Defense

Background

A **cryptographic hash function**[1] is a one-way function which takes input of any size and generates a fixed size output. Hash functions guarantee that it is infeasible to find two inputs which yield the same hash value or find an input which matches a given hash value. Small changes to an input should also generate a completely different hash. These properties make hash functions ideal for detecting file modifications.

Description of Defense

Our system utilizes **Blake3**[2], a secure, fast cryptographic hash function, to verify that DRM audio files have not been altered. Additionally, **two secret keys**, a metadata key and a chunk key, are generated to authenticate song headers and song audio chunks, respectively. This ensures that only songs that were protected using our provisioning tools may be played on the miPod.

```
import blake3
...
# hash the song header using the metadata key
b = blake3(key=metadata_key)
b.update(song_metadata)
header_hash = b.digest() #write this to .drm file
...
# hash each audio chunk using the chunk key
for song_chunk in song:
   b = blake3(key=chunk_key)
   b.update(song_chunk)
   chunk_hash = b.digest() #write this to .drm file
```

Figure 1. Song hashing process for protecting audio files.

When a song is loaded to be played on the miPod, the Audio Digital Rights Management System computes the header hash in secure memory and compares it to the pre-computed hash stored in the DRM audio file. The process is identical for song audio chunks.

Why It Didn't Work

A critical weakness in our design is that the song header and audio are verified **separately**, rather than as one **atomic unit**. Thus, an attacker can swap headers between two correctly provisioned songs. For example, if a user with privileges to play Song A but not Song B swaps the two song headers, they will now be able to play Song B, bypassing user authentication checks.

Remediation

The recommended solution to address the weakness described above is to generate a **cryptographic nonce**[3] for each song. A nonce is an arbitrary value which should be used only once. While hashing song chunks, the audio chunk is concatenated to the song nonce, shown below. Because the nonce is **unique to each song**, audio chunks that are swapped will have different hashes, resulting in failed integrity checks. This security guarantee will also apply to song headers.

[1] Cryptographic Hash Functions: https://en.wikipedia.org/wiki/Cryptographic_hash_function
[2] Blake3: https://github.com/BLAKE3-team/BLAKE3
[3] Cryptographic Nonce: https://en.wikipedia.org/wiki/Cryptographic_nonce

```
# hash each (audio chunk || song nonce) using the chunk key
for song_chunk in song:
    b = blake3(key=chunk_key)
-   b.update(song_chunk)
+   b.update(song_chunk+song_nonce)
    chunk_hash = b.digest() #write this to file
```

Figure 2. Introduction of song nonce to prevent song block swapping.

Another mitigation would be to store a hash of the entire DRM audio file and verify it before playback. This will also protect the Music Tamper flag, but is hard to implement due to performance constraints.

Attack

The Vulnerability

The eCTF rules state that "**music that fails integrity or authentication checks should not be played [at all]**."[4] In other words, if the miPod begins playing a song, it must play the entire song correctly. **4 out of the 7 teams** in the attack phase had this vulnerability, leading to many trivial Music Tamper captures. Even though the rules are explicit, these teams misinterpreted the rules thinking that it is acceptable for the miPod to begin playing a song and abort once a tampered audio chunk has been reached.

Attack Strategy

This attack only applies to systems which do not verify the integrity audio **before starting playback**. The attack consists of 2 phases: **reconnaissance** and **tamper**. In the reconnaissance phase, we want to understand the file format of the protected DRM audio files. We need to know how many bytes their song header is, what kind of metadata is appended to audio chunks, etc. We can gather this information from a team's provisioning tools, specifically, the *protectSong* script.

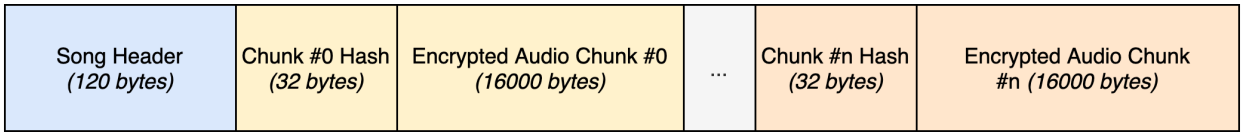

Figure 3. One team's DRM audio file format.

With this information in mind, the tamper phase is trivial. For example, if we wanted to zero the first byte of the 2nd audio chunk, we modify byte 120+16032+32 = 16184. We can use this shell command:

```
printf '\x00' | dd of=musictamper.drm bs=1 seek=16184 count=1 conv=notrunc
```

Figure 4. Bash command which zeroes out the 16184th byte in *musictamper.drm*.

This attack is a valid Music Tamper capture because the Audio Digital Rights Management System will begin playing the song and then abort once the second audio chunk is reached for playback.

Proposed Fix

This vulnerability is not the result of poor implementation, but rather a lack of understanding and proper **threat modeling**. Threat modeling[5] is the process by which possible threats are identified, objectives/goals are clarified, and features/mitigations are prioritized. Through this process, engineers should be able to clearly identify their attack surface and use these to drive their implementation.

[4] MITRE eCTF 2020 Rules, Ch. 3.2.2: https://huskyrecords.net/hellothere/2020_ectf_rules_v1.1.pdf
[5] Threat Modeling: https://en.wikipedia.org/wiki/Threat_model