Radix Coverage for Hardware Common Weakness Enumeration (CWE) Guide

Table of Contents
Introduction ........................................................................................................................................5
Using this Guide .............................................................................................................................5
Radix Security Rules ....................................................................................................................6
CWE-203: Observable Differences in Behavior to Error Inputs ..................................................7
CWE-226: Sensitive Information in Resource Not Removed Before Reuse ...............................8
CWE-276: Incorrect Default Permissions ....................................................................................10
CWE-440: Expected Behavior Violation .......................................................................................12
CWE-1053: Missing Documentation for Design ..........................................................................13
CWE-1189: Improper Isolation of Shared Resources on System-on-a-Chip (SoC) ......................14
CWE-1190: DMA Device Enabled Too Early in Boot Phase .......................................................16
CWE-1191: Exposed Chip Debug and Test Interface With Insufficient or Missing Authorization .............................................................................................................................18
CWE-1192: System-on-Chip (SoC) Using Components without Unique, Immutable Identifiers .................................................................................................................................19
CWE-1193: Power-On of Untrusted Execution Core Before Enabling Fabric Access Control .............................................................................................................................................19
CWE-1209: Failure to Disable Reserved Bits ...............................................................................21
CWE-1220: Insufficient Granularity of Access Control ...............................................................22
CWE-1221: Incorrect Register Defaults or Module Parameters ...............................................24
CWE-1223: Race Condition for Write-Once Attributes ..............................................................26
CWE-1224: Improper Restriction of Write-Once Bit Fields .........................................................27
CWE-1231: Improper Implementation of Lock Protection Registers .........................................29
CWE-1232: Improper Lock Behavior After Power State Transition ........................................30
CWE-1233: Improper Hardware Lock Protection for Security Sensitive Controls .................32
CWE-1234: Hardware Internal or Debug Modes Allow Override of Locks ...............................33
CWE-1240: Use of a Risky Cryptographic Primitive ..................................................................35
CWE-1241: Use of Predictable Algorithm in Random Number Generator ..............................36
CWE-1242: Inclusion of Undocumented Features or Chicken Bits ..........................................36
CWE-1243: Sensitive Non-Volatile Information Not Protected During Debug .........................38
CWE-1244: Improper Access to Sensitive Information Using Debug and Test Interfaces ...............................................................................................................................................39
| CWE-1245 | Improper Finite State Machines (FSMs) in Hardware Logic | 41 |
| CWE-1246 | Improper Write Handling in Limited-write Non-Volatile Memories | 42 |
| CWE-1247 | Missing or Improperly Implemented Protection Against Voltage and Clock Glitches | 44 |
| CWE-1248 | Semiconductor Defects in Hardware Logic with Security-Sensitive Implications | 44 |
| CWE-1251 | Mirrored Regions with Different Values | 44 |
| CWE-1252 | CPU Hardware Not Configured to Support Exclusivity of Write and Execute Operations | 46 |
| CWE-1253 | Incorrect Selection of Fuse Values | 48 |
| CWE-1254 | Incorrect Comparison Logic Granularity | 49 |
| CWE-1256 | Hardware Features Enable Physical Attacks from Software | 51 |
| CWE-1257 | Improper Access Control Applied to Mirrored or Aliased Memory Regions | 53 |
| CWE-1258 | Exposure of Sensitive System Information Due to Uncleared Debug Information | 54 |
| CWE-1259 | Improper Restriction of Security Token Assignment | 56 |
| CWE-1260 | Improper Handling of Overlap Between Protected Memory Ranges | 58 |
| CWE-1261 | Improper Handling of Single Event Upsets | 59 |
| CWE-1262 | Register Interface Allows Software Access to Sensitive Data or Security Settings | 61 |
| CWE-1264 | Hardware Logic with Insecure De-Synchronization between Control and Data Channels | 63 |
| CWE-1266 | Improper Scrubbing of Sensitive Data from Decommissioned Device | 65 |
| CWE-1268 | Policy Privileges are not Assigned Consistently Between Control and Data Agents | 67 |
| CWE-1269 | Product Released in Non-Release Configuration | 69 |
| CWE-1270 | Generation of Incorrect Security Tokens | 71 |
| CWE-1271 | Uninitialized Value on Reset for Registers Holding Security Settings | 72 |
| CWE-1272 | Sensitive Information Uncleared Before Debug/Power State Transition | 73 |
| CWE-1273 | Device Unlock Credential Sharing | 75 |
| CWE-1274 | Insufficient Protections on the Volatile Memory Containing Boot Code | 75 |
| CWE-1276 | Hardware Child Block Incorrectly Connected to Parent System | 77 |
| CWE-1277 | Firmware Not Updateable | 79 |
| CWE-1278 | Missing Protection Against Hardware Reverse Engineering Using Integrated Circuit (IC) Imaging Techniques | 79 |
| CWE-1279 | Cryptographic Operations are run Before Supporting Units are Ready | 79 |
CWE-1280: Access Control Check Implemented After Asset is Accessed .................. 81
CWE-1281: Sequence of Processor Instructions Leads to Unexpected Behavior (Halt and Catch Fire) ........................................................................................................... 82
CWE-1282: Assumed-Immutable Data is Stored in Writable Memory................... 83
CWE-1283: Mutable Attestation or Measurement Reporting Data ...................... 84
Appendix: Security Rule Types ........................................................................ 86
Legal Notices ..................................................................................................... 87
Introduction

MITRE's hardware Common Weakness Enumeration (CWE) database aggregates hardware weaknesses that are the root causes of vulnerabilities in deployed parts. A complete list can be found on the MITRE Hardware Design Webpage. Hardware CWEs are ideal to be used alongside internally developed security requirements databases and have been developed and submitted by both government and commercial design teams such as the Intel® Corporation and Tortuga Logic.

This guide can be used in conjunction with the CWE list as a resource to aid conversion from CWEs to Security Rules for use with Tortuga Logic’s Radix™ security verification tools. It also serves as a guide for design and verification teams to help them answer the question: “what security vulnerabilities should I verify?”

Using this Guide

In this guide each CWE is listed along with a template Security Rule that can be filled in with design-specific signals and used as a baseline test for the respective CWE.

Each Security Rule template is populated by placeholders with plain-language descriptions surrounded by curly braces, e.g. {{Placeholders in plain-language}}.

To transform these placeholders into valid Security Rules, replace the placeholder with design signals that match the description in the placeholder.

Because designs are diverse and CWEs apply generally, a bulleted list of mnemonic macros follow each template description that may be adapted to specific scenarios. Macros follow the pattern {{PREFIX}}_{{SUFFIX}}, where prefix maps to security objective (confidentiality, integrity, and availability) and the suffix maps to the asset to be secured (registers, access controls, general design elements, etc.)

Details about these macros are in tables in the Appendix at the end of this guide.

Each CWE section also contain an example illustrated using a hypothetical SoC which shows more specifically, using design signals how one can write one or several rules to verify the weakness is not present in the design.

To learn more about a specific CWE, follow the link in the CWE name at the beginning of the section referring to that CWE.

Tortuga Logic recommends that projects follow a Security Development Lifecycle (SDL) in addition to using security tools. Using this guide along with the MITRE Hardware CWE list, hardware security and development teams can take advantage of a 5-step process to streamline threat modeling and validation within their SDL, prior to committing a hardware design to silicon.
The 5-step CWE validation process to convert CWEs to Security Rules within the SDL:

1. Identify CWE(s) relevant to the threat model.
2. State plain-language security requirement identified in the CWE(s).
3. List the assets (in the form of data or design signals), objectives (confidentiality, integrity, availability), and security boundaries of the design as they correspond to step 2.
4. Use the Radix security rule template for the corresponding CWE in this document. Add design signals from step 3 to create security rules that can be validated with a hardware security verification environment, such as Radix™ from Tortuga Logic, alongside standard verification environments from Cadence®, Mentor® A Siemens Business, and Synopsys®.
5. Leverage the security verification environment to signoff that each CWE has been successfully checked.

This 5-step validation process provides a valuable bridge between the security engineers and architects who own the product security requirements, and the design and verification teams who are building and verifying proper functionality of the device. Often, Steps 1-3 are maintained by the security engineers and steps 4-5 are handled by the hardware design and verification teams.

Radix Security Rules

Security rules are expressed as information flow properties. They decouple the action from the observation. For example, the "action" is the activity that leads to the observable behavior. Using Radix Security Rules it is not necessary to specify the action, only that information should not flow from one location to another.

The core of the rule specification is the not flow operator (=/=>) which allows for specification of a source (signal or set of signals) to a destination (signal or set of signals). For example, a confidentiality rule related to a secret encryption key can be written as:

```
assert iflow(secret_key =/=> insecure_mem);
```

This rule states that the secret key should “not flow” or leak to an insecure memory where secret_key and insecure_mem are signals in the Verilog, SystemVerilog, or VHDL design. The not flow operator makes this specification easy and compact.

Optionally, specific conditions can be used to indicate when confidential information is being carried by design signals. These can be specified using the when keyword on the left-hand side of the no-flow operator (=/=>) along with a Boolean conditional expression based on design signals, such as when a Privileged-mode bit is set.
CWE-203: Observable Differences in Behavior to Error Inputs

Description

Differences in device behavior to an error input may be used by an attacker to gather security-relevant information about the device. The information may be as simple as whether a particular operation was successful.

Radix Security Rule Template

```plaintext
assert iflow (  
{{Signals carrying confidential information}}  
when ( {{Privileged-mode bit is set}} )  
=/=  
{{Signals visible to unauthorized actor}}  );
```

Rule Template Detail

Information should not flow from Signals carrying confidential information to Signals that can be observed by an unauthorized actor when a Privileged-mode bit is set.

Security Rule Types

- CONFIDENTIALITY_DATA
- CONFIDENTIALITY_SECURITY_STATE

Detection Example

The aes encryption engine uses two different length encryption keys and the time it takes to encrypt plaintext is different in the two cases. The untrusted cores core{0-N} may write data to sram and request it to be encrypted. When encryption is done, an interrupt is sent to the requesting core and the encrypted data can be read from sram. Verifying that the encryption done signal is asserted after the correct number of clock cycles after encryption is started is a required step in functional verification, but it
doesn’t verify that information doesn’t flow to a location where it is visible by an unauthorized actor.

Threat Model

By measuring the time encryption takes, an attacker can obtain information about which key was used.

Security Requirement

Different configurations of security sensitive operations should all take the same amount of time to avoid leaking information that can be used in a timing side channel attack.

Completing the Template Based on Design Signals and Security Requirement

From the requirement, the output of the encryption should not be available until after the longest encryption time. When \texttt{aes.done} is asserted, information from the \texttt{aes.csr.key} will flow to the \texttt{sram} because the encrypted data contains key information. This is OK as long as it doesn’t happen before \texttt{aes.done} is asserted. Radix security rules doesn’t support timing so a counter in the test environment, which is bound to the design, is used to count the number of cycles between \texttt{aes start} and done. Filling in the template gives the rule below.

\begin{verbatim}
assert iflow (
  aes.csr.key
  /==>
  aes.data_out
  unless (aes.done && aes.aes_counter_done) );
\end{verbatim}

**CWE-226: Sensitive Information in Resource Not Removed Before Reuse**

Description

When a device releases a resource such as memory or a file for reuse by other entities, information contained in the resource is not fully cleared prior to reuse of the resource.

Radix Security Rule Template

\begin{verbatim}
assert iflow (  
  {{Resource to be cleared}}
when (  
  {{Resource to be cleared}} != {{Value when cleared}} || ( ! {{Okay flag}} )  
  /==>
  {{Signals to be used by process}} );
\end{verbatim}
Rule Template Detail

Information from a resource *(Resource to be cleared)* must not flow to a new process *(Signals to be used by process)* when the resource does not equal the appropriate cleared value *(Value when cleared)* or clearing is not needed, indicated by the Boolean *Okay flag*.

*Value when cleared* will be based on design specifics. For example, if the *value when cleared* is the same as the secret value (e.g. the memory is zeroized but the secret data is "0") and the *okay flag* is incorrectly asserted, a violation will not be flagged so the rule needs to be adapted to the actual implementation.

Security Rule Types

- CONFIDENTIALITY_DATA
- CONFIDENTIALITY_REGISTER
- INTEGRITY_FSM_STATES

Detection Example

In the SoC design above, Hardware Root of Trust (HRoT) local SRAM *(sram)* has a privileged region bounded by *PRIV_END_ADDR* and *PRIV_START_ADDR*. In this example, we expect this region to be set to all zeros (zeroized) when the CSR *zeroize_status* within the tmcu.csr module is non-zero. In this example *zeroize_status* is non-zero only during zeroization, other values of zeroize_status indicate different stages of the zeroization process, and when *zeroize_status == 0* then zeroization has not been requested. Request of zeroization can be made via system software or anti-tamper circuitry. An exception is allowed when the *tmcu* is in privileged mode, indicated by the *tmcu.csr.priv_mode* register.

