Not so Smart: On Smart TV Apps

Marcus Niemietz, Juraj Somorovsky, Christian Mainka, Jörg Schwenk
Horst Görtz Institute for IT-Security
Ruhr-University Bochum, Germany
{*firstname.lastname*}*@rub.de*

***Abstract*—One of the main characteristics of Smart TVs are apps. Apps extend the Smart TV behavior with various functionalities, ranging from usage of social networks or payed streaming services, to buying articles on Ebay. These actions demand usage of critical data like authentication tokens and passwords, and thus raise a question on new attack scenarios and general security of Smart TV apps.**

In this paper, we investigate attack models for Smart TVs and their apps, and systematically analyze security of Smart TV devices. We point out that some popular apps, including Facebook, Ebay or Watchever, send login data over unencrypted channels. Even worse, we show that an arbitrary app installed on devices of the market share leader Samsung can gain access to the credentials of a Samsung Single Sign-On account. Therefore, such an app can hijack a complete user account including all his devices like smartphones and tablets connected with it. Based on our findings, we provide recommendations that are of general importance and applicable to areas beyond Smart TVs.

***Keywords*-Internet of Things; Smart TV; App; Single Sign-On; OAuth; TLS; File System; XXE; XHR; Privacy; Samsung**

I. Introduction

Only some years ago, a television (TV) was a medium to watch news, broadcasts, documentaries, or blockbusters. This behavior has changed with the vision of *Internet of Things*: devices are connected via the Internet to achieve a more comfortable and better life for people. So-called smart devices collect your calender appointments, weather-forecast information, Email, and Facebook messages. Legacy personal computers are less frequently used, because a lot of daily tasks can be accomplished using a smartphone, tablet, or smartwatch. Even the traditional TV has changed: Smart TVs are devices that combine a TV with a computer that is connected to the Internet. In some cases, this computer is integrated into the TV, in other cases one can extend a TV to a Smart TV by using digital media streamer like Google's Chromecast or Amazon's Fire TV. According to Strategy Analytics, there were close to 580 million Smart TV devices by the end of 2014 [1]. Gartner's 2014 consumer survey pointed out that almost 25 percent of U.S. and 32 percent of German households have a Smart TV [2]. Furthermore, Gartner forecasts that 87 percent of the shipped TVs will be Smart TVs in 2018. These data illustrate that such devices are attractive for attackers due to their distribution and way of connection.

Smart TV Apps. An important feature of a Smart TV are applications, which are also known as apps. Similarly to apps used on smartphones, Smart TVs come usually with some pre-installed apps. Further apps can be installed using Smart TV app stores or, in some cases, users can develop their own apps and install them directly from a USB flash drive.

The apps allow Smart TV users for using social networks like Facebook, streaming services like Netflix or Watchever, playing games like Angry Birds, or browsing common websites. Consequently, many of the apps operate with critical data such as login credentials or *Single Sign-On (SSO)* access tokens. In case an attacker can access these tokens, he is able to impersonate Smart TV users and access their confidential data. This raises several questions regarding Smart TV apps and their security.

Contribution. In this paper, we present a security case study on Smart TV apps. We first analyze different attack models. Based on these models, we analyze attack possibilities, which are new in the context of Smart TVs, but of general importance for apps used in similar scenarios. Afterwards, we evaluate these attacks using real Smart TV devices:

- We show that even trivial security mechanisms, like encryption with TLS, are not used by some Smart TV apps during the login procedures. In this relation, we discovered critical security vulnerabilities and privacy violations.
- On the example of the Smart TV market share leader Samsung, we show that Smart TV apps can be used to automatically get full file system read access. Therefore, we were able to steal all of the TV's saved data including SSO usernames and passwords.
- Furthermore, we detected many critical vulnerabilities by just starting an app created by the attacker. To name some examples, we were able to steal Wi-Fi passwords, SSO data, and secret OAuth credentials. To exploit these vulnerabilities, we inter alia bypassed Samsung's app browser engine restrictions and took advantage of incorrectly implemented system features like XML parsing procedures.
- We show that the exploitation of one TV can lead to impersonation attacks on millions of other TVs by stealing hard-coded OAuth credentials. More precisely,

we could extract OAuth credentials for Samsung, Ebay, and Facebook apps on our Samsung device. These credentials have to be kept private since they are used to authenticate the apps, and are shared amongst millions of other Samsung TVs.

Attack Generalization. Even though our attacks are used to gain credentials and other confidential data from Smart TVs and their apps, our attacker model and analysis strategy can be applied to different scenarios beyond Smart TVs as well, where the devices meet the following prerequisites:

- Connection to the Internet
- App rendering engines from a browser
- Installed attacker's app from an app store or via an attached storage device

This includes cars with technologies like BMW Connected Drive to use Twitter or Facebook directly from the board computer.[1] Nowadays, even glasses (e.g., Google Glass) and smart watches have apps.

Responsible Disclosure. We informed the manufacturers about the detected vulnerabilities to make Smart TV devices more secure.

