How to Create a Secure Development Lifecycle for Firmware

Presented by UEFI Forum

October 23, 2019

Welcome & Introductions

Moderator: Brian Richardson
Firmware Ecosystem Development
Member Company: Intel Corporation
@intel_brian

Panelist: Eric Johnson
Manager – Engineering Manager /
Security Coordinator
Member Company: AMI

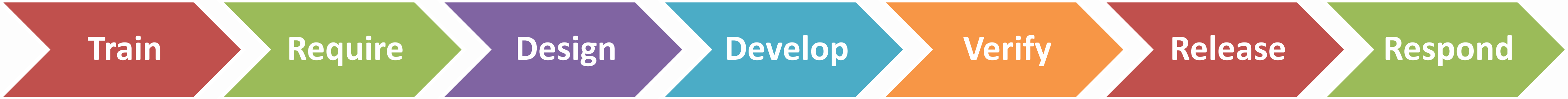
Panelist: Dick Wilkins
Principal Technology Liaison
Member Company: Phoenix Technologies

Panelist: Brent Holtsclaw
Security Researcher
Member Company: Intel

Secure Development Lifecycle (SDL)

Process for developing demonstrably more secure software, [pioneered by Microsoft]

Improves the capability to support, design, develop, test, and release secure software



Applying SDL to Firmware

Today we want to discuss how SDL can be applied to UEFI

This means understanding design elements unique to platform firmware, which are broken down into four major topics:

1. Secure Design
2. Secure Coding
3. Testing
4. Response To Security Vulnerabilities

As we cover these topics, please submit questions in the chat window. The panelists will take questions at the end of the webinar.

Secure Design… Where to Begin?

You can’t have a secure design unless you understand what your security threats are…

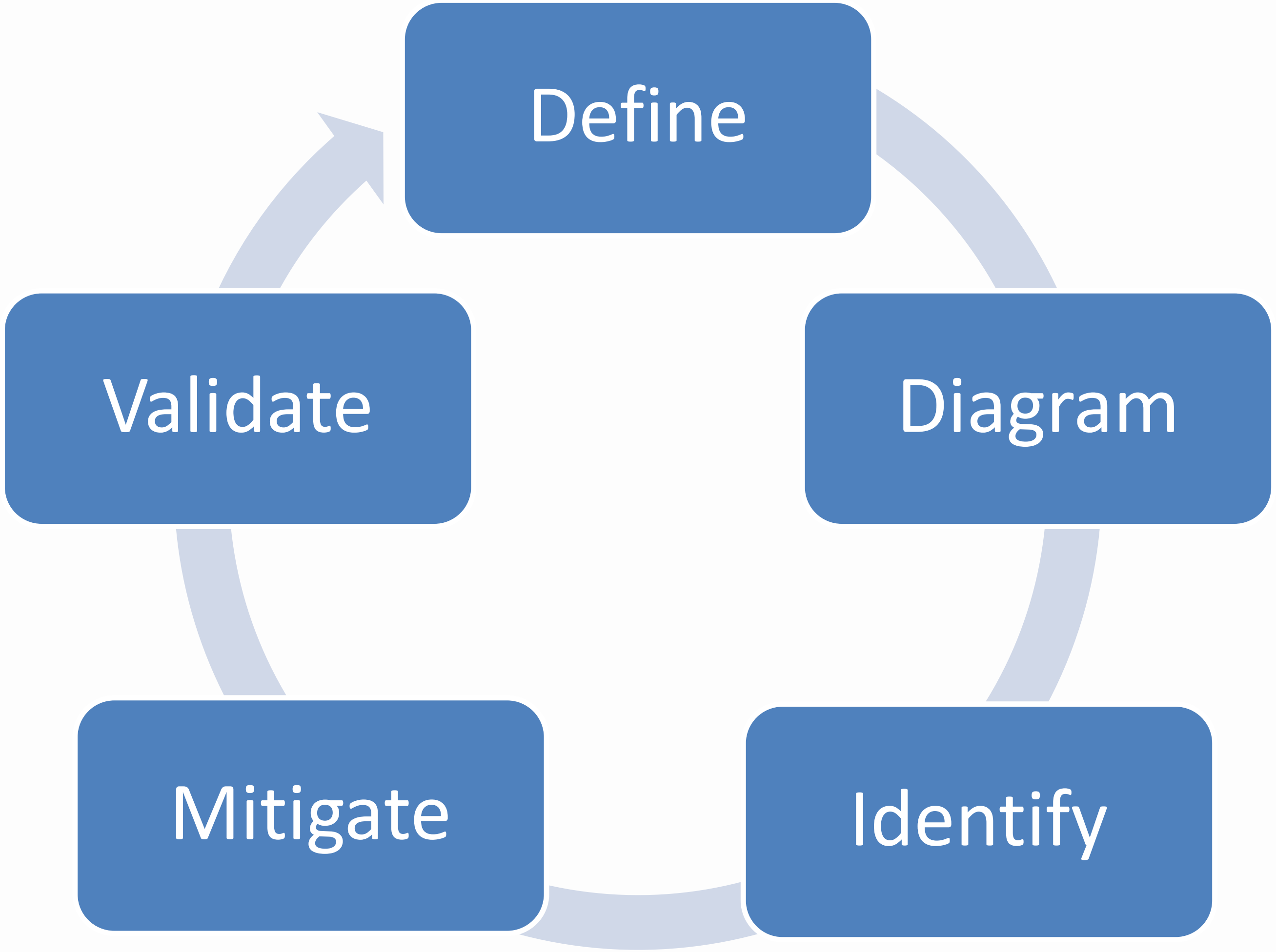
What is Threat Modeling?

