ENGR-E 399/599: Embedded systems reverse engineering

Lecture 12: Firmware update files

Austin Roach
ahroach@iu.edu

April 7, 2022

Mystery function

```
1
2  void FUN_00000000(int param_1,undefined *param_2,int param_3)
3
4  {
5    undefined *puVar1;
6    undefined *puVar2;
7
8    puVar2 = (undefined *)(param_1 + -1);
9    puVar1 = param_2;
10   while (puVar1 != param_2 + param_3) {
11     puVar2 = puVar2 + 1;
12     *puVar2 = *puVar1;
13     puVar1 = puVar1 + 1;
14   }
15   return;
16 }
17
```

- How many arguments does the function take?
- What are the types of the arguments?
- What does the function do?
- What does the function return?

Mystery function revealed

```
void *memcpy(void *dest, const void *src, size_t n)
```

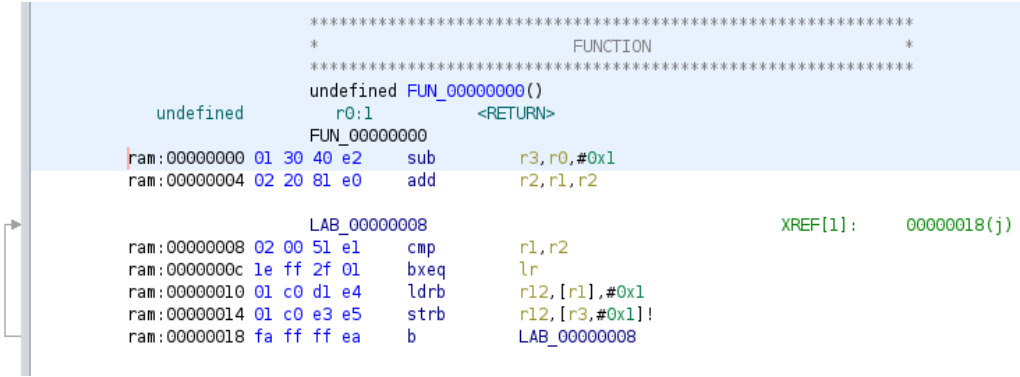
Description

The `memcpy()` function copies `n` bytes from memory area `src` to memory area `dest`. The memory areas must not overlap.

Return value

The `memcpy()` function returns a pointer to `dest`.

Disassembled memcpy()



Today's plan

- Points of access to a firmware update file
- Standard archives, compressed file formats, and filesystems
- Customized file formats
  - Examining structure
  - Identifying compression and encryption
  - Examples

Motivation for looking at firmware update files

First challenge for a project is finding *something* to analyze:

- Extract non-volatile memory
- Probe internal data buses
- Intercept external communications
- Connect to a debug port
- **Download a firmware update**

But, we may not know how the firmware update is packaged

Generic firmware update process

- Deviced attached to computer over USB for firmware update
- Dedicated application used for firmware update
- Firmware updates retrieved by application from a remote server

Our goal is to access the firmware information at some point in the update process

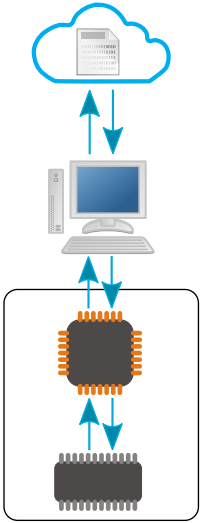
Step 1: Firmware update download

Can be accessed several ways:

- Product support site contains link to firmware update file for offline update
  - Go to website; download file; done.
- Find link to firmware update through analysis
  - Disassemble firmware update tool
  - Monitor network traffic
- Or capture the firmware update file contents in transit

Downsides:

- Firmware update file may be encrypted, compressed, or packaged in a confusing way



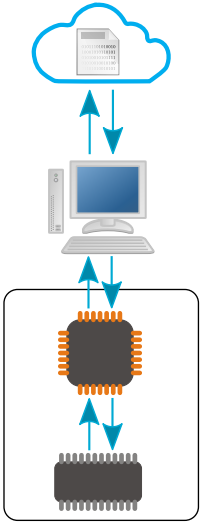
Step 2: Initial unpacking

Update file may be transformed from its downloaded form to the form that will be transferred to the device:

- Standard compressed archive formats (zip, tar, etc.)
  - Just use the corresponding extractor
- Customized archive format or encryption
  - Analyze installer utility to determine format
  - Capture extracted/unencrypted file from memory of update application
- Embedded in installer utility for offline update
  - Analyze installer utility/extract files

Downsides:

- May require a lot of analysis



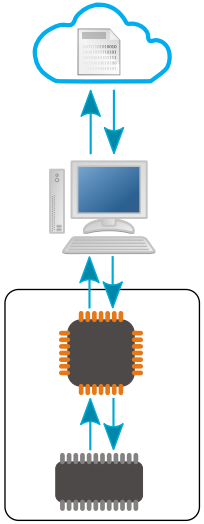
Step 3: Transfer to device

File can be intercepted in transit from the PC to the device

- Software tools to monitor bus traffic
  - USB monitoring with libpcap/Wireshark
- Hardware tools to monitor bus traffic
  - USB protocol analzyer

