More than 2 years later, here we are: a new write-up! And what a write-up!

It's time to dust off this old blog of mine and buckle up for what has to be one of the most intricate and fun challenges I've ever been a part of. I'm talking, of course, about exploiting the **Nvidia Falcon** microprocessor. I'll be joined by no one other than my good friend and collaborator **SciresM** who co-authors this write-up and, among others, played a huge role in reverse engineering and exploiting this exotic microprocessor.

Without further ado, let's start by going back to the beginning of 2018 when it all began...

The Beginning

The year is 2018. A little less than a month ago **plutoo**, **derrek** and **naehrwert** had just presented their Chaos Communication Congress talk about the Nintendo Switch and we all learnt about how they exploited bugs at different execution levels to compromise and, ultimately, defeat the console's security. You might recall that one of the exploit chains mentioned in the talk has been published here in a previous post. I'm naturally talking about **nvhax**, which consisted in exploiting multiple bugs to escalate from the browser's sandbox and into the **nvservices** system module. This was a very desired target back then due to its direct access to the GPU which, if you recall from my previous post, led to a full system compromise thanks to a **SMMU** bypass documented in the Tegra X1's **TRM**. Ah, good times!

Anyway, while efforts to further exploit the system were ongoing, I decided to deviate from that path for a bit and investigate what other devices could be accessed from within **nvservices**. One such device is the **TSEC** (Tegra SEcurity Coprocessor) whose primary purpose is to handle **HDCP**, but it has been historically repurposed for a few other tasks and this is where things get interesting.

The Switch's **eMMC** was dumped very early on and since a portion of the **bootloader** used to be stored in plaintext (see here), we were able to analyse it almost immediately and discover something... weird. You see, one of the first tasks the Switch's **bootloader** is in charge of is to generate keys. These keys are part of the console's cryptosystem which is further detailed here: here. One of these keys is a special console-unique key which, contrary to others, is derived inside the **TSEC**.

By reverse engineering the **bootloader**, we figured out how to initialize and talk to the **TSEC**. Furthermore, we could extract the firmware code that the bootloader would upload to the **TSEC** and analyse it offhand. Not much information is made public by **Nvidia** regarding the **TSEC** due to its purpose in handling secret **HDCP** keys, but, fortunately for us, the nouveau team had taken upon the task of deciphering this obscure device way before any of this. Armed with their knowledge on the subject, we now knew the **TSEC** is essentially a **Falcon** (FAst Logic CONtroller) microprocessor bundled with a few other peripheral device blocks. Falcon has its own proprietary architecture, but once again nouveau comes to the rescue, this time thanks to the envytools project which provides a **Falcon** disassembler and

documentation gathered from reverse engineering other **Falcon** based devices found inside various **Nvidia** GPU models over time.

Even though we could disassemble and study this firmware for a long time, it wasn't until **nvhax** that we could finally experiment with the **TSEC** and play around with its **Falcon** microprocessor. Using the newfound access **nvservices** gave us, I began poking at the **TSEC**'s MMIO region until I was able to fully upload firmware code into it and then I found the first of many bugs we would come to uncover over the next years...

TSEC and Falcon - A Crash Course

Before diving into specifics, let us first take a look into what exactly *is* the **Falcon**.

As mentioned before, **Falcon** stands for FAst Logic CONtroller and is a proprietary microprocessor developed by **Nvidia**. It was originally developed to replace **Xtensa** microprocessors used in the **VP2** video decoding acceleration engine found in GPUs from series G84 to series G96. As such, the very first appearance of a Falcon was in GPUs from series G98, where it served as the main controlling logic behind the new **VP3** engine. However, as GPUs grew in complexity, Nvidia began using and repurposing the Falcon for many other engines that make up a GPU.

I strongly recommend reading envytools' documentation on the Falcon for a deeper understanding of its origins and purposes.

Borrowing from their invaluable documentation, we can attest that a single Falcon powered unit is made of:

- The core CPU with separate code and data SRAM.
- MMIO space where control registers for the Falcon and other subunits are mapped to.
- Timer, watchdog and interrupt logic.
- An ICD (in-circuit debugger).
- An optional FIFO interface known as **METHOD**.
- An optional memory interface for external data transferring.
- An optional secure coprocessor known as **SCP**.
- Any additional logic specific to what the Falcon is supposed to control (video decoding, image compositing, etc.).

We also know that the Falcon has gone through several iterations, starting off with version 0 (found first in G98 GPUs) and up to version 6 (found in the latest Pascal and Volta GPUs). We will be focusing on version 5.1 during this write-up as this is the version of Falcon units found inside the Switch's SoC.

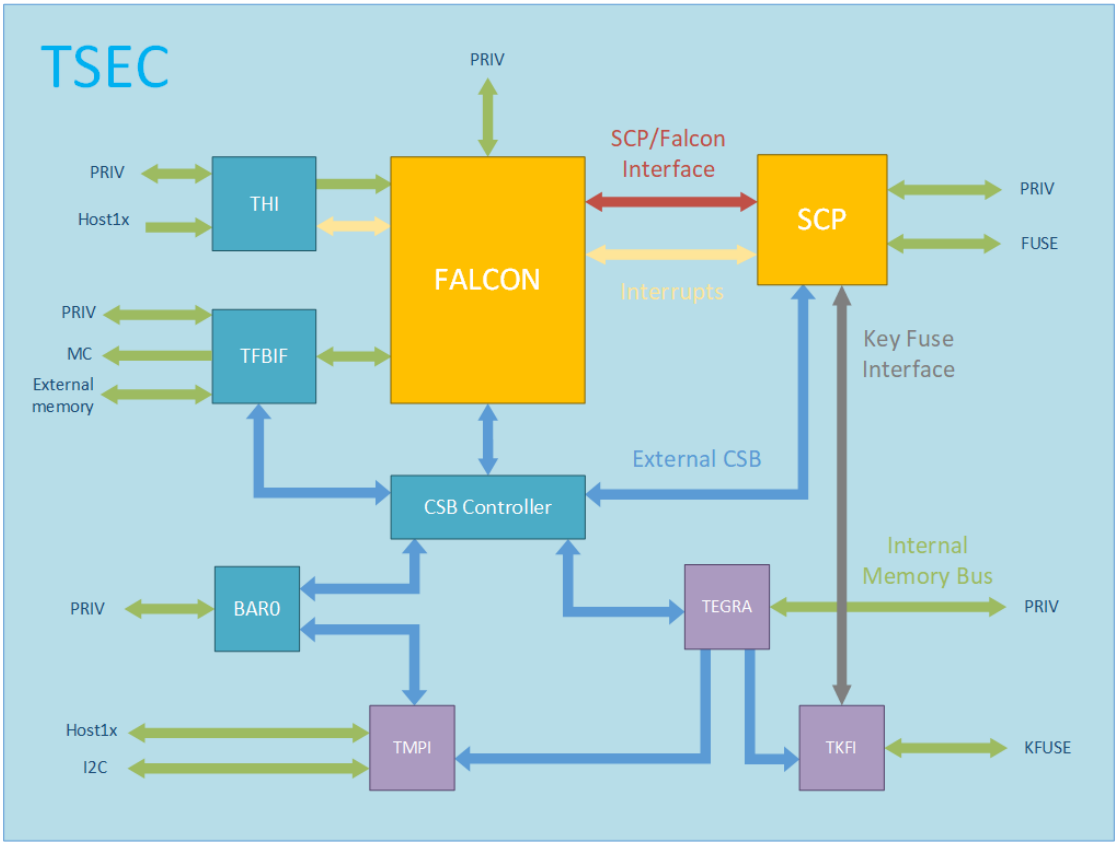
As it's known, the Switch's SoC is essentially a Tegra X1 (and later a Tegra X1+) which contains the following Falcon-based units:

- **TSEC** (also known as **TSECA** and formerly **SECOP**) **-** Tegra Security Co-processor, an embedded security processor used mainly to manage the HDCP encryption and keys on the HDMI link.
- **TSECB -** A second, unused instance of the TSEC for **Miracast** and **NVSI**.
- **NVDEC -** Video Decode Engine.
- **NVENC -** Multi Standard Video Encoder.
- **NVJPG -** JPEG Engine.
- **VIC -** Video Compositor.
- **USB3 -** The USB3/XUSB controller.
- **GPU PMU -** The GM20B Maxwell GPU's Power Management Unit.
- **GPU FECS -** The GM20B Maxwell GPU's Front-End Context Switch.
- **GPU GPCCS -** The GM20B Maxwell GPU's Graphics Processing Cluster.

Due to its non-standard role during the Switch's boot process, the **TSEC** was the primary focus of this write-up and as such it became pivotal in researching and documenting the Falcon itself and several subengines. Nonetheless, a good portion of the concepts applied to the TSEC can be mirrored to other units since they mostly only differ at the subengine level.

It comes without saying that, due to its proprietary nature, very little is known about TSEC or even Falcon itself. Amassing knowledge on these topics consisted of hunting for scarce references in public, open-source projects from **Nvidia** (for example, from the Tegra Gitweb) and cross-referencing with the **envytools** project. When combined with our own research, conducted while reverse-engineering the TSEC inside the Switch's SoC, this would culminate in an extensive and comprehensive analysis which has been being published over the years in its own page at the SwitchBrew Wiki.

To help visualize how we believe the TSEC looks like, I've put together a block diagram illustrating each of its components and how they link to each other.

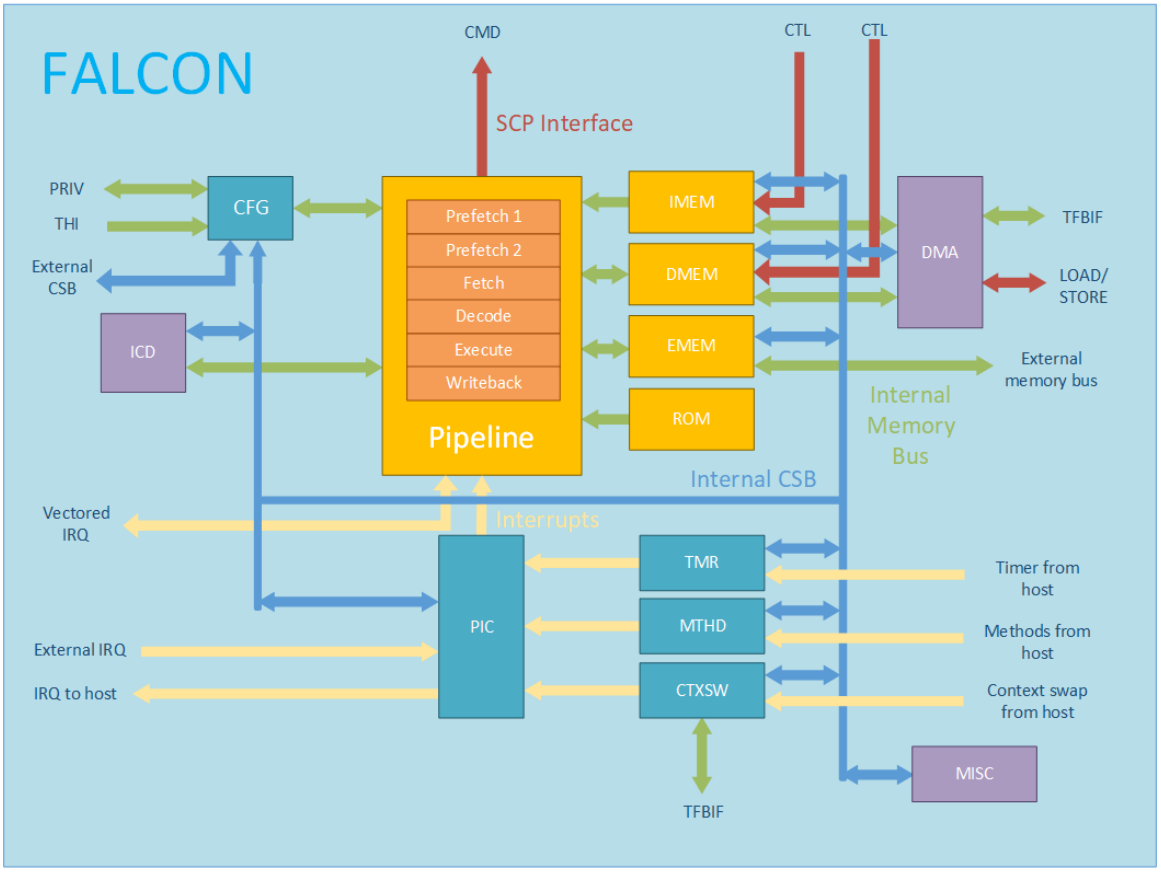


We've got some ground to cover here, so let's break the diagram down and analyse each block one by one.

