Finding stupid vulnerabilities in binaries

Armijn Hemel, MSc
Tjaldur Software Governance Solutions

June 3, 2015

About Armijn

- using Open Source software since 1994
- MSc Computer Science from Utrecht University (The Netherlands)
- core team `gpl-violations.org` from 2005 - May 2012
- owner Tjaldur Software Governance Solutions
- creator of Binary Analysis Tool (BAT)

Today's topic

Today I will talk about trying to find *obvious* security bugs in embedded devices.

This will *not* be about finding the next Stuxnet, but about fruit hanging so low you trip over it.

Recent security issues

Security has been a bit more on the radar recently:

- Heartbleed
- Shellshock

I can already tell you that in the embedded space there is a lot lot more out there waiting to be exploited.

With more and more devices being connected the risk will only increase. Examples: lightbulbs, IoT

Typical embedded use case

1. plug cables into device
2. turn device on
3. replace device when it breaks ("when the green light is no longer on")

Applying updates? Often not done (buuh!), and sometimes also very difficult or even impossible (industrial controllers).

Supply chains

Supply chains in the embedded industry typically follow the "waterfall model": it is extremely hard to fix issues downstream (license compliance, security bugs, etc.)

Bug fixes (whether it is licensing or security) are often not backported to older SDKs or chipsets and not actively pushed downstream. There seems to be very little "after sales support" or interest in getting old issues fixed (note: this differs per vendor).

ODMs also told me that once a chipset vendor releases a new SDK they (chipset vendor) often spend months doing all kinds of tests (security does not happen to be one of them), so there is some reluctance to do frequent updates.

Example: UPnP security bug

In 2006 I presented (amongst others) an exploit for certain UPnP stacks where input from the Internet was passed unchecked to `iptables` by `linux-igd`, running with full root privileges on the router.

I had days of fun!

`http://www.upnp-hacks.org/`

```
int pmlist_AddPortMapping (char *protocol,
                           char *externalPort,
                           char *internalClient,
                           char *internalPort) {
  char command[500];
  sprintf(command, "%s -t nat -A %s -i %s -p %s -m mport
      --dport %s -j DNAT --to %s:%s", g_iptables,
      g_preroutingChainName, g_extInterfaceName, protocol,
      externalPort, internalClient, internalPort);
  system (command);
  if (g_forwardRules) {
    sprintf(command,"%s -I %s -p %s -d %s -m mport
     --dport %s -j ACCEPT", g_iptables,g_forwardChainName,
     protocol, internalClient, internalPort);
    system(command);
  }
  return 1;
}
```

Industry responses

When I contacted companies responses were mixed: some companies were very responsive and engaging, others only did the bare minimum. One company told me “this device will be EOL soon, so we are not going to act on it”.

But many of the fixes were “one off” and (near) identical devices from other (or even the same) vendors were still vulnerable. There was no structural fix.

Even though SDK vendors and ODMs fixed (some of) the bugs I discovered in later SDKs and products, there were many devices on the market for a significant period of time (years) that were still vulnerable.

Haven't we learned?

Each time Linux moves into a new space the same old errors seem to be repeated:

- "../.." path bugs
- environment
- not sanitizing input
- etc.

I would not be surprised if some old Unix server bugs have been reimplemented in embedded and mobile and are exploitable again. Also, what will the "Internet of Things", "connected home", etc. bring security wise?

Isn't there anything we can do to raise the security bar?

Project: Deep Firmware Inspection

NCSC (`http://www.ncsc.nl/`) is the Dutch National Cyber Security Center, part of the Minstry of Justice and Safety.

NLnet Foundation (`http://www.nlnet.nl/`) is a Dutch non-profit organisation funding network and security research and development.

NCSC and NLnet Foundation have teamed up for the "Deep Firmware Inspection" (DFI) project.

The goal of this project is to do more proactive scanning of firmwares of embedded devices before they are deployed.

It is not about finding *all* bugs or very obscure bugs, just known, obvious and preventable ones. The end goal is to become "secure by default" and make these devices safer for everyone.

Relation to Linux Foundation’s Core Infrastucture Initiative

Linux Foundation started the “Core Infrastructure Initiative” to improve security and quality of core projects.

This project is not related, and does not overlap, since it works on the other end of the supply chain.

DFI project hygiene

- the whole project plan can be requested from the government ("Wet Openbaarheid Bestuur")
- as much as possible open source components will be used and/or released as open source.
- to avoid software patent threats ideas will be filed as so called "defensive publications" (explicit statements of prior art).
- plans are to do as much research as possible in the open (student projects, conference papers, etc.)
- plans to work with upstream sources (chipset manufacturers, ODMs)

Subprojects

- virus scanning (using ClamAV)
- correlating security information (for example CVE) with binary analysis
- (basic) source code analysis to find smells
- comparison of firmwares/binaries to known vulnerable firmwares/binaries
- analysis of encrypted ZIP files to see if known plaintext attacks are possible
- generate scripts vulnerability testers can follow to speed up their searches
- and some more

Primary building block: Binary Analysis Tool

Binary Analysis Tool (or: BAT) is a lightweight tool under an open source license that automates binary analysis.

