Western Digital®

WHITE PAPER NOVEMBER 2020

Replay Protected Memory Block (RPMB) – Protocol Vulnerabilities

Contents

Introduction

The Replay Protected Memory Block (RPMB) protocol, initially defined in JESD84-A441, is a security protocol that provides the ability for a host to store data in a specific memory area of the device in an authenticated and replay-protected manner.

An authentication key is programmed into the Universal Flash Storage (UFS) device memory in a secure environment, such as OEM production. This key is used to authenticate read and write accesses made to RPMB with a Message Authentication Code (MAC). Further, count registers and nonces are used to protect against replay attacks.

The RPMB is contained in its own well-known logical unit that can be divided into multiple regions, each with associated cryptographically significant parameters (authentication key, write counter, etc...). Each RPMB region can process a single authenticated operation at any given point in time.

In this document, we describe two vulnerabilities in an a priori reasonable implementations of the host side of the protocol and give possible mitigation strategies.

Relevant Use Cases for RPMB

Software Downgrade Protection

Consider a scenario in which a manufacturer must push several software updates to a device (such as a phone, a car, etc.). During the initial update, the new software image is written into the device's main area while the information about the software release version is stored in the RPMB.

Then, imagine that the manufacturer discovers that this version has a security breach or a safety bug. The manufacturer issues another software update to fix the problem. As before, the new software image is written into the device's main area and updated version information is stored in the RPMB.

But a hacker may use this same mechanism to downgrade the software on a user's device and take advantage of security breaches or bugs within previously released versions. They may try to mimic the procedure that the manufacturer used when pushing out the upgrade — but the hacker would actually push out an earlier, compromised version of the application or operating system.

Software using RPMB to protect itself from a downgrade attack would check for a new, updated version number during the upgrade procedure. If the **new** version number is lower than the one already present in RPMB, the installer would reject the **update**. Due to the nature of RPMB, there should be no way for an attacker to change the software version information stored in the RPMB, as that would require access to the secret key.

Brute-force Protection

In this example, RPMB can be used to ensure that only an authorized individual can unlock a device (such as a phone, car, or computer).

First, the user records a PIN code, fingerprint, or swipe sequence to unlock the device. Subsequently, whenever an attempt to unlock the device occurs, the time of each attempt is recorded in the RPMB. If too many attempts are made within a specified period of time, software controlling the lock will prevent further attempts to unlock the device for a set period of time. If a hacker tries to unlock the device — even with the use of automated tools that would try to inject every possible PIN code until the correct one is found — the hacker will be stopped when the specified number of incorrect unlock trials within a specified period is reached. The hacker cannot override the tracking of unlock attempts because the information about the attempts is stored in the RPMB. Unless the hacker stumbles upon the right PIN in the first few attempts, it becomes nearly impossible for someone to unlock the device without the pin.

Updateable Secure Boot

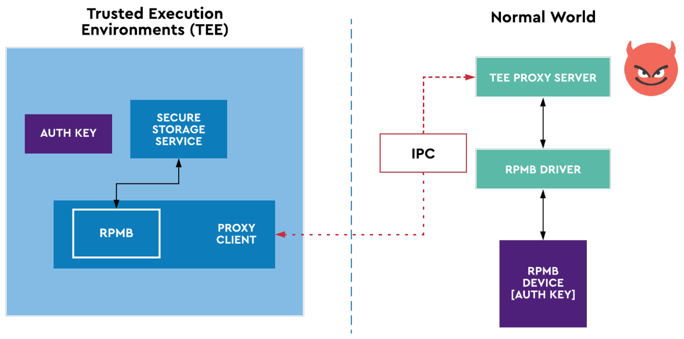
Prevention of an unauthorized (hacked) code from running on a device begins with an assurance that the very first piece of code read originates from the storage device for a legitimate execute. This initial code — the bootloader — is located on the device boot partition, which must be write-protected from malware modification.

A permanent write-protect mechanism offers users a reliable way to protect the boot area. Additionally, it also precludes manufacturers from updating those areas, if necessary. The secured write-protect feature ensures that every change to the write protection configuration itself must be authenticated by using the RPMB secret key. The secured write-protect mechanism is primarily used to protect the boot code, or other sensitive data on the device from changes or deletion by unauthorized applications.

Secure Updateable State for TEE

Trusted Execution Environments (TEE) can use the RPMB feature to provide replay-protected non-volatile storage areas for applications.