Downsides:

- Sometimes hard to identify the relevant data
- May require specialized tools

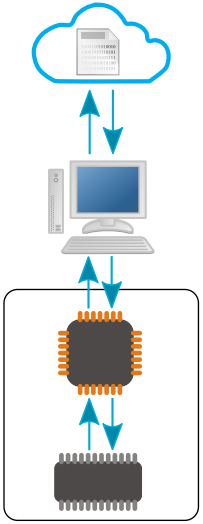


Step 4: On-device processing and storage in NVM

- Lots of variability here:
  - Decryption
  - Signature verification
  - Decompression
  - Dividing between various system components
- Understanding this process may allow the creation of modified firmware update files

Downsides:

- Need processor debug access to observe this process and capture the results

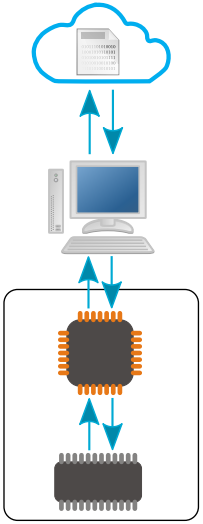


Step 5: Firmware load from NVM for use

- Sometimes executed directly from memory-mapped non-volatile memory
- Sometimes processed and loaded in SRAM/DRAM by some lower-level bootloader
  - Mask ROM
  - Embedded NOR
- Observe with processor debug access or by extracting nonvolatile memory

Downsides:

- Requires establishing debug access or hardware interface access



Standard file formats

- Very common for devices using a full embedded OS to use some standardized archive or filesystem
- More deeply embedded systems sometimes use these, but there tends to be much more variation and deviation from standards

(Compressed) Archives

- cpio archives
  - Linux initrd/initramfs images
  - `cpio` command to create/extract
- tar archives
  - Compressed tar archives sometimes used to store files within a file system
  - `tar` command to create/extract
- zip archives
  - Many file formats are secretly zip archives

cramfs filesystem

- Compressed, read-only filesystem
- zlib compressed
- Marked as obsolete in 2013 in the Linux kernel, replaced by squashfs
  - But making a comeback for systems with very little RAM?
- `mkfs.cramfs` to create
- Can mount the filesystems if Linux kernel has cramfs support
- Endian sensitive; may be able to change with `cramfsswap`

squashfs filesystem

- Compressed, read-only filesystem
- Supported by Linux kernel since the mid-2000s
- Supports various compression algorithms
- `mksquashfs`, `unsquashfs` for manipulation
- Can just mount the filesystem using a Linux kernel with squashfs support
- *Notorious for vendor-specific customizations*
  - `http://www.devttys0.com/2011/08/extracting-non-standard-squashfs-images/`

Read/write flash file systems

- Optimized for MTD (Memory Technology Devices): raw NOR/NAND flash memory
- Many variants:
  - Journaling Flash Filesystem 2 (JFFS2)
  - Unsorted Block Image Filesystem (UBIFS)
  - Yet Another Flash File System (YAFFS)
- Some filesystem-specific tools for manipulation
- Can mount the contents, typically using `mtdram` to emulate an MTD device in RAM

Other standard filesystems

- ISO 9660 (optical, bootable flash media)
- ext2, ext3, ext4 (Linux)
- Unix file system (UFS/FFS) (FreeBSD)
- FAT (Windows/DOS/various embedded applications)

U-boot images

- Standard image header:
  `https://github.com/u-boot/u-boot/blob/master/include/image.h`
- Multi-image images supported:
  - Bootloader
  - Kernel
  - Root filesystem
- `mkimage`/`dumpimage` to create/extract images
- Can also carve and analyze image parts by hand

UEFI

- UEFI firmware volume/files
- Executables in PE format
- Vendor-specific update formats
- Tools:
  - UEFITool: https://github.com/LongSoft/UEFITool
  - uefi-firmware-parser: https://github.com/theopolis/uefi-firmware-parser

Tools for customized file formats

binwalk

- Magic number identification within files
- Awareness of header formats for some embedded applications
- Recursive descent into filesystems and archives
- Often one of the first tools you want to use on an unfamiliar file

Interesting projects

- `hachoir`: File-type aware browsing of binary streams
- `firmware-mod-kit`: Tools for unpacking and packing various firmware file types
- `cpu_rec`: Automated instruction set architecture identification
  - Paper: https://www.sstic.org/media/SSTIC2017/SSTIC-actes/cpu_rec/SSTIC2017-Article-cpu_rec-granboulan.pdf
  - Source-code: https://github.com/airbus-seclab/cpu_rec

Application-specific tools

Random projects on Github

- Will typically work better for a particular system than a general-purpose tool like `binwalk`
- But be careful:
  - Often not well tested
  - May be unknowns or variations that are important to your project
  - You may have to fill in the gaps

Analyzing undocumented file formats

- File headers
- Entropy

File headers

- File formats are created by people
- Most people aren't totally crazy
- Even for unknown systems, appeal to logic
  - Try to identify structure
  - What quantities make sense to be in the header?
  - Make guesses and experiment

Common entries in headers

Things that help the parser:

Is this the right kind of file?	Magic number
What version of the format is this?	Version number
Is this file intact and uncorrupted?	Checksum(s)
How much data is there?	File size
How is the data organized?	Section table

And other quantities specific to particular applications

- System memory addresses where contents should be loaded/written
- For compressed file formats, metadata about blocks (offset, size, etc.)

