SAPIENZA
UNIVERSITÀ DI ROMA

FACT-VIS: a visual tool for the analysis and security of firmware

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica

Corso di Laurea Magistrale in Engineering in Computer Science

Candidate

Valerio Longo
ID number 1655653

Thesis Advisor

Prof. Giuseppe Santucci

Co-Advisor

Dr. Simone Lenti

Academic Year 2020/2021

Thesis defended on 23/03/2022
in front of a Board of Examiners composed by:

Prof. Massimo Mecella (chairman) (chairman)
Prof. Daniele Cono D'Elia
Prof. Riccardo Lazzeretti
Prof. Roberto Navigli
Prof. Giuseppe Santucci
Prof. Andrea Vitaletti

FACT-VIS: a visual tool for the analysis and security of firmware
Master's thesis. Sapienza – University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: longo.1655653@studenti.uniroma1.it

A mia madre, a nonna Mimina, a Sara

Acknowledgments

Vorrei prima di tutto ringraziare il Prof. Santucci ed il Dottor Lenti che fin da subito mi hanno dato la grande possibilità di sviluppare un progetto così interessante ed ambizioso e la loro disponibilità nel cercare di aiutarmi ad affrontare un argomento a me completamente nuovo.

Ringrazio mia madre che c'è sempre stata durante questi anni e che mi ha sempre messo al primo al posto.

Vorrei dire grazie a Sara, che non ha mai dubitato delle mie capacità anche quando ero io stesso a farlo e che mi dimostra ogni giorno quanto lei tenga a me.

Infine ringrazio tutti coloro che mi sono sempre stati accanto durante questo periodo molto particolare della mia vita, a Fugo che probabilmente ora sta sonnecchiando, alle persone con cui ho condiviso fiere, passioni e seminari di approfondimento sui massimi sistemi "evangelici", agli amici di sempre e alle nuove conoscenze fatte durante le intense sessioni di gaming notturno.

Gli avvenimenti di questi ultimi anni ci stanno costringendo a vivere situazioni che mai avremmo pensato di attraversare e fanno riflettere che la libertà del singolo è sì un diritto fondamentale ma che finisce quando inizia quella degli altri e che non bisogna mai dare nulla per scontato, sopratutto le persone che ti vogliono bene.

Concludo riassumendo il tutto rubando le parole di Bilbo Baggins: Conosco la metà di voi soltanto a metà; e nutro, per meno della metà di voi, metà dell'affetto che meritate.

Contents

Chapter 1

Introduction

The demand of generic electronics devices is growing year after year [1] [2] despite the continuing and pressing component shortages [3] and the reasons are strictly related to the more and more precise needs of all of us. These needs come from the most disparate fields that each one of us deal with everyday. From the **domestic** necessities to use smart cleaners, tv remote controls and digital assistants, to the **personal** needs to be connected using wearable smart devices and clothes for both fitness and fashion reasons. **Smartphone, laptops and in general computer components** are nowadays an obligation, and their sales have been boosted also by the COVID-19 pandemic which forced people to connect each others only through the usage of these smart technologies. All the **healthcare** high-tech systems, from the million euros worth machines to the personal smart-oximeter are not far behind, in fact the pandemic has led us to an ever more constant interest in this type of machines. And all of these connected systems are only related to the private consume. Widening the overview, we must consider the **industrial** and the fourth industrial revolution which intrinsically needs more and more automated and IOT systems. The current sad period made us think even about the **military** fields and the level of high and efficient electronics must be behind these enormous (and despicable) weapons. The ones cited are only a bunch in the ocean of smart system

[1] https://www.statista.com/statistics/272116/ce-industry-growth-us/

[2] https://www.businesswire.com/news/home/ResearchAndMarkets.com

[3] https://www.gartner.com/en/newsroom/market-share

necessities, but the main thing is that the companies that produces and sells these kind of systems are increasing in number. In these so much different fields and systems, there is a core link that binds everything together: **The Firmware**. All these electronic systems (from the tv remote controls to computers) can not function without firmware. Unlike regular programs, which have to go through API, the operating system and device drivers, **firmware communicates directly with the hardware** and it's typically stored in the flash ROM (read only memory) of the hardware device. Due to this, if a firmware is vulnerable to malicious attacks, **the entire device can be compromised**.

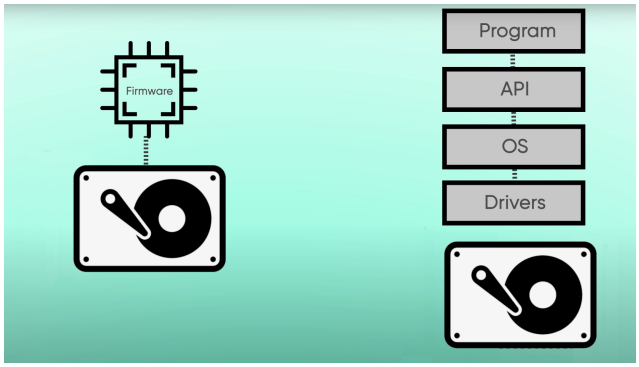

Figure 1.1. Visual difference between a regular program and a firmware

The more and more spreading of **off-the-shelf components** [21] (software already available on the market, purchasable and reusable from any company), makes easier the diffusion of these devices with **their vulnerabilities**. This scenario caused more and more people, institution and organizations with different backgrounds to take be interest about the security of these systems. Moreover, the number of new vulnerability reports submitted to the Common Vulnerabilities and Exposures (CVE) Database is growing year after year [22, 5] relying the fact that **the more devices are developed and sold, the more vulnerabilities born, the less secure those systems are** [13]. As example, recently have been discovered that more than 100 vendors and 150 devices are affected of three critical vulnerabilities such that they could enable hackers to remotely execute malicious code and take full control of devices, access sensitive data or alter configurations in impacted devices [23]. Also, the fact that almost no electronic devices can function without a firmware,