In order for the application to use the RPMB secure storage, the application only needs the normal-world kernel to relay RPMB requests to the storage device, but does not otherwise need to trust the normal world kernel.



Description of Vulnerabilities

Two attacks on the RPMB protocol have been identified. They allow the adversary to make the legitimate host user believe that the state of the RPMB is different from what it truly is.

We will give a generic description of these two scenarios, as well as identify their impacts on various identified use cases for RPMB. With primary focus to RPMB protocols that are relevant and describe the attack at a somewhat abstract level. However, one can easily translate the attacks to fully conform with the RPMB specification. In particular, though we don't explicitly specify all the components in the attack messages, we state the relevant ones and the rest can be worked out.

Note, the description focuses on two commands: Authenticated data write request, and Authenticated data read request. The described attacks apply equally if Authenticated data read/write requests are replaced by Secure Write Protect Configuration Block read/write requests.

A list of Common Vulnerabilities and Exposures (CVE) issued in this regard can be found on Western Digital's website.

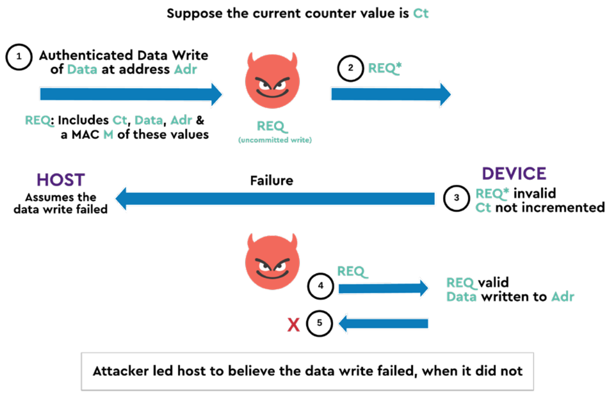
Scenario One

In the first attack scenario, the host is led to believe that the data write failed.

Initially the host submits an authenticated data write request (1) to the device. The request, REQ, includes the write counter value Ct, the data to be written, the address Adr, and a MAC M on these values. The adversary intercepts this request, memorizes it, and (2) forwards a modified version REQ* that will fail the MAC verification (thus the counter value is not incremented). We call REQ an uncommitted write because the attacker has prevented it from being committed to the RPMB.

As the MAC verification failed, the device responds with a failure (3) to the host. This could lead the host to believe that the request REQ failed and the memory contents at Adr have not been updated.

Later, the adversary forwards the initial request REQ (4) to the device and intercepts the successful status sent back by the device (thus committing an uncommitted write). As a result, the memory contents at Adr do get updated but the host does not get any information about it.


Suppose the current counter value is Ct
1 Authenticated Data Write of Data at address Adr
REQ: Includes Ct, Data, Adr & a MAC M of these values
REQ (uncommitted write)
2 REQ*
HOST Assumes the data write failed
Failure
DEVICE
3 REQ* invalid Ct not incremented
4 REQ
REQ valid Data written to Adr
X 5
Attacker led host to believe the data write failed, when it did not

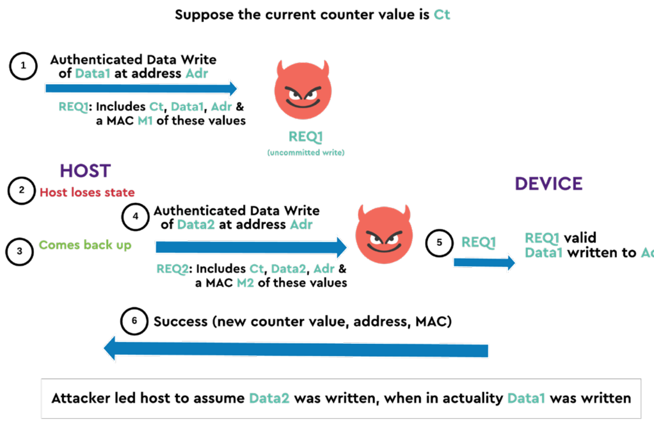
Scenario Two

In the second attack scenario, the host is led to believe that a piece of data A was written. However, it was another piece of data B that was written.

The host issues an authenticated data write request (1) to the device. The request, REQ1, includes the write counter value Ct, the data to be written Data1, the address Adr, and a MAC M1 on these values. The adversary intercepts this request, memorizes it and causes the host to lose state (2) – by cutting off power for instance. Thus, REQ1 stays with the attacker as an uncommitted write.

