallowed by default. In a different context where an attacker may be able to obtain such access by other means, however, the act of exploiting this vulnerability may in turn be able to grant them elevated privileges.

Summary Detecting command injection vulnerabilities can be done by performing static analysis and code searches on `system`-like calls, followed by manual review. The SDL allows for such activities to happen during implementation (Phases 8, 9, and 10). Similarly, CLASP activities including static analysis and code review provide ample means of detecting this vulnerability class. Note that there are two factors that significantly ease the ability of the attacker to exploit this vulnerability. The first is the weak check performed on only the first three characters of the input on line 2 of Listing 5.6. The second factor is the unnecessarily large character buffer on line 8, which allows the attacker to inject a long string command. Thus, remediation of the problem should ideally consider both these factors. This further highlights the importance of reviewing all portions of the code which contain or check input data that become executed in a command such as `system`.

5.5 Chapter Summary

In this chapter, we considered four case studies in the context of the system vulnerabilities and attacks that impact system security. These case studies exhibited a variety of vulnerability classes, their impact and threat, and how security processes could be formed to address them.

We demonstrated the lack of ASLR adoption on embedded devices, emphasizing the importance of enabling system hardening techniques. Exploring Linux system options during the design phase of a device contributes to securing it against potential violations of the confidentiality and integrity properties. The operational vulnerability class demonstrated the importance of considering the interaction of system commands and service defaults. Knowledge of application behaviour in conjunction with the Linux system during implementation would mitigate the threats posed by such vulnerabilities. The implementation vulnerabilities demonstrated security threats due to insufficient input sanitization and command injection. These vulnerabilities involve interaction with a wide number of system functions, namely, shell commands, system binaries, file access, environment variables, and Unix pipes. The importance of code review for scripts and programs was stressed—an exercise that can be performed meaningfully by the secure software methods found in the SDL and CLASP.

Chapter 6

Embedded Network Attacks

6.1 Chapter Overview

Network attacks pose a great threat to many platforms: personal computers, enterprise computing environments, and embedded systems. In the context of embedded systems, network attacks can sometimes exploit services in unexpected ways and have unforeseen consequences. Other more common attacks, such as DoS attacks, can also afflict embedded systems.

Network attacks may vary greatly depending on available services and applications. In terms of the network protocol stack covered in §4.3, most network attacks occur predominantly at the application layer. In some circumstances vulnerabilities may exist in software related to the `Transport` and `Network` layers. `CVE-2012-6638` is one such example, enabling remote attackers to cause a denial-of-service with crafted `TCP` packets. Thus, as with other platforms, it is important to consider the implication of different network protocols and modes of transport for embedded devices. Embedded systems, however, distinguish themselves from other platforms in terms of the application layer software that enable common network functionality. Therefore, concerning *software* security of networked embedded devices, we restrict our analysis of vulnerabilities to the application layer in this thesis

The case studies in this chapter cover network attacks on common application layer protocols that affect embedded systems, such as `DHCP`, `Telnet`, and `SSH`. We also demonstrate how the *interaction* of these protocols with web-facing applications can be exploited by an attacker. As in Chapter 5, each case study is structured to include a description and analysis, supporting the recommendations in Chapter 8.

6.2 Design Vulnerabilities

Similar to §5.2, we consider design concerns of the embedded Linux devices in terms of network services and software. Network services provide an endpoint to the outside world, requiring special attention during design. We concentrate on the necessity of protecting the integrity of such networking services, and the role of software updates. As shown in §5.2.1, limitations in design can significantly impact the security of a device, without constituting a vulnerability in itself.

In this section we address the growing need for real time software updates on embedded devices, specifically in the context of network facing software. Of the devices considered in this thesis, only mobile phones allow a reliable means of updating their own software. Red Balloon Security reports that only 7% of vulnerable devices are ever updated [25]. Although the implications of this issue affect all software on the device, including system and USB software, the risk posed by the lack of software updates is best highlighted in terms of remote attacks.

6.2.1 Case Study 5: Outdated Network Software

Overview Many embedded systems are produced without effective means of updating the device software or firmware. This is especially true of network routers and switches, but also of networked printers and smart TVs. There are many challenges in delivering updated software to a device. For one, manufacturers may or may not provision updated firmware on a product web page. Even when updated firmware is made available, it may require specific user interaction to apply an update—we consider a particularly difficult case with network routers and switches. Furthermore, implementing an automatic update mechanism can prove costly and complex for embedded devices. Consider that it may not be desirable to allow devices, such as printers, to retrieve data outside of a corporate network in order to update. In other instances, it may be critical that a device, such as a network switch, does not experience any downtime (whether due to updates or otherwise). Lastly, manufacturers may deem it unnecessary to issue updated software in some cases, meaning that an update mechanism is completely absent from the device. We consider further security implications that such scenarios lead to.

Technical Description In the course of our research investigation, we found a number of routers and switches to contain vulnerable outdated software, with no easy means of patching. Namely, vulnerable versions of the BusyBox client were found on a Huawei router and set-top box, as well as vulnerable SSH versions on Dell switches.

BusyBox verion 1.18.4 was found to run on the devices, and is affected by `CVE-2011-2716` [16]. In summary, this implementation vulnerability may allow an attacker to execute arbitrary code on the device through command injection. Importantly, from a design perspective, we wish to highlight that the vulnerability had been disclosed in 2011, preceding the manufacturing date of the router and set-top box. Similarly, a vulnerable version of SSH was found on Dell switches during our research (`CVE-2013-3594`), which allows remote attackers to perform a denial-of-service attack or even execute arbitrary code.

Analysis The devices running vulnerable code clearly constitute a security threat. Yet, from a design perspective, two important properties of this threat need to be taken into account. Firstly, we identified that known vulnerable versions of applications, such as BusyBox, can still be found on embedded devices long after the vulnerability has been disclosed. Secondly, it can be anticipated that new vulnerabilities will be present in devices as they are discovered over time, as seen with the Dell switches.

Patching these devices is a suitable course of action, but a number of problems persist. For instance, the Dell switches have not been released with an updated version to date, nor is it anticipated that a fix will be delivered. The proprietary nature of the Huawei router and set-top box imply that end-users will not be in a position to update their device. In both cases, there is no means by which the manufacturer can push an update to the devices, nor would it necessarily be appropriate. That is, routers and switches may be configured to only operate within a local network in accordance with a company's network security policy. Even more problematic is the case where the routers and switches are publicly exposed on the internet.

All properties of the CIA triad are potentially violated by the presence of these known vulnerabilities. Furthermore, these vulnerabilities are most severe when they are remotely exploitable over a network. For instance, a device may be compromised if remote execution succeeds, breaching the integrity of the device. Remote denial-of-service attacks, as mentioned above, violates the availability property. Moreover, these threats persist on embedded devices foremostly due to the lack of an update mechanism, and not because of a particular vulnerability.

Summary The difficulty of securing embedded devices with the latest software is evident from this case study. Although some progress has been made toward pushing software updates to mobile platforms such as Android, there remains a great level of version fragmentation, leaving many devices vulnerable to existing issues. While Android demonstrates that an update mechanism is desirable, we find that it may not readily translate to other devices such as switches within a corporate network.

In all instances, it is critical to update software packages that pertain to network functionality, which may allow attackers to perform remote exploits. Platform-specific runtime techniques for achieving this is a topic of ongoing research, as covered in Chapter 2. We stress, however, that the thinking behind this case study should also be present during the secure development process, where manufacturers consider the extent of potential threats by way of threat modelling. For example, during the design phase of the SDL, developers should consider how software updates will be made available, whether an update mechanism should be implemented, and how exposure of the network attack surface could be reduced.

6.3 Operational Vulnerabilities

In §5.3 operational vulnerabilities were described as being due to the interaction between software and it's operational context. An alternative way to view this is to consider that an operational vulnerability is not evident in the source code of the software. A concrete example of such a network-based attack is the consumption of a server's connection pool, resulting in denial-of-service. Although the effects of such an attack can be reduced, the potential risk persists due to the ability of other computers to connect. Thus, addressing such an attack may not be dependent solely on implementing additional code, but also by considering the risk that the device's operational environment poses. Next, we consider a specific case where attention should be given to the implicit trust between a device and the servers with which it communicates.

6.3.1 Case Study 6: Implicit Network Server Trust

Overview The `/etc/resolv.conf` is a plain text file on Linux based systems that is used by the Domain Name System (DNS) for performing hostname-to-IP lookups. An example of a such a file can be seen in Listing 6.1.

Listing 6.1: Example `/etc/resolv.conf` file.

```
search example.com
nameserver 172.16.1.254
nameserver 172.16.2.254
```

The `search` keyword indicates domain names that can be be used to resolve the fully-qualified domain name of a server when a partial domain name is supplied. The `nameserver` keyword designates the IPs of servers on the network that can perform hostname lookups. The contents of this file is implicitly trusted by system programs and services; consequent modification of it can present a security issue.

Technical Description The Linux-based decoder in our tests uses the BusyBox `udhcpc` DHCP client for network communication. The `udhcpc` client executes a script when a lease is obtained from the server. The script, in turn, populates the `/etc/resolv.conf` file with domain names and nameservers[1]. When operating as intended, the set-top box is able to correctly resolve addresses and communicate with the outside network, or the internet.

[1]The full script can be seen in Appendix B.1.

The set-top box implicitly trusts that the nameserver information supplied by the DHCP server is correct.

A DHCP server, however, may be an attacker controlled entity. We assumed the role of the attacker and used a minimal DHCP implementation to verify that the `/etc/resolv.conf` file on the set-top box is updated with our attacker-supplied nameserver.

Analysis It is acceptable for the DHCP server to supply DNS information, and for a device to update its `/etc/resolv.conf`. Consider, however, a scenario where the set-top box relies on an external host or IP address for updating its software. An attacker may utilize a DHCP server to perform a man-in-the-middle attack by supplying their own nameserver, which, when queried by the set-top box, responds with an attacker's version of the updated software.

Of course, for such an attack to succeed, we assume that the set-top box doesn't perform any verification on the software payload (which is unlikely). What is apparent is that there is no built-in protocol for the set-top box to verify the authenticity of the nameserver supplied by the DHCP server. On most devices, authentication checks are deferred to other applications in the network stack—these may be inadequate or absent. Such an attack can therefore principally affect the integrity and availability properties of a device.

Summary The notion of attacker-controlled variants of DHCP, DNS, or other network servers is an important consideration during software development and device configuration. The potential for an attacker to misuse the implicit trust of such servers will differ from device to device, and may require independent analysis to determine the overall risk. If appropriate, one way of reducing such risk is to protect the `/etc/resolv.conf` from modification by setting it to be read-only.

6.4 Implementation Vulnerabilities

Implementation vulnerabilities in network software directly enable remote attacks. Depending on the deployment environment of a device, it may be remotely accessible over a local network or from the internet. Like Chapter 5, we consider two case studies based on implementation vulnerabilities. These vulnerabilities relate closely to published CVEs in the domain of network attacks.

As evidenced by our case studies, these vulnerabilities can enable an attacker to initially compromise a device remotely. From such a position an attacker may pursue escalation of privileges in order to obtain complete control over a device. The case studies also reveal the susceptibility of embedded devices to these attacks when compared to other platforms.

6.4.1 Case Study 7: Command Injection with DHCP

Overview The potential for command injection, as introduced in §5.4.2, is a vulnerability class which also afflicts network software. An attack may be launched if network data is processed with characters that are meaningful in the execution context of a client or server. We consider such an attack relating to `CVE-2011-2716` which was briefly mentioned in §6.2.1.

Technical Description During attack surface analysis of network services on the Linux set-top box, it was discovered that version version 1.18.4 of BusyBox ran on the device. In 2011 a vulnerability was discovered in the source code of the client implementation, which does not properly sanitize certain options in DHCP server responses.

