Cinema time!

Andrey Labunets

Nikita Tarakanov

Hexacon

15th of October 2022

Paris, France

#WhoWeAre

- Nikita Tarakanov is an independent security researcher. He has worked as a security researcher in Positive Technologies, Vupen Security, Intel corporation and Huawei. He likes writing exploits, especially for OS kernels. He won the PHDays Hack2Own contest in 2011 and 2012. He has published a few papers about kernel mode drivers and their exploitation. He is currently engaged in reverse engineering research and vulnerability search automation.
- Andrey Labunets is a security researcher with more than a decade of experience in vulnerability research and reverse engineering.

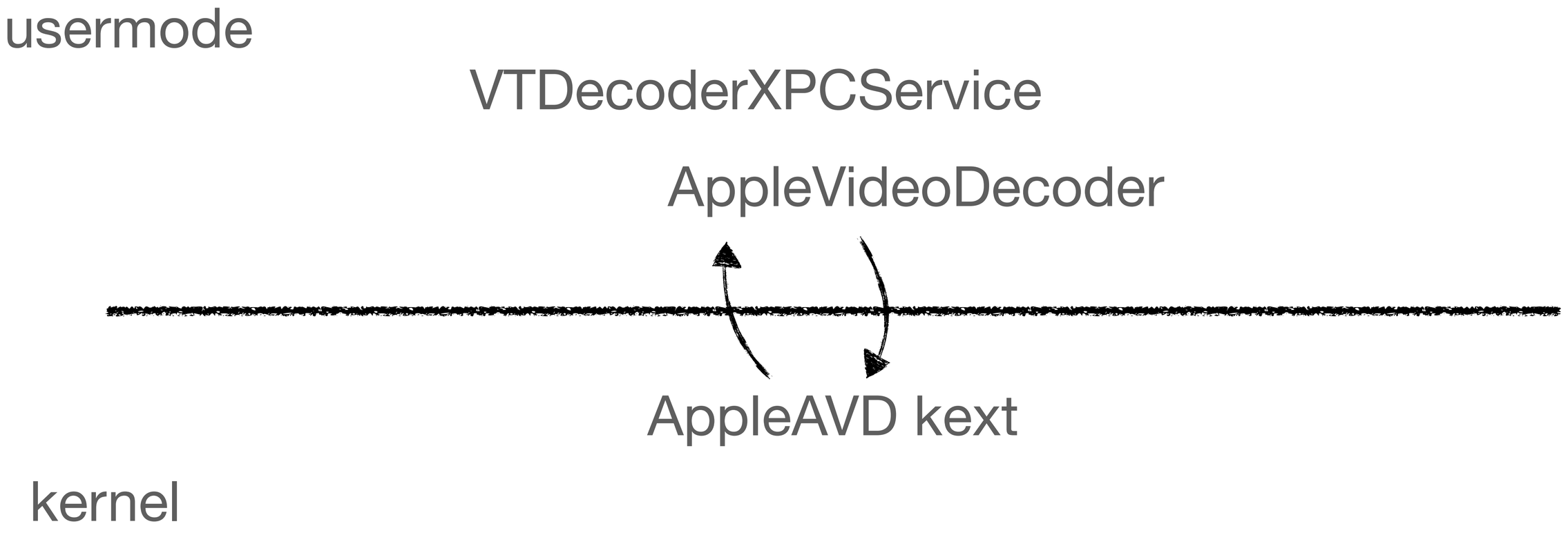
Agenda

- Video decoding subsystem overview
- AppleAVD internals
- AppleAVD attack surface
- Fuzzing approach and code analysis
- Results
- Previously disclosed vulnerabilities and exploitation
- Discussion
- Q&A

Video decoding subsystem overview

Video decoding subsystem

macOS Monterey (M1)



Video decoding subsystem

- Out of scope today - hardware components
- Main focus is AppleAVD kext internals on

  macOS Monterey

- You can find some info on AVD hardware here:
  - https://twitter.com/rqou_/status/1577967077955993600
  - https://github.com/rqou/m1-avd-reverse-engineering/blob/main/avd_emu.py

