Security Guidance for Critical Areas of Embedded Computing

Security Working Group

Peer Reviewed Document

January 2016

© 2016 prpl Foundation – All Rights Reserved

All rights reserved. You may download, store, display on your computer, view, print, and link to the prpl Foundation "Security Guidance for Critical Areas of Embedded Computing" at http://prpl.works/security-guidance/, subject to the following: (a) the Report may be used solely for your personal, informational, non-commercial use; (b) the Report may not be modified or altered in any way; (c) the Report may not be redistributed; and (d) the trademark, copyright or other notices may not be removed. You may quote portions of the Report as permitted by the Fair Use provisions of the United States Copyright Act, provided that you attribute the portions to the prpl Foundation "Security Guidance for Critical Areas of Embedded Computing"

Acknowledgements

Art Swift, President, prpl Foundation

Cesare Garlati, Chief Security Strategist, prpl Foundation

David Lau, Chair, Technical Steering Committee

Majid Bemanian, Co-chair, Security Working Group

Mike Borza, Co-chair, Security Working Group

Contributors

Duncan Bees
Pam Brown
Dave Buerger
Sherman Chen
Shoi Egawa
Ayoub Figuigui
Kathy Giori
Aaron Guzman
Fabiano Hesse
Mike Janke
Jesper Jurcenoks
Paul Kearney
Adam Lackorzynski
Larry Lapides
David Lingenfelter
Hauke Mehrtens
Phil Muncaster
Bill Montgomery
Corrado Rocca
Brian Russell
Eric Schultz
Nigel Stanley
Rahim Tafazolli
Srinivas Tatipamula
Ian White
Jasper van Woudenberg

Participating Organizations

ADB SA
Alert Logic
BT Technology
Broadcom Corporation
Cloud Security Alliance
Connect In Private Corp.
GCHQ
Home Gateway Initiative
IBM Corporation
Imagination Technologies
Imperas
Intel Corporation
Kernkonzept
OpenSky
Pontificia Universidade Católica do RS
Qualcomm
Riscure
SELTECH Corporation
Silent Circle
Synopsys Inc.
University of Surrey

© 2016 prpl Foundation – All Rights Reserved.

Contents

Acknowledgements 3

Preface 3

Executive Summary 5

1.0 Introduction to Securing Connected Devices 7

1.1 Threats to Individuals and Organizations...8

1.2 Challenges to Secure Embedded Systems..13

1.3 A New Approach 17

2.0 Fundamental Controls for Securing Devices 22

2.1 Challenges of Securing Devices 22

2.1.1 Inadequacy of Legacy Systems 22

2.1.2 Defining Targets for Protection 22

2.2 Requirement Specifications for Device Security 24

2.2.1 Root of Trust..24

2.2.2 Secure Boot Process...24

2.2.3 Chain of Trust Authentication 26

2.2.4 Hypervisor and Guest Operating System Authentication 27

3.0 Security by Separation..29

3.1 Using Separation by Virtualization to Secure Embedded Systems 29

3.1.1 Types of Attacks Addressed by This Guidance 30

3.1.2 Types of Separation for Embedded Systems 31

3.1.3 Opportunities for Virtualization and Secure Embedded Systems 32

3.2 Challenges to Achieving Separation 32

3.3 Approach #1 – Virtualization 33

3.3.1 Hardware-Assisted Virtualization 34

3.3.2 Paravirtualization...36

3.3.3 Hybrid Virtualization 38

3.3.4 Functional Requirements...38

3.4 Approach #2 - Linux containers 41

4.0 Secure Development and Testing..43

4.0.1 Virtual Platform-Based Development, Debug & Test 43

4.0.2 Joint Task Force Action Group (JTAG) Methodology 43

4.0.3 Extended JTAG for MIPS Reference Architecture...44

4.1 Challenges to Secure Development and Testing 45

4.1.1 Debugging for non-Virtualized Systems 46
4.1.2 Debugging for CPU Hardware Virtualization 47
4.1.3 Debugging for Root Execution Modes 48
4.1.4 Ensuring Proper Authentication for Hardware Debugging 49
4.2 Requirements 51
4.2.1 How It Works 51
4.2.2 Lighter-Weight Alternatives 52
5.0 Use Cases 53
5.1 Protection of Media 53
5.2 Separation of Networks 54
5.3 Separation of Secrets In a Product Supply Chain 54
Appendix 1 - Potential APIs 55
A1.1 API's that Would Be New Due to Virtualization Being Used for Security 55
A1.2 Second Level API's that Depend On Choice of Lower-Level Crypto-Functions 55
A1.3 API's that Exist Already for Secure Systems Without Virtualization 55

© 2016 prpl Foundation – All Rights Reserved.

Preface

Mainstream commentators often describe the Internet of Things (IoT) as if it's a far away vision of our technological future. In eye-catching features they talk of "Minority Report" style customer recognition systems, of subcutaneous health-monitoring chips and driverless cars. But all that headline-grabbing future gazing threatens to overshadow an important point: The Internet of Things is already here.

Gartner estimates there were around 4.9 billion connected "things" in use by the end of 2015, up 30% from 2014.[1] Meanwhile, IDC claimed that the worldwide IoT market was worth $656 billion in 2014.[2] It's already simplifying our supply chains, making our power grids more efficient, improving patient healthcare outcomes and making our homes smarter – not to mention providing us with tiny connected computers to wear on our wrists.

It's not a big stretch of the imagination to think that in a few years' time it will be hard to find a single electrical appliance, consumer technology or enterprise system which doesn't contain an element of "intelligence" and connectivity built in.

But there is a problem. The Internet of Things is also vulnerable to cyber attack.

In many ways, this isn't surprising if we consider that the standard Linux kernel running on many IoT devices now has over 19 million lines of code.[3] Errors are always going to creep in somewhere. Perhaps more troublesome is the proprietary software used by many IoT manufacturers. Here even fewer eyeballs are used to vet code. Yet the wealth of software analysis tools at the disposal of the average security researcher – or cybercriminal – today means "security by obscurity" is no longer an adequate defense. Black Hat 2015 was jam packed with new research into serious IoT vulnerabilities made possible in part thanks to reverse engineering of the underlying software.

Those same researchers have also been able to take advantage of another reality in today's IoT industry: many of the companies making connected systems today find it hard to get in-house developer expertise where it counts. The quality of engineering in the connectivity layer suffers particularly as a result.

IoT systems may fall at the most elementary level because the chip firmware is not cryptographically signed. It's an issue that has gone largely unnoticed until now, but that won't be the case for long. Already it's being exploited in the SYNful Knock attacks on Cisco routers. Lateral movement inside the hardware is another security concern that's largely been unaddressed by the industry.

---

[1] http://www.gartner.com/newsroom/id/2905717

[2] http://www.idc.com/getdoc.jsp?containerId=prUS25658015

[3] http://www.phoronix.com/scan.php?page=news_item&px=Linux-19.5M-Stats

© 2016 prpl Foundation – All Rights Reserved.

At the prpl Foundation it's not our business to be overly alarmist. We'll leave that to the headline writers. But IoT vulnerabilities have a potentially serious real world impact that could dwarf the current spate of data breaches which seem to fill the news each week. We hope our guidance in this document will help stakeholders to better understand these threats and challenges we face in securing IoT systems. It aims to lay out our vision for the journey we must take towards securing the Internet of Things, at a hardware level.

From the railroads to aviation to automobiles, the history of man is littered with industries that had to learn the hard way before taking safety and security seriously. It's up to us to learn from history, and take the essential steps for securing the new frontier of connected systems: the Internet of Things that will eventually touch everyone in our connected world.

Art Swift
President
prpl Foundation

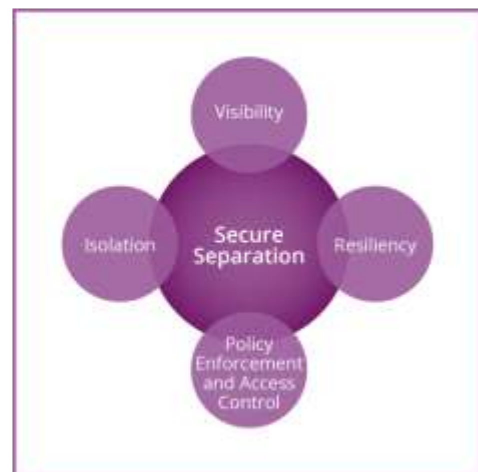
© 2016 prpl Foundation – All Rights Reserved.

Executive Summary

Security is a core requirement for manufacturers, developers, service providers and other stakeholders who produce and use connected devices. Most of these – especially those used on the "Internet of Things" – rely on a complex web of embedded systems. Securing these is a major challenge, and failure to do so can result in significant harm to individuals, businesses and to nations. Guidance in this report provided by prpl Foundation will help stakeholders address the challenges to securing embedded devices.

Our guidance focuses on a new hardware-led approach to create stronger security for embedded systems. We propose three general areas of guidance. These are not the only areas that require attention, but they will help to establish a base of action as stakeholders begin addressing security in earnest. These areas include:

Addressing fundamental controls for securing devices. Stakeholders must address two basic questions: "What are we trying to protect?" and "What is required to enable protection?" The answers result in a shortlist of fundamental controls necessary to implement hardware-based security. The core requirement is a trusted operating environment enabled with a secure boot process that is impervious to attack. This requires a root of trust forged in hardware, which establishes a chain of trust for all subsystems.



Using a Security by Separation approach. Security by Separation is a classic, time-tested approach to protecting computer systems and the data contained therein. Separation means functions cannot see or access other functions without authorization. By separating and restricting the availability and use of assets, security is enforced according to prescribed policy. In this way, software that implements one function does not have to trust software which is implementing another function – each is separated from each other. Our Guidance focuses on embedded systems that can retain their security attributes even when connected to open networks. It hinges on the use of logical separation created by a variety of methods: the most secure is

© 2016 prpl Foundation – All Rights Reserved.

hardware-based virtualization, which entails systems used to simulate, isolate and control IT assets. There are pros and cons to using other separation methods, but they can serve as interim implementations, such as paravirtualization, hybrid virtualization and Linux containers.

Enforcing secure development and testing. Finally, developers must provide an infrastructure that enables secure debug during product development and testing. Normally, hardware debugging through JTAG allows the user to see the entire system. A secure system needs to maintain the separation of assets even when using hardware debugging.

By embracing these initial areas of focus, stakeholders can take action to create secure operating environments in embedded devices by means of secure application programming interfaces (APIs). The APIs will create the glue to enable secure inter-process communications between disparate system-on-chip processors, software and applications. Open, secure APIs thus are at the center of securing newer multi-tenant devices. In this report, prpl Foundation offers guidance defining a framework for creating secure APIs to implement hardware-based security for embedded devices.

© 2016 prpl Foundation – All Rights Reserved.

1.0 Introduction to Securing Connected Devices

The Internet of Things represents nothing short of a revolution in modern computing. Definitions can vary but the International Telecommunication Union describes it thus:[4]

> *"A global infrastructure for the information society, enabling advanced services by interconnecting (physical and virtual) things based on existing and evolving interoperable information and communication technologies."*

The potential of the IoT is only just beginning to be realized. Yet its rate of growth is already staggering. Gartner estimates that 4.9 billion connected things were in use by the end of 2015, a 30% increase from 2014.[5] This will rise to 25 billion by 2020 as consumer-facing applications drive volume growth, while enterprise sales account for the majority of revenue. Gartner isn't the only market watcher predicting great things for this new computing paradigm. Accenture believes the "industrial" IoT alone will contribute $14 trillion to world output by 2030,[6] while ABI Research estimates wireless connected devices will number 40.9 billion globally by 2020.[7]

Whether one believes those estimates or not, the truth is that IoT is already seeping into every aspect of our lives – from healthcare to smart cities, connected cars to in-flight entertainment systems, and heavy manufacturing to national security. It offers the tantalizing prospect of making us more productive at work and happier and healthier at home, and for businesses offers a chance to improve agility, efficiency and quality of service.

In the consumer sphere the Internet of Things has the potential to transform our homes through smart energy meters to manage our utility supply more efficiently; connected home appliances that can order fresh supplies before we run out; and smart entertainment hubs that can be operated remotely for maximum comfort. That's not to mention the multitude of wearable technology products hitting the stores – from smart watches to fitness bands and beyond.

There are various use-cases in different market segments: Consider how the healthcare sector is already benefitting from connected medical devices that monitor the health of patients wherever they are in real-time.

---

[4] http://www.itu.int/ITU-T/recommendations/rec.aspx?rec=y.2060

[5] http://www.gartner.com/newsroom/id/2905717

[6] https://newsroom.accenture.com/subjects/management-consulting/industrial-internet-of-things-will-boost-economic-growth-but-greater-government-and-business-action-needed-to-fulfill-its-potential-finds-accenture.htm

[7] https://www.abiresearch.com/press/the-internet-of-things-will-drive-wireless-connect/

© 2016 prpl Foundation – All Rights Reserved.

Or infusion pumps designed to administer medication at precisely controlled rates. Then there's the automotive sector – a major driver of IoT, according to Gartner. Already the electronics in cars cost as much as 40% of the total production cost of the vehicle.[8] Sensors monitor and display data on fuel levels, engine performance and more, while internet-connected systems manage the in-vehicle entertainment. IoT will become even more pervasive as driverless cars become mainstream – with roadside sensors communicating with on-board equipment.

Advances like these bring many challenges for security and privacy. These systems collect, process, store and send data to back-end services, which often share it among multiple service providers. This raises privacy questions over whether the owner of an IoT device or system wants this information to be passed around to unseen actors. For compliance purposes, some data should be securely collected and transmitted without identifying respective owners. The connected sensors and devices also may represent a point of vulnerability which attackers could use to infiltrate and disrupt systems for their own ends. There also are serious safety implications arising from the compromise of medical and automotive systems.

These are not theoretical concerns. Already real world examples are emerging. Just look at the SYNful Knock attacks on Cisco routers, in which the router's IOS operating system is intentionally modified and reprogrammed into the device. This allows attackers to monitor all data traffic flowing in and out of the network and grants them a backdoor into the device itself. Incidents like this are possible because these devices lack a solid Root of Trust in hardware to enable secure boot of the vendor's digitally signed software. A related challenge is that unchangeable software also means that nobody can fix anything – even the open source community. The presence of locked down systems is a convenient way for vendors to ignore fixing bugs in long-sold systems. There exists, therefore, a pressing need to examine how best we can secure IoT devices, before they become so embedded in our computing platforms, and society, that modifying or replacing them becomes impossible.

This document examines where the major threats to individuals and organizations exist in the Internet of Things as we currently understand it; the challenges we may have in securing it; and how a hardware-based approach founded on open source, interoperable standards can help to mitigate those threats.

1.1 Threats to Individuals and Organizations

Flaws in Internet of Things devices represent a different kind of risk to those we're more commonly used to reading about in corporate IT systems. Web application, browser or OS vulnerabilities that attackers exploit to achieve persistence in target networks will at worst result in the theft of that data or possibly interrupt operational processes. Law suits, brand damage, falling share prices, and regulatory penalties – as well as the cost of clean-up and remediation – all potentially await the breached company.

[8] http://www.autoblog.com/2010/06/08/how-much-does-software-add-to-the-cost-of-todays-vehicles-how/

© 2016 prpl Foundation – All Rights Reserved.

However, vulnerabilities already discovered by many security researchers could lead to something far more serious than economic losses and brand damage. Let's take a look at a few examples:

Connected Vehicles

Security researchers Charlie Miller and Chris Valasek demonstrated at Black Hat 2015 a much-talked about hack of a 2014 Jeep Cherokee in which they managed to remotely control the vehicle's steering and brakes.[9] Their research resulted in the recall of over 1.4 million vehicles[10] and has profound implications for car safety as we move ever closer to the reality of driverless automobiles.