Investigation of the BusyBox source [3] reveals that the client program, `udhcpc`, does not sanitize certain characters in the `HOST_NAME`, `DOMAIN_NAME`, `NIS_DOMAIN`, and `TFTP_SERVER_NAME` DHCP options. This means that the client accepts a string containing any characters for these four options, including shell metacharacters such as '`;`', '`` ` ``', '`$`', and so forth. A typical DHCP server would use a small subset of characters, mostly alphanumeric, as specified in Section 2.3.1 of RFC 1035 [10]. However, using a custom DHCP server, an attacker could exploit this flaw by constructing a string containing shell metacharacters, and subsequently issue them to the client.

Recall from §6.3.1 that the `udhcpc` client runs a script once it obtains a lease from the DHCP server. Before the script is run, `udhcpc` populates a number of bash environment variables with the supplied string options. These variables are used by the script to configure the set-top box's network settings. For example, consider the followng line contained in the `udhcpc` script listed in Appendix A:

```
[ -n "$domain" ] && echo search $domain >> $RESOLV_CONF
```

This line appends the string contained in the `$domain` variable to the `resolv.conf` file. The `$domain` environment variable contains the string supplied by the DHCP server in the `DOMAIN_NAME` option. Next, we consider the risk that the vulnerability introduces.

Analysis Suppose that an attacker supplies a string such as:

```
example.com; rm /etc/passwd
```

Under normal conditions, strings supplied with DHCP options are trusted by the `udhcpc` script. Should the developer assume that it is safe to use the supplied string in other contexts, and perhaps execute it with an `eval` command in the script, then the `/etc/passwd` file will be deleted with the string above. Using the `eval` command in such cases is considered bad practice, but there is no way to guard against the way in which the string will be used once it is trusted by the operational environment. Furthermore, the string may also be used in other unanticipated places, such as a conditional statement. Indeed, the advent of the Shellshock bug demonstrates how command injection can be performed in DHCP clients [11].

The risk of this vulnerability is limited on the set-top box because the `$domain` environment variable is interpreted literally throughout this particular `udhcpc` script. However, this vulnerability has the potential to violate the integrity property of the device by giving the attacker a way to execute commands on the device for scripts where this is not the case.

Consider the risk of this vulnerability to desktop computer versus an embedded device. In the former case, the computer may operate within a local network which already has a DHCP server. This presents a challenge to an attacker who would seek to craft malicious, conflicting requests with a rogue DHCP server on the same network. Furthermore, such a scenario assumes that the attacker has access to the local network. However, in the latter case, the attacker may be interested in compromising a proprietary embedded device (such as a locked-down set-top box). In this scenario, the attacker is free to dictate the network environment without restriction so as to exploit the device's network attack surface.

Summary In §6.2.1 we considered the importance of updating software on embedded devices as a means to mitigate implementation attacks such as the one considered here. This case study, however, presents an important consideration in its own right. In particular, it highlights the importance of considering the exploit techniques that attackers may use to compromise a device. Embedded devices have the unique attribute of inherently trusting its network environment. Furthermore, because network software such as the `udhcpc` implementation is specific to resource-scarce embedded devices, it requires separate scrutiny. Translating this to a secure development process, such scrutiny entails threat modeling (such as those constructed in the scenarios above) as proposed by Phase 7 of the SDL and CLASP. To combat command injection and similar vulnerabilities, a manual code review as promoted by CLASP and SDL is appropriate.

6.4.2 Case Study 8: Cross-site Scripting via Telnet

Overview Cross-site scripting (XSS) attacks are listed as one of the top ten most common web application vulnerabilities [49]. Cross-site scripting attacks are not specific to a given platform, but rely on Javascript injection into web pages that are interpreted by a browser. However, the exploitation and threat of XSS attacks differ among contexts and platforms. This case study demonstrates how an XSS attack can be performed in the context of a Linux based router to obtain access to the administrator web panel. A unique attribute of this attack is that Javascript code is injected via the login prompt of the telnet service.

Technical Description This vulnerability affects a Huawei router that allows legitimate users with the appropriate credentials to log in via telnet or a web page. Users are then able to change configurations and settings through a command shell or web administration panel, respectively. However, when a login attempt over telnet fails, say, for username `foobar`, an entry is created in a log view of the administration panel as follows:

```
2013-12-12 17:06:01 [Error] [CLI] foobar loginfail!
```

This allows the attacker to inject input into the log page. When the input is crafted in a specific way, a XSS attack can be launched by injecting input that will be executed as Javascript code. In most scenarios, XSS is mitigated by sanitizing any user input that is rendered on web pages, such that the input is not interpreted as valid Javascript. Although the router sanitizes the username input by removing characters such as '$<$' and '$>$', it can be bypassed. Specifically, the log entries are contained in an HTML `textarea` tag. By first terminating the text area with a closing `<\textarea>` tag, the attacker is free to inject subsequent unsanitized input into the log page. Presumably, the router does not perform the sanitization check on input that extend the HTML `textarea`. In this manner, a successful exploit could be constructed such that the attacker could steal an authenticated session cookie from the victim. We consider this scenario in the following analysis.

Analysis The threat of this this vulnerability is that malicious Javascript code can be injected and executed in the context of a legitimate browser session when the log view is visited by the authenticated victim. The malicious code could send the session cookie of the victim to the attacker, which would allow the attacker authenticated access to the administrator interface without requiring login credentials.

This vulnerability violates the confidentiality property in divulging the session key information to the attacker. The integrity property of the device can be further breached once the attacker gains access to the administrator interface.

The severity of the vulnerability is limited by the need for an authenticated user to visit the log page. If the victim is known, the attacker may attempt to coerce them into clicking a link that goes directly to this page. Another factor that impacts the severity of the vulnerability is the exposure of the telnet service. By default, this service is enabled. There is further risk in that any unauthenticated attacker with access to the service can launch an attack.

Summary The danger of this vulnerability lies in insufficient input sanitization of data that is rendered on administrator web page, as found in NIST's classification. It was assigned CVE-2014-0337 [15] and was discovered during the course of this research. In the context of cross-site scripting, all such input should not contain characters that could be interpreted as valid HTML or Javascript code. Although this sanitization should not be performed by telnet, the presence of this service increases the attack surface, and in this instance, makes the XSS attack possible.

This case study demonstrates the necessity of considering the attack surface in terms of network services, an exercise that may be performed as put forth by SDL Phases 6 and 13. Performing correct sanitization would remediate this flaw, but identifying potential avenues of input in accordance with attack surface analysis would more effectively address the issue during development.

6.5 Chapter Summary

In this chapter four case studies were presented, emphasizing the danger of attacks that can be performed on the network attack surface of embedded devices. We showed the risk posed by vulnerabilities of outdated software, and the importance of protecting network entry points of devices from this perspective. Secure development processes must take into account such possibilities during the initial stages of designing a device by considering the device's lifecycle and potential network exposure. Case studies §6.3.1 and §6.4.1 reveal security issues resulting from the implicit trust that an embedded device exhibits within a networked environment. The former case study considers man-in-the-middle techniques that could allow an attacker to send forged data to a device which believes it is communicating with an authentic server, say, for purposes of information or software upgrade retrieval. In contrast, the attack in §6.4.1 relies on a bug in the software which may potentially allows an attacker full control over the device; the severity of which is more apparent. The last case study demonstrates how an attack can be launched by combining data passed to network services, and web-application layer attacks.

The complex state of networking software resulting from update mechanisms, configuration, and software bugs demand proper attention during the development process. Threat mitigation for these vary from effective design decisions to manual code reviews—methods which we derived from SDL and CLASP. Finally, our case studies reveal that the impact of vulnerabilities can be seemingly benign, or otherwise carry severe consequences; importantly, a secure development process should be able to detect and evaluate both.

Chapter 7

USB Attacks

7.1 Chapter Overview

Having considered the USB specification in §4.4, we are in a position to consider the USB attack surface. To date, security evaluation of the USB attack surface has not been explored as thoroughly as system and application software. While some research has been done in this area [40, 28], progress has been hampered by the lack of efficient, automated ways to test USB code. Often times special hardware is required to do so, such as a USB protocol analyzer. To echo the words of Joshua Wright, "Security will not get better until tools for practical exploration of the attack surface are made available" [65].

Nevertheless, just like the system and network attack surfaces, the USB attack surface may contain design, operational, and implementation vulnerabilities. In 2012, practical exploration of the USB attack surface was made possible through software-driven USB device emulation. The Facedancer is one such a hardware device that emulates USB devices, specifically for the purpose of fuzzing USB hosts and peripherals [23]. Currently, a few varieties of USB attack tools exist; our case studies primarily make use of the Facedancer [1] tool which has achieved a mature status after numerous hardware and software revisions. In this chapter, we first introduce a new framework for discovering USB bugs which improves upon existing tools. The Transparent Two-Way Emulation (TTWE) framework [60] was developed during the course of this thesis in the interest of achieving greater flexibility and affordability for USB fuzzing.

This chapter then documents the types of USB bugs we discovered during the course of this research paper (Case Study 9), as well as by other parties (Case Study 10, 11 and 12). Emphasis is placed on the concerning implications that USB bugs have on embedded devices. In particular, the ubiquity of USB functionality on embedded devices can serve as an entry point for an attacker. An attacker who leverages a vulnerability in the USB attack surface may be able to break into a device and obtain control of it by running their own code.

As before, the objective is to expound on the danger exhibited in the case studies below. From this analysis we proceed with secure recommendations, this time in the midst of the growing landscape of USB attacks.

7.2 USB Fuzzing by Transparent Two-Way Emulation

We deviate slightly from the structure of Chapters 5 and 6 by introducing the TTWE framework for exploring the USB attack surface. In Chapter 4 we discussed the behavior of the USB protocol, and the ubiquitous presence of the USB hardware port in devices. Due to these unique properties, we acknowledge that the notion of performing "security assessment" for the USB attack surface therefore necessitates a specialized approach, aided by specific tools.

The essential properties of the TTWE framework is that it is flexible, cost-effective, and lowers the knowledge requirement for performing USB fuzz testing. In particular, the TTWE framework enables man-in-the-middle modification of the USB communication between a host and device. Moreover, the user is not dependent on device- or platform-specific software to drive the USB testing.

While the framework allows platform-independent man-in-the-middle attacks on the USB protocol, this important feature is also possible with a USB analyzer [9]. However, Davis [28] mentions the limitations of the software provided with such tools: they lack a proper software API, requiring the user to make use of a custom scripting language [29]. Our framework does not place restrictions on the user's ability to manipulate USB data. The host and client emulation drivers are implemented in Python and expose the raw USB data over Unix pipes. If desired, other forms of inter-process communication (IPC) such as Unix sockets could also be employed. We opted for named pipes due to the simplicity of having host and peripheral emulation drivers attach to these endpoints, instead of implementing extra IPC functionality i drivers themselves. Moreover, the user is not restricted to Python, and if desired, may implement drivers that interact with the Facedancer in any preferred language. A further advantage of our framework is the ability to immediately replay USB communication from either a host or peripheral in an emulated fashion—the user need not write additional scripts to generate USB traffic.

Despite the software limitation of the USB analyzer, its capabilities go beyond that of other tools and pure software solutions. Davis [30] attested in 2013 that it is the preferred device for finding USB bugs, although at a cost of approximately $1,400. In contrast, the hardware required by the Facedancer's in our solution would cost approximately $250 at the time of writing [6].

The TTWE framework also introduces the concept of *endpoint hijacking*, whereby endpoints are transparently remapped between the emulated USB device and authentic USB device. This action is necessary to overcome the limitation of hardcoded USB peripheral endpoints. Because the overhead of the framework introduces a speed limitation of USB communication, we also make use of *handshake emulation*, a design optimization employed to improve transfer speed.