leads to the conclusion that it is crucial to provide security and **analyze the safety of firmware.** All these factors leaded to the creation of a standard methodology in order to **collect, analyze, highlight and evaluate firmware**. As we will see in the next chapter, does not exists single and full-automatic way to reach this goal, starting from the availability of the firmware itself. This branch of the cyber-security field need an extremely specific workforce which is able to go deep inside the binary code without leaving out the main software paradigms. Talking more in general, USA National's information [4] says that a generic supply of cybersecurity workforce is low with respect to its demand. This means that niches of this field **suffers of missing expert** especially for the figure of firmware analyst. This leads to the fact that this fields needs more than others to be helped using **specific tools aimed to provide specific analysis** firmware's related. So in general we can affirm that the problem of firmware analysis is not trivial and it requires specific skills to examine it, but the use of only in depth techniques could preclude amateurs and non expert of the domain to explore the flaws of their systems. **Does not exists a straightforward method to automatize the entire firmware analysis process**, this is because the analyst has to work with different elements which presents different problematic. For this reason, the we relied on **Visual Analytics principles**, which focuses on facilitate the user into perform analytical reasoning through interactive visual interfaces, using different visualization techniques. This field tries to connect the human cognitive capabilities with the computational power.

The aim of *FACT-Vis*, the tool which we are presenting, is to assist users into studying the flaws of a specific firmware with the help of visual analytics, providing different and customizable metrics in order to be used from the widest range of users based on their capabilities, from cyber-security equipes to the hobbyists. As the name suggests, the tool we're presenting is mainly connected to *FACT: Firmware Analysis and Comparison Tool*. It provides static information about the firmware status, the topology of the directories, the morphology of the files contained and some variable security aspects based on the plugin used. **FACT-Vis instead offers the possibility to have a deeper interaction between the user and the**

[4] https://www.cyberseek.org/heatmap.html

firmware allowing a better inspection of the data, while the coordination among all views is fundamental to see each possible projection of the information, for the purpose of studying them according to different points of view. Moreover, FACT-Vis integrates information not only from FACT, which remains the main source, but also from the *NVD - National Vulnerability Database*, in order to provide a complete and up-to-date match of the firmware's vulnerabilities. The tool is build for all these various kind of users can rely on, examining the report provided by FACT and plotting the most valuable information in the most appropriate way, highlighting the standard parts, the behaviour guidelines, the vulnerabilities and in general the flaws incidence with the possibility to interact with them in order to have the most suited report based on the user necessities. Moreover, FACT-Vis is build to facilitate users into firmware's analysis, which can leads to a more conscious usage of the electronic devices. The user profiling has been a core part of this thesis, we tried to develop the tool taking into account the main requirements that each kind of user needs. This process has been made with the help of periodical **meetings with domain's experts** which reported all the pros and the cons of the current build in order to proceed in the right sense without losing the aim.
The work has been made using a User Centred Design model and studying the state of the art in both fields of visual analytics applied to software and firmware analysis techniques,that we will see in the following chapters.

1.1 Thesis overview

In particular the thesis is divided in 5 chapters:

- **Chapter 2 - State of the art**: We will present the literature studied, the papers from which we relied on, the current methodologies and tools present in the firmware analysis field and the visual techniques currently used for similar purposes.
- **Chapter 3 - Design**: Here we will discuss all the design choices made for FACT-Vis and the reasons behind that.

- **Chapter 4 - FACT-Vis**: We will see the details of the implementation, the troubles in building some specific parts of the tool and how we solved it.
- **Chapter 5 - Conclusions**: The conclusion will sum up the work done and future ideas to expand, optimize and enhance FACT-Vis in order to be maintainable, scalable and more suitable for all.

Chapter 2

State of the Art

This chapter describes all the elements that surrounds the **firmware security analysis ecosystem**. We will start exploring the steps needed to reach the most optimized result, the main tools needed to elaborate the firmware; then we will see in very details what FACT is and the components which it's made of. After that we will review the visual methodologies more used in this field and the main reasons behind that.

2.1 Firmware analysis

2.1.1 OWASP methodology

OWASP (Open Web Application Security Project) open source foundation, attempted to formalize the **OWASP FSTM - Firmware Security Testing Methodology**, a nine staged guide to tailor security researchers, software developers, hobbyists, and Information Security professionals in conducting firmware security assessments.[26] The steps are summarized as follows:

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 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 focus of our work is to **enhances the stages 3-4-5** of the methodology. The stage 3 : *Analyzing firmware* is performed by FACT by *file_type and cpu_architecture plug-in* and let the user to analyze firmware file types and potential root filesystem metadata. This is significantly improved by FACT-Vis with the implementation of **visuals encoding and interactive graphics** related to the firmware overview.
The stage 4: *Extracting the filesystem* is performed by FACT with the *unpacker plug-in* and it is improved by FACT-Vis by the usage of an **ad-hoc aggregate error collection** of the plugin, in this way the system provides a way not only to get a general overview of the unpacked files, but also let the user see which file has been remained packed and the probable reasons behind that.
The stage 5 **Analyzing filesystem contents** is made by FACT thought the plugins *cpu_architecture, crypto_material, cve_lookup, software_components, software_components*, with the task to investigate if the firmware contains vulnerabilities of different natures. This stage is boosted by FACT-Vis with a peculiar **rank danger algorithm** aimed to elaborate the most suited analysis based on the user needs.

2.1.2 Technical solutions - Related Works

Firmwalker [15] is a bash script for searching the extracted or mounted firmware file system. In particular, it searches elements of interest such as:

- etc/shadow and etc/passwd
- list out the etc/ssl directory
- SSL related files
- configuration files
- script files
- .bin files
- keywords such as admin, password, remote, etc.search for common web servers used on IoT devices
- common binaries such as ssh, tftp, dropbear, etc.
- URLs, email addresses and IP addresses

This tool is also suggested by the OWASP methodology in the execution of stage 5 together with FACT. In our specific case, as illustrated previously, the majority of the output provided by firmwalker are also provided by FACT with the add of the automation and extensibility for which already talked about.
Another tool used to extract and analyze the firmware is the so called **FAT-Firmware Analysis Toolkit** [25]. This tool, even if is capable to extract and carve a firmware, as OWASP suggests, it's supposed to be the main instrument to pass the stage 6 - Emulating firmware phase. In fact the strength of FAT resides in the capability of **emulate** firmware and verify potential vulnerabilities in a safe environment.