Falcon

This is the base Falcon microprocessor. Since it was designed for multiple purposes, Falcon's hardware design can vary depending on which unit bundles it, but thanks to a keynote presented by Nvidia's **Joe Xie** in 2016 (on the topic of adopting RISC-V for their future security oriented hardware) we had the rare chance to see how Falcon looks like in the form of a block diagram.

Expectedly, the official diagram simplifies some aspects and hides away specifics, so I've put together my own version for what we believe the Falcon inside TSEC looks like.



We can observe the CPU itself follows the classic RISC pipeline, but with a six-stage instruction cycle: **prefetch 1**, **prefetch 2**, **fetch**, **decode**, **execute** and **writeback**. Falcon operates with a proprietary, variable length ISA that goes by several names such as "ucode" (used by **Nvidia**) or "fuc" (used by **nouveau**/**envytools**). Reverse-engineered documentation and tooling for this ISA is available from the **envytools** project here and here.

Programming the Falcon heavily depends on MMIO interaction with the host. As illustrated by the diagram, the **CFG** block interacts with **PRIV** which is what Nvidia calls the MMIO register space. Every time the host reads or writes a specific MMIO (or PRIV) register, the **configuration** controller ensures the access is mapped to the right place. This simple MMIO protocol is also called **CMEM**, as an allusion to **configuration memory**. On the other hand, the CPU itself also needs to access CMEM from its side and it does so by using specific instructions for reading or writing MMIO registers. These instructions trigger requests to the **CSB** or **configuration space bus** which is a low-bandwidth slave interface used internally on the circuitry.

Falcon's main memory consists of SRAM banks divided in **instruction memory** or **IMEM** and **data memory** or **DMEM**. All code is expected to be pre-emptively loaded to IMEM before starting the CPU. This can be achieved either by poking the **IMEMC**/**IMEMD**/**IMEMT** MMIO registers and uploading one word of code at a time, or by programming the **DMA** engine for transferring blocks of code from the host into IMEM. The **EMEM** block is essentially an interface for external memory and not actual SRAM.

A few more blocks can be observed such as the **TMR** which controls programmable and watchdog timers, the **MTHD** which controls a dedicated FIFO interface for submitting

"methods" and the **CTXSW** which controls context swapping. All these blocks generate their own interrupt signals which are handled by the **PIC**. The PIC also supports and routes interrupts from external sources, two of which (**SWGEN0** and **SWGEN1**) can be freely programmed by the user.

The two remaining blocks **ICD** and **MISC** are, respectively, an **in-circuit debugger** and a reserved, **miscellaneous** interface for future extensions to the hardware if deemed necessary.

In its generic form, Falcon is fully operational, programmable and controllable from the get-go. However, Falcon also provides first class support for a cryptographic accelerator called **SCP**. Only a handful of Falcon-based units, usually called **secretful** units, embed the SCP and TSEC is one of them. This means the Falcon inside TSEC supports **security modes**, which are explained to some extent by Nvidia itself here.

In practice, this means secretful units can operate in one of three modes:

- **Non-secure (NS) -** This is the most basic security mode, available to all Falcon units. In this mode Falcon works as if the SCP doesn't exist. As soon as the CPU is signalled to start, code execution fires right away from IMEM at its boot vector (address 0) and is fully controlled by the host.
- **Heavy Secure (HS) -** This is the most advanced security mode, available only to secretful Falcon units. In this mode Falcon becomes a black box, blocking accesses from the host to its MMIO register space. Entering this mode requires loading microcode officially signed by Nvidia.
- **Light Secure (LS) -** This is an intermediate security mode, available only to secretful Falcon units. In this mode Falcon works similarly as if in NS mode but is granted exclusive privileges when accessing external memory. Entering this mode requires elevating to HS mode first and initiating the transition to LS mode through a dedicated MMIO register, writable only in HS mode.

Naturally, it becomes extremely pertinent to analyse how HS mode works. According to Nvidia:
"The loading process involves tagging the IMEM block as secure, writing the signature into a Falcon register, and starting execution. The hardware will validate the signature, and if valid, grant HS privileges."

But... how *exactly*? The process consists of the following steps:

1. Start off in NS mode. The host uploads non-secure microcode to IMEM and starts the CPU.
2. Load secure microcode. The running non-secure microcode must upload secure microcode to IMEM by setting a specific bit (**IMEMC_SECURE**) when using the IMEMC/IMEMD/IMEMT MMIO registers.
3. Load the secure microcode's signature. The running non-secure microcode must write the 16-byte signature to a specific register. Secretful Falcon units expose a set of 8, 16-byte sized, registers (**$c0** to **$c7**) which point to SCP's internal SRAM. Register **$c6** must contain the signature value before attempting to escalate to HS mode.
4. Instruct Falcon on microcode's properties. The running non-secure microcode must program **SEC**, one of the core Falcon's **special purpose registers** or **SPRs**, with the secure microcode's starting address and size. Optionally, bit 17 of the **SEC** SPR may

also be set, which instructs the authentication process to use a secret **hardware encryption** layer: microcode will be treated as if **encrypted** with a **hardware secret** and the authentication process will attempt to decrypt it in place.

5. Jump to secure microcode. The running non-secure microcode must jump to any address that falls within the loaded secure microcode.
6. ?????
7. Profit! Falcon should now be executing the secure microcode under HS mode **if and only if** the loaded signature is valid.

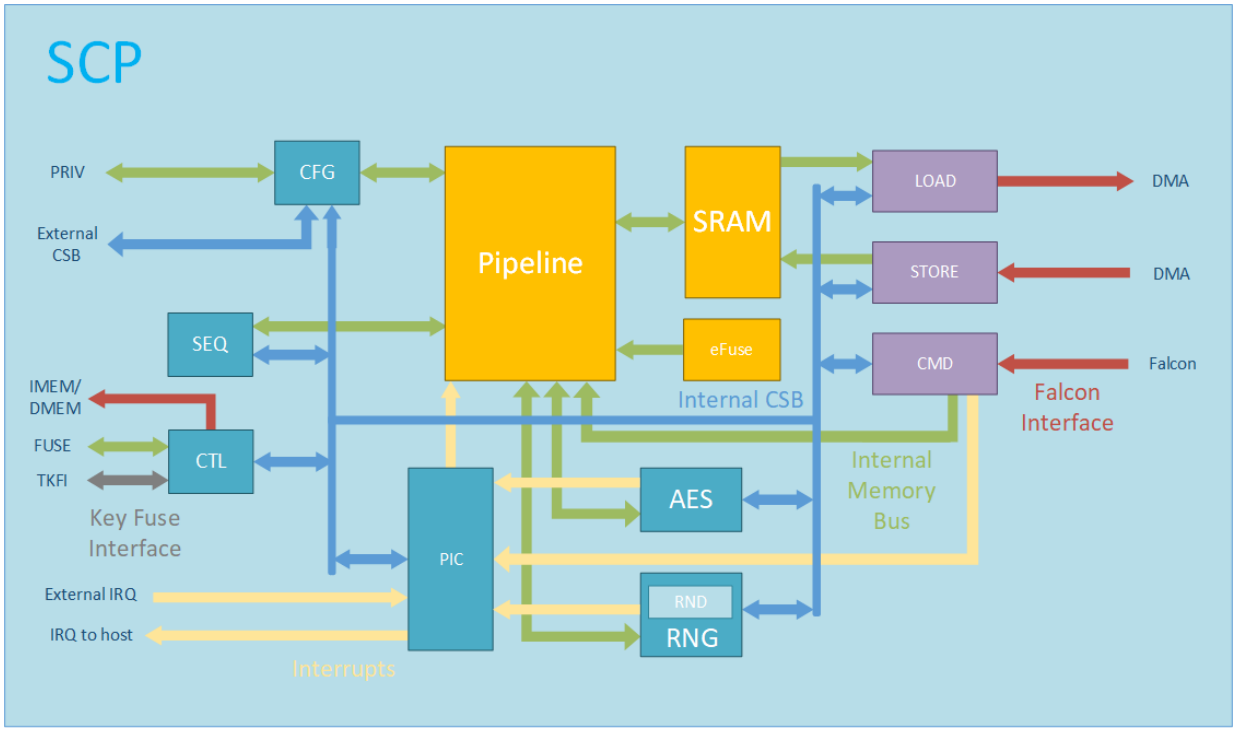
However, this raises a question: if Falcon boots straight off IMEM with host controlled, non-secure microcode, how is the secure microcode's signature validated?

To understand how this happens we must look into the other components, especially the SCP and its role in secretful Falcon units.

SCP

The **Secure Co-Processor** is an optional cryptography extension to the main Falcon core. This is by far the less understood component of Falcon-based units and virtually no documentation or information on it is publicly available.

I've put together a block diagram illustrating what we know about it from reverse-engineering the unit found inside the TSEC.



Contrary to the Falcon core, we do not have a clear idea on how the SCP's pipeline is organized. Instructions are called **commands** and arrive to SCP from a dedicated interface with Falcon. On the Falcon side, these commands correspond to a subset of instructions perceived as an extension to the main Falcon ISA. When the Falcon CPU fetches and decodes one of these instructions it stalls its own pipeline and submits a command to the SCP's **CMD**

block which, in turn, translates it for the SCP's pipeline to execute. The **envytools** project, specifically **mwk**, pioneered the research and documentation of this coprocessor here. But we have since been able to complement their findings with our own research here.

SCP's core logic and main purpose is to provide hardware-accelerated crypto operations. The **AES** block is instructed by the SCP's pipeline and has the sole purpose of providing an **AES-128-ECB** hardware implementation. The **RNG** block is also instructed by the SCP's pipeline, encapsulates a true random number generator or **RND** and provides additional logic for controlling and fine-tuning its circuitry. Both AES and RNG blocks generate interrupt signals on completion which are handled by the **PIC**. The PIC can route interrupt signals back to Falcon which sees them as coming from an external source or **EXT**. Additionally, the **SEQ** block provides logic for recording and executing **sequences** of crypto operations in the form of macros which are constructed by the pipeline when executing specific instructions.

The SCP also has its own **CFG** block to expose a handful of MMIO registers which can be accessed by the host via **PRIV** and by the Falcon via the **CSB**. This MMIO register space provides a way to debug and control certain aspects of the SCP's internal circuitry.

SCP's internal memory is mostly a mystery since it cannot be directly accessed. Nonetheless, it's reasonable to speculate the SCP contains at least one SRAM bank for the pipeline to operate on. The **LOAD** and **STORE** blocks interact with the Falcon's **DMA** block to provide a method for reading and writing memory between Falcon and the SCP. On the Falcon's side, DMA transfers use specific instructions, but their target is normally the host's external memory. However, Falcon provides a special purpose register called **CCR** which can only be configured through a special instruction. This register instructs the Falcon's DMA engine to change its target to the SCP and so, subsequent memory transfers will occur between Falcon and the SCP instead.

Possibly one of the most interesting features provided by the SCP are its **secrets**. One of the commands provided to Falcon allows loading a crypto register (**$c0** to **$c7**) with one of 64 secret values from non-volatile storage (**eFuse**). These values, once loaded, expose yet another peculiarity: SCP's crypto registers have **access control lists** or **ACLs**. The ACLs have been extensively documented here, but the gist of it is their values control if a crypto register is **readable**, **keyable** and/or **writable**. Additionally, ACLs propagate between crypto registers when executing certain commands.