Threat Model

In this example we assume that threats are bugs or malice in parts of the system external to the HRoT-local SRAM (sram) that may obtain privileged information from uncleared privileged regions after a mode-switch.
Security Requirement

Information from the range $PRIV\_END\_ADDR:PRIV\_START\_ADDR$ must not leave the SRAM once zeroization has been triggered unless the range is zeroized, the tmcu is in privileged mode, or zeroization has not been requested.

Completing the Template Based on Design Signals and Security Requirement

Based on the security goal we fill in the template from the example signals, obtaining the rule below. The values for $PRIV\_END\_ADDR$ and $PRIV\_START\_ADDR$ are parameters provided by the user.

```
assert iflow (
  sram.mem[PRIV_END_ADDR:PRIV_START_ADDR]
  /=>
  sram.$all_outputs
  unless ( sram.mem[PRIV_END_ADDR:PRIV_START_ADDR] == 0
    || tmcu.csr.priv_mode || tmcu.csr.zeroize_status == 0 ) );
```

CWE-276: Incorrect Default Permissions

Description

During installation, installed file permissions are set to allow anyone to modify those files.

Radix Security Rule Template

```
// Confidentiality
assert iflow ( 
  {{Asset}}
  /=>
  {{User-accessible signals}}
  unless ( {{Permission bits}} == {{Value indicating asset should be readable}}) );

// Integrity
assert iflow ( 
  {{User-accessible signals}}
  /=>
  {{Asset}}
  unless ( {{Permission bits}} == {{Value indicating asset should be writable}} ) );
```
Rule Template Detail

Information from the Asset must be prevented from being read and/or written with data from user-accessible signals unless the permission bits are correctly set to the value indicating asset should be readable/writable.

Depending on security requirements, either one or both of the templates may be used. Note: Values for the permission bits needs to be obtained from the design specification.

Security Rule Types

- CONFIDENTIALITY_ASSET
- INTEGRITY_ACCESS_CONTROL_CONFIG
- INTEGRITY_ASSET
- VERIFY_ACCESS_CONTROL_CONFIG

Detection Example

Assume that security-critical settings and identifiers are available in a range of registers bounded by PRIV_END_ADDR:PRIV_START_ADDR in the CSR module of the tmcu. These values are locked or unlocked (e.g. in an authenticated debug mode) based on the tmcu.csr.lock_bit register in the same module.

Threat Model

We assume that untrusted software (the threat) has access to the input and output ports of the hrot_iface and may incorrectly read / write security critical information.

Security Requirement

Information from the address range PRIV_END_ADDR:PRIV_START_ADDR must not be readable or writable via hrot_iface unless tmcu.csr.lock_bit is zero (or a value that indicates it is not locked)

Completing the Template Based on Design Signals and Security Requirement

Based on the security goal we fill in the template which gives the rule below. The values for PRIV_END_ADDR and PRIV_START_ADDR are constants provided by the user.
assert iflow (  
  tmcu.csr.mem[PRIV_END_ADDR:PRIV_START_ADDR]  
  =/>  
  hrot_iface.$all_outputs  
  unless ( tmcu.csr.lock_bit == 0 ));

assert iflow (  
  hrot_iface.$all_inputs  
  =/>  
  tmcu.csr.mem[PRIV_END_ADDR:PRIV_START_ADDR]  
  unless ( tmcu.csr.lock_bit == 0 ));

**CWE-440: Expected Behavior Violation**

**Description**

A feature, API, or function does not perform according to its specification.

**Radix Security Rule Template**

*Note:* Due to the breadth of CWE-440 there are many rule variations that could detect unexpected use.

```plaintext
assert iflow (  
  {{Signal with next state}}  
 when ( {{Next state is security-sensitive}} )  
 =/>  
  {{Signal with current state}}  
 unless ( {{State transition criteria}} ));
```

**Rule Template Detail**

A commonly implemented solution to detect and prevent accidental misuse of a feature or API is raise an interrupt and set an error flag. The template is designed to detect finite state machine (FSM) issues that may result in a security rule violation if the FSM is forced into an incorrect state. In this template, *Signal with next state, Signal with current state,* and *State transition criteria* are all in reference to the target FSM. The conditional *when Next state is security-sensitive* denotes the state which the FSM should not enter unless the correct *State transition criteria* are met.

**Security Rule Types**

- INTEGRITY_FSM_STATES
Detection Example

In this example, core0 implements an error handler using the programmable interrupt controller module core0.pic. The module contains an FSM that may transition into a state that is intended to triple fault the core0 CPU and reset the device. If an attacker can force this FSM into the state that triggers the triple fault then they may be able to lock the device into a reset loop and cause a denial of service attack.

Threat Model

We assume that the attacker can send arbitrary commands to the core0.pic hardware via software or hardware running at the user privilege level.

Security Requirement

core0.pic.fsm must not enter the triple fault-triggering state unless authorization flags associate with the privilege level are set.

Completing the Template Based on Design Signals and Security Requirement

Based on the security goal, assuming user privilege is "1" and above, we fill in the template which gives the rule below.

```verilog
assert iflow ( 
    core0.pic.fsm.next_state 
    when ( core0.pic.fsm.next_state == 4'b1011 ) 
    #=> 
    core0.pic.fsm.current_state 
    unless ( core0.privilege == 0 ) );
```

CWE-1053: Missing Documentation for Design

Description

The product does not have documentation that represents how it is designed.
Radix Security Rule Template

Radix currently does not cover this CWE, refer to the MITRE website for suggested mitigations.

CWE-1189: Improper Isolation of Shared Resources on System-on-a-Chip (SoC)

Description

The product does not properly isolate shared resources between trusted and untrusted agents.

Radix Security Rule Template

```
// Confidentiality
assert iflow ( {{Shared signals}}
when ( {{Privileged mode condition}} )
=/> {{Signals readable by untrusted agents}}
);

// Integrity
assert iflow ( {{Signals writable by untrusted agents}}
=/> {{Shared signals}}
unless ( ! {{Privileged mode condition}} ));
```

Rule Template Detail

A shared resource (*Shared signals*) must not influence *signals readable by untrusted agents* when the resource is in a *privileged mode condition*. Additionally, the shared resource must not be able to be influenced by *signals writable by untrusted agents* unless the resource is not in the privileged mode condition.

Security Rule Types

- ISOLATE_ASSET
- VERIFY_ACCESS_CONTROL_CONFIG
Detection Example

The Hardware Root of Trust (HRoT) local sram is memory mapped in the core{0-N} address space. The HRoT allows or disallows access to private memory ranges, thus allowing the sram to function as a mailbox for communication between untrusted and trusted HRoT partitions.

Threat Model

In this example, we assume that the threat is from malicious software in the untrusted domain. We assume this software has access to the core{0-N} memory map and can be running at any privilege level on the untrusted cores. The capability of this threat in this example is communication to and from the mailbox region of SRAM modulated by the hrot_iface.

Security Requirement

Information must not enter or exit the shared region of SRAM through hrot_iface when in secure or privileged mode.

Completing the Template Based on Design Signals and Security Requirement

From the requirement, if the priv_mode control and status register (CSR) within the tmcu is set to 1'b1 this corresponds to a secure mode when the privileged memory should not be accessible. The privileged range in the SRAM model is bounded by SHARED_START_ADDR and SHARED_END_ADDR. The values for SHARED_END_ADDR and SHARED_START_ADDR are constants provided by the user. Filling in the template gives the rule below.

```verilog
assert iflow (sram.mem[SHARED_END_ADDR:SHARED_START_ADDR]
when ( sram.csr.priv_mode )
=>
hrot_iface.$all_outputs );

assert iflow (hrot_iface.$all_inputs
```
=/>=

sram.mem[SHARED_END_ADDR:SHARED_START_ADDR]
unless (! sram.csr.priv_mode);

Note: The privilege mode bit used in the rule refers to privilege level of the resource being accessed. If the privilege mode is changed, information may propagate, and the rule be violated.

CWE-1190: DMA Device Enabled Too Early in Boot Phase

Description

The product enables a Direct Memory Access (DMA) capable device before the security configuration settings are established, which allows an attacker to extract data from or gain privileges on the product.

Radix Security Rule Template

```
assert iflow ( 
  {{DMA data input}}
  =/>=
  {{Data-carrying signals in the DMA controller}}
  unless ( {{Security settings are set}} ));
```

Rule Template Detail

The DMA data input must not flow to the Data-carrying signals in the DMA controller unless the security settings are set. This is not identical to the CWE description because enabling the DMA could be done in several ways. However, in each scenario where the CWE applies the security goal is for the DMA not to act on any data prior to setting up the security configuration to modulate DMA access.

This rule template covers many scenarios by triggering a rule violation when the DMA may move data prior to the system being in a secure state.

Security Rule Types

- INTEGRITY_ACCESS_CONTROL_MECHANISM
- VERIFY_ACCESS_CONTROL_CONFIG
Detection Example

The un-trusted dma in the SoC example may be enabled by one of the un-trusted cores core[0-N] and may access the entire memory range of the SoC before the security configuration is set up by the Hardware Root of Trust (HRoT).

Threat Model

An attacker running un-privileged code may set up the dma to read and write protected resources such as sram in the HRoT before the security policy is configured and thus access sensitive data.

Security Requirement

The un-trusted DMA must not be active until the full secure boot sequence is complete. This means data in the dma must not flow out of the dma and data must not flow into the dma until secure boot is complete.

Completing the Template Based on Design Signals and Security Requirement

From the requirement, data must not flow into or out of the dma until boot is complete. Filling in the template gives the rules below.

```plaintext
assert iflow ( dma.data_reg /=> dma.$all_outputs
unless ( tmcu.csr.boot_stage >= FULL_BOOT ) );

assert iflow ( dma.data_in /=> dma.data_reg
unless ( tmcu.csr.boot_stage >= FULL_BOOT ) );
```
CWE-1191: Exposed Chip Debug and Test Interface With Insufficient or Missing Authorization

Description
The chip does not implement or does not correctly check whether users are authorized to access internal registers.

Radix Security Rule Template

```plaintext
assert iflow (  
{{Internal signals}}
=/=>
{{Signals readable by untrusted agents}}
unless ( {{Debug is enabled}} ));
```

Rule Template Detail
Internal data should not flow to signals readable by untrusted agents unless Debug is enabled.

Security Rule Types
- ISOLATE_REGISTER
- ISOLATE_SECURITY_STATE
- VERIFY_ACCESS_CONTROL_CONFIG

Detection Example
Assuming the debug interface in the HRoT is disabled by default and is only enabled when sufficiently authorized e.g. through password protection. Unless debug is enabled, no information should flow from assets to the output of the debug module.

Threat Model
A malicious actor can read internal signals in the design through the debug interface if the disabling logic is bypassed.
Security Requirement

Information on the internal signals connected to the \textit{tbus} may flow to the debug interface outputs unless the debug interface is enabled.

Completing the Template Based on Design Signals and Security Requirement

From the requirement, we can fill in the template which gives the rules below.

\begin{verbatim}
assert iflow ( 
tbus.$all_outputs
=/> 
debug.$all_outputs
unless ( debug.enable == ENABLED ) );
\end{verbatim}

**CWE-1192: System-on-Chip (SoC) Using Components without Unique, Immutable Identifiers**

Description

The System-on-Chip (SoC) does not have unique, immutable identifiers for each of its components.

Radix Security Rule Template

Radix currently does not cover this CWE, refer to the MITRE website for suggested mitigations.

**CWE-1193: Power-On of Untrusted Execution Core Before Enabling Fabric Access Control**

Description

The product enables components that contain untrusted firmware before memory and fabric access controls have been enabled.

Radix Security Rule Template

\begin{verbatim}
assert iflow ( 
\{Signals storing untrusted firmware}\)
=/> 
\{Instruction read signals in untrusted components}\)
unless ( \{Access controls are enabled} ));
\end{verbatim}
Rule Template Detail

Information should not flow from *Signals storing untrusted firmware* to *Instruction read signals in untrusted components* unless *Access controls are enabled*.