II. Tested Devices

For our tests we picked five devices from two categories. First, we chose two Smart TVs: Samsung UE22H5670 and Grundig 42VLE922BL. Second, we chose devices that can be used to provide a Smart TV functionality: Apple TV, Google Chromecast, and Amazon Fire TV. All devices were tested with their latest available software versions (see Table I). We selected these devices for different reasons. According to Strategy Analytics, Samsung is the Smart TV market share leader with 25,4% in Q4'12 and 26,4% in Q4'13 [3]. A manufacturer that is not listed in the TOP-5 market share list is Grundig and thus suitable as a Smart TV manufacturer with devices that do not have a high market share. Furthermore, digital media streamers like Apple TV, Google Chromecast, and Amazon Fire "proved to be the fastest growing device category" [1] in Q3 2013.

Manufacturer	Device	Version
Samsung	**UE22H5670**	**2606**
Grundig	42VLE922BL	J5GRMR
Apple	TV	7.0.3
Google	Chromecast	27946
Amazon	Fire TV	53.1.1.0

Table I
Five tested Smart TV devices from different manufacturers. A thorough analysis was executed for the highlighted device of the market share leader Samsung.

In our work, we first tested these five Smart TV models for eavesdropping attacks and report on our findings in Section V. Motivated by the results and by the importance of Samsung in the segment of Smart TVs, we moved to a thorough analysis of our Samsung device, which described in Section VI.

III. Attacker

During our tests we used three types of attackers. Attacker *A* is a sniffer who eavesdrops the devices connection to the Internet. Attacker *B* is connecting an attached storage device to the TV and has thus physical access to it. Attacker *C* needs a victim who is visiting an attacker controlled page. We tested all our Smart TV devices with attacker A, and our Samsung Smart TV with attackers B and C.

A) Eavesdropper. This attacker is sniffing HTTP traffic by eavesdropping the device's connection. In our test setup, we connected each device as well as our eavesdropping device in an unencrypted Wi-Fi network.[2]

B) Attached Storage Attacker. The attacker connects an attached storage device like a USB flash drive to the TV. Applicable scenarios are freely accessible TVs in hotel rooms or stores selling consumer electronics.

C) Malware Attacker. The attacker creates a malicious app and uploads it to the Smart TV device's app store or attacker controlled website. The attack is executed if the victim downloads or executes the attacker's app on the Smart TV device.

Attacker C requires a social engineering attack. The attacker's aim is that the victim executes the attacker's app by clicking on a link in a spam mail or inside an instant messenger. If the victim is not installing the attacker's app directly from the app store, the downloaded app has to be copied on a storage device. This storage device must be afterwards connected with the TV. The attacker's app can for example be a simple calculator in the foreground; in the background, attacker's code is executed. For legal reasons, we did not verify whether it is possible to provide a malicious app on, for example, Samsung's app store. However, providing such apps seems to be achievable due to our analyzed privacy violating app VEVO described in Section V-B.

IV. Foundations

In the following, we present security topics relevant to our paper. Readers familiar with these topics can safely skip this section.

Transport Layer Security (TLS). TLS [4] is the most important security protocol on the Internet. It is located between the transport layer and the application layer in

[1] http://www.bmw.com/com/de/insights/technology/connecteddrive/2013/index.html

[2] For eavesdropping attacks, we used OS X 10 with the integrated W-LAN sniffer and Wireshark.

the TCP/IP reference model. Its main purpose is to protect integrity, authenticity and confidentiality in application protocols like HTTP or IMAP, so that these protocols can securely send critical data like passwords or cookies over insecure networks.

We are not going to analyze the TLS protocol structure. For our paper, it is important to know that confidential and security critical data must always be sent over TLS. Otherwise, an attacker with eavesdropping capabilities can see the transmitted data in plaintext.

XML and XXE. eXtensible Markup Language (XML) is a W3C data format [5]. It is used for transmission, validation and interpretation of data in different applications ranging from web services and office applications, to configuration files used in various servers and appliances. The huge number of application scenarios adapting XML technology resulted in a huge number of extension specifications allowing to define schemas for XML documents or to apply cryptographic primitives directly on the XML level. In the following, we introduce a standard called Document Type Definition (DTD), which is necessary for further attack descriptions.

DTD allows for declaration of new XML building blocks in the prolog of an XML document. These building blocks are called XML entities. XML entities are inserted into the XML document and resolved during XML document parsing. There are two types of XML entities: internal and external, see Listing 1 with a DTD declaration.

```
<!DOCTYPE config [
 <!ENTITY title "Configuration file">
 <!ENTITY ext SYSTEM "file:///text.txt">
]>
<config>
  <title>&title;</title>
  <detail>&ext;</detail>
</config>
```

Listing 1. An XML document containing a DTD declaration with an internal and external entity.

When an XML parser processes such a document, it first reads the entities in the XML prolog. Afterwards, it resolves all the entity occurrences in the document: `&title;` is replaced with a text `Configuration file` and `&ext;` is replaced with the content of the `file:///text.txt` file.

Resolving external entities can become very dangerous if an attacker controls content of processed XML files. This is the case in many applications, for example web services. If the attacked XML parser resolves external entities, the attacker can force the parser to read arbitrary system files and send them over the network. These attacks are referenced as eXternal XML Entity (XXE) attacks [6].