Figure 2.1. Full firmware emulation example on FAT

Now we will discuss the tools used to do firmware analysis and then we will proceed on the discussion about the visual solutions. Other firmware analysis tools, which circumvent the OWASP methodology and are present in literature are various. A first study approached was one called **PATCHECKO**, a vulnerability assessment framework that leverages deep learning and hybrid static/dynamic binary analysis to perform cross-platform binary code similarity analysis to identify known vulnerabilities without source code access with high accuracy [33]. PATCHECKO combines the study of the dynamic binary analysis with the static deep-learning binary analysis.

This is a complete new approach. In fact, as we already discussed, the unpacking problem here is slightly work-arounded: PATCHECKO does not require access to the source code of the target binary nor that of vulnerable functions. After the training of the model, it requires only the binary file.

For what about concern the dynamic analysis, **FIRMADYNE** [9] is a good example to talk about. This tools is presented as a Linux-based automated dynamic analysis system. This system at first downloads the firmware images from various websites. Then, through *Binwalk* are extracted. After the extraction they are emulated using *QEMU* [29] and then they are analized to check vulnerabilities and flaws.

Sometimes firmware are developed with some functionalities which, at the end of the development process, are not used anymore and instead of remove them, they are hidden. These feature can be also considered malicious (just taking as example a backdoor). *HumIDIFy: A Tool for Hidden Functionality Detection in Firmware* [34] is a system aimed to **find these kind of hidden features** through a semi-automated approach. **HumIDIfy** is build first with a classifier developed exploit several existing supervised learning algorithms and an adaptation of the semi-supervised self-training algorithm[39] Then, using semi-supervised learning, the system infers what kind of functionality a given binary has. After that, the classifier identifies the binaries from the firmware taken in examination. In order to compare the expected functionality profile, they developed a BFDL - Binary Functionality Description Language which encodes the static analysis to specific functionality traits of a binary. This work reached the 96.45% of accuracy.

Another kind of firmware analysis is the **firmware re-hosting** : given firmware/hardware combination (possibly through multiple execution rounds), the system understands what are the main expects from the surrounding hardware, and then attempt to replace the hardware altogether, so that the firmware analysis can be carried out with software-only components in an safe emulated environment. Several approaches has been made in this field. An example is *Toward the Analysis of Embedded Firmware through Automated Re-hosting* [17]. In this work they presented **PRETENDER**, a scalable program analysis able to generate models of peripherals automatically from recordings of the original hardware. They run this system on

multiple firmware samples across different hardware platforms demonstrated that automatic rehosting is possible and if thorough, this paradigm can be essential for the firmware impossible to unpack (or the ones with some shadows) but also applicable to the other, in order to provide a more complete and exhaustive analysis.

2.2 FACT

FACT - Firmware Analysis and Comparison Tool [7] is a tool born in 2015 and after 2 years of development became a reference point for what about concern it's field such that it is cited by OWASP as the main tool to put in act the stages 3-4-5.

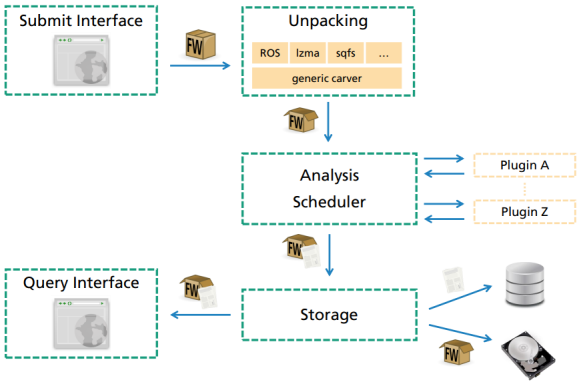

Figure 2.2. FACT Architecture

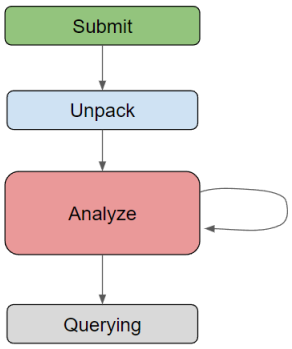

Figure 2.3. FACT Analysis steps

As visible in Figure 2.2 and 2.3, after the **submission** of the firmware, the *unpacker plugin* starts trying to **unpack** the filesystem and then, thanks to the different *plugins* that the user want to start, permits to do different*analysis* and to store the results which can be retrieved **querying the system**. Moreover, its extensibility let FACT to be scalable and easily improved developing a new plugin (or upgrading an existing one).
FACT has the capability of:

- Running analysis in **automated and in parallel** way;
- Combining various **unpacking tools**;
- Being easily **extendable** with the plugin system;

- Storing analysis results.

Now we will see some of the most important functionalities supported by FACT and we will give a brief overview of what they are and what is their purpose and implementation.

2.2.1 Binwalk and the Unpacker plug-in

The FACT unpacking capabilities strongly relies on *Binwalk*. Binwalk is a fast, easy to use tool for analyzing, reverse engineering, and extracting firmware images [30]. The main functionalities that this tool offers are:

- **Scanning Firmware**: Binwalk can scan a firmware image for many different embedded file types and file systems
- **File Extraction**: Binwalk can extract any files that it finds in the firmware image
- **Entropy Analysis**: Binwalk can help identify interesting sections of data inside a firmware image

For what about concern the entropy analysis, it's fundamental to get a measure of the **probability that a certain firmware has been encrypted or compressed (high entropy) or not (low).** The Unpacking process is based on the usage of a series of different plug-ins. Some are really specific to a certain filesystem (*SquashFS, linuxKernel*) and others generic, in case the specific ones doesn't fit the scope (*genericFS, generic_carver*). Different studies [12] highlighted that it's not an easy task to unpack a file system because of the Encryption and/or the proprietary packing software. This leads to an high entropy and **some mistakes during this phases are not uncommon**.