Security Rule Types

- INTEGRITY_ACCESS_CONTROL_MECHANISM
- INTEGRITY_FSM_STATES
- VERIFY_ACCESS_CONTROL_CONFIG

Detection Example

The un-trusted cores \texttt{core\{0-N\}} fetch instructions from instruction memory in the *peripheral IP & memory* sub-system and master transactions on the interconnect system. These processors should not be enabled until the interconnect access policy is programmed by the \texttt{tmcu} during the secure boot process.

Threat Model

Untrusted software executes before interconnect access policy is configured allowing access to the entire SoC memory space including secure areas.

Security Requirement

Untrusted software must not execute i.e. un-trusted cores may not read instruction memory before secure boot process is complete.

Completing the Template Based on Design Signals and Security Requirement

```
assert iflow(
    imem.idata
  =/>=
  core0.idata
unless ( tmcu.csr.boot_stage >>= FULL_BOOT ) );
```
CWE-1209: Failure to Disable Reserved Bits

Description

The reserved bits in a hardware design are not disabled prior to production. Typically, reserved bits are used for future capabilities and should not support any functional logic in the design. However, designers might covertly use these bits to debug or further develop new capabilities in production hardware. Adversaries with access to these bits will write to them in hopes of compromising hardware state.

Radix Security Rule Template

```
assert iflow ( {{Inputs to memory block with reserved range}} /==> {{Reserved range in memory array}} );
```

Rule Template Detail

Information should not flow from **Inputs to memory block with reserved range** to **Reserved range in memory array**.

Security Rule Types

- CONFIDENTIALITY_ASSET
- INTEGRITY_ASSET
- VERIFY_ACCESS_CONTROL_CONFIG

Detection Example

There are registers in *tmcu* that are reserved for future features. They may enable certain features during the development cycle but will be disabled in the production version of the design.
Threat Model

Features controlled by reserved bits are not properly disabled in the production version of the design allowing an adversary to enable unsupported features which may have negative security consequences.

Security Requirement

All reserved register bits should have no effect on design behavior and reserved register bits should not be writable.

Completing the Template Based on Design Signals and Security Requirement

From the requirement that information in bits in the reserved address range should not have any effect on design behavior, i.e. information should not flow to the output of the memory or register module which holds the information, we can write the rule below. The values for `RESERVED_ADDR_START` and `RESERVED_ADDR_END` are constants provided by the user.

```plaintext
assert iflow ( 
tmcu.csr.data_out 
when ( tmcu.addr >= RESERVED_ADDR_START && tmcu.addr <= RESERVED_ADDR_END )
  =>
  tmcu.$all_outputs );
```

*Note:* Additional Radix security rules may be required to verify additional register or memory module outputs and to ensure reserved register bits are not writable.

**CWE-1220: Insufficient Granularity of Access Control**

Description

The product implements access controls via a policy or other feature with the intention to disable or restrict accesses (reads and/or writes) to assets in a system from untrusted agents. However, implemented access controls lack required granularity, which renders the control policy too broad because it allows accesses from unauthorized agents to the security-sensitive assets.

Radix Security Rule Template

```plaintext
// Confidentiality
assert iflow ( 
  {{Asset}}
when ( {{Access controls enabled}} )
  =>
  {{Untrusted readers}} );
```
// Integrity
assert iflow (
{{untrusted writers}}
/=>
{{Asset}}
unless ( ! {{Access controls enabled}} ));

Rule Template Detail
Information should not flow from/to Asset to/from Untrusted readers/writers when access control is enabled.

Security Rule Types
• CONFIDENTIALITY_ASSET
• INTEGRITY_ACCESS_CONTROL_MECHANISM
• VERIFY_ACCESS_CONTROL_CONFIG

Detection Example

The sram in HRoT has an address range that is readable and writable by un-privileged software and it has an area that is only readable by un-privileged software. The tbus interconnect enforces access control for slaves on the bus but uses only one bit to control both read and write access. Address 0xA0000000 - 0xA000FFFF is readable and writable by the un-trusted cores core{0-N} and address 0xA0010000 - 0xA001FFFF is only readable by the un-trusted cores core{0-N}

Threat Model
The security policy access control is not granular enough as it uses one bit to enable both read and write access. This gives write access to an area that should only be readable by un-privileged agents.
Security Requirement

Access control logic should differentiate between read and write access and to have sufficient address granularity.

Completing the Template Based on Design Signals and Security Requirement

From the requirement that address 0xA0010000 - 0xA001FFFF should not be writable follows the rule below when filling in the template:

```c
assert iflow ( hrot_iface.data
when (mem.addr >= 0xA0010000 && mem.addr <= 0xA001FFFF)
  =/> mem.data );
```

CWE-1221: Incorrect Register Defaults or Module Parameters

Description

Hardware description language code incorrectly defines register defaults or hardware IP parameters to insecure values.

Radix Security Rule Template

```c
// Confidentiality
assert iflow ( {{Asset}}
  =/> {{User-visible signals}} );

// Integrity
assert iflow ( {{User-visible signals}}
  =/> {{Asset}} );
```

Rule Template Detail

Information should not flow from/to Asset to/from User-observable signals.

Note: these templates are generic because the consequences of insecure defaults in hardware description language source code are unpredictable in general. A thorough set of security rules is required to catch all cases and these rules must use values specified in the design specification, not what is coded in the RTL. Verifying correct
register default values is part of functional verification and can also be checked using SVA assertions.

Security Rule Types
- INTEGRITY_CONFIG
- VERIFY_ACCESS_CONTROL_CONFIG

Detection Example

The otp fuses in the Hardware Root of Trust (HRoT) are readable by untrusted software running on core\{0-N\} in debug mode only. In non-debug mode they are not accessible to untrusted software.

Threat Model

The default value of the register bit enabling debug mode is incorrectly set to 1 in the RTL. Hence, allowing untrusted software to read security sensitive data in normal operating mode.

Security Requirement

Default register values and instantiation parameters which has a security impact needs to be verified against the values specified in the design specification.

Completing the Template Based on Design Signals and Security Requirement

From the requirement that the otp fuses should not be readable follows the rule below based on the template. The rule doesn't include the value of the debug mode bit which means the rule will fail if debug mode is incorrectly set to"1".

```verilog
assert iflow ( otp.data /=> hrot_iface.$all_outputs );
```
CWE-1223: Race Condition for Write-Once Attributes

Description

A write-once register in hardware design is programmable by an untrusted software component earlier than the trusted software component, resulting in a race condition issue.

Radix Security Rule Template

```
assert iflow (
    {{Store signals in the LSUs of untrusted CPUs}}
    =>
    {{Write-once registers}}
    unless ( {{Write-once values are set}} ));
```

Rule Template Detail

Information should not flow from software through the Store signals in the Load Store Units (LSUs) of untrusted CPUs to Write-once registers unless the Write-once values are set i.e. they are have been previously written.

Security Rule Types

- INTEGRITY_ACCESS_CONTROL_MECHANISM
- ISOLATE_ASSET

Detection Example

The interconnect in the untrusted section of the SoC contains write-once registers that define the security access policy for all masters and slaves connected to the interconnect. The access policy registers are programmed by the tmcu during secure boot.

Note: Multiple implementations of rules for this CWE are possible due to other system requirements. Here, it doesn't say that the write-once register should not be accessible to untrusted components, or that they shouldn't be allowed to make an attempted write.
It also doesn’t say that the trusted and untrusted cores are fixed over the life of the device.

**Threat Model**

If there is a race condition and the access control policy registers are programmed by untrusted software before the trusted tmcu can program them during secure boot, a less restrictive access policy may be implemented giving a malicious actor access to security sensitive data.

**Security Requirement**

The write-once configuration registers should not be writable unless the write-once register is set.

**Completing the Template Based on Design Signals and Security Requirement**

From the security requirement we can fill out the rule template which gives the rule below:

```plaintext
assert iflow (
{core_0.lsu.store, core_1.lsu.store, core_2.lsu.store, core_N.lsu.store}
/==>
interconnect.csr.access
unless (interconnect.csr.write_once_status ));
```

**CWE-1224: Improper Restriction of Write-Once Bit Fields**

**Description**

The hardware design control register "sticky bits" or write-once bit fields are improperly implemented, such that they can be reprogrammed by software.

**Radix Security Rule Template**

```plaintext
assert iflow ( 
{{Store signals in the LSUs of untrusted CPUs}}
when {{write_once_status is set}}
=/=>
{{State-carrying signals of write-once registers}} );
```

**Rule Template Detail**

Information should not flow from software through the *Store signals in the LSUs of untrusted CPUs* to the *State-carrying signals of write-once registers* when the *write_once_status* is set.
Security Rule Types

- INTEGRITY_REGISTER
- VERIFY_ACCESS_CONTROL_CONFIG

Detection Example

The interconnect in the untrusted section of the SoC above contains write-once registers that define the security access policy for all the masters and slaves. The access policy registers are programmed by the tmcu during secure boot. Hence they must not be re-written by any of the untrusted CPUs.

Threat Model

If the write-once registers are implemented incorrectly so that the "written once" state depends on the data written, an attacker running un-privileged code may write one of the write-once registers after secure boot and thus altering the security access policy and potentially elevating its own privilege level.

Security Requirement

Write-once registers should be truly write-once and the status should not depend on the value written only if the status bit is set.

Completing the Template Based on Design Signals and Security Requirement

From the security requirement we can fill out the rule template which gives the rule below:

```
assert iflow ( 
{core_0.lsu.store, core_1.lsu.store, core_2.lsu.store, core_N.lsu.store} 
when (interconnect.csr.write_once_status == 1) 
=/>= 
interconnect.csr.access );
```
CWE-1231: Improper Implementation of Lock Protection Registers

Description

The product incorrectly implements register lock bit protection features such that protected controls can be programmed even after the lock has been set.

Radix Security Rule Template

```c
assert iflow ({{Protected register data signals}})
=/>={{State-carrying signals of protected registers}}
unless ( {{Protected registers are unlocked}} ));
```

Rule Template Detail

Information should not flow from software through the *Protected register data signals* to the *State-carrying signals of protected registers* unless the *Protected registers are unlocked*.

Security Rule Types

- INTEGRITY_ACCESS_CONTROL_MECHANISM
- INTEGRITY_REGISTER

Detection Example

The SoC above contains a thermal sensor with a programmable max temperature. Another register `tmcu.csr.temp_shutdown` determines the action to take when the max temperature is reached. If the register is programmed to "1", the SoC is shut down to avoid malfunction or damage. The max temperature register and the temp_shutdown register are protected by a lock bit. When trusted firmware sets the lock bit in `tmcu.csr.reg_lock` it is not possible to modify the registers. Due to a design bug, the temp_shutdown register is not protected by the lock bit.
Threat Model

Malicious software running on one of the un-trusted cores can potentially do a fault injection attack by disabling the `tmcu.csr.temp_shutdown` register thus allowing the device to overheat to a point where behavior is unpredictable and security features are no longer active.

Security Requirement

Critical registers which should not be modifiable after configuration, should be protected by a lock mechanism.

Completing the Template Based on Design Signals and Security Requirement

From the security requirement we can fill out the rule template which gives the rule below. Here we assume that `reg_lock == 0` indicate that registers are unlocked and hence writable.

```
assert iflow ( 
  tmcu.data_in //=>
  tmcu.csr.temp_shutdown 
  unless (tmcu.csr.reg_lock == 0) );
```

**CWE-1232: Improper Lock Behavior After Power State Transition**

Description

Register lock bit protection disables changes to system configuration once the bit is set. Some of the protected registers or lock bits become programmable after power state transitions (e.g., Entry and wake from low power sleep modes) causing the system configuration to be changeable.

Radix Security Rule Template

```
assert iflow ( 
  {{Protected register data signals}} //=>
  {{State-carrying signals of protected registers}} 
  unless ( {{Protected registers are unlocked}} ));
```

Rule Template Detail

Information should not flow from software through the *Protected register data signals* to the *State-carrying signals of protected registers* unless the *Protected registers are unlocked*. 
Security Rule Types

- INTEGRITY_ACCESS_CONTROL_MECHANISM
- INTEGRITY_REGISTER

Detection Example

When the SoC above enters a hibernate power state, the memory in the peripheral IP & memory sub-system is powered down and loses configuration settings. In normal mode, the configuration registers cannot be modified by the un-trusted cores core{0-N} when the tmcu.csr.reg_lock bit is set. When resuming operations from hibernate mode, the trusted processor tmcu will disable the lock bit and re-configure the memory before leaving the resume state.

Threat Model

Improper clearing of the lock bit after power state transitions enables malicious software to modify configuration registers.

Security Requirement

Security critical device configuration registers protected by a lock bit must remain protected when returning to operation after power state transitions.