Delegated Authorization: OAuth. OAuth is a framework for delegated authorization. In contrast to Single Sign-On Systems like OpenID [7] and SAML [8], the idea of OAuth is not to log in the user into an application, but to grant access rights on specific resources to it. In most cases, the application, referred to as the *OAuth Client*, is only authorized to access a subset of resources owned by the user. There are some variants of OAuth: OpenID Connect and Facebook Connect. Both are based on OAuth and for simplicity, we do not distinguish between these three and refer them by using the term *OAuth*.

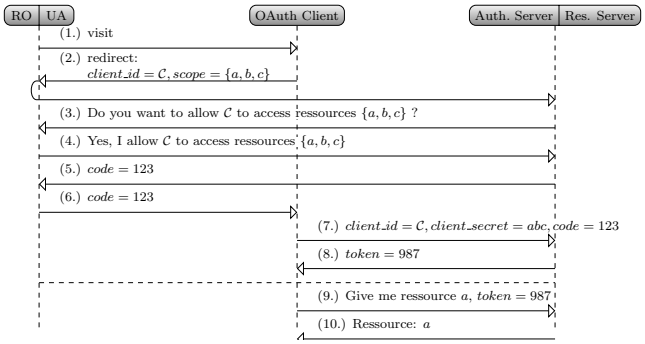

Figure 1. OAuth workflow simplyfied.

The basic OAuth protocol flow is depicted in Figure 1 and works as follows: A resource owner visits the website belonging to the OAuth client. Then, the client redirects the resource owner's user agent (UA, the browser) to the authorization server, for example to Facebook. The resource owner is asked to confirm that the OAuth client is allowed to access some resources. If the resource owner accepts, a so-called *access code* is given to him.[3] Next, the resource owner sends the code to the OAuth client. This can happen either by an HTTP redirect or by letting the resource owner entering it manually. Finally, the OAuth client can submit the access code to the authorization server to receive the *access token*. The access token is then submitted to a resource server in order to receive its resources. Generally spoken, the access token is comparable to a session cookie identifying the resource owner, but with restricted access rights.

Noteworthy is the fact that the security of OAuth is highly bound to the use of TLS. If either the access code (Steps 5 and 6) or the access token (Step 8 and 9) is transferred unprotected, an eavesdropping attacker can use it to access the same resources as the OAuth client. Furthermore, the OAuth client uses the credentials $(client\_id, client\_secret)$ in Step 7 to authenticate itself against the authorization server, and therefore it is also of high interest for an attacker.

UI Redressing. UI redressing is a technique to hijack the victim's actions like clicks and keystrokes. Introduced in 2008, Hansen and Grossman showed with clickjacking that `iframe` elements can be used to give the attacker

[3]For the sake of simplicity, we do not consider other OAuth flows, for example *implicit grant*.

access to the victim's camera and microphone by using ordinary HTML and CSS code [9], [10]. The crucial point of their attack was that a victim is clicking on an attacker's controlled web page and that these clicks will be actually used to click on an invisible `iframe` element loading the target web page (e.g., `macromedia.com`).

V. Eavesdropping Attacks

Eavesdropping attacks are well-known and a useful starting point for security reviews. Due to the high number of Smart TV devices it is interesting to know, whether an application can leak data if the user's connection is getting eavesdropped. Therefore, we analyzed apps from all five devices. First, we identified apps using security critical data such as login data or SSO tokens. Then, we checked whether the data is sent securely over an encrypted channel or not. Section V-A underlines that eavesdropping attacks are still a serious problem. On top of that Section V-B discloses a critical privacy violation and also a way to steal highly sensitive SSO data.

A. TLS Adoption

Each of our five tested devices was analyzed by looking at the HTTP traffic of their apps.[4] Table II shows the total number of apps, the number of apps with a login option (form, PIN, OAuth), and whether these login credentials were submitted in cleartext via HTTP. Our attacker A has in such a case the possibility to hijack login credentials if he eavesdrops the victim's unencrypted connection and if one of the vulnerable apps is used by the victim at the same time.

Manufacturer	Apps	Login	Unencrypted
Samsung	56	16	4
Grundig	34	7	3
Apple	28	17	1
Google	10	–	0
Amazon	20	4	0

Table II
We analyzed the overall number of apps and apps using login credentials in the provided Smart TVs. Afterwards, we analyzed whether the apps with login credentials send the login tokens unencrypted.

In the case of Samsung we picked the 25 pre-installed apps and the TOP-5 most popular apps from each of the seven available app store categories. Three apps from the app store set were in at least more than two categories and one app had a black screen. Therefore, we tested 31 different apps from the app store and the 25 pre-installed apps (56 in total). Two apps of the 25 pre-installed apps (WATCHEVER v2.200, ImmoScout24 v1.010) and two of the 31 app store apps (NewMoove v2.2003, Putpat TV v2.504) did not encrypt their login data via TLS. Furthermore, we have detected a critical privacy violation and information leakage channel discussed in Section V-B.

Grundig does not provide arbitrary developers a possibility to write new apps as is the case in Samsung Smart TVs. All the Grundig apps are developed by a company called Arcelikapps.[5] We could find altogether 34 apps in the Grundig app store. Seven apps used login credentials. Three out of these apps sent the login credentials unencrypted: Facebook, Ebay, and Viewster. The Facebook and Ebay apps used OAuth, and we were able to follow an unencrypted login procedure containing OAuth access tokens in plaintext. We analyze these problems closer in Section V-C. The Viewster app used a simple password login.