7.2.1 Design

With the aid of two bespoke hardware testing devices, namely, two Facedancer devices [1], we are able to expose USB communication to a Mediating Computer (MC) with a man-in-the-middle strategy. These devices are essentially USB controllers that can act as either a USB host *or* device. On its own, the Facedancer device can perform USB host or device emulation via software driven commands from a computer.

In our design, and with reference to Figure 7.1, we place one Facedancer in Peripheral Emulation Mode to interact with a USB `HOST`, and a second Facedancer acting in Host Emulation Mode to interact with a USB `PERIPHERAL`. By monitoring the hardware interrupts triggered on the Facedancer USB controllers, the MC is able to mediate communication by forwarding requests from the `HOST`, and responses from the `PERIPHERAL`. The `HOST` is under the impression that it is communicating with an authentic USB peripheral, but in fact it is an *emulated* USB peripheral whose responses are precisely that of the authentic USB `PERIPHERAL` monitored by the Facedancer in Host Emulation Mode. Similarly, the `PERIPHERAL` device is under the impression that it is communicating with an authentic host, whose requests are in fact *emulated* by continuously monitoring the authentic `HOST` using the Facedancer in Peripheral Emulation Mode.

The MC monitors the host by checking whether the Facedancer has received host requests or data, and also forwards device responses to the host via the Facedancer. This operation is performed by a USB client driver on the MC. In likewise manner, a USB Host driver monitors the Facedancer, and forwards host requests. The USB client and host driver exchange data via named pipes on the MC—this allows the USB data to be exposed and manipulated by any intermediary software, such as a mutation fuzzer, before the data continues on its normal course.

Whereas understanding the design is straightforward, there are a number of caveats which present themselves when implementing the framework; we address these in the next section.

7.2.2 Implementation

The design addresses the conceptual way in which we are able to mediate and tap into USB communication. There are, however, aspects of the USB

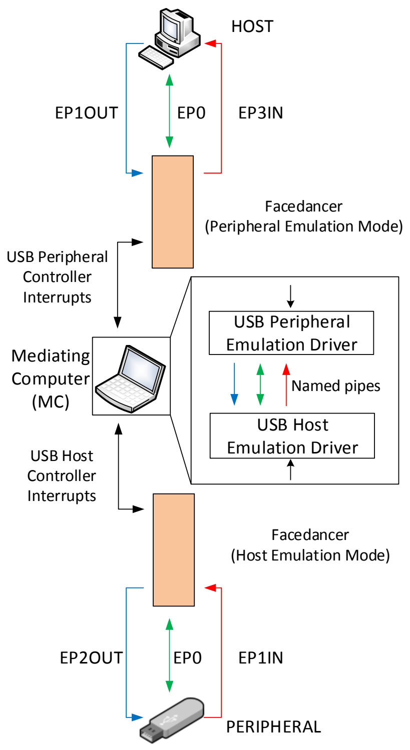
HOST
EP1OUT
EP0
EP3IN
Facedancer
(Peripheral Emulation Mode)
USB Peripheral Controller Interrupts
USB Peripheral Emulation Driver
Named pipes
Mediating Computer (MC)
USB Host Emulation Driver
USB Host Controller Interrupts
Facedancer
(Host Emulation Mode)
EP2OUT
EP0
EP1IN
PERIPHERAL

Figure 7.1: The TTWE Framework Architecture

protocol that do not translate readily to the framework design. These aspects include a) endpoint address numbering, and b) USB handshaking that pertain to control transfers without data stages. In this section we discuss the hardware and software requirements of the framework, and furthermore emphasize the role of software in overcoming the caveats mentioned.

7.2.2.1 Hardware

The functionality of the Facedancer device is central to the operation of the framework. Currently, the Facedancer is an affordable device ($75 at the time of writing) with attractive USB testing abilities for the purposes of our framework. However, any device with similar functionality can serve as a substitute; the framework is not specific to the Facedancer. For this reason, we briefly mention the major hardware components that bring about the required functionality.

As shown in Figure 7.2, the main hardware components of the Facedancer are an FTDI USB/serial adapter chip, a 16-bit microcontroller, and a MAX3421E USB controller chip. USB emulation can be performed by sending software-driven USB data and commands to the microcontroller via the FTDI adapter. The microcontroller drives the USB controller, which may be placed in either *host* or *peripheral* mode by toggling a `mode` bit in one of the controller registers. Recall that in Figure 7.1, we place one Facedancer's USB controller in host mode, and the other in peripheral mode. The microcontroller listens for responses from the USB controller, which are in turn forwarded back to the computer driving emulation. Basic firmware is required for the microcontroller to perform these actions; such firmware is readily available for the Facedancer components [5].

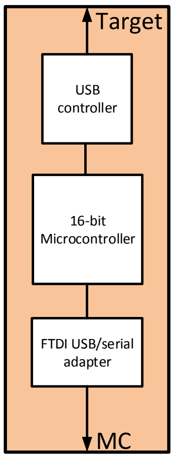

Figure 7.2: The Facedancer hardware design for performing USB device emulation

7.2.2.2 Software

The USB host and client drivers monitor the respective USB controller interrupts to determine when data can be sent and received. The drivers process the endpoint source and destination of data once it is available, and this data is sent between the drivers on dedicated named pipes. Another important responsibility of the software is handling special cases of the USB protocol. Problems that arise due to endpoint numbering are addressed by an *endpoint hijacking* approach. Complications in USB handshaking are handled by emulating certain aspects of the handshake procedure.

Endpoint Hijacking When a host receives an endpoint descriptor from a device in response to a `get_configuration` request, it is informed of the endpoint address(es) that the device intends to use for non-control transfers. For example, consider the USB device in Figure 7.1 (depicted as a mass-storage stick), which requires a bulk `IN` endpoint with address 1, and a bulk `OUT` endpoint with address 2. Whereas the direction (`IN` or `OUT`) is always from the perspective of the host, the endpoint addresses, directions, and transfer types are specified in an endpoint descriptor which are fixed by the peripheral's USB controller.

When the MAX3421E USB controller operates in Host Emulation Mode, it is able to send and receive on *any* endpoint number. However, when the MAX3421E USB controller of the Facedancer in Figure 7.2 is operated in Peripheral Emulation Mode, its endpoint capabilities are fixed in hardware, and cannot be changed. These capabilities are as follows:

Endpoint Address	**Direction**	**Transfer Type**
EP0	IN/OUT	Control
EP1	OUT	Non-control
EP2	IN	Non-control
EP3	IN	Non-control

Consider that an authentic USB peripheral may have *different* endpoint capabilities, which are also fixed. It is thus possible that endpoint addresses can be mismatched between the emulated peripheral's USB controller and the authentic USB peripheral's. In our case, the USB controller supports only an `OUT` direction on `EP1`, yet the authentic peripheral specifies that it must use `EP1` with an `IN` direction. To overcome this problem, we "hijack" and modify the endpoint descriptor with the MC. When the MC detects an endpoint descriptor being sent from the authentic peripheral in response to a `get_endpoint_descriptor` from the host, it creates a new mapping whereby everything received on `EP1IN` of the USB host emulator will be sent on `EP3IN` of the USB peripheral emulator. Similarly, a mapping is created for the `EP2OUT`/`EP1OUT` pipe.

With the endpoint mapping in place, the authentic host and device can continue communication on the perceived information pipes. Since the USB specification does not mandate the capabilities of endpoints other than endpoint 0, the nature of the endpoint addressing scheme in our framework is thus obscured from the host and device, allowing a transparent data channel. With this scheme we can cope with any manner of endpoint addressing that a peripheral may require, provided the USB controller supports the same number of endpoints.

Emulating Handshaking A further consideration in our framework concerns USB control transfers which do not have a data stage. These control transfers include `set_address`, `set_configuration`, `set_interface`, and `clear_feature` requests. Because these transfers do not have a data stage, the peripheral would simply respond with a status packet, such as an `ACK`. The framework makes provision for mediating data transfers across the pipes, but for status packets, one of two work-around solutions is required:

1. After forwarding the host request, blindly acknowledge it with the peripheral emulator, without knowing the authentic peripheral's status result.

2. Communicate the authentic peripheral's status result once available. This requires extra logic so that custom status messages can be communicated between emulator drivers.

We opted for the former approach. This allows us to asynchronously `ACK` requests, and assume that the request will be successfully processed by the authentic peripheral. This poses the question, what if the request is not processed successfully, and the authentic peripheral responds with something other than an `ACK`? In such an event, subsequent requests from the host would not receive a response, and the breakdown of communication would become apparent on the host and client driver software. During testing, we observed this communication breakdown when we neglected to blindly `ACK` the aforementioned requests. After doing so, we did not encounter the scenario again, and obtained our initial results.

7.2.2.3 Results

The TTWE framework culminated in two significant results. The first significant result is the ability to expose USB communication between an authentic host and device, enabling man-in-the-middle attacks without requiring prior knowledge of the USB protocol. The second significant result is the discovery of new bugs in USB software.

Device Emulation For demonstrating the ability to transparently emulate a host and device pair, we used a mass-storage USB stick and were able to perform mount, browse, read, and write actions with the host. By exposing this data through the MC and hijacking the endpoint descriptor, we are able, for example, to fuzz both the host mass-storage driver and the peripheral firmware. An example of this communication captured by the TTWE framework may be viewed in Appendix C.1.

USB Fuzzing We discovered a number of bugs in USB drivers of some popular operating systems, as well as a printer. This included USB printer, wi-fi dongle, and mass-storage device drivers. Due to the fact that these bugs do not directly impact any of the devices in our case studies, we limit the discussion of their details. Nevertheless, we emphasize the importance and threat of USB bugs, and promote the TTWE framework as an effective way to perform fuzzing in this area. In the interest of improving the implementation and the bug-finding results of the TTWE framework, a prototype is available publicly [12].

7.3 Design Vulnerabilities

The most critical design factor that enlarges the USB attack vector is the default inclusion of device drivers in the standard Linux distribution. As alluded to in §4.2, the Linux operating system includes drivers for various USB classes, supporting thousands of devices [8]. This is an advantage for desktop users of Linux, where the user's needs are unanticipated, thus making it sensible to provide the widest coverage of supported devices. To the contrary, it is likely in the best interest of a manufacturer of embedded devices to restrict the number of device drivers to that which is strictly necessary.

In this section we discuss the ramifications of the included-by-default thinking that afflicts the security of embedded devices.

7.3.1 Case Study 9: Default USB Driver Support

Overview We investigated the presence of default USB drivers on two proprietary set-top boxes. The set-top boxes are hardened so as to disallow access from the outside via the network port. Moreover, USB functionality for high-level applications, such as movie playback from a USB storage device, does not exist. The presence of the USB port on the set-top boxes are not intended to serve any additional functionality to the consumer. In some instances, the USB port may be used by field agents to update the firmware on the set-top boxes in rare circumstances. In development environments, the USB port may also be used to update firmware for testing.

By using the Facedancer tool, we determined that the USB ports on the set-top boxes are active, despite there being no high-level application that uses USB peripherals. That is, the Linux kernel is able to both enumerate and load the appropriate driver for a number of USB classes once a device is plugged in, including keyboards, mass-storage devices, and so forth.

Technical Description Listing 7.1 contains the output of a USB assessment tool called `umap` [13]. By emulating device responses of various USB peripheral classes, `umap` can discover whether a given peripheral is supported. Emulation is performed with the aid of the Facedancer, and responses are crafted with different *device descriptors* (as discussed in §4.4.4) to emulate different USB peripherals.