The attack concentrated initially on compromising the car's Uconnect 8.4AN/RA4 entertainment system which is connected to the Sprint cellular network by default. By reverse engineering they exposed vulnerabilities in the system, then exploited weak implementation of network protocols in the D-Bus inter process communication system – i.e. port 6667 was open. Once inside, they were able to pivot to the TI OMAP-DM3730 chip in the Uconnect head unit. Reverse engineering again helped them to modify the firmware on this chip and re-flash an image to execute arbitrary code. From there a lack of separation in the system meant they were able to move

[9] http://illmatics.com/Remote%20Car%20Hacking.pdf
[10] http://blog.fcanorthamerica.com/2015/07/22/unhacking-the-hacked-jeep/

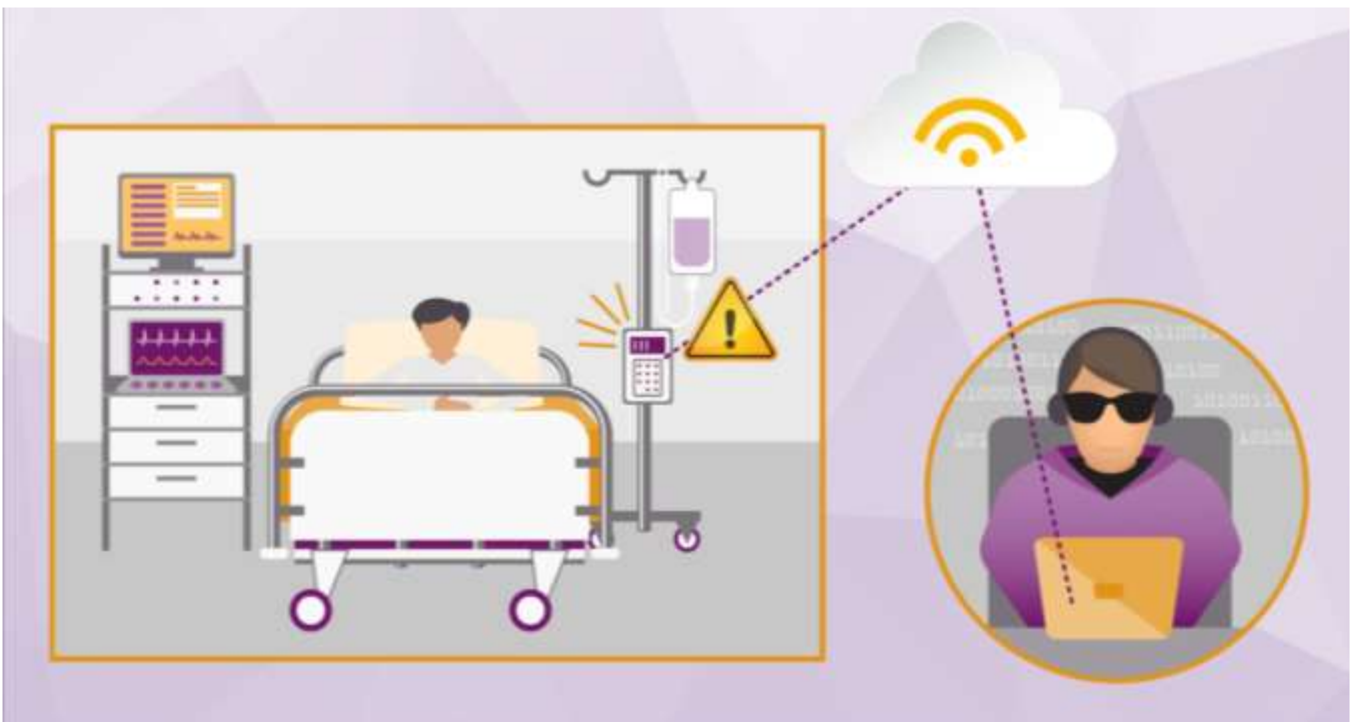
© 2016 prpl Foundation – All Rights Reserved.

laterally to the target CAN system (CAN MCU Renesas v850) which gave them remote access to many of the car's key functions including steering and brakes.

The researchers presented their findings after over a year of painstaking research. Nevertheless, in theory it highlights the potentially serious consequences of IoT security failings.

Healthcare

Healthcare is another industry that is coming to rely on connected devices and smart sensors to help medical professionals provide more effective patient care. However, the US Food and Drug Administration (FDA) was forced to warn hospitals[11] earlier this year against using a popular internet-connected drug infusion pump after research from Billy Rios revealed it could be remotely hacked.[12] Attacks like this may be harmful to human lives as medicine applied in wrong dosages becomes a potentially lethal weapon.



[11] http://www.fda.gov/MedicalDevices/Safety/AlertsandNotices/ucm456815.htm

[12] https://xs-sniper.com/blog/2015/06/08/hospira-plum-a-infusion-pump-vulnerabilities/

© 2016 prpl Foundation – All Rights Reserved.

It had the following warning:

> *"This could allow an unauthorized user to control the device and change the dosage the pump delivers, which could lead to over- or under-infusion of critical patient therapies."*

The affected devices were the Hospira Symbiq Infusion System (v3.13 and earlier), the Plum A+ Infusion System (v13.4 and earlier), and the Plum A+ 3 Infusion System (v13.6 and earlier). The manufacturer has claimed there are no known cases where these pumps have been accessed remotely by unauthorized parties. It is also claiming that most of these devices will be replaced in the next 2-3 years. However, with the healthcare IoT market set to be worth $117bn by 2020, according to MarketResearch.com, there's an increasing need for manufacturers to reengineer vital systems to ensure they can't be misused in this way.[13]

Weapons

In another Black Hat 2015 presentation, researchers Runa Sandvik and Michael Auger claimed to have found a way to hack the ShotView targeting system on Tracking Point's hi-tech Linux-powered rifles.[14] The company's .338 TP bolt-action sniper rifle is said to provide precise impact on targets out to .75 mile.[15] Although they claimed the company had "done a lot right" and minimized the attack surface, the researchers were still able to compromise the rifle via its Wi-Fi connection, exploiting software vulnerabilities to prevent the gun from firing, or even to cause it to hit another target according to their instructions.

[13] http://www.forbes.com/sites/tjmccue/2015/04/22/117-billion-market-for-internet-of-things-in-healthcare-by-2020/

[14] https://www.blackhat.com/docs/us-15/materials/us-15-Sandvik-When-IoT-Attacks-Hacking-A-Linux-Powered-Rifle.pdf

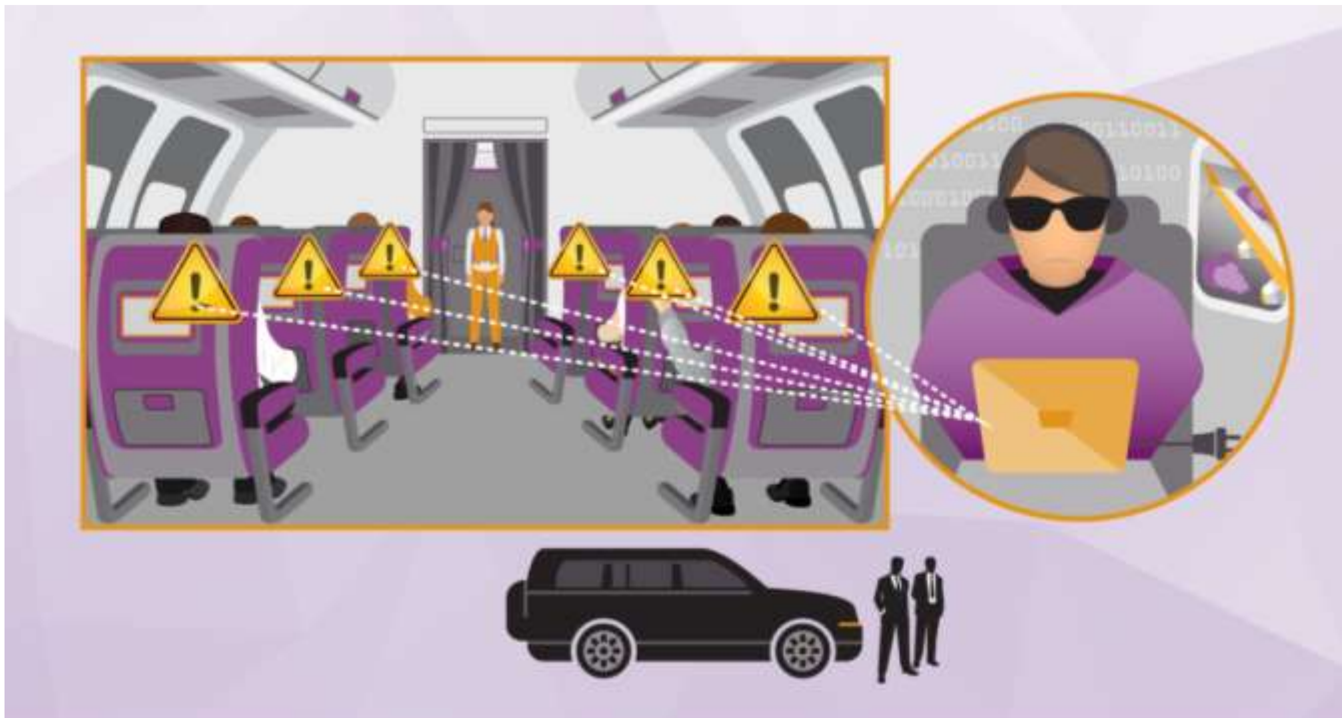
[15] http://tracking-point.com/precision-guided-firearms/precision-guided-bolt-action-338-lm

© 2016 prpl Foundation – All Rights Reserved.

Fortunately, a remote attack on the rifle couldn't make it fire as that requires a physical pull on the trigger. However, Sandvik and Auger were able to demonstrate how to effectively brick the rifle, making its computer-based targeting permanently unusable. For a weapon that costs $13,000 and could be highly dangerous in the wrong hands, the research is concerning.

Aircraft

Airplanes today are controlled by complex connected computer systems. Sensors all over the aircraft monitor key performance parameters for maintenance and flight safety. On-board computers control everything from navigation to in-cabin temperature and entertainment systems. One security researcher, Chris Roberts, was detained by the FBI and accused of hacking in-flight entertainment systems on a jet. He was apparently able to overwrite code on the airplane's Thrust Management Computer while aboard a flight, causing a plane to move laterally in the air.



Roberts denies having done this during a real flight and Boeing has claimed in-flight entertainment systems are isolated from flight and navigation systems.[16] However, when it comes to the aviation industry the stakes are even higher with regards to potential flaws in IoT systems. As airlines transition to even more advanced systems

[16] http://www.cnet.com/news/fbi-pulls-computer-security-expert-off-flight-after-he-tweets-about-hacking-its-systems/

© 2016 prpl Foundation – All Rights Reserved.

leveraging these technologies more attention needs to be focused on underlying system weaknesses that could represent a security and safety risk.

1.2 Challenges to Secure Embedded Systems

The work of Miller and Valasek, Rios, Roberts and many others has helped us to spot some of the major vulnerabilities in key IoT systems. A major factor affecting all challenges is complexity: IoT systems are extremely complex with many “moving parts.” A vulnerability that may affect one device used in a particular context might not affect equivalent devices from other manufacturers. At this early stage of IoT development, it is important for stakeholders to be vigilant in analyzing systems and subsystems for potential vulnerabilities. Eventually, future IoT devices will touch many aspects of our lives, such as appliances, door locks, phones and home alarms. Without securing IoT devices, a single vulnerable coffee maker might give someone access to your whole connected life. Below are significant issues that may have broad applicability in securing embedded systems:

Reverse Engineering

In all of the examples listed above, the researchers in question built their case thanks in no small part to their ability to reverse engineer the underlying executable code of the systems they were investigating. This means that, in theory, a malicious attacker could too. Miller and Valasek, for example, reverse engineered the Uconnect 8.4AN/RA4 entertainment system in the 2014 Jeep Cherokee they hacked, to reveal vulnerabilities they were subsequently able to exploit to gain an initial foothold inside. They then did the same to firmware on the key Renesas V850ES/FJ3 chip, which handles controller area network (CAN) communications. As they revealed in their paper : “the most important part was reverse engineering the IOC [inversion of control] application firmware because we knew it would reveal the code necessary to send and receive CAN messages from the bus.”[17]

The truth is that firmware binary code is usually published somewhere online if one knows where to look – to help users with ongoing maintenance. And if it isn’t there is a wealth of sophisticated debugging tools around, such as the Joint Test Action Group’s tool called JTAG, which can be used to extract a copy of the software from the device itself. Interactive disassemblers like IDA can generate assembly language source code from machine-executable code. In combination with other tools and techniques it is becoming easier than ever to reverse engineer a binary image, work out what it does, where its vulnerabilities are and how to exploit them. This vulnerability is especially critical for embedded systems because they are shipped out to users and are physically available to the attacker by design.

[17] http://illmatics.com/Remote%20Car%20Hacking.pdf

© 2016 prpl Foundation – All Rights Reserved.

This raises a significant issue with proprietary software, namely that the old concept of “security by obscurity” is no longer an effective defense. These designs are simply no longer a secret – or at least they can be reverse engineered to remove any residual ”obscurity.” And the manufacturers producing them have been caught time and again:

- In 2013, for example, a researcher managed to reverse engineer and find vulnerabilities in the Philips Hue IoT lighting system that he then wrote malware to exploit – causing a complete blackout.[18]
- It was discovered that Netgear GS105PE Prosafe Plus switches running firmware version 1.2.0.5 had been designed with hard-coded passwords that could allow an attacker to authenticate to the web server running on the device.[19]
- Even automated teller machine manufacturers have been caught. In 2014, two teenagers in Canada found an old operating manual online. Following the step-by-step instructions they reached the operator mode menu, guessed the default password and were able to change various fields such as the surcharge amount.[20]

Hazards of the Software Engineering Process

In the context of our security guidance for embedded systems, it is important to understand how the actual process of software engineering enables mistakes that can become vulnerabilities in firmware.

The software engineering process is similar for small and large developers alike. Engineers consider requirements for complex tasks, break them down into small, manageable processes, design the logic flow and assign programming teams to write the code. One might assume that a large company makes fewer mistakes because it has more hands on the project, but usually this is not the case. Each piece of software is handled by a team – typically a small number of programmers. And often, just some of these are writing the actual code and reviewing their own work. Due to abstraction of code, it’s very difficult for someone to do a quality review of someone else’s code. The best person to fix mistakes is the person who wrote the code (assuming they know what they’re doing in the first place!). As one combines dozens, hundreds and thousands of pieces of code written by different people into interconnected modules for the application or operating system, it’s no wonder that mistakes occur. The same is true for a tiny startup as much as it is for a giant developer like Microsoft, Adobe Systems or Apple.

Larger development teams can amplify natural mistakes due to the processes of running a large organization. There is a finite number of programmers who are constantly pulled in different directions as corporate priorities shift, switch projects within the company, or leave the organization and their old projects behind to new arrivals

[18] http://www.dhanjani.com/docs/Hacking%20Lighbulbs%20Hue%20Dhanjani%202013.pdf
[19] https://www.duosecurity.com/blog/hard-coded-and-default-passwords-massive-attacks
[20] http://www.zdnet.com/article/bank-of-montreal-atm-hacked-with-weak-password/

© 2016 prpl Foundation – All Rights Reserved.

who must come up to speed from scratch. Given the complexity of software today, mistakes are bound to happen.

And there's another related issue that can lead to mistakes: the use of software libraries created by someone else. Libraries offer a shortcut to programming with pre-written code, sub-routines and other tools. Programmers commonly use plug in content from libraries to accelerate coding – particularly for complex processes such as implementing a myriad of protocols for communications over the Internet. Using libraries is a mainstream practice, but plugging in code written by someone else can lead to profound vulnerabilities. This vulnerability is called out as A9 in the OWASP Top 10.[21] For example, Stagefright is a library used for playing multimedia formats such as MP4; it is a core part of the Android operating system. In July 2015, a bug was revealed that lets an attacker perform operations on an Android device with remote code execution and privilege escalation. Use of this essential library compromised about 95 percent of Android devices – approximately 950 million smartphones. [22]

Hazards of the software engineering process produce vulnerabilities in some of the most widely attacked software out there. As a consequence, users are subjected to a never-ending river of updates and patches. If the IT department does not keep up with timely patching, an entire organization's network and digital assets may be subject to exploitation of these vulnerabilities. For example:

- **Microsoft Windows**: Among the most exploited software ever designed. On the one hand this is because it is hugely popular throughout the world. But in 2014 Microsoft apps were responsible for almost one quarter (23%) of the Top 50 most common vulnerabilities, according to Secunia.[23]
- **Adobe Flash**: Flash has earned a nasty reputation for a constant arrival of new vulnerabilities. Many security experts recommend disabling or uninstalling Flash.[24] Flash is now being shunned by major publishers; most recently the BBC decided to drop support for Flash.[25] There are more than 550 Flash Common Vulnerabilities and Exposures (CVEs).[26]
- **Apple iOS:** Even version 9 – Apple's most recent effort – contained patches for over 100 vulnerabilities. There are about 700 CVEs in iOS.[27]

[21] https://www.owasp.org/index.php/A9_2004_Application_Denial_of_Service
[22] https://en.wikipedia.org/wiki/Stagefright_%28bug%29 and http://www.pcmag.com/article2/0,2817,2491035,00.asp
[23] https://secunia.com/resources/reports/vr2015/
[24] http://krebsonsecurity.com/2015/06/a-month-without-adobe-flash-player/
[25] http://www.bbc.co.uk/news/technology-34399754
[26] https://www.cvedetails.com/vulnerability-list/vendor_id-53/product_id-6761/Adobe-Flash-Player.html
[27] https://www.cvedetails.com/vulnerability-list/vendor_id-49/product_id-15556/Apple-Iphone-Os.html

© 2016 prpl Foundation – All Rights Reserved.