Apple TV does not provide any app store so that we tested all of its 28 pre-installed apps. Just one app out of 28 apps did not encrypt its connection during the log in process (WATCHEVER).[6] This app is also contained in our Samsung Smart TV app set.

Google Chromecast does not have apps in a way like Samsung or Apple. Apps are triggered and controlled by smartphones, tablets, or laptops. Therefore, a device like a smartphone can be used as a remote controller. However, we tested five apps by using the pre-installed Google Chrome with the extension *Google Cast*. To name one example, we opened YouTube on the TV. We did this by clicking on the TV button of YouTube's web page shown in our laptop's Chrome browser. We could not find any login information because the data were always transmitted over encrypted connections.

Amazon's Fire TV does not have pre-installed apps but it offers an app store. We installed the 20 most popular free apps. Four of them are handling login data and all connections were encrypted.

B. Privacy Violation and SSO Hijacking

During our tests we analyzed the HTTP traffic initiated by the app VEVO (v3.701) on our Samsung Smart TV. Please note that VEVO was also in our Apple TV testbed. However, in the case of Apple TV the HTTP connection was encrypted and we were not able to reproduce our Samsung results on this device. VEVO can be used to watch music videos, artist videos, and original shows on the user's TV. During our analysis of the login procedure we have discovered HTTP GET requests going to scorecardresearch.com (cf. Figure 2). This website is a service from a market research company, primary analyzing surveys and web tagging data.

```
http://b.scorecardresearch.com/p?...&
    ns_jspageurl=file:///mtd_down/widgets
```

[4] Our testbed was in Germany. For this reason, some of our tested apps are only available for the German market.

[5] http://arcelikapps.com/

[6] We could not identify the version number of WATCHEVER. However, the app was tested with its newest version on 28/03/2015.

```
/normal/111399001425/index.html?
country=DE&language=17&lang=en-GB&
modelid=14_X14_2D&server=operation&
remocon=0_650_259_22&area=PANEURO&
product=0&mgrver=6.1571&ssoid=
USERNAME&ssopw=PASSWORD...
```

Listing 2. With dots truncated HTTP GET request initiated by the VEVO app.

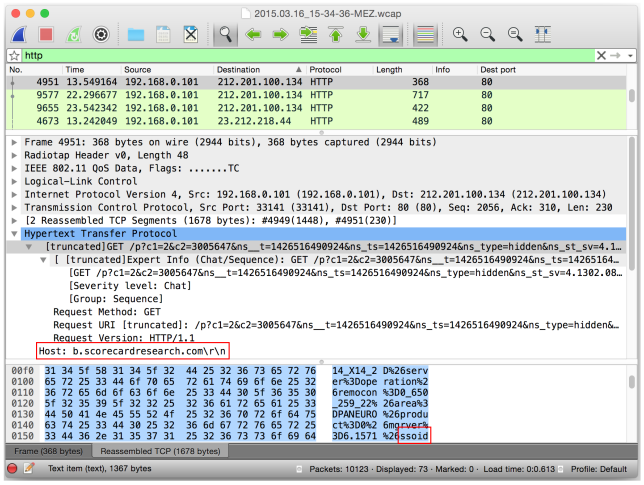

Figure 2. In the case of VEVO's Samsung Smart TV app, the user's Samsung SSO data was transmitted in cleartext to an advertiser.

The requests going to this website contain inter alia the following data: Video artist, video player information, current video channel, user identifier, and Samsung SSO username including its password (cf. Listing 2).

Please note that these requests are transmitted regularly to provide information like the currently watched video. The crucial point is that the credentials of the user's Samsung SSO account are submitted unencrypted via HTTP.[7] Therefore we have at least two attack scenarios. First, the marketing company is able to login with the user's Samsung SSO account and thus they could have access to all provided services. Second, an eavesdropper could sniff the connection and steal the SSO account's data.

We evaluated the impact of this issue: By using the SSO login data, we could visit Samsung's configuration website[8] and got access to the following services:

- Samsung GALAXY Apps (download applications)
- Find My Mobile (manage missing devices)
- MILK (radio streaming service)
- PEN.UP (social network)
- S Health (e.g., fitness and health management)
- Samsung Wallet (e.g., tickets and coupons)

[7]We require that the victim is logged in with its account. This is a convenient way to not type in credentials for different services and apps.

[8]https://account.samsung.com

Furthermore, the attacker has access to Samsung Cloud services that can be used to store files from the TV, PC, or mobile phone. Additional services can exist depending on the country of which the user comes from. All in all, the attacker has access to data across devices that are linked with the Samsung SSO account.

As a critical side effect and privacy violation, the market research company and also the attacker can clearly identify the user.

C. Unencrypted Facebook and Ebay OAuth Tokens

Facebook and Ebay apps for Grundig Smart TVs use OAuth to gain access tokens for web services and receive user's data. As mentioned above, these tokens are transmitted unencrypted, over HTTP. In the following, we give a summary on the login procedure with the Facebook app. A similar workflow is established with the Ebay app.

