Persistence in Linux-Based IoT Malware

Calvin Brierley, Jamie Pont, Budi Arief, David J. Barnes, and Julio Hernandez-Castro

School of Computing, University of Kent, Canterbury, England
{C.R.Brierley, J.Pont, B.Arief, D.J.Barnes, jch27}@kent.ac.uk

Abstract. The Internet of Things (IoT) is a rapidly growing collection of "smart" devices capable of communicating over the Internet. Being connected to the Internet brings new features and convenience, but it also poses new security threats, such as IoT malware. IoT malware has shown similar growth, making IoT devices highly vulnerable to remote compromise. However, most IoT malware variants do not exhibit the ability to gain *persistence*, as they typically lose control over the compromised device when the device is restarted. This paper investigates how persistence for various IoT devices can be implemented by attackers, such that they retain control even after the device has been rebooted. Having persistence would make it harder to remove IoT malware. We investigated methods that could be used by an attacker to gain persistence on a variety of IoT devices, and compiled the requirements and potential issues faced by these methods, in order to understand how best to combat this future threat. We successfully used these methods to gain persistence on four vulnerable IoT devices with differing designs, features and architectures. We also identified ways to counter them. This work highlights the enormous risk that persistence poses to potentially billions of IoT devices, and we hope our results and study will encourage manufacturers and developers to consider implementing our proposed countermeasures or create new techniques to combat this nascent threat.

Keywords: IoT · security · malware · persistence · attack · proof of concept

1 Introduction

A standard piece of advice typically given to affected users for removing malware from an Internet of Things (IoT) device is to restart it, as most forms of IoT malware lack the ability to maintain *persistence* [3, 4]. This is because, in general, IoT malware is stored and executed from within temporary filesystems that reside in Random-Access Memory (RAM) [32]. As this type of memory is volatile, the stored programs and data are lost when the device loses power, including any changes that the attacker may have made to the filesystem.

However, there have been some families of IoT malware that are able to maintain persistence in some form [15, 27]. If persistent IoT malware becomes more prevalent, many IoT devices will not be recoverable at all once they have been infected. Therefore, it is increasingly crucial for IoT developers both *to understand their devices' potential vulnerabilities to persistence* and *to implement preventative measures to prohibit attackers from exploiting them.* These two aims serve as the motivation for our work.

Contributions. The key contributions of this paper are:

- We summarise IoT persistence and its role in IoT based malware.
- We explain the challenges currently preventing IoT malware from establishing persistence.
- We outline methods that could be used by IoT malware to gain persistence.
- Finally, we explore how this will change the approach of IoT malware and how attackers could achieve and use persistence to perform new and previously infeasible attacks, and what can be done to counter this threat.

The rest of the paper is organised as follows. Section 2 provides some background on Linux malware, IoT based malware and persistence. We also highlight previous research and some of the challenges attackers may encounter when attempting to gain persistence on IoT devices. Section 3 describes several methods that could be used by attackers to gain persistence on various types of IoT device, along with their requirements and limitations. Section 4 shows the results of attempting to gain persistence on four vulnerable IoT devices using these methods. Section 5 discusses some potential countermeasures that could be implemented to prevent an attacker from gaining persistence on an IoT device. Section 6 covers our conclusions and defines some recommended further work.

2 Background

Various families of malware have increasingly attacked IoT devices. Popular botnets such as Bashlite and Mirai have infected hundreds of thousands of devices and have been responsible for one of the largest DDoS attacks in history [33, 11]. Fortunately, this type of IoT malware is relatively simple to remove. By restarting the device, the malware will be unloaded from volatile memory, removing the infection from the device when it reboots [3, 4].

However, some malware (such as Mirai) often exhibits worm-like behaviour [5] and after hijacking a device, it will scan the Internet for more victims to infect. While users would sometimes restart their devices (either deliberately or coincidentally) and clear the infection, it would not remove the underlying issue. The devices could easily be reinfected, possibly within minutes [3].

In effect, this behaviour has led to competitions between botnet authors, each seeking to maximise their share of the limited number of vulnerable IoT devices. Some malware even exhibited security features to remove competing malware. Mirai, for instance, would search for strings present in competing malware, kill any associated running processes and close any potentially vulnerable services running on specific ports to prevent any further attacks by competitors [5]. However, as these changes were not persistent they were removed when the device was reset.

2.1 Persistent IoT Malware

IoT malware capable of making persistent changes that secure its presence would be able to maintain control over the device through reboots, both removing the

requirement to reinfect the equipment and helping towards keeping competitors at bay. The ability to secure persistence would also allow significant changes to the device to persist after rebooting, allowing for more creative types of malware and attacks, such as ransomware [10] or long term spyware. This would also provide a means for the malware operator to install additional malicious features, such as modules that can attack other devices on the infected device's network.

Currently, restarting an infected device will remove the majority of IoT malware, but with persistence, the user would have to modify the flash memory of the device to remove the infection. This is something not usually readily available nor practical to an average user. If the malware can also prevent updates or factory resets, specialist equipment or access to a debug/programming interface may be required to clear the infection. This is considered too complicated for most IoT users to perform and may lead to IoT devices being discarded, or worse, knowingly left in an infected state.