Broken Firmware Updates

There's another common weakness at the heart of most IoT systems and that involves the lack of timely updates and lack of authenticating updates. The majority of electronics have a means to update the key firmware in the chip, allowing the manufacturer to run updates patching the security vulnerabilities that will inevitably be found. Unfortunately, many manufacturers do not update in a timely manner – even when notified by security researchers. Delays often occur due to the complexity of coordinating changes between various teams and code bases throughout the supply chain.

A more serious and fundamental factor is that firmware is rarely cryptographically signed, meaning that an attacker could in theory replace it with new software of their choosing. This is akin to handing criminals a key and allowing them to replace the lock.

Miller and Valasek exploited this security oversight in their research. They found that the IOC could be re-imaged with firmware and that "no cryptographic signatures are used to verify the firmware is legitimate." This meant that they could effectively craft specific CAN messages "to make physical things happen to the vehicle."

A similar security gap has enabled real world attackers to compromise hundreds of Cisco routers in the so-called 'SYNful Knock' campaign. On this occasion the attackers found out the router access credentials, and were able to implant a modified Cisco IOS operating system image on the router's firmware that persists on the machines even after reboot. The image itself was designed to provide unrestricted access using a secret backdoor password and it allowed the attackers "to load different functional modules from the anonymity of the internet," according to the FireEye researchers who found it.[28] With persistent privileged access to a router, the attackers could monitor all the data flowing in and out of the device.

These attacks were both possible because in most IoT devices there's no boot up mechanism to establish whether the software running on the chip is signed, certified or otherwise approved by the vendor. Chip firmware in IoT devices should be updateable, but not in a way that allows anyone with the right set of skills to re-flash it with their own code.

Lateral Movement

As discussed, it's impossible to eradicate all vulnerabilities from software, and errors are particularly likely to creep into both proprietary and open source code. But we can look to lock down and isolate those flaws by restricting lateral movement inside a targeted system. From an engineering perspective, the developers of many modern technology systems containing IoT sensors are focused on minimizing hardware design specifications wherever possible. This is understandable – integrating as many functions as possible into one piece of

[28] https://www.fireeye.com/blog/threat-research/2015/09/synful_knock_-_acis.html

© 2016 prpl Foundation – All Rights Reserved.

hardware reduces costs with a lightweight design. Unfortunately, many manufacturers fail to separate operations of internal systems as this increases costs relative to a lightweight design integrating as many functions as possible into one piece of hardware.

The omission of separation and isolation makes lateral movement inside a compromised platform or system easier than ever. It's the lack of separation that allows for attacks on shared devices. There's no way that Miller and Valasek should have been able to pivot from the compromised Uconnect entertainment head unit to the CAN bus responsible for sharing steering, braking and other critical functions of the vehicle. Separation was not used to implement security. Similarly, researcher Chris Roberts should not have been theoretically able to get anywhere near the Thrust Management Computer of an aircraft having compromised its In-Flight Entertainment system, as the FBI search warrant application alleges.[29]

Lateral movement inside a compromised system is a tactic favored often by sophisticated attackers. Their strategy is to establish an initial beachhead in a corporate network via malware hidden inside email or an email attachment sent via spear phishing, and download command-and-control malware to an endpoint. However, this path alone will usually not provide access to targeted data. So the attackers then look to escalate privileges on non-administrative users and gain access to more high value targets – ultimately finding those who they can use to access the database they need.

Lateral movement also helped all of the researchers we've highlighted earlier to achieve their respective end goals. It must be limited if we're ever going to improve IoT security.

1.3 A New Approach

As researchers around the world have already demonstrated, Internet of Things technology is exposed to vulnerabilities in several critical areas, raising the very real prospect that actual physical harm could be inflicted on end users. In the following pages we'll outline in detail our guidance for a new hardware-led approach that can help IoT developers overcome these challenges.

Before that, let's summarize briefly some prerequisites to address challenges described in the previous section. Keep in mind that those were but some of the challenges and more are sure to present additional obstacles in addressing security for embedded systems. For this reason, aspects to our new approach below are preliminary and will need to evolve as stakeholders step up their focus and resolve on security.

[29] http://www.wired.com/wp-content/uploads/2015/05/Chris-Roberts-Application-for-Search-Warrant.pdf

© 2016 prpl Foundation – All Rights Reserved.

Open Source

Running open source software in IoT systems can bring many benefits, including the collective effort of global contributors to help produce stable software. Obviously, being open source is no guarantee of quality. But as we noted, the hazards of software engineering combined with the size and complexity of modern commercial software programs almost guarantees the resulting software will not be a clean, secure end product. It's no surprise that major developers like Microsoft and Adobe are forced to release security updates. Many updates are released on a regular schedule, such as Microsoft's monthly "Patch Tuesday" event; many others are suddenly released as "urgent," and require IT professionals to drop everything and deal with yet another emergency resulting from coding mistakes. Compare the code creation and maintenance processes of these commercial entities to more eyeballs on a typical piece of open source software and it's easy to see why many regard the open approach as the preferred path to stable, secure code. And participants in open source projects are not bedroom hobbyists. Global contributors to Linux and other open source software hail from major technology companies.

Another benefit of open source is co-existence of a mutual contributors' goal of striving for quality and usability. Too often with proprietary software, features are added or removed according to commercial imperatives, internal politics or other corporate dynamics rather than the best interests of the software and users. In open source it's all about doing what's best for the software and the end-user community. There's a clean, clear, Darwinist logic at play in the open source community where only the best code survives. This is in stark contrast to the closed world, where software companies frequently build new products on the foundations of old, adding extra complexity and introducing unintended errors.

Open source is not infallible, of course. Major vulnerabilities like the Heartbleed bug in OpenSSL have taught us that. But when there are problems, the community is on hand to provide updates to fix the issue, sometimes in a matter of hours. It's common to have a rolling process producing and making available near-real-time updates (e.g. Linux Debian's security model). This is certainly not the case with proprietary code – Google only recently announced its commitment to monthly updates for Android, which is mostly open source code.

Developers also need to work within open standards. Some of the brightest minds all over the planet are working on designing and building platforms, systems, supporting software and applications for IoT. For example, weak implementation of network protocols enabled Miller and Valasek to infiltrate the Jeep's D-BUS via an open port (6667), for example. The result is that the network stack often becomes the path of least resistance for attackers.

This is why we need global, interoperable IoT standards. They effectively allow firms to outsource the trickiest work to the subject matter experts. These experts create the most secure standards and frameworks possible for designers to follow.

© 2016 prpl Foundation – All Rights Reserved.

Secure Boot

The software in too many embedded devices features a potentially fatal flaw – it's not cryptographically signed. This makes it easier for a hacker to reverse engineer the code, modify it, re-flash the firmware and reboot to execute arbitrary code. IoT products must be designed so they only boot up if the software running on the chip is signed by the vendor. Signing will help prevent any other parties from tampering with the software, but signing alone will not prevent reverse engineering the code and discovering potentially exploitable bugs such as a missing input validation. Without secure boot, a third party could replace the very brain of a device and instruct it to carry out arbitrary actions. And that's exactly what the researchers described earlier were able to do. And it's what the attackers who hacked Cisco routers did recently.[30]

To enable a Root of Trust we need a secure boot anchored in the device hardware, as described in Section 2.2.2. This is to ensure that the system boots up only if the very first piece of software to execute is cryptographically signed by a trusted entity – i.e. the vendor. It needs to match on the other side with a public key or certificate that is hard-coded into the device and therefore made completely unreplaceable. By anchoring this Root of Trust into the hardware, tampering becomes impossible. A determined attacker might still be able to extract the original firmware via JTAG, reverse engineer and modify it but it won't match the public key burned into the hardware – so the first stage of the boot up will fail and the system will simply refuse to come to life.

Once the root of trust has been established, that initial piece of software will make identity and integrity checks with the next piece in the boot chain and so on until the system is fully and securely operational. The integrity check will eventually continue at runtime to make sure no modifications are applied after system boot.

Hardware-level Virtualization

Too many embedded systems allow for lateral movement within the hardware, allowing attackers to jump around inside until they find a way to exploit what they're really after. It's understandable that manufacturers are trying to rationalize, collapsing as many functions as possible within one single piece of hardware, like a system-on-a-chip (SoC). But from a software perspective, there's no reason why these separate functional domains should be visible to each other on a shared platform. No one should be able to access an airplane flight control system via its on-board entertainment platform, for example. Similarly, white hats like Miller and Valasek should not be able to move from a Jeep's head unit to its CAN bus to take control of the vehicle.

The answer lies in hardware-assisted virtualization to separate each software entity, keeping critical components secure and isolated from the rest. This new foundational tier of security requires a secure hypervisor – a lightweight, compact piece of software with relatively few lines of code – to provide a virtual

[30] https://www.fireeye.com/blog/threat-research/2015/09/synful_knock_-_acis.html

© 2016 prpl Foundation – All Rights Reserved.

environment for each software element to run in parallel. The hardware-based focus of this approach to virtualization means less software is required to implement strong security such as with paravirtualization, as we describe in Section 3.3.2. Using more software grows the probability of vulnerabilities. Less software reduces the attack surface, which is a good thing for when the Internet of Things eventually hosts billions of devices.

If we think about it from a risk management perspective, no software is 100% safe from exploitation. But this method of secure separation enables more confidence in attack prevention: if one element is compromised, at least the attackers will not be able to use it as a stepping stone into other areas of the system. Secure separation like this would have meant Miller and Valasek were able to interfere with the Jeep's in-car entertainment system, but crucially not then move to the vital system which controlled steering and brakes.

Researchers are exploring new ways to achieve security by separation by adding independent standalone "security appliance" subsystems on a chip. These devices consist of a CPU, memory and a hardened operating system and perform various security-related processes such as intrusion detection and prevention. A major benefit of this approach will be decoupling the exploitation of main hardware from the security components. As a result, hardware-based security will become even stronger. The system may require these individual virtual services to speak to each other. Inside a vehicle there may need to be communication between entertainment system and engine system so the volume of the radio turns up automatically as the car accelerates and outside noise increases, for example. The answer here is secure inter-process communication that allows instructions to travel across the secure separation we've created in a strictly controlled mode. Incidentally, this architecture is a perfect fit for companies whose very business models are based on copyrighted content, like Netflix. If there's no secure separation between a video stream and, say, a rogue Android app in a modern smart TV, that content could leak with serious financial implications for the service provider.

Beginning the Journey

The model we have described above and that we'll outline in more detail below is an ideal – the "promised land" of hardware security. But in practical terms not everyone is going to be able to get there straightaway. Chip support for this kind of hardware-assisted virtualization is not yet widespread. In the short term, strengthening security in embedded systems will require the use of alternative or hybrid approaches.

One example is paravirtualization, which is a software-based technique for creating virtual or separate processes on embedded systems without hardware support. Paravirtualization uses virtual memory or page tables as does hardware-assisted virtualization. Paravirtualization is a practical alternative but can result in performance issues without implementing additional measures.

© 2016 prpl Foundation – All Rights Reserved.

For some developers, a good intermediate step is the use of Linux containers, which enables running multiple isolated applications from the one kernel. Secure elements and Root of Trust[31] might not be available in most hardware today, but this should not prevent security conscious manufacturers from encrypting and signing their firmware and from making security patches available in a timely manner. Finally, modern systems must provide an infrastructure enabling secure debug during product development and testing. In addition, innovative JTAG implementations should secure debug ports to help enforce separation when multiple service providers are integrating their offerings on a single platform.

[31] See section 2.2 for details about Root of Trust and other secure elements.

© 2016 prpl Foundation – All Rights Reserved.

2.0 Fundamental Controls for Securing Devices

There is broad interest and financial incentive in achieving strong security for what will eventually be billions of devices used on the edge of the Internet of Things. But our guidance on hardware-based security is not just for the Internet of Things; this guidance can effectively protect all kinds of mass-scale, IP-controlled devices with embedded systems, such as home gateway routers, televisions, mobile devices and automotive systems. Our guidance here addresses two basic questions: "What are we trying to protect?" and "What is required to enable protection?" The answers result in a shortlist of fundamental controls necessary to implement hardware-based security. The key to their implementation will be new open Application Programming Interfaces (APIs) that enable secure inter-process communications between multiple system-on-chip processor platforms, interfaces and applications. The practical end-goal is achieving "Security by Separation," which we address in the next section.

2.1 Challenges of Securing Devices

2.1.1 Inadequacy of Legacy Systems

There are two general categories of potential attacks on devices. Physical attacks entail tampering with a device to gain access to and control of its functionality. A physical attack may also be done with the intent to discover how a device works and clone it for economic or other benefits. Remote attacks entail accessing a device via a compromised communications path, such as its connection to the Internet or via inter-process communications between hardware elements. Our goal for hardware-based security is to protect devices from both types of attacks.

Challenges of preventing attacks like these are new for manufacturers and OEMs. Typically, embedded systems have relied for security on a static proprietary architecture, a single operating system, applications from a single provider, and little-to-no need for network communications. While this was adequate in the past, new requirements for devices that must support multiple service providers on the same platform require secure inter-process communications between all system elements over several separate domains.

2.1.2 Defining Targets for Protection

The definition of "What to protect?" hinges on answers to several critical issues:

- **Is the device secure?** The first challenge is ensuring the device is secure. When a device is booted, stakeholders depend on it coming up in a stable, known state. Assuring the booted state requires preventing someone from altering the known state. Assurance includes physical security (the essence of

© 2016 prpl Foundation – All Rights Reserved.

which for many boxes consists of the screws holding the faceplate to the box!), but also must include virtual access for most future attacks will be via network communications.

- **Is the software on the device trusted and stable?** Stakeholders also depend on the integrity of software running on the device. Software begins with the platform bootloader, initialization, loading the operating system and applications. Software comprises the device's state and secure operating environment (also called the Trusted Execution Environment or other terms depending on the particular device). The state or operating environment must be secure to fulfill operational intent of the embedded device.
- **Has the device been compromised after deployment?** Maintenance of security is critical for ensuring that the device is continuously working as intended. When millions (or billions) of devices are deployed, stakeholders need to know on an ongoing basis if the devices have been compromised. Detection of compromise is a major new issue that must be addressed for the new genre of embedded devices. Detection of compromise for IT devices is a common process in enterprises. But it's far more difficult in typical low power embedded devices that contain minimal available processing power and memory, and are globally deployed by millions or billions of units. For consumer devices, security cannot be an afterthought and needs to be provided almost "for free" in order to not compromise thin profit margins on low-cost embedded devices.
- **How to recover a compromised device?** The related issue is if a device has been compromised, how can it be recovered to its intended state? This is another new frontier since manufacturers and OEMs will need to maintain the ability to securely communicate with millions or billions of deployed devices and execute recovery processes.
- **Is the platform's data secure?** Another new frontier is data protection, particularly for devices in the Internet of Things. Data captured by embedded devices is either at rest, in use by the device, or in motion when sent elsewhere. The use of virtualized IT infrastructures and cloud computing means that data will usually reside somewhere in the cloud. How is the data being protected? What can happen if an unauthorized person accesses device data? Can they compromise device operations? Can they use the data to personally identify users and their usage patterns of devices? Are manufacturers and OEMs equipped to notify consumers when their personal data has been breached? Data security is becoming a mandatory issue to consumers who are reading everyday about data breaches at iconic institutions.

By addressing development issues with security guidance from prpl Foundation, manufacturers and OEMs will be able to confidently proceed with plans for securing the new genre of embedded devices that will empower the world with innovative computing capability on a global scale.

© 2016 prpl Foundation – All Rights Reserved.

2.2 Requirement Specifications for Device Security

Requirements for the new genre of device security are the functions that must be present to address the security issues described above. They include:

- Root of Trust Device
- Secure Boot Process
- Chain of Trust with Authentication and encryption
- Hypervisor and Guest Operating System Authentication
- Secure API

2.2.1 Root of Trust

Roots of trust are functions and data in a device or embedded system, the correctness of which must be implicitly trusted to assure intended operation. Taken together, these are usually called the device Root of Trust (RoT). These constitute the foundation for integrity of the device. A RoT may be implemented entirely in hardware, where the RoT cannot be changed because it is built into the hardware of the system. A RoT may be implemented in a bootstrap read-only memory (ROM) that provides instructions for the processor, allowing some flexibility to make changes to the RoT without requiring radical changes to the architecture of the integrated circuit. One may also implement parts of the RoT in cryptographically signed firmware or software that executes in random access memory (RAM); this approach provides the most flexibility as new versions of RoT code can be provided to fielded devices. Our Guidance on implementing a RoT is for one that is at least partially based on firmware or software and uses cryptographic processes to provide authentication of later stage code. Our recommended RoT configuration incorporates the following elements:

- *A state machine* (whether implemented in hardware, immutable firmware or any combination) to bring the device processor out of its reset state to a state in which it is running authenticated code.
- A *root-of-trust public key* immutably bound to the device that represents the highest authority for issuing authorized firmware or software for the device.
- *Cryptographic primitives* for hashing, public key cryptography and optionally symmetric encryption.
- *Secure boot process* that can use the cryptographic primitives to authenticate the firmware or software that will ultimately run on the system in its mission mode.