2.2.2 Cve lookup and Software Components

In order to check the software components which are present inside the firmware, FACT is provided by a custom OS Tagging functionality based on data extracted using *Ghidra* [16]. For the Common Vulnerabilities and Exposures (CVE) instead,

FACT searches them inside an internal databases and during the analysis performs some queries in order to retrieve the score and other details. This means that the **CVE list it's not up-to-date** with respect to the official *NVD* - National Vulnerability Database provided by NIST - National Institute of Standards and Technologies [24].
The CVE list is a collection of public vulnerabilities and security flaws. They are scored through the so called **CVSS - Common Vulnerability Scoring System**. This is a way of assigning severity rankings from zero (least severe) to 10 (most severe). According to the FIRST - Forum of Internet Response and Security Teams [1], This score system is valuable because:

1. Provide to the industry a **standard** capable of rank vulnerabilities, helping critical information flow more effectively.

2. The formula for determining the score is **public and freely** distributed, providing transparency.

3. **Prioritize risk** taking (or not taking) into account some elements based on the context, providing both a general and specific metrics.

Currently the CVSS scoring system is at its third iteration (CVSSv3) and has three metrics for ranking a vulnerability: A **base score**, which describes how much hard it is to exploit the vulnerability and how much damage that vulnerability could cause. The base score comes derives from two subequations.

- The **Exploitability** score, which measures how vulnerable the flaws is to attack based on four metrics:

    1. UI - User Interaction: if the user requires or not another user to exploit the vulnerability. binary score.

    2. AV - Attack Vector: how hard it is for an attacker to actually access the vulnerability. e.g. if the attacker needs to be physically present to exploit the flaws, this score will be low.

[1] https://www.first.org/

3. PR - Privileges Required: the level of privileges the attacker must have to exploit the flaw. e.g. no privilege means maximum score.
4. AC - Attack Complexity: how special must be the conditions in which the attacker can control to exploit the vulnerability. e.g. if the attacker can repeatedly exploit the vulnerability, this will bee low.

- The **Impact** score, which measures how badly the exploit will impact on the component. It's mainly composed by the AS - Authorization Scope, a binary score which reflects the possibility (or not) of an exploit to reach other components but if the AS change does not occur, these three metrics are considered:

  1. Confidentiality: low if not restricted information has been compromised, high otherwise.
  2. Availability: high if the exploit lead to a persistent and serious disruption, low otherwise.
  3. Integrity: low if the data corrupted doesn't have substantial consequences, high otherwise .

Moreover, CVSS provides two more scores, which are not collected by FACT, these are the temporal score, which consider the awareness of people of the vulnerability and the environmental score, which changes based on the organization.

2.2.3 User and Password-plugin

This particular plug-in is developed with the purpose to **find and crack** some encrypted strings found in the file analyzed. The plug-in is built exploiting *John the Ripper password cracker*. This tool is an Open Source password security auditing and password recovery tool. John the Ripper supports hundreds of hash and cipher types, and it's a fundamental for FACT to retrieve plain credentials inside the file objects.

2.2.4 Exploit mitigation

It's not unusual to enrich an executable file with some techniques to make it harder to exploit software vulnerabilities reliably. *Checksec* [8] is a tool built with the purpose to **find the these mitigations techniques** (if present) and report them. FACT uses checksec to analyses ELF binaries within a firmware for present exploit mitigation techniques. In particular, it finds:

- **RELRO - Relocation Read-Only**: The ELF sections are reordered so that the ELF internal data sections (.got, .dtors, etc.) precede the program's data sections (.data and .bss). Non-PLT Global Offsets Table (GOT) is read-only and PLT-dependent GOT is still writeable. Its variant, the Full RELRO adds the feature that the entire GOT is also (re)mapped as read-only. [18]
- **FORTIFY SOURCE**: GCC and GLIBC security feature that attempts to detect certain classes of buffer overflows
- **PIE - Position Independent Executables**: the binary file and all of its dependencies are loaded into randomized locations within virtual memory each time the application is executed. Used to reinfoce the file against Return Oriented Programming (ROP) attack. It also checks for PIE/DSO (dynamic shared objects) and PIE/REL [28]
- **NX - No-execute bit**: used in CPUs to segregate areas of memory used to store either code or data
- **CANARY** : pieces of code used to report threats when encountered .
- **CLANG CFI- Control Flow Integrity**: schemes designed to abort the program upon detecting certain forms of undefined behavior that can potentially allow attackers to subvert the program's control flow.[11]
- **SAFESTACK** : separation of the program stack in two regions (safe and unsafe) in order to prevent buffer overflows [31]
- **RPATH - (Run-time search Path)** : overrides the system default shared library searching paths, encoding into the header of the executable the path

to the shared libraries [10]

- **RUNPATH** - An easier to use evolution of RPATH
- **STRIPPED SYMBOLS**: during compilation, certain symbols are included in the binary, mostly for debugging. Removing them makes it somewhat harder to infer the software's inner workings.

2.2.5 Crypto-plugin

In order to **detect crypto material** like SSH keys and SSL certificates, FACT exploits *YARA* [38]: This is a tool aimed at (but not limited to) helping malware researchers to identify and classify malware samples. With YARA it's possible to create descriptions of malware families based on textual or binary patterns. Each description, a.k.a rule, consists of a set of strings and a boolean expression which determine its logic. FACT converts the yara results (started with a crypto rule match) and reports its output.