When the user installs the Facebook app, the app requests him to use his browser and visits http://tv.grundig.com/fb. This initiates an OAuth authorization procedure: the user is forwarded to facebook.com and is asked to authorize the Grunding app to access Facebook resources. Then, he receives a nine digit access code. The nine digit code can be supplied to the Grundig app. After inserting the code into the app, the app invokes the following address: http://tv.grundig.com/fb/getToken.php?code=[code]. It is then redirected to http://grundigapps.com and receives an OAuth token. The token is stored locally on the Smart TV app and is used for further communication with Facebook. An example of such a token is depicted in Listing 3.

```
1 {
2   "access_token":"CAABuxhx...",
3   "extend_token":"true"
4 }
```

Listing 3. An OAuth token example used for access Facebook resources. The token has no expiration date.

This token can be used to access Facebook resources, view friends and photos, or post on the timeline. These capabilities are also gained by an attacker, who eavesdrops the OAuth token. This is possible since the whole communication is executed over an unencrypted channel.

In case of the Ebay app, the app authorization procedure is conducted over an encrypted TLS channel. However, the resulting OAuth access token is finally transmitted over an unencrypted channel. Thus, attacker A can similarly gain an OAuth access token and access Ebay resources.

VI. Samsung: Attached Storage and Web Attacks

Our findings in Section V point out that Samsung has a high number of apps with unencrypted connections and at least one app with a massive privacy violation and SSO account hijacking possibility. Combined with the leading

market share situation, this motivated us for an in-depth analysis of our Samsung Smart TV device and its vulnerabilities to the attached storage and web attacks.

While executing these attacks, the attacker first creates an app and then installs it on victim's Smart TV. He can achieve this either by using a USB drive (Attacker B) or Samsung's app store (Attacker C). Obviously the malicious app can cause damage or affect user's sensitive data. For example, it can use the web API to control TV and audio channels, turn on camera, connect to the Internet, or execute arbitrary functions.[9] However, we are going to analyze how a malicious app can harm the whole Smart TV system, and what functions it can execute *beyond the web API*. This is also motivated by Samsung's advertisements on their developer page saying that their Smart TVs have security modules to protect users against malicious apps.[10] We could not find any documentation on these modules, but we verified that HTML elements are restricted, e.g. the `iframe` element which is often used for UI redressing attacks.

Motivated by the attacks described in Section V-B, we start this section with a search for possible reasons why the VEVO app sends Samsung SSO token to further web sites: We found out, the reason was the Document Object Model (DOM) saving Samsung's SSO credentials in objects like `document.location`. Afterwards, we analyze possibilities to access the whole Smart TV file system. At the end, we give an example how to access secret data using an XXE attack.

A. Web Attacks

Stealing SSO Data. Samsung Smart TV apps are built of HTML, Cascading Style Sheets (CSS), and can contain scripts, for example, JavaScript. Thus the developer of an app is allowed to access the DOM for using objects like `document.title`. If the user is logged in with his Samsung SSO account on his TV, an attacker could steal the user's account if one requirement is fulfilled: The user has to open the attacker's application. If this is the case, the malicious app can use the DOM with `document.location` to steal the victim's credentials.

An example for a `document.location` output is shown in Listing 4. It starts with the given protocol handler `file://` followed by the path of our USB flash drive. The app with the name SSO was executed on 09/03/2015 and the site executing the JavaScript code is `index.html`. Next to information like the country code and model number there is given the SSO identifier and password.

```
file:///dtv/usb/sda1/SSO6_0.100_Europe_
    20150309/index.html?country=DE&...
    ssoid=USERNAME&ssopw=PASSWORD&...&
    realmodel=UE22H56000&...
```

Listing 4. With dots truncated `document.location` from an app loaded through a flash drive.

By implementing a code as shown in Listing 5, the attacker could steal these user credentials. The app requests a not existing file from attackers.org consisting of the Base64 encoded URL from `document.location`. Due to this code, the attacker and owner of attackers.org just has to check the error logs and decode the Base64 string to retrieve the user data.

```
<script>
document.write(
  '<img src="http://www.attackers.org/'+
      btoa(document.location)+'">'
  );
</script>
```

Listing 5. Attacker's code stealing the user's SSO login data by just letting the victim open the attacker's app.

Web Attacks and UI Redressing. By looking on the user agent, we analyzed that two different browser engines are used. We analyzed the features of both browser engines and it turned out that OWASP TOP-10 attacks can be carried out in the web and app browser engine.[11] By looking on the additional risks listed in the OWASP TOP-10 sheet, we discovered restrictions for clickjacking attacks. Samsung's app browser does not show `<iframe>` elements. We bypassed this restriction by using the `<object>` element with the media type `text/html` (cf. Listing 6). This bypass turned out to be very important and it will be discussed in Section VI-B.

```
<object type="text/html"
  data="http://www.example.org"
  style="width:100%; height:100%">
```

Listing 6. Our truncated source code of the UI redressing attack bypassing the `iframe` element restriction.