Completing the Template Based on Design Signals and Security Requirement

From the requirement, we can fill in the template which gives the rules below for the core0 processor register.

```c
assert iflow ( memory.data_in /= memory.csr.configuration
unless (tmcu.csr.reg_lock == 0 && power_state != HIBERNATE_RESUME));
```
**CWE-1233: Improper Hardware Lock Protection for Security Sensitive Controls**

**Description**

The product implements a register lock bit protection feature that permits security sensitive controls to modify the protected configuration.

**Radix Security Rule Template**

*Note:* Several security rules may be written for this CWE. In addition, the user needs to review which address ranges and memories should be protected by a lock bit mechanism.

```plaintext
assert iflow ( {{User-accessible-signals}}

=/>= {{Security-critical Device Configuration}}

unless (Lock protection disabled) );
```

**Rule Template Detail**

Signals controllable by untrusted software or agents must not flow to *Security-critical Device Configuration* memory ranges or registers unless the lock protection mechanism is disabled i.e. the device configuration space must not be modified by un-trusted agents.

**Security Rule Types**

- **ISOLATE_SECURITY_STATE**
- **VERIFY_ACCESS_CONTROL_CONFIG**

**Detection Example**

The SoC contains a thermal sensor with a programmable max temperature. Another register `tmcu.csr.temp_shutdown` determines the action to take when the max temperature is reached. If the register is programmed to "1", the SoC is shut down to avoid malfunction or damage. The max temperature register and the temp_shutdown...
register are protected by a lock bit. When trusted firmware sets the lock bit in `tmcu.csr.reg_lock` it is not possible to modify the registers. Due to a design bug, the `temp_shutdown` register is not protected by the lock bit. It is also assumed that the lock bit `tmcu.csr.reg_lock` remains set during normal operation.

**Threat Model**

Malicious software running on one of the un-trusted cores can potentially do a fault injection attack by disabling the `tmcu.csr.temp_shutdown` register thus allowing the device to overheat to a point where behavior is unpredictable and security features are no longer active.

**Security Requirement**

Critical registers which should not be modifiable after configuration, should be protected by a lock mechanism. The lock bit should be set during normal operation, i.e. after secure boot is done.

**Completing the Template Based on Design Signals and Security Requirement**

From the security requirement we can fill out the rule template which gives the rules below. Note that the second rule will fail during boot since `boot_done` will be 0 and additional qualifying signals may be required.

```plaintext
assert iflow (tmcu.data_in !=> tmcu.csr.temp_shutdown unless (tmcu.csr.reg_lock == 0));

assert iflow (tmcu.csr.reg_lock == 1 && tmcu.csr.boot_done == 1);
```

**CWE-1234: Hardware Internal or Debug Modes Allow Override of Locks**

**Description**

System configuration protection may be bypassed during debug mode.

**Radix Security Rule Template**

```plaintext
assert iflow ({{user-controllable signals}}
when (lock bit is set)
```
=/=>
{{Security-sensitive configuration locations}}

assert iflow (  
{{User-controllable signals}}  
when (lock bit is set)  
=/=>  
{{Security-sensitive configuration locations}}  
unless (debug mode || scan mode) );

Rule Template Detail

Information should not flow from untrusted software through the User-controllable signals to the Security-sensitive configuration locations when the lock bit is set. This rule applies if untrusted software is never allowed to modify the configuration registers or memory locations when the lock bit is set regardless of other hardware internal or debug modes being set. If some accesses are allowed, add the (debug mode || scan mode) condition to check specific modes.

Note: Depending on design implementation, only one of the template rules applies since the condition when access is allowed is different.

Security Rule Types

• INTEGRITY_ACCESS_CONTROL_MECHANISM

Detection Example

Trusted firmware running on the tmcu configures memory in the peripheral IP & memory subsystem during secure boot. The memory.csr.configuration register is protected by the tmcu.csr.lock_bit which is set after configuration is done by the tmcu. When the SoC is in debug or scan mode, the lock bit for the configuration register is overridden.

Threat Model

If untrusted software is able to control either the scan or debug mode bits it will override the lock bit for the configuration register so that it can modify the memory configuration.
Security Requirement

Depending on design intent, there are two different requirements. If overriding the lock bit for the configuration is a design bug or oversight, then configuration should only be writable when the lock bit is not set. If overriding the lock bit for the configuration is intended, then no other way of writing the register should be possible.

Completing the Template Based on Design Signals and Security Requirement

From the requirements, we can fill in the template which gives the rules below for the two cases:

```
assert iflow (memory.data_in
when (tmcu.csr.lock_bit == 1)
==>
memory.csr.configuration );

assert iflow (memory.data_in
when (tmcu.csr.lock_bit == 1)
==>
memory.csr.configuration
unless (tmcu.csr.scan_mode || tmcu.csr.debug_mode ) );
```

CWE-1240: Use of a Risky Cryptographic Primitive

Description

This device implements a cryptographic algorithm using a non-standard or unproven cryptographic primitive.

Radix Security Rule Template

Radix currently does not cover this CWE, refer to the MITRE website for suggested mitigations.
CWE-1241: Use of Predictable Algorithm in Random Number Generator

Description

The device uses an algorithm that is predictable and generates a pseudo-random number.

Radix Security Rule Template

Radix currently does not cover this CWE, refer to the MITRE website for suggested mitigations.

CWE-1242: Inclusion of Undocumented Features or Chicken Bits

Description

The device includes chicken bits or undocumented features that can create entry points for unauthorized actors.

Radix Security Rule Template

Note: Chicken bits should be permanently disabled in production devices or adequate protection should ensure they are not controllable by users. Designers need to document the address range for accessing chicken bits in internal documentation. Depending on system requirements, the chicken bits may or may not be readable registers. If they readable, additional rules may be required if read access to the bits is restricted.

```plaintext
assert iflow ( {{User-accessible signals}} =/> {{Chicken bits in registers or memory}} );
```

Rule Template Detail

The Radix Security Rules checks that information doesn't flow from User-accessible signals to Chicken bits in registers or memory. That is, un-authorized software or agents can't control the chicken bits.

Security Rule Types

- CONFIDENTIALITY_ASSET
- INTEGRITY_ASSET
Detection Example

The SoC design contains chicken bits in a range of memory in the `tmcu.csr`. These bits must not be controllable in a production device by un-authorized users. Only trusted software running on the `tmcu` may write to chicken bits.

Threat Model

Un-trusted agents being able to control chicken bits may enable features that violate security access policy thus giving the un-trusted agent access to privileged data.

Security Requirement

Chicken bits should not be controllable in a production device unless done by a trusted agent.

Completing the Template Based on Design Signals and Security Requirement

From the requirement, we can fill in the template which gives the rule below. The values for `CHICKEN_END` and `CHICKEN_START` are constants provided by the user.

```plaintext
assert iflow (
    tmcu.$all_inputs
    =/=>
    tmcu.csr.mem[CHICKEN_END:CHICKEN_START]
unless (tbus.security_ID == TMCU);
```

Note: Chicken bits must be documented in internal specifications to enable verification and hidden in external documentation.
CWE-1243: Sensitive Non-Volatile Information Not Protected During Debug

Description
Access to security-sensitive information stored in fuses is not limited during debug.

Radix Security Rule Template

```plaintext
assert iflow ( 
{{Security-sensitive Fuse Values}}
when (debug mode enabled) 
=/=>
{{User-accessible signals}});
```

Rule Template Detail

Information in blown fuses or ROM, Security-sensitive Fuse Values, should not flow to User-accessible signals such as an untrusted debugger when the device is in debug mode.

Security Rule Types

• ISOLATE_REGISTER

Detection Example

Security sensitive information is stored in blown fuses in the otp block and in a section of the rom.mem. During normal operation mode, access control methods prevent untrusted system components from reading this data.

Threat Model

Data in rom.mem and otp fuses may be visible through the debug interface when the device is in debug mode and normal access control may not be set up. This would give access to sensitive data through an untrusted debugger.
Security Requirement

Data in *rom* and *otp* must not be readable through the debug interface

Completing the Template Based on Design Signals and Security Requirement

From the requirement, we can fill in the template which gives the rule below. The values for *FUSED_END* and *FUSED_START* are constants provided by the user.

```plaintext
assert iflow ( {rom.mem[FUSED_END:FUSED_START], otp.$all_outputs} when (tmcu.csr.debug_mode) /==> debug.$all_outputs );
```

**CWE-1244: Improper Access to Sensitive Information Using Debug and Test Interfaces**

Description

The product's physical debug and test interface protection does not block untrusted agents, resulting in unauthorized access to and potentially control of sensitive assets.

Radix Security Rule Template

```plaintext
// Confidentiality
assert iflow ( {{Security-critical signals}} /==> {{User-accessible debug interface}} unless (debug authentication == TRUE) );

// Integrity
assert iflow ( {{User-accessible debug interface}} when (debug authentication == FALSE) /==> {{Security-critical signals}} );
```

Rule Template Detail

Information should not flow from/to the *Security-critical signals* to/from the *User-accessible debug interface* unless the debug interface access has been properly authenticated.
Security Rule Types

- INTEGRITY_ACCESS_CONTROL_MECHANISM
- VERIFY_ACCESS_CONTROL_CONFIG

Detection Example

A debugger connected to the debug interface needs to provide a correct response to a challenge in order to access internal registers of the Hypothetical SoC, for example registers in the $tmcu$. If the authorization is successful, the $debug.authentication$ register is set to "1".

Threat Model

The challenge response authorization is not applying to all debug accesses thus giving an attacker access to some security sensitive registers.

Security Requirement

There should be no read or write access to security sensitive registers unless the debug agent has been properly authenticated.

Completing the Template Based on Design Signals and Security Requirement

From the requirement, we can fill in the template which gives the rules below:

```plaintext
assert iflow (
  $tmcu.csr.rdata
  /==>
  $debug.$all_outputs
) unless ($debug.authentication == 1);

assert iflow (
  $debug.$all_outputs
  when ($debug.authentication == 0)
  /==>
  $tmcu.csr.wdata
);
```
CWE-1245: Improper Finite State Machines (FSMs) in Hardware Logic

Description

Faulty finite state machines (FSMs) in the hardware logic allow an attacker to put the system in an undefined state, to cause a denial of service (DoS) or gain privileges on the victim's system.

Radix Security Rule Template

```c
assert iflow ( 
{{FSM next state}}
=>
{{FSM current state}}
unless (next state is valid) );
```

Rule Template Detail

Critical Finite State Machines in the design should not be able to enter undefined states where the behavior is undefined. The Radix Security Rule ensures the FSM next state variable does not flow to the FSM current state variable unless the Next State is a valid state.

Security Rule Types

- INTEGRITY_FSM_STATES

Detection Example

The aes.csr FSM determines read, write or read/write access permissions for registers in the aes module based on source security ID. The FSM have 4 valid states: IDLE, RD, WR and RDWR. The state is one-hot encoded with 4 state bits for performance reasons which means there are many possible undefined states.
Threat Model

An attacker may cause the FSM into an undefined state where access permissions are not enforced allowing access to security sensitive registers

Security Requirement

Security sensitive FSMs should not be able to enter undefined states

Completing the Template Based on Design Signals and Security Requirement

From the requirement, we can fill in the template which gives the rule below:

```csharp
assert iflow ( 
aes.csreg.next_state 
)=>
aes.csreg.current_state
unless (aes.csreg.next_state == IDLE || aes.csreg.next_state == RD ||
aes.csreg.next_state == WR || aes.csreg.next_state == RDWR) );
```

CWE-1246: Improper Write Handling in Limited-write Non-Volatile Memories

Description

The product does not implement or incorrectly implements wear leveling operations in limited-write non-volatile memories.

Radix Security Rule Template

Note: Functional verification should ensure that write leveling is implemented correctly as this is part of the device's defined functionality. However, Radix Security Rules can also be used since violations have security implications.

```csharp
assert iflow ( 
{{Store data}}
when (write limit for location reached)
=>
{{Limited-write Non-volatile memory location}}
unless (write done by trusted processor) );
```
Rule Template Detail

*Store data* from an un-secure processor must not flow to a *limited-write Non-volatile memory location* if the maximum number of write for that location has been reached. A trusted processor may be allowed to write anyway depending on system design.

**Security Rule Types**

- INTEGRITY_FSM_STATES

**Detection Example**

![Diagram](image)

The *nvm* flash memory implements write leveling to prevent premature failures of the memory. The *nvm* has a register that defines the maximum number of writes per location. This register is only readable and writable by the secure *tmcu* processor.

**Threat Model**

A malicious actor may be able to bypass the write leveling logic and perform a large number of writes to the same location thus making part of the flash unreliable. This may lead to undefined states in the system where security policies are not enforced or may enable a denial of service attack.