Wikipedia: “Threat modeling is a process by which potential threats, such as structural vulnerabilities can be identified, enumerated, and prioritized – all from a hypothetical attacker’s point of view.”

Why Should You Threat Model?

- Firmware is an attractive target
  - Key link in chain of trust
  - Malware in firmware is invisible to host OS
- Firmware is rarely updated by end user
  - Attackers have years to find vulnerabilities in code
- Documented threat model useful for quality assurance, new hires, supplier audits, etc.

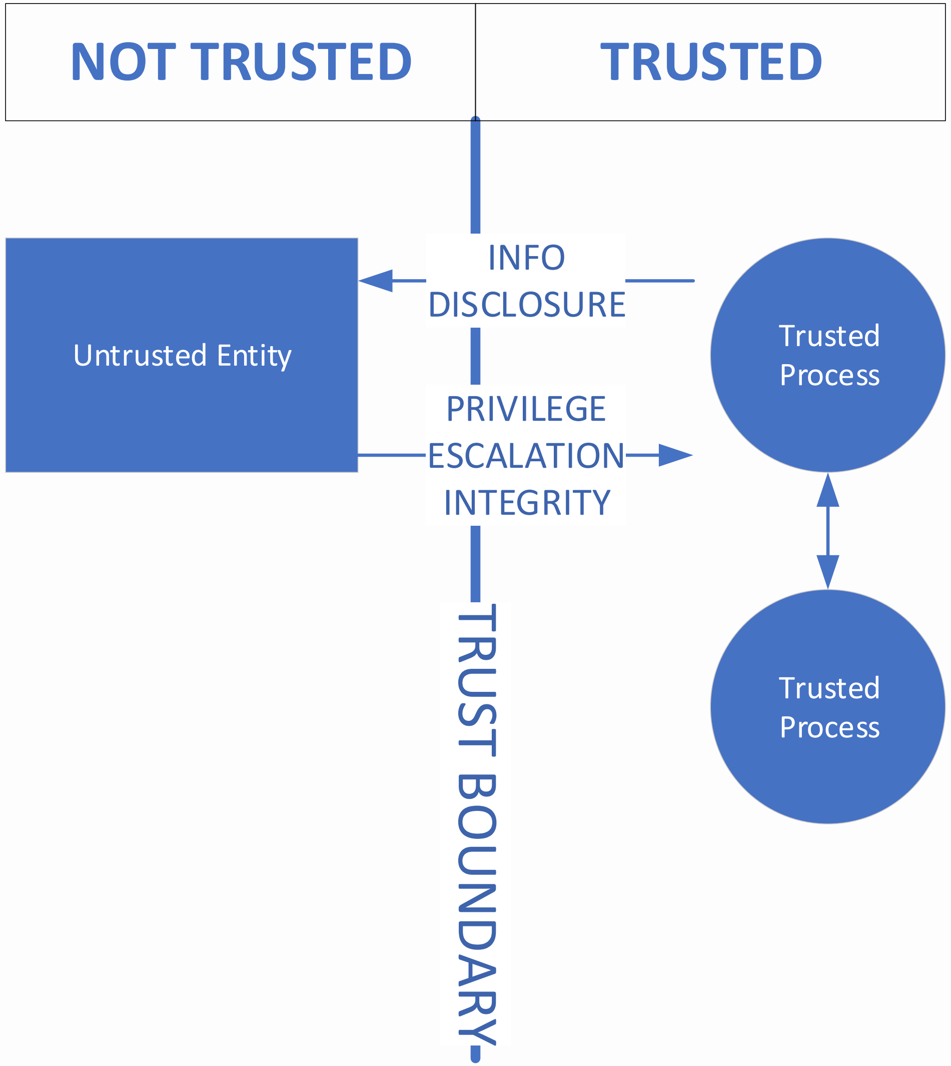
Threat Modeling Process



Define Security Requirements

