Unanticipated Snapshots: A New Type of Attacks against Intel SGX

In this document, we present a new type of attack against Intel SGX named unanticipated snapshot attacks. To the best of our knowledge, the attack has not been reported by anyone before. And the attack affects all versions of Intel SGX SDK with a feature named Intel SGX Protected File System Library (SGX–PFS), which is the target of the new attack. SGX–PFS provides the essential ability to secure the file I/O for SGX applications. Thus, any vulnerability in SGX–PFS may impact a great portion of SGX SDK's users.

The remainder of the document is organized as follows. We will first describe the new type of attacks in general, then explain why SGX–PFS is vulnerable to it, and finally present a demo to show such attacks are practical and effective. The demo shows how an attacker can exploit the vulnerability in SGX–PFS to gain complete access to a Redis server protected by SGX. We will also give suggestions on how to fix the vulnerability in SGX–PFS.

About SGX–PFS

SGX–PFS provides protected files API for SGX enclaves. The API is similar to its libc's counterpart (fopen, fread, fwrite, etc), but offers the extra protection required by SGX enclaves: the files are encrypted and saved on the untrusted disk during a write operation, and they are verified for *confidentiality* and *integrity* during a read operation. SGX–PFS also offers *freshness* in the sense that a protected file of SGX–PFS cannot be partially rolled backed to a previous version. A protected file is also guaranteed to be *consistent* regardless of any crashes.

SGX–PFS is known to have some security limitations. For example, a documented non–objective is swapping attacks (swap two protected files of the same name). Another example is side–channel attacks (using side channels like file sizes and file offsets). These are known issues that are documented by Intel and understood by end–users. But what we are about to describe–––the unanticipated snapshot attacks–––has been overlooked by both Intel (as far as we know) and the SGX community.

Unanticipated snapshots

For our discussion, we define a *snapshot* as the persistent state of a system at a point of time. For example, we can create a snapshot of a container by saving a container's file changes with "docker commit" .

In a normal (non–SGX) environment, the creation of snapshots is almost always triggered by legitimate users on purpose. But in an SGX environment, since the adversary can control everything beyond the enclave, including monitoring every I/O operation and saving every bit of on–disk data, the snapshot of an enclave may be captured by the adversary at an arbitrary timing. In other words, the snapshots of an enclave may be taken involuntarily by the adversary. And some of the snapshots can be a surprise to the application logic inside the enclave, causing unexpected behaviors of the enclave, even security loopholes in some cases (as we will show later).

We call the snapshots that are unanticipated by an enclave *unanticipated snapshots*. We call an attack that attempts to cause unexpected behaviors of an enclave by capturing and replaying unanticipated snapshots of the enclave an *unanticipated snapshot attack*.

Why SGX–PFS is vulnerable

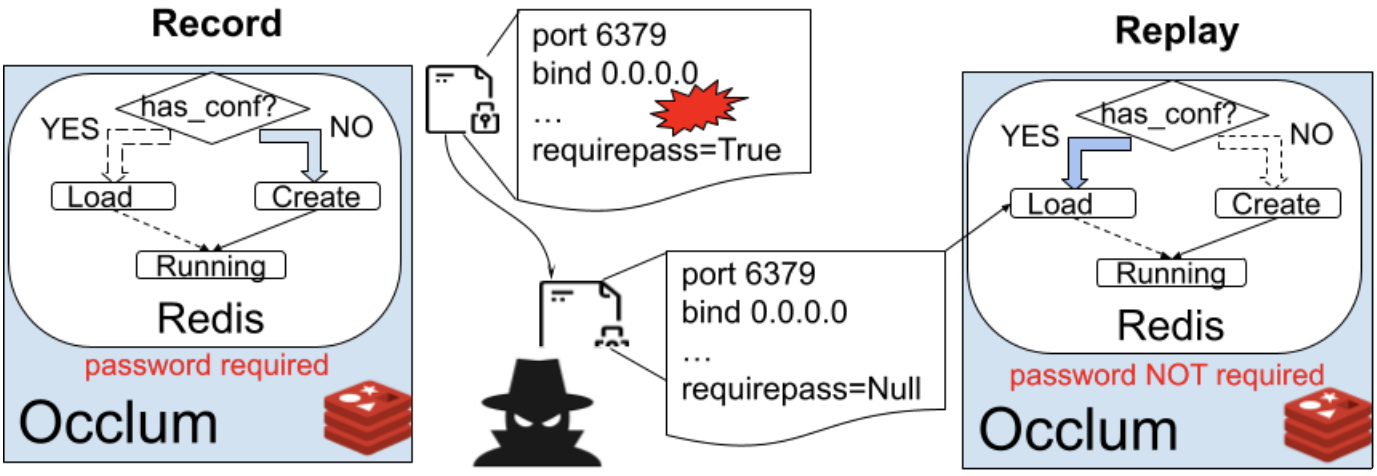
We find SGX–PFS may generate unanticipated snapshots of a protected file. Thus, generally speaking, any SGX application that uses SGX–PFS is vulnerable to unanticipated snapshot attacks.

To understand why SGX–PFS generates unanticipated snapshots, we need to first understand the inner working of SGX–PFS. For each protected file, SGX–PFS maintains three key data structures: a Merkle Hash Tree (for security), a cache (for efficiency), and a recovery log (for consistency). The cache has a fixed size and contains the most recently used nodes of the Merkle Hash Tree. The use of the cache can greatly improve the performance of SGX–PFS. When the cache is full, SGX–PFS needs to flush the dirty nodes of the cache to the disk, generating a snapshot of the protected file. The problem is that the flush operation is completely transparent to the users of SGX–PFS. And thus, the generated snapshots may be unexpected to the users of SGX–PFS.