The RoT public key is discussed further in Section 2.2.3.

2.2.2 Secure Boot Process

A trusted or secure boot process is used to start a device in a secure fashion. It is a basic, fundamental requirement for securing devices. Secure boot validates the integrity of software before it is loaded for

© 2016 prpl Foundation – All Rights Reserved.

execution. Integrity validation prevents unauthorized modifications to code and data when the system initially starts running that code. Secure boot builds trust in the system starting from a root of trust by using a secure bootstrap subsystem in read-only memory or a hardware state machine, and authenticates the code with asymmetric keys.

In creating an approach to secure boot, the Root of Trust elements are typically kept as simple as possible to assure correctness. "Ratcheting" multiphase bootstrap systems are common. These boot the system through a series of phases starting from very simple to complex (an example is provided in Figure 2.1). Each phase adds features and functionality to the booted device.

Our recommended secure boot process needs a trusted hash function, public key signature verification algorithm, and a root public key for the signing hierarchy. Public keys and object signatures may be passed using proprietary formats or in certificates. The X.509 certificate is a common standard format. This process is shown in Figure 2.1.

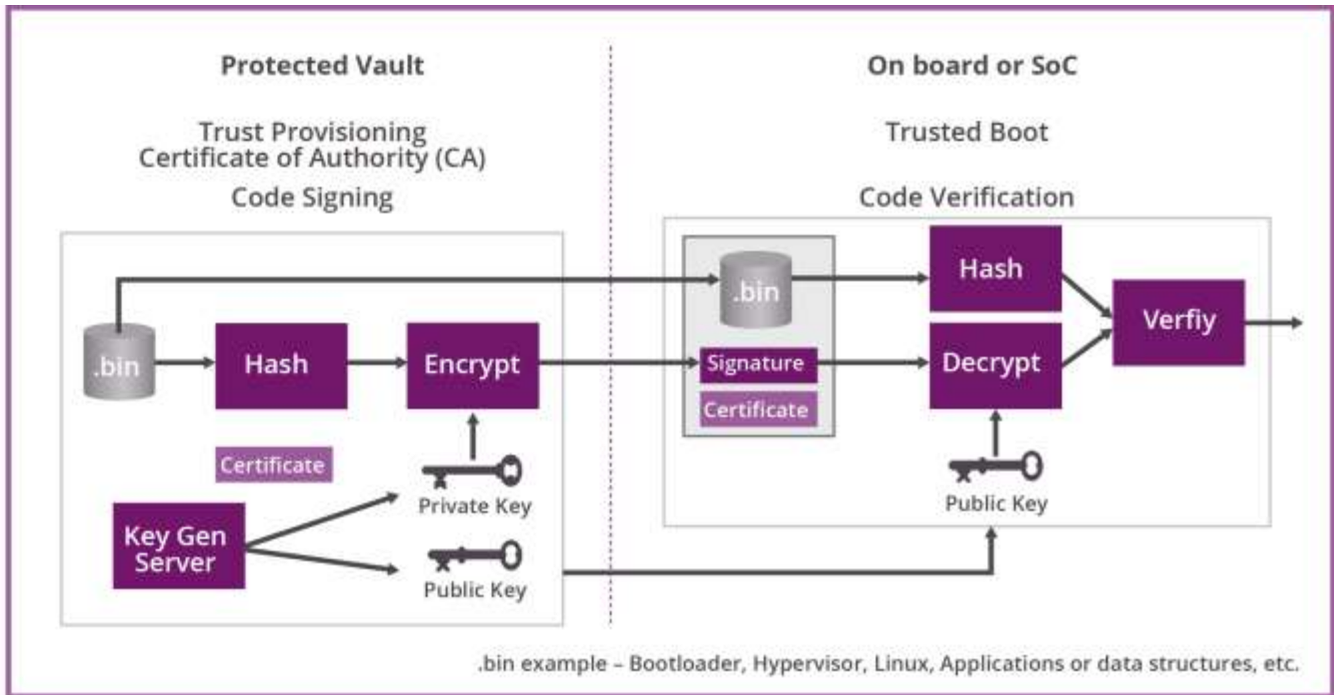


Figure 2.1: Code Integrity Enables Secure Boot

© 2016 prpl Foundation – All Rights Reserved.

Following is an example of a step-by-step secure boot process for a security scenario using hardware virtualization and a two-stage boot loader:

1. Root-kernel mode executing.
2. First boot loader is located at reset exception vector – this is what is executed first and would be executing from ROM, OTP-ROM or other immutable memory.
3. The first stage boot loader verifies that the secondary boot loader is authentic, i.e. has been signed by an issuer traceable to the RoT public key.
4. The first stage boot loader loads the secondary boot loader into on-chip RAM (OCM).
5. Control is now passed to the secondary boot loader.
6. The second boot loader authenticates and loads the hypervisor into its designated RAM, which should occur only after the hypervisor is successfully authenticated.
7. Control is passed to hypervisor as the chain of trust, which is now running in the isolated area of DRAM.
8. The hypervisor sets up the partitions provisioning up to several VMs/Guests.
9. The hypervisor loads Linux-0 image into DRAM designated for VM0.
10. The hypervisor authenticates Linux-0 image with the RoT.
11. Repeat steps (9) & (10) for VM1/Linux-1.
12. Repeat steps (9) & (10) for VM2/Linux-2.
13. Repeat steps (9) & (10) for VM3/Linux-3.
14. The hypervisor proceeds to run VM/Guest context switching (CS) according to the established policies.

At any stage, if code authentication fails, the system can transition to a safe state. The specific definition of a safe state will be context dependent: failed authentication of the second stage boot loader may result in the system halting; failed authentication of a VM may simply result in the VM not being started.

2.2.3 Chain of Trust Authentication

It is common for the RoT public key that is stored in the device to be the root of a chain of public keys. Storage is typically in one-time password memory or in ROM. Lower keys in the chain are signed by an authority possessing higher keys in the chain, ultimately terminating in the RoT key holder. For example, the signing authority for code may have its signing public key signed by the chip manufacturer's RoT key. If the signing authority is ever compromised, its public key can be revoked and new firmware is then signed by a different signing authority with its own unique key authorized (signed) by the manufacturer's RoT key. It would not be uncommon for the hypervisor to be signed by a different signing authority than the second stage boot loader. The use of chains of trust, traceable to the RoT public key, allows the establishment of trust that the various firmware sources are all authentic. Figure 2.2 shows the result of these processes, which is a secure operating environment enabling trusted execution of embedded platform functionality.

© 2016 prpl Foundation – All Rights Reserved.

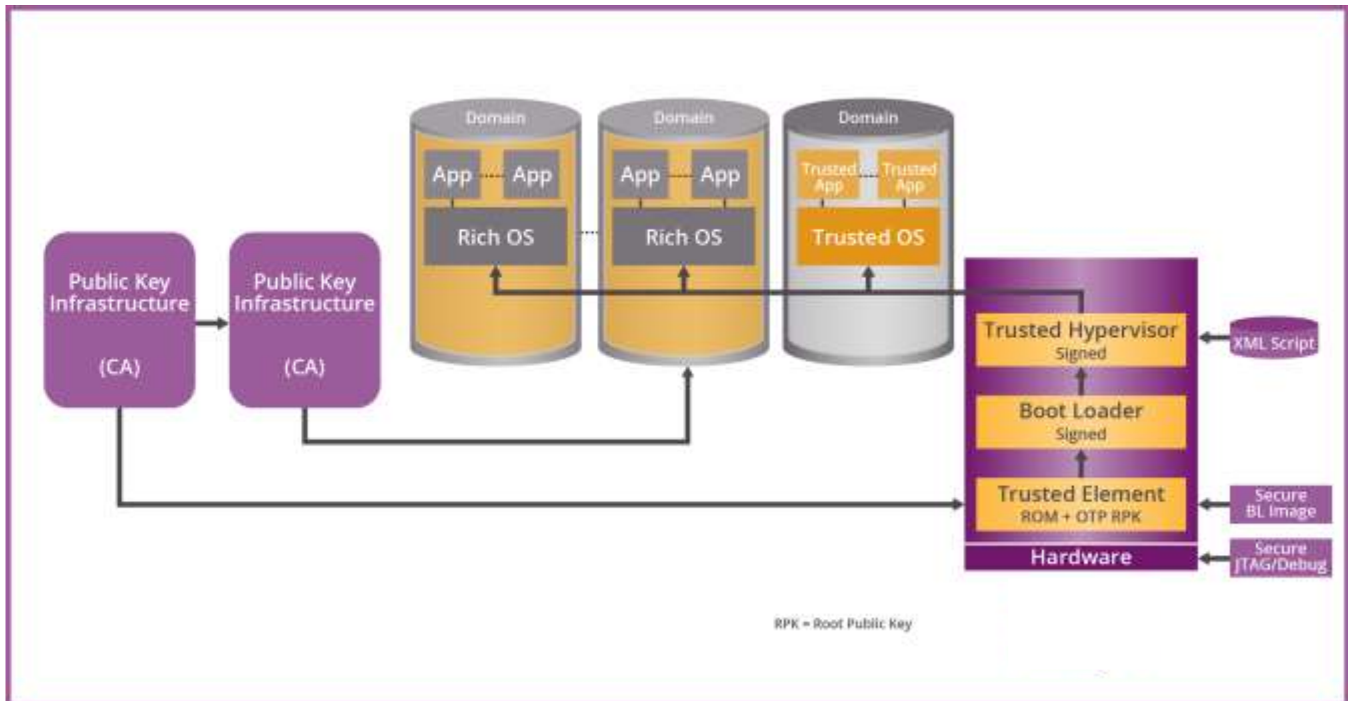


Figure 2.2: Required Components for Chain of Trust Authentication

2.2.4 Hypervisor and Guest Operating System Authentication

When virtualization is used, it is controlled by a hypervisor, which is a small body of privileged code that sits above the hardware and manages system-on-chip resources by defining access policies for each domain to control which domains have access to which resources and under what conditions. The domains are called virtual machines (VMs) or guests. For security, it's vital for the hypervisor and guests to establish a chain of trust with authentication services in the secure operating environment. This allows the hypervisor to be securely anchored to the hardware and trusted for all operations.

For example, in a typical MIPS embedded system, a translation lookaside buffer (TLB) is used by memory management hardware in a hierarchal fashion to improve virtual address speed. When using hardware-assisted virtualization, a four-level hierarchy TLB enables isolation of VMs from each other. The hierarchy consists of two major layers: Guest TLB and Root TLB. Each VM/Guest operates in a traditional two level address space with user and kernel modes, but the hypervisor is responsible for allocation of this address space to guests using the Root TLB to separate guests from each other. In this way, little or no modification is required for the guest VM kernel and applications.

© 2016 prpl Foundation – All Rights Reserved.

The hypervisor in privileged mode is in control of the Root TLB, redirecting the guest access to the correct physical address. The hypervisor controls all guest CPU bus transactions per pre-established access policies – thus enforcing isolation policies among the guests. In paravirtualized systems, the OS kernel is de-privileged so that accesses to sensitive addresses are trapped to the hypervisor.

The isolation between the guest application and the privileged root kernel is the first step in securing the hypervisor. It is also important to assure that the root kernel's "attack surface area" is minimized and all root kernel front and back door entries are restricted and under tight supervision of the hypervisor. A small-footprint hypervisor is preferred for an embedded platform. Today, there are many proven third party small-footprint hypervisors available for embedded applications.

A trusted hypervisor is central to the overall security of the embedded platform. However, in order to enforce a trusted hypervisor, additional components are required. Elements such as secure network-on-chip, secure graphics processing unit, and others play a key role in creating a secure system-on-chip. The root of trust intends to enforce the following and more: Authenticate the origin and the integrity of the hypervisor images before execution boots the system into a secure hypervisor state and ensures that it runs only trusted code; and subsequently establishes the hypervisor as the chain of trust.

Figure 2.3 shows how to establish an embedded system with multiple isolated and secure environments.

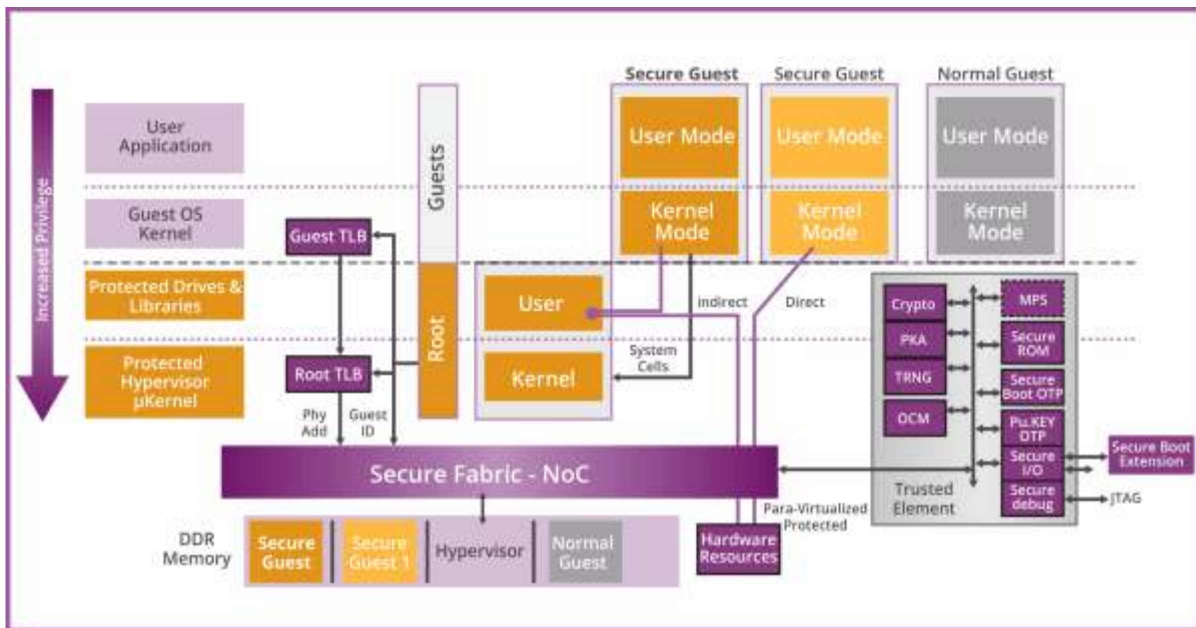


Fig. 2.3: Elements for Creating a Secure Embedded System

© 2016 prpl Foundation – All Rights Reserved.

3.0 Security by Separation

Security by Separation is a classic, time-tested approach to protecting computer systems and the data contained therein. Security is the policy principle for protecting an asset. *Separation* was historically associated with "air-gapped" systems not interconnected by a network. In the context of our Guidance, separation is a technical mechanism used to implement and maintain security. Separation may entail the use of different physical devices or other means, such as memory mapping. By separating and restricting the availability and use of assets, security is enforced according to prescribed policy. It is often said that the only secure system is one that is not connected to any other system – and even then an "air gapped" system might be compromised by non-traditional means (e.g. Stuxnet virus compromise on Iranian uranium enrichment centrifuges used in nuclear reactors). However, in a world where much value is ascribed to the interconnection of systems to create networks, a physically and logically isolated system is not very interesting to most people. Our Guidance focuses on systems that can retain their security attributes even when connected to open networks.

Consider the example of data center security, for which there are two typical categories of separation mechanisms: physical and logical. Physical separation includes the use of multiple facilities housing assets, locks, security guards, and procedural controls such as restricted access to the server room or a storage facility containing backup tapes. Examples of logical separation include resource access privileges assigned by job role, unique user IDs and passwords for authenticating access to systems, use of digital certificates to authenticate the use of particular servers or applications, restricted ability to use specific applications on the network or alter system configurations.

Virtualization is another technology that can achieve logical separation. Virtualized systems are used to *simulate, isolate and control* IT assets such as hardware platforms, operating systems, storage devices and resources for networking. Many cloud-based computing architectures are based on virtualization, and its use has surged in data centers of all kinds. Data centers typically use virtualization to consolidate physical systems by running more software on fewer devices. But virtualization really has an old history, as "Virtual Machines" (VMs) were pioneered in the early 1960s by companies such as Bell Labs, General Electric and IBM Corporation. Our guidance uses virtualization to bring direct benefits for securing embedded systems.

