New attack Surface: Use just one WebAudio Vulnerability to rule the Safari

1. Background introduction

In the past, Safari vulnerability researchers often focused on the DOM or JS engine, but some system libraries used by Safari, such as audio, video, font, etc., haven't received enough attention. There are few successful cases using vulnerabilities found in these modules to break Safari. Due to the built-in heap isolation mechanism of Safari, the heap used by these system libraries is not the same as the heap where the DOM objects and JS objects are located. As a result, the out-of-bounds writing vulnerabilities in these modules make it extremely difficult to overwrite some key JS objects. These vulnerabilities are difficult to exploit alone without the coordination of an info leak. But I found that there is a bug that can overwrite JS objects in a clever way, bypassing Safari's heap isolation, ASLR and other defense mechanisms, and finally achieve arbitrary code execution. I demonstrated the attack at the Tianfu Cup International Cyber Security Competition hosted by the Chengdu Municipal Government of China, and successfully pwned Safari with only one shot.

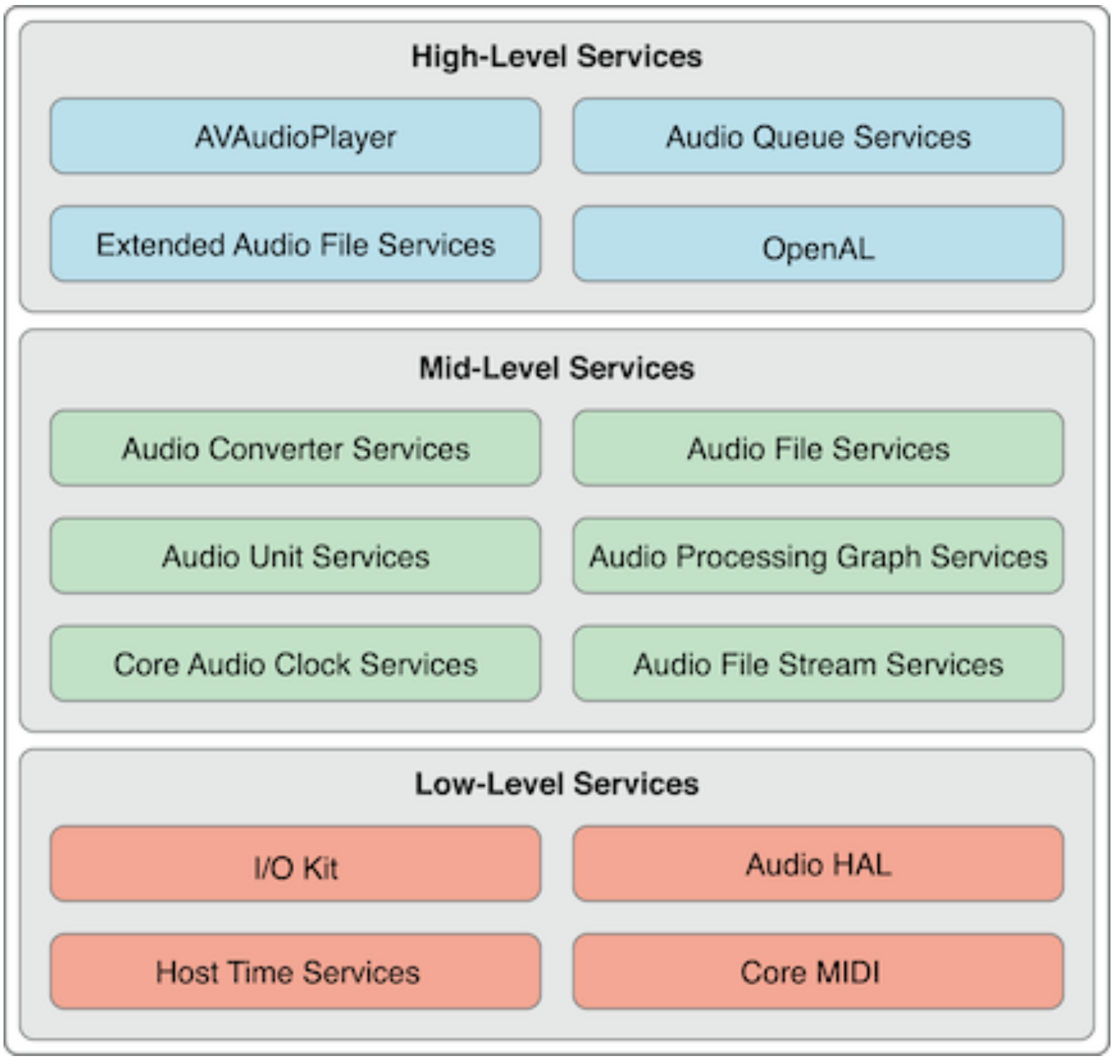
2. WebAudio bug hunting

2.1 WebAudio module introduction

The WebAudio module is a library and provides rich API for playing, encoding, decoding, and transcoding audio on macOS/iOS. We can first take a look at a classic audio playback process. Take MP3 as an example:

1. Read an MP3 file from disk
2. Analyze information such as sample rate, bit rate, duration, etc., and separate audio frames in MP3
3. Decode the separated audio frame to get PCM data
4. Perform sound effect processing on PCM data (equalizer, reverberator, etc., optional)
5. Decode PCM data into the audio signal
6. Give the audio signal to the hardware for playback

We can see that steps 2, 3, and 5 are all analyzing a piece of the input buffer, which is very suitable for fuzzing.
Apple provides APIs at different levels for these functions. Some frameworks in Low-Level Services are mostly libraries that deal with hardware. We don't care about this. We are mainly concerned about the frameworks in Mid-Level and High-Level, to see which framework is more appropriate to build our harness.



