Read It Twice! A mass-storage-based TOCTTOU attack

Collin Mulliner and Benjamin Michéle
Security in Telecommunications
Technische Universität Berlin and Telekom Innovation Laboratories
{collin,ben}@sec.t-labs.tu-berlin.de

Abstract

Consumer electronics and embedded devices often allow the installation of applications and firmware upgrades from user-provided mass-storage devices. To protect the integrity of these devices and the associated electronic markets, the software packages are protected by cryptographic signatures. The software installation code assumes that files on attached mass-storage devices cannot change while the storage device is connected. The software installation is therefore not bound to the file integrity check, thus laying the foundations for a time-of-check-to-time-of-use (TOCTTOU) attack. This work presents a TOCTTOU attack via externally attached mass-storage devices. The attack is based on emulating a mass-storage device to observe and alter file access from the consumer device. The TOCTTOU attack is executed by providing different file content to the check and installation code of the target device, respectively. The presented attack effectively bypasses the file content inspection, resulting in the execution of rogue code on the device.

Keywords: *race condition, USB, mass-storage, consumer electronics, software attestation*

1. Introduction

Consumer electronics today are heavily targeted by the hacking and modding community with the primary goal to modify or replace the software running the devices. To fulfill this goal the attacker (the modder) has to execute his own code on the target device. In most cases the attack further needs to gain system or root privileges on the target device.

There are many ways to achieve code execution and firmware replacement mostly depending on the type of device and software running on it. Low cost devices are mostly not hardened against hacking and modding. Here the effort mostly comes down to figuring out file and firmware formats or finding the serial console on the hardware. More costly devices contain more sophisticated security measures. Here firmware upgrades are protected by cryptographic signatures. Attacking the more costly and thus protected devices comes down to finding and exploiting software bugs to achieve code execution often requiring a lot of effort.

Many embedded systems and especially consumer electronics (CE) support the installation of software and firmware upgrades through attached mass-storage devices. Most commonly, USB mass-storage devices are used for this, such as flash drives and hard disks. Depending on the type of embedded system, Secure Digital (SD) and Compact Flash (CF) cards are also popular.

In this paper we present a novel time-of-check-to-time-of-use (TOCTTOU) attack that targets file content. We attack software installation and firmware upgrade code that reads files from an external mass-storage volume. Our attack is based on an emulated mass-storage device that allows to change the content of files while the mass-storage volume is connected to the attacked target.

Our attack method is based on a number of observations that are present on many different consumer electronics devices today. The main observation is that code for software installation and firmware upgrade is separated into two parts: *check* and *install*. If each part implements its own file access it is potentially prone to a TOCTTOU attack.

This work demonstrates a practical implementation of such an attack against a Linux-based TV-set. We show that we are able to install a shared library on the system, which is then loaded by the main application running on the TV-set. Our code runs with root privileges. Our attack currently is the only method to root a specific series of Samsung TV-sets. We further present a tool to analyze the behavior of CE devices to deter-

mine if a device might be susceptible to mass-storage-based TOCTTOU attacks.

Similar issues exist in the areas of trusted computing and software attestation. One party tries to verify or measure the integrity of another party (the other partie's code) before accessing or using it. If a time window between measurement and access exists, the software attestation might be vulnerable to a TOCTTOU attack.

The contributions of this paper are the following:

- **Read It Twice (RIT) Attack** which is a mass-storage-based TOCTTOU attack based on the condition that software installation and firmware upgrade code are separated into two parts: check and install. If each code part individually reads file(s) from an external mass-storage device an exploitable TOCTTOU condition might exist. Our attack also specifically accounts for a possible existing block and file system cache on the target device. Our approach is different from traditional TOCTTOU attacks as we target the content of files and not their permissions.
- **USB-Mass-Storage RIT Attack Implementation and Evaluation** against a Samsung TV-set. Using this attack we were able to gain code execution and root privileges on our target device.
- **Mass-Storage File Access Analysis Method** and tool for black box investigation of file access to external mass-storage devices. This analysis method allows to detect possible TOCTTOU conditions in firmware upgrade and software installation code of embedded systems that read files from external mass-storage devices.