Both set-top boxes were found to support the same peripherals. The `umap` Listing 7.1 confirms that the set-top boxes support the USB Audio, HID, Mass Storage, and Hub classes.

Listing 7.1: umap output of set-top box peripheral support

```
01:01:00 - Audio : Audio control : PR Protocol undefined
 **SUPPORTED**
01:02:00 - Audio : Audio streaming : PR Protocol undefined
 **SUPPORTED**
```

```
02:02:01 – CDC Control : Abstract Control Model : AT commands V.250

02:03:ff – CDC Control : Telephone Control Model : Vendor specific

02:06:00 – CDC Control : Ethernet Networking Control Model : No class–specific
          protocol required

03:00:00 – Human Interface Device : No subclass : None
 **SUPPORTED**
06:01:01 – Image : Still image capture device : Bulk–only protocol

07:01:02 – Printer : Default : Bidirectional interface

08:06:50 – Mass Storage : SCSI : BBB
 **SUPPORTED**

09:00:00 – Hub : Default : Default
 **SUPPORTED**
0a:00:00 – CDC Data : Default : Default

0b:00:00 – Smart Card : Default : Default
```

Analysis Support for a USB peripheral class indicates that the host is able to load a valid driver for the peripheral. Like other software, Linux USB device drivers contain bugs. For instance, `CVE-2013-2888` [14] is a vulnerability afflicting the core HID driver of Linux systems up to version 3.11, which can allow an attacker to execute arbitrary code on a host. The capability of an attacker to execute arbitrary code on a proprietary device, such as a set-top box, presents a severe threat. Moreover, the number of discovered bugs in Linux device drivers has increased dramatically as of 2013, as can be viewed in Appendix C.2. With the likelihood of more bugs being discovered, the challenge of updating driver software is analogous to that of Case Study 5.

It is not the intention of the manufacturers to include the support of these drivers—they were simply present when the Linux kernel was built. Each additional driver has the potential to introduce multiple vulnerabilities on the device, which, like `CVE-2013-2888`, can violate any of the CIA properties. Risk is further increased due to drivers running as root within the kernel, meaning that device compromise through a driver bug will give the attacker complete control in the root context. For this reason, the USB attack vector has historically been used to obtain escalated privileges in devices such as smartphones and consoles, in a process also commonly known as jailbreaking.

Summary The threat posed by default drivers can be significantly reduced at the design phase of embedded devices. Developers and manufacturers need to question whether it is sensible to include Linux drivers for peripherals such as keyboards, mass-storage sticks, and microphones. In an embedded setting, it is unlikely that a device must support all of these functions, if indeed any at all. Thus, incorporating attack surface reduction with an understanding of the device requirements is effective in reducing the risk considered in this case study. Both of these actions are contained in the SDL and CLASP processes requirement, with the intention of preventing an attacker from exploiting potentially insecure code.

7.4 Operational Vulnerabilities

Operational vulnerabilities, as described before, are those that result from some kind of configuration or error in set up. Here we consider an operational vulnerability in the form of a *race condition*, specifically, a time-of-check-to-time-of-use (TOCTTOU) bug. Race conditions affect a broad variety of software, including operating system kernels and web applications.

The time-of-check-to-time-of-use (TOCTTOU) software bug is one caused by a state change in a property between the time that it is *checked* and the time that it is *used* by a program. For example, the "check" portion during normal operation may entail verifying that software is cryptographically signed, and afterward is installed, or "used". This case study draws on Mulliner's discovery of a TOCTTOU vulnerability in the domain of embedded devices [48]. We proceed beyond Mulliner's contribution of the exploitation details, and consider the implications of that this attack has in terms of secure software processes.

7.4.1 Case Study 10: USB TOCTTOU Attack

Overview The TOCTTOU attack in question makes use of USB mass-storage emulation, and was performed on a Linux-based Samsung smart TV. Termed the "Read It Twice" attack [48], the attacker relies on a USB host device which first reads a filesystem in order to verify a software package and then subsequently installs the software package. It becomes possible to exploit this behavior if the host software assumes that the contents of the connected mass-storage device does not change between the *check* and *install* operation. By coercing the smart TV to read the software package twice, the attacker gains the opportunity to change the software package after the validation check, but before installation.

Technical Description The smart TV supports running custom user application software, supplied via USB mass-storage. User-supplied applications run with restricted privileges in Adobe flash file format. However, the TV also supports running games that make use of shared libraries native to the Linux operating system. Games run with root privileges, and can interact directly with the Linux system—however, the TV does not support users to supply application software in the "game" category.

The "Read It Twice" attack succeeds in bypassing this check, and allows a user-supplied application to run as a "game" in root context. This is achieved by first allowing the TV to read a benign software package on the USB device.

The TV verifies that the software package is safe to install, and proceeds with installation. Critically, the TV performs a second read from the USB filesystem when it is ready to install the package, and not from its own memory. At this point, the attacker replaces the application binary contents by switching the USB filesystem in order to classify the application as belonging to the "game" category. The TV subsequently installs the modified binary as a "game" binary and not an Adobe flash file. When the user selects and runs the application on the TV, the application is executed. A possible payload may be to start the `telnet` daemon on the TV, which would then run with root privileges:

```
int Game_Main(char *path, char *udn)
{
  system("telnetd &");
  return 0;
}
```

Analysis One reason that this attack succeeds is due to the TV reading the application from the USB filesystem into memory twice. This behaviour occurs when the application size is larger than the TV's memory, which stores a cache of the USB filesystem contents. Because the TV has limited memory resources, an application size of 260MB forces the TV to reread storage blocks of the application from the USB filesystem after the check. The second reason that the attack succeeds is due to the attacker being able to replace the contents of these storage blocks by using an emulated USB mass-storage device and filesystem. That is, the attacker can detect, using an emulated USB device, when the TV requests the application storage blocks again, and then returns the modified binary.

This vulnerability can yield the attacker full control over the device, and may also give them the ability to modify the firmware of the TV. The integrity property of the TV is first violated, which can allow the attacker to exfiltrate sensitive data, violating the confidentiality property. The attacker could also turn off or destroy the contents of the TV, violating the availability property. However, such a threat is of little consequence given the other capabilities that the attacker can acquire. Addressing the threat of this vulnerability requires an understanding of how reading data from external devices can be exploited in a TOCTTOU attack.

Summary This TOCTTOU operational vulnerability falls under the "race condition" category of NIST's vulnerability classifications. It is made possible due to the interaction between the TV's checking software, the operating system's caching mechanism, and the attached peripheral USB device. Generally, TOCTTOU attacks are prevented by performing an atomic operation on critical data. In this case, it is important for the TV to both check and install the same application in a single operation. One way of ensuring this is to have the TV first copy the entire application binary to persistent storage, such as a harddrive on the TV. Naturally, this may imply additional or larger storage devices, thereby increasing cost of manufacturing.

This vulnerability is challenging to anticipate during design and testing of a device in a secure development process. For successful identification of the vulnerability, developers need to consider how much to trust external devices such as USB peripherals. Any data that originates from a USB peripheral should be treated with care; it is not enough to assume, for instance, that information from the same location on the filesystem remains the same between consecutive reads. With case-studies such as these, developers are in a position to identify this potential vulnerability in the USB attack surface, an activity put forth by CLASP. Further analysis and review of the attack surface according to SDL Phase 6 also guards against the vulnerability's presence at production.

7.5 Implementation Vulnerabilities

Implementation vulnerabilities due to memory corruption bugs are commonly found in USB drivers and operating system software. While not all USB bugs classify as exploitable vulnerabilities, the presence of such bugs suggest less-than thorough testing of software which may harbour security-relevant bugs. These bugs can constitute a real threat, which can result in privilege escalation and arbitrary code execution when exploited.

Our case studies demonstrate memory corruption vulnerabilities in the form of buffer overflows that violate security properties of embedded Linux devices. The first case study relates to third-party USB drivers, whereas the second relates to the core HID driver of Linux. These case studies demonstrate how an attacker can leverage USB bugs to compromise a device, and how this capability should influence the secure development process.

7.5.1 Case Study 11: USB String Descriptor Buffer Overflow

Overview The most common memory corruption vulnerability afflicting USB drivers is the buffer overflow. These have been successfully triggered and exploited with the string descriptor described in §4.4.4. We consider an instance of this vulnerability, exhibited in the third-party Linux driver of the Auerswald PBX/System Telephone product, which resulted in `CVE-2009-4067` [61].

Technical Description The flaw is due to a buffer allocation of only 100 bytes for the device name obtained from a string descriptor, whereas the USB specification allows string descriptors of lengths up to 255 bytes. Consider line 15 in the source listing of the driver 7.2 containing the bug. The value of

`AUSI_DLEN` is 100, and by supplying a device descriptor string larger than this value, an attacker is able to write arbitrary data to the `dev_desc` buffer. This will overwrite elements of the structure represented by the buffer, and can lead to arbitrary code execution.

Listing 7.2: /drivers/usb/misc/auerswald.c

```
/* Try to get a suitable textual description of the device */
/* Device name:*/
ret = usb_string( cp->usbdev, AUSI_DEVICE, cp->dev_desc, AUSI_DLEN-1);
  if (ret >= 0) {
    u += ret;
    /* Append Serial Number */
    memcpy(&cp->dev_desc[u], ",Ser# ", 6);
    u += 6;
    ret = usb_string( cp->usbdev, AUSI_SERIALNR, &cp->dev_desc[u],
        AUSI_DLEN-u-1);
    if (ret >= 0) {
        u += ret;
        /* Append subscriber number */
        memcpy(&cp->dev_desc[u], ", ", 2);
        u += 2;
        ret = usb_string( cp->usbdev, AUSI_MSN, &cp->dev_desc[u],
            AUSI_DLEN-u-1);
        if (ret >= 0) {
          u += ret;
        }
    }
  }
```

Analysis The Auerswald driver runs in the root context of the Linux operating system, and could allow an attacker to execute arbitrary code in this context. While this is a general threat for vulnerable versions of the Linux kernel in desktop environments, it is an even greater threat for proprietary embedded devices. As pointed out in §7.3.1, this type of vulnerability gives opportunity for the attacker to compromise the device through the USB attack surface.

The buffer overflow is a vulnerability class outlined by NIST which testers should be aware of. An attack launched against this driver has the potential to undermine the integrity property of the device by granting a way for the attacker to access the device. After that point, an attacker can continue to violate the confidentiality and availability properties of the device, if desired.

Summary Due to the prevalence of buffer overflows in software, many tools and techniques exist to detect them. The buffer overflow features in CLASP's vulnerability categories being classified as a "Range Error". In the SDL, buffer overflows can be prevented during each of the Phases 8 through 12 when performing software security testing.

This case study shows how portions of USB code can introduce vulnerabilities that an attacker can use to compromise the device. Review, testing, and updates of USB code should therefore be considered during a secure development process of a device with USB functionality.

7.5.2 Case Study 12: Kernel Memory Corruption with USB

Overview Where Case Study 11 involves a third-party USB driver, it is also possible for vulnerabilities to occur in the core USB drivers of Linux. Core USB drivers serve an important role in allowing custom drivers to interface with the kernel. It is therefore a critical component to the USB driver stack, and poses a significant threat should it contain a vulnerability.

Here we consider `CVE-2013-2888`, which affects the core HID driver of the Linux kernel. We form our analysis around an error in the parsing of USB data, resulting in a memory corruption vulnerability.