Last but not least, the SCP contains an overseer block called **CTL**. This block controls a number of obscure, miscellaneous features of the SCP's cryptosystem such as the **debug mode**, the **lock** mechanism and the HDCP 1.x **private key** loading system.

THI

The **Tegra Host Interface** exposes the Falcon's **METHOD** logic to the outside. Through the **Host1x**, a Tegra specific module that acts as a DMA engine for register access to graphics and multimedia modules, clients can submit **methods** to a FIFO-based system that can be accessed by the **MTHD** block on the Falcon's side. This provides a more sane and faster way to process large amounts of read/write requests to MMIO regions when dealing with protocols such as HDCP or MPEG.

TSEC has a THI block for handling HDCP and context swap commands (see here) which are then processed in Falcon ucode (**nvhost_tsec.fw**). The THI has its own MMIO register space for configuring a few miscellaneous features (synchronization, clock gating, etc.) and is the only block that is not connected to the **CSB**, which means the Falcon core cannot access it directly.

TFBIF

The **Tegra FrameBuffer InterFace** is an adaptation of the GPU's equivalent **FBIF** which controls memory transfers between the Falcon core and the outside world (which used to be restricted to the framebuffer). On Tegra SoCs the framebuffer nomenclature remains, but the concept does not apply anymore. Instead, the TFBIF block provides access to any external memory source available (**IRAM** or **DRAM**) and acts as a **memory controller client interface** or **MCCIF**. It has its own MMIO register space not only for controlling and debugging memory accesses, but also to provide a **backdoor** system to bypass the **SMMU** (Tegra's IOMMU) only accessible from **Heavy Secure** mode.

BAR0

The **BAR0** is a repurposed controller for accessing certain modules' MMIO register spaces. When originally integrated in GPU logic, the BAR0 controller provided a DMA-like system to access the entire PCIe BAR0 region. On Tegra SoCs, this block provides access to other **Host1x** clients' memory space which allows TSEC, for example, to read/write SOR1 MMIO registers.

CSB

The **Configuration Space Bus** is a low-bandwidth slave interface that connects each block to a master controller. This CSB controller is what allows the Falcon core to read/write MMIO registers from other blocks.

TEGRA

The **TEGRA** block provides access to 2 interfaces available only to Tegra SoCs: the **TMPI** and the **TKFI**. Most of its MMIO register space is programmable but unknown, with only one register exposing control signals for the interfaces connected to the TEGRA block.

TMPI

The **Tegra Master Port Interface** connects the **BAR0** controller to the outside world. This allows BAR0 to work as a **Host1x** master and provides **I2C** access, which in turn allows the TSEC to use BAR0 for accessing other Host1x's clients' MMIO register spaces.

TKFI

The **Tegra Key Fuse Interface** connects the SCP's **CTL** block to the Tegra's **KFUSE**. The KFUSE hardware is an interface for an eFuse array containing encrypted per-SoC HDCP 1.x keys. While KFUSE can be externally accessed and programmed to read out these encrypted keys, the key needed to decrypt them, also per-SoC and known as **PKEY** or **PRIVKEY**, is

hidden away inside KFUSE with no known way to access it. However, through the TKFI, the SCP's CTL block is the only known component capable of reading out the PKEY. When this process is requested, SCP will populate secret 0x3F with the PKEY. We speculate this is achieved by intercepting requests to secret 0x3F and diverting access from SCP's internal eFuses to hidden MMIO registers inside the SCP's CTL block. If the PKEY is never requested, SCP's secret 0x3F is always empty by default.

Now that we understand each of TSEC's components, let's go back to the Falcon's **Heavy Secure** mode which is, undoubtedly, the crux of this entire cryptosystem.

Something the reader likely noticed is that we skipped one of the Falcon's blocks: **ROM**. This is the answer to our pending question about how a secure microcode's signature is validated. You see, Falcon doesn't really have a bootrom, which makes sense as it's intended to be a fully programmable, general purpose microprocessor. When powered up, the Falcon simply remains halted until the host uploads code to its **instruction memory** (**IMEM**) and signals the CPU to start running from the **boot vector** (hardcoded as address 0).

So why do we need ROM? According to Nvidia:
"TSEC heavy-secure (HS) hardware is capable of authenticating its own code autonomously using its Secure Boot ROM and signature verification keys. The on-chip secure memory enables tamper resistant secure storage and transaction verification. TSEC implements a random number generator (RNG), and has a Falcon engine that supports AES-128b; no other cryptographic primitives or key sizes are supported. Two independent instruction queues (capable of holding up to 16 instructions) are used to provide encryption support for DRM schemes, including protected content encryption/ decryption."

In practice, what happens is once we've followed the steps necessary for loading secure microcode and try to call it, a special hidden exception (oddly named **INV_INS** or **invalid instruction**) is raised. Falcon alerts the SCP with a dedicated interrupt signal and then switches code execution from **IMEM** to this **Secure Boot ROM**. At this point, the SCP activates the **lock** mechanism: the SCP's **CTL** block intercepts and drops all read/write accesses to SCP's MMIO register space and to Falcon's IMEM and DMEM. This effectively turns Falcon into a black box as we can no longer observe neither its internal memory nor the current SCP's state.

The Secure Boot ROM is also mapped to address 0, but no known mechanism exists to read its contents so this, effectively, classifies it as **execute-only**. Among other tasks, this ROM issues to the SCP the necessary commands to validate the secure microcode's signature. If the validation succeeds, code execution is switched back to IMEM and both Falcon and the SCP begin operating in HS mode.

Leaving HS mode can only be done by the now running secure microcode by clearing a specific SCP's MMIO register and jumping outside of the secure microcode's boundaries. This process automatically downgrades the security mode back to **non-secure** without invoking the Secure Boot ROM again.

The Bugs

So, with our crash course concluded, we can finally shift our attention to what bugs and flaws haunt the TSEC and its cryptosystem.

Our first stop is a piece of microcode, exclusive to the Nintendo Switch, that runs on its SoC's TSEC unit during the key derivation stages of the bootloader: the **HOVI firmware**. We've called it this way due to a couple strings found inside it which begin with "HOVI_". HOVI likely stands for HOrizon nVIdia as an homage to the Nintendo-Nvidia team behind its development ("Horizon" being the name of the Switch's OS and "Nvidia" being self-explanatory) and these strings would later be found to act as seeds for generating keys inside the microcode.

A detailed analysis of this firmware's code can be found here, but, in its original form, it can be summarized as a 3-stage payload made of the following:

- **Non-secure stage** – This stage runs in **non-secure** mode from address 0. It bootstraps the Falcon and uploads the next stage as **secure microcode**. Additionally, this stage copies a small block of data, necessary for the next stages, to DMEM.
- **Secure stage** – This stage runs in **heavy secure** mode. It starts by preparing the SCP and calculating an **AES-CMAC** (using a key derived from SCP's secret 0x26) hash over the previous non-secure stage. The calculated MAC is compared with a value stored in the data previously copied to DMEM and if the values mismatch, execution is aborted. Finally, this stage decrypts (**AES-128-CBC** using a key derived from SCP's secret 0x26) and uploads the next stage as **secure microcode**.
- **Secure and encrypted stage** – This stage also runs in **heavy secure** mode. It generates a key by combining SCP's secrets 0x00 and 0x3F with the seed strings (which start with "HOVI_") passed through the data previously copied to DMEM by the non-secure stage. This stage then uses the TSEC's **BAR0** to write the key in **SOR1**'s MMIO register space.

Once this firmware code is done, SOR1 will contain a key spread across 4 of its MMIO registers and the host can now read it out for using on the next bootloader's stages. If you recall the paragraph about **TKFI** from the crash course section, you'll notice secret 0x3F is used to keep the per-SoC HDCP 1.x PKEY, but only if it's requested. This is in fact done by the **secure stage** while preparing the SCP and involves setting a bit in one of the SCP's MMIO registers. Therefore, the key generated by the **secure and encrypted stage** will be per-SoC or, more broadly speaking, per-console, which explains the thought process Nintendo and Nvidia went through here: provide an extra source of secrecy for deriving console unique keys. This adds a new layer of software security as this per-console key is, in fact, a protection against **microscope attacks**, which forces attackers to compromise both the **hardware** and the **software** of the system.

This firmware was later encapsulated in more microcode, but we'll get to that in a while. For now, let's focus on the first issue we discovered involving TSEC and this specific firmware's code: the **maconstack** bug.

maconstack

Aside from being used in key derivation by the bootloader, the TSEC is also used at runtime for the task it was originally designed for: handling HDCP. The **nvservices** system module uploads firmware microcode known as "nvhost_tsec.fw" to the TSEC and it's this microcode that sets up the TSEC's Falcon for receiving HDCP related **methods** through the **THI** block which, in turn, is used by the Host1x complex for dealing with HDCP over **DisplayPort** or **HDMI**.

Shortly after we exploited the nvservices system module, got direct access to several graphics related components (GPU, NVDEC, TSEC, etc.) and figured out how to upload arbitrary microcode to the TSEC, I decided to try uploading the **HOVI firmware** and observe the TSEC's MMIO register space as well as the Falcon's IMEM and DMEM regions. Right away we found the per-console key generated at boot time could be regenerated and extracted like this, but that was far from interesting especially since at this point the entire Switch's chain of trust had already been compromised. What really stood out from these experiments was found inside the dumped Falcon's DMEM.

While not actually necessary, the HOVI firmware's **secure stage** tries to be extra cautions and ensures the **non-secure stage** hasn't been tampered with by hashing its contents and comparing with a known hash value. However, calculating this hash relies on a microcode implementation of **AES-CMAC** using SCP's native **AES-128-ECB** hardware acceleration and the SCP must always transfer the result of its operations back to Falcon's memory. This means the final, calculated hash will be stored in Falcon's DMEM at some point and it must be cleared once the comparison is done.

You see where this is going, right? They forgot to clear the calculated hash from memory and since the chunk of data where the hash to compare to is part of the non-secure stage and never subjected to any kind of validation, we can now use the secure stage as an oracle for generating the expected hash corresponding to a crafted payload and replace the hash of reference with this value. Additionally, since the underlying source of secrecy for the AES-CMAC algorithm is a key generated from one of SCP's secrets and these cannot be modified (bar a hardware revision, of course), the payload can be perpetually used for generating valid hash values.

This issue was found in January 2018 and documented a few months later here. While pretty funny, we can't say it's exactly useful. Well, at least not yet…

smmuhax

For months our focus shifted away from the TSEC because the only non-standard use it had on the Switch was generating a per-console key, which we were now able to get on demand as a far more powerful vulnerability in the **BPMP**'s (the **Arm7 CPU** also known as the **Boot and Power Management Processor**) bootrom (see CVE-2018-6242) obliterated the Switch's chain of trust.

However, things were about to change when the Switch had its 6.2.0 version firmware update. People had postulated the TSEC could've been used better in the Switch's boot chain, providing even more secrecy and thus partially restoring the chain of trust. Firmware 6.2.0 gave us exactly that as we witnessed a major reform of the **HOVI firmware** that caught everyone by surprise. Nintendo and Nvidia effectively found a way to rebuild a chain of trust now using TSEC and its secrets to protect the bootloader.

