Security Assessment of Magma Collective's Dark Crystal Protocol

TABLE OF CONTENTS

EXECUTIVE SUMMARY

Scope and Methodology

Include Security performed a security assessment of Magma Collective’s Dark Crystal Protocol. The assessment team performed a 9 day effort spanning from July 23rd, 2021 – August 4th, 2021, using a code review standard assessment methodology which included a detailed review of all the components described in a manner consistent with the original Statement of Work (SOW).

Assessment Objectives

The objective of this assessment was to identify and confirm potential security vulnerabilities within targets in-scope of the SOW. The team assigned a qualitative risk ranking to each finding. Recommendations were provided for remediation steps which Magma Collective could implement to secure its applications and systems.

Findings Overview

IncludeSec identified 1 finding which was deemed to be “Low-Risk” that could pose some tangible security risk. Additionally, 3 “Informational” level findings were identified that do not immediately pose a security risk.

IncludeSec encourages Magma Collective to redefine the stated risk categorizations internally in a manner that incorporates internal knowledge regarding business model, customer risk, and mitigation environmental factors.

Next Steps

IncludeSec advises Magma Collective to remediate as many findings as possible in a prioritized manner and make systemic changes to the Software Development Life Cycle (SDLC) to prevent further vulnerabilities from being introduced into future release cycles. This report can be used by as a basis for any SDLC changes. IncludeSec welcomes the opportunity to assist Magma Collective in improving their SDLC in future engagements by providing security assessments of additional products. For inquiries or assistance scheduling remediation tests, please contact us at remediation@includesecurity.com.

RISK CATEGORIZATIONS

At the conclusion of the assessment, Include Security categorized findings into five levels of perceived security risk: Critical, High, Medium, Low, or Informational. **The risk categorizations below are guidelines that IncludeSec understands reflect best practices in the security industry and may differ from a client's internal perceived risk. Additionally, all risk is viewed as "location agnostic" as if the system in question was deployed on the Internet. It is common and encouraged that all clients recategorize findings based on their internal business risk tolerances. Any discrepancies between assigned risk and internal perceived risk are addressed during the course of remediation testing.**

Critical-Risk findings are those that pose an immediate and serious threat to the company's infrastructure and customers. This includes loss of system, access, or application control, compromise of administrative accounts or restriction of system functions, or the exposure of confidential information. These threats should take priority during remediation efforts.

High-Risk findings are those that could pose serious threats including loss of system, access, or application control, compromise of administrative accounts or restriction of system functions, or the exposure of confidential information.

Medium-Risk findings are those that could potentially be used with other techniques to compromise accounts, data, or performance.

Low-Risk findings pose limited exposure to compromise or loss of data, and are typically attributed to configuration, and outdated patches or policies.

Informational findings pose little to no security exposure to compromise or loss of data which cover defense-in-depth and best-practice changes which we recommend are made to the application. Any informational findings for which the assessment team perceived a direct security risk, were also reported in the spirit of full disclosure but were considered to be out of scope of the engagement.

The findings represented in this report are listed by a risk rated short name (e.g., C1, H2, M3, L4, and I5) and finding title. Each finding may include if applicable: Title, Description, Impact, Reproduction (evidence necessary to reproduce findings), Recommended Remediation, and References.

INTRODUCTION

The Dark Crystal Protocol

Dark Crystal is a protocol and set of libraries and guidelines designed to make social management of cryptographic keys easy. It is particularly suited for peer-to-peer, decentralized applications, and is being integrated into the Briar messaging app.

At a high level, the protocol uses Shamir's Secret Sharing to split a secret key into multiple shards, and shares those shards among trusted peers of the secret owner. If the secret owner needs to recover their secret key, after confirming with a threshold of peers, they can request their shards and recombine them. The Dark Crystal website contains directions and considerations of how every step of this process can best be implemented into applications.