When the host comes back (3), the host may then attempt to issue an authenticated data write request to the same location but with some different data. Suppose it now issues a new request, REQ2, which includes the counter Ct and the address Adr but a different piece of data Data2 (4). The adversary intercepts this new request and (5) forwards the old request REQ1 to the device instead.

Note, REQ1 is still a valid request because the counter value has not changed. As a result, Data1 gets written to the device. When the host (6) receives the successful response back, it would assume that REQ2 was successfully executed and Data2 has been written at Adr.


Suppose the current counter value is Ct
1 Authenticated Data Write of Data1 at address Adr
REQ1: Includes Ct, Data1, Adr & a MAC M1 of these values
REQ1 (uncommitted write)
HOST
2 Host loses state
3 Comes back up
4 Authenticated Data Write of Data2 at address Adr
REQ2: Includes Ct, Data2, Adr & a MAC M2 of these values
5 REQ1
DEVICE
REQ1 valid Data1 written to Adr
6 Success (new counter value, address, MAC)
Attacker led host to assume Data2 was written, when in actuality Data1 was written

Impact of Attack

In the second attack scenario, the host was led to believe that a piece of data A was written. However, it was another piece of data B that was written.

Software Version Downgrade Attack Prevention

When two successive updates to the software are issued, an adversary may be able to use Attack 2 to cause the system to believe that the RPMB holds the latest version number while it actually holds the penultimate one. The adversary may then be able to cause the system to downgrade to this penultimate software version and to maliciously use the associated vulnerability for far longer than it should have.

Brute-force Attack Prevention

In this use scenario, the attacks described in this document do not appear to provide any significant benefit to an adversary for a reasonably implemented host software stack.

Updateable Secure Boot

If two successive updates to the boot code are attempted, with the second one issued to mitigate a security vulnerability present in the first one, the adversary may be able to cause the issuing host to believe that the later version is installed when only the former one was.

Secure Updateable State for TEE

There are a variety of use cases one can envision for an application running in a TEE. In nearly all of them, it is important that the application be able to trust the TEE storage to correctly write the data that was sent, and not older uncommitted write.

Depending on the design of the file system the impact of the vulnerability may be narrower or wider-ranging. Specifically, a non-journaling file system which could already reach an inconsistent state when faced with sudden power loss will obviously be vulnerable.

Proposed Mitigations

Consistency Model of RPMB

In the traditional shared-memory model of distributed systems, the single weakest notion of register practically useful for building bigger systems is a safe register. At the very least, a safe register needs to provide confidence the property that read operations not concurrent with write operations should return the most recent successfully written value.

The previously described attacks show that the RPMB, as seen from the perspective of the legitimate user, does not even provide a safe register. Indeed, Attack 2 in particular, shows a scenario where a subsequent read would return a different value from the most recently written one.

In this section, we will provide a mechanism to address the two previously described vulnerabilities and ensure that RPMB at each address can be treated as a safe register.

Furthermore, for most applications, safety is typically an insufficient property and stronger notions like regularity or atomicity/linearizability are required. The proposed mitigation technique will actually ensure regularity as well. For a system in need of atomicity and eventual termination of writes, standard file system consistency techniques – two stage logging, journaling etc.. – need to be used by the host system together with the proposed mitigation.

Initialization Sequence

We know that a write request contains a counter value and a MAC value (among other things). The device checks that the counter value matches with the current write counter, and that the MAC is valid. If both checks pass, it increments the counter value and includes the new value in the response message. If either of the checks fail, then the old counter value is included.

We will use the counter value in the response messages to check whether **some** write request was executed successfully or not.

We propose that an initialization sequence be run every time the host recovers from a loss of state. The steps of the sequence are described below. First, the host reads the value of the write counter; let's say it's Ct. Then, it runs a loop that tries to write some data to a **dummy block** with counter value Ct. The loop runs until the host sees a value of Ct+1 in the response message.

```
[1] Read the write counter value (say it’s Ct)
[2] DO
[3] Authenticated-Data-Write of some data to the dummy
    block with counter value Ct
[4] WHILE (counter value in response NOT EQUAL TO Ct + 1
```

The dummy block could be any block that is not normally used for reading and writing. We later discuss what steps to take if such a block is not available.