Technical Description Recall from §4.4.4 that USB hosts rely on a device's report descriptor in order to interpret subsequent input (or reports) such as a key press from the device. Reports can have Global items that define them, including a `Report ID` [17]. When specifying the Global item of length 8 bits, the two least significant bits are reserved for the number of bytes that follow the item type, and the remaining six bits are the item identifier. The bits identifying a `Report ID` is `100001nn`, where nn reserves the number of bytes that follow. For example, if we wanted to assign a report ID of `1` for an input report, the report ID would become `100001`<u>`01`</u>, followed by the report ID we would like to assign, `00000001`. Now, even though the specification states that a report ID may assume a value of 1-255 (expressed as one unsigned byte), it is possible to specify that two bytes follow the `Report ID` item type, e.g. `100001`<u>`10`</u>, since two bits are allocated for this purpose. In the case of `CVE-2013-2888`, the Linux HID report parser incorrectly honors a report ID item that indicates a report ID of size 2 bytes. Despite this, the buffer containing report IDs accommodates only 256 possible values, which is correct according to the USB specification.

Listing 7.3: /include/linux/hid.h

```c
struct hid_report_enum {
 unsigned numbered;
 struct list_head report_list;
 struct hid_report *report_id_hash[256];
};
```

When a 2 byte Report ID is specified, with length greater than 256, a heap buffer overflow occurs in the `report_id_hash` buffer in Listing 7.3. Kernel memory is subsequently overwritten, which is severe enough to give the attacker the ability to execute arbitrary code in kernel (root) context.

Analysis A vulnerability such as `CVE-2013-2888` that allows kernel memory corruption directly poses a threat to the integrity and availability properties of a device. When exploited naively, an attacker is able to disrupt the availability of the device by inducing a crash through kernel memory corruption. Exploiting the vulnerability in a manner that diverts execution to attacker-supplied code can compromise the device completely. For example, the attacker may supply a report ID value which is in fact a valid kernel memory address.

Heap buffer overflows occur due to insufficient range checking, as categorized by CLASP. Again, the buffer overflow is the most prevalent error as classified by NIST. It is often challenging to predict the ease with which exploitation can occur given a buffer overflow. However, the risk posed by vulnerabilities such as `CVE-2013-2888` is regarded to be of such a severe nature that it is patched in the kernel, regardless of whether it has been successfully exploited. Likewise, secure development processes should not rely on the existence of an exploit before addressing memory corruption vulnerabilities.

Summary Secure development processes need to be able to guard against USB implementation vulnerabilities. Automated analysis techniques as put forth by SDL, as well as manual code inspection found in CLASP, can help alleviate this task. However it is difficult to rely on these methods for finding all the bugs in USB code; this is particularly true of vulnerabilities that are more subtle, such as `CVE-2013-2888`. As mentioned in Case Study 9, one of the most effective ways to guard against this implementation vulnerability may be to exclude the HID driver completely. Alternatively, mechanisms need to exist in order to promptly update the driver when a vulnerability is discovered—a challenging task given the nature of embedded devices, and in light of Case Study 5. Finally, USB implementation vulnerabilities reveal that buffer overflow bugs are still extant in critical pieces of software, which should be given due attention during testing.

7.6 Chapter Summary

As demonstrated, the maturity of tools in the realm of USB security exposes further software attack vectors in embedded devices. With that, we introduced the TTWE framework for fuzzing USB communication and found it to be effective at easing the task of the attacker to both find and exploit vulnerabilities by USB emulation. By profiling the USB driver support of embedded devices, we showed with Case Study 9 that the USB attack surface can be unnecessarily large, thereby making a device vulnerable to implementation bugs in third-party and core Linux drivers. It is also possible for complex attacks to arise with USB functionality, due to embedded hardware memory restrictions and peripheral devices in Case Study 10.

With the expectation that the discovery of USB bugs will increase, such attacks consequently imply additional incorporation of considerations during the software development lifecycle from Chapter 3. Additional considerations primarily include the removal of unnecessary software during device design, and thorough testing of USB drivers. These actions correspond to attack surface analysis and secure software testing of SDL and CLASP.

Chapter 8

Secure Development Recommendations

8.1 Overview

In this chapter we deliver on the objective to provide recommendations for securing software in embedded Linux devices. These recommendations pertain to the attack surfaces of the system, network, and USB portions of an embedded device. We substantiate each recommendation with a combination of the case studies in prior chapters. Not only does this enforce the importance of each recommendation, but it also reveals how the case studies relate to one another across vulnerability classes, attack surfaces, and the threats that they present. Moreover, this categorization allows us to directly correlate each recommendation with associated methods found in SDL and CLASP.

Each recommendation is accompanied by a visual depiction of the case studies that relate to it, according to Figure 8.1. This base figure is shown as three coloured circles containing the case study numbers, grouped by attack surface. Case studies are positioned radially from the center of the base figure, according to the three vulnerability classes (design, operational, and implementation). During software development, we expect secure methods to progress from design decisions, down to implementation testing—an equivalent progression is seen while moving from the outer radius to the inner. The recommendations are discussed roughly according to this progression. By filling the case study circles that relate to each recommendation, this diagrammatic representation serves as an elegant way to summarize the results of earlier chapters.

For reference, each case study number and name are supplied in Table 8.1.

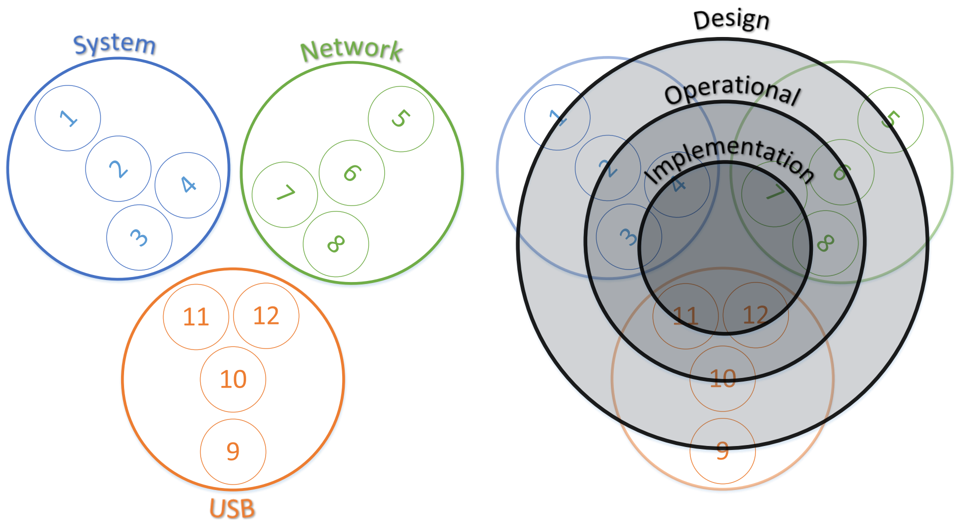

Figure 8.1: Case Study Categorization Diagram

1. Address Space Layout Randomization
2. Service Misconfiguration
3. Escaping Sandboxes
4. System Command Injection
5. Outdated Network Software
6. Implicit Network Server Trust
7. Command Injection with DHCP
8. Cross-site Scripting via Telnet
9. Default USB Driver Support
10. USB TOCTTOU Attack
11. USB String Descriptor Buffer Overflow
12. Kernel Memory Corruption with USB

Table 8.1: Case Study Reference

8.2 Recommendations

An overview of our recommendations are listed below, followed by a detailed motivation for each.

1. Enable Secure Options and Hardening

2. Omit Needless Software

3. Establish Software Maintenance Practices

4. Distrust the Operational Environment
5. Restrict Permissions of Execution Contexts
6. Perform Input Sanitization

1: Enable Secure Options and Hardening

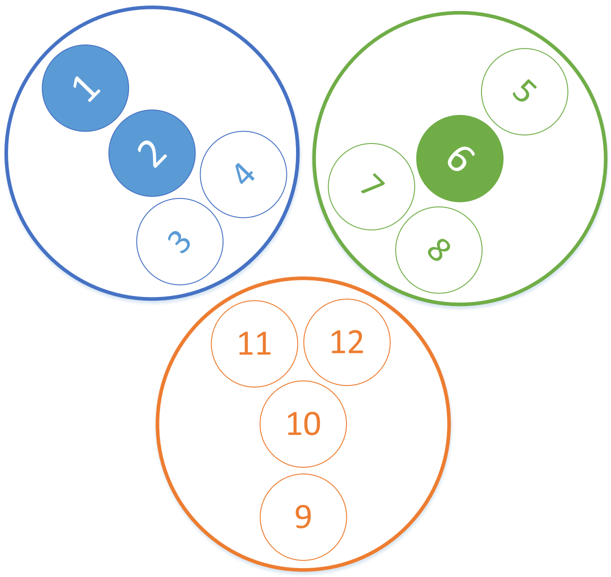

Figure 8.2: Recommendation 1 Case Study Diagram

Case Studies 1, 2, and 6 attest to the importance of enabling options available to the developer for improving device security. This may be an easy exercise, such as enabling a built-in option of the `ps` command of Case Study 2. On the other hand, enabling options (such as ASLR) that influence performance may require more careful consideration. On all accounts, these preventative measures can mitigate the ability of the attacker to violate the integrity and confidentiality properties of a device. Hardening techniques such as ASLR and enabling non-executable memory are effective in combating exploit techniques.

We note from Figure 8.2 that these options affect design and operational vulnerabilities in the system and network attack surfaces. ASLR support for Linux *kernel* memory is not yet fully supported, and hence this recommendation is not readily applicable to software which run in the kernel, such as most USB drivers. Furthermore, this recommendation addresses issues in the operational and design vulnerability classifications. Thus, the secure development process should evaluate the applicability of software options during early stages of design, as well as during later stages of implementation. We advocate that as the impact of vulnerabilities on embedded systems become greater, previous attitudes of software security options need to transition from being "optional" (according to Yaghmour [66]) to becoming "strongly recommended".

Developers and manufacturers need to consider the risk and likelihood of an attacker taking advantage of an unhardened device. Because hardening options can have an impact on device performance and function, we suggest that this recommendation be applied during the design of a device, and to specifically consider threat scenarios as put forth in SDL Phase 7. This coincides with CLASP's activities of defining security and system requirements.

2: Omit Needless Software

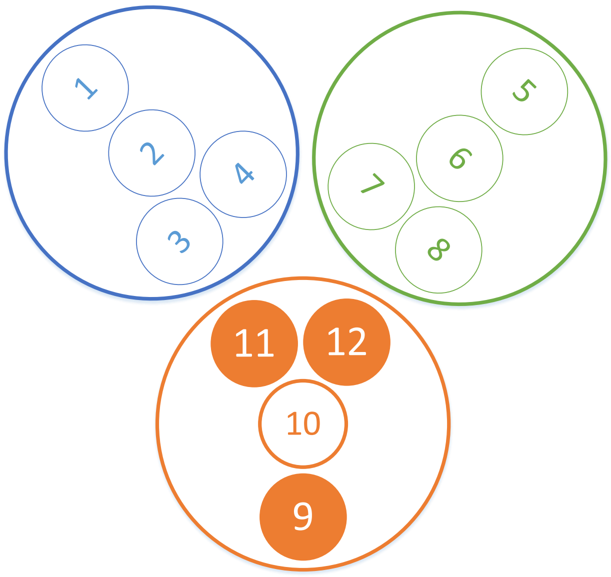

Figure 8.3: Recommendation 2 Case Study Diagram

As reflected by Figure 8.3, the USB attack surface best exemplifies the reasoning behind omitting needless software. Case Study 9 is central to the recommendation of omitting USB drivers that do not contribute to a device's core functionality. In our analysis of this problem, we note that most USB drivers operate in the root context of a device. This leaves the device vulnerable to attacks over the USB channel which can allow an attacker complete control. This primarily affects the integrity property of a device, which can lead to further security property violations. Implementation vulnerabilities in USB software also demonstrate how the availability property of a device can be affected through memory corruption. While we considered that threats on the USB attack surface are mitigated by the challenge of writing exploits for embedded architectures, it is an unnecessary risk to take.