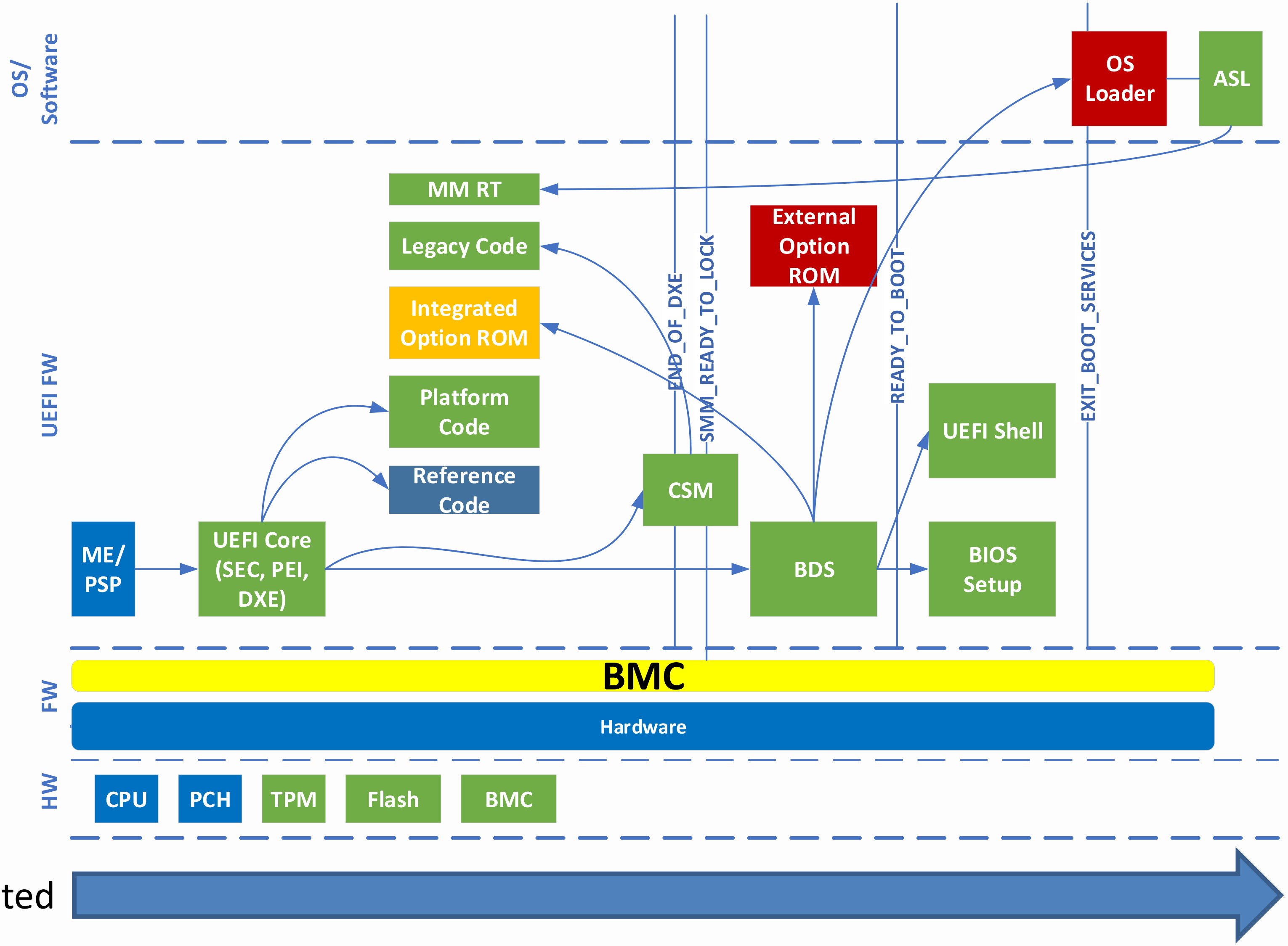
- Consider what the component does and how it fits into your platform
- Requirements may be functional, non-functional, or derived
  - Functional requirement defines what the system should do
  - A non-functional requirement puts constraints on how the system may do something
  - A derived requirement is not explicitly stated, but is necessary to fulfill derived or non-derived requirements
- A valid requirement must satisfy these questions:
  - Is it testable?
  - Is it measurable?
  - Is it complete?
  - Is it clear and unambiguous?
  - Is it consistent with other requirements?