Entropy

Shannon entropy

$$S = -\sum_{i=0}^{N-1} P_i \log_2 P_i$$

- S is the Shannon entropy
- N symbols
- P_i is the probability of each symbol
- $0 \log_2 0 \equiv 0$

A very unequal distribution

$$S = -\sum_{i=0}^{N-1} P_i \log_2 P_i$$

- $P_x = 1$
- $P_{i \neq x} = 0$
- $S = 0$

S = 0 for a perfectly unequal distribution

A perfectly equal distribution

$$S = -\sum_{i=0}^{N-1} P_i \log_2 P_i$$

- $P_i = 1/N \Rightarrow S = -N(1/N)\log_2(1/N) = -\log_2(1) + \log_2 N = \log_2 N$
- For $N = 2^n$, $S = n$
- $S = 8$ for $N = 2^8$
- $S = 16$ for $N = 2^{16}$

$N = 2^8$ is typically the "entropy" reported by RE tools. S = 8 for a perfectly equal distribution.

Half way in between

$$S = -\sum_{i=0}^{N-1} P_i \log_2 P_i$$

- $P_i = 2/N, \quad i = 0...N/2 - 1$
- $P_i = 0, \quad i = N/2...N - 1$
- $S = (-N/2)(2/N)(\log_2 2 - \log_2 N) = -1 + \log_2 N$
- $S = n - 1$ for $N = 2^n$

$S = 7$ if half of byte values are evenly distributed, half are unrepresented.

What does this have to do with embedded systems?

Different types of data have different characteristic entropy:

- Compressed or encrypted data: nearly 8
- Machine code (architecture dependent): around 5-6.5
- English ASCII text: around 4.5
- English UTF-16 text: around 3.2

Entropy can be used to find natural divisions in file data, identify candidate data types

Visualizing entropy

- Line graphs (binwalk/custom tools)
- Color coding (Ghidra)
- Space-filling curves (`binvis.io`)

Digraphs and trigraphs

- Can carry more information than single bytes
- Useful for architecture identification (see `cpu_rec`)
- Also awesome presentations: https://www.youtube.com/watch?v=4bM3Gut1hIk
  - Ohmygoodness, they finally open-source part of this!: https://inside.battelle.org/blog-details/battelle-publishes-open-source-binary-visualization-tool
  - https://github.com/Battelle/cantordust

Examples

Pinecil

Open-source USB soldering iron

- https://wiki.pine64.org/wiki/Pinecil
- https://github.com/pine64/pinecil-firmware-updater

Pinecil firmware updater

```
$ file pinecil_firmware_updater_macos64_1.3.dmg
pinecil_firmware_updater_macos64_1.3.dmg: zlib compressed data
$ 7z l pinecil_firmware_updater_macos64_1.3.dmg
...
$ 7z x pinecil_firmware_updater_macos64_1.3.dmg
```

Pinecil firmware update

File is Intel hex...

- Import into Ghidra as data
- Nothing recognizable...

```
$ srec_cat Pinecil.hex -intel -offset -0x8000000 -o Pinecil.bin -binary
$ ~/git/cpu_rec/cpu_rec.py Pinecil.bin
Pinecil_8000000.bin
full(0xbeb8)    RISC-V
chunk(0x6800;26)    RISC-V
```

- Import into Ghidra as 32-bit RISC-V code

espurna

Open-source home automation firmware

- https://github.com/xoseperez/espurna/

espurna file exploration

- xxd ... nothing recognizable
- `binwalk -E` ... doesn't look encrypted or compressed
- Examine strings, get some idea about the processor type

```
$ cpu_rec.py espurna-1.14.1-magichome-led-controller-20.bin
full(0x7fad0)  Xtensa
chunk(0x20000;64)   Xtensa
```

espurna analysis

- Open in radare2/rizin
- `e asm.arch=xtensa`
- `s 0x20004`
- `pd 40`
- Note zeros for re-alignment of function starts. (Have to manually step to correct start of function.

gl.inet-mv1000

Open-source wifi router firmware

- https://docs.gl-inet.com/en/3/release_notes/

gl.inet-mv1000 file analysis

- `xxd`
- `binwalk -E`
- `binwalk`
- `carve openwrt-mv1000-emmc-3.105.img 0x1200000 > openwrt-mv1000-emmc.squashfs`
- `unsquashfs openwrt-mv1000-emmc.squashfs`