B. File System Access

File Browser. A desired goal while analyzing the Samsung TV was to get *file system access*. As the TV is not shipped with a file browser and the included Media browser is only capable to see the content of, for example, Image/Video/Audio files on an attached flash drive, we had to build one on our own. To implement it, we could just rely on standard HTML, CSS, and JavaScript features. Due to the restricting of not being allowed to use the `<iframe>` element, we had to circumvent this restriction. As mentioned in Section VI-A, we found a bypass by using the HTML `<object>` element as a replacement for it. We used the `<object data="file:///">` tag to create a simple file browser app. Note

[9] *Web API*, Samsung, http://developer.samsung.com/web-api, May 2015

[10] *TV Apps Security*, Samsung, http://www.samsungdforum.com/Support/TVAppsSecurity, May 2015

[11] *TOP-10*, OWASP, https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project, May 2015

that commonly `<iframe>` and `<object>` elements are intended to use the `http://` protocol, but by using the protocol `file://` and adding a third slash, we were able to see the content of the TV's root directory. Figure 3 shows that we were able to reload the attacker's app, type in a path and optionally a filename, and load the typed in data inside the `<object>` element. We were also able to *send contents of the selected files via an XMLHttpRequest (XHR)* to the attacker's server.

Typically, files can be easily accessed by using a pointing device like a mouse. Due to the reason that some keyboards are not supported or just supported in a limited way, we used Virtual Keyboard in the input field.[12] An interesting fact we observed is that the underlying system seems to be a UNIX system, but we were allowed to access all files (incl. system files) on the TV without any restrictions.

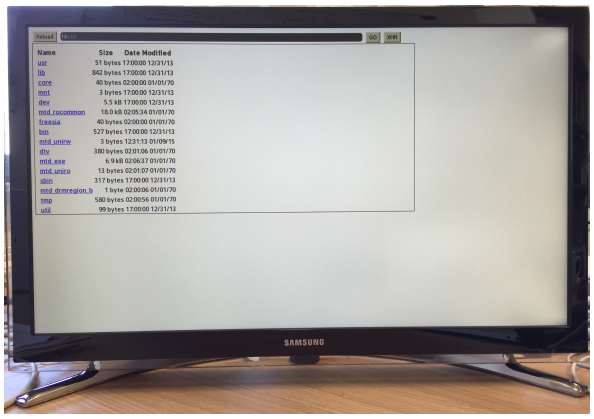

Figure 3. Self-written file browser as an attacker's app using the `file` protocol.

XMLHttpRequest. Our file browser allowed us to read arbitrary files and display them on the TV screen. We then went one step further: We wanted to be able to download arbitrary files for a better analysis. One way to do this is to let the application use the XHR API. We read the file using XHR, encoded it with Base64, and sent it to our self-controlled server as GET requests. Because GET requests are restricted in their length, we spitted the Base64 encoded file content and performed multiple GET requests successively. We were able to download all files of the file system and we integrated this functionality in our file browser app. It can be triggered by clicking on the `XHR` button (see Figure 3). However, an attacker could also define the file to download directly by typing in the path and filename into the address bar of our file browser. Please note that an application of the attacker has to be opened to execute this attack.

XML External Entity. Another possibility to download files automatically from the Smart TV is by using XXE attacks. We will elucidate this approach in the following section.

[12] *Virtual Keyboard*, Rob Garrison, http://mottie.github.io/Keyboard/

C. Reading Samsung TV's Sensitive Data

By using our file access mechanisms, we were able to steal sensitive data which should be unaccesable by the attacker. Our findings include:

- Wi-Fi data (incl. passwords)
- App and Web browser cookies, local storage
- Browser history
- Common user inputs (incl. typed in passwords)
- Samsung SSO username
- Samsung's Samsung and Facebook credentials

Wi-Fi data are saved in cleartext so that we were able to read out the hotspot's name and also its password by accessing one file. As it is in the case of Wi-Fi data, cookies can be read out and thus the attacker has the possibility to hijack them. The app Zattoo saves the user's credentials (username/password) inside the app's local storage file. As it is on Desktop systems, the user's browser history can be read out.

A critical security risk was spotted out by analyzing a file called `UDBCOMMON`. It saves frequently entered user data. This included in our case terms like `www` and `google`, but also our Samsung's SSO as well as Facebook's username and password. Please note that the user's username of his Samsung account is in contrast to the password always saved on the TV. Thus, the user has to type in his password every time he is logged out. Last but not least, we have analyzed apps saved in a `widgets` folder. This folder contains SSO procedures with several keys in a static JavaScript file, including Samsung's and Facebook's API-key and API-secret for OAuth. This allows us to, for example, impersonate Samsung's Facebook app. Such an impersonation attack can be used to steal user data and to write postings on the user's board.

Attack Example: Stealing Data With XXE. In the following we show how to automatically read out data on the TV. We steal the user's Samsung SSO username by just connecting a USB flash drive with the TV – no more user interaction is required. Though we go into detail regarding attacker B, please note that attacker C could also do this attack.

Imagine the following scenario: The attacker has a USB flash drive with a self-created Samsung Smart TV app containing the code given in Listings 7 and 8. After connecting his USB flash drive with the TV, the attacker checks the logfile on the attacker's server. The logfile shows the SSO account saved on the Smart TV. Please note that the attack with the app works automatically, even every time when the TV restarts.

In the first step, the TV is looking for apps on the USB flash drive by checking for a `config.xml` file. In the case that there is such a file, it will be parsed by the XML engine.