The rest of this paper is organized as follows. In Section 2 we introduce our novel *Read It Twice* attack. In Section 3 we provide a brief overview of our target, a TV-set. Section 4 presents our Mass-Storage File Access Analysis method and tool. Our method is general and can be used for black box analysis of arbitrary devices that read files from a USB mass-storage device. In Section 5 we present a practical implementation of our RIT attack against the software installation subsystem of our TV-set. In Section 6 we discuss related work and in Section 7 we briefly conclude.

2. The Read It Twice Attack

Our *Read It Twice* (RIT) attack is based on the observation that software installation and firmware upgrade code on embedded devices assumes that files on an attached mass-storage device will not change during the installation. In general, an installation consists of the following steps:

Check the software package or firmware upgrade. This step verifies version numbers and cryptographic signatures of the packages that are going to be installed.

Install the actual software or firmware upgrade. This step copies the files from the external storage device to the internal storage or flashes the firmware.

The check and installation phases are not combined in an atomic operation as file contents are assumed to be immutable while the mass-storage device is plugged-in.

Our RIT attack works as follows:

Given the *file-X* that is expected by the check-install code. We have the benign *file-B* and the modified version *file-M*. We construct a mass-storage device that can observe the read requests to *fileX*. For the first access to *fileX* our mass-storage device serves the benign *file-B*. This is likely the check code that calculates and compares the cryptographic hashes or verifies other parameters contained in *fileX*. For the second read access to *fileX* our mass-storage device serves our modified *file-M*. This is likely the installation phase of software install code.

The attack succeeds if the check code verifies the signature of the benign file *file-B* and then the install code uses the modified file *file-M*. Effectively our attack circumvents the signature check and/or file content inspection.

2.1. Boundary Conditions

There are two boundary conditions for our RIT attack that have to be resolved in real world implementations. These are:

File size of the benign file and the modified file likely need to be equal. Further the filesystem usage must be exactly the same to guarantee that both files are located within the same blocks in each filesystem image.

Block cache. Embedded devices running sophisticated operating systems such as Linux, BSD, and Windows implement a block cache. If the target file fits in the block or filesystem cache the attack has to be adjusted so that the install code will read the file from the attached device rather than from the block cache.

Another boundary condition is that the target device does not copy the software package or firmware upgrade file to internal memory before checking it. This is an obvious countermeasure for our attack that we briefly discuss in Section 5.3. In the remainder of this paper we will discuss these boundary conditions and how we dealt with them to successfully launch our attack.

3. The Samsung TV-set

Our target TV-set is the Samsung LE32B650T2-PXZG [13], a 32 inch version of the Samsung B series LCD-based television set. We chose this TV-set because this series has a very active modding community called SamyGO [15]. Through this community many technical details of this line of Samsung TVs are available to the general public. Our TV model has a Common Interface Plus (CI+) [1] slot for Conditional Access Modules (CAM). CI+ was developed to protect digital video broadcasts and offers a protected data path between the CAM and the TV set. Vendors put a lot effort in securing these TV sets as a compromise is likely to give access to decrypted video broadcasts.

The TV-set consists of a display and a computing unit. The computing part is an ARM-based Linux system. Besides the audio and video interfaces such as HDMI, SCART, and an antenna plug the TV-set features an Ethernet interface and multiple USB interfaces. The USB interface can be used to plug in a USB WiFi adapter (to replace the Ethernet connection) or for connecting USB storage devices such as USB flash disks or hard disks. The TV-set is able to play back audio and video files from the USB storage devices.

This TV is equipped with 290MB of RAM and a total of approximately 650MB of flash memory for permanent storage.

This line of Samsung TV-sets is one of the first that offers installation of widgets and games. On the TV-set these features are accessible through the *Content* menu. The content subsystem can launch two kinds of executables: Adobe Flash files and native code loaded as a shared object (a simple `.so` file). Content packages can be executed and installed from a USB drive.