3.1 Using Separation by Virtualization to Secure Embedded Systems

The goal of separation used for security purposes is to create and preserve a trusted operating environment for an embedded system. Separation is intended to prevent exploitable defects in one virtualized environment from propagating to adjacent virtual environments, or to the physical platform as a whole. Failures that occur in one environment are limited to that environment. Of course, when an adversary has a greater level of access, the challenge grows to fend off attacks. The greatest level of access is full physical access to the host system. Secure

© 2016 prpl Foundation – All Rights Reserved.

separation allows an embedded system to process sensitive data securely on behalf of client applications, and to continue doing so if one of the virtual environments is compromised. Separation also enables protection across and between all subsystems of a system-on-a-chip within a unified memory architecture. This means protection covers not only the CPU, but also graphics processors, audio and video processors, communications subsystems, and other subsystems of the chip.

The strategic goal of our security guidance for embedded systems – particularly using virtualization – is to achieve widespread, multiplatform enablement of trusted operating environments that are not limited to a single trusted computing domain, a single application environment, or to the CPU. Unlocking this interoperability is the Holy Grail of security for embedded systems.

3.1.1 Types of Attacks Addressed by This Guidance

As we described in Chapter 2, there are two general categories of potential attacks on devices.

Physical attacks entail tampering with a device to gain access to and control of its functionality, or to discover how a device works and clone it for other benefits.

Remote attacks entail accessing a device via a compromised communications path, such as its connection to the Internet or via inter-process communications between hardware elements.

Our Guidance to secure devices with a hardware-based multi-domain system mitigates the risk of remote attacks and makes it much more difficult to execute a physical attack.

Exploitation of a software-based vulnerability may originate remotely or via physical access to a device. With these attacks, the process entails reading pre-existing code and data in memory, and/or overwriting that code or creating new code in memory to take over the system. Hardware-based virtualization limits the amount of information that can be retrieved from the system if a malicious user breaks into a part of the system.

In our Guidance, the Root of Trust must not be removable from the system. This ensures that authentication for what the Root of Trust is meant to do can never be disabled. Also, the memories and storage used by the Root of Trust must be accessible only by the Root of Trust. This prevents tampering with authentication by the Root of Trust. With these requirements, physical attacks on the system are much more difficult to succeed.

Note that side-channel attacks, such as examination of heat signature or power consumption signatures require different security mechanisms. Separation of memory address spaces does not prevent these types of side attacks.

© 2016 prpl Foundation – All Rights Reserved.

3.1.2 Types of Separation for Embedded Systems

Our Guidance below describes pros and cons of four virtual separation technologies and dependent issues. The first three are variations on the use of hierarchical protection domains implementing "supervisor" and "user" modes – the architecture for which implements security policy at the hardware/firmware kernel, and enables a trusted operating environment for the rest of the system. Containerization implements separation solely by software.

- **Hardware-Assisted Virtualization** – Hardware virtualization creates a guest virtual machine on a platform; for embedded systems, the platform is a system-on-a-chip integrated circuit and any peripheral chips on the board. The virtual machine simulates a real computer and its operating system in order to host the operation of a guest operating system and applications. The platform's firmware that creates and controls a virtual machine is called a hypervisor or Virtual Machine Manager. The guest operating system and applications *can run unmodified* in a fully hardware virtualized system. For our Guidance, hardware-assisted virtualization is the "gold standard" for security by separation.
- **Paravirtualization** – Paravirtualization uses privileged software (the *hypervisor*) to virtualize the machine. Subsystems that may be used in more than one guest virtual machine must be managed by the hypervisor to separate accesses from different virtual machines. Since the hardware does not mediate access from multiple guests, paravirtualization *requires the guest operating system and programs to be modified* for the exact API to run in its environment. For our Guidance, paravirtualization is the "silver standard" for security by separation.
- **Hybrid Virtualization** – This implementation uses partially hardware-assisted virtualization with some paravirtualized devices. The reason for using hybrid virtualization is that most existing embedded system platforms do not have hardware-based virtual function devices for all subsystems that require this capability. Consequently, hybrid virtualization is likely to dominate implementations for the next five years or more until platforms evolve to meet security and business manufacturing/support requirements – particularly for deployment of millions or billions of consumer devices.
- **Linux Containers** – Containerization uses features of modern Linux kernels to provide separation of application environments from each other in a single operating system. Containerization extends familiar ideas such as separating users from each other through operating system controls. These controls may include user and group privileges, and operation of applications in a chroot "jail" with separated process groups, namespaces and access controls. The isolated systems, or containers, are created and controlled by a single Linux host. The chief advantage is that this approach is currently supported by platforms in high-volume consumer devices. Due to limitations described below, containerization is considered to be an intermediate step toward implementing security by separation with hardware. For our Guidance, the use of Linux containers is the "bronze standard" for security by separation.

© 2016 prpl Foundation – All Rights Reserved.

3.1.3 Opportunities for Virtualization and Secure Embedded Systems

Securing embedded systems with virtualization unlocks the ability to support functionality from multiple manufacturers and solution providers in a single system. Here are a few new possibilities:

- **Secure Environment for Multiple Providers**. Virtualization can provide a secure virtual machine that provides security services or acts as a "helper" to a rich execution environment such as a Linux host on one device. For example, secure video processing could be performed in a constrained Linux environment in its own guest virtual machine, digital rights management could be performed in a secure virtual machine running a Trusted Operating System (Trusted OS), and the main application with user interface, storage and other applications could be implemented in Android.
- **Multiple Deployable Secure Applications**. Virtualization can separate secure applications from each other in distinct secure virtual machines to avoid license, reliability and trust issues. It can also provide new models for service operators and providers in deploying their services on one device. Virtualization also keeps "separate things" separated and safe – both from being harmed by a failure or from harming other elements of the embedded system.

3.2 Challenges to Achieving Separation

To achieve our strategic goal of security by separation for embedded systems, hardware platforms must provide suitable capabilities to support virtualization. Examples of these capabilities include:

- Ability to enforce separation of memory and I/O resources.
- At least a 3-level CPU privilege state model for the main processor complex.
- At least a 3-level memory management unit (MMU) and page tables for hypervisor, guest/kernel, and guest/user.
- I/O virtualization for at least some peripherals.
- Virtual machine ID tagging of bus transactions.
- Network-on-chip port firewalls that gate communications between bus devices should also be aware of the identifier that is unique across all virtual machines (called VMID).

Without these capabilities, embedded systems will continue to be at risk because any device with its own processor programmed in RAM can potentially become a staging point for attacks. Also, subsystems with direct memory access engines addressing memories at the physical level are open to potential malicious manipulation. Both of these types of devices must either incorporate controls to force them to operate within the memory space of their virtual machines, or access must be paravirtualized to make the devices effectively be owned by the hypervisor. In the absence of a hypervisor user layer, it is common to allocate a host or "Domain 0" virtual machine to implement paravirtualized access to hardware subsystems at lower privilege than the hypervisor kernel.

© 2016 prpl Foundation – All Rights Reserved.

Another major practical requirement for embedded systems is more flexible, dynamic and secure inter-process communications (IPC). Manufacturers and developers need to be able to add and manage IPC channels between multiple virtual machines. These facilitate faster, lower latency and lower energy consumption communications between VMs than the network-based protocols used in data center and desktop virtualization systems. Embedded systems require IPC channels to be created and managed at runtime; currently, IPC channels are allocated and defined during configuration and require rebooting to enable more services. Finally, interoperability requires a naming schema and abstractions to address IPC channels. Perhaps this can be accomplished with publication and discovery of secure services on offer.

These are major architectural issues that must be resolved in order to achieve full hardware virtualization in embedded systems. In many cases, the solution approaches are known, but standard hardware designs do not yet include virtualization support. Achieving their resolution will take time. As noted, it is likely that a combination of hardware-assisted virtualization and paravirtualization will be normal for at least the next five years. This means manufacturers and solution developers must do the best they can with existing platforms and options for security by separation in embedded systems. For instance, secure separation of VMs can be achieved using hardware assistance; one approach to separating VMs is using an extra layer of hardware memory virtualization. Maintaining a consistent view of memory addressing across subsystems can simplify security analysis and reduce the potential for configuration errors allowing cross-VM leakage. Paravirtualization may still be appropriate for some types of subsystems. Finally, Linux containers also may be used as an interim step toward full hardware virtualization.

3.3 Approach #1 – Virtualization

Virtualization is a logical separation technology used to *simulate, isolate and control IT assets* such as hardware platforms, operating systems, storage devices and resources for networking. Virtualization allows the running of multiple systems and applications on a single platform. In the IT data center world, consolidation of infrastructure, easing manageability and reducing capital and operational costs are powerful benefits of virtualization. For embedded systems, virtualization is a clear path to “doing more with less” on millions or billions of globally deployed devices. While data center virtualization now separates the management of computing resources, storage and networking into distinct virtual domains that can be combined to create a complete computing environment, embedded virtualization will focus on sharing the subsystems of a single system-on-a-chip integrated circuit in separate VMs during the foreseeable future.

Virtualization also offers many benefits for embedded systems security. Separation is used to create and preserve a trusted operating environment. It is intended to prevent exploitable defects propagating beyond the environment in which the failure occurs. Separation allows an embedded system to process sensitive data securely on behalf of client applications, and to continue doing so if a peer VM is compromised. Separation also

© 2016 prpl Foundation – All Rights Reserved.

enables protection across and between all subsystems of a system-on-a-chip within a unified memory architecture. This means protection covers not only the CPU, but also graphics processors, audio and video processors, communications subsystems, and other hardware subsystems. Following are security ramifications of the four paths to virtualizing platforms for embedded systems.

3.3.1 Hardware-Assisted Virtualization

Hardware-assisted virtualization creates a virtual machine on a platform. The virtual machine simulates a real computer and its operating system in order to host the operation of a guest operating system and applications. The crux of this scheme that enables everything to work is strict control of shared memory from the point-of-view of guest system components that are accessing the actual underlying hardware platform. The CPU, graphics processor, and all other devices on the platform must have a consistent view of this memory and its accessibility from any particular guest VM.

The creation, allocation of resources and access rights to VMs, and management of guest virtual machines is the job of the hypervisor. A hypervisor may be monolithic, or may consist of a minimal kernel ("microkernel") and associated user layer. In embedded systems, we generally prefer a microkernel approach as the minimal complexity of a small piece of trusted software is easier to inspect for defects that may reduce security.

A critical function of the hypervisor is to allocate memory for all resources on the platform. If errors creep into memory management, there is a risk of device or application failures. As for security, the occurrence of memory management errors creates significant opportunity for malicious exploits of the embedded system. Effective memory management is the key to securing virtualized embedded systems.

Figure 3.1 below shows the different views on memory as seen by the hypervisor and guest VMs in translating virtual memory addresses to physical address in a four layer memory management scheme. The hypervisor microkernel operates at the root/kernel (highest) privilege level and controls all physical memory. The microkernel provides a very limited set of features: threads, memory management, inter-process communications and scheduling. Non-core hypervisor utilities provide memory allocation, driver paravirtualization, and other functionality. Each virtual machine starts in an apparent physical memory space allocated by the hypervisor. The guest VM level memory management unit and other memory management functions of the guest OS kernel are oblivious to the actual physical location of data in memory. User space allocations are managed normally by the guest OS kernel.

© 2016 prpl Foundation – All Rights Reserved.

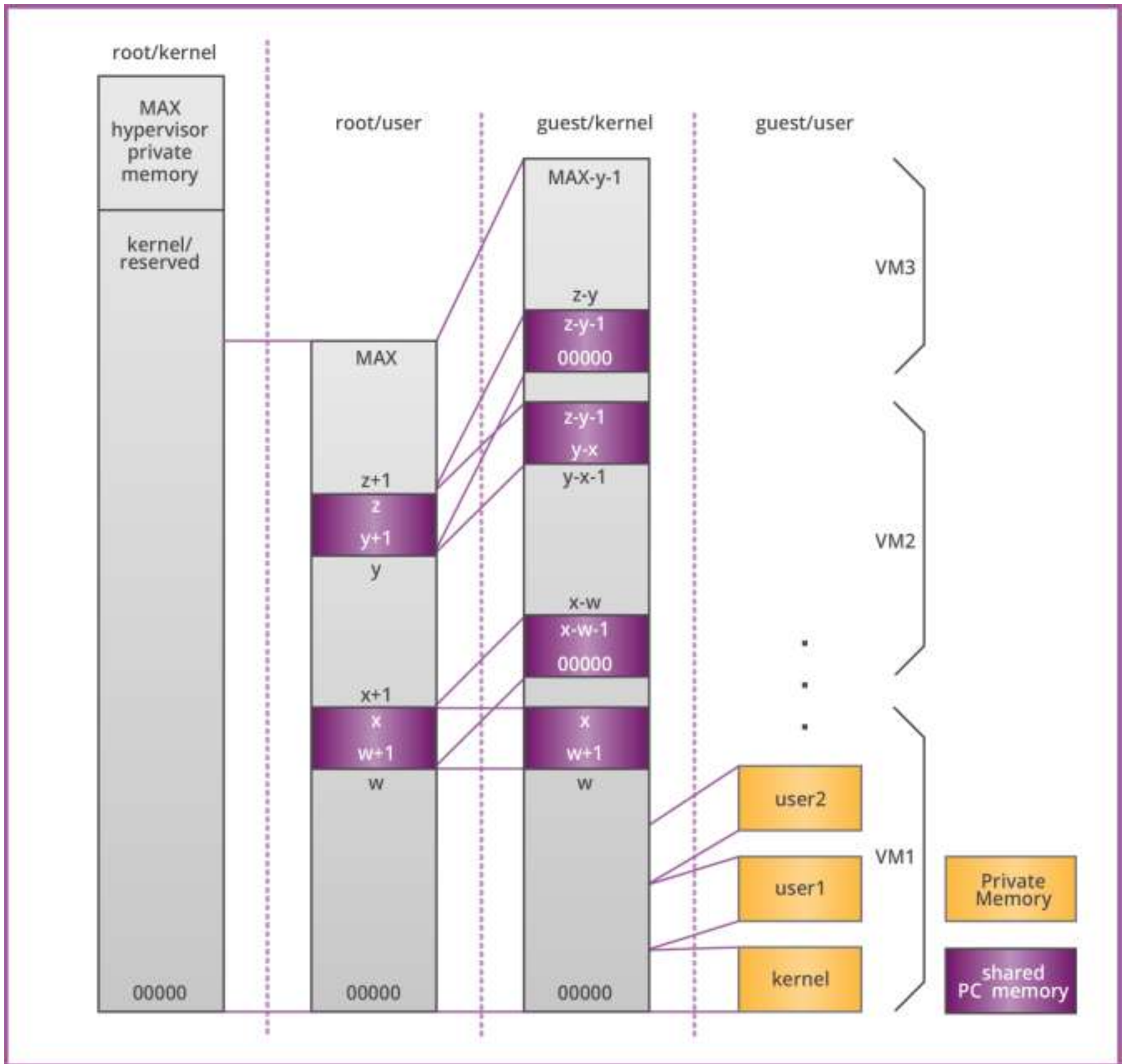


Figure 3.1: Translating Virtual Memory Allocations to Physical Memory

© 2016 prpl Foundation – All Rights Reserved.

Summary for Hardware-Based Virtualization

As mentioned in the introduction, one of the attractions to using hardware-based virtualization for embedded systems is that the operating system and device drivers can run unmodified, so there are no porting requirements. Also, CPU virtualization typically offers faster performance than other modes of virtualization. From a security perspective, the guest operating system is left unmodified, so it is harder to detect that it is operating in a virtual machine – and it is less susceptible to attacks even if the guest detects that it is operating in a VM. Also, hardware virtualized peripherals are not as dependent on the correct operation of the hypervisor to assure separation. The main caveat for hardware-based virtualization is that the hypervisor-managed second-level memory mapping is managed correctly to ensure each guest is separated from each other.

3.3.2 Paravirtualization

Paravirtualization allows a virtualization approach to be used on existing hardware that does not directly support virtualization in hardware. A fully paravirtualized solution requires changes to memory management, graphics, video, audio and display subsystems, and networking subsystems. These changes replace direct access to those functions by guest operating systems with hypercalls to the hypervisor to obtain access to the subsystems. The hypervisor mediates access on behalf of its virtual machines to maintain separation of resources between VMs.

Paravirtualization essentially relaxes the requirement to keep the programming model intact. Developers use a software application programming interface instead of hardware design changes to support multiple client VMs. Some of the benefits include:

- Simpler hardware design at chip level.
- No need to modify CPU privilege modes, such as the ability to run on hardware that does not support extra privilege levels

The downside is that paravirtualization requires the guest operating system and possibly some applications to be modified for the exact API to run in its environment. The result is a significant development, porting and configuration management commitment for guest VM operating systems. More security-critical software raises the potential for software or configuration defects to enable vulnerabilities in the embedded system. Paravirtualized memory management and additional device drivers will need to be back-ported to different versions of Linux or other guest operating systems. No modifications are required for a hardware-assisted virtualized environment, whereas guest operating systems must be modified for a paravirtualized environment.