2.2 Challenges With Gaining Persistence

There are two key challenges currently faced by IoT attackers when attempting to gain persistence on IoT devices:

- **Read-Only**. IoT devices often have data that is set to read-only for various reasons, such as to prevent accidental modifications due to programmer error. This feature may also prevent attackers from making the necessary modifications to the stored data or filesystems in order to gain persistence.
- **Variance**. Each device is likely to have different hardware, update mechanisms, software, architecture and filesystem types. Fortunately for IoT developers, the variation in IoT devices makes it quite difficult for attackers to create a universal method for gaining persistence. However, if an attacker were to develop a method that affects a large number of devices with similar implementation, it could reduce the required time investment immensely, leading to persistent IoT Malware becoming more common.

2.3 Previous Persistent IoT Malware and Related Work

After identifying an increase in the presence of Linux based malware, researchers analysed 10,548 samples over a year to gain a better understanding of the techniques used by malware authors [12]. They highlighted the quick development and deployment of insecure IoT devices as a potential motive for attackers to target Linux for malware development.

As part of this analysis, they found that 21.10% (1,644) of the analysed samples implemented persistence methods. Some of these methods can be applied to IoT devices, but the attacker must be able to modify the filesystem. As mentioned previously, IoT devices often set certain data as read only, which would prevent these methods from working.

Some variants of IoT malware have achieved persistence, but these are less common and they rely on the device having a writeable filesystem, which may

reduce the applicability of this approach. We examine two examples of persistent IoT malware below.

Torii is a variant of Mirai that adds several features, most notably the introduction of six techniques to gain persistence [15]. Each technique modifies files on the infected device which are executed as part of the boot process, such as:

- `.bashrc`, which is executed whenever an interactive bash session is started;
- `initab`, which is used to determine which processes should be ran during the Linux boot process at certain runlevels;
- `crontab`, which is used to execute files at a certain time or interval.

Modifications to these files would allow the attacker to set particular programs or shell scripts to be run when the device boots.

VPNFilter is a complex IoT malware which affects a large number of routers [30]. It is believed to have been developed by "Fancy Bear", a Russian based hacker group [31]. Its modular structure allows many features to be implemented, ranging from man-in-the-middle attacks to SCADA sniffing. Additionally, VPNFilter seems to include a section of code to erase and rewrite Memory Technology Devices (MTDs)[1], which could potentially be used to brick the device by wiping segments of the device's storage [28]. VPNFilter modifies the `/etc/config/crontab` file, which will run the malware (which has presumably already been written to memory) every 5 minutes [27, 29], even when the device is rebooted.

3 Methods for Gaining Persistence

Due to the challenges described in section 2.2, no universal methods to gain persistence on IoT devices have yet been identified. Instead, our approach is to use a *collection* of methods to gain persistence on *certain subsets* of IoT devices.

We have identified several viable methods that could be used by an attacker to gain persistence on a variety of IoT devices. A summary of these methods can be found in Table 1 and a detailed overview of each is provided in the following subsections. The description of each method includes a list of *requirements* for its applicability, its *feasibility*, and any *potential issues* that may prevent it from working effectively. A malware writer could perform reconnaissance to ascertain which method should be used, or simply attempt each method sequentially until they gain persistence. Some methods could be used in conjunction with others to improve their chances of success.

The techniques described assume that the attacker has gained access to the shell (such as via a guessable telnet password), and can run arbitrary commands. Ideally, the attacker should be able to determine the storage capabilities of the device and identify the device model. Many of these techniques also require the identification and modification of filesystems and partitions in flash memory. The `/proc/mtd` file contains the partition definition and a name set by the developer via MTD, which may indicate its purpose. These partitions can be accessed by

[1] Memory Technology Devices (MTD) are commonly used to communicate with flash devices to manage storage on IoT devices.

Table 1. IoT Persistence Methods

ID	Method	Modified Partition	Ease of Use
A	Modifying Writeable Filesystems	Filesystem	Easy
B	Recreating Read-Only Filesystems	Filesystem	Medium
C	Initrd/Initramfs Modification	Kernel	Hard
D	"Set Writeable Flag" Kernel Module	N/A	Hard
E	Update Process Exploitation	Filesystem/Kernel	Device Dependent*
F	UbootKit	Bootloader	Hard

*Device update processes differ, so the complexity of exploits will vary.

using the files `/dev/mtd`*X* or `/dev/mtdblock`*X* where *X* is the partition index. The attacker can also find a list of mount points and their filesystem types in the `/proc/mounts` file, or use analytic tools such as Binwalk [25] to identify recognisable file headers and metadata.

3.1 Modifying Writable Filesystems

When an IoT device has a writeable filesystem by default, the attacker should be able to modify the filesystem directly via the shell, allowing them to edit important files that run on startup.

Requirements: The device must use a writable filesystem (e.g. `yaffs2`/`jffs2`). The MTD filesystem partitions must be writeable. The attacker must be able to modify the startup scripts.

Feasibility: This is the simplest method and does not require any additional tools. If the filesystem is writeable by default, the attacker can copy their malware to a known location on the device, then modify the startup scripts so that it is executed when the device is rebooted. This is similar to the technique used by `VPNFilter` and `Torii`, as described in section 2.1.