The HOVI firmware now becomes a 5-stage payload made of the following:

- **Non-secure boot stage** – This stage runs in **non-secure** mode from address 0. It behaves similarly to its previous version by bootstrapping the Falcon and copying a block of common data to DMEM, but it now jumps to a new stage: the **secure boot loader stage**.
- **Secure boot loader stage** – This stage runs in **non-secure** mode. It starts by uploading the **secure keygen loader stage** as **secure microcode**, running it and waiting for it to finish. After returning, it uploads and runs the **secure boot** stage.
- **Secure keygen loader stage** – This stage runs in **heavy secure** mode. It is identical to the **secure stage** from the pre-6.2.0 version, meaning it now uploads, decrypts and runs the **secure keygen stage**.
- **Secure keygen stage** – This stage runs in **heavy secure** mode. It is identical to the **secure and encrypted stage** from the pre-6.2.0 version, meaning it generates and outputs the per-console key.
- **Secure boot stage** – This stage runs in **non-secure** and **heavy secure** modes. It is further split into a small initialization stub, running in **non-secure** mode, which uploads the rest of the code as **hardware encrypted secure microcode** and runs it. The bulk of this stage performs complicated, timing attack resistant verifications of fundamental Tegra's MMIO register spaces such as the **Clock and Reset Controller** or **CAR**, the **Memory Controller** or **MC** and the **Power Management Controller** or **PMC**. If everything seems right, it takes on the task of halting the **BPMP** and firing off the **CCPLEX** (the **Arm Cortex-A57 CPU complex**), preparing the **Security Engine** or **SE** and using it for decrypting the Switch's bootloader. Once done, it manually overwrites the **BPMP**'s exception vectors with the bootloader's address and resumes execution.

What this all means is that code execution is temporarily diverted from the **BPMP** to the TSEC which now has unrestricted, system-wide access to all of Tegra's memory space. This allows the TSEC to implement its own cryptosystem to protect further bootloader code from being accessed or tampered with, meaning that even if the BPMP's bootrom is compromised and attackers can replicate the chain of trust, they will face encrypted bootloader code that only the TSEC is capable of decrypting. Naturally, steps were taken to ensure attackers are unable to simply coerce the TSEC into decrypting the bootloader for them, but…

The very first thing we attempted was, obviously, trying to run this new HOVI firmware under a controlled environment by leveraging the access we now had to the entire system, specifically, the BPMP's bootrom. To our surprise, the firmware was able to detect it wasn't running under the official Switch's firmware and aborted execution all together, leaving the BPMP stuck in an infinite loop. We knew something had to access the BPMP's IRAM at some point because the bootloader had to be decrypted *somewhere*. Eventually, we had the idea of dumping the **MC**'s MMIO register space in hope that it would shed some light about potential memory accesses and lo and behold, something was trying to access memory it shouldn't be able to, specifically, a CAR's MMIO register. We knew the only thing running at this point was the TSEC, so it must've been the one attempting the access which was… strange.

Tegra SoCs provide something called the **AHB Redirection Controller** or **ARC** path, which is a protocol arbiter for accessing IRAM. Normally, the IRAM is meant to be used only by the BPMP, but the boot process requires using certain peripheral blocks such as, for example,

flash media devices, which sit behind the **AHB** bus. Through this path, requests from devices connected to the AHB bus are redirected by the MC to IRAM, granting them access to the only source of memory available at boot time since configuring and training the **DRAM** is a much more involved process only achievable further down the boot chain. The ARC path consists, fundamentally, of a base and a limit physical memory addresses which can be configured by writing their values to dedicated MC's MMIO registers. This is normally done at a certain point during boot using values **0x40000000** (start of IRAM) and **0x4003F000** (end of IRAM minus 0x1000 bytes), but when analysing the new bootloader code in firmware version 6.2.0, we noticed the ARC path being reconfigured right before the TSEC takes over. Even more bizarre, the base and a limit memory addresses were being set to **0** and **0x80000000** (start of DRAM). Until then we had no idea this was possible, but it turns out the ARC path can not only redirect accesses to IRAM but to *all* physical memory available, which includes, as expected, every device's MMIO register space.

So now we know the *how*, but we still need to figure out the *why*. At this point we began joking around the idea of trapping the TSEC inside the **SMMU**, which is Tegra's IOMMU to which all and any peripheral device block is subjected to if address translation is enabled and configured. From reverse engineering the "nvhost_tsec.fw" code in **nvservices**, I was fairly convinced by then that TSEC had a way to bypass the SMMU when operating in **heavy secure** mode so they must've ensured it was using it, right? Well…

As we began to realize this could actually be reasonable, **naehrwert** rushed to adapt some SMMU configuration code *et voilà*, we could now sandbox the TSEC by falsely mapping all the physical memory address ranges it needed to random virtual memory addresses we controlled. We did have to overcome one simple obstacle: even though the SMMU can be fully configured through MC's MMIO registers, one of those registers requires a **secure write** which is tied to code running on the **CCPLEX** with **TrustZone** access level. Expectedly, that register is what effectively kickstarts the SMMU and since its primary purpose is controlling memory accesses requested by peripheral devices connected to the AHB bus, it makes perfect sense that only the CCPLEX would be allowed to fire it up. We quickly got the BPMP starting the CCPLEX with a tiny payload just so it would set the register we needed and await further instruction.

Once we had everything in place, it became a simple matter of copying the values of MMIO registers the TSEC would check for into the virtual mappings and adjusting them, as necessary. Eventually, we managed to coerce the TSEC into decrypting the bootloader for us and extracting all the keys it generated. This concept was further fine-tuned and implemented in Atmosphère.

```
void smmu_emulate_tsec(void *tsec_keys, const void *package1, size_t package1_size, void
*package1_dec) {
    volatile tegra_tsec_t *tsec = tsec_get_regs();

    /* Backup IRAM to DRAM. */
    memcpy((void *)SMMU_IRAM_BACKUP_ADDR, (void *)0x40010000, 0x30000);

    /* Copy package1 into IRAM. */
    memcpy((void *)0x40010000, package1, package1_size);

    /* Setup TSEC's address space. */
    uint32_t *pdir = smmu_setup_tsec_as(1);

    /* Allocate pages for MMIO and IRAM. */
    volatile uint32_t *car_page = smmu_alloc_page(1);
    volatile uint32_t *fuse_page = smmu_alloc_page(1);
```

```
    volatile uint32_t *pmc_page = smmu_alloc_page(1);
    volatile uint32_t *flow_page = smmu_alloc_page(1);
    volatile uint32_t *se_page = smmu_alloc_page(1);
    volatile uint32_t *mc_page = smmu_alloc_page(1);
    volatile uint32_t *iram_pages = smmu_alloc_page(48);
    volatile uint32_t *expv_page = smmu_alloc_page(1);

    /* Map all necessary pages. */
    smmu_map(pdir, 0x60006000, (uint32_t)car_page, 1, _READABLE | _WRITABLE | _NONSECURE);
    smmu_map(pdir, 0x7000F000, (uint32_t)fuse_page, 1, _READABLE | _NONSECURE);
    smmu_map(pdir, 0x7000E000, (uint32_t)pmc_page, 1, _READABLE | _NONSECURE);
    smmu_map(pdir, 0x60007000, (uint32_t)flow_page, 1, _WRITABLE | _NONSECURE);
    smmu_map(pdir, 0x70012000, (uint32_t)se_page, 1, _READABLE | _WRITABLE | _NONSECURE);
    smmu_map(pdir, 0x70019000, (uint32_t)mc_page, 1, _READABLE | _NONSECURE);
    smmu_map(pdir, 0x40010000, (uint32_t)iram_pages, 48, _READABLE | _WRITABLE |
_NONSECURE);
    smmu_map(pdir, 0x6000F000, (uint32_t)expv_page, 1, _READABLE | _WRITABLE | _NONSECURE);

    /* Enable the SMMU. */
    smmu_enable();

    /* Loop retrying TSEC firmware execution, in case we lose the SE keydata race. */
    uint32_t key_buf[0x20/4] = {0};
    unsigned int retries = 0;
    while (true) {
        if (retries++ > TSEC_KEYGEN_MAX_RETRIES) {
            fatal_error("[SMMU] TSEC key generation race was lost too many times!");
        }

        /* Load the TSEC firmware from IRAM. */
        if (tsec_load_fw((void *)(0x40010000 + 0xE00), 0x2900) < 0) {
            fatal_error("[SMMU]: Failed to load TSEC firmware!\n");
        }

        /* Disable the aperture since it has precedence over the SMMU. */
        mc_disable_ahb_redirect();

        /* Clear all pages. */
        memset((void *)car_page, 0, SMMU_PAGE_SIZE);
        memset((void *)fuse_page, 0, SMMU_PAGE_SIZE);
        memset((void *)pmc_page, 0, SMMU_PAGE_SIZE);
        memset((void *)flow_page, 0, SMMU_PAGE_SIZE);
        memset((void *)se_page, 0, SMMU_PAGE_SIZE);
        memset((void *)mc_page, 0, SMMU_PAGE_SIZE);
        memset((void *)iram_pages, 0, 48 * SMMU_PAGE_SIZE);
        memset((void *)expv_page, 0, SMMU_PAGE_SIZE);

        /* Copy CAR, MC and FUSE. */
        safe_memcpy((void *)car_page, (void *)0x60006000, 0x1000);
        safe_memcpy((void *)mc_page, (void *)0x70019000, 0x1000);
        safe_memcpy((void *)&fuse_page[0x800/4], (void *)0x7000F800, 0x400);

        /* Copy IRAM. */
        memcpy((void *)iram_pages, (void *)0x40010000, 0x30000);

        /* TSEC wants CLK_RST_CONTROLLER_CLK_SOURCE_TSEC_0 to be equal to 2. */
        car_page[0x1F4/4] = 2;

        /* TSEC wants the aperture fully open. */
        mc_page[0x65C/4] = 0;
        mc_page[0x660/4] = 0x80000000;

        /* Run the TSEC firmware. */
        tsec_run_fw();

        /* Extract the keys from SE. */
        volatile uint32_t *key_data = (volatile uint32_t *)((void *)se_page + 0x320);
        uint32_t old_key_data = *key_data;
        uint32_t buf_counter = 0;
        while (!(tsec->TSEC_FALCON_CPUCTL & 0x10)) {
            const uint32_t new_key_data = *key_data;
```

```
            if (new_key_data != old_key_data) {
                old_key_data = new_key_data;
                key_buf[buf_counter] = new_key_data;
                buf_counter++;
            }
        }

        /* Enable back the aperture. */
        mc_enable_ahb_redirect();

        if (buf_counter == 8) {
            break;
        }
    }

    /* Check if the TSEC firmware wrote over the exception vectors. */
    volatile uint32_t *tsec_done_check = (volatile uint32_t *)((void *)expv_page + 0x200);
    if (!(*tsec_done_check)) {
        fatal_error("[SMMU]: Failed to emulate the TSEC firmware!\n");
    }

    /* Copy back the extracted keys. */
    memcpy((void *)tsec_keys, (void *)key_buf, 0x20);

    /* Manually disable TSEC clocks. */
    tsec_disable_clkrst();

    /* Clear TSEC's address space. */
    smmu_clear_tsec_as(1);

    /* Return the decrypted package1 from emulated IRAM. */
    memcpy(package1_dec, (void *)iram_pages, package1_size);

    /* Restore IRAM from DRAM. */
    memcpy((void *)0x40010000, (void *)SMMU_IRAM_BACKUP_ADDR, 0x30000);
}
```

However, the glory was short-lived as a new firmware update, version 7.0.0, was released a couple months later. This update fixed the issue by actually using the TSEC's memory access **backdoor** I had previously theorized about: the TSEC is able to configure the **TFBIF** for bypassing the SMMU altogether by writing to 2 MMIO registers (**TSEC_TFBIF_TRANSCFG** and **TSEC_TFBIF_REGIONCFG**) which can only be accessed when running in **heavy secure** mode. Additionally, more and more complex verification checks were introduced to the microcode which rendered any kind of emulation tricks useless.

We can't "cheat" anymore. It's time to break the TSEC once and for all…

Calls, Stacks and Crypto

Looking for a way to defeat the TSEC led us to split up and explore different paths so, at the time, **SciresM** began working with **motezazer** while I joined **plutoo** and **shuffle2** in hunting for vulnerabilities. With the help of **mwk** and **karolherbst** from nouveau/envytools, we acquired further knowledge about Falcon and its intricacies, which led us to uncover a rather amusing design flaw.