The challenge of omitting needless software arises when manufacturers are unsure of the functionality that a device should have throughout its entire lifetime. For example, it may happen that a set-top box should eventually support mass-storage devices for video playback. On the other hand, it is unlikely that a set-top box should ever support a USB printer. By considering potential device support, manufacturers can reliably rule out unnecessary drivers and significantly reduce the USB attack surface.

This recommendation serves to guard a device from having an unnecessarily large attack surface, particularly due to USB drivers. We therefore suggest that this recommendation is best applied during attack surface analysis of the design. Recall from Chapter 3 that attack surface analysis is an action performed in both SDL Phase 6 and in CLASP. It would also prove effective to confirm removal of omitted software during attack surface review (SDL Phase 13).

3: Establish Software Maintenance Practices

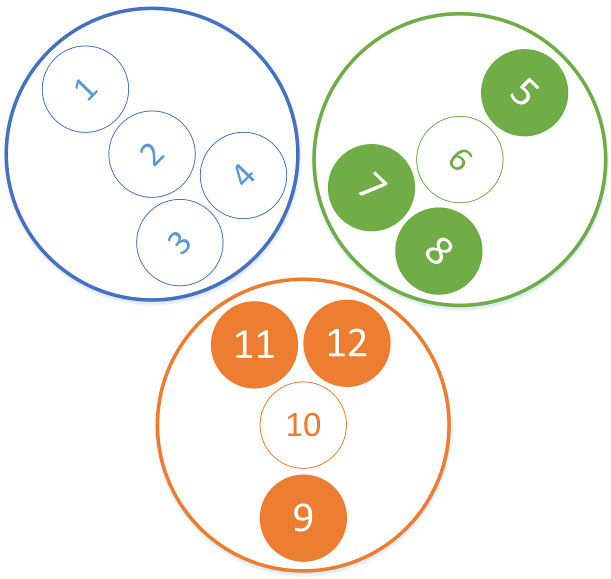

Figure 8.4: Recommendation 3 Case Study Diagram

The need for maintaining software in embedded devices is demonstrated by case studies 5, 7, 8, 9, 11, and 12. Although there are benefits for keeping all portions of software updated, it is most critical for software which is attacker-facing. Therefore, this recommendation relies on vulnerabilities in the network and USB attack surfaces of Figure 8.4. It comes with the supposition that the attacker wishes to gain initial entry into a device, whether remotely over a network or via USB. The threat that this attack poses to device manufacturers are exemplified by proprietary set-top boxes, smart TVs, and mobile devices.

The nature of keeping software updated on embedded devices, as per Case Study 5, presents a great challenge. Due to this, our recommendation does not mandate that every device support dynamic updates of software, although such a solution is certainly preferable. Rather, we advocate that developers consider the repercussions of omitting an update mechanism, and evaluate the associated risk. For example, the XSS attack in Case Study 8 could be used to obtain login credentials of the router. However, it may be determined that the associated software carries a risk that could be addressed by other means, for example monitoring network activity associated with such an attack. On the other hand, if the device relies heavily on USB software which may contain a pervasive bug, such as that of Case Study 12, it may be appropriate to implement a software update mechanism.

In summary, we stress the importance of manufacturers to realize an update mechanism when required, or to firmly establish that it is not necessary. This decision should ideally be made during the design of a device, but it is likely to impact the implementation phase. Moreover, should software updating be included, the mechanism itself will need to be tested. Thus, we suggest that threat modelling (from SDL Phase 7 and CLASP) accompany the decision to include or omit an update mechanism. In either case, we also suggest software security testing according to the SDL and CLASP. Thorough security testing will provide additional assurance in the event that software will not be updated during the device lifetime. If an update mechanism is provided, software relating to it should necessarily undergo security testing as well.

4: Distrust the Operational Environment

From Figure 8.5, we observe that distrusting the operational environment rests on the respective operational vulnerabilities of each attack surface category, as well as the two implementation vulnerabilities of Case Study 7 and 12. The implementation vulnerabilities of this recommendation contribute examples where software functionality is built into the device, and cannot be removed. Yet, some of these inclusions, such as the DHCP and USB support, require extra operational consideration.

These case studies revealed how the complex interaction between filesystems, system services, network services, and peripheral devices cause vulnerabilities in a device. Operational vulnerabilities are particularly difficult to detect, and likely constitute the most *unanticipated* attacks that developers need to consider. The attack scenario may violate any of the three CIA properties, depending on the vulnerability, and constitute threats of varying levels. For example, the attacker might reveal passwords (Case Study 2), impersonate trusted network resources (Case Study 6), or compromise the device with a peripheral (Case Study 10).

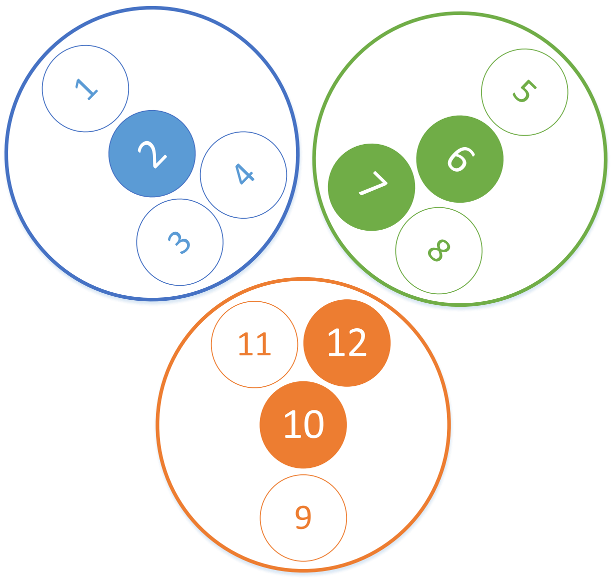

Figure 8.5: Recommendation 4 Case Study Diagram

To mitigate the likelihood of operational vulnerabilities occurring, one needs to assume that the device operates in hostile environment. With this attacker-like mindset, it becomes more conceivable to anticipate attacks. For example, consider the options that an attacker has at his disposal when attacking a device. He might ask "Which built-in command-line options, network service software, or peripheral support is available for attack on the device?" We suggest that this question needs to be asked, during threat modelling practices of SDL and CLASP. Addressing this question, however, will also necessitate security testing in accordance to SDL and CLASP. For example, it might be anticipated that peripherals can affect the security properties of a device, but such a suspicion may require specific testing as presented in Case Studies 6 and 10.

5: Restrict Permissions of Execution Contexts

An underlying threat of case studies 1, 3, 4, 7, and 9 is the danger of a program operating in privileged mode, such as root. As shown in Figure 8.6, elements of design and implementation vulnerabilities contribute to this threat when the objective of an attacker is to achieve privileged access to a device. Such an attack scenario is purposed to compromise the integrity of the device; the attacker is not interested in divulging confidential information or limiting service availability in the context of privilege escalation attacks.

Vulnerabilities due to design decisions should be addressed during earlier stages of software development. Hardening techniques discussed in Case Study 1 is one way to weaken the ability of the attacker to gain privileged access by

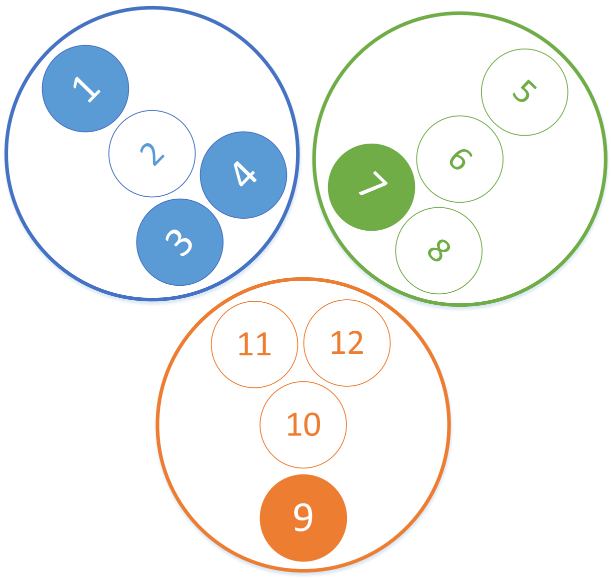

Figure 8.6: Recommendation 5 Case Study Diagram

mitigating exploitation. However, hardening is of particular importance for software that by necessity runs in privileged mode, such as drivers. In terms of attack surface reduction, it is important for developers to question whether custom drivers require running as root; if not, it should be implemented to run with restricted access to filesystem resources and memory. For instance, Linux printer drivers do not require root permissions to function, unlike USB drivers. Beyond drivers, network services may require running as root, but can place a logged-in user in a restricted execution environment. Similar methods of restricting permissions is the act of sandboxing the execution of a program. While this method can prove effective, developers should evaluate the accompanying risk in case the sandbox can be escaped, as observed in Case Study 3.

Evaluation of sandboxing techniques are difficult to anticipate during design, and must be tested toward the end of the development lifecycle. Attack surface analysis, threat modelling, and security testing are methods put forth by SDL and CLASP which are all applicable for achieving this recommendation. Attack surface analysis is effective if programs with privileged execution environments can be removed or isolated from an attacker. If this is not possible, threat modelling assists in the likelihood that an attacker will be able to exploit a vulnerability in such a program. Finally, security testing of programs that harbour a significant threat will provide manufacturers with greater assurance.

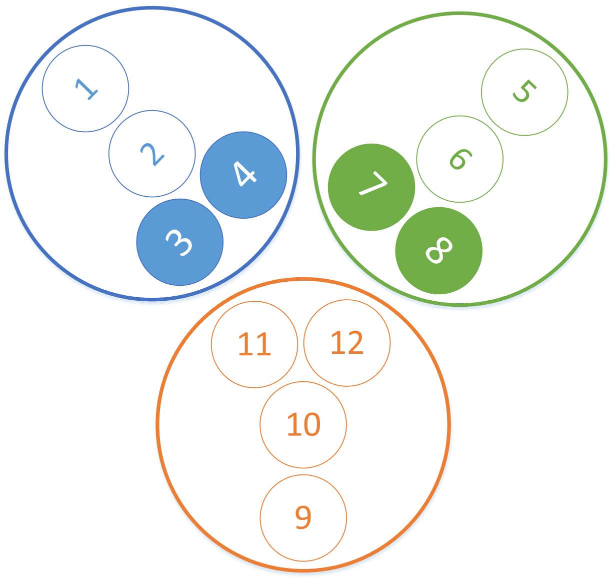

Figure 8.7: Recommendation 6 Case Study Diagram

6: Perform Input Sanitization

Lack of input sanitization is a leading cause of implementation vulnerabilities, as revealed by case studies 3, 4, 7, and 8. Figure 8.7 illustrates that the network and system attack surface are principally susceptible to these vulnerabilities. An important consequence of this point is that an attacker can use bugs resulting from lack of input sanitization to gain remote access to a device, and thereafter obtain escalated privileges on the system. Such a scenario echoes the analyses covered previously, emphasizing that attacks lead primarily to a violation of the integrity property, and secondarily to the confidentiality property.

Afflicted software of the system included programs written in C, bash, and Javascript, implying that developers can't rely on pinning these vulnerabilities to a specific implementation language. Mitigation of these vulnerabilities entail input sanitization, the application of which will vary from device to device. Appropriate strategies include whitelisting a set of commands, or blacklisting meaningful characters depending on the context. Regardless, the important aspect of this recommendation is that the secure development process should identify and test areas that require input sanitization.