Potential Issues: The attacker must be able to obtain write permissions for the files they are attempting to modify, which is dependent on the privileges held by the exploited application or compromised account used by the attacker.

Furthermore, the writable filesystem must store files that can lead to the execution of arbitrary code on startup. Otherwise, while the attacker may be able to store malware permanently, they will not be able to set it to run when the device is booted. The filesystem may also be mounted as read only, so additional steps may be required to remount it as writeable.

3.2 Recreating Read-Only Filesystems

If the device is using a compressed read-only filesystem, the attacker will not be able to modify its files directly. Instead, the attacker can use specialised tools to recreate the filesystem.

Requirements: The device must use a compressed read-only filesystem (such as `cramfs`/`squashfs`). The attacker must be able to modify the flash

partition which contains the read-only filesystem. The attacker must have the required software to recreate the filesystem.

Feasibility: While it is not possible to modify files *within* compressed read-only filesystems, it is possible to replace the entire filesystem in flash memory with a modified version. To create a new version of the filesystem the attacker must first obtain the compressed version, which resides in flash memory.

Once the attacker has identified the partition that holds the filesystem, they can use the MTD subsystem to read it from flash to a file, which can then be extracted and modified to their requirements. The attacker can then re-pack it in the correct format. For `squashfs` and `cramfs` filesystems, this requires using the `mksquashfs` and `mkcramfs` utilities respectively. The old version stored in the filesystem partition can then be overwritten via the MTD files in `/dev`.

Potential Issues: Filesystems can vary significantly, even those of the same format. If the replacement filesystem type is different from what is expected by the device, it might not be interpreted correctly, which will lead to a failure during the boot process. For this approach to be practical, the attacker must match the used filesystem as closely as possible.

Read-only filesystems may prove challenging to modify, as it is unlikely that the tools used to build a new filesystem will be included on the exploited device. For device updates, it would be expected that another machine would generate a new filesystem that is then transferred to the device itself. To follow this same philosophy, the attacker would need to copy the filesystem from the infected device to an external computer, then modify it using the required tools. It would then need to be uploaded back to the device for writing. Filesystems are likely to be much larger than the average malware upload, and as they will need to be uploaded to each infected device; this might not scale well if used for a large number of devices.

Alternatively, attackers could compile and upload the required tools for use on the devices themselves. However, as there are likely to be many different filesystem types and device architectures, this may be not easy to manage.

3.3 Initrd and Initramfs Modification

As part of its booting processes, the Linux kernel may utilise an appended `initrd` or `initramfs` filesystem [18]. This is an initial filesystem which allows some setup of the device to be performed before mounting the real filesystem.

Requirements: The device must use an `initrd` or `initramfs` filesystem. The attacker must be able to modify the flash partition that contains the kernel.

Feasibility: First, the attacker must identify the MTD partition that contains the Linux kernel. Once the correct partition has been identified, the attacker must analyse it and determine the offset of the filesystem that is appended to the kernel. After carving out the relevant data, they must save the original kernel and filesystem separately. The attacker can then extract and modify the filesystem to include their required malware. Typically, an `initramfs` filesystem will be contained in a CPIO archive, which will likely also be compressed, and as such, this may require multiple extraction steps. The extraction process must

then be reversed, and the resulting filesystem can then be appended to the original kernel. Finally, this data can be used to overwrite the original kernel flash partition.

Potential Issues: The kernel may be stored on the flash chip as an image for use with a chosen bootloader. This may require the attacker to take additional steps to recreate the image and maintain compatibility with the bootloader, such as the inclusion of image headers that the bootloader may use to boot from the partition effectively. As with Method B: unless the filesystem modifications are performed locally, large amounts of data may need to be transferred via the Internet, which might not scale well.

3.4 "Set Writeable Flag" Kernel Module

MTD can be used to manage partitions of flash memory. Developers may unset the `MTD_WRITEABLE` flag for partitions that are unlikely to need modification, which may also prevent attackers from making modifications that would allow them to gain persistence. This method allows an attacker to re-enable the `MTD_WRITEABLE` flag from within userspace if the requirements are met. While this method may not allow an attacker to gain persistence on its own, it may allow other methods to circumvent the read-only protections that were put in place by the developers.

Requirements: The Linux kernel must support loadable modules. Access to a device's kernel header files or source tree will improve the kernel module's odds of being compatible.

Feasibility: The `MTD_WRITEABLE` partition flag can be difficult to modify from userspace at runtime. However, by using a Loadable Kernel Module (LKM), an attacker could force this flag to be set from kernel space. There are existing kernel modules that have been created to implement this [19, 16].

Kernel modules typically need to be compiled against the targeted kernel source to be compatible. This is normally achieved by having access to either the kernel's headers or source tree [1]. If IoT developers use modified software that falls under the GNU Public License (GPL), they may be required to make the corresponding source code available [26]. The attacker can use this to compile the kernel module for the targeted device.