Understand Trust Boundaries



From Wikipedia: a boundary where program data or execution changes its level of "trust"

Create a Platform Diagram



More Trusted

Less Trusted

Identify Threats

- Threats can be identified by analyzing the security requirements and platform diagram
- Threats should be categorized for further analysis
  - Techniques for analyzing threats: STRIDE, DREAD, PASTA, LINDDUN, etc.

Carnegie Melon University Threat Modeling Guide: https://insights.sei.cmu.edu/sei_blog/2018/12/threat-modeling-12-available-methods.html

STRIDE

Threat	Property Violated	Definition	Example
Spoofing	Authentication	Impersonating someone or something else	Pretend to be OEM, administrator, etc.
Tampering	Integrity	Modifying data or code	Modifying SPI part, S3 Resume Script, etc.
Repudiation	Non-repudiation	Claiming to have not performed an action	Claiming you did not physically open computer case
Information Disclosure	Confidentiality	Exposing information to an unauthorized user	User password left in memory
Denial of Service	Availability	Denying or degrading services to users	Preventing system boot or use of a resource
Elevation of Privilege	Authorization	Gain unauthorized capabilities	Allowing MM arbitrary code execution

Defense in Depth

- Provide complementary layers of security that work together to protect platform
- Compromising one layer does not allow the compromise of the entire system
- Example: Hardware root of trust + flash protection through MM + cryptographically signed firmware -> Remote attestation capability for auditing

Security Through Obscurity

- Firmware binaries are freely available online
- Tools to analyze binaries are available
- Security researchers are decompiling binaries
  - Most 3rd party reports received include disassembled code

Fail Safe

- Default platform configuration should be as secure as possible
- Avoid fail-open conditions where a specific value is used to enable security
  - This prevents degraded security by tampering with platform setup variables
- Corruption of platform configuration should not result in platform hang

Trust No One

- Use a hardware root of trust to protect against tampering
- Protect SPI access (both for NVRAM and firmware itself)
- Cryptographically measure and validate code before execution
- Lockdown MM before loading 3rd party code
- Validate all buffers / inputs into Management Mode
- Follow secure coding standards

Secure Coding… Common Problems?

Secure Coding

- Enemy #1, Buffer overflow/overrun
- Other common coding errors
  - Arithmetic over/underflows
  - Leaving manufacturing back-doors
  - Cryptography, poor choices
  - Time-of-check-time-of-use (TOCTOU) race conditions
  - Memory leaks

Reducing Attack Surfaces

- Reduce complexity
  - Remove unneeded features/services
  - Disable network ports/services that will not be used
- Study your threat model for opportunities
- Fuzz testing of required interfaces

Compiler Features

- Static analysis
- Runtime Checks
  - Stack cookies
  - Heap checking
  - No Execute (NX) data
- These features are available in the open-source Tianocore implementation but must be enabled
- If checks fail, make sure they don’t result in a DoS

Special Considerations for Firmware

- Special considerations for Management Mode (SMM, Trustzone, Ring -2 code)
  - MM code MUST never call code outside of SMRAM because an attacker could have maliciously modified that code
  - MM code MUST validate input parameters from untrusted sources to prevent buffer reads/writes that extend into SMRAM
  - MM code MUST copy input parameters and validate and use the copy, to prevent time-of-check-time-of-use (TOCTOU) vulnerabilities
- Because this code is so critical, special, in-depth code reviews are warranted

Special Considerations (Cont.)

- Secure Firmware Update
  - Don’t “roll your own.” Use common, open source update code whenever possible
  - Review custom implementations for vulnerabilities that have been found and fixed in the open source implementation
  - Enforce Signed Capsule Updates
  - Enforce Rollback Protection
  - Insure you are NOT using Manufacturing Mode for field updates
  - Use a Hardware Security Module (HSM) or Signing Authority for private key protection

Testing Firmware… How Hard Can It Be?