**Security Requirement**

Non-volatile memory locations should not be writable by un-trusted agents when the maximum number of writes for the location has been reached.

**Completing the Template Based on Design Signals and Security Requirement**

From the requirement, we can fill in the template which gives the rule below:

```plaintext
assert iflow (nvm.wdata
when ((nvm.num_writes[nvm.addr] >= nvm.csr.max_num_writes) && nvm.we) =>
  nvm.mem
unless (nvm.security_source_id == TMCU)
```
CWE-1247: Missing or Improperly Implemented Protection Against Voltage and Clock Glitches

Description

The device does not contain or contains improperly implemented circuitry or sensors to detect and mitigate voltage and clock glitches and protect sensitive information or software contained on the device.

Radix Security Rule Template

Radix currently does not cover this CWE, refer to the MITRE website for suggested mitigations.

CWE-1248: Semiconductor Defects in Hardware Logic with Security-Sensitive Implications

Description

The security-sensitive hardware module contains semiconductor defects.

Radix Security Rule Template

Radix currently does not cover this CWE, refer to the MITRE website for suggested mitigations.

CWE-1251: Mirrored Regions with Different Values

Description

The product's architecture mirrors regions without ensuring that their contents always stay in sync.

Radix Security Rule Template

*Note:* This CWE overlaps with the functional requirement to keep two resources in sync and this should be addressed in the functional verification plan. Radix Security Rules can be used to ensure no illegal data access when resources are not in sync as this has security implications.

```assert iflow ({{data controlled by duplicated resource}} when (update in progress)```
Assuring that, for example, data in a cache location is equal to data in memory can be done using functional assertions such as:

```vhdl
assert (@ posedge clk) (update not in progress) => (core0.dcache[flush_dst] == nvm.mem[flush_dst]);
```

### Rule Template Detail

Data in a duplicated resource must not flow to a location that is user accessible while an update of the resource controlling the data is in progress.

#### Security Rule Types

- INTEGRITY_FSM_STATES
- INTEGRITY_MEMORY_REGION
- VERIFY_MEMORY_REGION

#### Detection Example

![Diagram showing SoC design example](image)

There are 3 processors in the SoC design example, core0, core1 & core2, running user code. For performance reasons, there is one main Memory Management Unit (MMU) and one shadow MMUs. The main MMU handles memory accesses by core0 and the shadow MMU handles memory accesses from core1 & core2. Updates to the main MMU is done by the tmcu and then the main MMU updates the shadow MMU through messages on the interconnect. If the accessible address range for the untrusted cores is updated in the main MMU, there is a time when the three processors may have access to different memory ranges.

#### Threat Model

A malicious agent running on core2 may be able to access memory locations outside its allowed range because the shadow MMU does not yet have the same configuration as the main MMU. If core2 can flood the interconnect with traffic and delay the update request, the update to the shadow MMU may be delayed, extending the time available for access.
Security Requirement

No access to memory through the shadow MMU should be allowed while an update is in progress.

Completing the Template Based on Design Signals and Security Requirement

From the requirement, we can fill in the template which gives the rules below:

```plaintext
assert iflow (memory.rdata
  when (core0.mmu.updating)
  =/=>
  {core1.data_in, core2.data_in} );

assert iflow ( {core1.data_out, core2.data_out}
  when (core0.mmu.updating)
  =/=>
  memory.wdata );
```

**CWE-1252: CPU Hardware Not Configured to Support Exclusivity of Write and Execute Operations**

Description

The CPU is not configured to provide hardware support for exclusivity of write and execute operations on memory. This allows an attacker to execute data from all of memory.

Radix Security Rule Template

```plaintext
assert iflow ( {{Store data}}
  when (address is inside instruction memory)
  =/=>
  {{Memory}} );

assert iflow ( {{Memory}}
  when (address is outside instruction memory)
  =/=>
  {{instruction fetch unit}} );
```
Rule Template Detail

A processor must not be able to write data in instruction memory and it should only be able to read instructions from instruction memory. *Store data* in the Load Store Unit (LSU) should not flow to memory location in the instruction address range and data in Memory should not flow to *instruction fetch unit* in the CPU if the address is outside the address range for instruction memory.

Security Rule Types

- INTEGRITY_FSM_STATES
- ISOLATE_MEMORY_REGION

Detection Example

In this example, the *tmcu* doesn't have support for write exclusivity and the SoC doesn't have an Memory Protection Unit (MPU) or Memory Management Unit (MMU) to isolate memory regions as execute only. Hence, the *tmcu* load store unit can write to the entire *sram.mem* memory and the instruction fetch unit can execute code in the entire *sram.mem* as well.

Threat Model

An attacker can write malicious code to memory and later execute the code.

Security Requirement

If the processor lacks support for write exclusivity, other logic must implement this functionality to prevent writes to instruction memory and instruction fetch from outside instruction memory.

Completing the Template Based on Design Signals and Security Requirement

From the requirement, we can fill in the template which gives the rules below. The address range in *sram* used for instruction memory is defined in the *tmcu.mpu.imem_start_addr* and *tmcu.mpu.imem_end_addr*.

```c
/* no data write to inst mem region */
assert iflow (%
```
TORTUGA LOGIC | RADIX COVERAGE FOR HARDWARE COMMON WEAKNESS ENUMERATION (CWE) GUIDE

tmcu.lsu.store_data
when ( tmcu.lsu.store_addr >= tmcu.mpu.imem_start_addr
&& tmcu.lsu.store_addr <= tmcu.mpu.imem_end_addr )
=>
sram.mem);

/* no inst read from outside inst mem region */
assert iflow (
  sram.mem
when ( tmcu.ifetch.load_addr <= tmcu.mpu.imem_start_addr
|| tmcu.ifetch.load_addr >= tmcu.mpu.imem_end_addr )
=>
tmcu.ifetch.load_inst );

CWE-1253: Incorrect Selection of Fuse Values

Description
The logic level used to set a system to a secure state relies on a fuse being unblown. An attacker can set the system to an insecure state merely by blowing the fuse.

Radix Security Rule Template

```systemverilog
assert iflow ( {{fuse_values == blown_value && secure_feature_enabled}} );
```

Rule Template Detail
A fuse having its blown value and the corresponding security feature being enabled should always hold true. The requirement in this CWE should also be verified during functional verification. The Radix security rule could also be written as a System Verilog Assertion since the condition should always hold true.

Security Rule Types
- VERIFY_ACCESS_CONTROL_CONFIG
- VERIFY_ASSET
Detection Example

The control word defined by the fuses in \texttt{otp} determine what security related features are enabled in the chip following manufacturing. For example, scan mode is permanently disabled after manufacturing test by blowing the corresponding fuse (the value 1 represents a blown fuse) e.g. bit [0] in \texttt{otp.fuses}.

Threat Model

If disabling scan mode incorrectly corresponded to fuse value \(== 0\), an attacker could blow the fuse and thus enable scan mode and get access to every register in the design through the JTAG port.

Security Requirement

A blown one time programmable fuse should always correspond to the most secure, restrictive operating mode of the device.

Completing the Template Based on Design Signals and Security Requirement

From the requirement, we can fill in the template which gives the rule below.

\begin{verbatim}
assert iflow (
  otp.fuses[0] == 1 && jtag.scan_enable == 0);
\end{verbatim}

\textbf{CWE-1254: Incorrect Comparison Logic Granularity}

Description

The product's comparison logic is performed over a series of steps rather than across the entire string in one operation. If there is a comparison logic failure on one of these steps, the operation may be vulnerable to a timing attack that can result in the interception of the process for nefarious purposes.
Radix Security Rule Template

```plaintext
assert iflow (
    {{pass or fail status}}
when (time for longest operation)
  /=>
    {{Signals readable by untrusted agents}} );
```

Rule Template Detail

The result of a comparison operation must not be visible to an untrusted agent at different latency if the comparison may take a different amount of time to complete depending on the result of the compare.

*Note:* Radix security rules don’t support timing in the rule itself, however, a signal can be created in the testbench and bound to the design and then used in the security rule.

Security Rule Types

- INTEGRITY_ASSET
- INTEGRITY_FSM_STATES
- ISOLATE_MEMORY_REGION

Detection Example

The debug unit checks a user-provided password to grant access to a user. The password is 64 bits but the comparison logic is implemented using an 8 bit comparator, checking each byte of the password on consecutive clock cycles. If the password compare fails in the first byte, the fail status signal is asserted and access is denied. If all 8 compares pass, access is granted.

Threat Model

By measuring the time between request and the fail indication, the timing side channel leaks information about which byte of the password does not match. If you know which byte fails, it is easy to guess the correct password.
Security Requirement

The compare function should indicate pass/fail after the same amount of time regardless if the fail happened before the last byte.

Completing the Template Based on Design Signals and Security Requirement

From the requirement, we can fill in the template which gives the rule below. The signal `debug.check_wait` is created in the testbench and is asserted for 8 cycles while the compare is in progress. It is bound to the debug instance. The amount of time required to complete the comparison is design dependent and 8 cycles is just used as an example.

```plaintext
assert iflow (
  {debug.passwd_comp.pass, debug.passwd_comp.fail}
when (debug.check_wait)
  !==>
  debug.$all_outputs);
```

CWE-1256: Hardware Features Enable Physical Attacks from Software

Description

Software-controllable device functionality such as power and clock management permits unauthorized modification of memory or register bits.

Radix Security Rule Template

```plaintext
assert iflow (  
  {{User-accessible signals}}
  !==>
  {{Asset}}
  unless ( {{Permission bits}} == {{Value indicating asset should be writable}} ));
```

Rule Template Detail

Information in the Asset which control physical parameters on chip must be prevented from being written with data from user-accessible signals unless the permission bits are correctly set to the value indicating asset should be writable.

Security Rule Types

- INTEGRITY_ACCESS_CONTROL_CONFIG
• INTEGRITY_ASSET
• VERIFY_ACCESS_CONTROL_CONFIG

Detection Example

Assume that security-critical settings for scaling clock frequency and voltage are available in a range of registers bounded by PRIV_END_ADDR:PRIV_START_ADDR in the tmcu.csr module of the tmcu. These values are writable based on the lock_bit register in the same module. The lock_bit is only writable by the tmcu.

Threat Model

We assume that untrusted software (the threat) has access to the input and output ports of the hrot_iface. If untrusted software can clear the lock_bit or write the clock frequency and voltage registers due to inadequate protection, a fault injection attack could be run.

Security Requirement

Information in the address range [PRIV_END_ADDR:PRIV_START_ADDR] must not be writable via hrot_iface unless tmcu.csr.lock_bit is zero. The tmcu.csr.lock bit must never be writable from the hrot_iface interface.

Completing the Template Based on Design Signals and Security Requirement

Based on the security goal we fill in the template which gives the two rules below. The values for PRIV_END_ADDR and PRIV_START_ADDR are constants provided by the user.

