Attack: Readback of bitstream, extraction of BRAM contents to retrieve secrets

All designs use a Microblaze (MB) application to implement the DRM on the FPGA using the MB IP core. This core stores the code and data information inside the available Block RAM (BRAM) resources available on the FPGA. By default, the BRAM area stays uninitialized, but it is later filled in with the MB binary in the provisioning process. By design, primary MB memory is divided bit-wise into multiple BRAM elements, and the exact order is arbitrary and scrambled, making it challenging to reassemble the original data. The contents of the BRAM are however of high importance for the attacker, as they contain key material and PIN hashes. Being able to obtain the PIN hashes, for example, would enable brute-forcing of PINs using external and more powerful hardware, and having the secret keys makes it possible to decrypt confidential information or even reproduce designs.

For this competition, the bitstream gets generated, encrypted, and signed by MITRE before it is distributed to the teams. That encrypted bitstream can be only decrypted and programmed by MITRE's custom first-stage-bootloader (FSBL) using the encryption keys stored on the board efuses, which makes it only executable on the officially provided boards. All Xilinx 7-series FPGAs support a "readback" operation, where the configuration memory of the FPGA is read back for verification of correct programming, which is available over JTAG/USB, however all encrypted bitstreams disable this functionality by writing security bits to the FPGA. This makes it difficult to retrieve the bitstream after initial boot, and even if possible, all the BRAM contents would be still scrambled. Our proposed attack consists of 2 phases: retrieval of the FPGA configuration and reassembly of the BRAM contents.

For the first phase, we analyzed different ways to read out the configuration. While readback over JTAG is blocked, we discovered that readback over the Zynq-7000 PCAP interface was still available. PCAP is a configuration bus that connects the ARM core (PS) and the FPGA core (PL) of Zynq-7000 SoCs, and is used by the FSBL for the initial configuration. Interestingly and counterintuitively, readback security bits set for the bitstream do not have any effect on this interface. The PCAP interface is controlled completely from the PS. The documentation on this interface is very sparse and there are no public implementations of readback tools, though there is a kernel patch from 2018 that never made it into upstream[1]. Using an enormous reverse-engineering effort, we created a kernel-backport for a custom "Petalinux" build and later a bare metal ARM tool executable by U-Boot via "uEnv.txt" that is capable of issuing the correct command sequence to read back the BRAM contents over PCAP.

For the second step the BRAM information needs to be assembled. The first method we tried was to manually program the extracted configuration memory on a non-MITRE board, which took up to 2 days. This would help us to have more boards available for automated online PIN brute-forcing. However, we eventually got most of the remaining flags by figuring out how to decode the readback contents back to the original MB binary. We used SymbiFlow's Project X-Ray data[2] as well as custom statistical brute-force, where we obtained 100 possible bit-sequences, of which 32 would correspond to the bits in each MB memory word in some random design-specific order. After brute-forcing that 3200 combinations (<1 min), we were able to reassemble the original MB binaries. Then we extracted the secrets, or simply modified and repackaged the MB. However, the limitation of this attack is that it would be unsuitable to extract secrets hardcoded into the FPGA logic.

[1] https://patchwork.kernel.org/patch/10546629/, interestingly there is current kernel support for the ZynqMP. However, the support for the normal Zynq was never implemented nor documented
[2] Prjxray: Documenting the Xilinx 7-series bit-stream format https://github.com/SymbiFlow/prjxray

Defense: Chain of dependencies and hiding secrets in the fabric

The guiding principle of our design's protections was "defense in depth" — we implemented security features and hardening over many different domains so that even if our design does contain some vulnerability, we have other protections that prevent successful exploitation. For our design we assumed that there might be a way to extract information from the Microblaze, e.g. by using readbacks or by using an unknown vulnerability in our code.
One important feature of our design is the **user key derivation chain**, which is used to turn user PINs into keys that can be used (with the correct region key) to decrypt songs that the user has authorization to play. The goal of this feature was to protect our *Pin Extraction* and *Unauthorized Play* flags. We wanted derivation steps that correctly authenticate users and authorize users to play their songs, making the user PIN itself mandatory for playback and digital-out. We wanted the brute-forcing of PINs, both online (using the attack boards) and offline (using external computation) to be as difficult as possible, with it ideally taking longer to complete than the duration of the Attack Phase.

The derivation steps follow a chain of operations, starting with the user PIN input: (A) HMAC on the PS side; (B) HMAC on the PL side; (C) Argon2ID key derivation on the PS and PIN verification using 2 bytes of the derived bytes as "check bytes", the remaining bytes passed on to the next step; (D) PL mixing using a secret that is hardcoded into the FPGA fabric for final derivation.[3] The results of these steps are cryptographic keys that can be used to play and share songs (if authorized) .[4]
This process accomplishes all of the following goals:

- **User authentication:** We check that the PIN is correct in step C using "check bytes" stored in the user information file. This allows our design to verify successful logins on the PS side without querying the PL. The PL verifies the derived keys against the desired public key regardless.
- **Authorized song decryption:** Songs can only be decrypted if the PIN is known and the derivation process yields the correct user keys. Only knowing the "check bytes" for a user from the information file is not sufficient to continue to step D: the rest of the hash from part C is needed, which is not stored anywhere.
- **Offline brute force protection:** To get to the step C check, the PIN must be combined with HMAC secrets from both the PS and PL, which are time-consuming to reverse-engineer. The Argon2 parameters are tuned to be as slow as possible within the time constraints on the board, however we do not rely on its security for offline-attacks. Since we only use 2 check bytes, statistically every 65025 pins a pin which passes the first check. There will be approximately 1500 offline-passing 8-digit PINs, which will all need to be checked by attempting to decrypt/play a user's song. This will need to be done online since step D includes a secret hardcoded in the FPGA logic, which cannot be extracted without extremely difficult (questionably possible) reverse engineering.
- **Online brute force protection:** The Argon2 parameters prevent trying PINs online faster than the 1 second login time constraint. Additionally, Step B on the MB locks out the DRM module for the exact time it needs to compute the Argon2ID. Thus, the rate-limiting cannot be bypassed. Similar to the offline-attack, there are a number of invalid pins that will pass the "check bytes", which is intended by design.

Every invalid pin that is passed to the FPGA will cause a reset of the board and a mandatory delay of 4 seconds. To keep the HMAC in the ARM binary on the untrusted OS secure, the binary is stripped of symbols, obfuscated and packed. The attacker would need to reverse-engineer this binary first before being able to build a custom binary.

[3] Definitions are: "PS" for the ARM side in the untrusted OS, "PL" for the FPGA based logic
[4] The key can be described as: ukey = FPGA_Mixer(Argon2ID_PS(HMAC_PL(HMAC_PS(pin)))[0:13])