As mentioned before, entering **heavy secure** mode requires **calling** the **secure microcode** after the necessary steps for authentication have been completed. Falcon's calling convention revolves around passing function arguments in general purpose registers **$r10** to **$r14**, with any additional arguments being passed through the stack, and return values being retrieved from general purpose register **$r10**. Opcodes **call** and **lcall** are used to invoke a function and

Falcon does so by pushing the return address onto the stack and **branching** to the target function's address. Note that I have emphasized "branching"; that is because in Falcon's ISA a **call**, a **branch** and a **jump** are all pretty much the same thing when it comes to invoking **secure microcode**. As it turns out, the verification of where a secure microcode's block starts is done continuously by a TLB-like system called **IMCTL**, which compares the next **PC** register's value with a list of addresses marked as belonging to secure microcode. This means that even **NOP sledding** is a viable way of reaching secure microcode and triggering the authentication mechanism.

However, contrary to calls, branches and jumps don't store the return address in the stack. So, what would happen if a particular piece of secure microcode assumes it has been "called" instead of "branched" to? The **ret** opcode or one of its variants can be usually found at the epilogue of any microcode function and does the exact opposite of the **call** opcode: it pops a return address from the stack and jumps to it. If a function assumes it has been "called", it will try to return using the **ret** opcode, but if in reality it has been "branched" into, there will be no return address to pop from the stack, or rather it will pop whatever value is pointed to by the current **SP** register's value. At a first glance, this is more of a nuisance than a critical design flaw as it implies programmers must predict this scenario and avoid stack memory management to be off by 4 bytes. Furthermore, if we assume the Falcon's ISA is accompanied by dedicated, proprietary tools such as compilers, assemblers and so on, it becomes rather obvious that somewhere along the line people will forget to handle this issue properly.

Let us go back to the **HOVI firmware** in its original, pre-6.2.0 version. By now, an astute reader might've noticed the **secure stage** is vulnerable to a stack smashing attack: by leveraging the **maconstack** bug we can craft our own **non-secure stage** and lie about its size (since we fully control the chunk of data it comes from) so, when the secure stage copies the non-secure stage to DMEM for calculating it's **AES-CMAC** hash, it will overflow the current stack buffer with whatever we want. This flaw allows us to build and run a ROP chain from within the HOVI firmware's secure stage and thus compromise the Falcon's **heavy secure** mode for the first time!

```
void load_keygen(void *key_buf, int key_version, bool is_blob_dec) {
    u32 res = 0;

    u32 dmem_start = 0;
    u32 blob0_addr = 0;
    u32 blob0_size = *(u32 *)(key_buf + 0x70);      // BUG: arbitrary size

    // Load blob0 code to the start of the data segment
    memcpy_i2d(dmem_start, blob0_addr, blob0_size);

    // Generate "CODE_SIG_01" key into c4 crypto register
    gen_usr_key(0, 0);

    // Encrypt buffer with c4
    u8 sig_key[0x10];
    enc_buf(sig_key, blob0_size);

    u32 src_addr = dmem_start;
    u32 src_size = blob0_size;
    u32 iv_addr = sig_key;
    u32 dst_addr = sig_key;
    u32 mode = 0x02;   // AES-CMAC
    u32 use_imem = 0;

    // Do AES-CMAC over blob0 code
    do_crypto(src_addr, src_size, iv_addr, dst_addr, mode, use_imem);

    // Compare the resulting hash with the one from the key buffer
```

```
    if (memcmp(dst_addr, key_buf + 0x10, 0x10))
    {
        res = 0xDEADBEEF;
        return res;
    }

    u32 blob1_size = *(u32 *)(key_buf + 0x74);

    // Decrypt Keygen blob if needed
    if (!is_blob_dec)
    {
        // Read Stage2's size from key buffer
        u32 blob2_size = *(u32 *)(key_buf + 0x78);

        // Check stack bounds
        if ($sp > blob2_size)
        {
            u32 blob2_virt_addr = blob0_size + blob1_size;
            u32 blob2_phys_addr = blob2_virt_addr + 0x100;

            // Read the encrypted Keygen blob
            memcpy_i2d(dmem_start, blob2_phys_addr, blob2_size);

            // Generate "CODE_ENC_01" key into c4 crypto register
            gen_usr_key(0x01, 0x01);

            u32 src_addr = dmem_start;
            u32 src_size = blob2_size;
            u32 iv_addr = key_buf + 0x40;
            u32 dst_addr = dmem_start;
            u32 mode = 0;   // AES-128-CBC
            u32 use_imem = 0;

            // Decrypt Keygen blob with AES-128-CBC
            do_crypto(src_addr, src_size, iv_addr, dst_addr, mode, use_imem);

            // Upload decrypted Keygen into Falcon's code segment
            bool use_secret = true;
            memcpy_d2i(blob2_virt_addr, dmem_start, blob2_size, blob2_virt_addr,
use_secret);

            // Clear out the decrypted blob
            memset(dmem_start, 0, blob2_size);
        }
    }

    // The next 2 xfer instructions will be overridden
    // and target changes from DMA to crypto
    cxset(0x02);

    u32 crypto_reg_flag = 0x00060000;
    u32 blob2_hash_addr = key_buf + 0x30;

    // Transfer the Keygen auth hash to crypto register c6
    xdst(0, (blob2_hash_addr | crypto_reg_flag));

    // Wait for all data loads/stores to finish
    xdwait();

    // Save previous cauth value
    u32 cauth_old = $cauth;

    // Set auth_addr to blob2_virt_addr and auth_size to blob2_size
    $cauth = ((blob2_virt_addr >> 0x08) | (blob2_size << 0x10));

    u32 hovi_key_addr = 0;

    // Select next stage key
    if (key_version == 0x01)                // Use HOVI_EKS_01
        hovi_key_addr = key_buf + 0x50;
    else if (key_version == 0x02)           // Use HOVI_COMMON_01
        hovi_key_addr = key_buf + 0x60;
    else if (key_version == 0x03)           // Use debug key (empty)
        hovi_key_addr = key_buf + 0x00;
    else
        res = 0xD0D0D0D0
```

```
    // Jump to Keygen
    if (hovi_key_addr)
        res = exec_keygen(hovi_key_addr, key_version);

    // Clear out key data
    memset(key_buf, 0, 0x7C);

    // Restore previous cauth value
    $cauth = cauth_old;

    return res;
}
```

The same exact result comes from abusing the call vs. branch issue: using the **maconstack** bug allows us to craft a non-secure stage that **branches** to the secure stage instead of calling it. Since the secure stage assumes it has been "called", it will try to return using a variant of the **ret** opcode, which leads to the takeover of its return address and allows us to perform ROP under Falcon's **heavy secure** mode.

We found both issues roughly around the same time and other parties also found either one or both of them independently at some point.

Whichever method used, we can now do ROP in **heavy secure** mode so, where to go from here? Naturally, we were curious as to why the final **secure and encrypted stage** was even encrypted to begin with so, **plutoo** built a ROP chain that would leverage certain functions to decrypt and dump this stage whose pseudocode representation follows:

```
void gen_tsec_key(void *key_addr, int key_type) {
    // This will use TSEC DMA to look for 0x34C2E1DA in host1x scratch space
    u32 host1x_res = check_host1x_magic();

    // Failed to find magic word
    if (host1x_res != 0)
        return;

    u32 crypto_reg_flag = 0x00000000;

    // The next 0x02 xfer instructions will be overridden
    // and target changes from DMA to crypto register
    cxset(0x02);

    // Transfer the seed in key_addr to crypto register c0
    xdst(0, (key_addr | crypto_reg_flag));

    // Wait for all data loads/stores to finish
    xdwait();

    crypto_reg_flag = 0x00020000;

    if (key_type == 0x01)        // HOVI_EKS_01
    {
        // Load selected secret into crypto register c1
        csecret($c1, 0x3F);

        // Encrypt the auth signature with c1 and store in c1
        csigenc($c1, $c1);

        // Load selected secret into crypto register c2
        csecret($c2, 0x00);

        // Bind c2 register as the key for enc/dec operations
        ckeyreg($c2);

        // Encrypt the seed from key_addr and store in c2
        cenc($c2, $c0);

        // Bind c2 register as the key for enc/dec operations
        ckeyreg($c2);
```

```
        // Encrypt the auth signature with c2 and store in c2
        csigenc($c2, $c2);

        // Bind c2 register as the key for enc/dec operations
        ckeyreg($c2);

        // Encrypt c1 and store in c2
        cenc($c2, $c1);

        // The next 0x02 xfer instructions will be overridden
        // and target changes from DMA to crypto register
        cxset(0x02);

        // Transfer the resulting key from crypto register c2 to key_addr
        xdld(0, (key_addr | crypto_reg_flag));

        // Wait for all data loads/stores to finish
        xdwait();
    }
    else if (key == 0x02)        // HOVI_COMMON_01
    {
        // Load selected secret into crypto register c2
        csecret($c2, 0x00);

        // Bind c2 register as the key for enc/dec operations
        ckeyreg($c2);

        // Encrypt the seed from key_addr and store in c2
        cenc($c2, $c0);

        // Bind c2 register as the key for enc/dec operations
        ckeyreg($c2);

        // Encrypt the auth signature with c2 and store in c2
        csigenc($c2, $c2);

        // The next 0x02 xfer instructions will be overridden
        // and target changes from DMA to crypto register
        cxset(0x02);

        // Transfer the resulting key from crypto register c2 to key_addr
        xdld(0, (key_addr | crypto_reg_flag));

        // Wait for all data loads/stores to finish
        xdwait();
    }

    // Use TSEC DMA to write the key in SOR1 registers
    sor1_set_key(key_addr);

    return;
}

void keygen(void *key_addr, int key_type) {
    u32 falcon_rev = *(u32 *)TSEC_FALCON_HWCFG1 & 0x0F;

    // Falcon hardware revision must be 5
    if (falcon_rev != 0x05)
        exit();

    // Clear interrupt flags
    $flags.ie0 = 0;
    $flags.ie1 = 0;
    $flags.ie2 = 0;

    // Set the target port for memory transfers
    $xtargets = 0;

    // Generate the TSEC key
    gen_tsec_key(key_addr, key_type);

    // Clear the cauth signature
    csigclr();

    // Clear all crypto registers
    cxor($c0, $c0);
    cxor($c1, $c1);
```

```
    cxor($c2, $c2);
    cxor($c3, $c3);
    cxor($c4, $c4);
    cxor($c5, $c5);
    cxor($c6, $c6);
    cxor($c7, $c7);

    return;
}
```

To understand what's going on here, we must look once again into the **SCP**'s concept of **secrets**. As mentioned during the crash course, the SCP provides access to an array of 64, 16 byte-sized values stored in some kind of non-volatile storage, presumably, eFuses. Additionally, the SCP's crypto registers are subjected to ACLs which dictate the kind of access allowed on each register.

The SCP's **csecret** command loads the desired secret into a crypto register of your choosing, but this process also forces a predefined ACL to be set on the destination register. A table of which secret sets which ACL can be found here but, essentially, a crypto register with a secret loaded into it is always **writable** and can only have one out of three possible **secure** (as in, only allowed in **heavy secure** mode) ACL combinations: **readable** and **keyable**, **unreadable** but **keyable** or **unreadable** and **not keyable**.

If a crypto register is **readable** and **keyable**, its value can be accessed by transferring it from the SCP to Falcon's DMEM and it can be used as a key for AES operations within the SCP; if a crypto register is **unreadable** but **keyable**, transferring it will always output zeroes, but it can still be used as a key for AES operations within the SCP; if a crypto register is **unreadable** and **not keyable**, transferring it or trying to use it as a key for AES operations will always result in zeroes. This raises a question: why have secrets that not only can't be read, but also can't be used for AES? Enter the SCP's concept of **signatures**.