At the start of the initialization sequence, the attacker may have uncommitted writes with counter value Ct or less. It could use them later to launch attacks like the second one above. The purpose of the initialization sequence is to **invalidate** the uncommitted writes by incrementing the counter. Once the counter is incremented, attacker cannot commit any of the uncommitted writes it has, thus preventing problems later.

The loop above terminates when the host receives a response message with an incremented counter value, so some write request must have successfully executed. This write request may not be one of the requests generated in the loop; it could even be one of the uncommitted writes that the attacker holds. Either way, all the other uncommitted writes of the attacker become invalid.

Once the initialization sequence is complete, it is safe to issue reads and writes to the RPMB.

Reading and Writing

As the adversary does not have any valid uncommitted writes left, reading is safe.

In order to write data D to a given address Adr, the following loop should be executed, assuming the write counter value is Ct*.

```
[1] DO
[2] Authenticated-Data-Write of some data to the dummy
    block with counter value Ct
[3] WHILE (counter value in response NOT EQUAL TO Ct + 1
```

Note that the only write requests generated by the host with counter value Ct* are the ones that write D to Adr. Thus, if the counter value is incremented in the response message, it could only be because one of these write requests have been executed successfully. This means that D has been written to Adr.

The attacker could certainly interfere with the write process. It could block a write request with Ct* from being executed, it could try to commit one of its uncommitted writes, it could block the response messages, etc. However, if a response message indeed contains Ct*+1, then only a write request generated in the loop could have successfully executed.

Attacks Reconsidered

Let us briefly see how the proposed mitigation strategy prevents the two attacks we described earlier.

In the first attack, when the host gets a failure response from the device, it assumes that the write attempt has failed. With the new write procedure we have, the host will see that the response message still contains the old counter value, so it will continue to generate new write requests. The host will give up only when a response message contains the new counter value, which ensures that the write attempt has succeeded.

In the second attack, the host begins writing to Adr right after it comes back up (recovers state). This allows the attacker to commit an old write (REQ1) with some different data without the host noticing it. In the proposed mitigation, the host will first run the initialization sequence before it does any **actual** reading or writing. Once the initialization sequence is complete, the attacker cannot commit any of the uncommitted writes from before. The host will try to write Data2 to Adr now with the help of the loop described earlier, which ensures that Adr is populated with Data2 only.

Additional Concerns

Every time the host recovers state, like after a power failure, it must run the initialization sequence before it attempts to do any real reading or writing.

Once the initialization sequence is complete, writes should be carried out one at a time. A new write operation should start only after the previous one has finished. If multiple writes are carried out at the same time (for the same RPMB region), then the attacker may be able to launch some attacks.

Frequently Asked Questions

Would the proposed mitigation lead to a significant increase in the number of times the write counter is incremented?

An adversary could cause the write counter to be incremented many times by forcing the initialization sequence to happen over and over again. The counter, however, is a 32-bit value, so it can easily accommodate the extra write operations.

What should I do if no dummy-block is available?

If a dummy-block is unavailable, it is possible to modify the mitigation strategy to re-use an existing block (say B*) for the initialization sequence. Instead of writing some data to B* directly, the host must first read from it (say the read data is D*). Then, the host should try to write D* to B* in the same manner as an arbitrary value is written to the dummy-block in the mitigation strategy. After the loop terminates, the host must perform one additional step. It should read B* to make sure that it still contains D* (since an attacker could commit an uncommitted write from before with some different data). If it doesn't, the host should take appropriate steps.

When should the initialization sequence be run?

The initialization sequence should be run whenever the host software may have lost state. It is necessary to run the initialization sequence to guarantee that all adversary-held uncommitted writes are invalidated. Loss of state may happen for a variety of reasons like power-cycle, soft reboot, software restart, etc.

References

[1] EMBEDDED MULTIMEDIACARD (e•MMC) e•MMC/CARD PRODUCT STANDARD, HIGH CAPACITY, including Reliable Write, Boot, Sleep Modes, Dual Data Rate, Multiple Partitions Supports, Security Enhancement, Background Operation and High Priority Interrupt (MMCA, 4.41), March 2020

[2] Android Trusty Tee
https://source.android.com/security/trusty

[3] Op-TEE
https://www.op-tee.org/

[4] Western Digital Security Bulletin WDC-20008
https://www.westerndigital.com/support/productsecurity/wdc-20008-replay-attack-vulnerabilities-rpmb-protocol-applications

Western Digital.

5601 Great Oaks Parkway
San Jose, CA 95119, USA

www.westerndigital.com