```c
assert iflow (hrot_iface.data =/=>
              tmcu.csr.mem[PRIV_END_ADDR:PRIV_START_ADDR]
            unless (tmcu.csr.lock_bit == 0));

assert iflow (hrot_iface.data
```
CWE-1257: Improper Access Control Applied to Mirrored or Aliased Memory Regions

Description
Aliased or mirrored memory regions in hardware designs may have inconsistent read/write permissions enforced by the hardware. A possible result is that an untrusted agent is blocked from accessing a memory region but is not blocked from accessing the corresponding aliased memory region.

Radix Security Rule Template

```c
assert iflow ( {{Protected memory}}
when (local memory address) 
/=/> {{Signals readable by untrusted agents}} );
```

Rule Template Detail
Security sensitive memory mapped registers or memory areas must not flow to signals readable by untrusted agents even if an aliased address is used.

Security Rule Types
- INTEGRITY_ACCESSCONTROL_MECHANISM
- VERIFY_ACCESSCONTROL_CONFIG

Detection Example
ROM in the SoC is 64k and it is mapped to address 0x08000000 - 0x0800FFFF. The ROM is security sensitive and is only readable by the tmcu. In order to simplify the address decoding logic, the ROM only decodes the lower 16 address bits and relies on the Memory Protection Unit (MPU) to enforce access control.

Threat Model

One of the untrusted cores, core{0-N} has read permission from address 0x0400beef. Due to improper access control, the ROM decodes the address as 0xbeef and responds to the read request even though the un-trusted core should not have access to address 0x0800_0000 and above.

Security Requirement

No location in ROM should flow to untrusted agents in the system even if the address at the ROM is in the correct range due to memory aliasing.

Completing the Template Based on Design Signals and Security Requirement

From the requirement, we can fill in the template which gives the rule:

```plaintext
assert iflow (rom.data
when (rom.address >= 0x0000 && rom.address <= 0xFFFF)
=/=>
hrot_iface.$all_outputs );
```

Note: The when condition in the rule is not needed if the untrusted core is not allowed to read any data in rom.

CWE-1258: Exposure of Sensitive System Information Due to Uncleared Debug Information

Description

The hardware does not fully clear security-sensitive values, such as keys and intermediate values in cryptographic operations, when debug mode is entered.

Radix Security Rule Template

```plaintext
assert iflow ({{Sensitive Register}}
when ! (register clear request)
=/=>
```

© 2020 Tortuga Logic
Rule Template Detail

Security sensitive registers must be cleared when a clear request signal is asserted and their contents must not flow to other locations until the contents has been cleared.

Security Rule Types

- INTEGRITY_FSM_STATES
- VERIFY_SECURITY_STATE

Detection Example

Keys for AES are stored in internal registers `aes.csr.key`. These registers are blocked for access by software and other untrusted agents of the SoC. When the design is in debug mode, all registers are accessible through the debug interface. To avoid keys being accessible to unauthorized users, they will be cleared when entering debug mode. The register clear request signal in the template is generated from the debug mode signal in the tmcu. A register in aes indicate that the value is zero.

Threat Model

If the aes key register is not cleared when entering debug mode, an untrusted debugger can gain access to the keys.

Security Requirement

The aes key register and other sensitive registers should be cleared when entering debug mode.

Completing the Template Based on Design Signals and Security Requirement

From the requirement, we can fill in the template which gives the rule for the aes key register. The `tmcu.clear_request` signal is asserted when the `tmcu` enables debug mode and de-asserted on completion of clearing the register.
CWE-1259: Improper Restriction of Security Token Assignment

Description

The System-On-A-Chip (SoC) implements a Security Token mechanism to differentiate what actions are allowed or disallowed when a transaction originates from an entity. However, the Security Tokens are improperly protected.

Radix Security Rule Template

```plaintext
assert iflow ( 
{{Sensitive Register}}
when ( not access by secure processor )
=/> 
{{Sensitive Register}}
unless Reset );
```

Rule Template Detail

Security sensitive registers must not be changed unless a write is done by a secure processor or they are set to reset values during chip reset.

Security Rule Types

- CONFIDENTIALITY_SECURITY_STATE
- INTEGRITY_SECURITY_STATE
Detection Example

Access to secret registers such as AES key registers is configured in access-policy registers and is determined based on security identifiers. Each agent in the SoC have a Security identifier register which is programmed to a unique value by the tmcu during secure boot.

Threat Model

If one of the untrusted cores, core{0-N} can change its own or another agent's security identifier, the access policy will allow access to registers that would otherwise be protected from this agent.

Security Requirement

The Security Identifier for an agent should not change unless it is programmed by the tmcu or it is cleared during reset.

Completing the Template Based on Design Signals and Security Requirement

From the requirement, we can fill in the template which gives the rules below for the core0 processor register.

```c
assert iflow ( 
  core0.csr.SECURITY_ID 
  when (tbus.master != TMCU_ID) 
  =>/=>
  core0.csr.SECURITY_ID 
  unless (reset) );
```

Note: Additional security rules for each unsecure master is required.
CWE-1260: Improper Handling of Overlap Between Protected Memory Ranges

Description
The product allows address regions to overlap, which can result in the bypassing of intended memory protection.

Radix Security Rule Template

```plaintext
assert iflow ( {{Privileged data signals}}
when ( {{Privileged address condition}} )
=/>=>
{{Signals readable by untrusted agents}} );

assert iflow ( {{Signals writable by untrusted agents}}
=/>=>
{{Privileged data signals}}
unless ( ! {{Privileged mode condition}} ) );
```

Rule Template Detail

Data in a privileged address range in a shared memory (Privileged data signals) must not influence signals readable by untrusted agents. This should always hold true even if the non-privileged address range overlap the privileged address range. Additionally, the privileged address range in shared memory must not be able to be influenced by signals writable by untrusted agents.

Security Rule Types
- ISOLATE_DATA
- ISOLATE_MEMORY_REGION

Detection Example
The Hardware Root of Trust (HRoT) local \textit{sram} is memory mapped in the \textit{core[0-N]} address space and is also accessible to the \textit{tmcu}. The address range for privileged memory space is defined in \textit{tmcu.csr} registers. It is only readable and writable by privileged software. The un-trusted cores, \textit{core[0-N]} can define an unprivileged area in memory where they have read and write access.

\textbf{Threat Model}

In this example, we assume that the threat is from malicious software in the untrusted domain. We assume this software has access to the \textit{core[0-N]} un-privileged memory map and can also change the location of its un-privileged area. If the software running in the untrusted domain can program the un-privileged memory area to overlap with the privileged memory area, this would allow the malicious software to read and write privileged memory.

\textbf{Security Requirement}

Data in the privileged area of memory must not flow to the \textit{hrot_iface} and data on the \textit{hrot_iface} must not flow to the privileged area of memory regardless of how the non-privileged address space is programmed

\textbf{Completing the Template Based on Design Signals and Security Requirement}

From the requirement, the privileged memory should not be read or write accessible. The privileged range in the SRAM model is bounded by \textit{PRIV\_START\_ADDR} and \textit{PRIV\_END\_ADDR}. The values for \textit{PRIV\_END\_ADDR} and \textit{PRIV\_START\_ADDR} are constants provided by the user. Filling in the template gives the rule below.

\begin{verbatim}
assert iflow ( 
    sram.mem[PRIV\_END\_ADDR:PRIV\_START\_ADDR] 
  >>=
    hrot_iface.$all\_outputs );

assert iflow ( 
    hrot_iface.$all\_inputs 
  >>=
    sram.mem[PRIV\_END\_ADDR:PRIV\_START\_ADDR] );
\end{verbatim}

\textbf{CWE-1261: Improper Handling of Single Event Upsets}

\textbf{Description}

The hardware logic does not effectively handle when single-event upsets (SEUs) occur.
Radix Security Rule Template

```
// Confidentiality
assert iflow (  
{{Security sensitive signals}}
when ( {{SEU detected}} )
/=>
{{Signals readable by untrusted agents}} );

// Integrity
assert iflow (  
{{Signals writable by untrusted agents}}
when ( {{SEU detected}} )
/=>
{{Security sensitive signals}} );
```

Rule Template Detail

Security sensitive information should not flow to *signals readable by untrusted agents* when the circuit is in an error state caused by a Single Event Upset. Additionally, security sensitive data must not be able to be influenced by *signals writable by untrusted agents* when the circuit is in an error state caused by a Single Event Upset.

Security Rule Types

- CONFIDENTIALITY_ASSET
- INTEGRITY_ASSET
- ISOLATE_ASSET

Detection Example

The SoC above has logic to detect errors, e.g. parity on memory data or duplicated logic running in lock-step, to detect Single Event Upset (SEU) faults. When an SEU fault is detected, an error bit is set in CSR and software on *tmcu* tries to recover back to normal.
operation. During this error state, secure data for example in the SRAM must not be overwritten or be visible to un-trusted cores or on the debug interface.

Threat Model

In this example, we assume that the threat is from a malicious user either relying on a random event or intentional fault insertion to bring the design into an error state where normal security policies may no longer apply. If malicious software running in the untrusted domain is able to bypass the security policy, this would allow the malicious user to read and write privileged memory. If the malicious user has physical access to the chip, he can try to access secure data through the debug interface which may no longer be protected in an error scenario.

Security Requirement

Data in the sram memory must not flow to the debug module and data from the debug module must not flow to the memory when an SEU is detected

Completing the Template Based on Design Signals and Security Requirement

From the requirement, the sram memory should not be read or write accessible from the debug module when the SEU error bit is set. The SEU may affect the SRAM address so it is safer to check the entire memory. Filling in the template gives the rule below.

```plaintext
assert iflow ( sram.mem when (tmcu.csr.SEU_err) /==> debug.$all_outputs );

assert iflow ( debug.data when (tmcu.csr.SEU_err) /==> sram.mem );
```

CWE-1262: Register Interface Allows Software Access to Sensitive Data or Security Settings

Description

Memory-mapped registers provide access to hardware functionality from software and if not properly secured can result in loss of confidentiality and integrity.
Radix Security Rule Template

*Note:* Due to breadth of access control policies for different memory mapped registers, there are many rule variations to detect disallowed accesses by software.

```plaintext
assert iflow (  
  {{Sensitive Data}}
  =/=>
  {{Signals readable by untrusted agents}}
  unless permitted condition );

assert iflow (  
  {{Signals writable by untrusted agents}}
  =/=>
  {{Sensitive Data}} );
```

**Rule Template Detail**

Sensitive data should not flow to *signals readable by untrusted agents* unless a "permitted condition" is true. Additionally, sensitive data must not be able to be influenced by *signals writable by untrusted agents*.

**Security Rule Types**

- ISOLATE_DATA
- ISOLATE_REGISTER

**Detection Example**

Assume that the registers in the HRoT aes core are memory mapped in the *core{0-N}* address space. All the aes core memory mapped registers have an access control policy specifying read or write access and access by *tmcu* or *core{0-N}*. For example, the aes.reg.key register is only readable and writable by *tmcu*, the *aes.reg.data_in* register is writable but not readable by *core{0-N}* and the *aes.reg.data_out* register is only readable when the *aes.done* bit is set.
Threat Model

In this example, we assume that the threat is from malicious software running on one of the untrusted processors. It will attempt to read and write all memory mapped registers in the aes address space hoping the access control policy is insufficient. If any access succeeds, information about the key may be obtained.

Security Requirement

Only the \texttt{aes.reg.data\_in} register is writable by \texttt{core[0-N]} and only the \texttt{aes.reg.data\_out} register is readable by \texttt{core[0-N]} when encryption is done. The \texttt{aes.data\_out} register is never writable and the \texttt{aes.data\_in} register is never readable by \texttt{core[0-N]}.

Completing the Template Based on Design Signals and Security Requirement

From the requirement, we can fill in the template which gives the rules below for the \texttt{data\_out} register. Similarly, rules are required for the \texttt{data\_in} register.

```plaintext
assert iflow (
  aes.reg.data\_out
  =>
  hrot_iface.all\_outputs
  unless (aes.done));

assert iflow (
  hrot_iface.data
  =>
  aes.reg.data\_out);
```

CWE-1264: Hardware Logic with Insecure De-Synchronization between Control and Data Channels

Description

The hardware logic for error handling and security checks can incorrectly forward data before the security check is complete.

Radix Security Rule Template

```plaintext
assert iflow (
  {{privileged data}}
  when (permission to access data has not yet been granted)
  =>
  cache);
```
Rule Template Detail

Privileged data shall not flow to the cache if the requestor doesn’t have permission to access data or permission check is not completed.

Security Rule Types

- CONFIDENTIALITY_ASSET
- INTEGRITY_ASSET
- ISOLATE_ASSET

Detection Example

The tmcu and core{0-N} processors are interconnected through AXI. The security policy (bus firewall) is implemented in an IP block separate from the data routing interconnect. The Hardware Root of Trust (HRoT) processor, tmcu should not be able to share data with the untrusted core{0-N} processors and the bus firewall prevents the untrusted processors from accessing for example rom data in the HRoT.

Threat Model

If the firewall logic becomes de-synchronized with the data routing an untrusted processor may be able to read privileged data before the security policy is ready and enforced

Security Requirement

All privileged data is buffered or blocked by the interconnect until it has determined that the requestor has permission to access the data

Completing the Template Based on Design Signals and Security Requirement

Based on the security goal we fill in the template which gives the rule below.

```
assert iflow (
  axi_bus_requester_1.wdata
  when ( !axi_firewall.access_table[1][2] )
```
CWE-1266: Improper Scrubbing of Sensitive Data from Decommissioned Device

Description

The product does not properly provide a capability for the product administrator to remove sensitive data at the time the product is decommissioned. A scrubbing capability could be missing, insufficient, or incorrect.

Radix Security Rule Template

Note: Radix Security Rules do not apply to post-RTL verification. Please implement policies to scrub sensitive data when product in use. Security rules may be written to verify scrubbing implementations in RTL or software during the design phase.