Project Scoping

On July 23rd, 2021, the assessment team began reviewing the Dark Crystal protocol for security vulnerabilities, specifically looking for protocol design flaws or security concerns in the Java implementation. The following code repositories were reviewed during this assessment:

sss
Third-party repository offering a C language implementation of Shamir's Secret Sharing. It is designed to be side-channel resistant and leverages operating system randomness.

dark-crystal-shamir-secret-sharing
This wrapper provides **Java Native Access** bindings to the **sss** C library, enabling its functions to be called from Java code.

dark-crystal-secret-sharing-wrapper
A higher-level wrapper for the secret sharing functionality, including improvements such as shard signing and x-coordinate obfuscation.

dark-crystal-key-backup-crypto-java
These cryptographic functions, including a partial implementation of the NaCL Networking and Cryptography library including the XSalsa20, Poly1305, and EdDSA algorithms, are leveraged by the rest of the system.

dark-crystal-key-backup-message-schemas-java
These protobuf schemas are for messages sent and received in the **Dark Crystal** protocol, along with JSON validators. The messages can contain shard shares, along with both required and optional metadata.

dark-crystal-key-backup-java
This high-level implementation of the protocol brings together all the other projects. It is currently a work in progress.

In addition, Dark Crystal documentation was consulted carefully to understand the project aims, particularly:

- Protocol Specification
- Threat Model
- Worked Example

Glossary

The following definitions from the **Dark Crystal** documentation are used throughout this report:

- Peer: an individual within a given social network
- Key: a peer's or secret-owner's encryption key
- Secret: the data to be backed up and potentially recovered
- Secret-owner: the peer to whom the data belongs
- Shard: a single encrypted share of the secret
- Custodian: a peer who holds a shard, generally a friend or trusted contact of the secret owner
- Threshold: the number of shards required to recover the secret

Scope

The aim of this assessment was to review the **Dark Crystal** implementation for security design flaws and security-relevant bugs.

The **Dark Crystal** threat model explicitly states that Dark Crystal "does not aim to protect against attacks that target a user's hardware or operating system". For example, the implementation does not address the fact that encryption keys may remain in process memory. Therefore, such platform attacks were outside the scope of this assessment. Similarly, the protocol is transport-agnostic, and it is up to the application to encrypt messages and make them indistinguishable from normal application traffic to mitigate passive surveillance and metadata leaks.

In addition, it is important to note that multiple protocol features discussed in the documentation are not yet implemented in the **dark-crystal-key-backup-java** project. The project is still a work in progress, and this report does not specifically identify security features which are currently missing but planned for the future. However, due to their importance to the protocol, it is worth noting a couple significant areas which are currently unimplemented.

Revocation of shards

Due to the possibility of a secret owner finding one or more custodians to be malicious, or wanting to make a previously shared secret unrecoverable, revocation of shards is a desirable capability to add to the base implementation. As discussed in the Dark Crystal documentation, this would work by the secret owner requesting all benevolent custodians to delete their shards (or ephemeral keys in the case of an append-only log), aiming to make it impossible to meet the share's threshold.

Long-term persistence of shards

Custodians may lose shards, such as by losing their phones or uninstalling the application. A secret owner might not know that they can no longer reach the threshold to recover their secret, until they actually try to combine. One suggestion for the **Dark Crystal** protocol would be for peers to send heartbeat messages over a background channel, so a peer can periodically show the secret owner that they still hold their shard. This could provide feedback to the secret owner within the application, and so a secret owner could determine whether their sharded secret has become unrecoverable, at which point they could create a new share.

CRITICAL-RISK FINDINGS

No “Critical-Risk” findings were identified during the course of the engagement.

HIGH-RISK FINDINGS

No “High-Risk” findings were identified during the course of the engagement.

MEDIUM-RISK FINDINGS

No “Medium-Risk” findings were identified during the course of the engagement.

LOW-RISK FINDINGS

L1: Out-of-Date Library in Use

Description:

dark-crystal-key-backup-crypto-java uses the **SpongyCastle** library, which is obsolete. **SpongyCastle** is a repackaging of the popular **BouncyCastle** library, which implements cryptographic algorithms in Java.