The following is a functional description of the middle and high-level interfaces: (can be circled in the figure for easy display)

Audio File Services: Read basic information such as sampling rate, bit rate, duration, etc. and separate audio frames, let’s call these processes "audio analysis", complete the 2nd step in the playback process;

Audio File Stream Services: Similar to Audio File Services, but mainly for streaming media playback, 2nd step of the playback process;

Audio Converter services: Audio data conversion, 3rd step;

Extended Audio File Services: a combination of Audio File Services and Audio Converter services;

AVAudioPlayer/AVPlayer (AVFoundation): Advanced interface, which can complete the entire audio playback process (including local file and network stream playback, except for step 4);

If we choose an advanced service, such as AVPlayer, we need to use a graphical interface, which is not convenient for Fuzzing. Audio File Services, Audio File Stream Services, Audio Converter Services, Extended Audio File Services are all good choices. For the 2nd step, compared with Audio File Services, Audio File Stream Services can read files from the memory which makes our fuzzing much more efficient, so we finally selected Audio File Stream Services. For 3rd step, the audio decoding process, I chose the relatively easy-to-use Extended Audio File Services for Fuzzing.

2.2 Previous work

Audio is not a new attack surface. As early as 2017, riusksk achieved several CVEs in Audio. In 2019, he achieved 3 more. But other than that, indeed less attention has been paid to vulnerabilities in Audio. Before 2019, browser bug hunting was focused on modules such as JS and wasm.

2.3 Binary Bug Hunting Technology under MacOS

Then let me briefly introduce the binary bug hunting technology I used on macOS. This year project zero disclosed how they exploited the vulnerabilities in ImageIO. They wrote an instrumentation tool called TrapFuzz for it. This tool is very suitable for library fuzzing, and the fuzz efficiency is pretty high.

The harness needs to be carefully designed. The audio parsing step may be relatively simple. It can be divided into three steps, open, analysis and close. But for Audio decoding, various options need to be set for the output PCM format, such as sampling rate, number of channels per frame, number of frames per Packet, etc. Since we want to increase the probability of reproducing our crashes in Safari, we should refer to parameter settings in it. Another important step is to collect as many seeds as possible. I have crawled hundreds of thousands of seeds from the Internet, and then we can start a enjoyable Fuzzing.

2.4 Results achieved

At present, we have obtained 16 CVEs in total, which includes 9 out-of-bounds reads and 7 out-of-bounds writes , and there are still tens of vulnerabilities in progress that have been submitted to Apple. CVE-2021-1747 is the vulnerability I used in this Tianfu Cup. I used this vulnerability to achieve arbitrary code execution in Safari.

3. Safari vulnerability exploitation

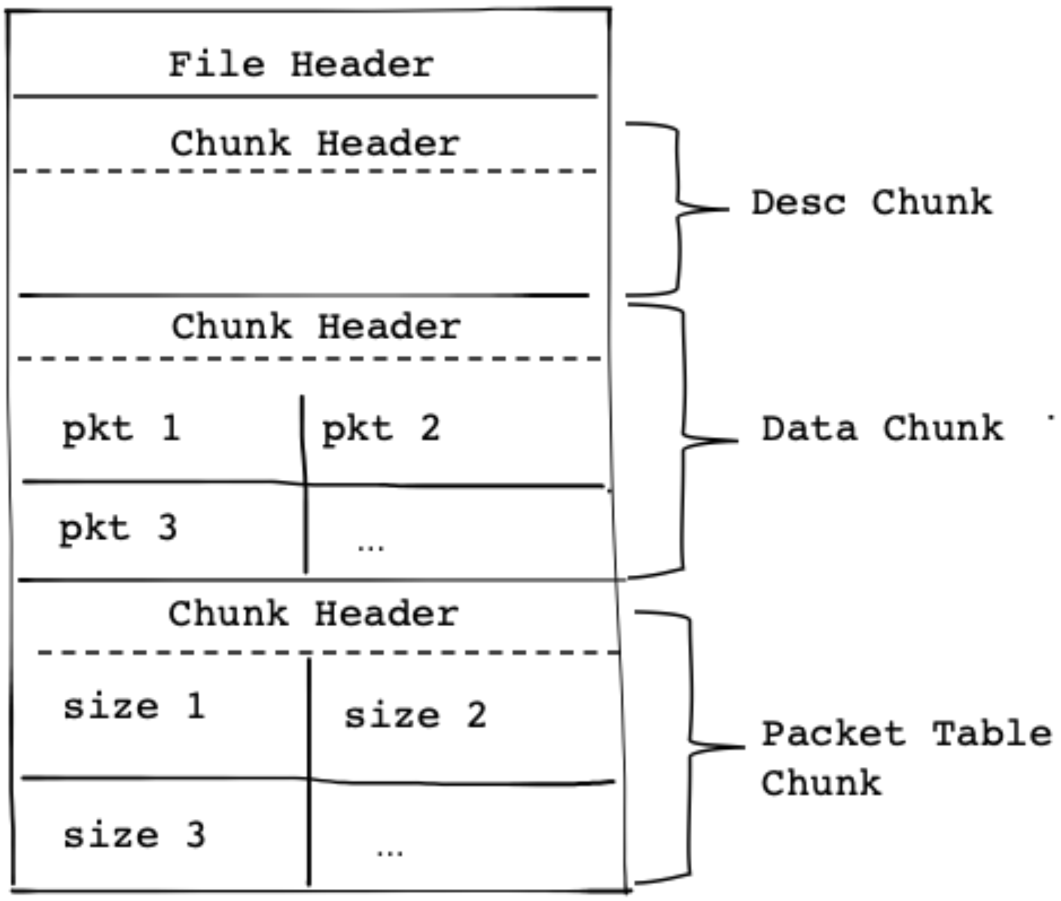
Let me introduce the cause of the vulnerability I used on the Tianfu Cup and the exploitation details.