The `config.xml` file usually saves information about the app, for example the name, version, and author. In contrast to a benign file, the attacker adds a reference to an external parameter entity shown in Listing 7.[13]

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE roottag [
<!ENTITY % ext SYSTEM "file:///mtd_chmap
    /operating_account.data">
<!ENTITY % dtd SYSTEM "file:///dtv/usb/
    sda1/App_1_Europe_20150309/attacker.
    xml">
%dtd;
]>
<widget xmlns="http://www.samsung.com/">
  <cpname>&send;</cpname>
  ...
</widget>
```

Listing 7. Truncated `config.xml` file of the attacker's app.

In the second step, the Samsung SSO username saved in the file `operating_account.data` will be read out to the parameter entity `ext`. To send this file to the attacker, there is given `attacker.xml` (cf. Listing 8). This file defines an external entity `send`, which is constructed from the attacker's URL attackers.org and the hijacked SSO username. Finally, by resolving the `&send;` entity from the `config.xml`, the Smart TV sends an HTTP request to attackers.org containing the account data as a parameter name.

```
<!ENTITY % all "<!ENTITY send SYSTEM '
    http://www.attackers.org/tv/?%ext
    ;'>">
%all;
```

Listing 8. Sending hijacked data to the attacker's server.

Due to specific XML parser restrictions, we could only read one line files. We assume that this behavior was caused by a libxml parser used in the analyzed Samsung Smart TV.

VII. Related Work

In this section we discuss and compare related work on Smart-TVs, apps, and OAuth.

TOCTOU attacks. Mulliner et al. showed with a time-of-check-to-time-of-use (TOCTOU) attack called "Read It Twice!" on Samsung's LE32B650T2-PXZG that software installations can be exploited by emulating mass-storage devices [11]. For their attack, they used a non-modified software package or firmware upgrade with a valid software signature during the *Check* process of the TV's update routine. After that they replaced their checked file with a modified one. Due to the reason that the *Install* routine runs with root privileges, their attack can be used to jailbreak the TV and thus get access to file system data.

In this paper we are able to get access to the file system without jailbreaking it. Our attacks do not rely on a valid signature nor on a TOCTOU attack. We are able to automatically hijack sensitive user data like the SSO account by just connecting an attached storage device like a USB flash drive. Moreover, the device can be attacked if the attacker has physical access to the TV or if the victim is downloading and executing the attacker's app (e.g., from an app store).

HbbTV attacks. In the past there were discussed several Hybrid Broadcast-Broadband Television (HbbTV) attacks [12], [13], [14]. One attack with a high impact is from Oren et al. They showed that broadcast streams providing HTML, CSS, and JavaScript code can be used to attack websites on the Internet [15]. In contrast to ordinary Same Origin Policy restrictions, HbbTV data are in their own origin and able to access websites. The authors used this security issue by picking up a TV channel, adding malicious HbbTV data to it, and sending it with a strong signal to the victim. Thus, the victim's TV rendered a malicious page automatically by just requiring that the victim is viewing the malicious TV channel.

While Oren et al. attacked targets on the Internet with the help of TVs, we are attacking TVs to get their sensitive data. They highlighted with different attack types that request forgery and exploit distribution attacks are a high risk. In our case, we can inter alia hijack critical data like the victim's SSO account and therefore we are able to attack his whole account environment; this includes services like digital wallets for saving payment data or health data provided by fitness bracelets. Furthermore, we are able to hijack secret Samsung and Facebook OAuth credentials, which are implemented on several millions of devices.

Code Injection Attacks on HTML5 Apps. Jin et al. [16] created a study regarding the risk of HTML5-based code injection attacks on mobile apps. They underlined that these apps have many data channels like barcode, SMS, WIFI, Contact, and NFC, which can be used for code injection attacks. As an example they created a malicious QR code with payload containing geolocation data that are automatically sent to the attacker's server if the victim is reading this QR code with a vulnerable app.

In contrast to Jin et al., we do not execute code injection attacks through data channels relying on malicious user inputs provided by vulnerable apps. We use a USB flash drive to execute our code directly and are therefore not limited to data channels and vulnerable apps. Furthermore, we are able to eavesdrop an unencrypted connection and hijack sensitive information like login data. Jin et al. mentioned that the Android file system can be used to store malicious code on it. Please note that Android does not allow one application to

[13] External parameter entities have to be used to read a file content and insert it in a further external entity in order to send it over the network, see [6].

read data from another application.[14] We are able to read out the whole file system including data on internal and external storages like cookie files, log files, or the USB flash drive where the application is executed from.

OAuth. OAuth has been analyzed in different contexts. In 2011, Chatfield et al. published a widespread analysis of 96 popular websites acting as OAuth clients [17]. Their work also includes eavesdropping of tokens and impersonation of clients by using stolen or guessed SSO credentials. The paper founds the basic structure for our analysis of OAuth. At CCS 2014, Chen et al. expanded the analysis field of OAuth to mobile apps [18]. Our analysis is a consequent follow up by transferring the concepts of attacks on OAuth to a new platform: Smart TVs. In contrast to mobile devices, the app concepts for Smart TVs are relatively new, and, there are more different app distribution systems (e.g., Samsung and Grundig have their own stores) and apps are often built up using HTML, CSS, and JavaScript only; for example, in contrast to Android apps.

