🐈✨

Defensive Measure -- The Protectonator

As evidenced by the inclusion of the power trace collector, the theme of this year's eCTF competition revolved around performing power trace analysis of the attacker-controlled DZ. We assumed during the design phase that protecting our design against Differential Power Analysis (DPA) attacks would be critical to securing the key material which protected our flags.

From a practical standpoint, DPA attacks against cryptosystems revolve around collecting a number of power traces which themselves are associated with some known data about the system (e.g., an attack against AES decryption of known ciphertexts), aligning these traces via correlation/convolution, and creating a model with which to test hypotheses. The attack succeeds when a hypothesis with a particularly high correlation is discovered. Our Protectonator system provides countermeasures against both the alignment of the power traces as well as the formation of hypotheses in the first place.

The Protectonator itself is a system of systems, and is composed of the following primitive components: our choice of XChaCha20 as our cryptographic cipher; the XChaCha20-based CSPRNG seeded with a combination of static and dynamic data, used to add random delays to sensitive operations; and the swarm key.

During the design phase, we performed research against the existing literature regarding DPA attacks against various cryptosystems. During this process, we noted a distinct lack of published work pertaining to the XChaCha20 cipher. We performed our own analysis of XChaCha20 with respect to determining how difficult the implementation of a DPA attack would be, and came to the following conclusion: due to its nature as an Add, Rotate, XOR (ARX) cipher, the formation of a hypothesis against the algorithm would be nontrivial. Existing literature supports this assumption (https://eprint.iacr.org/2019/335.pdf).

Since we understood that performing DPA attacks required aligning power traces, we implemented a CSPRNG itself based off of the XChaCha20 keystream generation function to generate randomness used to delay sensitive operations. These random delays make aligning traces nontrivial as attackers would need to write software to recognize them and filter them from traces before they might be correlated. The CSPRNG is seeded from three sources: the SBC binary as a device-unique secret, the SSS during provisioning time, and from the SYSTICK counter (which is nondeterministically updated as messages are sent and received). These random sources are mixed via the Blake2B hashing algorithm.

Our message format includes a header which contains a randomly-generated XChaCha20 key encrypted with a key derived from a X25519 key exchange. We wished to harden the recovery of message keys against DPA. Any attack against this information would require knowledge of the ciphertext so that hypotheses could be correctly formed. The encrypted keys are combined with a XOR pad, the swarm key, to make recovery more difficult. Any successful attack against this mechanism would first need to leak the swarm key. This key is split into two components which are independently applied to encrypted message keys, doubling the work an attacker would need to perform to recover the key.

Over the course of the competition, no team was able to recover flags against our design which would imply a DPA capability, so the effectiveness of our countermeasures went untested. Prior to entry into the attack phase, we attempted to develop DPA attacks against our design to little success. However, our defenses were ad-hoc, and could benefit from a more formal analysis as to how they perturbe power traces and interfere with attacker efforts.

🐈✨

Offensive Measure -- Extraction of UAVID using Side channel observations

All designs use different methods of encrypting traffic, requiring specific side channel tooling for each of the teams to extract the key material. We tried to find a side channel method which can extract at least the UAVID from all designs. For this, we created the Sidechannel Hardware Attack and Reconnaissance Kit (SHARK). SHARK automatically abuses a vulnerability inside the CPU code of the Dropzone (DZ), injects an implant, and communicates with it. The DZ can receive broadcasts, direct messages, and FAA messages. For the competition, the DZ is limited so that it cannot actually send messages out, neither via the normal radio interface, nor the FAA channel. The only way to extract information is through observing CPU or Controller activity. The FAA channel can be used to send commands to the implant on the CPU.

Our approach was to use side channel observations as an exfiltration channel. The first version of our attack utilized message length as a way of encoding bits of information. However, this method proved to be more difficult when encountering countermeasures such as rate-limiting. The final version of our attack used the generation of FAA channel messages as a signaling method. We use shannon entropy as a measurement to draw conclusions about the activity of the controller.

From prior successful attacks, we knew that the UAVID flag is in the format “ectf{uavid_xxxxxxxxxxxxxxxx}”. We used this fact to reduce the amount of data needed to be exfiltrated from the DZ. When run, the implant on the CPU waits until it receives a broadcast message or a direct message, and searches for the string “uavid”. If it exists, the rest of the flag is extracted and encoded into 64 bits (without error correction). After the message is received by the controller, we block all further incoming communication to avoid side effects which could corrupt our measurement.

To extract the information, we use 2 second slots per bit. Between each bit, the CPU will sleep for 2 seconds. If a bit is a *1*, the CPU will send a message over the FAA channel (which will not actually leave the DZ). When a bit is a *0*, it will just wait. In total the attack takes between 2 to 3 minutes to complete.In the power trace, we can observe changes of the shannon entropy in relation to a previously measured baseline. If a controller waits for messages, we will see an entropy very close to the baseline. When a message is sent, regardless of the particular kind of messages, the Shannon entropy will visibly increase.

While this attack would have worked against all teams in the attack phase, we only used it on a few due to time constraints and technical issues with the deployments. The attack can be also used for decryption of targeted messages against designs which use global keys and do not validate message recipients. One limitation of our method is that the DZ does not have access to a high-precision timer, and thus noise from the emulated environment can corrupt the side channel. This issue required us to repeat the measurements multiple times to recover any problematic bits.

In theory, the attack can be prevented by masking wait cycles with random activity, ensuring that the baseline entropy would be high. This would make it difficult for an attacker to observe individual bits. In addition, rate limiting on the FAA channel can be a solution. This attack could be greatly improved by performing extra encoding steps prior to exfiltration. For example, utilizing Manchester encoding with ECC could reduce the overall runtime by reducing corruption.