According to Nvidia's patent "Providing secure access to a secret", the SCP was designed with a very particular use case in mind: binding secret values to microcode signatures. For this purpose, three special SCP commands are provided: **csigcmp**, **csigclr** and **csigenc**. We will get to csigcmp later so, for now, let's focus exclusively on csigenc and csigclr. If you recall the necessary steps for loading **secure microcode**, one of them requires loading one of the SCP's crypto registers (specifically, **$c6**) with a **signature** value that can only be generated by Nvidia. During authentication, if and only if the signature is valid, its value is copied to an isolated region in the SCP's SRAM. Then, when running in **heavy secure** mode, the csigenc command can be used for encrypting (**AES-128-ECB**) the contents of any given crypto register using the current microcode's signature value as key. The resulting value is lifted from any ACL restrictions and can, thus, be read or used as a key for any other AES operations. Expectedly, the csigclr command clears the signature value from SRAM **permanently**, which means the csigenc command can no longer be used in the current microcode's context.

This is a very clever way to protect the integrity of both signed microcode and secret values that shouldn't be directly accessible and that is exactly what we observe in the HOVI firmware's **secure and encrypted stage**: two keys (one for testing purposes, other for production usage) are generated using seed values and the SCP's secrets 0x00 and 0x3F which are further combined with the currently active signature value.

Very interesting, but nothing particularly ground-breaking. Nonetheless, we still wanted to exploit this stage and, hopefully, achieve ROP under its context. Why? Well, if you take a look at the previous **secure stage**, you'll notice it clears the contents of all SCP's crypto registers (except **$c6**, because it contains the current signature value) as soon as possible and, regardless of which method used to achieve ROP, we always hijack execution after that. By now, we already suspected the secure microcode authentication process had to use the SCP's crypto registers somehow; could it be they forgot to clear them and that's why this task ends up falling on the microcode?

For some reason, the **secure and encrypted stage** completely forgets to clear the crypto registers' contents on entry and only does it towards the end, before returning. Could we somehow hijack code execution in between? Yes, we can!

Looking at how the **secure and encrypted stage** generates keys, **shuffle2** had a great idea: since we control the memory address where the key's seed is read from, which happens to be also where the final generated key is written to, and we also control which type of key we want to generate (the production key is per-console, but the testing key is not), if we know the value of SCP's secret 0x00, we can get a plaintext/ciphertext pair. We control all of the registers' values that are pushed onto the stack as well as the key's seed address and the key type because these are passed as arguments to the **secure and encrypted stage**. All we need is to pass the stack's address of the 16 byte-sized region whose last 4 bytes overlap with the current return address and brute-force a value that, once encrypted, will turn the return address into whichever value we want! Luckily for us, SCP's secret 0x00 is **readable**, but how can we spill its contents back into Falcon's DMEM? Enter the **nouveau** team's expertise. Thanks to **mwk**'s years of work on probing and documenting secretful Falcon-based units, most of the SCP's MMIO register space had already been reversed engineered by the time we began poking at it.

As mentioned during the crash course, the SCP provides MMIO registers for **debugging** and **controlling** some of its features. One of these registers, which we call TSEC_SCP_CMD, encodes information about the last SCP's command that had been processed. However, it turns out this register is also **writeable**! We later figured out its true purpose is to provide a debugging window, allowing clients to read and write SCP's commands via MMIO. Knowing this, we went back to the **secure stage** and located a gadget for calling the **iowr** opcode, which tells the Falcon to issue a MMIO write request to the **CSB** (which then routes it to the appropriate MMIO register space within the Falcon-based unit's hardware). By carefully assembling a ROP chain around this gadget, we could now write to arbitrary MMIO registers from within the Falcon while running in **heavy secure** mode! This allowed us to issue the **csecret** command to the SCP using the **TSEC_SCP_CMD** register and then dumping the loaded secret's contents to DMEM, for extraction. Expectedly, only **readable** secrets would show a non-zero value, but that's good enough for now.

With the value of secret 0x00 in hand, we can finally take over the **secure and encrypted stage**. We found a good enough plaintext/ciphertext pair which allowed us to jump over those last **csigclr** and **cxor** commands so, now we can abuse the main function's epilogue to build a ROP chain inside this stage as well. Naturally, the very first thing we tried was spilling out the contents of each SCP's crypto register and this is where the real fun begins! Registers **$c0**, **$c1**, **$c2** and **$c7** were empty, register **$c6** still had the current microcode's signature value loaded in, but registers **$c3**, **$c4** and **$c5** had been populated somewhere in between transitioning from the **secure stage** to the **secure and encrypted stage**! Our suspicions were

right: whatever code was responsible for the authentication process forgets to clear at least some of the SCP's crypto registers.

Ironically, this also led to the realization that our method for previously extracting the value of secret 0x00 was unnecessarily complicated… You see, the reason why the SCP's crypto registers **$c0** to **$c2** were empty is because they are never used during the authentication process and this opens up a new, really simple way of extracting **readable** secrets:

1. While still in **non-secure** mode, use the SCP's **csecret** command to load the secret into one of the first 3 SCP's crypto registers (**$c0**, **$c1** or **$c2**). This is perfectly valid and allowed by the cryptosystem as the crypto register with the loaded secret now has an ACL value that prevents its contents from being read or used as a key while we are **not** in **secure** mode.
2. Enter **secure** mode and spill the contents of the previously loaded SCP's crypto register to DMEM.

And… that's it. This highlights something we, at the time, didn't quite understand about the cryptosystem: aside from arithmetic and signature related commands, SCP's crypto operations can still be used in **non-secure** mode, it's just that we will never get to see actual values at any point due to ACLs being set and propagated through crypto registers. However, the operations *are* being executed nonetheless and once we enter **secure** mode, we get to see their calculated values. This was something that **SciresM** and **motezazer** astutely observed way before we did and put into practice during their parallel efforts.

Their method for extracting **readable** secrets also relied on abusing the **HOVI firmware**'s **secure and encrypted stage** and consisted of the following steps:

1. Observe we control the key's seed/destination address and the key type as these are passed as arguments to the **secure and encrypted stage**.
2. Observe the **cxset** Falcon opcode changes the behaviour of **xdst**, **xdld** and **xdwait** opcodes: memory transfers will now occur between **Falcon** and the **SCP** instead of between **Falcon** and the **TFBIF**. When transferring memory to/from the SCP, the destination/target crypto register's index is encoded in the high 16 bits of the **xdst**/**xdld** opcode's second operand which, in the **secure and encrypted stage**'s case, happens to be the key's seed/destination address we control.
3. Select the key type that only depends on secret 0x00 and craft a key's seed/destination address where the high 16 bits encode the index of an unused SCP's crypto register such as **$c7**, for example.
4. While in **non-secure** mode, load whatever value you want into one of the SCP's crypto registers unused by the authentication process such as **$c0**, for example. Load secret 0x00 into the **$c2** register, for example, and use the SCP's AES commands to **decrypt** the contents of **$c0** using the value of **$c2** as key. Store the result in **$c0**.
5. Launch the **secure and encrypted stage**. The key's seed will be read into **$c7** instead of **$c0**, which will keep the value of your choosing because it goes unused by the authentication process. Since this stage will attempt to **encrypt** the value of **$c0** with secret 0x00, it will effectively **reverse** what was done before in **non-secure** mode allowing **$c0** to retain whatever value you chose. Finally, this stage will use the resulting value to **encrypt** the current microcode **signature**'s value and store the result back into the key's seed/destination address.

6. Observe we now have a way to offline brute-force an appropriate value for taking over the **secure and encrypted stage**'s stack without ever needing to know the value of secret 0x00.
7. With the necessary value in hand, craft a ROP chain for spilling any SCP's crypto register's contents into DMEM. Since **$c1** also survives the authentication process and is never modified by the **secure and encrypted stage** when the test key type is chosen, we can now load any secret into **$c1** in **non-secure** mode and spill its contents into DMEM via ROP.

One way or the other, we now know exactly which registers are used during the authentication process, so… what now? Can we somehow use this to figure out the algorithm behind secure microcode's authentication? Unsurprisingly, the answer is yes.

From their research on the very first version of Falcon-based hardware (also known as **version 0**), **mwk** had been able to reconstruct the algorithm behind the authentication process. Unfortunately, the algorithm had changed at some point and for the TSEC, being a **version 5.1** Falcon-based unit, none of the algorithm's concepts would apply. Nonetheless, through **mwk**, we came to learn about more useful SCP's MMIO registers: TSEC_SCP_DBG0, TSEC_SCP_DBG1 and TSEC_SCP_DBG2. These 3 registers are used together for debugging the SCP's **LOAD**, **STORE** and **SEQ** blocks and, if you recall the crash course, the SCP's **SEQ** block provides an interface for recording sequences of commands as **macros** and later executing them when deemed fit. Arguably, the most interesting use for these 3 registers is the ability to "playback" the last recorded and/or executed **SEQ** macro which raises an interesting question: can we see if and how the authentication process uses **SEQ** macros? To answer this, we first had to improve our ROP chain so we could write arbitrary MMIO registers. Luckily, the **secure and encrypted stage** has the perfect gadget for the task, which also happens to be the *only* place in the whole microcode where the **iowr** opcode is used!

```
00000b01: f9 80              C  push $r8
00000b03: fe 48 01              mov $r8 $sp
00000b06: f4 30 f8              add $sp -0x8
00000b09: 92 89 04              sub b32 $r9 $r8 0x4
00000b0c: a0 9a                 st b32 D[$r9] $r10
00000b0e: 92 89 08              sub b32 $r9 $r8 0x8
00000b11: a0 9b                 st b32 D[$r9] $r11
00000b13: 92 89 04              sub b32 $r9 $r8 0x4
00000b16: bf 9f                 ld b32 $r15 D[$r9]
00000b18: 92 89 08              sub b32 $r9 $r8 0x8
00000b1b: bf 99                 ld b32 $r9 D[$r9]
00000b1d: fa f9 00              iowr I[$r15] $r9
00000b20: fe 84 00              mov $sp $r8
00000b23: fc 80                 pop $r8
00000b25: f8 00                 ret
```

Now *that's* lucky! We expanded our ROP chain and began poking at these debug registers in hopes of uncovering something that would allow us to understand what's going on during authentication. As previously mentioned during the crash course, authentication takes place in hidden **secure bootrom**, but we didn't know this for sure at the time. What we did know was that one of the Falcon's **SPRs** had to be configured prior to initiating the authentication process. The **envytools** project calls this SPR **cauth** and, even though we later found out its official name to be **SEC**, it kind of became common sense to refer to the theorized authentication algorithm as the "cauth algorithm".

Our new task was clear: investigate the cauth algorithm as much as possible. Through our newly enhanced ROP chain, we began poking at SCP's MMIO registers with **heavy secure**

mode privileges and were able to confirm the algorithm does indeed use **SEQ** macros so, we replayed the last recorded macro and got ourselves one piece of the puzzle!

Meanwhile, parallel to our efforts, **SciresM** and **motezazer** were about to make a handful of valuable discoveries…

Investigating CAUTH

Independently of **hexkyz**/**plutoo**/**shuffle2**, **motezazer** and I (**SciresM**) had just gone through a roughly similar but different process together to get ROP under HS mode.

With the ability to execute ROP in HS mode, we set our sights on the next obvious target: we wanted to somehow sign our own payloads, so that we could investigate the system more thoroughly.

The obvious thing to do is to investigate the "cauth" algorithm for flaws, to try to find a way to break it.

We begin investigating cauth by outputting all registers values upon entry to HS mode to find:

- $c0 is not touched by cauth.
- $c1 is not touched by cauth.
- $c2 is unknown/we can't dump it yet due to it being overwritten before we get ROP.
- $c3 is secret/touched by cauth.
- $c4 is secret/touched by cauth.
- $c5 is readable in **heavy secure** mode but not in **non-secure** mode and contains an unknown value.
- $c6 is readable and contains the code signature we put in it.
- $c7 is readable and contains zero, which we put in it.

This analysis is pretty interesting, but we can do a little better.