3.1 Crash Analysis

The vulnerability exists in the `ACOpusDecoder::AppendInputData` function of the WebAudio module, which will cause out-of-bounds write when parsing CAF files. At position one there is a code similar to the bounds checking, but it does not take effect, and at position two the `memcpy` function is called, causing out-of-bounds write.

```
__int64 __fastcall ACOpusDecoder::AppendInputData(ACOpusDecoder
*this, const void *a2, unsigned int *a3, unsigned int *a4, const
AudioStreamPacketDescription *a5)
{
  ...

  if ( a5 )
  {
    v8 = a5->mDataByteSize;
    if ( !a5->mDataByteSize || !*a4 || (v9 = a5->mStartOffset,
(a5->mStartOffset + v8) > *a3) || this->buf_size ) // (1). bound
checking does not take effect here.
    {
      result = 0LL;
      if ( !v8 )
      {
        this->buf_size = 0;
LABEL_19:
        v13 = 1;
        v12 = 1;
        goto LABEL_20;
      }
      goto LABEL_16;
    }
    if ( v9 >= 0 )
    {
      memcpy(this->buf, a2 + v9, v8);   // (2) . where out-of-bounds
write
      v14 = a5->mDataByteSize;
      this->buf_size = v14;
      result = (LODWORD(a5->mStartOffset) + v14);
      goto LABEL_19;
    }
    ...
}
```



Let me briefly introduce the CAF file format. Here I draw a simplified version of the CAF file format. The CAF file starts with the File Header, and then is composed of various types of Chunk. Each Chunk has a Chunk Header, which records the size of the Chunk. Desc Chunk mainly stores some metadata of the file, Data Chunk stores all the Packets, and Packet Table Chunk records the size of each Packet. During parsing, the Packet Table Chunk will be read first to obtain the size of each Packet, and then go to Data Chunk to read the corresponding Packet.