Early versions of Android bundled an old, restricted version of **BouncyCastle** under the original classpath of the library. This led to the creation of **SpongyCastle** to provide developers access to the full suite of **BouncyCastle** algorithms without classpath clashes with the restricted version. However since Android 3.0, released in 2011, Android changed the classpath for the bundled limited **BouncyCastle** library, enabling application developers to install a complete **BouncyCastle** without conflicts. So for modern Android devices the **SpongyCastle** workaround is no longer necessary, and the library is now unmaintained.

Impact:

The version of the **SpongyCastle** library in use is 1.58, which according to the GitHub releases page is the latest, published in 2017. This trails far behind the current version of **BouncyCastle**, 1.69. Several vulnerabilities in **BouncyCastle** have been published and fixed in the intervening years, while **SpongyCastle** has not received any fixes since 2017.

The assessment team was unable to identify any vulnerabilities affecting the specific classes and methods used in **Dark Crystal**. However if new vulnerabilities are found in **BouncyCastle**, an attacker could use these to focus exploit attempts on **Dark Crystal**, knowing that applications using **SpongyCastle** do not receive updates to their cryptographic algorithm library.

Reproduction:

Lines 7-13 of **build.gradle** in **dark-crystal-key-backup-crypto-java** show that **com.madgag.spongycastle** is set to version **1.58.0.0**:

```
dependencies {
  implementation 'com.madgag.spongycastle:core:1.58.0.0'
  implementation 'net.i2p.crypto:eddsa:0.2.0'
  implementation 'org.whispersystems:curve25519-java:0.5.0'
  testImplementation 'org.junit.jupiter:junit-jupiter-api:5.3.1'
  testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.3.1'
}
```

Recommended Remediation:

The team suggests switching to the latest version of **BouncyCastle** and using its implementations of the **XSalsa20**, **Poly1305**, and **Blake2b** algorithms.

References:

Spongy Castle: is it obsolete?
Stack Overflow: How to add Bouncy Castle algorithm to Android

INFORMATIONAL FINDINGS

I1: Malicious Custodian Can Disrupt Recovery

Description:

The **Dark Crystal** Threat Model documentation discusses the possibility of one or more malicious custodians. A custodian might wish to learn a user secret, or to disrupt a secret owner's reconstruction of their secret. The assessment team noted several opportunities where the implementation could be improved to detect and report adversarial custodian behavior.

Impact:

A single malicious custodian, especially one involved in a large threshold share, could cause significant disruption for secret owners.

After a failure to combine shards, the current implementation does not pass detailed information about the failure reason back to the calling code. Even if a secret owner determines that one custodian is submitting corrupted data during the combining protocol, the secret owner must attempt combines using every threshold group of custodians minus one, until the malicious custodian is left out and the combine finally works. This is a best-case scenario; without detailed information from the library, after an unsuccessful combine a secret owner may assume their secret is lost forever.

Reproduction:

Rather than a publicly verifiable scheme, for several reasons **Dark Crystal** opted to use the classic variant of Shamir's Secret Sharing with improvements, including signed shards. Signed shards, created by a private key generated by the secret owner, mitigate the simple case where a malicious custodian corrupts their shard or the appended ciphertext, because the custodian must submit their signature with the shard. If the signature is missing or fails to verify, the share can be rejected by the secret owner.

However in the **verifyAndCombine()** method, lines 199-209 of the file **SecretSharingWrapper.java** in the **dark-crystal-secret-sharing-wrapper** project, the exception does not provide information about which share caused verification to fail:

```
public static byte[] verifyAndCombine(List<byte[]> signedShares, PublicKey publicKey)
         throws Exception {
   EdDSA edDSA = new EdDSA();
   List<byte[]> rawShares = new ArrayList<>();
   for (byte[] signedShare : signedShares) {
     if (!edDSA.verifyMessage(signedShare, publicKey))
       throw new GeneralSecurityException("Unable to verify share");
     rawShares.add(EdDSA.detachMessage(signedShare));
   }
   return combine(rawShares);
 }
```