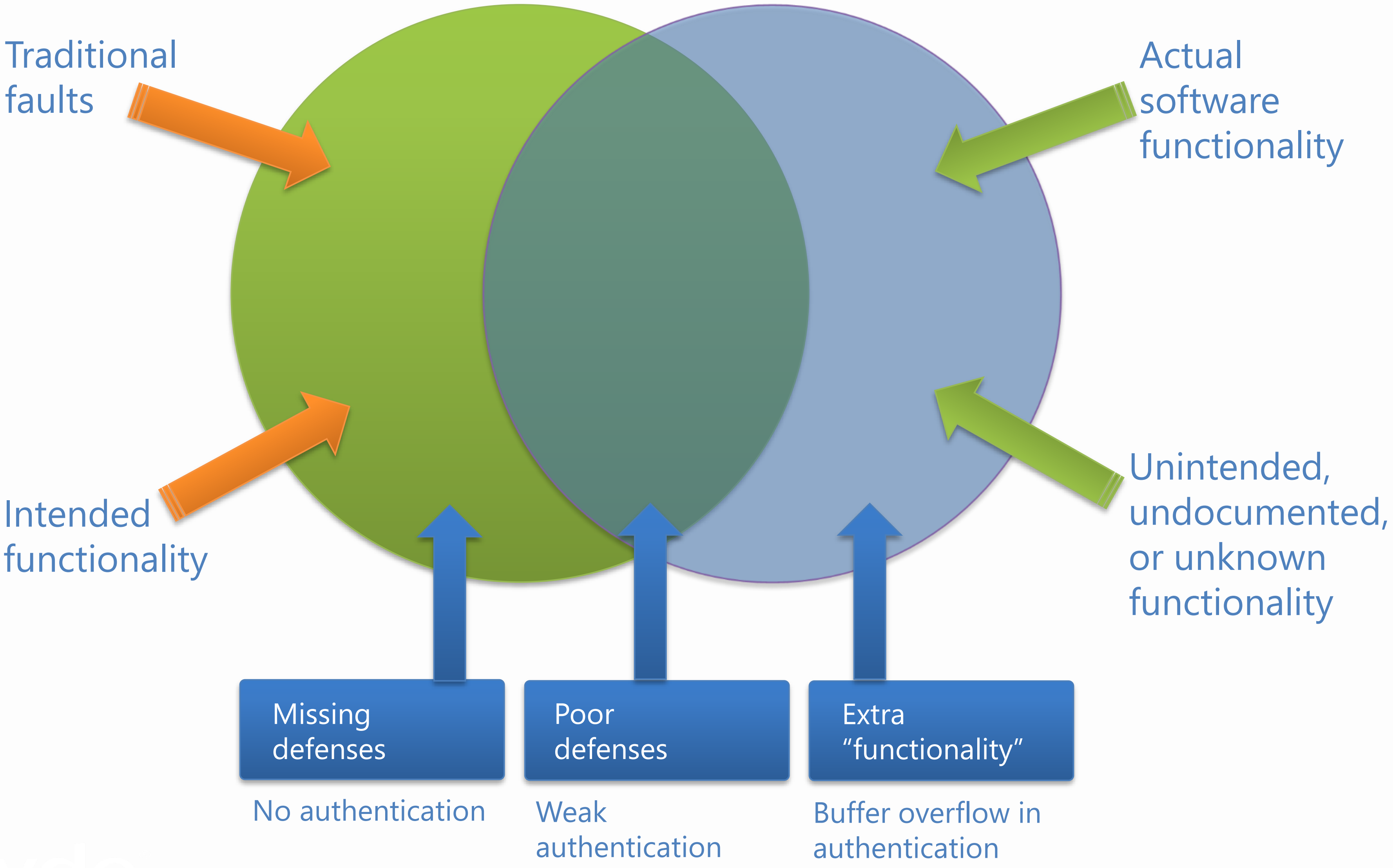
Testing Firmware is Different

- Limited methods of handling errors
    - Asserts lead to hangs
    - Many OS and compiler security measures are designed to lead to exceptions or unloading the driver
        - In firmware, these unloading a driver means not booting and exceptions mean hanging
        - This limits release usage of null-pointer detection, invalid access exceptions, stack and heap checking
    - A hang is a denial-of-service

- Logging and debug checks add code and lead to flash size issues

- Dynamic analysis tools don’t normally work with firmware out of the box

Security Testing



Traditional faults

Intended functionality

Actual software functionality

Unintended, undocumented, or unknown functionality

Missing defenses

No authentication

Poor defenses

Weak authentication

Extra “functionality”