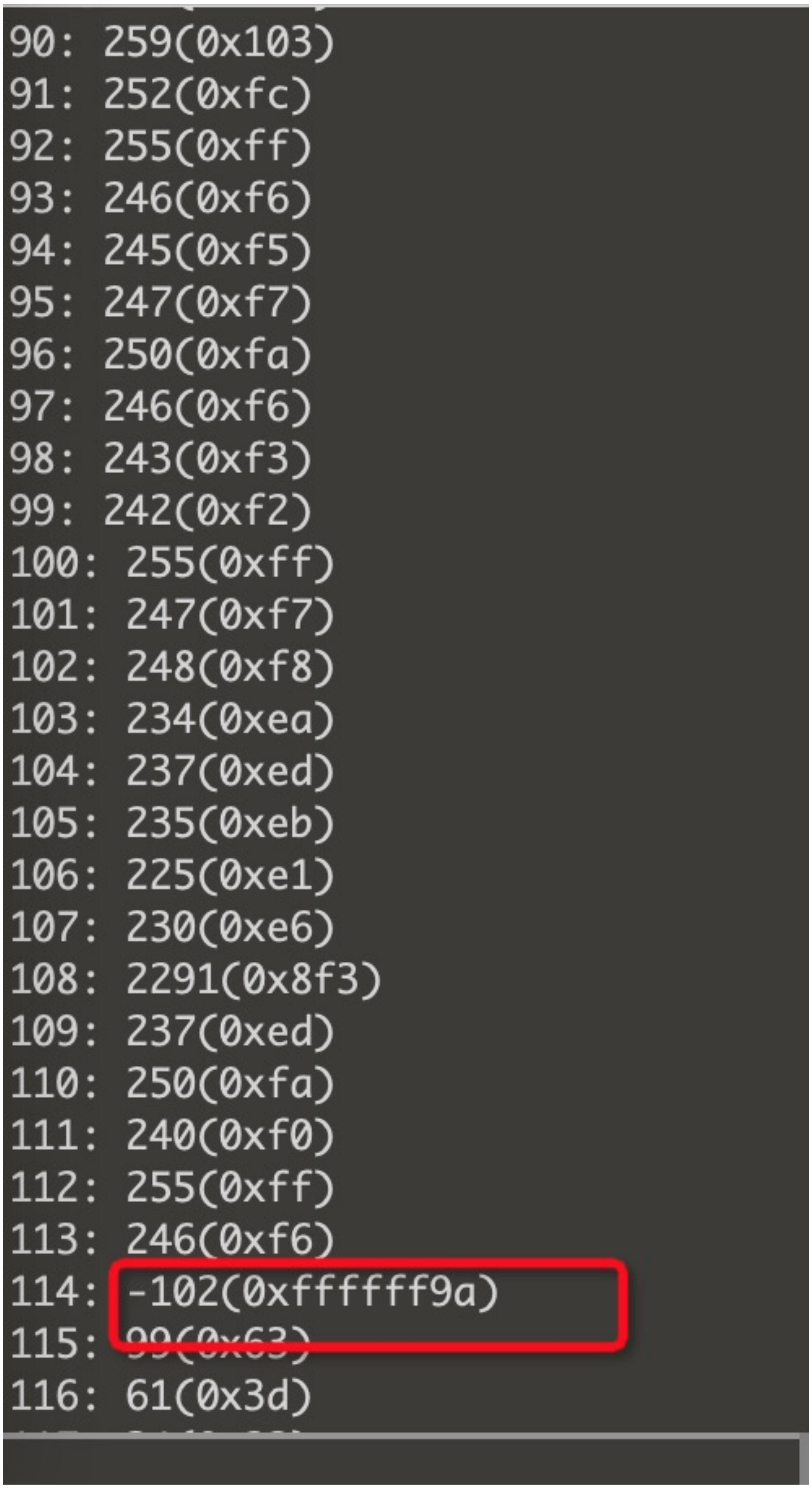
In order to analyze this vulnerability, I specially wrote a 010 Editor template to parse the CAF file. This is part of template code.

```
BigEndian();
struct CAFAudioFormat {
    double mSampleRate;
    uint32 mFormatID;
    uint32 mFormatFlags;
    uint32 mBytesPerPacket;
    uint32 mFramesPerPacket;
    uint32 mChannelsPerFrame;
    uint32 mBitsPerChannel;
};

…
struct File {
    …
    struct CAFFileHeader {
        uint32  mFileType;
        uint16  mFileVersion;
        uint16  mFileFlags;
    } cafFileHdr;
…

} file;
```

Then we analyze the CAF file that caused the crash, and run it with the template file of 010 editor, you can see the following output:

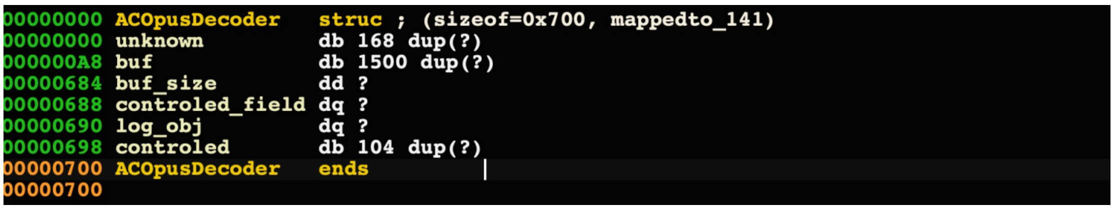


```
90: 259(0x103)
91: 252(0xfc)
92: 255(0xff)
93: 246(0xf6)
94: 245(0xf5)
95: 247(0xf7)
96: 250(0xfa)
97: 246(0xf6)
98: 243(0xf3)
99: 242(0xf2)
100: 255(0xff)
101: 247(0xf7)
102: 248(0xf8)
103: 234(0xea)
104: 237(0xed)
105: 235(0xeb)
106: 225(0xe1)
107: 230(0xe6)
108: 2291(0x8f3)
109: 237(0xed)
110: 250(0xfa)
111: 240(0xf0)
112: 255(0xff)
113: 246(0xf6)
114: -102(0xffffff9a)
115: 99(0x63)
116: 61(0x3d)
```

The first column is the serial number of the packet, and the second column is the size of the packet(in both decimal and hexadecimal format). It can be seen that the size of the 114th packet is a negative number. It can be speculated that the program runs into a problem when processing packets with negative sizes. Because the code is too complicated and there was not much time before Tianfu Cup at that time, I did not conduct a detailed analysis of the cause of the vulnerability. Here I mainly share my exploitation process.