The busybox-based Linux system executes a large binary called `exeDSP`. This binary controls the entire TV-set. It is responsible for showing the On Screen Display (OSD) to navigate through the TV channels, changing the TV settings, interacting with UPnP servers, and for accessing the content applications. The `exeDSP` application runs as user `root`, i.e., with full privileges.

```
 1 <?xml version="1.0" encoding="utf-8"?>
 2 <contentlibrary>
 3  <contentpack id="tocttou">
 4  <category>Wellness</category>
 5  <title language_id="English">tocttou</title>
 6  <startpoint language_id="English">
 7   tocttou.so</startpoint>
 8  <thumbnailpath>tocttou.bmp</thumbnailpath>
 9  <totalsize>1</totalsize>
10 </contentpack>
11 </contentlibrary>
```

Figure 1. Example `clmeta.dat` file that we used for testing and the RIT attack.

3.1. Samsung TV Software Packages

Software packages with content for the Samsung TV consist of a minimum of three files [14]: The executable code (Adobe Flash or a shared object), a bitmap (the application icon), and the package description in the `clmeta.dat` file. Figure 1 shows an example of such a package description. The important values in the `clmeta.dat` file are the `startpoint` and the `category`. The startpoint specifies which file contains the executable code; in this example it is the shared object `tocttou.so`. The category specifies the kind of application. The TV-set recognizes categories such as *Game*, *Children*, and *Wellness* among others.

Interestingly, the category implies the kind of executable code expected to be contained in the package. Games come in the form of shared objects while packages belonging to the other categories are Adobe Flash-based.

Every game's shared object has to provide a single function called *Game_Main* [14] that is called once the shared object is loaded by the `exeDSP` process. Applications can be developed in pure C and by using the Simple DirectMedia Layer (SDL) [2] library that is pre-installed on the Samsung TV-sets. Figure 2 shows the code for a very simple exemplary plugin.

3.2. The SamyGO Jailbreak

Previously, before the introduction of CI+ [1] devices, Samsung TV-sets allowed to execute and install

```
int Game_Main(char *path, char *udn)
{
   system("telnetd &");
   return 0;
}
```

Figure 2. Simple Samsung content library application that executes system(3).

applications based on shared objects from USB mass-storage devices. Execution of an application is straightforward: The application was selected and run from the content menu. The only requirement for installing an application on the TV-set was to enable write access to the TV-set's internal memory via the *WiseLink Write* option in the TV configuration [16]. Once the application was installed, the shared object could be loaded by the `exeDSP` process. As `exeDSP` is run with root privileges, code from the user-supplied shared object is executed with root privileges, too, hence taking over control of the TV system. This enabled the installation of custom applications like a *telnetd* server or flashing of customized firmware versions to the TV set.

Flashing a customized firmware involves additional steps, which are not important in the scope of this work. They can be found on the SamyGO Wiki [15].

4. Mass-Storage File Access Analysis

To determine if a specific device is susceptible to our RIT attack the behavior of its check and install code has to be analyzed. We developed a Mass-Storage File Access Analysis tool that allows black box analysis of the access behavior to USB mass-storage devices.

Our analysis tool is implemented using a gumstix [10] board running Linux. The gumstix board is equipped with a USB OTG [4] port and thus can emulate a USB client device. The Linux USB stack already has support for USB mass-storage emulation through the gadget API [3]. The driver source is in `file_storage.c` (in `linux/drivers/usb/gadget/`), which compiles as the kernel module `g_file_storage.ko`.

The g_file_storage module works in a simple way. At module load time, the filename of a filesystem image or block device is passed to the module as a parameter. The module exports the given file or block device as a USB mass-storage volume. Each block requested by the host is read from the file by the module and sent back to the host via USB.

Our tool works as follows:

g_file_storage.ko operates as designed. The given file is exported as an emulated mass-storage volume. Blocks requested by the host are read from the file and sent back to the host.

Block and file system access tracking. We track every block read access and match the file and directory associated to that block. This allows us to monitor which files are accessed by the host and how often a given file is actually read from our emulated block device. We implemented this for FAT16 and FAT32 [12] as these are the common filesystems for our target devices. Other filesystem types can easily be added if required.