After compiling and uploading the LKM to the target device, the attacker can use the `insmod` utility to insert the module into the kernel. Once inserted, the module is able to set all MTD partitions to be writeable, after which the attacker can use one of the other techniques to gain persistence.

Potential Issues: If the device's kernel header and source code are unavailable, it may be difficult to compile the LKM such that it remains compatible. However, a defensive IoT tool "HADES-IoT" demonstrated that loadable kernel modules could be compiled without the support of the original developer [9].

The developer may be able to prevent this method from being used by configuring the Linux kernel to verify the signature of kernel modules when they are loaded [2]. The attacker will not be able to forge a signature for kernel modules if they do not have access to the developer's cryptographic keys.

3.5 Update Process Exploitation

Most devices are expected to receive updates over their lifetime, either to provide new user features or patches for security issues. However, vulnerable update implementations can potentially be used to attack the device and gain persistence.

Requirements: The device must implement a vulnerable update function, such that the attacker can forge fake updates. The attacker must be able to access the update function.

Feasibility: If an attacker gains access to a vulnerable update function, they may be able to provide a false firmware update which is accepted by the device. For example, researchers found vulnerabilities in devices produced by Disney [8] and Netgear [7], which allowed them to upload modified firmware. An attacker could use these modified updates to include malware and configuration files such that arbitrary code is run each time the device is booted.

Potential Issues: The requirements for this method are quite niche. It not only requires that the attacker has access to the update process (for which they will likely need to be authorised), but the process itself must also be vulnerable in such a way that the updates are not verified before being implemented.

As the update process will differ from device to device, what may work for one is very unlikely to work on another. The attacker will need to reverse engineer the required format of the update for each targeted device's update process. If the forged update is incorrectly formatted, the update process may be halted, preventing the attacker from gaining persistence.

The attacker could attempt to modify the filesystem of an existing firmware file provided by the developer, but the update process may also need to interpret metadata defined by the developer. As such, the attacker will be expected to recreate the metadata, such as file sizes or checksums. Some tools are available that may assist in this process, such as the "Firmware Mod Kit" [24]. This will not work for all update formats, especially if the developer has obfuscated, encrypted or signed the firmware they make available.

3.6 Ubootkit

`Das U-Boot` (Normally shortened to `U-Boot`), is a universal bootloader designed for use with a variety of embedded devices [14]. It is commonly used in IoT devices to manage the booting process into the main operating system.

Requirements: The device must implement U-Boot as its bootloader. The attacker must be able to modify the bootloader flash partition.

Feasibility: Researchers have produced an attack that demonstrates the creation of persistent root-level access in IoT devices, dubbed "`UbootKit`" [35].

If the filesystem MTD partition is marked as read-only, it may prevent some of the other methods from being used. `UbootKit`, however, targets the bootloader partition. If the bootloader partition is writeable, `UbootKit` can modify `U-Boot` in such a way that when the device is next booted, it will run arbitrary code written by the attacker. `UbootKit` will use this vulnerability to corrupt subsequent boot stages and modify startup scripts during Linux's boot sequence, gaining the ability to make persistent changes.

Table 2. Device Persistence Methods Exploits

Device	Persistence Method(s)	Exploit
Netgear R6250 Router	Recreate Read-Only Filesystem & "Set Writeable Flag" Kernel Module	Command Injection CVE-2016-6277 [21]
D-Link DCS-932L	Initrd/Initramfs Modification	Buffer Overflow CVE-2019-10999 [22]
Yealink SIP-T38G	Modify Writeable Filesystem	Command Injection CVE-2013-5758 [20]
WiPG-1000	Modify Writeable Filesystem	Command Injection CVE-2019-3929 [23]

Potential Issues: The authors of `Ubootkit` state that it can be applied to other devices and architectures than those used in the demonstration [35], but that it would require modification. This technique relies on patching the bootloader and kernel of the device with new shellcode at specific offsets. As the bootloader and kernel will differ slightly on each targeted device model and version, determining the correct shellcode modifications may be time-consuming.

4 Experimental Proof of Concepts and Results

To test the viability of the techniques described in the previous section, we applied them to a range of IoT devices. We chose these devices as they have been known to be vulnerable, with publicly available exploits. Some had also been previously targeted by IoT malware. For persistence to be considered a viable and realistic attack method, the following two constraints were applied:

- No physical access to the device must be required during the process. Persistence must be achievable remotely, preferably over the Internet.
- The method of persistence must allow an attacker to force the device to run a custom application when the device is rebooted.

During our testing, we examined some local files on the device that are commonly found on Linux based systems to gather information about the device, such as `/proc/mtd` to identify partitions and `/proc/mounts` to identify filesystems. These would help determine the best technique to apply when attempting to gain persistence on that device.

4.1 Netgear R6250 Router (using Methods B and D)

The Netgear R6250 router is one of many routers that had a command injection vulnerability present in their web server [21, 17]. We used this vulnerability to gain access to the shell and begin reconnaissance.

First, we read the `/proc/mounts` file and found that the router used both a `jffs2` and `squashfs` filesystem. We initially targeted the `jffs2` filesystem as it was writeable by default and would have been the easiest to modify. However, it

was mounted to `/tmp/openvpn` and only contained configuration files, so while we were able to make persistent modifications to the directory, it would not cause any arbitrary execution when the device was rebooted.