2.2.6 FACT Graphic User Interface

As previously said, FACT automatize several processes like:

- Unpacking
- Feature Extraction
- Archiving
- Generation of statistic
- Keeping track of metadata

This automation let the system to collect the stage 3,4 and 5 of the OWASP FSTM. But what it can not automate is a **complex and custom analysis**. Here **FACT-Vis tries to simplify the work**. While the work behind the implementation of a scalable, automated and powerful system is well developed, complex and follows the state of the art, the result's analysis system has not yet reached the same level. In

this regard, it's needed to say that FACT offers a GUI which provides only static information about what the system did and does not support deeper analysis.

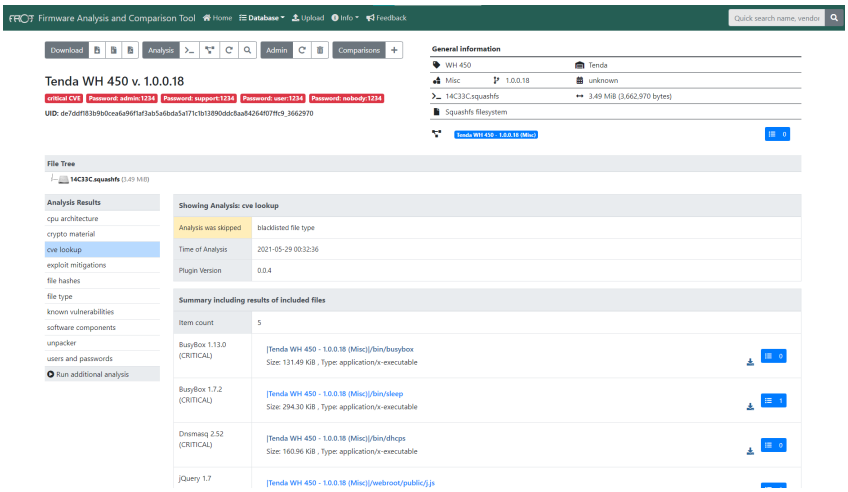

Figure 2.4. FACT GUI

The interface **doesn't offer an immediate review of the firmware's status**. The GUI, except for the general information and the colored (red or yellow) labels, it **does not provides neither a complete overview**. Also, the interactions are limited for the single plug-ins, **blocking a possible comparative study between different plug-ins** and one for the exploration of the file system. In conclusion, the **GUI doesn't offer a tool to perform accurate firmware analysis**. Instead, it locks the domain's expert to go back and forth between the source code and the GUI in order to extract some interesting results. Moreover, FACT provides a report for which misses important results[27], and in this way the user can not trace back what were the steps analysis made and can not customize the experience based on its purpose.

2.3 Visual techniques

A visual tool dedicated to binary analysis is **BinVis** [4]. This tool permits to have visual overview of the binary file. Specifically BinVis can help into looking for suspicious parts in packed or encrypted files like binaries, and to locate relevant offsets. It provides a visual overview for an easier orientation and to generate deeper

insights. This is a practical example of **how the visual techniques can be**

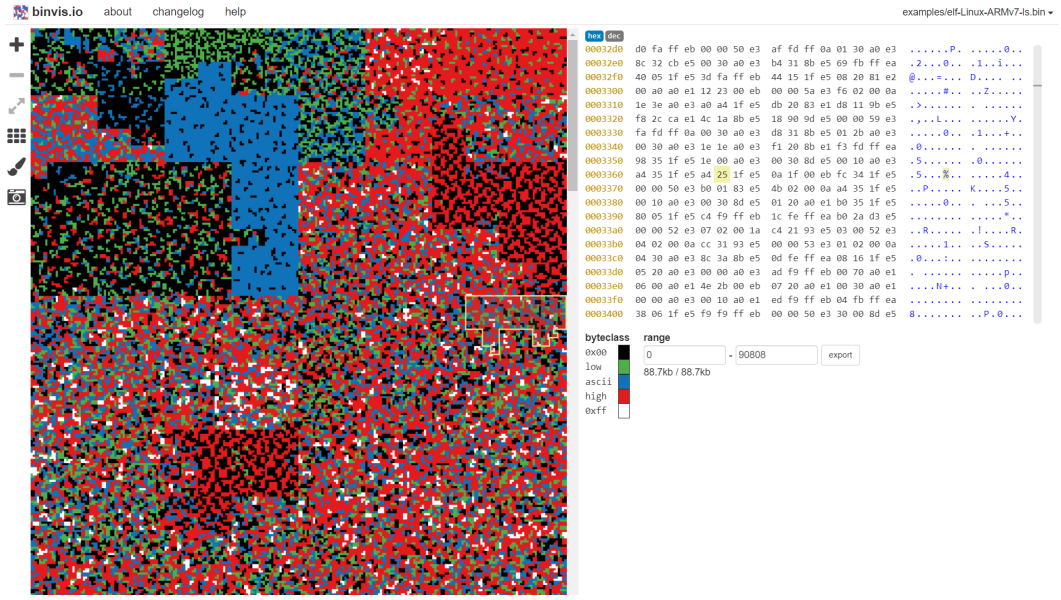

Figure 2.5. How Binvis appears after the loading of the firmware

applied in the firmware field of study and the fact that is cited by the standard methodology strengthens the relationship between these two disciplines.

In order to have a more general overview about the visual techniques used in Software Visualization, *Experiences from Performing Software Quality Evaluations via Combining Benchmark-Based Metrics Analysis, Software Visualization, and Expert Assessment* [37] offers an interesting statement, namely that stratification of results based on different target audiences (e.g., business division, IT division), was deemed as an important aspect when presenting the evaluation results. So when approaching on this field, an optimal way to deal with the large spectrum of audience is to *stratify the results* so it's crucial **to understand the target and build some variant techniques in order to access at the same information.** *Visualization of the Static Aspects of Software: A Survey* [6] also helped us into reduce the selection of the possible visualization to use. This study proposes different techniques to visualize software, reviewing and categorizing them according to their characteristics and features. This study encouraged us to try and implement the **Sunburst, TreeMap and File Tree**. For what about concerns the Treemap, *Cesar: Visual Representation of Source Code Vulnerabilities*[3] is a tool aimed to

show the flaws hidden in source code.