AppleAVD internals

AppleAVD internals

Codebase overview

- AppleAVD - one of the largest kexts in macOS
  - **~120 KLOC** in IDA decompiler
- Large part of this codebase are actual decoders, which process parts of media input in kernel space

```
  if ( (_DWORD)a3 == 301 )
  {
    createLilyDLghDecoder(this);
  }
  else
  {
    if ( (_DWORD)a3 != 308 )
      goto LABEL_39;
    createDahliaLghDecoder(this);
  }
}
else
{
  switch ( (_DWORD)a3 )
  {
    case 0x13C:
      createRadishLghDecoder(this);
      break;
    case 0x144:
      v10 = createClaryLghDecoder(this);
      break;
    case 0x190:
      v10 = createIxoraLghDecoder(this);
      break;
    default:
```

AppleAVD internals

Codebase overview

- AppleAVD - one of the largest kexts in macOS
  - **~120 KLOC** in IDA decompiler
- Large part of this codebase are actual decoders, which process parts of media input in kernel space

```
{
  createCloverLghDecoder(this);
  goto LABEL_4;
}
if ( (int)a3 <= 300 )
{
  switch ( (int)a3 )
  {
    case 20:
      v10 = createSalviaA0LghDecoder(this);
      goto LABEL_4;
    case 21:
      v10 = createSalviaLghDecoder(this);
      goto LABEL_4;
    case 22:
    case 23:
    case 24:
    case 25:
    case 27:
      goto LABEL_39;
    case 26:
      createViolaLghDecoder(this);
      goto LABEL_4;
    case 28:
      v10 = createLotusLghDecoder(this);
      goto LABEL_4;
    default:
      if ( (_DWORD)a3 != 300 )
        goto LABEL_39;
      createLilyCLghDecoder(this);
      break;
  }
```

AppleAVD internals

Entry points (external methods)

- Accessed through AppleAVDUserClient
- Most interesting external methods:
  - AppleAVDUserClient::createDecoder
  - AppleAVDUserClient::decodeFrameFig
- Data transfer between user / kernel.
  (media data, NAL units, etc)
  - IOSurface

AppleAVD internals

Entry points (external methods)

- This set of external methods covers most of the kext functionality:
  - AppleAVDUserClient::setCallback
  - AppleAVDUserClient::createDecoder
  - AppleAVDUserClient::setCryptSession
  - AppleAVDUserClient::decodeFrameFig

AppleAVD internals

createDecoder

- Sets up one of 3 decoders of our choice

```
codec_type = (unsigned __int8)in->codec_type;
*(_DWORD *)&this->m_AppleAVDUserClient.codec_type = codec_type;
*(_DWORD *)&this->m_AppleAVDUserClient.field7892_0x1ed4 = in->pad0x24;
this->m_AppleAVDUserClient.decoder = 0LL;
switch ( codec_type )
{
  case 1:
    if ( this->m_AppleAVDUserClient.deviceType < 0xDu )
      goto LABEL_41;
    v21 = (CAVDAvcDecoder *)operator new(0x8642B0uLL);
    decoder = CAVDAvcDecoder::CAVDAvcDecoder(
                v21,
                this,
                (unsigned int)this->m_AppleAVDUserClient.deviceType,
                in->pad0x1B != 0);
    break;
  case 3:
    v22 = (CAVDLghDecoder *)operator new(0x65C0uLL);
    decoder = (CAVDAvcDecoder *)CAVDLghDecoder::CAVDLghDecoder(
                                  v22,
                                  this,
                                  this->m_AppleAVDUserClient.deviceType,
                                  in->pad0x1B != 0);
    break;
  case 2:
    v19 = (CAVDHevcDecoder *)operator new(0x21CBB8uLL);
    CAVDHevcDecoder::CAVDHevcDecoder(v19, this, this->m_AppleAVDUserClient.deviceType, in->pad0x1B != 0);
    break;
  default:
```

```
struct _sAppleAVDCreateDecoderIn
{
  _DWORD width;
  _DWORD height;
  _DWORD nal_buf_len;
  _DWORD val0xc;
  _BYTE val0x10[9];
  _BYTE codec_type;
  _BYTE pad0x1A;
  _BYTE pad0x1B;
  _BYTE pad0x1C;
  _BYTE pad0x1D;
  _BYTE ichat_usage_mode;
  _BYTE pad0x1F;
  _BYTE memCacheMode;
  _BYTE val0x21;
  _BYTE val0x22;
  _BYTE val0x23;
  _DWORD pad0x24;
  _DWORD pad0x28;
  _DWORD pad0x2c;
  _DWORD val0x30;
  _DWORD wsk[33];
  _DWORD surface_id;
  _DWORD decrypt_mode;
  _DWORD sleepWakeTransitionTimeout;
  _BYTE pmgrRequestTimeout[20];
};
```