We instead decided to target the `squashfs` filesystem as it was mounted as the root directory. We read `/proc/mtd` and identified a partition named "rootfs", which was most likely the root filesystem. We read the partition and found it was using `squashfs` version 4.0, with `xz` compression.

Gaining Persistence. After extracting the files, we modified the result to include a file named `testfile` in `/bin`, then re-created the filesystem using the `mksquashfs` utility. We then uploaded the generated filesystem to the temporary memory of the router. We overwrote the existing filesystem by writing our modified version to `/dev/mtdblock15`. When we rebooted the device, the `testfile` was readable, indicating a persistent edit.

Read-Only MTD Partitions. During our exploitation of the device, we found that some of the partitions, notably the bootloader, had been marked as read-only via MTD. We were able to compile the Netgear's `mtd-rw` kernel module against the firmware's GPL source (https://kb.netgear.com/2649/NETGEAR-Open-Source-Code-for-Programmers-GPL) and confirmed that inserting the module would allow attackers to set MTD partitions as writeable from userspace.

4.2 D-Link DCS-932L (using Method C)

The DCS-932L is a web-connected camera for both indoor and outdoor use. Customers can access the camera remotely via a web browser or linked application.

This camera has a buffer overflow vulnerability that allows an attacker to gain access to the shell and run arbitrary commands [22]. We used this to gain access to the device and investigate how it manages its storage. We read the `mounts` file and found only temporary and pseudo filesystems were being used, leading us to believe that it was using `rootfs` as its main filesystem, which should be appended to the end of the kernel. For this device, we used Method C, modifying the `initramfs` so that a custom filesystem would be loaded.

We read the `/proc/mtd` file and identified an MTD partition named "kernel" which we copied to a host machine to analyse. Using Binwalk [25], we found that the filesystem could be extracted in three stages, as shown in Figure 1.

1. Stage one was the raw data of the partition as it was stored on the flash chip. It was made up of a 64-bit uImage Header, and LZMA compressed

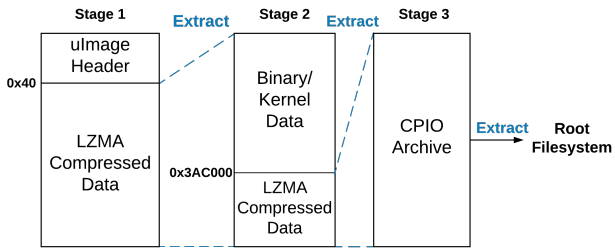

Fig. 1. Extracting the DCS-932L's Root Filesystem from the Kernel Partition

data. The uImage header contained metadata that the U-Boot bootloader can use to boot the kernel contained in the LZMA payload. We extracted the LZMA compressed data in preparation for stage two.
2. Stage two consisted of the kernel and some further LZMA compressed data. We identified that the LZMA data began at the offset `0x3AC000`, so we carved the data from this offset to the end of the file. We then decompressed this data into stage three.
3. After extracting the LZMA data, we were left with a CPIO archive, which we could then extract or mount to view the root filesystem of the device.

Gaining Persistence. To gain persistence on this device, we needed to modify the kernel partition in such a way that the device would be able to boot and mount it correctly. To test our process, we changed the root filesystem to contain a file named `testfile` in the `/bin` directory, then began to reverse the process we used to extract it. First, we compressed the filesystem into a CPIO archive. We then needed to compress the CPIO archive using LZMA. However, the compression used by the device was non-streamed. To recreate this as best as possible, we used an old version of "LZMA utils" (https://tukaani.org/lzma/). We then prepended the original binary/kernel data and compressed it using LZMA. Finally, we had to add a new uImage header. As uImage headers include checksums to check the integrity of the image contents and the header itself [13], we could not simply prepend the original, as the checksums would fail to match when the device starts, causing a fault. Instead, we created a new header with the `mkimage` utility. The arguments to recreate the metadata, such as the architecture, load address and firmware name, were found by referring to the previous header. We uploaded the new image to the device in temporary memory. As the kernel flash partition was writeable, we could copy it from temporary memory to flash memory via the MTD subsystem.

After restarting the device, we found our `testfile` was present in `/bin`, indicating a successful persistent modification. Attackers could use this technique to modify various startup scripts to perform malicious actions or even run applications included in the new filesystem.

4.3 Yealink SIP-T38G (using Method A)

The SIP-T38G is an Internet-connected VoIP desk phone, allowing users to manage multiple calls and messages. We gained control of the device using an adaptation of an existing exploit for previous versions of the phone [20], which allowed us to investigate the device further.

We read the `/proc/mounts` file and found that the device used `yaffs2` filesystems mounted to multiple locations, including the root (`/`), `/boot`, `/phone`, `/data`, `/config` and `/etc` directories.

As `yaffs2` is a writeable filesystem with an MTD user module, we wrote to the filesystem via the shell. The `/etc` directory held scripts that are run at boot-time, which we could modify to run custom shell commands or applications when the system next boots.