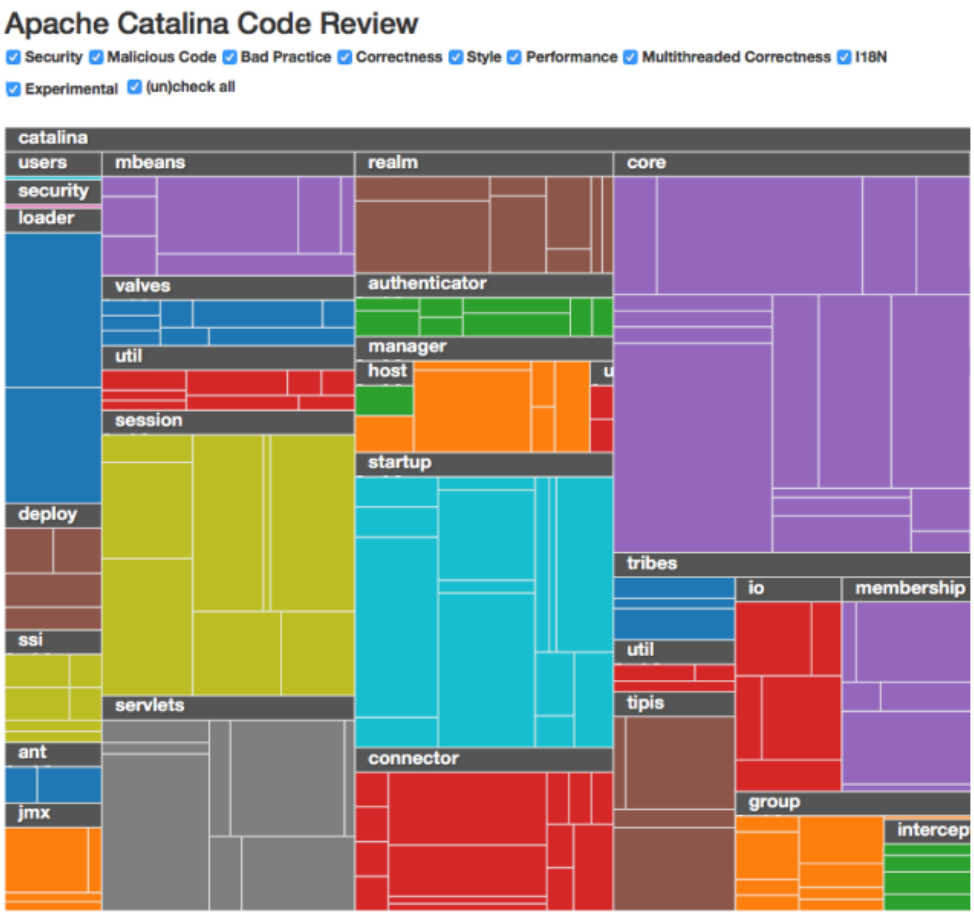

Figure 2.6. Cesar's treemap

The figure 2.6 shows the distribution of selected vulnerability categories in Catalina package using a treemap.

The visual tool tries to patch some *FindBugs*'s usability issues. FindBugs is a open source software which uses static analysis to look for bugs in Java code [14] and the usage of a Treemap as main view has been demonstrated to reach this goal. Even *Visual Analysis of Malware Behavior Using Treemaps and Thread Graphs* [36] asses that the usage of a treemap can effectively support the analysis. But *Trinius et al*[36] also studied another techniques: the thread graphs.

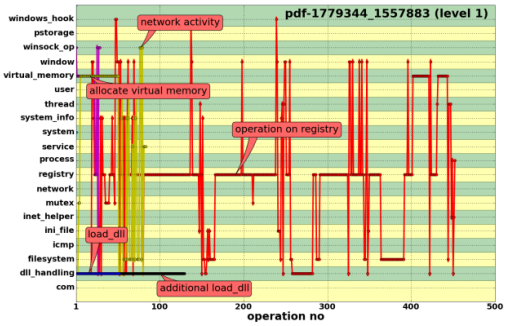

Figure 2.7. Thread graph

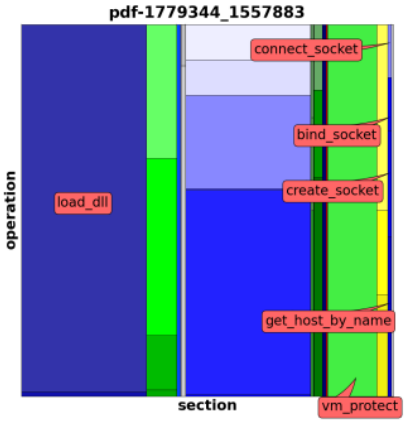

Figure 2.8. Treemap

The Figures 2.8 and 2.7 reports the different views got using the same input: a malicious PDF file. The treemap blue section (load_dll) of the malicious PDF files is slightly expended (due to the fact that a malicious PDF have to load more libraries than a safe one). In the thread graph it corresponds in a single colored line indicating the loading of DLL files. Moreover, the treemaps of the malicious PDF files has an additional, light-green colored section to the left of the yellow colored section. This is needed to the virtual memory section. The PDF document need this additional memory to save variables and malicious code. The thread graph shows the same information plotting in the virtual memory row a noticeable green line.
In our work, even if the thread graph results a valid alternative, has been decided to not considering it in order to examine more deeply the treemap features.
Even *Vulnus: Visual Vulnerability Analysis for Network Security* [2] exploits the treemap chart. This work intersect our domain, especially for which concern the **CVSS metrics** and more in general software vulnerabilities.