The higher-level **receiveReply()** and **receiveForward()** methods of **dark-crystal-key-backup-java** in file **KeyBackup.java**, lines 197-210, which receive shards from custodians, do not perform signature verification, leaving it to **verifyAndCombine()**:

```
private boolean receiveReply(Reply reply) throws Exception {
   if (!reply.isInitialized()) throw new Exception("Reply badly formed");

   // Check which secret this reply is for
   byte[] rootId = reply.getRoot().toByteArray();
   byte[] shard = reply.getShard().toByteArray();
   OwnSecret ownSecret = getOrCreateSecretByRootId(rootId);
```

```
    // Check branch reference to 'close' request (getBranch)
    byte[] branch = reply.getBranch().toByteArray();
    ownSecret.closeRequest(branch);

    return ownSecret.addShard(shard);
  }
```

The "forward" protocol additionally must handle the secret owner needing to receive the signing public key from custodians, since the secret owner might have lost access to it. The current implementation in **KeyBackup.java**, lines 234-238, accepts whichever key is received from a custodian, and is therefore also susceptible to a malicious custodian submitting an incorrect key:

```
   if (forward.hasSignaturePublicKey()) {
      byte[] publicKeyBytes = forward.getSignaturePublicKey().toByteArray();
      PublicKey signaturePublicKey = edDSA.PublicKeyFromBytes(publicKeyBytes);
      ownSecret.setAuthorSigningPublicKey(signaturePublicKey);
    }
```

Recommended Remediation:

The assessment team recommends that shard verification occur at a higher level, in the **dark-crystal-key-backup-java receiveReply()** or **receiveForward()** function in file **KeyBackup.java**, so the identity of the custodian who sent a malformed shard can be displayed in the UI of the calling application. Additionally the team recommends that if multiple different signing keys are submitted to the secret owner in the forward protocol, the library should catch this case and report it to the calling application, which can then decide how to handle it.

References:

Dark Crystal – Choosing a threshold-based secret sharing implementation
A Simple Publicly Verifiable Secret Sharing Scheme and its Application to Electronic Voting

I2: Resource Exhaustion Under No/Weak Consent Custodian Model

Description:

The **Dark Crystal** Threat Model documentation discusses three possible models for consent when a custodian receives a shard from a secret owner:

1. No Consent: the custodian's software automatically stores the shard.
2. Weak Consent: the custodian's software automatically stores the shard but prompts the custodian for whether they would like to keep it.
3. Strong Consent: the custodian must confirm they would like to receive a shard before it is stored on their device.

The implementation itself does not make any assumptions about which form of consent is used. However, beyond the concerns raised in the documentation about the no/weak consent models, the implementation does not address resource exhaustion attacks.

Impact:

A malicious peer in a social network could send a large number of shards to prospective custodians. The ciphertexts can be of arbitrary size and are stored indefinitely according to the **Dark Crystal** protocol. Since shard messages are keyed on the root ID, a hash of share metadata, and because the library does not implement or suggest any rate-limiting measures, an attacker could exhaust the file system storage of a custodian — preventing them from receiving further shards — or cause the calling application to crash.

Reproduction:

Lines 97-102 of **KeyBackup.java** in **dark-crystal-key-backup-java** switch on the type of **Dark Crystal** message received:

```
 public void receiveDarkCrystalMessage(byte[] message, byte[] authorEncryptionPublicKey, byte[]
authorSigningPublicKey) throws Exception {
    DarkCrystalMessage.DarkCrystal darkCrystalMessage = DarkCrystalMessage.DarkCrystal.parseFrom(message);
    switch (darkCrystalMessage.getMsgCase().name()) {
      case "SHARD":
        receiveShard(darkCrystalMessage.getShard(), authorEncryptionPublicKey, authorSigningPublicKey);
        break;
```