4.4 WiPG-1000 (using Method A)

The WiPG-1000 is a presenter that allows users to stream their screen from other devices on the same network. We used a command injection vulnerability [23, 6] to start a `telnet` daemon, which we used to interact with the device via the shell remotely.

After connecting via `telnet`, we read `/proc/mounts` to identify the root mount. We found that the presenter used two types of storage, a flash chip and an Embedded Multi Media Card (eMMC). The eMMC used an `ext2` filesystem, which was mounted to the root directory as read only. We were able to remount it as write enabled with the `mount` utility, after which we were able to easily modify the filesystem via shell commands, which persisted through reboots.

4.5 Results Summary

There were significant variations in the structure of the devices we sought to exploit, with the different types of storage implementations requiring a variety of methods to be applied. However, we were able to gain persistence on every device by applying the described techniques.

While the implementation of these tests was performed manually for this paper, aspects of these techniques could be automated. As device reconnaissance for selecting the correct method was very time-consuming, automation of this step would be essential for large scale attacks. Hard coding the appropriate method when a specific model of the device is detected is a possibility, but this would require manually identifying the best method for *each* device. Alternatively, method identification could be performed when a device is exploited, but this may be quite complicated to implement without generating false positives. If performed incorrectly, this could also lead to the device being bricked.

We have created a process graph to show the best method for gaining persistence, by prioritising on those which require the lower complexity to be implemented. This graph can be seen in Figure 2.

5 Countermeasures

We propose several countermeasures that could be used to mitigate the risk or prevent the threat caused by these persistence methods. Due to the variance of IoT devices, there are no "perfect" countermeasures, but those that are implemented will frustrate attackers in their attempt to gain persistence. As a consequence, these countermeasures will make the device a less appealing target.

- **Data Signing.** The use of signatures allows verification that the data contained on the flash chip has not been modified, which can prevent an attacker from gaining persistence. For example, uBoot has a "trusted boot" feature that can check whether an image is correctly signed before continuing the boot process [34]. By cryptographically signing each stage of the booting process – including the bootloader(s), operating system and filesystem – each