Our tool enables us to conduct black box analysis of USB mass-storage-based firmware upgrade and software installation code running on embedded systems such as CE devices. We were able to gather the following information about our Samsung TV-set by using this tool.

File and directory access during the check and install phase of the target device. This allows us to identify the files actually being accessed by the check and install code during a firmware update or software package installation.

Files read by the check and install code. This discloses if files are read completely or only partially.

Timing of file access. The time of each block access is logged accurately, thus allowing the observation of delays between accesses to consecutively read files. This can provide hints about processes such as signature checks after having read one file and before reading the next one.

The output of our analysis tool provides a great starting point for designing our attack. Figure 3 shows an example output for the installation of a content library application on our Samsung TV set.

This output allows to deduce some of the TV set's internal functioning by matching user interface interactions to file access patterns. After invoking the USB inspection menu of the content library (11:18:56), the TV set scans each directory for a `clmeta.dat` file. Each of these files is read to populate entries in each of the content categories, being *Wellness* in our example software package.

As the user opens the *Wellness* category (11:19:10), the TV set reads the corresponding bitmap file of each package as indicated in the `clmeta.dat` file. These

```
11:18:56 TOCTTOU     (DIR)
11:18:56 CLMETA.DAT  (471b) [/TOCTTOU]
11:18:56 CLMETA.DAT  -> read completed!
11:18:56 CACHE       (DIR)
11:18:56 CLMETA.DAT  (450b) [/CACHE]
11:18:56 CLMETA.DAT  -> read completed!
11:19:10 CACHE.BMP   (843758b) [/CACHE]
11:19:10 CACHE.BMP   -> read completed!
11:19:10 TOCTTOU.BMP (490734b) [/TOCTTOU]
11:19:10 TOCTTOU.BMP -> read completed!
11:19:56 TELNETD     (1745016b) [/TOCTTOU]
11:19:56 TELNETD     -> read completed!
11:19:56 TOCTTOU.SO  (4608b) [/TOCTTOU]
11:19:56 TOCTTOU.SO  -> read completed!
```

Figure 3. The output shows files and directories being access. In addition it shows that files are being read completely and the time at which the files have been accessed.

bitmaps are then displayed to the user as the package's icon.

Finally, the user selects an application to be installed (11:19:56), which in our example is the *TOCTTOU* package. The package content, i.e., the entire directory, is then copied from the mass-storage device to the internal flash memory. This includes the executable files `telnetd` and `tocttou.so`.

After installation, the user can launch the freshly installed application from internal memory by selecting it in the content menu. Each time the application is launched, the TV set will analyze the `clmeta.dat` to choose how to launch the application, i.e., as Adobe Flash or as a shared object.

Note that the `clmeta.dat` and the bitmap file is not read again as it is copied to the internal memory. This indicates that these files are copied directly from the internal block cache and that no second access to the block device occurred (cf. Section 2.1).

Based on the results of our analysis we design and implement our RIT attack and corresponding tool, which we present in the following Sections.

5. The RIT Attack

Our Samsung LE32B650 TV, like all current Samsung CI+ TV-sets, does not allow to execute or copy applications based on shared objects from a USB drive. Only Adobe Flash-based applications may be executed and copied from USB drives. But pre-installed games such as *WiseStar* are based on the shared object interface, therefore, shared object-based applications still seem to be supported.

Our RIT attack is based on the observation that shared object applications are supported while only Flash applications can be copied to the device. The goal of our attack is to trick the TV-set into copying an application that is based on a shared object to the TV-set's internal memory from which it can be executed.

The software check-install code determines the kind of application by inspecting the clmeta.dat file, as shown in Figure 1 and described in Section 3.1. The categories *Wellness* and *Children* denote Adobe Flash-based applications that may be installed, i.e., copied to internal flash memory. The *Game* type application is based on a shared object and thus may not be installed. We call this first part the *check* code as described in Section 2.