AppleAVD internals
decodeFrameFig

- Processes plaintext frames…

```
process_plaintext_frame:

    _reference_index = in->reference_index;
    *&this->m_AppleAVDUserClient._reference_index = _reference_index;
    deviceType = this->m_AppleAVDUserClient.deviceType;
    if ( (deviceType == 28 || deviceType >= 0x12F) && (this->m_AppleAVDUserClient.field
    {
      (this->m_AppleAVDUserClient.decoder->__vftable->VASetParams)(
        this->m_AppleAVDUserClient.decoder,
        43LL,
        *&this->m_AppleAVDUserClient.field7296_0x1c80 + ((in->framenumber << 9) & 0x1E0
      _reference_index = in->reference_index;
    }

    kernel_debug(0x2B680050u, _reference_index, in->display_index, in->index_target2, 0

    res = (this->m_AppleAVDUserClient.decoder->__vftable->VADecodeFrame)(
            this->m_AppleAVDUserClient.decoder,
            buf,
            in->dataLength,
            in->framenumber,
            in->display_index,
            in->reference_index,
            in->index_target2,
            out + 3552);
```

```
/* 110 */
struct __attribute__((aligned(4))) _sAppleAVDDecodeFrameFigIn
{
  _QWORD mapPixBuf_address;
  _DWORD dataLength;
  _DWORD framenumber;
  _DWORD display_index;
  _DWORD reference_index;
  _DWORD index_target2;
  _DWORD decrypt_byte_offset;
  _DWORD allocSize;
  _BYTE VASetDisableSkipToIDR_val;
  _BYTE val0x25;
  _BYTE val0x26;
  _BYTE val0x27;
  _BYTE isEncrypted;
  _BYTE val0x29;
  _BYTE val0x2a;
  _BYTE val0x2b;
  _DWORD initialClearBytes;
```

AppleAVD internals

decodeFrameFig -> decodeFrameFigHelper_DecryptFrame

- Processes plaintext frames…
- …and DRM content
  - Likely it's FairPlay Streaming
  - FairPlay code obfuscation complicates some analysis

```
__int64 __fastcall AppleAVDUserClient::decodeFrameFigHelper_DecryptFrame(
{
  char *plaintext_buffer; // x1
  unsigned __int8 *buffer; // x2
  __int64 v6; // x0
  __int64 res; // x21
  AppleAVD *v8; // x0
  unsigned int dataLength; // w3
  __int64 contextID; // x4
  int v11; // w5

  plaintext_buffer = this->m_AppleAVDUserClient.plaintext_buffer;
  buffer = this->m_AppleAVDUserClient.buffer;
  switch ( *&this->m_AppleAVDUserClient.decrypt_mode )
  {
    case 0:
      v6 = AppleAVD::decryptFrame(
             this->m_AppleAVDUserClient.appleAVD,
             plaintext_buffer,
             buffer,
             in->dataLength,
             *&this->m_AppleAVDUserClient.contextID);
      goto LABEL_10;
    case 1:
      v8 = this->m_AppleAVDUserClient.appleAVD;
      dataLength = in->dataLength;
      contextID = *&this->m_AppleAVDUserClient.contextID;
```