This negative aspect does not mean a paravirtualized environment is de facto insecure. Toward this end, software developers are encouraged to systematically approach the use and verification of security-focused practices. One technique with the microkernel is to run operating system components as less-privileged user-mode applications, as shown in Figure 3.2. This reduces the trusted computing base operating at the highest

© 2016 prpl Foundation – All Rights Reserved.

privilege level for trusted applications. Typical hypervisors include the L4Re microkernel, PikeOS and seL4 microkernel.

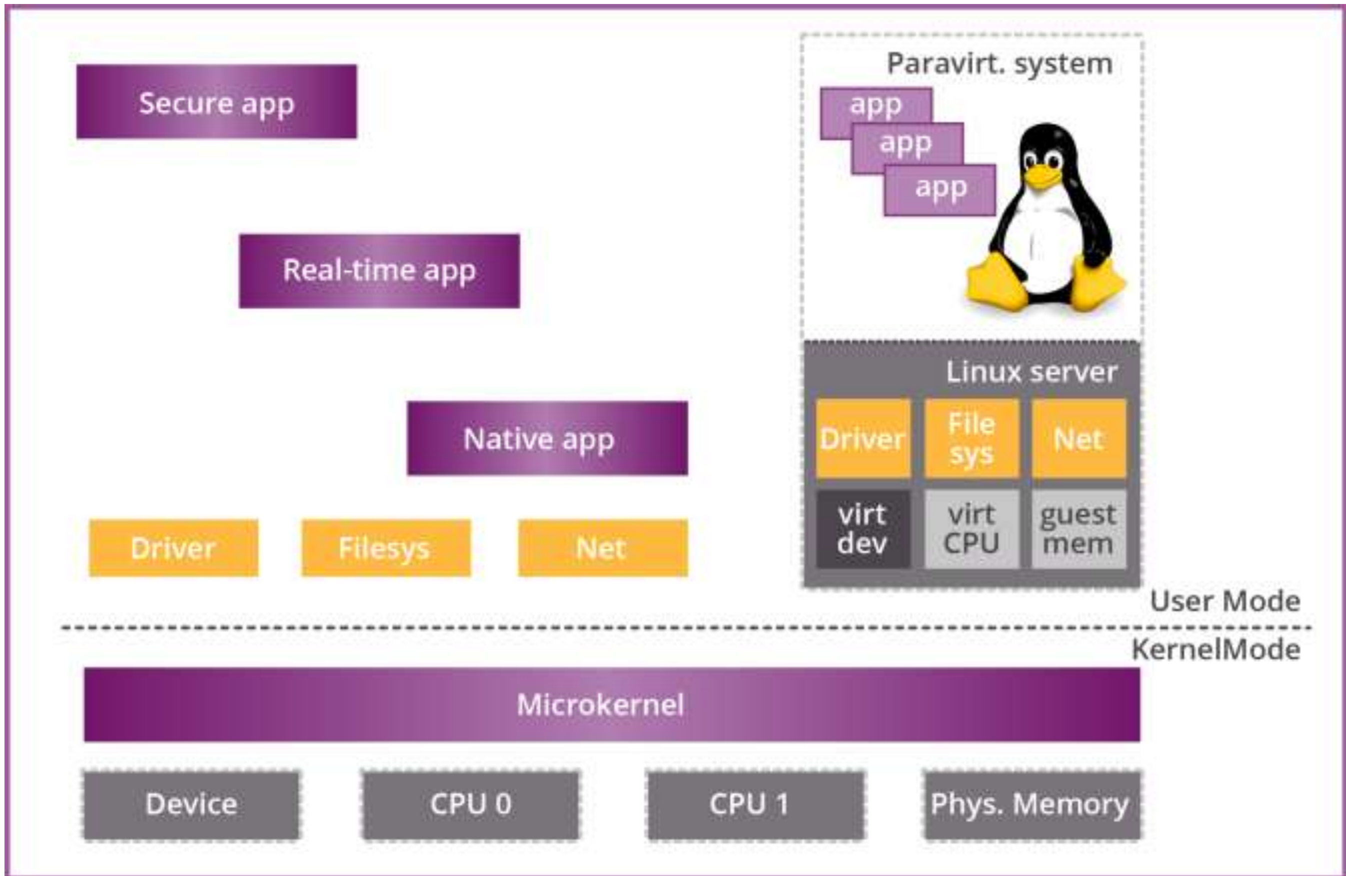


Figure 3.2: Microkernel-based System

Another technique is adapting the use of the microhypervisor for paravirtualization. In this mode, a developer would run the Virtual Machine Monitor as untrusted user-mode applications – even using one VMM per guest. Example systems include the L4Re microkernel and Nova microhypervisor.

When developing for a paravirtualized environment (or any trusted system), it's important to remember that complexity defeats security. Developers can fight complexity by removing untrusted systems from virtual machines in the trusted operating environment. They also can minimize an application's trusted operating environment by removing dependencies to unneeded components and isolating noncritical functions into secure compartments. Minimizing trusted operating environments helps achieve certification and verification of security.

© 2016 prpl Foundation – All Rights Reserved.

Summary for Paravirtualization

Embedded systems that are paravirtualized may offer some performance benefits because they offer a simpler, more direct interface than an interface emulating hardware. As a result, the system can produce faster I/O virtualization. They also can achieve a significant measure of security provided developers take special precautions to minimize the trusted operating environment and carefully design, implement and test APIs to ensure robustness and resistance to common failure modes such as buffer overflow. However, paravirtualization often requires modifying the operating system and device drivers along with initial and maintenance porting efforts. Securing paravirtualized systems is a challenge because the software stack is larger and fewer developers scrutinize these paravirtualized components than for e.g. the mainstream Linux kernel, and so may allow the introduction of latent vulnerabilities. Device drivers and other hypervisor dependencies mean the attack footprint may be larger than for hardware-assisted virtualization.

3.3.3 Hybrid Virtualization

As mentioned, a hybrid virtualization uses partially hardware-assisted virtualization with some paravirtualized devices. The reason for using hybrid virtualization is that most existing embedded system platforms do not implement the hardware hooks such as vitualizable network interface or GPU necessary for full hardware-assisted virtualization (see section 3.2). Consequently, hybrid virtualization is likely to dominate implementations for the next five years or more until platforms evolve to meet security and business manufacturing/support requirements – particularly for deployment of millions (or billions) of consumer devices.

3.3.4 Functional Requirements

Complexity is the enemy of security. With hardware-assisted virtualization, the nexus of security is the microkernel and its associated hypervisor – a small footprint operating system focused on simplicity in providing a platform with robust performance, security and isolation of services. Basic features include:

- **Execution of VCPUs** – the state required to support execution of a virtual machine.
- **Scheduling** – enabling virtual machines to access the physical processor cores and resources.
- **Memory management** – controlling how virtual machines share the physical platform's memory.
- **Inter-process communications** – IPC channels transfer messages between domains and are the cornerstone for inter-procedure calls.

The microkernel core system is responsible for starting and managing the allocation of runtime resources, such as execution time slots, to individual virtual machines. The hypervisor includes device driver code and APIs used by virtual machines to access it, memory allocation request handlers that VMs use to request access to memory, definitions of the runtime environments for virtual machines, and the virtual machine launcher that actually starts VMs.

© 2016 prpl Foundation – All Rights Reserved.

Following are examples of microkernels/microhypervisors available now for development and use with many platforms.

L4Re

The L4Re system is a third-generation microkernel-based hypervisor system supporting multicore systems and hardware-assisted virtualization. L4Re provides a framework for building multi-component embedded systems. Components include a client/server communication framework, common service functionality, virtual file system infrastructure, popular libraries, and the L4Linux multi-architecture virtualized Linux system. L4Re supports x86 and ARM platforms and is being ported to MIPS. Licensing is mostly under GPL v2 with other licensing options available.

The MIPS port includes single-source secure kernel services extensions with the ability to create, open and manage sessions between the remote execution environment and trusted execution environment. In addition, it provides the ability to open, manage and communicate through secure IPC channels. The port also provides a standard set of APIs for secure applications with services to remote execution environment client applications (see Figure 3.3). The secure kernel services are compatible with GlobalPlatform trusted execution environment specifications.[32]

[32] See http://www.globalplatform.org.

© 2016 prpl Foundation – All Rights Reserved.

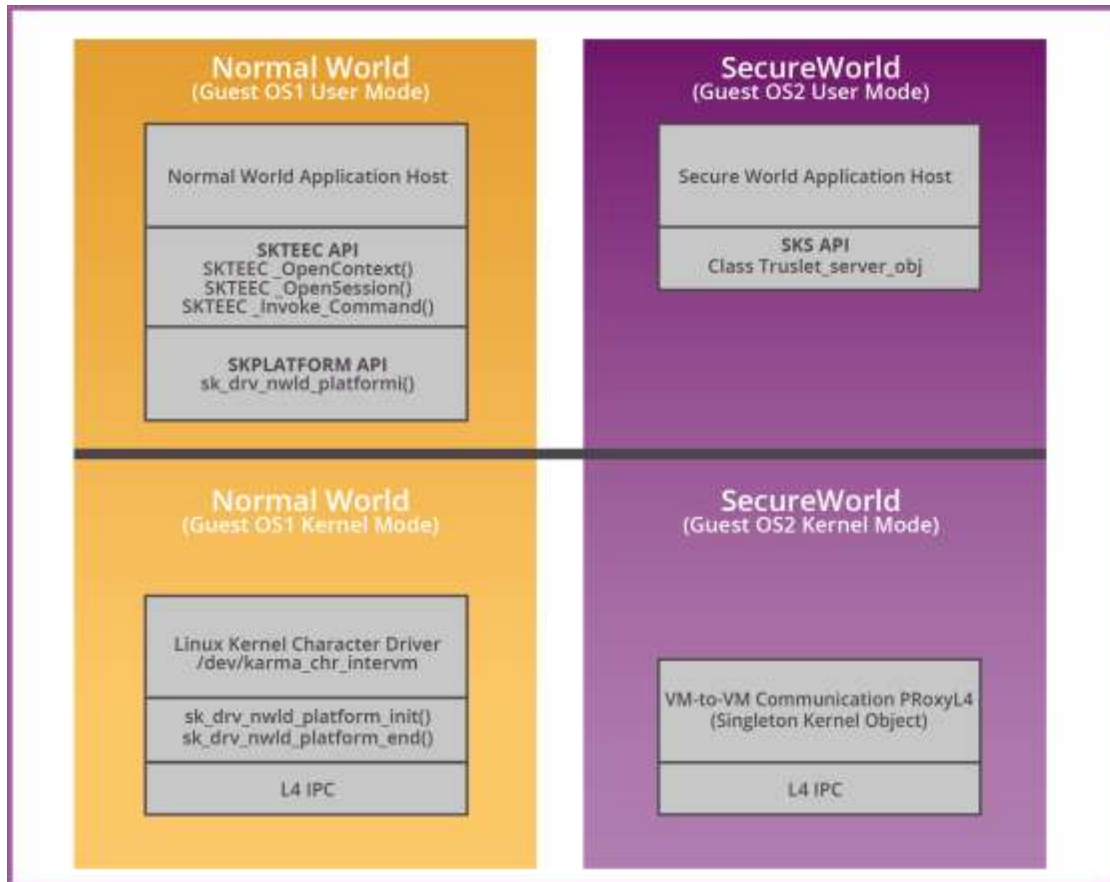


Figure 3.3: Secure Kernel Service Extensions to L4Re

For more information about L4Re, see http://l4re.org.

HellFire Hypervisor

HellFire Hypervisor is designed for the MIPS M5150 embedded processor. It provides a lightweight virtualization layer by supporting the MIPS VZ hardware virtualization extensions. HellFire enables real-time virtualization support and extended services such as system services, real-time management and inter-VM communication. It also supports HellFire OS (RTOS) with support for Linux as an ongoing effort.

HellFire has less than 7,000 lines of code in C language and assembly, supports up to eight guest instances, and provides full virtualization of the processor and para-virtualization for extended services. It also provides strong temporal isolation with an EDF algorithm for real-time VCPUS. As a work-in-progress, recent simulation testing

© 2016 prpl Foundation – All Rights Reserved.

shows optimistic performance results with low overhead for context switching and communication. HellFire is intended to be licensed as open code under a GPL model.

The HellFire project is from Pontificia Universidade Católica do Rio Grande do Sul (PUCRS) in Brazil. For more information contact Fabiano Hessel at Fabiano.Hessel@pucrs.br.

FEXER OX Hypervisor

FEXER OX Hypervisor supports both 32bit and 64bit MIPS and the MIPS VZ Module. The Type 1 “bare metal” hypervisor provides a small footprint (32KB) and minimal CPU power occupation (less than 1%.). Supported OS's include Linux, Android, QNX, T-Kernel/iTRON and others.

Achieving strong security for the operating system was a key design goal of FEXER OX. For example, in running Android and SecureOS simultaneously, security related processes and personal data handling are executed and handled by SecureOS. The architecture’s security mechanism within an embedded system is fully protected from free software that often poses risks of attacks by external hackers.

FEXER OX is used in the automotive market and for printing/imaging applications in Japan. It has been certified for ISO 26262 and other safety standards.

FEXER OX is proprietary and was developed by SELTECH. There is no obligation of source code disclosure to public. For more information, contact info@seltech.co.jp.

3.4 Approach #2 - Linux containers

Linux containers provide a virtualized environment for running multiple isolated Linux systems on a single Linux kernel. The isolated systems, or containers, are created and controlled by a single Linux host. For embedded systems such as those used in routers used in homes and small offices, a commonly used Linux distribution is OpenWRT with a fully writable file system with package management. Main components are the Linux kernel, util-linux, uClibc or musl, and BusyBox. It provides a framework enabling full customization of the device without having to build firmware around the application.

Linux containers provide isolation of the application’s view of the operating environment without requiring virtual machines. This open, software-only approach is supported by platforms used in high-volume consumer devices. It can provide faster performance than hardware-assisted virtualization or paravirtualization, requires a low operational overhead, fast startup, and quick provisioning of new containers.

From a security perspective, the Achilles heel of Linux containers is they all rely on protection of the Linux kernel. If the Linux kernel is breached, all of the embedded system’s containers, their applications, data and

© 2016 prpl Foundation – All Rights Reserved.

inter-process communications are vulnerable to exploits. Linux containers are fundamentally less secure than hardware-assisted virtualization because they rely on the monolithic Linux kernel for the whole platform security. To better protect shared resources, the Linux kernel requires better capabilities for denial of service prevention, prioritization of resource usage, memory management, and file system type and properties. Another challenge is updating the Linux kernel and its OS within containers comprised in millions or billions of deployed systems to correct vulnerabilities such as potentially compromising namespace isolation of containers. This update process alone may be practically insurmountable.

For more information about Linux containers, see https://en.wikipedia.org/wiki/LXC and https://linuxcontainers.org/. For more information about OpenWRT, see https://openwrt.org/.

© 2016 prpl Foundation – All Rights Reserved.

4.0 Secure Development and Testing

In rounding out our security guidance for embedded systems, stakeholders must consider how they will ensure secure development and testing – primarily of the firmware, and particularly of the early boot software and operating system. Emulating and debugging embedded systems is a challenging task, especially as embedded microprocessor cores grow more complex, performance requirements rise, and software grows larger. Debugging is an essential requirement for the ongoing security and stability of embedded systems.

4.0.1 Virtual Platform-Based Development, Debug & Test

Instruction-accurate virtual platforms provide a software simulation environment where the same executable code that would be run on hardware is run on the simulator. Put simply, the software should not know that it is not running on hardware. Models of the hardware are instruction-accurate only, and have no timing information. The virtual platform can be thought of as an Instruction Set Simulator (ISS), traditionally used for software development and test, but now extended to the full system-on-chip, or even the board or system.

Virtual platform-based environments have a number of advantages over real hardware, such as often being available before the hardware, and having much more controllability and observability. Virtual platforms are also deterministic; that is, simulation runs are repeatable. These advantages can be used for the development, debug and test of secure software, including hypervisors, secure operating systems and secure applications. If functional changes are needed to improve security in the hardware, related changes can be tested using this methodology. With these advantages, virtual platform-based methodologies have value for processor IP providers, system-on-chip developers, software vendors and the eventual embedded system vendors, although the use cases and necessary tools differ in each situation.

4.0.2 Joint Task Force Action Group (JTAG) Methodology

The predominant method used to debug embedded systems is an in-circuit emulator and debugger. This is a hardware device connected to the microprocessor with an interface based on the Joint Test Action Group (JTAG) methodology codified in 1990 in IEEE Standard 1149.1-1990, *Standard Test Access Port and Boundary-Scan Architecture.* JTAG enables external control of the microprocessor for debugging. It allows the loading and running of instructions and associated data registers, and starting and stopping of software with external tools. The debugger allows viewing the execution of code in the processor one instruction at a time.