Lines 171-175 receive shards with the secret owner's encryption and signing keys, create a **ReceivedShard** object, and store it in a **receivedShards** list:

```
 private void receiveShard(Shard shardMessage, byte[] authorEncryptionPublicKey, byte[] authorSigningPublicKey)
throws Exception {
    if (!shardMessage.isInitialized()) throw new Exception("Shard badly formed");
    ReceivedShard receivedShard = new ReceivedShard(shardMessage, authorEncryptionPublicKey,
authorSigningPublicKey);
    receivedShards.put(toHexString(receivedShard.getRootId()), receivedShard);
  }
```

Recommended Remediation:

The assessment team recommends considering limiting the size of data received and creating separate shard storage with limited size for each individual peer. Then a peer who performed this attack could only exhaust their own shard storage allocation. If this finding is addressed in the application implementation rather than in the **Dark Crystal** library, the team recommends mentioning this consideration in the documentation. Alternatively encouraging Strong Consent as the preferred option for custodian consent would alleviate the risk.

References:

CWE-400: Uncontrolled Resource Consumption

I3: Lack of Validation in Share Size May Allow Buffer Overrun

Description:

The **dark-crystal-shamir-secret-sharing** project implements a **Java Native Access (JNA)** interface to the C-language dsprenkels**sss** library. This enables **sss** library methods to be called without having to rewrite the entire algorithm in Java. It also offers an opportunity to mitigate memory safety issues inherent to the C language using input length verification. This enables providing a library interface which is less prone to accidental misuse by developers.

However, the project contains methods which are vulnerable to buffer over-reads when called incorrectly. Buffer over-reads in the context of memory-unsafe languages such as C are when memory is accessed beyond the boundary of a fixed-length buffer. A **Secrets** class goes some way toward preventing misuse, but its use is not enforced.

Impact:

A developer using the **dark-crystal-shamir-secret-sharing JNA** library could potentially write code where the data stored in a secret would include not just the provided secret, but also additional bytes located in contiguous heap memory of the **libsss.so** library.

The risk is minimized because the **Dark Crystal** documentation encourages developers to use the higher-level **dark-crystal-shamir-secret-sharing-wrapper** rather than directly using the **JNA** interface. Further, the additional bytes are unlikely to contain significant data such as a previous secret beyond the lifetime of the process. Nevertheless, it is preferable to eliminate out-of-bounds-reads which do not occur in typical Java code.

Reproduction:

In the C-language **sss** library, within the **sss_create_shares()** function in file **sss.c**, line 94, 64 bytes of provided secret data is copied to a buffer to be encrypted:

```
/* AEAD encrypt the data with the key */
memcpy(&m[crypto_secretbox_ZEROBYTES], data, sss_MLEN);
```

In the **JNA** library, the **SSS.createShares()** method provides the interface to this low-level function. **SSS.createShares()** accepts a byte array for the secret as its first argument. If the method is called with a secret less than 64 bytes, then the C **memcpy()** function will copy past the end of the secret, encrypting additional memory into the secret, as demonstrated by the following test:

```
package org.magmacollective.libsss;

import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;

import java.util.Arrays;
import java.util.List;

public class TestEncodeDecode
{

        @Before
        public void prepare()
        {
                NativeUtil.configureJnaLibraryPath();
        }

        @Test
        public void createShares()
        {
                byte[] secret = "a secret".getBytes();
                List<byte[]> shares = SSS.createShares(secret, 3, 2);
                byte[] reconstructed = SSS.combineShares(shares);
                System.out.println(Arrays.toString(reconstructed));
                 // Output: [97, 32, 115, 101, 99, 114, 101, 116, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
37, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -96, 24, 91, -36, -69, 127, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -127, 0,
0, 0, 0, 0, 0, 0]

        }
}
```

Recommended Remediation:

The **Secrets.create()** method sets a minimum length of 64 bytes for the secret and pads it with zeroes. However **SSS.createShares()** and **SSS.createKeyshares()** do not require use of this helper method. The assessment team recommends that each method verify it receives a 64-byte or 32-byte array, respectively, before proceeding to call C functions.

References:

Buffer over-read