3.2 Turn out-of-bounds write into arbitrary address write

Here I first did a reverse analysis of the relevant code. The buffer written out of bounds exists in the structure called ACOpusDecoder. The fields of this structure are as follows:

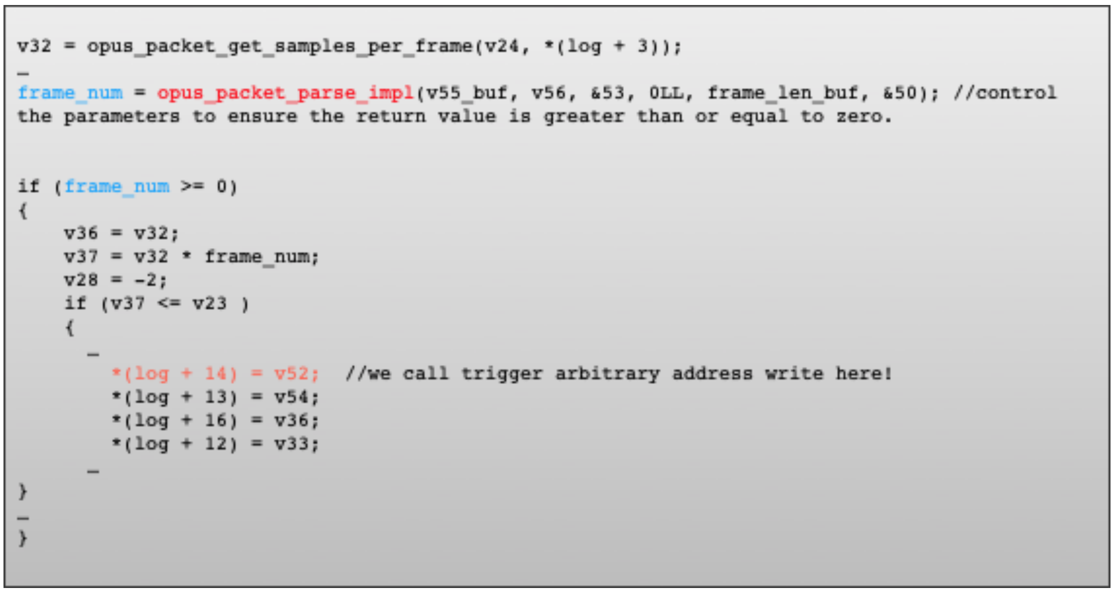


```
00000000 ACOpusDecoder   struc ; (sizeof=0x700, mappedto_141)
00000000 unknown         db 168 dup(?)
000000A8 buf             db 1500 dup(?)
00000684 buf_size        dd ?
00000688 controled_field dq ?
00000690 log_obj         dq ?
00000698 controled       db 104 dup(?)
00000700 ACOpusDecoder   ends
00000700
```

I first did a reverse analysis of the relevant code. The buffer written out of bounds exists in the structure called ACOpusDecoder. The fields of this structure are as follows:
The `buf` field is written out of bounds, which has a total of 1500 bytes. The following fields such as `buf_size`, `controlled_field`, `log_obj`, and `controlleded` are all controllable.

This is a picture taken from source code of "ACOpusDecoder::ProduceOutputBufferList" function. The program calls `opus_packet_get_samples_per_frame` function first, then it calls `opus_packet_parse_impl` function. The return value is `frame_num`. if `frame_num` is greater than or equal to zero. We enter the true branch. Then the program compares v37 and v23, if v37 is less than or

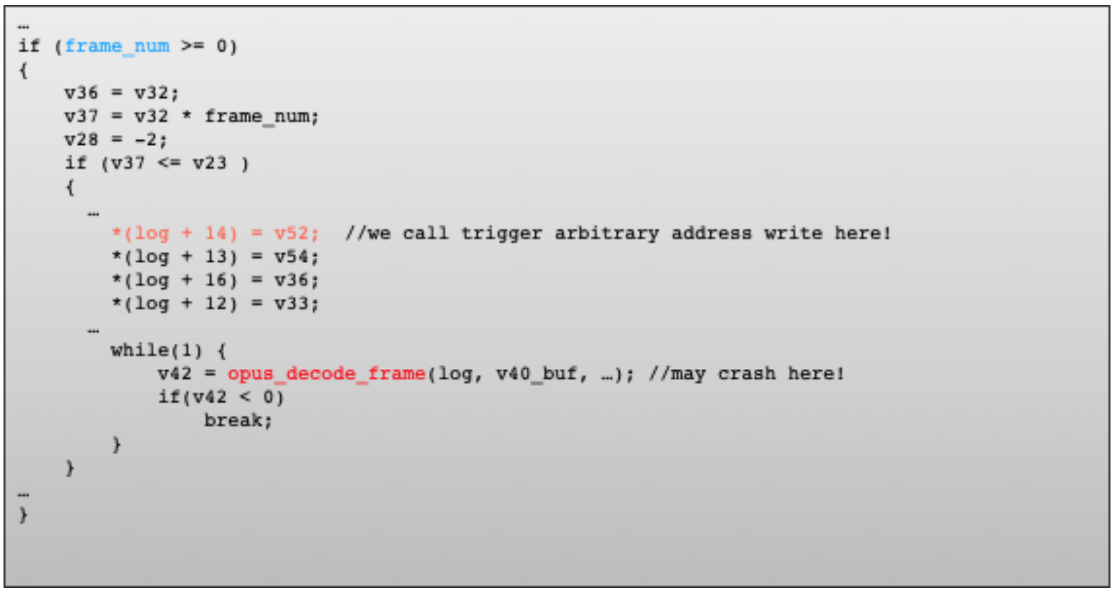
equal to v23, we enter the true branch. In the true branch , program writes some value to addresses related to `log` object.
Next, we have two goals. One is to go to the location where arbitrary address write occurs, and the written value must meet certain conditions; the other is to make the program not crash immediately after causing arbitrary address write. In the first step, we can do it by controlling the value of some variables, we need to ensure the return value of `opus_packet_parse_impl` is greater than or equal to 0, to reach the point where arbitrary address write happens. This step is relatively simple, only needs to control the value of a few variables.