© 2016 prpl Foundation – All Rights Reserved.

4.0.3 Extended JTAG for MIPS Reference Architecture

System software debug support in JTAG is provided by many chip vendors, tool vendors and open source tools for architectures such as ARM, MIPS, PowerPC and x86. Most provide extensions for particular platforms such as "halt mode debugging" and accessing registers and data buses without halting the processor core. In the context of this Guidance, references to JTAG refer to the Extended JTAG (EJTAG) Specification (Document No. MD00047). EJTAG is a hardware/software subsystem that provides comprehensive debugging and performance-tuning capabilities for MIPS microprocessors and for system-on-a-chip components that have MIPS processor cores. It exploits the infrastructure provided by the IEEE 1149.1 JTAG Test Access Port (TAP) standard to provide an external interface, and extends the MIPS instruction set and privileged resource architectures to provide a standard software architecture for integrated system debugging.

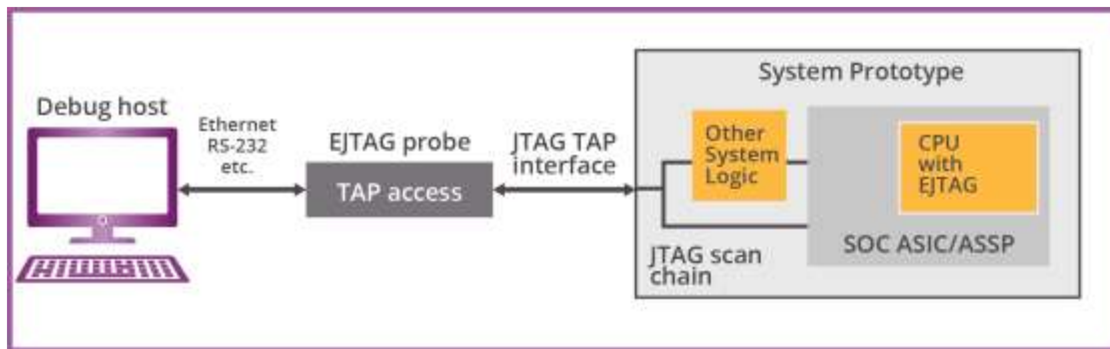


Figure 4.1: Setup of Debug System with EJTAG

EJTAG hardware support consists of several distinct components: extensions to the MIPS processor core, the EJTAG Test Access Port, the Debug Control Register, and the Hardware Breakpoint Unit. The figure below shows the relationship between these components in an EJTAG implementation. Some components and features are optional, and are implemented based on the needs of an implementation.

© 2016 prpl Foundation – All Rights Reserved.

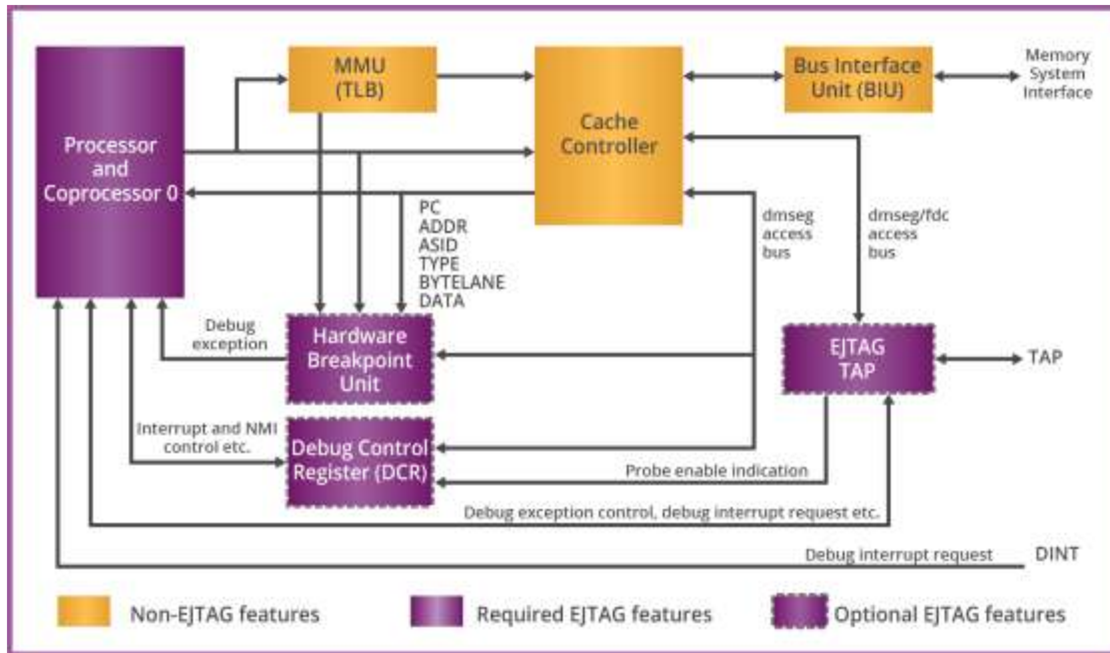


Figure 4.2: Simplified Block Diagram of EJTAG Components

4.1 Challenges to Secure Development and Testing

There are two fundamental issues that require resolution for secure development and testing of embedded devices and software.

Problem 1 – Software

When debugging at the hypervisor level (meaning the most privileged execution mode level, using a hardware debugger or a software debugger), can we restrict the debugging access to only a subset of the Guests?

Problem 2 – Hardware

When using a hardware debugger, how can we authenticate that the user has rights to debug the various parts of the system?

© 2016 prpl Foundation – All Rights Reserved.

4.1.1 Debugging for non-Virtualized Systems

For debugging systems without CPU virtualization, consider there are two execution modes: (1) kernel-mode with all the privileges and (2) user-mode without many privileges. Figure 4.3 shows process flow for each mode.

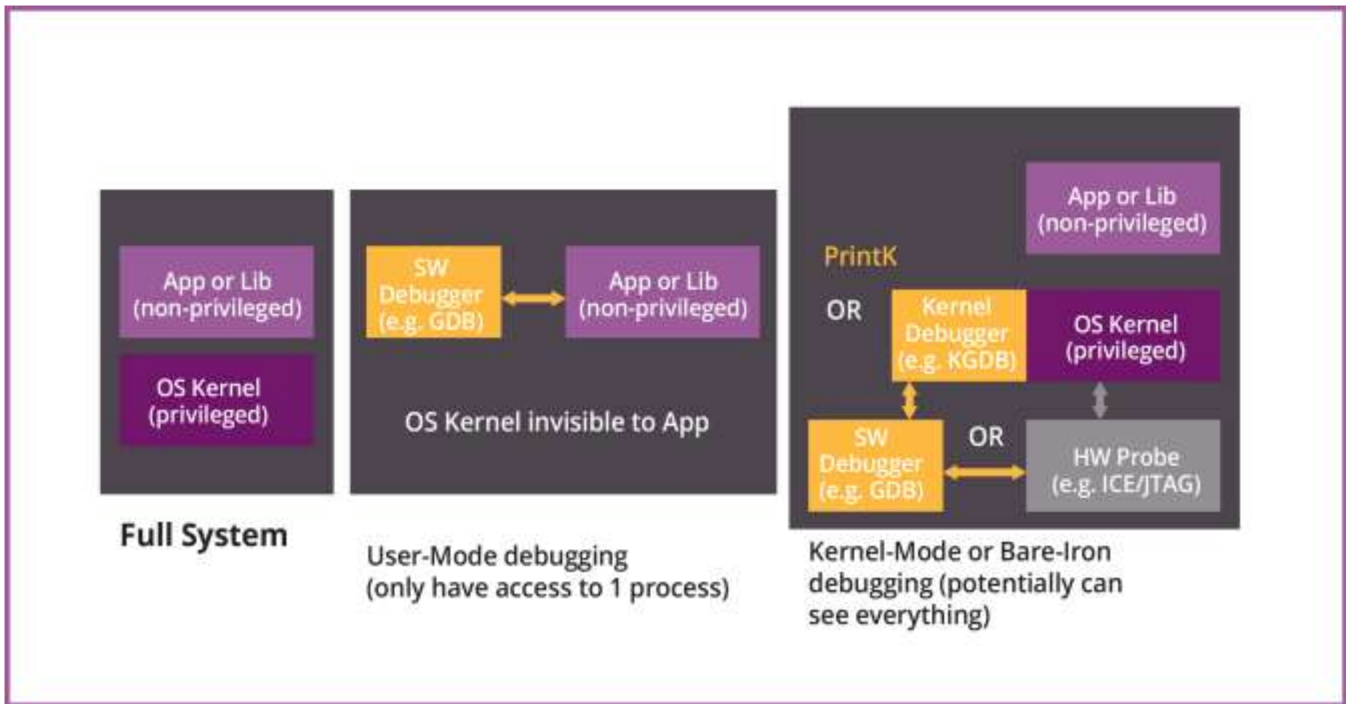


Figure 4.3: Debugging for non-Virtualized Systems

Software debugging typically employs user-mode applications or processes. The software debugger usually is GDB, the GNU Project debugger, which connects to the application/process. In this situation, one can only connect to processes that one has rights to. Access privileges are maintained by the operating system. For example, if there are two users, and User1 starts a spreadsheet application, User2 will be unable to connect to User1's particular invocation of the spreadsheet application.

Debugging the Operating System kernel typically employs print statements using print k. Alternately, the OS kernel might have a built-in debugger for connections (using something like GDB). Or, one can use in-circuit debugging (also called JTAG debugging or hardware debugging). In a MIPS environment, using the EJTAG debugger is a possibility that also poses a security issue discussed later in this section.

© 2016 prpl Foundation – All Rights Reserved.

4.1.2 Debugging for CPU Hardware Virtualization

In context of debugging CPU hardware virtualization, there are four execution modes instead of two. For example, with the MIPS VZ architecture, there is Root-Kernel; Root-User; Guest-Kernel and Guest-User. Figure 4.4 shows process flow for software debugging of guests.

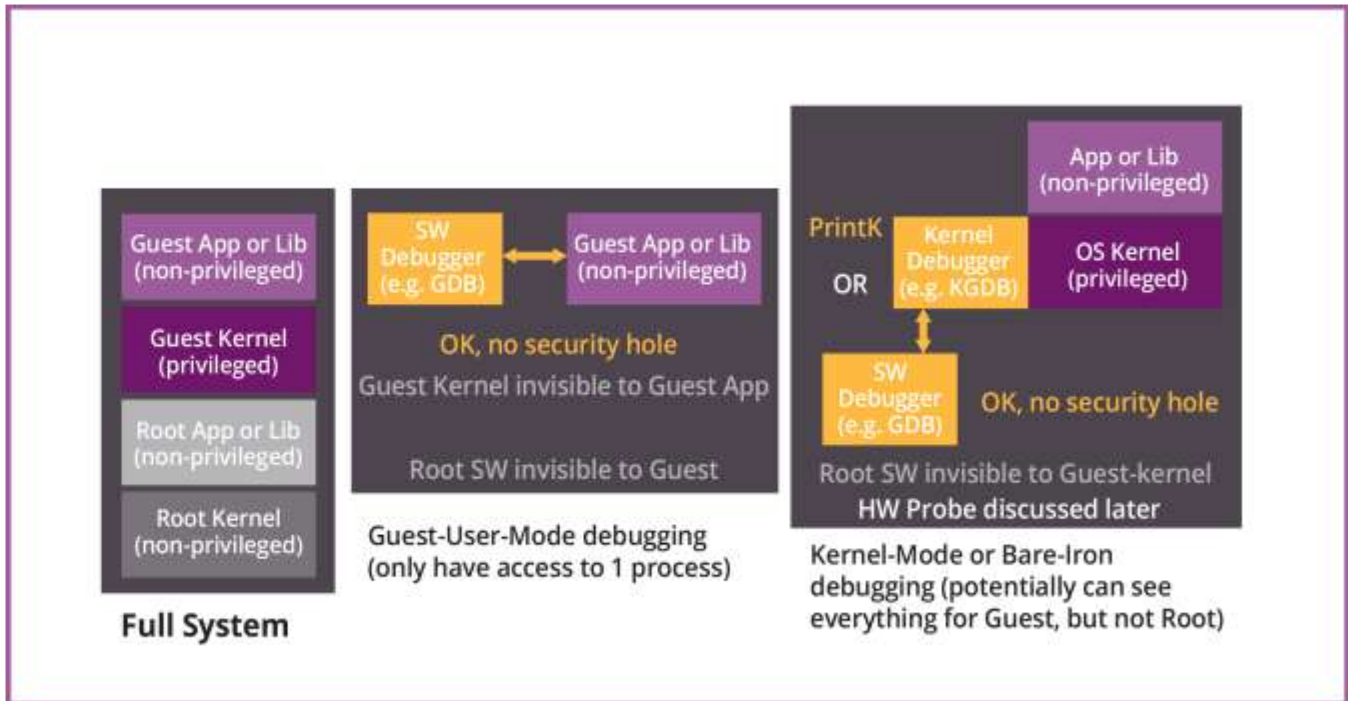


Figure 4.4: Software Debugging of MIPS VZ Guests

Debugging under Guest-User is the same as before. One only has access to the processes that one has rights to, which are maintained by the GuestOS. This presents no new security issues. Debugging under Guest-Root is also the same as before. One can only have access to that user's particular GuestOS resources. The other GuestOS addresses spaces are protected from debugging by the hypervisor. So again, there are no new security issues.

© 2016 prpl Foundation – All Rights Reserved.

4.1.3 Debugging for Root Execution Modes

Process flow for debugging under the Root Execution Modes for MIPS VZ is shown in Figure 4.5.

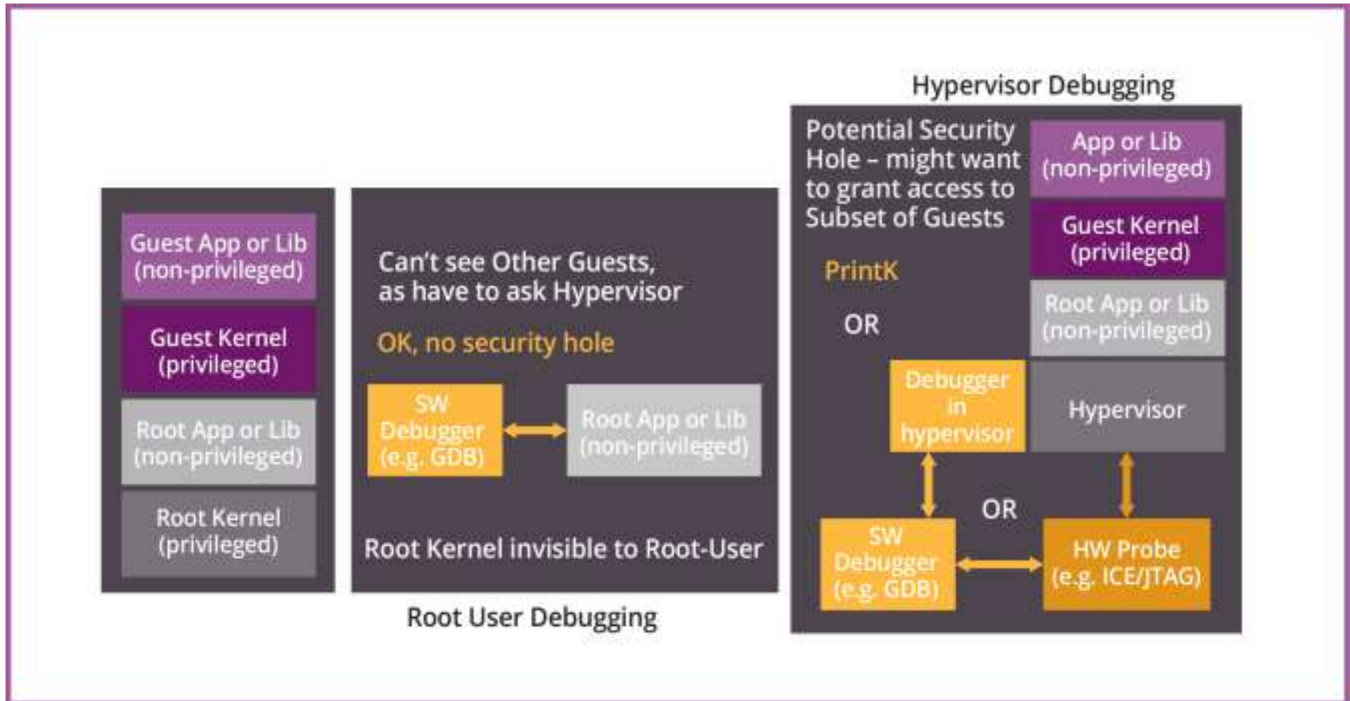


Figure 4.5: Software Debugging of MIPS VZ Root

When debugging under Root-User, it's the same as before: One only has access to the processes that one has rights to, which are maintained by the Hypervisor. This presents no new security issues. When debugging user Root-Kernel in the execution mode, one is meant to have access to everything. Having access to everything is the potential security issue.