- demystify binary analysis by codifying knowledge
- make it easier to have reproducable results
- common language for binary analysis

BAT is a generic framework for binary analysis. Until now focus of BAT was primarily on software license compliance.

BAT has been in development since late 2009, BAT 21 was released on May 21 2015.

BAT modules

BAT is extremely modular and it comes with several modules:

- unpacking over 35 file systems and compressed files
- report on common properties (file type, size, etcetera)
- search for license markers and identifiers
- advanced string identifier search
- dynamic ELF linking verification
- kernel module analysis
- many more

This will be extended with security modules.

Analysis steps

Steps to determine if a binary contains particular source code:

1. extract binary files from blobs (firmwares, installers, etc.) recursively (if needed)
2. extract identifiers (strings, function names, variable names, etcetera) from binary files and compare these to (publicly available) source code
3. use other information like file names, presence of other files, package databases, etcetera, for circumstantial evidence

Identifier search

BAT extracts string constants, function/method names, variable names, etcetera, from binaries and compares them with a large database of strings, function/method names, etcetera extracted from source code.

Given enough data it is possible to make an educated guess of which version of a program was used (including which source code files).

After finding the source code files that have been most likely used, you can correlate the information with security information.

Identifiers and information extracted

- string constants (using `xgettext` and regular expressions for some Linux kernel code)
- function names (C) and method names (Java) (using `ctags`)
- variable names and Linux kernel symbols (C), field names and class names (Java) (using `ctags`)
- Linux kernel module info (using regular expressions)
- various characteristics of the file (SHA256 checksum, etc.)

String constant example

```
...
 } else {
    printf("%s %s: status is %x, should never happen\n",
           inst->prog, inst->device, status);
    status = EXIT_ERROR;
 }
 ...
```

Using security information from CVEs (1)

With BAT you get a pretty accurate match of which source code files have been used to build a binary. By relating security information to source code files you can see if a binary is possibly vulnerable.

Challenge: CVE information is not very structured and vastly incomplete, but useful information can be extracted:

- package name
- version
- origin

Using security information from CVEs (2)

Sometimes pathnames can be found in the description. Using package name, version, origin and a database it is possible to correlate this information and find out what packages were missing in the CVE.

Other information that can be extracted:

- patches
- Git repositories with commit information

This information could possibly be used as well to find more vulnerable software.

Using security information from CVEs (3)

Extracting security information from CVEs by correlating with a database seems to work very well:

- vulnerable versions of packages found that were not in CVE reports
- versions of packages found that were mentioned in CVE reports but which were *not* vulnerable
- vulnerable other packages found (but no “unfixed” bugs)

Scripts to be released soon.

Analysing source code to search for smells (1)

The LTSI project showed that a lot of open source software is used unmodified or minimally modified ("Yami nabe project").

This makes it possible to research source code in advance and record any "known smells", because chances are high that the unmodified, or minimally modified version was used to create the binary.

Then after matching the binary to source code a list of security problems can easily be retrieved from a database.

Analysing source code to search for smells (2)

CERT has a list of known smells in the “CERT secure coding standard”:

“Do not call `system()`” is ENV33-C in their list.

Some of these can be easily spotted using regular expressions. Other smells require a bit more work to find, for which I am planning to use “joern” (security research tool).

Comparing binaries to known vulnerable binaries

Because many companies use the same suppliers (down to using the same casing) the firmwares of their devices are often very similar, or identical.

By recording information about vulnerable firmwares/binaries and comparing new ones (SHA256 and TLSH checksums, bsdiff, identifiers) to this information it will become easier to find vulnerabilities in devices that were not mentioned in for example a CVE report.

Analysing ZIP files to find out if KPA is possible

For ZIP files it is possible to do a known plaintext attack (KPA). By comparing CRC32 checksums to a database of known files it is possible to find files to (later) do a KPA.

Generating scripts for security audits

Using information from the audit it should be possible to generate a script for security testers to follow and focus their attention first to known issues to confirm any suspicions.

Current status

Finished (or almost finished) are:

- virus scanning using ClamAV
- analysing encrypted ZIP files to see if KPA is possible (but not reported yet)
- analysing CVE reports

In progress:

- comparing binaries using SHA256 and TLSH checksums, bsdiff and identifiers

In the next few months I will work on the other functionality.

My hopes

I hope that this project will lead to a few things:

- more attention for security bugs (detection, fixing, prevention) in the supply chain
- more easily unlockable security information
- more devices "secure by default"

as well as implementing any cool ideas that you come up with during Q&A.

Questions?

Contact

- armijn@tjaldur.nl
- http://www.tjaldur.nl/
- Binary Analysis Tool: http://www.binaryanalysis.org/