We can observe that given a secret register **$cx**, we can execute $csigenc $cy $cx to obtain non-secret $cy = aes_enc($cx, current_sig).

We can further observe that if two secret registers **$cx**, **$cz** contain the same value, they will produce the same result via **$csigenc**. In other words, if $cx == $cz, then csigenc($cx) == csigenc($cz).

This allows us to test if two secret registers are the same and to compare them to fixed values, even when we can’t read their contents!

We then repeat our tests, comparing all the secret values to each other/various permutations of each other.

We find:

- $c4 == $c6.
    - Thus $c4 is the "computed" signature.

- $c4 = aes_enc($c3, $c5).
    - Thus $c5 is presumably calculated from the code pages, somehow, and $c3 is a secret key.
- $c3 = aes_enc(csecret 0x1, $c7).
    - This explains why changing $c7 makes cauth fail...it changes the key being used!

This is very interesting! We learned that the cauth procedure looks something like:

- $c3 = aes_enc(csecret 0x1, $c7)
- $c5 = <calculate MAC, somehow>
- $c4 = aes_enc($c3, $c5)
- verify($c4 == $c6).

Thus we conclude that the security of cauth depends on the inability of the attacker to calculate either of **$c3** (it is secret) and **$c5** (it uses an unknown algorithm).

However, we notice immediately a big **cryptographic flaw**! One of the fundamental assumptions this procedure makes is that you can never know the value of aes_enc(csecret 0x1, $c7).

However, consider the case where we choose $c7 = <signature of a payload we have ROP under>.

We can then use ROP to obtain the value $csigenc(csecret 0x1) == aes_enc(csecret 0x1, <signature>).

This means that if we set $c7 = <signature> before entering cauth, we know the value of $c3!

Thus, the remaining security depends on the unknown algorithm to calculate $c5.

Here, I (SciresM) sat and thought about the constraints on the algorithm for a bit, and came up with the following:

- The algorithm must take as input the data inside pages, and possibly where they're loaded.
- To save on ROM space, the algorithm is almost certainly a "macro", which means it operates on 16-byte blocks.
- If data is processed block-by-block, it can only really use three operations: encrypt, move, and xor, in some combination.
- The algorithm is a black box, but we can still try to minimize the randomness of the input.

Given the operation looked so constrained, it seemed worthwhile to try to recreate a simple MAC by hand.

Thus, we dumped the calculated $c5 of a single 0x100 page containing all zeroes loaded at address 0x0 = A3C5F7283D30973DBDAA5AF5974DF9A2.

After around an hour of trying various combinations of encrypt/xor in python, I lucked out, and the following produced a matching value:

```
key = '\x00' * 16, cur = '\x00' * 16, prev = '\x00' * 16
for i in xrange(0x11):
    enc = encrypt(key, cur)
    cur ^= enc
```

This was confusing, since it seemed to imply that the key was all zeroes, but it was a major piece of progress.

Next the calculated value of $c5 was dumped for a single 0x11 page containing all 0xFFs loaded at address 0x0 = A17F68121B1F40E93B3221AA0BDBAB6F.

I was quickly able to find a match:

```
cur = '\x00' * 0x10
for i in xrange(0x10):
    enc = encrypt('\xFF'*0x10, cur)
    cur = cxor(enc, cur)

enc = encrypt('\x00'*0x10, cur)
cur = cxor(enc, cur)
```

Thus, it seems the algorithm takes each 16-byte block in the loaded page, encrypts the current MAC with it, then xors the result into the current MAC.

The last block is 0x00s, and intuitively either this is the address of the page, or the starting value is the address of the page.

We quickly determine that the last block is indeed the address (as little endian), leaving the cauth algorithm with zero unknowns.

With the algorithm in hand (and a known $c3-$c7 pair), we successfully sign our test payload, obtaining **arbitrary secure mode code execution**!

We weren't quite done, though -- we were also interested in investigating how the "hardware encryption" works for cauth, since we were interested in obtaining the plaintext of secure binaries.

This turned out to be "easy", though...we notice that the last blocks of a secret payload are identical with value 1DE36458FA9EC298D5B45774B582E711.

This implied that **AES-ECB** is being used, and also the last blocks were probably "unused"/all-zero.

We did the "obvious" thing after noticing this: we used our newfound secure execution to dump the value of aes_enc($csecret X, zeroes) for all csecrets that we can encrypt data with.

It turns out that aes_enc($csecret 6, zeroes) == 1DE36458FA9EC298D5B45774B582E711, and **the encryption is just aes_enc($csecret 0x6, <data>)**.

We then noticed something else interesting...when we calculated the expected value of $c5 for an encrypted payload, we got the correct result when calculating the *encrypted* data.

This implied that decryption is performed *after* the payload is validated...and sure enough, it is.

This was really interesting, because it implied that we can intentionally put the "wrong" value of the encryption bit in the relevant register, and TSEC will still consider the signature valid and attempt to jump to either encrypted-code=garbage, or decrypted-plaintext=garbage.

If you're familiar with past exploits for the 3DS, the idea of jumping to garbage should be a familiar one to you. If we're lucky, the garbage will randomly contain a "jump" instruction. If it does, we can set up a secure/authenticated page there ahead of time, and the processor will begin executing the code we've set up in advance while keeping the target payload's signature loaded.

This does indeed work, and it allows arbitrary execution under a number of interesting payloads! Notably, we used this to gain execution with Nintendo's 7.0.0 SecureBoot payload, and thus dumped the keys it generates without having to find any flaws in the payload itself.

As a fun historical side-note, this trick was discovered less than 24 hours before Nintendo released 7.0.0, so we discovered it just in time. :)

With all that said, we were in a really good position at this point. We could sign and execute arbitrary code in secure mode, and we could execute code with the signature loaded for some fraction of payloads that corresponded to good garbage. But we were still worried -- there were ways to mitigate our attacks, and we didn't want to find ourselves unable to derive new cryptographic key material in a future system update that addressed our attacks. After attacking cauth cryptographically, the only thing left to really turn our attention to was the actual code implementing the algorithm, so we did…

Poking at Secure Boot ROM

At this point, everyone else who had been involved in TSEC work had stopped working on the project since we'd achieved pretty much all our goals. **Hexkyz**/I (**SciresM**) had been working independently up until this point, but we decided that if we wanted to compromise TSEC's **Secure Boot ROM** (hereafter "ROM"), trying to do it alone would be a fool's errand. Thus, we decided to start sharing all knowledge and to start working together.

From our work on cauth, we felt like we had a pretty good idea of what kinds of things ROM was doing -- that said, there aren't any clear avenues to begin messing with ROM. It's execute-only, and it's pretty much a "black box" where it executes and then either your payload begins executing in HS mode or the Falcon CPU takes an exception (and usually halts).

That said, there was a fairly big question remaining about ROM... how exactly was it managing the data it was using? Our understanding of ROM suggested it would need to use some kind of data memory, storing return addresses to stack (to perform function calls) and reading data from memory into crypto registers in order to perform its cryptographic algorithms.

The only obvious data memory available was DMEM... and sure enough, filling DMEM with a fixed value and then performing the cauth procedure allowed us to observe changes to DMEM after ROM finished executing!

In particular, we found:

- ROM was using DMEM below the value of **$sp** on entry was as stack space.
- ROM doesn't clean up DMEM after it's done, and we can see some stack residue, including all of the general purpose registers (presumably from saving all registers to stack) and what looked like some return addresses(!).

This was **extremely** interesting! Not only was ROM using memory we had access to as stack, but its usage of **$sp** meant that **we could control what part of DMEM ROM uses as stack**!

This isn't immediately exploitable on its own, but it was a sign of sloppiness and a minor mistake -- we might expect a more careful implementation might use a fixed address as stack, as a way of limiting user-control over ROM's behaviour.

Seeing a sign that ROM wasn't written flawlessly was extremely encouraging at the time, motivating even closer looks. Eventually, hexkyz hit upon a brilliant idea: if we know/control what memory ROM is using as stack, what would happen if we tried to mess with that memory using DMA?

Performing DMA on falcon is a pretty particular process; there are a few steps involved:

- Use a special instruction ("**cxset**") to configure the transfer, setting the source/destination type etc.
- Initiate the transfer using either of two special instructions ("**xdld**"/"**xdst**") depending on the direction of the transfer.
- Because **DMA transfers are completely asynchronous**, wait for all pending transfers to complete using a final special instruction ("**xdwait**").

The really important thing to observe is that DMA is **asynchronous**. This makes a lot of sense from an optimization point of view -- you might want to start a DMA transfer, and then be productive by doing some work while it's happening instead of just waiting for it to finish.

That said, it creates a potential problem from a security perspective. If you have to execute an instruction to guarantee no DMA transfers are ongoing, this means any code which doesn't know when the last **xdwait** was executed can't guarantee that there isn't a DMA transaction in flight, and thus can't trust data it's using. So, it's a requirement to know when the last xdwait was executed to safely interact with DMEM. You can probably see where this is going…

If ROM "forgets" to execute an xdwait on entry, DMA could be scheduled and potentially occur asynchronously in the middle of ROM!

Sure enough, we tried some simple tests, scheduling a bunch of DMA transfers to read and write to stack and then jumping to ROM:

- DMA transfers to copy stack elsewhere worked sometimes, resulting in us getting more and different stack traces in the middle of ROM.
- DMA transfers to copy other data memory to stack sometimes cause ROM to fail to complete. This suggests we're overwriting stack/return addresses, and thus **have ROP under ROM**!

This was an amazing discovery -- with ROP under ROM, we theoretically have everything we need to exploit ROM and ensure our win permanently.

Unfortunately, "theoretically" is a very, very key word.

ROM is **execute-only**. This means we don't have (and can't get) a copy of its code.

How do you exploit a binary you **don't have the code for**?

The Exploit - Je Ne Sais Quoi

There are a few questions I (**SciresM**) find particularly useful to ask, when confronted with a difficult problem. **What do I have? What do I want? How can I use what I have to get what I want?** Asking these explicitly often leads to noticing things I might miss otherwise, and I consider this kind of review essential to the development process. So, with that in mind...

What do we have?

- We have the ability to control **$c6** and **$c7** on ROM entry, which are the signature and keyseed respectively.
- We have the ability to control what **address** and **size** ROM is validating, and to control the contents of the **pages** ROM is validating.
- We have the ability to control whether ROM **decrypts** the pages it's validating using **secret 0x6**.
- We have the ability to control **$sp** on ROM entry, which lets us control what region of memory ROM uses for **stack**.
- We have the ability to spam **asynchronous DMA transactions** before ROM entry. This gives us two capabilities:
    1. We can obtain a dump of stack at a "random" point in ROM execution that we don't control.
    2. We can overwrite stack with **arbitrary**, **controlled data** at a "random" point in ROM execution.
- We can set an **exception vector** up, so that if ROM fails to validate the secret pages/signature we can perform an "autopsy" (stack/register inspection) to see what might have gone wrong.
- We can poke at all of the exposed TSEC **registers**, and see if anything interesting happens to ROM when we do so.

What do we want?

We fundamentally want to **break** the security model in some way that can't be defended against or recovered from.

The TSEC cryptosystem relies on the security of the "csigenc" mechanism, which encrypts the current signature with a value. If we can control the currently loaded signature, this would completely break the security model. So, one idea for a "concrete goal" is to make ROM complete successfully, but make it use an **arbitrary value** we control as the **signature** instead of the correct one. This seems like a plausible, achievable goal.

Another strategy would be to somehow use ROM's **special permissions** to obtain the value of **secrets**. If we can somehow find a way to get ROM to output secret values (maybe ROM can bypass secrets or something?), this would also break the cryptosystem. This seems less plausible, but within the realm of theoretical possibility.

A third strategy would be to somehow execute our own **custom code** instead of ROM, while still retaining ROM's special permissions (somehow?). Seems very unlikely and probably not possible, but a guy can dream.

How can we use what we have to get what we want?

Instrumentally, all of our goals boil down to "execute a ROP chain that does the thing we want".