The *install* code that actually copies the application files to the TV-set's internal memory is not checking the `clmeta.dat` file and just copies the whole subdirectory. If we can change the `clmeta.dat` file from category *Wellness* to category *Game* our tocttou application will change to a shared object-based application. As a result we will have our code running inside the exeDSP process which runs with root privileges.

Note that the execution of code from shared object files is prohibited only for files from external storage. Once this code is executed from internal storage, no restrictions apply. Therefore, an attacker has to persuade the check-install code to copy the shared object file and a corresponding `clmeta.dat` to internal memory.

The remainder of this section describes our attack tool and how it is leveraged to perform our RIT attack against the Samsung TV set.

5.1. The Attack Tool

Our attack tool is an extension of our analysis tool, basically two features were added. These are:

Trigger file monitoring. Our version of g_file_storage takes an additional filename as the *trigger file*. Every time the *trigger file* is read from the block device the *trigger counter* is incremented. The match of block access to filename is done via our block and filesystem access tracking code that we added for our analysis tool (cf. Section 4).

Filesystem switch. When the *trigger counter* reaches the *trigger value*, which is passed as a parameter to our version of g_file_storage, all block

requests are redirected to the modified filesystem image. Effectively, the volume is switched to the modified filesystem image.

Hence, our version of the g_file_storage module requires three additional parameters: the path to our modified filesystem image, a filename as the *trigger file*, and the *trigger value* to switch between the original and the modified filesystem image.

5.2. Executing the Attack

To execute the attack we require two FAT filesystem images. The first image contains the benign filesystem (compare file-B in Section 2). The second image contains the modified filesystem (file-M). Both filesystems contain exactly the same files with the exception of the `clmeta.dat` file. In the modified filesystem the `clmeta.dat` file of our tocttou application specifies the category *Game* whereas it is *Wellness* in the benign image.

Note that the two filesystem images are required to be completely identical as the host will use the FAT of the benign image to request blocks of the modified image. This can be achieved by first creating the benign image and modifying a copy thereof to create the modified image. Alternatively, copying the files in exactly the same order to both images will yield suitable images.

To execute the attack we have to further ensure that the `clmeta.dat` file is actually read multiple times. The problem is, as described in Section 2.1, that the TV runs Linux and thus has a block cache. Therefore, the TV will read each block from the external storage device only once and then store it in the block cache. To resolve this issue and to force the TV to re-read the `clmeta.dat` file from our tocttou application, we added a second application named *cache*. The second application is basically a copy of our tocttou application with a different name. To circumvent the block cache, we padded the `clmeta.dat` file from the cache application with spaces until it hit 260 Megabytes. When the TV-set reads this large file all previously cached blocks are discarded. The file size of 260MB was determined by experimenting with various reasonable values, i.e., common RAM sizes. Using our block access tracking code we can easily observe if the blocks of the `clmeta.dat` from our tocttou application are requested again by the host. If this happens, the blocks are not in the block cache anymore and our attack succeeds.

Figure 4 shows the basic concept of our attack. The tool's output for a successful RIT attack is shown in Figure 5.

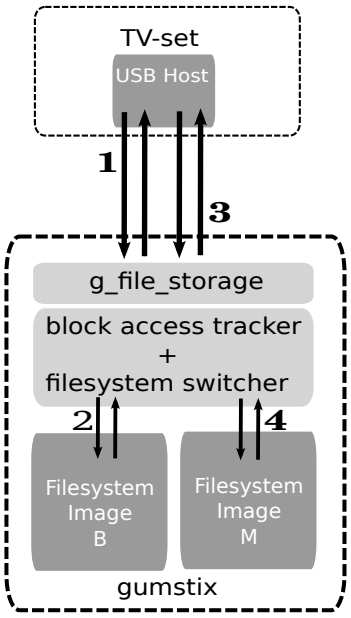

Figure 4. Our attack in 4 steps: 1) The TV-set request block 23 from our emulated storage device. 2) Our tool reads block 23 from the benign filesystem image and delivers it to the TV-set. 3) At a later point, the TV-set again requests block 23 form the storage device. 4) This time our tool reads block 23 from the modified filesystem image and delivers the block to the TV-set.