AppleAVD internals

setCryptSession

- Allocates data buffers for (decrypted?) data, initializes (session?) parameters
- From AppleVideoDecoder:

```
67  struct _sAppleAVDSetCryptSessionIn
68  {
69    _QWORD crypt_ref;
70    uint8_t decrypt_mode;
71    uint8_t pad0x9;
72    uint8_t pad0xa;
73    uint8_t pad0xb;
74    uint8_t pad0xc;
75    uint8_t pad0xd;
76    uint8_t pad0xe;
77    uint8_t pad0xf;
78    _QWORD pad0x10;
79  };
80
```

```
241
242         // set_parameter later calls AppleAVD setCryptsession external method
243         v39 = set_parameter(*(unknown2 **)&v11[80].__opaque[40], 22, (unsigned __int8 *)v116.value);
244         if ( (_DWORD)v39 )
245         {
246           v33 = v39;
247           syslog_DARWIN_EXTSN(
248             3LL,
249             "AppleAVD: AppleAVD_HEVCVideoDecoder: frame# %d, Could not set kAppleAVDSetCryptRef, err %d\n");
250           goto LABEL_59;
251         }
0004D5E8 AppleAVDWrapperHEVCDecoderDecodeFrame:235 (55E8)
```

AppleAVD attack surface

AppleAVD attack surface

- In the past, vulnerabilities in AppleAVD have been both found by researchers and exploited in-the-wild:
  - CVE-2022-22675 - *in-the-wild*
    - https://googleprojectzero.github.io/0days-in-the-wild/0day-RCAs/2022/CVE-2022-22675.html
  - CVE-2022-22674 - *in-the-wild*
  - CVE-2018-4384
    - https://bugs.chromium.org/p/project-zero/issues/detail?id=1641

- *Which AppleAVD attack vectors are most likely to be actively exploited?*

AppleAVD attack surface

- AppleAVD media processing is performed with **decoders**, wrapped in **AppleAVD logic**:
  - AppleAVD logic is AppleAVDUserClient, AppleAVD, and other kext classes
  - Decoders and AppleAVD logic are mostly independent of each other
  - This is a very simplified way to approach kext's large codebase
- Decoders might be easier to attack remotely
- Attacking AppleAVD logic might require more control over AppleAVD objects, achieved, for example, with specific external method arguments in local privilege escalation (LPE) scenarios

AppleAVD attack surface

- To explore the most straightforward remote attack vectors, we extracted decoders from the kext, rebuilt them, and tested directly with a coverage-guided fuzzer
- To investigate the outer AppleAVD logic, we reconstructed the logic of external methods and manually reviewed object initialization, memory operations, and interaction between components
- We left all possible firmware and hardware vectors out of scope

Fuzzing approach and code analysis

Fuzzing decoders

- AppleAVD decoders process media data, such as Network Abstraction Layer (NAL) units, parameter sets (SPS, PPS)
- This code is implemented in CAVDAvcDecoder, CAVDLghDecoder, CAVDHevcDecoder
- Parsing is done inside virtual VAStartDecode and VADecodeFrame

Fuzzing decoders

Seed corpus generation

- First, we fuzzed ffmpeg with a small set of publicly available templates
- ffmpeg fuzzer generated a sufficiently large seed corpus for AppleAVD
- AppleAVD expects NAL units in a slightly different format, so we preprocessed the resulting seed corpus for AppleAVD

Fuzzing decoders

Target code setup and fuzzing

- We built the target from a pseudocode extracted from IDA decompiler
  - In our experiments with AppleAVD and other macOS subsystems, the control flow does not differ from the original machine code control flow
- We wrote a tiny interface to handle IDA types, ARM intrinsics, and reimplement selected parts of macOS library code