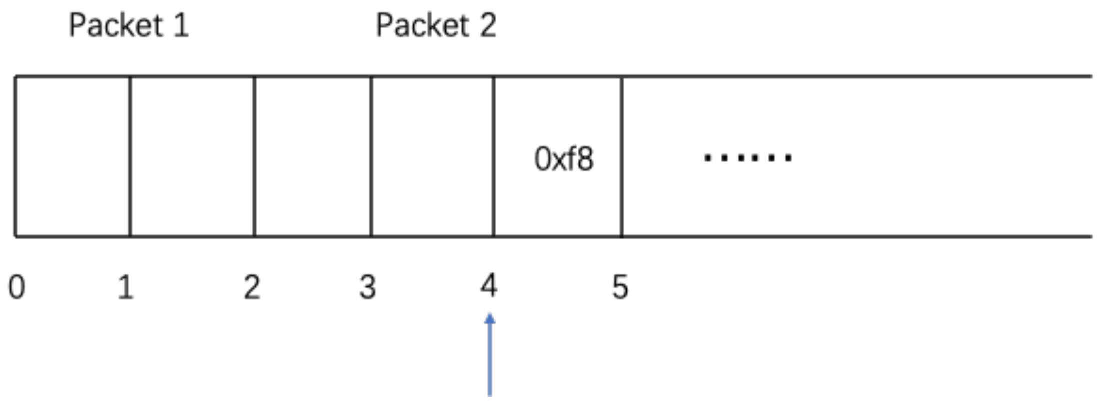
```
v32 = opus_packet_get_samples_per_frame(v24, *(log + 3));
…
frame_num = opus_packet_parse_impl(v55_buf, v56, &53, 0LL, frame_len_buf, &50); //control
the parameters to ensure the return value is greater than or equal to zero.


if (frame_num >= 0)
{
    v36 = v32;
    v37 = v32 * frame_num;
    v28 = -2;
    if (v37 <= v23 )
    {
      …
        *(log + 14) = v52;  //we call trigger arbitrary address write here!
        *(log + 13) = v54;
        *(log + 16) = v36;
        *(log + 12) = v33;
      …
}
…
}
```

Some twists and turns occurred in the second step. After an arbitrary address writing occurs, we find that the program always crashes in `opus_decode_frame`. According to the conventional analysis, if we want to achieve arbitrary address write, it will crash after write occurs. If we try to avoid the crash, we can't achieve arbitrary address write. But in the process of reverse engineering, I found a new bug, the function used for parsing the packet does not check the length of the packet, and will cause an out-of-bounds parsing. So, I constructed two packets that overlap each other.



```
…
if (frame_num >= 0)
{
    v36 = v32;
    v37 = v32 * frame_num;
    v28 = -2;
    if (v37 <= v23 )
    {
      …
        *(log + 14) = v52;  //we call trigger arbitrary address write here!
        *(log + 13) = v54;
        *(log + 16) = v36;
        *(log + 12) = v33;
      …
        while(1) {
            v42 = opus_decode_frame(log, v40_buf, …); //may crash here!
            if(v42 < 0)
                break;
        }
    }
…
}
```