5.3. Countermeasures

There are many possibilities for countermeasures against our RIT attack; here we only discuss one simple solution that is straightforward.

The most obvious and easiest to implement is to first copy the software or firmware files to internal storage and then execute the checks on the copy. After all checks have passed, the installation process is executed using the internal copy of the files. This countermeasure is only possible if the target device has enough unused storage space. Alternatively, the copy could also be held in free memory (RAM), if available.

This simple copy-based countermeasure will only work for devices that contain excess storage or system memory. Consumer electronics devices are mostly built to be cheap and use as few resources as possible, often having just enough storage to support the operational functionalities of the device. More sophisticated countermeasures have to be developed to protect devices that do not contain enough memory to implement a simple copy-based check-install procedure.

```
 1 TOCTTOU     (DIR)
 2 CLMETA.DAT  (471b) [/TOCTTOU]
 3 CLMETA.DAT  -> read completed! [1/2]
 4 CACHE       (DIR)
 5 CLMETA.DAT  (272630223b) [/CACHE]
 6 CLMETA.DAT  -> read completed! [2/2]
   [device switched!]
 7 CACHE.BMP   (843758b) [/CACHE]
 8 CACHE.BMP   -> read completed!
 9 TOCTTOU     (DIR)
10 TOCTTOU.BMP (490734b) [/TOCTTOU]
11 TOCTTOU.BMP -> read completed!
12 TELNETD     (1745016b) [/TOCTTOU]
13 TELNETD     -> read completed!
14 TOCTTOU.SO  (4608b) [/TOCTTOU]
15 TOCTTOU.SO  -> read completed!
16 CLMETA.DAT  (471b) [/TOCTTOU]
17 CLMETA.DAT  -> read completed! [3/2]
```

Figure 5. Output of our attack tool for the RIT attack. In line 6 the device is switched to the modified filesystem image after the access to the clmeta.dat file in the cache directory. The clmeta.dat in the cache directory is 260MB in size to flush tocttou/clmeta.dat from the block cache of the TV-set. The attack is finalized in line 16 where the TV-set reads clmeta.dat again but this time the block are read from the modified filesystem image.

6. Related Work

Related work falls into four different areas. First, general race conditions and file-based TOCTTOU attacks, second USB mass-storage-based attacks, third USB-based attacks, and fourth trusted computing and software attestation.

Bishop et al. [7] characterized and analyzed a similar class of TOCTTOU attacks that targets file access on UNIX systems. Their work mainly targeted file permission checks versus file open operations. In this paper we present a novel TOCTTOU attack against the *content of files*.

The PSjailbreak and the open source version PSGroove [6] attack to jailbreak Sony's PlayStation 3 is based on a rogue USB device. The exploit leverages a memory corruption bug in the PlayStation USB driver code that parses USB device descriptors. The attack works by emulating multiple USB devices with malicious content in the device descriptors. The attack allowed arbitrary code execution on the PlayStation 3.

In [17] the authors added malicious functionality to the USB drivers of an Android-based smartphone to attack the attached computer while the phone was connected for charging its battery. The malicious functionality emulated a keyboard and mouse to interact with the computer.

There are various attacks based on USB mass-storage devices [8, 11] that contain autorun files for either Windows or Linux.

The GTV Hacking scene [5] found that the Sony Google TV could be rooted and downgraded by a simpler version of our RIT attack. Their attack is based on three USB flash drives that contain different versions of firmware and firmware meta files. The attack works by inserting and removing the USB flash disk during a firmware upgrade. Since the Sony GTV software doesn't seem to check if a device is inserted and removed during the upgrade process this simple attack is able to downgrade the firmware version. After the downgrade the device can be rooted since the older version contains a software vulnerability that allows shell command injection. Our attack is similar but targets more hardened systems that can detect removal of the upgrade medium such as the USB flash drive.

Our work also falls into the area of trusted computing and software attestation. One of the challenges in these areas is to prove the integrity of data stored in the off-chip memory, i.e., memory not part of the actual CPU. [9] presents a survey on memory authentication mechanisms and attacks that also deal with external memory and storage. Our attack relies on the ability to modify the content of an external storage device that is connected to the target system.