```plaintext
assert iflow (
    {{Sensitive data location}} /=>
    {{Sensitive data location}}
    unless (! scrubbing started and done ) );
```

Rule Template Detail

Sensitive data to be scrubbed should not flow to the same location when scrubbing has started and is done i.e. sensitive data values are cleared.

Security Rule Types

- CONFIDENTIALITY_ASSET
- VERIFY_SECURITY_STATE
Detection Example

The `aes.csr.key` register contains sensitive information that needs to be removed after use. This is done by HW setting the key to all zero. The operation is started by the tmcu writing "1" to the `aes.csr.data_clear` register. The `aes.csr.clear_done` bit is set to "1" when the clear operation is done. The tmcu will then clear the `aes.csr.data_clear` bit to complete the operation.

Threat Model

If the clear is not successful, bits of the key may remain that could leak to unauthorized locations.

Security Requirement

No information should flow from the key register prior to clearing back to the key register when clearing is done.

Completing the Template Based on Design Signals and Security Requirement

Based on the security goal we fill in the template which gives the rule below. This rule only checks for leakage of the key after the clearing is completed so it will fail if the clearing of the key was not successful.

```verilog
assert iflow (
    aes.csr.key
    when (aes.csr.data_clear == 0)
    =>
    aes.csr.key
    unless !(aes.csr.data_clear & aes.csr.clear_done));
```
CWE-1268: Policy Privileges are not Assigned Consistently Between Control and Data Agents

Description
The product's hardware-enforced access control for a particular resource improperly accounts for privilege discrepancies between control and write policies.

Radix Security Rule Template

```c
assert iflow ( {{Sensitive signals}} => {{User-accessible signals}} );

assert iflow ( {{User-accessible signals}} => {{Critical configuration registers}} );
```

Rule Template Detail

*Note:* Several Security Rules may be required to check for this CWE since it may be caused by incorrect software or incorrect hardware. Untrusted software through *User-accessible signals* should not flow to any of the critical configuration registers. Sensitive signals, e.g. register values should not flow to *User-accessible signals*. These rules will flag any confidentiality and integrity violation on the specified signals which may be stricter than the intended access policy. Please review the system design documentation and add exceptions to rules as required.

Security Rule Types
- CONFIDENTIALITY_ASSET
- INTEGRITY_ASSET
- ISOLATE_ASSET
Detection Example

All the registers in the aes block are protected by different access policies. For example, the `aes.csr.key` register is readable and writable by the `tmcu` only. The `aes.csr.status` register is readable by a specific master when the `aes.csr.control` registers is enabled for that specific master.

Threat Model

A less privileged access policy may override a higher privileged access policy due to incorrect implementation thus allowing access for an untrusted agent to privileged data.

Security Requirement

Sensitive access controlled signals should adhere to intended access control policy as defined in design specification.

Completing the Template Based on Design Signals and Security Requirement

From the requirement, we can fill in the template which gives the rules below for the `aes.csr.control` register. Verify information flow assuming no access for untrusted agents and update rules based on design specification if incorrect violations are seen.

```plaintext
assert iflow ( aes.csr.control /=> hrot_iface.$all_outputs );

assert iflow ( hrot_iface.$all_inputs /=> aes.csr.control );
```
CWE-1269: Product Released in Non-Release Configuration

Description

The product released to market is released in pre-production or manufacturing configuration.

Radix Security Rule Template

*Note:* Radix detects pre-silicon vulnerabilities, however, we consider errors in the process (such as faulty scripts to be run by the OEM) that would result in release of devices in an insecure configuration to be covered under this CWE. Radix can be used to verify that asset protection is implemented correctly with respect to manufacturing status.

```plaintext
assert iflow ( {{Asset}} /==> {{User-visible signals}}
unless ( {{Manufacturing is not complete}} );

assert iflow ( {{User-visible signals}} /==> {{Asset}}
unless ( {{Manufacturing is not complete}} );
```

Rule Template Detail

Information from/to *Asset* must not flow to/from *User-visible signals* unless *Manufacturing is not complete*.

Security Rule Types

- CONFIDENTIALITY_ASSET
- INTEGRITY_ASSET
- ISOLATE_ASSET
Detection Example

The SoC has status registers that aid in debug and development that should not be readable after the device is manufactured. For example, the `tmcu.csr.info` register should not be readable by any agent after manufacturing. After manufacturing, the fuse `otp.manufacturing_done` is blown, i.e. set to 1 to indicate that manufacturing is done. A separate post-manufacturing verification step ensuring the fuse is blown is required.

Threat Model

An untrusted agent is able to read status registers that contain sensitive information in the manufactured device.

Security Requirement

Status registers containing sensitive information should not be readable when the manufacturing done fuse is blown.

Completing the Template Based on Design Signals and Security Requirement

From the requirement, we can fill in the template which gives the rule below.

```plaintext
assert iiflow (
    tmcu.csr.info
    =/=>
    tmcu.$all_outputs
    unless (otp.manufacturing_done == 0)
);  
```

*Note:* All registers that should not be readable post manufacturing needs to be listed as source in one or many Radix rules.
CWE-1270: Generation of Incorrect Security Tokens

Description

The product implements a Security Token mechanism to differentiate what actions are allowed or disallowed when a transaction originates from an entity. However, the Security Tokens generated in the system are incorrect.

Radix Security Rule Template

*Note:* Radix security rules can be written to ensure that the access policy is followed and that an agent cannot change its own security identifier. See security rule example for CWE-1259.

```plaintext
assert iflow ( {{access policy register == 0}} && {{boot done}} );
```

Rule Template Detail

Any access policy register should be set to 0 after system boot i.e. no access to any protected resources. This is likely more restrictive than intended but will report incorrect settings by software. The user should review the design specification and update the rule accordingly.

Security Rule Types

- ISOLATE_ACCESS_CONTROL_MECHANISM
- VERIFY_ACCESS_CONTROL_CONFIG

Detection Example

Access to secret registers such as AES key registers is configured in access-policy registers and is determined based on security identifiers. Each agent in the SoC has a Security identifier register which is programmed to a unique value by the `tmcu` during secure boot.
Threat Model

If the security identifiers are programmed incorrectly an un-trusted agent may get access to sensitive data.

Security Requirement

All security identifiers should be set to 0 after boot meaning no access.

Completing the Template Based on Design Signals and Security Requirement

From the requirement, we can fill in the template which gives the rule below for the core0 processor. The rule will fail if the boot process is simulated since boot_done will be 0 so additional qualifying signals may be required.

```
assert iflow (core0.csr.access_policy = 0 && tmcu.csr.boot_done);
```

*Note:* This security rule needs to be created for each un-secure bus master in the SoC.

**CWE-1271: Uninitialized Value on Reset for Registers Holding Security Settings**

Description

Security-critical logic is not set to a known value on reset.

Radix Security Rule Template

*Note:* This CWE is more effectively checked by a RTL lint tool as it doesn't require specifying each register to check, instead all uninitialized registers are reported. Checking unknown values in specific registers may be done in SVA or Radix Security Rules.

```
assert iflow ({{Security critical control register !== 'bX}} && {{device reset de-asserted}});
```

Rule Template Detail

Security critical control registers should not have unknown values after the device is reset.

Security Rule Types

- VERIFY_REGISTER
- VERIFY_SECURITY_STATE
Detection Example

The flip flop implementing the `debug.csr.enable` register is not connected to reset.

**Threat Model**

The `debug.csr.enable` bit is unknown after reset. In the real chip, it will randomly take the value 0 or 1 meaning the chip may be in debug mode after reset. An attacker can repeatedly reset the chip until debug mode is enabled.

**Security Requirement**

All security critical control registers should be set to a known value during reset.

**Completing the Template Based on Design Signals and Security Requirement**

From the requirement, we can fill in the template which gives the rule below assuming that reset is active high.

```plaintext
assert iflow (debug.csr.enable !== 1'bX && (~reset));
```

**CWE-1272: Sensitive Information Uncleared Before Debug/Power State Transition**

**Description**

Sensitive information may leak as a result of a debug or power state transition when information access restrictions change as a result of the transition.

**Radix Security Rule Template**

```plaintext
assert iflow ({{Security-critical signals}} /==>
```
Rule Template Detail

Security critical signals should not flow to *User-accessible signals* e.g. read by software, unless the device is operating in a privileged operating mode. If the operating mode is changed to "user mode" or "low power mode" the security critical signals should not be accessible.

Security Rule Types

- CONFIDENTIALITY_ASSET
- ISOLATE_ASSET

Detection Example

When running in privileged mode, the *dma* will copy security sensitive data from *otp.data* to registers in the *aes*. However, it first copies data from *otp* to *sram* and then from *sram* to *aes* registers. When the device transitions from privileged mode to "user mode" or "low power mode" the data that was moved must not flow outside the Hardware Root of Trust (HRoT).

Threat Model

When the device transition from one mode to another, sensitive data may remain in registers or memory locations that are readable by un-trusted agents in the new mode and thus leak sensitive data

Security Requirement

Security sensitive data accessed in one operating mode must not be accessible when transitioning to another operating mode.

Completing the Template Based on Design Signals and Security Requirement

From the requirement, we can fill in the template which gives the rule below.
assert iflow ( 
otp.data
==>
hrrot_iface.$all_outputs
unless (tmcu.csr.priv_mode) );

CWE-1273: Device Unlock Credential Sharing

Description
The credentials necessary for unlocking a device are shared across multiple parties and may expose sensitive information.

Radix Security Rule Template
Radix currently does not cover this CWE, refer to the MITRE website for suggested mitigations.

CWE-1274: Insufficient Protections on the Volatile Memory Containing Boot Code

Description
The protections on the product's non-volatile memory containing boot code are insufficient to prevent the bypassing of secure boot or the execution of an untrusted, boot code chosen by an adversary.

Radix Security Rule Template

assert iflow ( 
{{Signals storing boot code}}
==>
{{User-accessible signals}} );

assert iflow ( 
{{User-accessible signals}}
==>
{{Signals carrying boot code}}
unless ( {{Boot is complete}} ));
Rule Template Detail

For confidentiality, information must not flow from the boot code storage location (Signals storing boot code) to User-accessible signals. For integrity, information from User-accessible signals must not flow to Signals carrying boot code from User-accessible signals unless Boot is complete.

Note: confidentiality and integrity rules are subtly different. For confidentiality, the storage signals (typically an RTL model of a boot ROM) are the source. This differs from the integrity targets, which are the many signals carrying boot code that might be interfered with prior to full boot.

Security Rule Types

• INTEGRITY_FSM_STATES
• ISOLATE_ASSET

Detection Example

As part of the secure boot process in the SoC above, the tmcu fetches bootloader code from non-volatile memory, tnvm and writes it to sram.

Threat Model

If the device has insufficient protections, an adversary could read the bootloader code from tnvm memory, modify it or replace it and write it to sram before boot is complete and thus have the system boot using malicious code.

Security Requirement

The memory storing boot code should not be readable by un-trusted agents and the sram storing boot code should not be writable by un-trusted agents until boot is completed.

Completing the Template Based on Design Signals and Security Requirement

From the security requirement we can fill out the rule template which gives the rules below. The values for BOOT_CODE_ADDR_HI and BOOT_CODE_ADDR_LO are constants provided by the user.
assert iflow (
  tnvm.mem
  =/>
  hrot_iface.$all_outputs );

assert iflow (
  hrot_iface.$all_outputs
  =/>
  sram.mem[BOOT_CODE_ADDR_HI:BOOT_CODE_ADDR_LO]
  unless ( tmcu.csr.boot_complete ) );

CWE-1276: Hardware Child Block Incorrectly Connected to Parent System

Description
Signals between a hardware IP and the parent system design are incorrectly connected causing security risks.

Radix Security Rule Template

Note: Many functional testing strategies can be used to check for correct connectivity in general. The template below covers the security use case where known security-critical signals will be isolated from those that are user-accessible when the hardware block is correctly connected.

assert iflow (  
  {{Security-critical signals of hardware block}}
  =/>  
  {{User-accessible signals}} );

assert iflow (  
  {{User-accessible signals}}
  =/>  
  {{Security-critical signals of hardware block}} );

Rule Template Detail
Information should not flow from/to the Security-critical signals of hardware block to/from User-accessible signals.

Security Rule Types
• CONFIDENTIALITY_ASSET
• INTEGRITY_ASSET
• ISOLATE_ASSET

Detection Example

The interconnect in the untrusted sub-system of the SoC uses a security level bit to
determine if a transaction is secure or not. core0 is configured as secure and core1 is
configured as un-secure by the tmcu at boot time. The nvm memory uses the security
level bit to determine if a read transaction is allowed or not. If security_level == 0, a read
is allowed, if security_level == 1, it is not. During implementation the nvm.security_level
input is incorrectly tied to 0 instead of connected to the corresponding interconnect
signal. This means all masters, even un-secure ones will be allowed to read the nvm.

Threat Model
Malicious software running on an un-trusted core will be able to access secure data
since access control is effectively disabled.

Security Requirement
Trusted data must not flow to un-trusted agents based on the security level of the
master. Use the security level programmed in the destination to detect illegal data flows
caused by incorrect connections of the IP.

Completing the Template Based on Design Signals and Security Requirement

From the security requirement we can fill out the rule template which gives the rule
below:

```c
assert iflow (
    nvm.mem
when (core1.csr.security_level == 1)
==> core1.$all_inputs );
```

Note: Additional rules will be required for all masters connected to the interconnect.
**CWE-1277: Firmware Not Updateable**

**Description**
A product's firmware cannot be updated, leaving weaknesses present with no means of repair and the product vulnerable to attack.

**Radix Security Rule Template**
Radix currently does not cover this CWE, refer to the MITRE website for suggested mitigations.

**CWE-1278: Missing Protection Against Hardware Reverse Engineering Using Integrated Circuit (IC) Imaging Techniques**

**Description**
Information stored in hardware may be recovered by an attacker with the capability to capture and analyze images of the integrated circuit using techniques such as scanning electron microscopy.

**Radix Security Rule Template**
Radix currently does not cover this CWE, refer to the MITRE website for suggested mitigations.

**CWE-1279: Cryptographic Operations are run Before Supporting Units are Ready**

**Description**
Performing cryptographic operations without ensuring that the supporting inputs are ready to supply valid data may compromise the cryptographic result.

**Radix Security Rule Template**
```
assert iflow ( {{Crypto IP outputs}}
when ( {{Crypto IP self-test not passed}} )
=/>= {{User-accessible signals}} );
```
Rule Template Detail

Information should not flow from the Crypto IP outputs to User-accessible signals when the Crypto IP self-test not passed.

Security Rule Types

- ISOLATE_ASSET
- VERIFY_SECURITY_STATE

Detection Example

The aes block runs a self-test after reset to ensure system integrity. The aes engine must not be used unless the self-test passes. This means no read or write accesses are allowed from untrusted agents outside the Hardware Root of Trust (HRoT).

Threat Model

A malicious actor is able read and write data to and from aes before the self-test is completed and has passed and may alter the state or access privileged data.

Security Requirement

There should be no access to the crypto block by untrusted agents until it's self-test passes.

Completing the Template Based on Design Signals and Security Requirement

From the requirement, we can fill in the template which gives the rules below:

```plaintext
assert iflow (aes.$all_outputs when (aes.self_test_passed == 0) => hrot_iface.$all_outputs);
assert iflow (hrot_iface.$all_outputs)
```
when \((\text{aes.self\_test\_passed} == 0)\)  
\(\implies\)  
\(\text{aes.$all\_inputs}$\);

### CWE-1280: Access Control Check Implemented After Asset is Accessed

**Description**

A product's hardware-based access control check occurs after the asset has been accessed.

**Radix Security Rule Template**

```plaintext
assert iflow (  
    {{Asset}}  
  ===>  
  {{Asset point of use}}  
  unless ( {{Access control check is successful}} ));
```

**Rule Template Detail**

Information about the `Asset` must not reach the `Asset's point of use` unless the `Access control check is successful`. Note: Finding bad coding styles and missing reset values should be done as part of functional verification.

**Security Rule Types**

- INTEGRITY_ACCESS_CONTROL_MECHANISM
- ISOLATE_ASSET
- VERIFY_ACCESS_CONTROL_CONFIG

**Detection Example**
Read access to the `sram.csr.configuration` register is only allowed for the `tmcu`.
Hardware access control checks that the bus master doing the read is valid before the register value is driven to the output.

**Threat Model**

If the register value is driven on the bus before the access control check is complete, an un-trusted agent may get access to privileged information.

**Security Requirement**

Information in access controlled registers must not be available before the access control check is complete and successful.

**Completing the Template Based on Design Signals and Security Requirement**

From the requirement, we can fill in the template which gives the rules below:

```plaintext
assert iflow (sram.csr.configuration =/= sram.$all_outputs
unless ( (tbus.master == TMCU_ID) && sram.access_control_done ) );
```

**CWE-1281: Sequence of Processor Instructions Leads to Unexpected Behavior (Halt and Catch Fire)**

**Description**

Specific combinations of processor instructions lead to undesirable behavior such as locking the processor until a hard reset performed.

**Radix Security Rule Template**

Radix currently does not cover this CWE, refer to the MITRE website for suggested mitigations.
CWE-1282: Assumed-Immutable Data is Stored in Writable Memory

Description

Immutable data, such as a first-stage bootloader, device identifiers, and "write-once" configuration settings are stored in writable memory that can be re-programmed or updated in the field.

Radix Security Rule Template

```plaintext
assert iflow ( {{User-accessible signals}} =/=> {{Memory storing immutable data}} );
```

Rule Template Detail

Information from User-accessible signals should not affect the Memory storing immutable data.

Security Rule Types

- INTEGRITY_DATA
- INTEGRITY_MEMORY_REGION

Detection Example

In the SoC above, cryptographic hash digests, encryption keys, the first stage bootloader and other trusted data is stored in ROM and one-time programmable fuses. This data is not modifiable after manufacturing. Another case is if the immutable data is stored in sram because the memory type was not defined in the design specification. The immutable data is accessible in the address range [IMM_ADDR_START: IMM_ADDR_END], defined in the system specification.
Threat Model
An attacker is able to modify a hash digest stored in what was supposed to be read only memory. Any code the attacker loads after that can be verified as trusted.

Security Requirement
All immutable code or data should be programmed into ROM or write-once memory that cannot be modified in the field.

Completing the Template Based on Design Signals and Security Requirement
From the requirement, we can fill in the template which gives the rules below. The first rule may incorrectly flag a violation since there is no condition checking if data is actually written to the memory as opposed to data flowing to the data input of the memory only. However, any write attempt to immutable data may indicate a security vulnerability. The second rule doesn’t specify which physical memory data must not flow to, only the address range that should not be writable.

```
assert iflow ( 
    tbus.$all_inputs 
    =/> 
    {rom.mem.data, otp.mem.data} );

assert iflow ( 
    hrot_iface.data 
    when (tbus.addr <= IMM_ADDR_END && tbus.addr >= IMM_ADDR_START) 
    =/> 
    tbus.data );
```

CWE-1283: Mutable Attestation or Measurement Reporting Data

Description
The register contents used for attestation or measurement reporting data to verify boot flow are modifiable by an adversary.

Radix Security Rule Template

```
assert iflow ( 
    {{User-accessible signals}} 
    =/> 
    {{Memory storing attestation and/or measurement data}} );
```
Rule Template Detail

Information from User-accessible signals should not affect the Memory storing attestation and/or measurement data.

Security Rule Types

- INTEGRITY_MEMORY
- INTEGRITY_REGISTER
- INTEGRITY_SECURITY_STATE

Detection Example

The final hash value calculated on the code during secure boot is written to a register by the tmcu. This value is readable by the untrusted cores core{0-N} and by the debug interface. It is not writable by any untrusted agent or debug module.

Threat Model

An attacker is able to modify the final hash values due to a design bug so that malicious code that failed verification will now pass and it appears that secure boot passed even though the system is running malicious software.

Security Requirement

Measurement reporting data such as the final hash value should be stored in read only registers or have access protection to prevent modification.

Completing the Template Based on Design Signals and Security Requirement

From the requirement, we can fill in the template which gives the rule below.

```plaintext
assert iflow (
    {hrot_iface.$all_outputs, debug.$all_outputs}
    /==>
    tmcu.csr.hash);
```
## Appendix: Security Rule Types

Prefixes are correlated with security objective

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Common Rule Pattern</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONFIDENTIALITY</td>
<td>asset =/&gt; {attacker_observables}</td>
<td>no-flow to outputs</td>
</tr>
<tr>
<td>INTEGRITY</td>
<td>{attacker_controllables} =/&gt; asset</td>
<td>no-flow from inputs</td>
</tr>
<tr>
<td>ISOLATE</td>
<td>asset =/&gt; {invalid_outputs} {invalid_inputs} =/&gt; asset</td>
<td>Two rules, logical union of confidentiality and integrity</td>
</tr>
<tr>
<td>VERIFY</td>
<td>asset == secure_value</td>
<td>asset value does not enable violation of security objectives</td>
</tr>
</tbody>
</table>

Suffixes are correlated with asset type

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCESS_CONTROL_CONFIG</td>
<td>Mixture of ACL values and derived configuration info. May be implicit. Ask about external information sources such as files input to testbenches. Likely use-case for VERIFY prefix</td>
</tr>
<tr>
<td>ACCESS_CONTROL_MECHANISM</td>
<td>Stateful logic, may require widgets to interact with or inject taint or observe FSM states. ACLs to firewalls may require multiple rules for each whitelisted endpoint pair (src/dest, master/slave, host/device)</td>
</tr>
<tr>
<td>ASSET</td>
<td>Refers to general assets requiring either a little or a lot of design-specific customization over the rule pattern</td>
</tr>
<tr>
<td>DATA</td>
<td>Usually regions in ROM or NVM containing data or local SRAM or TCM. For each data-at-rest location track all nearest data-carrying signals</td>
</tr>
<tr>
<td>FSM_STATES</td>
<td>Track the next state and add exceptions for transitions e.g. next_state =/&gt; current_state unless ( list_of_conditions )</td>
</tr>
<tr>
<td>MEMORY_REGION</td>
<td>A shared region of a memory resource, does not refer to the data in that region, often with ISOLATE-style rules for non-interference. Beware physical address aliasing: ensure that all memory ranges that refer to the physical region are tracked.</td>
</tr>
<tr>
<td>REGISTER</td>
<td>Likely not register itself, security bugs residing in “sticky” registers, locks, lock ranges, and the control logic are found following the pattern inputs =/&gt; my_reg unless ( condition )</td>
</tr>
<tr>
<td>SECURITY_STATE</td>
<td>ISOLATE-style rules when combined with CONFIDENTIALITY. Similar to REGISTER plus CONFIDENTIALITY to ensure state registers are secret (when applicable) and incorruptible.</td>
</tr>
</tbody>
</table>
Legal Notices

Terms of Use for Materials from the MITRE CWE Database

CWE™ is free to use by any organization or individual for any research, development, and/or commercial purposes, per these CWE Terms of Use. The MITRE Corporation ("MITRE") has copyrighted the CWE List, Top 25, CWSS, and CWRAP for the benefit of the community in order to ensure each remains a free and open standard, as well as to legally protect the ongoing use of it and any resulting content by government, vendors, and/or users. CWE is a trademark of MITRE. Please contact cwe@mitre.org if you require further clarification on this issue.

LICENSE

CWE Submissions: By submitting materials to The MITRE Corporation’s ("MITRE") Common Weakness Enumeration Program (CWE™), you hereby grant to MITRE a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to use, reproduce, prepare derivative works of, publicly display, publicly perform, sublicense, and distribute your submitted materials and derivative works. Unless otherwise required by applicable law or agreed to in writing, it is understood that you are providing such materials on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE.

CWE Usage: MITRE hereby grants you a non-exclusive, royalty-free license to use CWE for research, development, and commercial purposes. Any copy you make for such purposes is authorized on the condition that you reproduce MITRE’s copyright designation and this license in any such copy.

DISCLAIMERS

ALL DOCUMENTS AND THE INFORMATION CONTAINED IN THE CWE ARE PROVIDED ON AN “AS IS” BASIS AND THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS OR IS SPONSORED BY (IF ANY), THE MITRE CORPORATION, ITS BOARD OF TRUSTEES, OFFICERS, AGENTS, AND EMPLOYEES, DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION THEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.

IN NO EVENT SHALL THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS OR IS SPONSORED BY (IF ANY), THE MITRE CORPORATION, ITS BOARD OF TRUSTEES, OFFICERS, AGENTS, AND EMPLOYEES BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE INFORMATION OR THE USE OR OTHER DEALINGS IN THE CWE.

Tortuga Logic Terms of Use and Copyright

This Radix Coverage for Hardware Common Weakness Enumeration (CWE) Guide ("Guide"), including all material and information included in this Guide, is provided on an “as is” basis. Tortuga Logic disclaims all warranties of any kind, either express or implied. Tortuga Logic shall not be liable for any direct, indirect, punitive, incidental, special, or consequential damages that result from the use of, or inability to use, this Guide.

TORTUGA LOGIC is a trademark of Tortuga Logic, Inc. Other product or company names are the property of their respective owners. They are used in the Guide for identification purposes only and do not imply endorsement or affiliation.