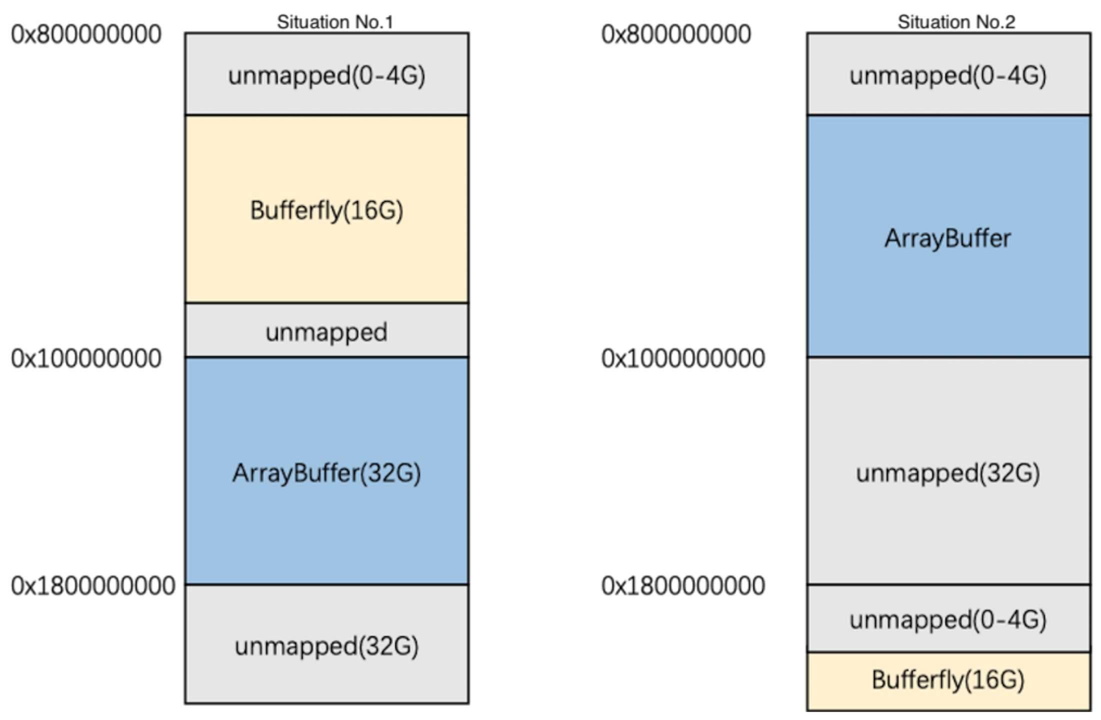


Packet 1 is two bytes. The function used for parsing assumes that each packet is at least 4 bytes. So, it will be parsed out of bounds to Packet 2. The 0xf8 in Packet 2 is regarded as the TOC field in Packet 1, and finally, we avoid a crash and achieve arbitrary address write in the meantime.

3.3 Heap spray, break ASLR!

Usually even with the ability to write arbitrary address, if the program's ASLR protection is done well, you have to find an information leak bug to exploit the vulnerability. However, there are some problems in the implementation of Safari's heap, which leads us to spray the value we control on a fixed address by heap spray. With arbitrary address writing, the first thing that comes to mind is to overwrite the length field in JSArray, or the length field in ArrayBuffer. Due to Safari’s Gigacage mechanism, even if the length field of ArrayBuffer is overwritten, we cannot read or write meaningful content, so I finally selected JSArray.

JSArray in Safari uses Butterfly to store its length and content. If the length of one JSArray is overwritten, then the content of the next JSArray can be read and written out of bounds, and the two primitives fakeobj and addrof can be constructed. I first tried to spray 2 G of memory, and found that my Butterfly sometimes sprays between 0x800000000-0x1000000000, and sometimes sprays between 0x1800000000-0x1c00000000. Due to the heap isolation mechanism of Safari, different types of objects are in different heaps. Butterfly is in a heap called Gigacage in Safari. Some research on the Gigacage heap has found that the base address of Gigacage is predictable, and there are two types of Gigacage, one can store Butterfly and the other can store ArrayBuffer. For these two types of heaps, Gigacage does small randomization, one is Butterfly is on the upper side, and the other is ArrayBuffer is on the upper side. As shown below. Let’s look at the situation One, starting from 0x800000000, an unmapped area of 0-4G will be randomly generated, and then Butterfly’s heap will be located below. In the second case, starting from 0x1800000000, an unmapped area of 0-4G will be randomly generated, followed by the Bufferfly heap. In either case, the degree of randomization of the base address is very small.