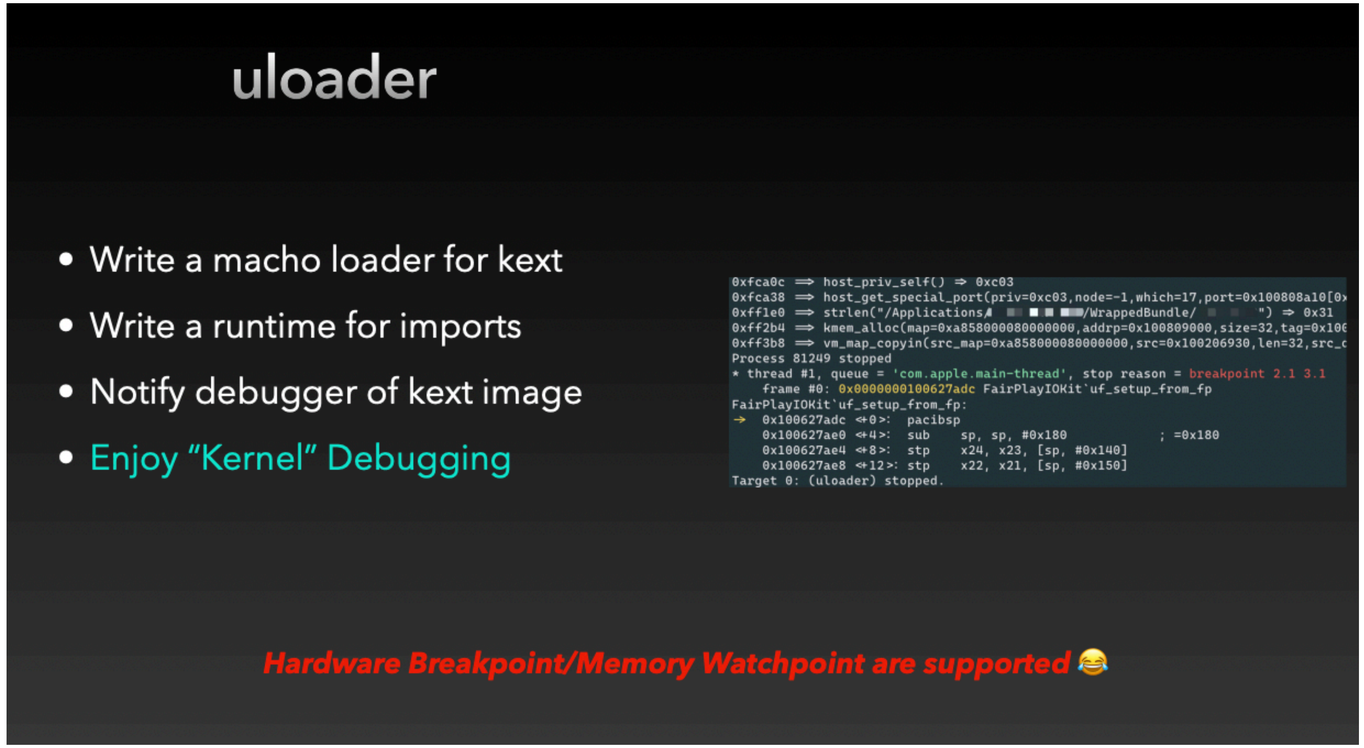
- Results: fuzzing ~3KLOC of CAVDAVCDecoder+AVC_RBSP with AFL++ resulted in a single unexploitable crash (an artifact of fuzzing setup) and 96% coverage

Fuzzing decoders

Road not taken - alternative fuzzing setup

- Another approach by Junzhi Lu, Xindi Wang, Ju Zhuto* runs kexts in user mode with a custom macho loader for debugging, but could be useful for fuzzing too.

- Our approach with code extraction gives source code-level flexibility to fuzz selected code paths with debug symbols.



*https://github.com/pwn0rz/fairplay_research

Manual analysis

- Analysis of 3 external methods revealed an issue with inconsistent checks:
  - CVE-2022-46694 fixed in iOS 16.2 and iPadOS 16.2
  - https://support.apple.com/en-us/HT213530

```
275
276 bufSize = (unsigned int)(this->m_AppleAVDUserClient.width * this->m_AppleAVDUserClient.height);
277
278
279 // err if width * height > 0x3FFFFFFFC0000000 = 4611686017353646080
280 if ( (((unsigned int)this->m_AppleAVDUserClient.width
281       * (unsigned __int64)(unsigned int)this->m_AppleAVDUserClient.height) & 0xFFFFFFFF00000000LL) != 0
282   || ((4 * bufSize) & 0xFFFFFFFF00000000LL) != 0 )
283 {
284   res = 3758097085LL;
285   _os_log_internal(
286     &dword_0,
287     (os_log_t)&_os_log_default,
288     OS_LOG_TYPE_DEFAULT,
289     "AppleAVD: bufSize overflow in %s %d",
290     "setCryptSession",
291     2537LL);
292 }
293 else
294 {                                               // buf_size is bounded
295   LODWORD(bufSize) = 4 * bufSize;
296   res = 3758097085LL;
297   if ( (unsigned int)bufSize >= 100000000 )     // bufsize between [100000000, 4294967296]
298                                                 // will be capped at 100000000.
299                                                 //
300                                                 // However, the original width and height will
301                                                 // later be used to determine buffer length
302                                                 // when this buffer is accessed
303     LODWORD(bufSize) = 100000000;
304   if ( (unsigned int)bufSize <= 1048576 )
305     bufSize = 0x100000LL;
306   else
307     bufSize = (unsigned int)bufSize;
308   allocMemIn.size = bufSize;
309   allocMemIn.val0x14 = 1;
310   if ( (unsigned int)AppleAVDUserClient::allocateMemory(
311                        this,
312                        &allocMemIn,
313                        &kVASetDecryptSessionData_allocMemOut,
314                        1uLL,
315                        0) )
316   {
000CA21C __ZN18AppleAVDUserClient15setCryptSessionEP27_sAppleAVDSetCryptSessionInP28_sAppleAVDSetC
```