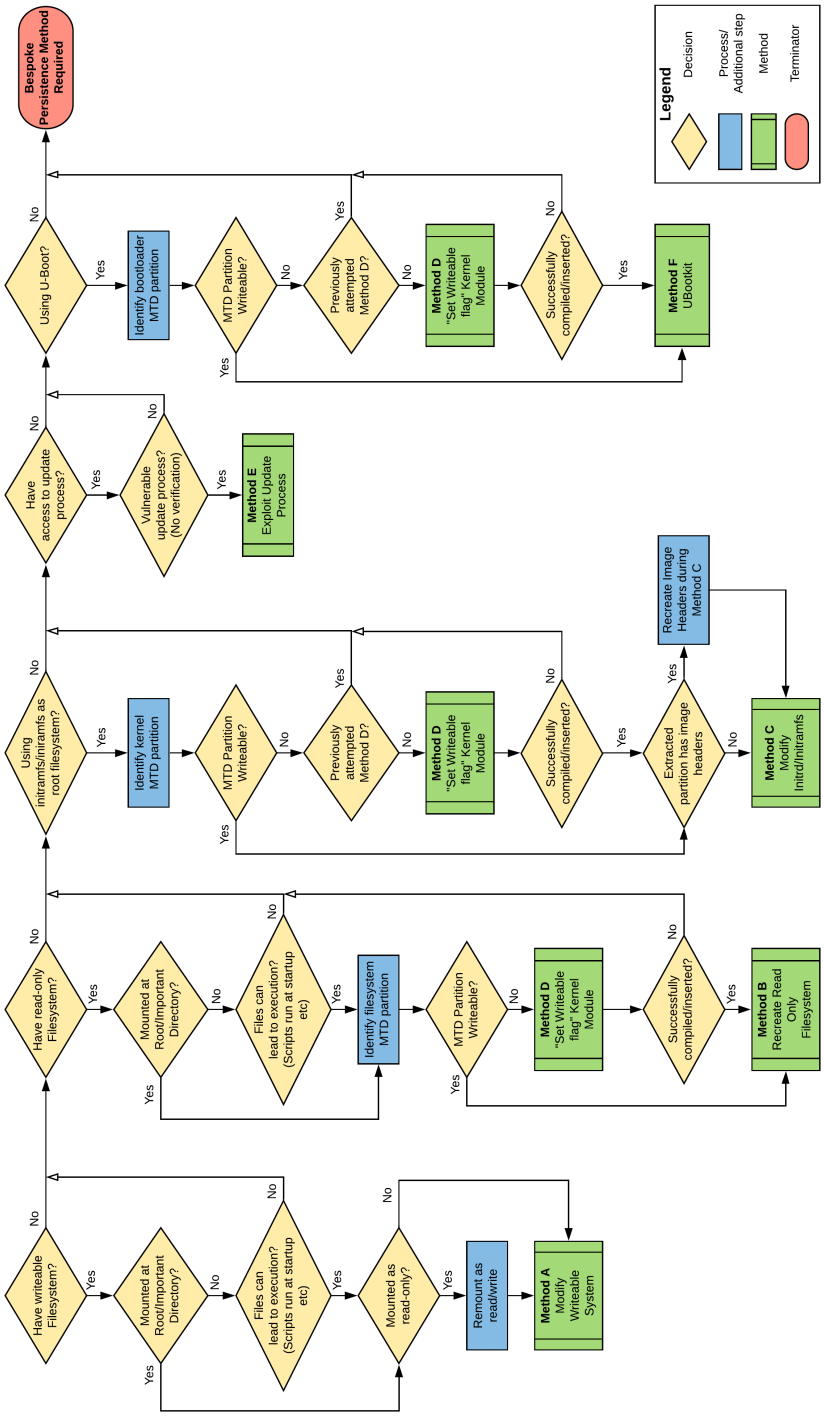

Fig. 2. Process to Gain Persistence

step can verify the signature of the next, creating a chain of trust. If a stage has been modified, its signature will not be valid, and the device will fail to boot. Immutable memory should be used to bootstrap the process, so that an attacker cannot modify the chain of trust at the very first stage. As the attacker should not have the developer's cryptographic keys, they would not be able to forge a signature for any modifications they might have made to the protected stages.

- **Principle of Least Privilege.** All of these methods require an attacker to modify data on the device. By running potentially exploitable applications at a lower privilege level and only allowing certain privileged accounts to interact with the storage device or make persistent changes to important files, attackers would not be able to make modifications to gain persistence.
- **HADES-IoT.** HADES-IoT is a system designed for use on IoT devices, which provides a process whitelisting feature [9]. HADES-IoT records a hash of benign executables that are run in an uninfected state during a "profiling" stage. When a new process is spawned, HADES-IoT can compare it against its list of known benign executable hashes, preventing unknown processes from being created. This can frustrate attackers attempting to gain persistence and prevent uploaded malware from running.
- **Device Updates.** The methods outlined in our paper assume that an attacker has shell access. Users can prevent attackers from abusing these methods by regularly updating their device to patch vulnerabilities and prevent exploitation that would provide a shell access to the attacker.
- **Effective Factory Resetting.** IoT devices often include a "factory reset" feature that can be used to restore corrupted partitions to their original state. This could be used by victims to remove malware from the device if the process can reset partitions that have been modified by an attacker.

6 Conclusions and Future Work

In this work, we have discussed the increasing threat of persistence in IoT malware. We outlined the challenges that currently prevent IoT persistence from being easily achieved. We then detailed techniques that attackers could use to gain persistence on IoT devices, describing their requirements, what methodology they can use and which potential issues they might encounter. We demonstrated our ability to achieve true persistence in a wide range of different IoT devices. Based on our findings, we outlined a potential process to identify the best method of obtaining persistence. Finally, we listed several possible countermeasures that can be used to hinder attackers from getting persistence on vulnerable IoT devices.

Whilst we were able to gain persistence on all of our targeted devices, the variations on device structure and implementation meant that it was a time-consuming process that involved significant manual work. An attacker would almost certainly want to automate this for massive-scale attacks. One possible approach is to search for or remotely fingerprint vulnerable devices and then launch the method appropriate for that model.

Additionally, whilst it was straightforward to gain persistence on some of the devices we tested, others required more sophisticated methods that were time-consuming to discover and implement. Attackers may soon look towards automating both the discovery and the implementation of these more involved methods for abusing them in large scale operations.

Finally, further research should be performed to discover new countermeasures against persistence attack on IoT devices, for example through novel network intrusion detection systems that are effective for IoT scenarios.

References

1. Building external modules, https://www.kernel.org/doc/html/latest/kbuild/modules.html [Accessed: August 2020]
2. Kernel module signing facility, https://www.kernel.org/doc/html/v4.15/admin-guide/module-signing.html [Accessed: August 2020]
3. What is the mirai botnet?, https://www.cloudflare.com/learning/ddos/glossary/mirai-botnet/ [Accessed: August 2020]
4. What's a mirai botnet doing with my router? (2016), https://blog.f-secure.com/whats-a-mirai-botnet-doing-with-my-router/ [Accessed: August 2020]
5. Antonakakis, M., April, T., Bailey, M., Bernhard, M., Bursztein, E., Cochran, J., Durumeric, Z., Halderman, J.A., Invernizzi, L., Kallitsis, M., et al.: Understanding the mirai botnet. In: 26th {USENIX} security symposium ({USENIX} Security 17). pp. 1093–1110 (2017)
6. Baines, J.: Crestron am/barco wepresent wipg/extron sharelink/teq av it/sharp pn-l703wa/optoma wps-pro/blackbox hd wps/infocus liteshow - remote command injection (2019), https://www.exploit-db.com/exploits/46786 [Accessed: August 2020]
7. Birngruber, S., Hehenberger, F., Gründlinger, P., Zeilinger, M., Vymazal, D.: Netgear nighthawk firmware update vulnerability (2017), https://iot-lab-fh-ooe.github.io/netgear_update_vulnerability/ [Accessed: August 2020]
8. Bozzato, C., Wyatt, L.: Circle with disney firmware update signature check bypass vulnerability (2017), https://talosintelligence.com/vulnerability_reports/TALOS-2017-0405 [Accessed: August 2020]
9. Breitenbacher, D., Homoliak, I., Aung, Y.L., Tippenhauer, N.O., Elovici, Y.: Hades-iot: A practical host-based anomaly detection system for iot devices. In: Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security. pp. 479–484 (2019)
10. Brierley, C., Pont, J., Arief, B., Barnes, D.J., Hernandez-Castro, J.: Paperw8: an iot bricking ransomware proof of concept. In: Proceedings of the 15th International Conference on Availability, Reliability and Security. pp. 1–10 (2020)
11. Cloudflare: Famous ddos attacks — the largest ddos attacks of all time, https://www.cloudflare.com/learning/ddos/famous-ddos-attacks/ [Accessed: August 2020]
12. Cozzi, E., Graziano, M., Fratantonio, Y., Balzarotti, D.: Understanding linux malware. In: 2018 Symp. on Security and Privacy (SP). pp. 161–175. IEEE (2018)
13. Denk, W.: u-boot/image.h (2020), https://github.com/u-boot/u-boot/blob/master/include/image.h [Accessed: August 2020]
14. Glass, S.: Das u-boot – the universal boot loader (2020), http://www.denx.de/wiki/U-Boot [Accessed: August 2020]