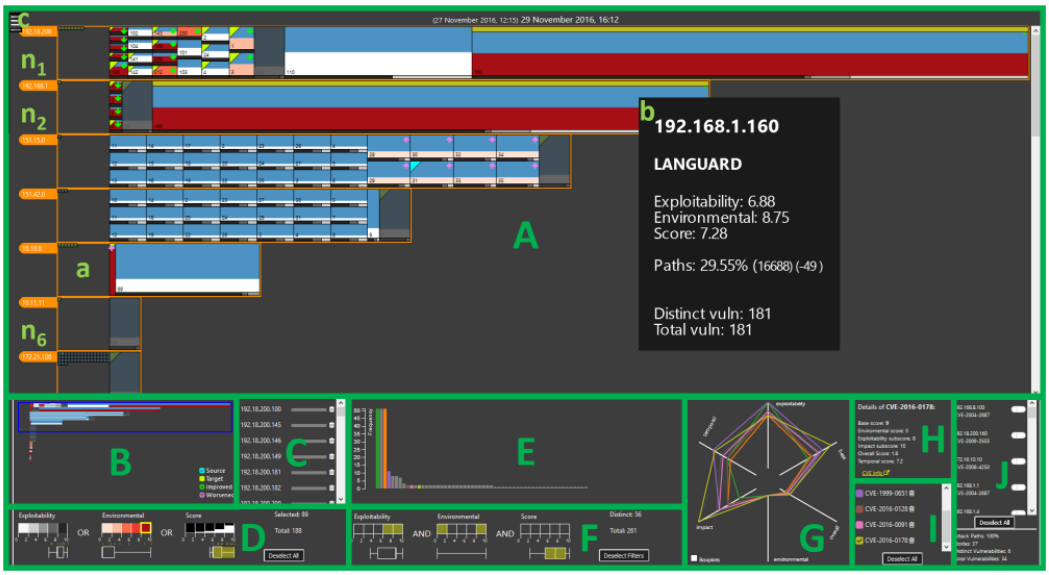

Figure 2.9. Vulnus visualization

Vulnus is a system aimed to analyze the vulnerabilities of computer networks providing both an overview and details of such issues. It's strictly connected with the CVSS (as FACT-Vis) and attack graph. Moreover, Vulnus proposes a modification of the CVSS score helping in reduce the number of interventions on a certain network. It also offer a *"what-if scenario"* which permits the exploration of the effect of fixing activities without applying them to the real system, helping the user in the situations in which applying the approximated optimal fixing strategy is not viable for organization constraints. The system exploit a peculiar treemap composed by two hierarchical level, each bar is a sub-network composed by nodes. These nodes encodes the vulnerability scores (Base, Environmental and Exploitability) the cardinality of vulnerabilities and if it is a target of a source. In order to allow comparisons, each treemap's bar is scaled proportionally to the top-most one, minimizing the change in aspect-ratio of the leaves and allowing a consistent comparison among elements belonging to different bars. But *An evaluation of space-filling information visualizations for depicting hierarchical structures* [32] showed up that the performance of the tasks we needed are equivalent to the ones with a Sunburst visualization, but the last one is easier to learn and more pleasant than a Treemap. Moreover, how highlighted by the work of *SymNav: Visually Assisting*

Symbolic Execution [1], contrary to the Treemap, a **Sunburst can add more information to the leaves**.

In conclusion, following the ideal to provide different ways to explore similar but not equal data and to support the search analysis based on some folder instead, we relied on a classical file directory tree exploration which still remains a valid visualization technique [20].

Chapter 3

Design

In this chapter we will describe the design of FACT-Vis. The flow of the design process is freely inspired by the *nested model* which consist of four nested layers which describe the path starting from the domain problem until the intuition of the actual solution. This model offers several advantages like :

- The highlighting of the user needs (user-centered design);
- Compare the abstraction of a problem with similar ones applied in different contexts;
- Shorten the distances between the designer and the domain experts.

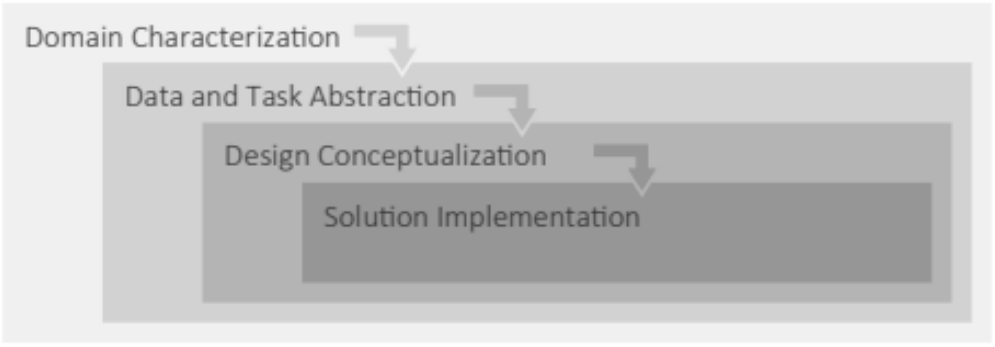

Figure 3.1. The nested model is presented as a encapsulated hierarchy which starts with the knowledge of the scenario and ends with the actual implementation [35]

Has already mentioned, the architecture building relies to various **meetings with**

domain's experts which helped us into defining the best user need to be supported and informally evaluate the proposed solutions.

3.1 Domain Characterization

In this section we will familiarize with the domain, profiling the **users** in order to line a separation between each kind of usage, exploring the purposes of every customer's typology. Moreover we must characterize the general goals that the tool must achieve, compiling a series of **requirements** based on the user necessities.

3.1.1 Users

The first step of the nested layer and as starting point of the design process, we must categorize the users and their specific application domain problem. As largely cited, FACT-Vis is built with the main focus to support the analysis for a wide range of users; Their characterization is thus a key point. We consider three kind of users based on their domain knowledge and skills.