AppleAVD analysis: results

- Fuzzed one of three decoders - 1 non-exploitable crash (fuzzing setup at fault), not an issue
- Reviewed control flow and interaction between 3 external methods - 1 finding (CVE-2022-46694)

- Limited results suggest AppleAVD logic (vs.decoders) is more error-prone
- AppleAVD can still be exploited via relatively simple memory corruption bugs
- A subset of decoder bugs leading to subtle conditions and data-only attacks was left out of scope, but it is an interesting future direction to explore

Previously discovered vulnerabilities

- CVE-2018-4348 - memory corruption
- CVE-2020-9958 - out-of-bound write
- **CVE-2022-22675 - overflow in parseHRD**
- **CVE-2022-32788 - overflow in parseSliceHeader**

Previously disclosed vulnerabilities and exploitation

Overflow in parseHRD

- cpb_cnt_minus1 = Read_byte_from_stream();
- *HRD_data_in_spsList = cpb_cnt_minus1;
- HRD_arrays = HRD_data_in_spsList + 0x104;
- Index = 0
- do {
- *(_DWORD *)&HRD_arrays[4 * index - 0x100] = Read_dword_from_stream();
- *(_DWORD *)&HRD_arrays[4 * index - 0x80] = Read_dword_from_stream();
- HRD_arrays[index] = = Read_byte_from_stream();
- } while (index++ < *HRD_data_in_spsList);

Overflow in parseHRD

- Size of array is 0x20 elements we can copy up-to 0x100 elements
- We can overflow adjacent element in spsList array
- We can overflow adjacent memory to spsList array (first element in ppsList)

Overflow in parseSliceHeaders

```
counter_oob = 0;
while ( 1 ){
    *((_BYTE *)SliceHeaderBuffer + counter_oob + 47) = Read_byte_from_stream();
    *((_DWORD *)SliceHeaderBuffer + counter_oob + 55) = Read_dword_from_stream();
    *((_DWORD *)SliceHeaderBuffer + counter_oob + 21) = Read_dword_from_stream();
    counter_oob++;
    If ( *((_BYTE *)SliceHeaderBuffer + counter_oob + 47) == 0x3 ) {
        break;
    }
}
```

Overflow in parseSliceHeader

- Size of SliceHeader buffer is 0x480 located in CAVDAvcDecoder object
- We can overflow adjacent fields in CAVDAvcDecoder object
- CAVDAvcDecoder object is huge (0x8642B0 byes) in KHEAP_KEXT
- We can spray CAVDAvcDecoder objects and smash pointers in it
- Problem is that we have to win race between using vtable (PAC) and pointers to other objects

Discussion

Discussion

- AppleAVD runs about ~100KLOC of parsers in kernel on all incoming media.
  - Can this functionality be moved to user land instead or isolated?

- Some input validation is spread across multiple external methods, processing is partially performed by an obfuscated FairplayIOKit.
  - Can we even inspect the security of Apple media pipeline in any meaningful way?

Discussion

- Thanks to Max Dmitriev (I_Greek) and Berk Cem Göksel (@berkcgoksel)!

- Questions?