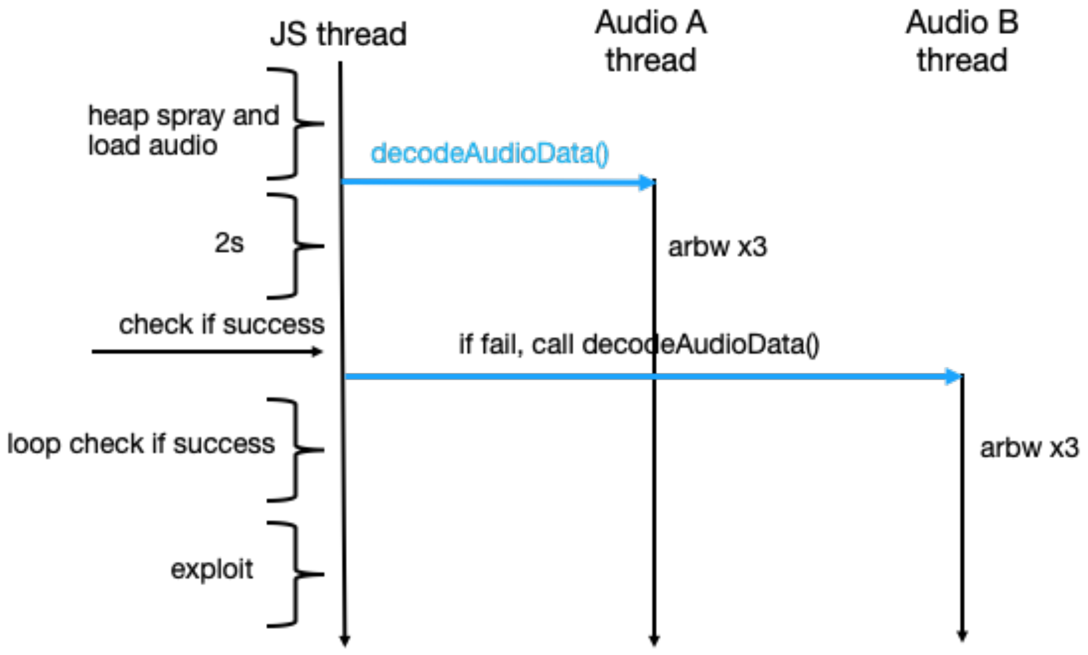


I first tested it on a machine with 16G memory. In order to improve the success rate, I sprayed 4G. But later I found that Safari monitors the memory used by each render process. If the memory used is too large, it will be killed. So, I chose to spray 2.5 G finally, but this will cause the success rate to drop a little. But this is not a big problem. We are lucky to be able to trigger arbitrary address writing multiple times by modifying our CAF file, to pull back the success rate. In order to facilitate the transformation of CAF files, I used ASM syntax to describe CAF files. ASM syntax supports the definition of various width values, such as byte, word, dword, qword, and also supports tags, placeholders, and repeating a value multiple time. This picture is a part of my ASM file, which defines the CAF file header, DESC chunk, some values used to trigger the vulnerability, and so on. Then I will introduce how the exploit process is implemented.

```
_CAF_FILE_HEADER:
    db  0x63, 0x61, 0x66, 0x66, 0x00, 0x01, 0x00, 0x00
_DESC_CHUNK:
    db 0x64, 0x65, 0x73, 0x63, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x20, 0x40, 0xE7, 0x70, 0x00,
    db 0x00, 0x00, 0x00, 0x00, 0x6F, 0x70, 0x75, 0x73, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    db 0x00, 0x00, 0x03, 0xC0, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00,
0x00, 0x00
…
    db 0x8c, 0x58                 ; length of packet which triggers
out-of-bounds address write
    db 0x2                  ; length of packet which triggers
arbitrary address write
    db 0x86, 0x2f                 ; length of packet which prevents
crash
…
times 70000000 db 8 ; length of packets used for padding
    db 0x81, 0x70, 0x81, 0x7F, 0x81, 0x76  ; padding
    db 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x1A ;
negative length used for triggering the vulnerability
```

3.4 Thanks to multithreading! Give exploit code enough time to execute before program crashes

The following sequence diagram explains the entire exploit process. In the beginning, there was only one JS thread. We do heap spray first, and constructed the audio file in the memory, then call the `decodeAudioData` function which performs the audio decode process. Since Safari decodes the audio in a separate thread, The Audio A thread will be started here. Let's first assume that the memory layout after the heap spray is the situation 1 mentioned above. Then the Audio A thread will trigger arbitrary address write for three times in address range 0x800000000 to 0x1000000000 , and the JS thread will detect whether the length of JSArray is changed after two seconds, if it is changed, it means that the heap layout is indeed case 1, and then the subsequent exploit code can be executed. If it is not changed, it means the heap layout is case 2, then call decodeAudioData() for the second time, start Audio B thread to decode audio, this time we will trigger arbitrary address write in address range 0x1800000000 to 0x1c00000000. The JS thread loops to check whether the length of the JSArray is changed, and if it is changed, it will execute subsequent exploits. If it fails, it means that the entire exploit has failed.



Besides, there is a problem that needs to be solved, that is, after the audio file is decoded, when the free function is called to clean up the resources, a crash will be triggered. There are several ways to solve this problem, one is to repair the damaged heap, and the second is to make the audio decoding time very, very long, and our exploitation process ends before the decoding is over. The first approach is too complicated because we need to search the heap and it takes a certain amount of time to repair the heap. The program may crash when we are repairing the heap. So, we choose the second approach at last, I constructed a 600M CAF file with more than 70 million packets. It will take about 50s to decode all of these packets, which is enough for my exploit.

3.5 Old school, arbitrary address read/write to arbitrary code execution

After covering the length field of JSArray, we can construct fakeobj and addrof primitives, and then we can use these two primitives to construct arbitrary address read and write primitives. At last we write the shellcode into the JIT area to execute arbitrary code. These are all old-school things, I won't go into details here. Audiences who are interested in it can read saelo's article 《Attacking JavaScript Engines - A case study of JavaScriptCore and CVE-2016-4622》.

4. Conclusion

Throughout the process of discovering and exploiting the WebAudio vulnerability, I think I can summarize the following three points. One is that in addition to the DOM and JS engine, there are still many unconcerned attack surfaces in Safari. The second is that there are still a lot of low hanging fruits in libraries without source code. The third is that even with a lot of modern defense mechanisms, they cannot completely prevent advanced attackers from exploiting vulnerabilities. Safari's ASLR defense mechanism still needs to be strengthened.