If all of the software (meaning all of the GuestOSes) come from one company, then there is no problem. But what if Guest1 comes from CompanyA while Guest2 comes from CompanyB, and Guest3 comes from CompanyC? When CompanyA is debugging Guest1-OS, CompanyC might not want CompanyA to see everything inside Guest3-OS. Figure 4.6 shows a graphical example of this problem.

© 2016 prpl Foundation – All Rights Reserved.

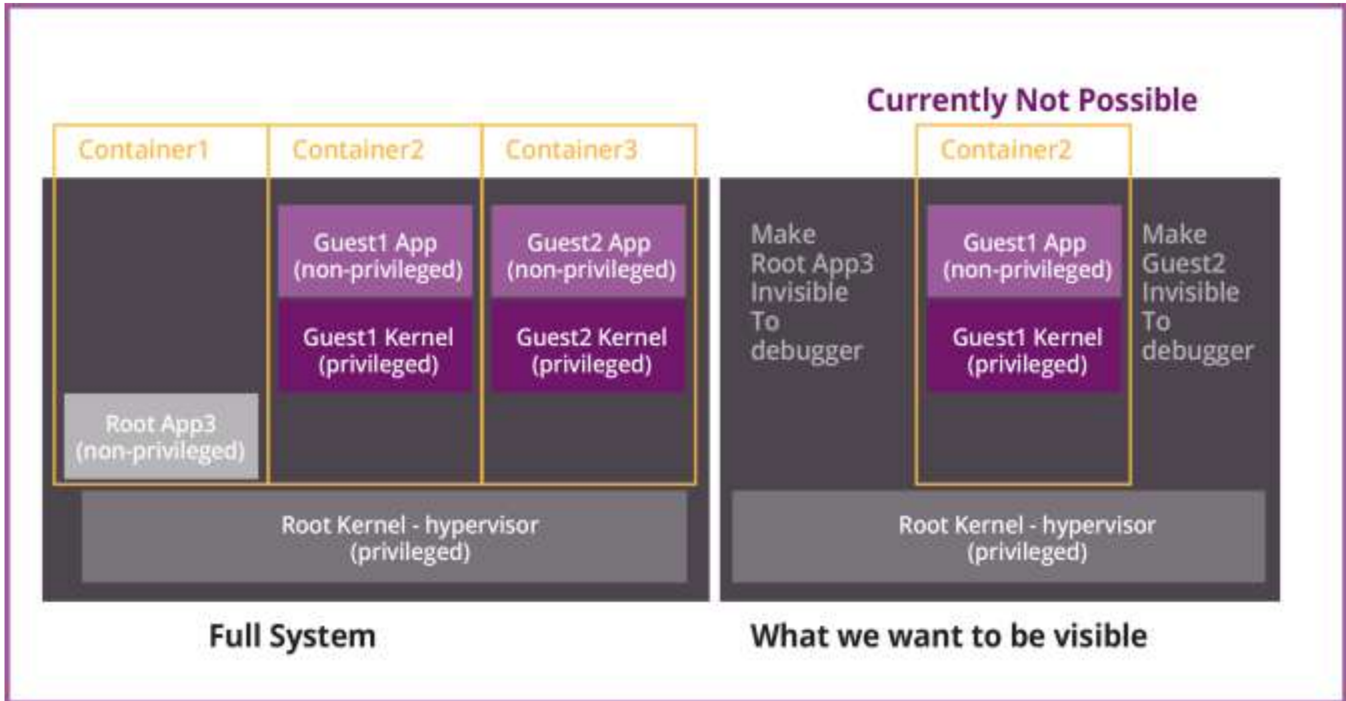


Figure 4.6: The Problem in Limiting Scope of SW Debugging of Root

On the left is the whole system with 3 containers. On the right is what we want to make visible to the debugger (just Container2). We want Container1 and Container3 to be hidden from the debugger. Currently, this is not possible when debugging at the Root-Kernel level. That is, there is no way to protect Container1 or Container3 from the debugger if we wanted to do so. This is the first security problem to be solved.

4.1.4 Ensuring Proper Authentication for Hardware Debugging

The second security problem is ensuring proper authentication for hardware debugging. Currently, the debug hardware probe controls the CPU. This means no software can prevent the probe from taking over the system, as shown in Figure 4.7.

© 2016 prpl Foundation – All Rights Reserved.

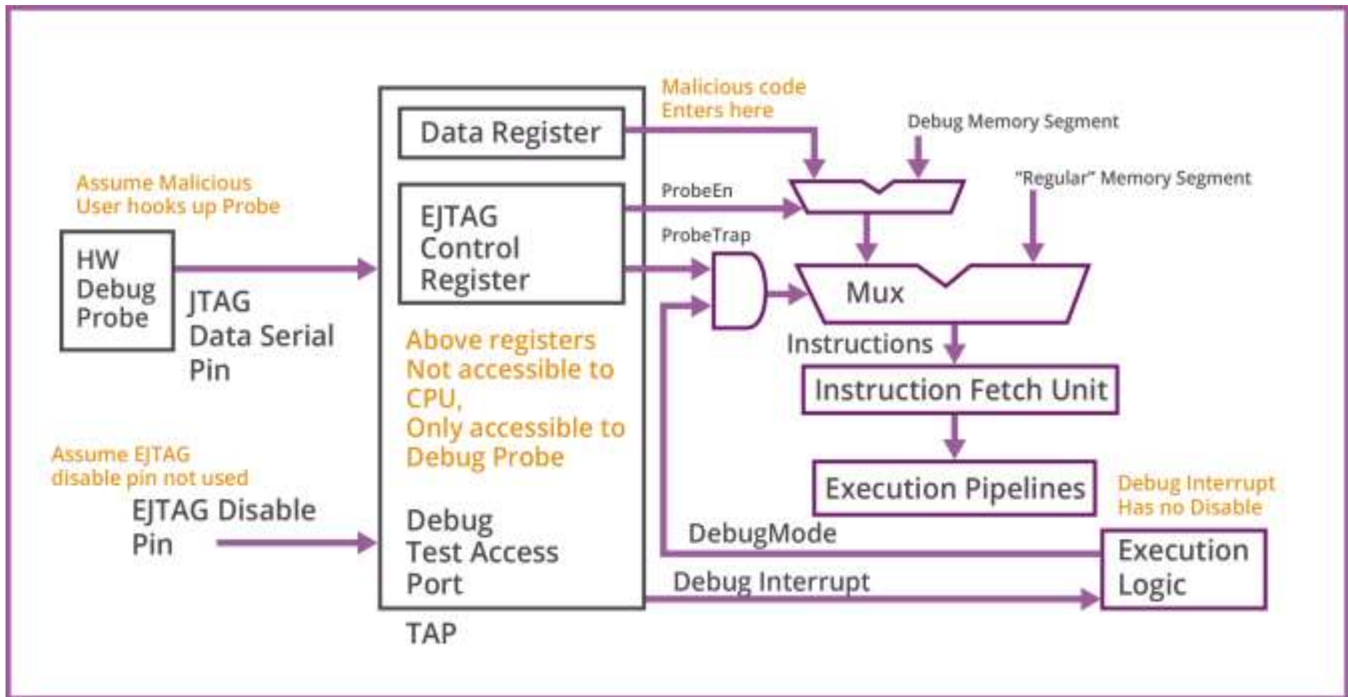


Figure 4.7: Debug Hardware Probe Controls the CPU

This diagram shows the logic of the EJTAG Test Access Port (TAP) on the left. On the right is a high-level representation of the CPU.

The TAP has a control register controlling where the CPU gets instructions – either "regular memory" or "debug memory" or from the TAP data register. This TAP register can be written by a hardware probe connected to the TAP, shown at the upper-left. The TAP data register is only writeable by the hardware probe. The TAP also has another control register that can send a Debug interrupt into the CPU. Again, this register can only be written by the hardware probe connected to the TAP, shown at the lower-left. These three TAP registers are only writeable/accessible to the hardware probe connected to the TAP. The CPU does not have access to these two control registers.

This situation allows a malicious user to take over the CPU once they have attached a probe to the TAP.

This is how the malicious user takes over the system:

1. The user has physical access to the system.
2. The user knows that the EJTAG disable pin is not asserted, that is, EJTAG debugging remains enabled.
3. The user hooks up a HW probe to the TAP.

© 2016 prpl Foundation – All Rights Reserved.

4. The user sets up the TAP control register so that the CPU gets its instructions from the TAP data register while in Debug Mode.
5. The user sends a Debug Interrupt into the CPU.
6. The Debug Interrupt is not maskable. That is, the CPU must take a Debug exception and enters Debug Mode.
7. Due to the setting of the TAP control registers, the CPU starts executing code from the TAP data register.
8. The instructions in the TAP data register comes from the HW probe which the malicious user is driving. Meaning the CPU starts executing code from the malicious user.

4.2 Requirements

Our proposed solution in this Security Guidance carries the assumption that a unique identifier be created for each GuestOS, called GuestID. Key elements of the proposed solution include:

1. The solution shall use multiple Digital Signatures that authenticate debug rights.
2. There shall be one signature and one public key to give rights to debug the hypervisor.
3. There shall be another signature and public key for each GuestID to give debug rights for that guest – e.g. A signature & key to debug Guest1, a different signature & key to debug Guest2, yet another signature & key to debug Guest3.

4.2.1 How It Works

Our proposed solution for embedded devices uses asymmetric cryptography with two private keys. For each GuestID, there are two keys. One key is burned into the device and is invisible to the outside world. The solution is implementation-specific whether the keys are unique for one system-on-chip die or are shared for all system-on-chip dies. The system-on-chip vendor must decide whether to do per-die fuses/programming.

There is also one plaintext signature for each GuestID. This signature is generated by a PRNG and is held in a register visible to the outside world. Each time the register is read, a new value is created by the PRNG.

The user reads the plaintext signature, which is held in a register that is visible to the rest of the system.

The trusted user is given the other private key and encrypts the plaintext signature.

The user sends the newly encrypted signature back to the CPU. This encrypted signature is held in another register.

© 2016 prpl Foundation – All Rights Reserved.

When the encrypted signature is written back to the CPU, the CPU is triggered to check the authenticity of the user. This encrypted signature is verified by the CPU using the private key. If the decrypted signature matches the original plaintext signature, then the user has gained the right to debug. If the decrypted value does not match the original plaintext signature, debug access is denied.

While the CPU is decrypting the signature, the instructions being executed by the CPU can never be exposed to the outside world. Doing so, would allow malicious users to get the private key and thus break the security system.

After each signature is verified, then the CPU keeps new internal state bits which say whether the user has rights to debug the associated SW entity (e.g. the hypervisor address space or Guest1 address space).

These internal state bits are reset to zero upon power-up and also some time after the probe is disconnected. Upon reset, the associated signature is either reset or updated with another value to prevent subsequent matches.

4.2.2 Lighter-Weight Alternatives

For systems, where one signature and one key per Guest is too-heavy weight, below are lower-cost options.

A middle-ground option beyond the minimalist hardware disable pin is using one signature and one key-pair to allow debug access to everything. This option is not per-Guest/per-Address-space access but rather all or nothing access. The purpose of this middle-ground option is:

- To avoid having to physically change something on the board, such as move a jumper or re-solder a wire.
- Increased security to prevent a malicious user from making the above physical changes to the system.

© 2016 prpl Foundation – All Rights Reserved.

5.0 Use Cases

In Chapter 3, we introduced the idea of security through separation. To be more explicit, this means the code for each function resides in its own address space and cannot access the address space or code of other functions unless the system explicitly allows for that access.

In older systems, such separation would be implemented with separate physical systems. For example, Function1 would be implemented by CPU1 with Memory1 and NonVolatileStorage1, which are not physically accessible by the sub-system that implements Function2 (with its own CPU2, Memory2, and NonVolatileStorage2).

For the systems proposed by our Guidance, we would implement each function sub-system as a virtual machine that shares the same physical resources (CPU, Memory, and NonVolatileStorage) for all functions. To ensure security, each virtual machine is restricted to its own addresses and cannot access the addresses used by the other virtual machines (unless specifically intended by the system design).

By using virtualization, the separation is at the Operating System level, not just at the application process level. In this manner, even if a malicious user breaks past the user level protection between applications and acquires control of the operating system, the attacker will not be able to access other parts of the system.

Following are a few examples of security by separation with virtual systems.

5.1 Protection of Media

In media-playing systems, software is often used to control Digital Rights Management and Conditional Access. Historically, software stacks for each function runs in one shared address space and has to trust each other. This trust is problematic if each software function is coming from a different company.

In the systems that we propose with address separation, each of these functions would run in its own address space or “container.” Separation of address spaces would allow different vendors to provide their respective software stacks on the same physical system. For example, the system might include WideVine DRM from one vendor, PlayReady DRM from a second vendor and Conditional Access software from a third vendor. One could place each of these stacks into its own “container” and each of these stacks would be protected from the others. If a malicious user gained control of one these systems, the attacker would be blocked from access to the other DRM/CA code.

Another use of virtual containers could place the entire video path for unencrypted video into its own container, which would be separate from the rich operating system where the user is requesting to play the video. This prevents any malicious user from looking at the memory to copy the decrypted video.

© 2016 prpl Foundation – All Rights Reserved.

5.2 Separation of Networks

Internet Service Providers are starting to use home Wi-Fi routers for two functions: one to enable a private network for the particular customer in whose home the router is located; the other is to support a "public" network for other customers who are passing by. Allowing strangers to use your home router is certainly a security concern for many people. The security of the system could be improved by moving each of these networks into their own container. The Wi-Fi hardware would be shared between the two networks by para-virtualizing the drivers in each of the containers. In this manner, even if an attacker took control of the public network at the operating system level, they still could not access the private network.

5.3 Separation of Secrets In a Product Supply Chain

Here is an example of protecting assets among the supply chain of companies producing a product. Implementing separation throughout the supply chain requires management and storage of keys among all participants.

Let's say a Power company is using smart meters to get better understanding of power usage by its customers.

The system-on-chip vendor might have a proprietary radio which is especially longer-range or lower-power than their competitors. The system-on-chip vendor wants to hide internal components of the radio to protect their competitive advantage. Their secrets include the software driver for the radio.

The Power company has confidential information about its customers usage of power. It also has secrets on how to access its communication network.

The Power company might allow third-party vendors to sell additional services based on information from the smart meters. For example, a third-party company might want to gather information about which customers are most appropriate for using solar cells, and then sell that information to solar installation companies.

The system-on-chip vendor wants to protect its radio secrets from the Power company as well as all of the potential third-party companies – and of course, any malicious user who tries to access the system.

The Power company only wants the third-party companies to be able to access its network through the approved mechanisms and not hack into its network through other means. The Power company would want each third party to collect only the types of usage information that is agreed upon between the two companies – and not access other additional information in the smart meter.

By keeping each third-party company's software running in its own separate container, the Power company can greatly strengthen the protection of secrets owned by companies using the smart meter.

© 2016 prpl Foundation – All Rights Reserved.

Appendix 1 - Potential APIs

A1.1 API's that Would Be New Due to Virtualization Being Used for Security

1. How to reach RoT crypto-functions from Guest-kernel
    a. Candidates – re-use Global Platform Internal Core API – Crypto section, TPM Chip API, NFC SecureElement
2. How to reach RoT crypto-functions from Root-User
    a. Candidates – re-use Global Platform Internal Core API – Crypto section, TPM Chip API, NFC SecureElement
3. How to instantiate GuestOSes - a standard way among hypervisors?
    a. Candidates – ARINC 653 partition management
4. How to authenticate user for Debugger
5. How to tell Debugger which Guest is running
6. How to do update?
    a. How to do update of the whole system?
    b. How to do update of just hypervisor?
    c. How to do update of just particular GuestOS?

A1.2 Second Level API's that Depend On Choice of Lower-Level Crypto-Functions

(meaning their internals would have to be unique for each system)

1. How to request authentication of memory block - potentially different for each system
2. How to request decryption of memory block - potentially different for each system

A1.3 API's that Exist Already for Secure Systems Without Virtualization

1. How to communicate from Guest/Container to another Guest/Container
    a. Candidates – Global Platform Client API, ARINC 653 interpartition communication
2. How to write TEE/TrustedOS apps interact with TEE/TrustedOS?
    a. Candidates - Global Platform Internal Core API
3. How do TEE/Trusted apps talk to the outside world?
    a. Candidates – Global Platform Sockets API

© 2016 prpl Foundation – All Rights Reserved.