7. Conclusions

In this work we presented a novel time-of-check-to-time-of-use (TOCTTOU) attack that targets the content of files based on emulated mass-storage devices. Our attack is called Read It Twice! (RIT). We designed and implemented a USB mass-storage-based version of our novel TOCTTOU attack to inject a shared object into a Samsung TV-set bypassing the implemented security checks. The shared object is executed with root privileges thus it is effectively rooting or jailbreaking the device.

We believe that our RIT attack applies to many kinds of consumer electronics and embedded systems that install software packages from external storage devices, given the same boundary conditions apply. Mainly the check and install code must not be bound together.

Although we developed our proof-of-concept attack for USB-based mass-storage, we believe the attack transfers to any kind of flash and disk technology as long as the block access can be observed and altered. A prime example of another kind of external storage device is a Secure Digital (SD) or Multi Media Card (MMC) card. Any storage device technology that can be emulated can be used to carry out our RIT attack. Storage devices containing a sophisticated processor may be modified via software to carry out a RIT attack without emulation but directly returning different data for multiple read operations of the same block.

In the future, manufacturers need to take special care when reading sensitive data from external attached mass-storage devices. The device could be emulated to carry out an attack such as our *Read It Twice!* attack.

Acknowledgements

The authors would like to thank Patrick Stewin, Dmitry Nedospasov, and Jean-Pierre Seifert for reviewing early versions of this paper.

References

[1] CI Plus (CI+). `http://www.ci-plus.com`.

[2] Simple DirectMedia Layer. `http://www.libsdl.org`.

[3] USB gadget API of the Linux kernel. `http://kernel.org/doc/htmldocs/gadget.html`.

[4] USB on-the-go. `http://www.usb.org/developers/onthego/`.

[5] About Sony Downgrade + Rebooter (Root). `http://gtvhacker.com/index.php/About_Sony_Downgrade_+_Rebooter_(Root)`, February 2012.

[6] Y. Alaoui. PSGroove. `https://github.com/psgroove/psgroove`, December 2010.

[7] M. Bishop, M. Bishop, and M. Dilger. Checking for Race Conditions in File Accesses. *Computing Systems*, 9:131–152, 1996.

[8] A. Crenshaw. Plug and Prey: Malicious USB Devices. `http://www.irongeek.com/downloads/Malicious%20USB%20Devices.pdf`, January 2011.

[9] R. Elbaz, D. Champagne, C. H. Gebotys, R. B. Lee, N. R. Potlapally, and L. Torres. Hardware mechanisms for memory authentication: A survey of existing techniques and engines. *Transactions on Computational Science - Special Issue on Security in Computing*, 4:1–22, 2009.

[10] Gumstix Inc. gumstix. `http://www.gumstix.com/`, 2012.

[11] J. Larimer. USB autorun attacks against Linux. `http://blogs.iss.net/archive/papers/ShmooCon2011-USB_Autorun_attacks_against_Linux.pdf`, January 2011.

[12] Microsoft Corporation. Microsoft Extensible Firmware Initiative FAT32 File System Specification, FAT General Overview On-Disk Format. `http://msdn.microsoft.com/en-us/windows/hardware/gg463084`, December 2006.

[13] Samsung Inc. LE32Bxxx LCD TV. `http://www.samsung.com`.

[14] SamyGO. Creating Content Library applications - SamyGO. `http://wiki.samygo.tv/index.php5/Creating_Content_Library_applications`.

[15] SamyGO. SamyGO, Samsung Firmware on the GO. `http://www.samygo.tv`.

[16] SamyGO. Service Menu Enabling Add/Delete in Content Manager - SamyGO. `http://wiki.samygo.tv/index.php5/Service_Menu\#Enabling_Add.2FDelete_in_Content_Manager`.

[17] Z. Wang and A. Stavrou. Exploiting smart-phone USB connectivity for fun and profit. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 357–366. ACM, 2010.