Proof of concept

At a first glance, you may think unanticipated snapshots harmless. Yes, they are–––at least for most of the time. But for sophisticated attackers, they can be useful weapons to penetrate the protection of SGX...
Let's look into a concrete example.

We take the ‘Redis’ server as an example to show how an attack can make a Redis server lose its authentication using the above ‘snapshot attack’. We assume that the Redis server operates a database of sensitive data (e.g., medical records) on a cloud. That is why it needs to be protected by an SGX enclave. For the ease of using SGX, we adopt Occlum (https://github.com/occlum/occlum) to deploy this proof of concept demo.



Occlum provides an encrypted file system that uses SGX PFS to protect each individual file or directory. Thus, Redis can enjoy transparent file encryption. In this setting, the Occlum application starts in two steps: 1) create a Redis configuration file, "redis.conf", if it does not exist, and 2) start the Redis server to serve user requests. The "redis.conf" file needs to be created on the fly because it contains the credentials (e.g., the ‘requirepass’ field) of authenticated Redis clients, which are fetched from a trusted source on startup via remote attestation.

Note that the ‘requirepass’ entry is the most important security–related setting in Redis config. It confines that Redis client must use the password specified in this field to access Redis server. When the ‘requirepass’ is not specified, no password is required to access the server.

Due to the transparent flush behavior, SGX PFS might produce a truncated "redis.conf" during the Redis config file preparation step. This file may include ‘port’, ‘bind’, and every other field but not the ‘requirepass’. Even worse, this can be controlled by a privileged attacker. In theory, the attacker has the ability to intercept the untrusted part of an SGX application, to catch the exact snapshot file (the one without ‘requirepass’ field). Afterward, the attacker can start a new enclave with the truncated version, and successfully run the Redis server without a password authentication process.

We also prepare example code to demonstrate this PoC.

Instructions

Here we give detailed instructions to reproduce the attack using the provided code.

1 Build Occlum/SDK

First of all, anyone willing to run the demo must install Occlum and modified SGX SDK (optional). We provide a Dockerfile for fast Occlum/SDK installation. The modified SGX SDK is to help the attacker to decide the timing.

2 Build the target program

The target program is at **bash_redis** folder.

The victim program is running a bash-shell script (**occlum_bash.sh**). It first fetches a provisioned password, to generate a customized Redis configuration file, with the 'requirepass' entry filled. Then it invokes a Redis server, according to the customized config file.

To build the config generation program, run make. To build the Redis server, execute **./download_and_build_redis_glibc.sh**

3 Run the demo

Prepare 2 terminals, one for the victim and one for the attacker.
Run **run_redis.sh** at the victim terminal. This script will start up the target program.

3.1 Record phase

The attacker needs to take a snapshot (using **./take_snapshot_step-1.sh**) when he/she can determine that the snapshot is flushing. In the demo, the attacker can get a clear notice from the victim's terminal to launch the 'take snapshot' script. This is because we modified the untrusted part of Intel SGX SDK, printing helper messages to facilitate the attacker to decide the timing. After the Redis server is running, run **./take_snapshot_step-2.sh** to complete the snapshot collection.

3.2 Replay phase

The attacker can then execute **replay_redis.sh** to replay the enclave using our collected snapshot. After that, any client can log onto the Redis server without password authentication.

Video

https://drive.google.com/file/d/1UC742H_bMerNG-YaNnS5EpNUopx8GO2h/view?usp=sharing

Impact

We believe the impact of the problem is wide since the snapshot attack can be conducted in other cases. For example, Gramine (formerly called *Graphene,*

https://github.com/gramineproject/gramine) is using PFS to protect disk I/O. Utilizing the unanticipated snapshot attack illustrated above, an attacker can get a truncated manifest file where there should have been some key configurations (e.g., sgx.file_check_policy = "[strict|allow_all_but_log]").

Recommended solution

The unanticipated snapshot attacks arise from the gap between the possible on–disk states expected by an application and those allowed by the disk I/O stack. In theory, this gap can be bridged by manipulating the entire persistent state of an application with a transactional database. But in practice, this solution hardly works. For one thing, it is difficult, if not impossible, for programmers to figure out all possible on–disk states of a complex system and the transitions between them. Further, most applications persist at least part of their states with file systems, whose guarantees for the atomicity, ordering, or consistency of file system operations are not only weak but also vary from one implementation to another.

A good remedy to the 'snapshot attack' is to devise *a log–structured block device* for the TEEs. Our insight is that using a log–structured design is not only superior in performance but also convenient to achieve security goals.

The flush operation in SGX–PFS has only a job of persistency, which is to flush all dirty data as well as metadata to the storage media. To defend against such attacks, the devised log–structured block device has one extra requirement of atomicity—either all or nothing is flushed.

For this purpose, we introduce the data commit record to the journal. We say a data segment record uncommited if it is not followed with at least one data commit record. All uncommitted data segment records are ignored when 1) recovering from a crash, and 2) performing both data and index checkpointing.
More specifically, here is how a flush should be performed in four steps: 1) Write the dirty portion of the current segments to the disk; 2) Write data segment records for the current segments to the journal; 3) Write a data commit record to the journal; 4) Issue a flush request to the disk.