15. Ilascu, I.: New iot botnet torii uses six methods for persistence, has no clear purpose (2018), https://www.bleepingcomputer.com/news/security/new-iot-botnet-torii-uses-six-methods-for-persistence-has-no-clear-purpose/ [Accessed: August 2020]
16. jclehner: mtd-rw: Write-enabler for mtd partitions (2016), https://github.com/jclehner/mtd-rw [Accessed: August 2020]
17. Land, J.: Multiple netgear routers are vulnerable to arbitrary command injection (2016), https://www.kb.cert.org/vuls/id/582384/ [Accessed: August 2020]
18. Landley, R.: ramfs, rootfs and initramfs (2005), https://www.kernel.org/doc/Documentation/filesystems/ramfs-rootfs-initramfs.txt [Accessed: August 2020]
19. mwarning: mtdrw (2019), https://github.com/mwarning/mtdRW [Accessed: August 2020]
20. NIST: Yealink voip phone sip-t38g - remote command execution (2014), https://nvd.nist.gov/vuln/detail/CVE-2013-5758 [Accessed: August 2020]
21. NIST: Cve-2016-6277 detail (2017), https://nvd.nist.gov/vuln/detail/CVE-2016-6277 [Accessed: August 2020]
22. NIST: Cve-2019-10999 (2019), https://nvd.nist.gov/vuln/detail/CVE-2019-10999 [Accessed: August 2020]
23. NIST: Cve-2019-3929 detail (2019), https://nvd.nist.gov/vuln/detail/CVE-2019-3929 [Accessed: August 2020]
24. rampageX: firmware-mod-kit (2019), https://github.com/rampageX/firmware-mod-kit [Accessed: August 2020]
25. ReFirmLabs: Binwalk (2019), https://github.com/ReFirmLabs/binwalk [Accessed: August 2020]
26. Smith, B.: A quick guide to gplv3. Free Software Foundation, Inc. Online: http://www. gnu. org/licenses/quick-guide-gplv3. html. Referred **4**, 2008 (2007)
27. Sophos: Vpnfilter botnet (2018), https://news.sophos.com/en-us/2018/05/24/vpnfilter-botnet-a-sophoslabs-analysis/ [Accessed: August 2020]
28. Sophos: Vpnfilter botnet: a sophoslabs analysis: part 2 (2018), https://news.sophos.com/en-us/2018/05/27/vpnfilter-botnet-a-sophoslabs-analysis-part-2/ [Accessed: August 2020]
29. Talos Intelligence: New vpnfilter malware targets at least 500k networking devices worldwide (2018), https://blog.talosintelligence.com/2018/05/VPNFilter.html [Accessed: August 2020]
30. Talos Intelligence: Vpnfilter update: Vpnfilter exploits endpoints, targets new devices (2018), https://blog.talosintelligence.com/2018/06/vpnfilter-update.html [Accessed: August 2020]
31. Tung, L.: Fbi to all router users: Reboot now to neuter russia's vpnfilter malware (2018), https://www.zdnet.com/article/fbi-to-all-router-users-reboot-now-to-neuter-russias-vpnfilter-malware/ [Accessed: August 2020]
32. Vignau, B., Khoury, R., Hallé, S.: 10 years of iot malware: a feature-based taxonomy. In: 2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C). pp. 458–465. IEEE (2019)
33. Woolf, N.: Ddos attack that disrupted internet was largest of its kind in history, experts say (2016), https://www.theguardian.com/technology/2016/oct/26/ddos-attack-dyn-mirai-botnet [Accessed: August 2020]
34. Yamada, M.: verified-boot.txt (2017), https://github.com/u-boot/u-boot/blob/master/doc/uImage.FIT/verified-boot.txt [Accessed: August 2020]
35. Yang, J., Geng, C., Wang, B., Liu, Z., Li, C., Gau, J., Liu, G., Ma, J., YANG, W.: Ubootkit: A worm attack for the bootloader of iot devices. BlackHat Asia (2019)