This recommendation relies most heavily on security testing of device software during and after implementation, as set forth in Chapter 3. Software can include Linux binaries, BusyBox, network services, third-party drivers, and third-party applications. Applicable methods advocated by SDL and CLASP are listed in Table 3.1, and will depend on the properties of the software. During our case studies we found that manual code review was most effective for software security testing when the source code was available and under 100 lines of code. Furthermore, explicit testing of software with unsanitized values through fuzzing can specifically detect input sanitization bugs.

8.3 Evaluation

The recommendations in this chapter are given on the basis of our 12 case studies, covering design, operational, and implementation vulnerabilities over the system, network, and USB attack surfaces. With each recommendation, we infer appropriate methods from the SDL and CLASP, including attack surface analysis, threat modelling, and security testing. The advantage of this approach is that it aids parties involved during software development of *embedded devices* to be security-conscious. This is achieved by delivering concrete examples of vulnerabilities and ways to mitigate common flaws and bugs using recognized methods.

Given this approach and a finite number of case studies, we are limited in inferring a non-exhaustive list of recommendations. It is indeed possible that embedded devices may benefit from additional practices that are not included in our recommendations. Furthermore, there may be instances where specific devices, such as smartphones, may benefit from security considerations not applicable to other devices, such as smart TVs. In this regard, our recommendations are expected to be applicable to all Linux based devices, and applicable to devices which operate under other embedded operating systems as well.

Chapter 9

Conclusion

9.1 Objectives

In this thesis, we identified the growing need for securing embedded devices, particularly in Linux based devices. In addressing this need, we assert that security vulnerabilities commonly result from negligent practices during software development of such devices. To this end, we deliver concrete recommendations for addressing classes of vulnerabilities by way of 12 case studies. With respect to the objectives established in Chapter 1, we achieved the following:

- We derived appropriate techniques, namely, attack surface analysis, threat modeling, and security testing methods from the SDL and CLASP development processes in Chapter 3. These methods were found to be applicable to embedded Linux devices, and were used to evaluate each case study in Chapters 5 through 7.
- The vulnerability assessment techniques were applied over the system, network, and USB attack surfaces through a number of case studies, in Chapters 5, 6, and 7 respectively.
- Vulnerabilities were categorized according to three classes, namely, design, operational, and implementation vulnerabilities. For each of the 12 case studies in Chapters 5 through 7, an overview describes the process of vulnerability discovery, followed by a detailed analysis of risk in terms of the three CIA properties.
- Six actionable recommendations were delivered in Chapter 8 on the basis of the case study analyses.

9.2 Contributions

Consider the contributions of this thesis, bulleted below, as introduced in Chapter 1.

- A case study for applying secure software development processes to embedded Linux devices

In Chapter 2 we observed the absence of software development processes that exist for embedded systems. Although many assessment methods exist, it is not clear how these may be applied to embedded devices, and whether they are applicable. In our work, we demonstrated the methods of *attack surface analysis*, *threat modeling*, and *security testing* in the context of embedded devices. We derived these methods from the SDL and CLASP development processes on the basis of our comparison in Chapter 3. This work therefore spans both the practices underpinning secure development, and the application thereof, making for a significant contribution.

- The discovery of new vulnerabilities on embedded Linux devices

This contribution refers to `CVE-2014-0337`, `CVE-2013-3594`, `CVE-2013-3595`, `CVE-2013-3606`, and Case Study 1-6, 8, and 9. Not only does this contribution emphasize the effectiveness of applying the aforementioned methods, but it also exposes the developer to common classes of mistakes with *concrete* examples.

- The Transparent Two-Way Emulation framework, a novel testing framework for the USB attack surface

In Chapter 7 we developed the TTWE framework [60], and demonstrated its advantages for discovering USB bugs. Our framework improves upon current testing methods by affording greater flexibility and affordability. Moreover, the development of improved USB testing tools highlight just how critically device drivers should be assessed in devices.

- Concrete recommendations based on our case studies, in conjunction with assessment methods.

We briefly compare and evaluate our recommendations to that of McGraw, who formulates a taxonomy of seven security errors, based on the principle that "people are good at keeping track of seven things, plus or minus two" [58].

McGraw's seven points make for appropriate candidates to compare to our six recommendations on the basis of advocating secure practices while remaining concise. In comparison to McGraw's assertions of security errors, we found that there is significant overlap with our recommendations and case studies. We summarize McGraw's assertions below (in bold), and indicate which case studies and recommendations they relate to, where applicable.

1. **Input validation and representation**, including input sanitization, relating to Recommendation #6: Perform Input Sanitization.

2. **API abuse**, such as expecting trustworthy DNS information, relating to Recommendation #4: Distrust the Operational Environment.
3. **Security features**, including privilege management and hardening, relating to Recommendation #1: Enable Secure Options and Hardening.
4. **Time and state**, including race conditions present in program execution and filesystems, relating to Case Study 10: TOCTTOU attacks.
5. **Errors that disclose information to the attacker**, relating to Case Study 2: Information disclosure.
6. **Code quality**, where poor code quality leads to buggy behavior, relating to Recommendation #3: Establish Software Maintenance Practices.
7. **Encapsulation**, which entails setting up boundaries between programs, relating to Recommendation #5: Restrict Permissions of Execution Contexts.

This result is satisfying, since McGraw's points were not considered at the outset of this study. Even so, our recommendations can be seen as relating closely to the security errors identified by McGraw. The primary benefit of our results, however, are that they exhibit two properties that surpass those of McGraw, in that they are

- substantiated by concrete case studies, and
- are applicable to the specific domain of embedded devices.

To summarize, this thesis makes significant contributions in the way of demonstrating the translation of software assessment practices to actionable secure recommendations, substantiated by a comprehensive analysis of case studies. Due to this approach, it must be acknowledged that while our results are valuable, they are non-exhaustive.

9.3 Future Work

Due to the non-exhaustive nature of this thesis, additional benefit can be derived by extending the scope of Chapter 1. For one, this thesis only considered Linux-based devices, yet we expect that many of the assessment methods are applicable to embedded devices that run other operating systems. Although an effort was not made to explore this possibility, it would be a worthy exercise to determine the applicability of our methods, USB testing tool, and recommendations for such devices.

The empirical approach of our approach implies that considering additional case studies may contribute to, or even form new recommendations. It is difficult to predict the impact of additional case studies, as these are dependent on the nature of the vulnerability that is considered. We maintain that exploring additional case studies may serve to enhance secure practices. There may, however, be a limit to the usefulness of additional vulnerabilities before the analysis becomes redundant, especially when they belong to the same vulnerability class. Nevertheless, this thesis can be reinforced by additional, well-chosen examples.

Because the goal of achieving secure devices is a moving target, it is to be expected that software assessment methods change and evolve over time. USB attacks are a good example, where the improvement of testing tools (although not fully mature) are only now alerting manufacturers to their alarming effects. Thus, some security properties will become more critical than others as time passes. Though we consider all of our recommendations to be important, there is no rank of importance associated with them. Therefore, systematic prioritization of applied methods and testing would be beneficial and complementary to the notion of secure software development processes.

In Chapter 8, we introduced the TTWE USB testing tool, although it is not without shortcomings. The continuous development and application of the TTWE USB testing tool is of great interest at the time of writing. Addressing the identified shortcomings will afford further state-of-the-art capabilities for assessing the USB attack surface on millions of embedded devices.

Appendices

Appendix A

USB Specification

A.1 Standard USB Requests

Request Number	Request Type
00h	Get_Status
01h	Clear_Feature
02h	Set_Feature
03h	Set_Address
04h	Get_Descriptor
05h	Set_Descriptor
06h	Get_Configuration
07h	Set_Configuration
08h	Get_Interface
09h	Set_Interface
0ah	Synch_Frame

A.2 Standard USB Descriptor Fields

Field Value	Descriptor Type	Required
01h	device	yes
02h	configuration	yes
03h	string	no
04h	interface	yes
05h	endpoint	no, but commonly used
06h	device_qualifier	yes, for full and high speed devices
07h	other_speed_configuration	no
08h	interface_power	no
09h	OTG (on-the-go)	no
0ah	debug	no
0bh	interface_assocation	no

Appendix B

Full Source Code Listings

B.1 UDHCPC configuration script

```
#!/bin/sh
# udhcpc Interface Configuration
# Based on http://lists.debian.org/debian-boot/2002/11/msg00500.html
# udhcpc script edited by Tim Riker <Tim@Rikers.org>

[ -z "$1" ] && echo "Error: should be called from udhcpc" && exit 1

RESOLV_CONF="/etc/resolv.conf"
[ -n "$broadcast" ] && BROADCAST="broadcast $broadcast"
[ -n "$subnet" ] && NETMASK="netmask $subnet"

case "$1" in
        deconfig)
                /sbin/ifconfig $interface 0.0.0.0
                ;;

        renew|bound)
                /sbin/ifconfig $interface $ip $BROADCAST $NETMASK

                if [ -n "$router" ] ; then
                        while route del default gw 0.0.0.0 dev $interface ; do
                                true
                        done

                        for i in $router ; do
                                route add default gw $i dev $interface
                        done
                fi

                echo -n > $RESOLV_CONF
                [ -n "$domain" ] && echo search $domain >> $RESOLV_CONF
                for i in $dns ; do
                        echo nameserver $i >> $RESOLV_CONF
                done
                ;;
esac

exit 0
```

Appendix C

USB bugs

C.1 Mass-storage device functionality by TTWE on a Linux Host

Bold numbers indicate hijacked endpoint numbers.

DIR	EP	DATA (Base 10)
OUT	[0]	[128, 6, 0, 1, 0, 0, 64, 0]
IN	[0]	[18, 1, 0, 2, 0, 0, 0, 64, 143, 5, 135, 99, 2, 1, 1, 2, 3, 1]
OUT	[0]	[0, 5, 25, 0, 0, 0, 0, 0]
OUT	[0]	[128, 6, 0, 1, 0, 0, 18, 0]
IN	[0]	[18, 1, 0, 2, 0, 0, 0, 64, 143, 5, 135, 99, 2, 1, 1, 2, 3, 1]
OUT	[0]	[128, 6, 0, 6, 0, 0, 10, 0]
IN	[0]	[10, 6, 0, 2, 0, 0, 0, 64, 1, 0]
OUT	[0]	[128, 6, 0, 2, 0, 0, 9, 0]
IN	[0]	[9, 2, 32, 0, 1, 1, 0, 128, 100]
OUT	[0]	[128, 6, 0, 2, 0, 0, 32, 0]
IN	[0]	[9, 2, 32, 0, 1, 1, 0, 128, 100, 9, 4, 0, 0, 2, 8, 6, 80, 0, 7, 5, **1**, 2, 64, 0, 0, 7, 5, **130**, 2, 64, 0, 0]
OUT	[0]	[128, 6, 0, 3, 0, 0, 255, 0]
IN	[0]	[4, 3, 9, 4]
OUT	[0]	[128, 6, 2, 3, 9, 4, 255, 0]
IN	[0]	[26, 3, 77, 0, 97, 0, 115, 0, 115, 0, 32, 0, 83, 0, 116, 0, 111, 0, 114, 0, 97, 0, 103, 0, 101, 0]
OUT	[0]	[128, 6, 1, 3, 9, 4, 255, 0]
IN	[0]	[16, 3, 71, 0, 101, 0, 110, 0, 101, 0, 114, 0, 105, 0, 99, 0]
OUT	[0]	[128, 6, 3, 3, 9, 4, 255, 0]
IN	[0]	[18, 3, 49, 0, 57, 0, 54, 0, 50, 0, 51, 0, 55, 0, 51, 0, 54, 0]
OUT	[0]	[0, 9, 1, 0, 0, 0, 0, 0]
OUT	[0]	[161, 254, 0, 0, 0, 0, 1, 0]
IN	[0]	[0]
OUT	[1]	[85, 83, 66, 67, 1, 0, 0, 0, 36, 0, 0, 0, 128, 0, 6, 18, 0, 0, 0, 36, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
IN	[3]	[0, 128, 4, 2, 31, 0, 0, 0, 71, 101, 110, 101, 114, 105, 99, 32, 70, 108, 97, 115, 104, 32, 68, 105, 115, 107, 32, 32, 32, 32, 32, 32, 56, 46, 48, 55]
IN	[3]	[85, 83, 66, 83, 1, 0, 0, 0, 0, 0, 0, 0, 0]
OUT	[1]	[85, 83, 66, 67, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 6, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
IN	[3]	[85, 83, 66, 83, 2, 0, 0, 0, 0, 0, 0, 0, 1]
OUT	[1]	[85, 83, 66, 67, 3, 0, 0, 0, 18, 0, 0, 0, 128, 0, 6, 3, 0, 0, 0, 18, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
...	...	...