VIII. Conclusions & Lessons Learned

Our work highlights that new technologies like Smart TVs are a valuable target for attackers. We pointed out that best practices like SSL/TLS are not used properly. Smart TVs contain sensitive data that can even lead to a full SSO account compromise revealing health, credit card, or cloud data. Looking on the generality, apps are a crucial component in the current technology market. Volkswagen has announced that they offer connected-car features provided by the Apple Watch app Car-Net.[15] One of Car-Net's features is to remotely locking and unlocking doors.

Learning from the attacks described in this paper, the Smart TV and app developers should consider the following best practices:

SSL/TLS. Due to the high media focus on POODLE, BEAST, and CRIME, we have expected that apps are protected with TLS [19]. It was was surprising that some apps like WATCHEVER do not even protect important login procedures. In general, sensitive data should only be submitted through an encrypted connection.

Saving Sensitive Data. Samsung is using OAuth for accessing resources. It is surprising that they save the user's SSO account in the Smart TV's DOM. We expected that they use the user's credentials to get an OAuth token; this token could be saved on the TV. Another example is the `UDBCOMMON` file which saves common user inputs including passwords. Saving user inputs might be useful in cases like autocomplete features. However, this is critical in the case of passwords. To sum it up, important sensitive data must not be stored on the TV.

[14]Saving Files, *Android*, http://developer.android.com/training/basics/data-storage/files.html, Apr. 2015

[15]http://media.vw.com/release/977/

File System Access. We were surprised by the fact, that we could get *full* file system access on Samsung TVs. We would have expected a sandboxing/chrooting mechanism on a per-app basis comparable to Android or iOS apps. Even a simple discretionary access control (DAC) is missing: Our app is able to read all files, including system files, on the TV. This flaw shows how important data separation is.

Acknowledgements

The research was supported by *3curity GmbH* and the *German Ministry of research and Education (BMBF)* as part of the `VERTRAG` research project.

References

[1] S. David Watkins, "Global connected tv device tracker: Q4 2014," Online, http://is.gd/SIWJJx, March 2015.

[2] R. van der Meulen and J. Rivera, "Gartner predicts live video broadcasting will be the new selfie by 2017," Online, http://www.gartner.com/newsroom/id/2934717, December 2014.

[3] R. Briel, "Samsung global smart tv market share reaches 26%," http://is.gd/Rsno0k, February 2014.

[4] T. Dierks and E. Rescorla, "RFC 5246 - The Transport Layer Security (TLS) Protocol Version 1.2," Tech. Rep., Aug. 2008. [Online]. Available: http://tools.ietf.org/html/rfc5246

[5] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. (2008, November) Extensible markup language (xml) 1.0 (fifth edition). [Online]. Available: http://www.w3.org/TR/REC-xml/

[6] T. D. Morgan and O. A. Ibrahim, "Xml schema, dtd, and entity attacks: A compendium of known techniques," Online, http://www.vsecurity.com/download/papers/XMLDTDEntityAttacks.pdf, 2014.

[7] specs@openid.net, "OpenID Authentication 2.0 – Final," Dec. 2007. [Online]. Available: https://openid.net/specs/openid-authentication-2_0.html

[8] Organization for the Advancement of Structured Information Standards, "Security assertion markup language (saml) v2.0," 2005.

[9] R. Hansen and J. Grosmann, "Clickjacking," Online, http://www.sectheory.com/clickjacking.htm, May 2015.

[10] L.-S. Huang, A. Moshchuk, H. J. Wang, S. Schecter, and C. Jackson, "Clickjacking: Attacks and defenses," in *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX, 2012, pp. 413–428. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/huang

[11] "Read it twice! a mass-storage-based tocttou attack," in *Presented as part of the 6th USENIX Workshop on Offensive Technologies*. Berkeley, CA: USENIX, 2012. [Online]. Available: https://www.usenix.org/conference/woot12/workshop-program/presentation/Mulliner

[12] M. Herfurt, "Security issues with hybrid broadcast broadband tv (hbbtv)," in *30th Chaos Communication Congress*, 2013.

[13] ——, "Security concerns with hbbtv," https://mherfurt.wordpress.com/2013/06/01/security-concerns-with-hbbtv/, June 2013.

[14] M. Ghiglieri, F. Oswald, and E. Tews, "Hbbtv - i know what you are watching," Online, May 2013.

[15] Y. Oren and A. D. Keromytis, "From the aether to the ethernet—attacking the internet using broadcast digital television," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 353–368. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/oren

[16] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri, "Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: ACM, 2014, pp. 66–77. [Online]. Available: http://doi.acm.org/10.1145/2660267.2660275

[17] K. Chatfield, V. Lempitsky, A. Vedaldi, and A. Zisserman, "The devil is in the details: an evaluation of recent feature encoding methods," 2011. [Online]. Available: http://eprints.pascal-network.org/archive/00008315/

[18] E. Chen, Y. Pei, S. Chen, Y. Tian, R. Kotcher, and P. Tague, "Oauth demystified for mobile application developers," *CCS14*, 2014.

[19] C. Meyer and J. Schwenk, "Sok: Lessons learned from ssl/tls attacks," in *Information Security Applications*. Springer, 2014, pp. 189–209.