Buffer overflow in authentication

Targeted Code Review

- Where did you get your code? What process do they use?
  - May increase/reduce need for additional review
  - Participate in the projects you use

- Identify high risk code
  - Threat modeling helps identify where weaknesses can lead to vulnerabilities
  - Smart people have written down their experience. Use it
    https://legacy.gitbook.com/book/edk2-docs/edk-ii-secure-coding-guide/details
    https://legacy.gitbook.com/book/edk2-docs/edk-ii-secure-code-review-guide/details

- High security risk is always a high review priority. What else?
  - Old code/new code – Assumptions kill
  - Code that runs with elevated privileges
  - Code with a history of previous vulnerabilities
  - Complex code
  - Code with a high number of changes

Top 25 Software Weaknesses

ID	Name	Score
CWE-119	**Improper Restriction of Operations within the Bounds of a Memory Buffer**	**75.56**
CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	45.69
CWE-20	**Improper Input Validation**	**43.61**
CWE-200	**Information Exposure**	**32.12**
CWE-125	**Out-of-bounds Read**	**26.53**
CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	24.54
CWE-416	**Use After Free**	**17.94**
CWE-190	**Integer Overflow or Wraparound**	**17.35**
CWE-352	Cross-Site Request Forgery (CSRF)	15.54
CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	14.10
CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	11.47
CWE-787	**Out-of-bounds Write**	**11.08**
CWE-287	Improper Authentication	10.78
CWE-476	**NULL Pointer Dereference**	**9.74**
CWE-732	**Incorrect Permission Assignment for Critical Resource**	**6.33**
CWE-434	Unrestricted Upload of File with Dangerous Type	5.50
CWE-611	Improper Restriction of XML External Entity Reference	5.48
CWE-94	Improper Control of Generation of Code ('Code Injection')	5.36
CWE-798	**Use of Hard-coded Credentials**	**5.12**
CWE-400	**Uncontrolled Resource Consumption**	**5.04**
CWE-772	**Missing Release of Resource after Effective Lifetime**	**5.04**
CWE-426	Untrusted Search Path	4.40
CWE-502	Deserialization of Untrusted Data	4.30
CWE-269	Improper Privilege Management	4.23
CWE-295	Improper Certificate Validation	4.06

https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html

Unit Testing

- Develop unit tests with modules/libraries
  - Use unit test to verify functionality
  - Update unit tests to catch regressions
  - Make engineering friendly!

- Enable unit test code as part of a Continuous Integration (CI) and Continuous Deployment (CD) process
  - Run unit tests as part of or triggered by patch acceptance process
  - Use unit tests to catch regressions early

- Use existing frameworks when possible
  - https://github.com/tianocore/tianocore.github.io/wiki/Host-Based-Firmware-Analyzer

Other Tools

- Static analysis tools

- Fuzzers throw convincing but garbage data at an interface.
  - Ex: LibFuzzer and AFL

- Code Coverage Tools check how much code was actually exercised when fuzzing.
  - May provide indications of dead code paths

- Hardware Setting Validators check hardware settings against most secure configuration.
  - Ex: CHIPSEC @ https://github.com/chipsec/chipsec

Testing Methodologies

- How is data validated
  - Malicious data
  - Sensitive data

- Look for known bad patterns
  - Improper type/size of data
  - Empty pass phrase
  - Test/Dev keys and certs
  - Previous coding errors found with the codebase

- Assume a high risk module/interface is compromised
  - Where could an attacker transition
  - What can an attacker enable/disable

- Evaluate security features
  - How are they enabled/disabled
  - How are they protected

Response To Security Vulnerabilities

Response To Security Vulnerabilities

- Have a plan and identified team to:
  - Root cause issues
  - Develop/deploy fixes
  - Inform customers/clients
  - Update your testing

UEFI Security Response Team (USRT)

UEFI Security Response Team

- Made up of members from UEFI Promoters and others
- Primary Goals:
  - Provides a point-of-contact for security researchers and others, to report issues and vulnerabilities to the membership of UEFI
  - Works with UEFI members to enhance and coordinate responses to actual and perceived vulnerabilities
  - Works closely with the TianoCore open-source community
- Please report vulnerabilities you find to the USRT:
  https://uefi.org/security or security@uefi.org

Summary

Summary

1. Understand the firmware threat model, and how it differs from other software
2. Write code with fewer complexities and smaller attack surfaces
3. When you test, think like an attacker
4. Have a plan for firmware updates and issue reporting

Q&A

Thank you for attending

For more information, visit uefi.org