C.2 CVEs assigned to Linux USB Drivers

2013	**2012**	**2011**	**2010**	**2009**
CVE-2013-1285	CVE-2012-2693	CVE-2011-0712	CVE-2010-4656	CVE-2009-4067
CVE-2013-1286	CVE-2012-3723	CVE-2011-2295		
CVE-2013-1287				
CVE-2013-2888				
CVE-2013-2889				
CVE-2013-2890				
CVE-2013-2891				
CVE-2013-2892				
CVE-2013-2893				
CVE-2013-2894				
CVE-2013-2895				
CVE-2013-2896				
CVE-2013-2897				
CVE-2013-2898				
CVE-2013-2899				
CVE-2013-3200				

List of References

[1] goodfet.sourceforge.net/hardware/facedancer21/. Accessed: 27/12/2013.

[2] Android Security Overview. https://source.android.com/devices/tech/security/. Accessed: 4/9/2014.

[3] BusyBox Source. lists.busybox.net/pipermail/busybox/2012-August/078309.html. Accessed: 4/9/2014.

[4] Dropbear SSH. https://matt.ucc.asn.au/dropbear/dropbear.html. Accessed: 4/9/2014.

[5] Facedancer Firmware Source Repository. https://goodfet.svn.sourceforge.net/svnroot/goodfet. Accessed 25/05/2014.

[6] int3cc. http://int3.cc/products/facedancer21. Accessed 23/11/2014.

[7] Linux Programmer's Manual CHROOT(2). http://man7.org/linux/man-pages/man2/chroot.2.html. Accessed: 4/9/2014.

[8] Linux USB drivers. https://github.com/torvalds/linux/tree/master/drivers/usb. Accessed: 4/9/2014.

[9] MQP Electronics. http://www.mqp.com/usb500.htm. Accessed 25/05/2014.

[10] RFC 1035, Domain Names - Implementation and Specification. http://tools.ietf.org/html/rfc1035. Accessed: 4/9/2014.

[11] Shellshock Exploit via DHCP. http://blog.trendmicro.com/trendlabs-security-intelligence/bash-bug-saga-continues-shellshock-exploit-via-dhcp/. Accessed 23/11/2014.

[12] TTWE Framework Prototype. https://github.com/rvantonder/ttwe-proto. Accessed 25/08/2014.

[13] The USB host security assessment tool. https://github.com/nccgroup/umap. Accessed: 4/9/2014.

[14] Vulnerability Summary for CVE-2013-2888. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-2888`. Accessed: 4/9/2014.

[15] Vulnerability summary for cve-2014-0337. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0337`. Accessed: 4/9/2014.

[16] Vulnerability Summary for CVE-2014-2716. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-2716`. Accessed: 4/9/2014.

[17] Device class definition for human interface devices (HID). Technical report, USB Implementers Forum, 1996-2001.

[18] Intel White Paper: Rise of the Embedded Internet. Technical report, Intel, 2010.

[19] Tiago Alves and Don Felton. Trustzone: Integrated hardware and software security. *ARM white paper*, 3(4), 2004.

[20] William A Arbaugh, David J Farber, and Jonathan M Smith. A secure and reliable bootstrap architecture. In *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*, pages 65–71. IEEE, 1997.

[21] Jan Axelson. *USB complete: everything you need to develop custom USB peripherals*. Lakeview Research, 2001.

[22] Hristo Bojinov, Dan Boneh, Rich Cannings, and Iliyan Malchev. Address space randomization for mobile devices. In *Proceedings of the fourth ACM conference on Wireless network security*, pages 127–138. ACM, 2011.

[23] Sergey Bratus, Travis Goodspeed, Pete C. Johnson, Sean W. Smith, and Ryan Speers. Perimeter-crossing buses: a new attack surface for embedded systems. In *8th Workshop on Embedded Systems Security (WESS)*, sept. 2013.

[24] John Clark, Sylvain Leblanc, and Scott Knight. Compromise through usb-based hardware trojan horse device. *Future Generation Computer Systems*, 27(5):555–563, 2011.

[25] Ang Cui. Aesop: Ubiquitous embedded security in the post-post-pc threat environment. Technical report, Red Balloon Security, 2014.

[26] Ang Cui, Michael Costello, and Salvatore J Stolfo. When firmware modifications attack: A case study of embedded exploitation. In *Network and Distributed System Security Symposium*, 2013.

[27] Ang Cui and Salvatore J Stolfo. Defending embedded systems with software symbiotes. In *Recent Advances in Intrusion Detection*, pages 358–377. Springer, 2011.

[28] Andy Davis. USB - undermining security barriers. In *Black Hat Briefings*, aug. 2011.

[29] Andy Davis. Fuzzing USB devices using frisbeelite. Technical report, NGS Secure Research, jan. 2012.

[30] Andy Davis. USB driver vulnerabilities whitepaper. Technical report, NCC Group, jan. 2013.

[31] Bart De Win, Riccardo Scandariato, Koen Buyens, Johan Grégoire, and Wouter Joosen. On the secure software development process: CLASP, SDL and Touchpoints compared. *Inf. Softw. Technol.*, 51(7):1152–1171, July 2009.

[32] Jeroen Domburg. `http://spritesmods.com/?art=hddhack`. Accessed: 23/04/2014.

[33] Mark Dowd, John McDonald, and Justin Schuh. *The art of software security assessment: Identifying and preventing software vulnerabilities*. Pearson Education, 2006.

[34] William Enck, Machigar Ongtang, and Patrick McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 235–245. ACM, 2009.

[35] J. Gregoire, K. Buyens, B. De Win, R. Scandariato, and W. Joosen. On the secure software development process: Clasp and sdl compared. In *Software Engineering for Secure Systems, 2007. SESS '07: ICSE Workshops 2007. Third International Workshop on*, pages 1–1, May 2007.

[36] USB Device Working Group. `http://www.usb.org/developers/docs/devclass_docs/`. Accessed: 28/04/2014.

[37] Greg Hoglund and Gary Mcgraw. *Exploiting Software How to Break Code*. Addison-Wesley Professional, 2004.

[38] M. Howard. Building more secure software with improved development processes. *Security Privacy, IEEE*, 2(6):63–65, Nov 2004.

[39] Michael Howard and Steve Lipner. The Security Development Lifecycle. *Change*, 34(May):352, 2006.

[40] MWR InfoSecurity. `https://labs.mwrinfosecurity.com/blog/2011/07/14/usb-fuzzing-for-the-masses/`. Accessed: 28/04/2014.

[41] Tobias Klein. *A bug hunter's diary*. No Starch Press, 2011.

[42] Paul Kocher, Ruby Lee, Gary McGraw, Anand Raghunathan, and Srivaths Moderator-Ravi. Security as a new dimension in embedded system design. In *Proceedings of the 41st annual Design Automation Conference*, pages 753–760. ACM, 2004.

[43] P. Koopman. Embedded system security. *Computer*, 37(7):95–97, July 2004.

[44] David LeBlanc and Ben Howard. *Writing Secure Code*. Pearson Education, 2002.

[45] Gary McGraw. Exploiting Software: How to Break Code. In *Invited Talk, Usenix Security Symposium, San Diego*, 2004.

[46] Gary McGraw and Bruce Potter. Software security testing. *IEEE Security and Privacy*, 2(5):81–85, 2004.

[47] Charlie Miller. Mobile attacks and defense. *Security & Privacy, IEEE*, 9(4):68–70, 2011.

[48] Collin Mulliner and Benjamin Michele. Read it twice! a mass-storage-based TOCTTOU attack. In *6th USENIX Workshop on Offensive Technologies*, aug. 2012.

[49] OWASP. Owasp top 10 - 2013, the ten most critical web application security risks. Technical report, The Open Web Application Security Project, 2013.

[50] Emil Protalinski. Accessed: 4/9/2014.

[51] P. Raghavan, A. Lad, and S. Neelakandan. *Embedded Linux System Design and Development*. Taylor & Francis, 2005.

[52] Srivaths Ravi, Anand Raghunathan, Paul Kocher, and Sunil Hattangady. Security in embedded systems: Design challenges. *ACM Transactions on Embedded Computing Systems (TECS)*, 3(3):461–491, 2004.

[53] Karen Scarfone, Murugiah Souppaya, Amanda Cody, and Angela Orebaugh. Technical guide to information security testing and assessment. *NIST Special Publication*, 800:115, 2008.

[54] Bruce Schneier. `https://www.schneier.com/blog/archives/2014/01/security_risks_9.html`. Accessed: 22/04/2014.

[55] Arvind Seshadri, Adrian Perrig, Leendert Van Doorn, and Pradeep Khosla. Swatt: Software-based attestation for embedded devices. In *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on*, pages 272–282. IEEE, 2004.

[56] Asaf Shabtai, Yuval Fledel, and Yuval Elovici. Securing Android-Powered Mobile Devices Using SELinux. *IEEE Security & Privacy*, 8(3):36–44, 2010.

[57] Ruhma Tahir, Zara Hamid, and Hasan Tahir. Analysis of AutoPlay Feature via the USB Flash Drives. In *Proceedings of the World Congress on Engineering*, volume 1, 2008.

[58] Katrina Tsipenyuk, Brian Chess, and Gary McGraw. Seven pernicious kingdoms: A taxonomy of software security errors. *Security & Privacy, IEEE*, 3(6):81–84, 2005.

[59] A. Ukil, J. Sen, and S. Koilakonda. Embedded security for internet of things. In *Emerging Trends and Applications in Computer Science (NCETACS), 2011 2nd National Conference on*, pages 1–6, March 2011.

[60] Rijnard van Tonder and Herman Engelbrecht. Lowering the USB fuzzing barrier by transparent two-way emulation. In *8th USENIX Workshop on Offensive Technologies*, aug. 2014.

[61] R. D. Vega. Linux kernel USB device driver - buffer overflow. Technical report, MWR InfoSecurity Security, oct 2009.

[62] R. D. Vega. Linux kernel caiaq USB drivers buffer overflow vulnerability. Technical report, MWR InfoSecurity Security, march 2011.

[63] John Viega. Building security requirements with CLASP, 2005.

[64] John Viega and Gary McGraw. *Building secure software: how to avoid security problems the right way.* Pearson Education, 2001.

[65] Joshua Wright. `http://code.google.com/p/zigbee-security/`. Accessed: 27/12/2013.

[66] Karim Yaghmour, Jon Masters, Gilad Ben-Yossef, and Philippe Gerum. *Building Embedded Linux Systems.* O'Reilly Media, 2008.

[67] Junfeng Yang, Ang Cui, John Gallagher, Sal Stolfo, and Simha Sethumadhavan. Concurrency attacks. In *Presented as part of the 4th USENIX Workshop on Hot Topics in Parallelism.* USENIX, 2011.