If we want to write a ROP chain, we need to know things about ROM -- what return addresses are we going to overwrite and what happens when we "return" to various addresses. We need some kind of **information leak**, and we can set a first goal of leveraging our "dump stack", "autopsy a signature validation failure", and "poke at registers blindly" primitives to gain some more information. We can also try overwriting various **return addresses** with various values, and see what changes in the final stack dump when we do that.

Once we know enough things about ROM and have our gadgets, we'll put the puzzle pieces together and write a chain that accomplishes one of our goals. This feels a little "draw the rest of the owl", but we can cross that bridge when we better have our bearings.

The above discussion might seem really abstract and useless to you, but I promise that the most important step to tackling a **seemingly intractable problem** is to try to **divide it into smaller parts** that you know how to work on. Doing this exercise was really valuable at the time, and gave us some clear trajectories.

I (**SciresM**), started focusing on trying to use our dump stack/autopsy primitives. Meanwhile, **hexkyz** started trying to look at the available registers to see if there might be anything interesting to try poking at.

I wrote a helper program that repeatedly executed a TSEC payload which spammed DMA transfers (0x4000 of them, because "0x4000 is a big number") and then jumped to ROM, waited a bit, and dumped stack. The dumped stack contents were then compared to the ones we've dumped in past loop iterations, and if they're different then they were saved. We don't have control over when the DMA transfer that copies out stack mid-execution does its magic,

but by trying over and over and logging all unique dumps we could create a bunch of snapshots, and piece together what ROM was doing.

Short summary of what I observed from stack dumps:

1. ROM stores all general purpose registers to stack.
2. ROM subtracts **0x44** from **$sp**, and zeroes out that region.
3. ROM pushes a return address (**0xA4**) on stack.
4. ROM pushes another return address (**0x24A**) on stack.
5. ROM subtracts **0x10** from **$sp**, aligns down $sp to 0x10, and then clears 0x10 bytes.
6. I didn't get anything else meaningful at this point, other than maybe another return address (0x2D7).

Now that I knew where two return addresses were on stack, the only obvious path forward was to try overwriting them and seeing what happens. This was mostly fruitless, but there was one interesting discovery made: **ROM is mirrored**! If you manage to return to an address after ~0x370, ROM executes successfully, but the stack trace you get when dumping from HS mode has all of the return address ORed with 0x8000.

Some more testing confirmed that ROM is indeed present at both **0x0** and **0x8000**, which seemed potentially interesting but not clearly useful.

I also observed that it was possible to cause ROM to report an **authentication failure** despite "correct" signature loaded, which confirmed we could interfere in the process, but I didn't really understand what I was doing/what we were overwriting at this point.

Science done, I regrouped with hexkyz to discuss and theorize. Fortunately, he'd had **substantially** better luck than I'd had.

For starters, thanks to **mwk**, hexkyz knew how to debug **SEQ** macros. As we know, it is possible, briefly, to observe ROM's cryptographic macros as they're executing so, by replaying them we found that **authentication** looks something like this, at some unknown address:

```
...
// This runs in a loop for each 0x100 bytes page.
cs0begin 0x03
cxsin $c4
cenc $c3 $c5
cxor $c5 $c3
ckeyreg $c4
cxor $c5 $c5
cs0exec 0x11
...
// Use secret 0x01 as key and $c7 as seed.
csecret $c3 1
ckeyreg $c3
cenc $c3 $c7
ckeyreg $c3
cenc $c4 $c5
csigcmp $c4 $c6
...
```

And we found that **secret page decryption** looks something like this, at some unknown address:

...

```
// Use secret 0x06 as key.
cs0begin 0x03
cxsin $c3
cdec $c4 $c3
cxsout $c4
csecret $c5 0x06
ckexp $c5 $c5
cs0exec 0x10
ckeyreg $c5
...
```

This wasn't the only gold hexkyz struck, though... he also noticed an interaction with the **TSEC_FALCON_IBRKPT#** registers. These are debugging registers that normally do what you'd expect ("set a breakpoint"), but which don't work quite right in ROM. For most addresses, breakpoints don't seem to work... but for a select few they seem to, and the Falcon immediately halts and locks down.

More importantly, though, the IBRKPT register contains a special bit (30) called **SKIP**. If this bit is set, then instead of breaking the bit will be cleared and the Falcon will continue as normal (breaking if it executes that address again). Amazingly, **this works on ROM addresses**. Hexkyz found that if we set the **SKIP** bit, it will be cleared if ROM executes an instruction at a given address.

With this knowledge in hand, I wrote a new brute-forcer payload: we would try to set a SKIP breakpoint on every ROM address, try to execute ROM, and see what happened. If the bit was cleared, that address was executed. If the processor halted, that instruction is one of the weird ones that breakpoints work on, and was executed at least twice.

By trying every address, this **gave us the location and sizes of all ROM instructions**. There was no way to know what the instructions do, but knowing their sizes constrains the possibilities heavily, since instructions are **variable-length**. This was a huge increase in our knowledge. Over the next week or so, we mapped out and theorized what parts of ROM might be doing what... notably, **we were able to use this to figure out where the cryptographic operations ROM performs are located** (since we know their instruction widths!).

The most important thing we learned was that the most important ROM instruction, the one which sets the loaded authentication signature, "**csigcmp $c4, $c6**" was located at **0xB8**.

We now had a new concrete goal: **find some way to execute that instruction with control of both $c4 and $c6**.

This was a lot of progress all at once...and then hexkyz made *another* discovery.

There are two additionally interesting debug registers called **TSEC_FALCON_TRACEIDX** and **TSEC_FALCON_TRACEPC**. In non-secure mode, TRACEPC gives you the current **PC** and TRACEIDX allows you to make it contain the N-to-last one, rather than the most recent. The maximum "depth" allowed for TRACEIDX is limited by the hardware and, in TSEC's case, this limit is set to 8 addresses. In HS mode however, this feature was meant to be entirely **disabled** but, possibly due to interrupt delivery issues within the Falcon CPU, TRACEPC activates whenever a **branch/jump/call/return** executes and ends up **recording the address of these instructions**.

There were two ways in which this was interesting: first, we could recover the last 8 branches that ROM executed after entry into our HS payload! This includes the last few branches ROM takes, giving us yet more information. Second, and even better, **it interacts with the weird breakpoint behaviour**! When we got ROM to break (on one of the few addresses where it does), we could interact with TRACEPC from the BPMP and see the last few branches.

What followed was... a lot of brute forcing. A **lot** of brute forcing, ROPing to various addresses and capturing TRACEPC values, capturing TRACEPC values at various breakpoints, capturing TRACEPC values under various success/failure conditions... much of this writeup is abbreviated already, but I would like to be clear that *this* is especially abbreviated. We spent two and a half weeks where we basically made TRACEPC brute force dumps for ~6 hours a day, chatting all the while about what we were learning and expanding our knowledge of ROM's behaviour.

I brute forced breakpoints to inspect TRACEPCs at every address. I brute forced a payload that tried to ROP from every return address I knew about to every possible address, outputting all interesting TRACEPC values. I brute forced ROP "chains", with multiple returns to every combination of addresses.

Most of the data we got was useless, but some of it was **exceptionally interesting**.

Some highlights from the two-and-a-half week brute force party:

- **The addresses where breakpoints work are all call instructions**. We think there's some issue (similar to the TRACEPC issue in HS mode) with the hardware state machine that makes breakpoints always work on calls, but we're still not sure exactly what's going on there.
- **ROM "probes" the pages it's loading by branching to them** -- this causes an immediate return to ROM, and is used as a weird way of validating that the pages are executable.
- The 0x10-aligned block that ROM touches on stack is in fact **the "trailer" block used to mix the address of the loaded page into the cauth hash.**

At this point, I had the critical eureka moment.

We want to execute the special "csigcmp $c4, $c6" instruction with control of **$c4** and **$c6**.

As we learned from **SEQ** macros, ROM uses "cxsin $c4" to load blocks in for hashing. This means that when hashing is done, **$c4** contains the **"trailer"/address block**.

The trailer block is on **stack**, which means that with some lucky DMA we can **overwrite** it. This means that we can control **$c4** when the hash function is returning.

"When the hash function is returning" -- with especially lucky DMA, we can gain ROP at this time.

Finally, the pieces came together: we schedule a bunch of spam DMA transactions to **overwrite** the part of stack where the **trailer block** (and **return address** for page probing)

are located. Our ROP chain returns to the part of the ROM that compares the **signature**, making sure that after that we're in a state where ROM can safely execute its exit routine.

At this point the search space was small. I wrote a test payload to begin brute forcing post-return-address stack contents, and pretty quickly **this worked**! We successfully entered HS mode with $c4 == $c6 == $current_signature == all zeroes.

My test payload did csecret $c0, 0x1; csigenc $c0; and I was successfully able to dump the value of aes_enc(csecret 0x1, $c7), **thereby recovering the "official" NVidia TSEC signature key**.

I shared the critical breakthrough with hexkyz, and he quickly improved/iterated on the payload, working out the missing pieces to make it stable and flexible -- this strategy ended up successfully giving us **arbitrary HS code execution with signature = arbitrary fixed value**.

This **completely breaks** the TSEC security model from a cryptographic perspective -- if we have a target payload which generates some key, we can set our signature = that payload's signature, and do the same key generation they do to recover the same keys.

This allowed us to finally dump all of the keys from 6.2.0's TSEC firmware (which was not vulnerable to the wrong crypto attack we used on 7.0.0), and when 8.1.0 released updating the TSEC SecureBoot firmware to mitigate all software vulnerabilities we knew about, we were able to use this attack to dump those new keys, too.
Because this is a (unmitigable!) hardware issue in all Falcons which have **SCP**, not just TSEC -- we were also able to use the same attack on the Falcon unit used for **GPU** power management, recovering its (different) **signing key** as well.

We still don't have the code for ROM and suspect it is impossible to dump without a microscope attack or something. What we think is most important to take away is just how **little information** you actually need to **exploit** a binary. Brute forcing return addresses can lead to more information leaks, which gives you more information, which lets you improve your brute force's search space, which [...].

Even without the tracepc/breakpoint tricks, ROM is a **solvable problem**. All you would really need to solve it is to notice the trailer block on stack is the trailer block, which presumably you do if you load your payload at any address other than 0x0, and so the block is clearly your payload's address.

Grinding away at that cycle has changed our viewpoints on --x payloads: a dedicated attacker with sufficient time and motivation will eventually be **able to brute force a ROP chain, even without access to the code to construct gadgets**.

An insane amount of work went into this exploit's development -- during February of 2019, Hexkyz/I put in some combined ~300-400 hours slowly building up a working knowledge about abstract behaviour of ROM and figuring out how to complete this puzzle. This work was the highlight of both of our years -- our heartfelt thanks to NVidia for creating such an interesting problem to tackle <3.

The End

And that's a wrap! We hope you enjoyed reading this as much as we enjoyed working on it.

As for what comes next, documentation of the TSEC and Falcon units in general is an ongoing process over at the envytools' project and our own SwitchBrew Wiki, so we suggest keeping an eye on those if you wish to learn even more.

Furthermore, a relatively recent project aims to provide tools to tinker with Falcon microprocessors: faucon. Written in Rust and created by developer **vbe0201**, this project delivers a number of extremely helpful programs for dealing with all things Falcon such as an assembler, a disassembler and even a work-in-progress emulator. We strongly suggest giving it a look.

Additionally, if you're a **Ghidra** user, we suggest checking out the ghidra_falcon plugin developed by **Thog**. While still a work-in-progress, it provides a nice alternative to the **envytools** disassembler for handling Falcon code.

Last but not least, in no particular order, we wish to thank and acknowledge the following for their invaluable contributions:

- **The nouveau/envytools project**
- **mwk**
- **karolherbst**
- **motezazer**
- **shuffle2**
- **plutoo**
- **naehrwert**
- **Nintendo**
- **NVidia**

As usual, have fun!