- **User A** - the Amateur. this kind of user has very **low** knowledge of the application domain with respect to the next ones. It wants to use the system only in order to be aware of the generic flaws of its device. It doesn't know how to do complex tasks. The tools must be designed to support **a generic overview, a summary and accomplish a fully-automated analysis**.

- **User B** - the Laborer. This kind of user has a **medium** level of knowledge of the application domain. It uses the system to check the firmware integrity and review some of its specification like if it's present a specific CVE or a particular kind of danger. The tool must be designed to support **a semi-automated analysis and filtering some aspects of the firmware**. *The Laborer is capable to do the same tasks of the Amateur.*

- **User C** - The Cyber-expert. This kind of user has a really **high** knowledge of the application domain. It uses the system not only to check the firmware integrity, but also to flag which component is compromised and which one is

secure, it wants to deeply explore the firmware. The tool must be designed to support a **complete manual analysis and have a very detailed overview**. *The Cyber-expert is capable to do the same tasks of the Laborer and of the Amateur.*

Users	**Knowledge/Skills**	**Supported functionalities**
User A	LOW	Overview, summary (Fully-automated analysis)
User B	MEDIUM	Semi-automated analysis
User C	HIGH	Manual analysis

Table 3.1. Users categorization.

After this first skimming and categorization, we decided, with the help of the NICE - National Initiative For Cybersecurity Education [1], to go deeper in the user profiling. NICE is a framework aimed to categorize roles according to the task they perform and the required knowledge. NICE provide the Framework for Specialty Areas and Work Role which became fundamental in order to collect the requirements and to profile the users.
All the 3 typologies of users (A-B and C) are splitted is various roles, in order to be more precise in the definition of who use the system. However, before starting to illustrate the categorization of the user, we have to clarify that for what about concern the user A we included also profiles that **are not inside the NICE framework** like amateurs and hobbyist for which skills and tasks have been already described in the generic user design process.

Technical Support Specialist

The first one is the TSS - **Technical Support Specialist**, which is located in the field of OM - OPERATE and MAINTAIN. It provides the support, administration, and maintenance necessary to ensure effective and efficient information technology system performance and security. More specifically, it works in the STS - Customer

[1] https://www.nist.gov/itl/applied-cybersecurity/nice/about

Service and Technical Support specialty area. This role is aimed to provides technical support to customers who need assistance utilizing client-level hardware and software in accordance with established or approved organizational process components. Above others, what identifies well this role is:

ID	TSS description
T0125	**Install and maintain network infrastructure device operating system software (e.g., IOS, firmware)**.
T0468	**Diagnose and resolve customer reported system incidents, problems, and events**..
K0116	Knowledge of file extensions
A0122	Ability to design capabilities to find solutions to less common and more complex system problems.
A0034	Ability to develop, update, and/or maintain standard operating procedures (SOPs).
S0039	Skill in identifying possible causes of degradation of system performance or availability and initiating actions needed to mitigate this degradation.

Table 3.2. Main Tasks, Skills and Knowledge description of a Technical Support Specialist

.

Executive Cyber Leadership

The EXL - **Executive Cyber Leadership** is located in the OV - OVERSEE and GOVERN, the area which provides leadership, management, direction, or development and advocacy so the organization may effectively conduct cybersecurity work and the main goal of its role is to executes decision-making authorities and establishes vision and direction for an organization's cyber and cyber-related resources and/or operations. This is what defines him.

ID	EXL description
T0263	**Identify security requirements specific to an IT system in all phases of the system life cycle.**
T0509	**Perform an information security risk assessment.**
K0147	Knowledge of emerging security issues, risks, and vulnerabilities.
K0314	Knowledge of industry technologies' potential cybersecurity vulnerabilities..

Table 3.3. Main Tasks, Skills and Knowledge description of an Executive Cyber Leadership

.

Both these two roles require **basic IT knowledge and primitive Cyber security fundamentals**. They only needs of a **fully-automated analysis**

The discriminant behind a user A and a User B is the **K0322 - Knowledge of embedded systems**. In fact, this knowledge let the user to support a **semi-automated analysis** of the firmware.

Systems Security Analyst

The SSA - **Systems Security Analyst**, located in the field of OM - OPERATE and MAINTAIN as the TSS, is responsible for the analysis and development of the integration, testing, operations, and maintenance of systems security. It specifically works on the ANA - Systems Analysis role. This role is aimed to designs information systems solutions to help the organization to operate more securely, efficiently, and effectively. Brings business and information technology (IT) together. It's mainly identified by:

ID	SSA description
T0085	**Ensure all systems security operations and maintenance activities are properly documented and updated as necessary.**
T0243	**Verify and update security documentation reflecting the application/system security design features.**
A0015	Ability to conduct vulnerability scans and recognize vulnerabilities in security systems.
K0075	Knowledge of security system design tools, methods, and techniques.

Table 3.4. Main Tasks, Skills and Knowledge description of a Systems Security Analyst .

Threat/Warning Analyst

The TWA - **Threat/Warning Analyst** came from AN - ANALYZE. It performs highly-specialized review and evaluation of incoming cybersecurity information to determine its usefulness for intelligence. This role is not properly associated with user B, but can be seen has a dividing line between the ultra expert and the medium level that a user can be. Its role is to Develops cyber indicators to maintain awareness of the status of the highly dynamic operating environment. Collects, processes, analyzes, and disseminates cyber threat/warning assessments. What identifies it is:

ID	TWA description
T0708	**Identify threat tactics, and methodologies.**
T0748	**Monitor and report changes in threat dispositions, activities, tactics, capabilities, objectives, etc. as related to designated cyber operations warning problem sets.**
K0449	Knowledge of how to extract, analyze, and use metadata.
K0480	Deep knowledge of malwares.
S0229	Skill in identifying cyber threats which may jeopardize organization and/or partner interests.

Table 3.5. Main Tasks, Skills and Knowledge description of a Threat/Warning Analyst .

Now we describe the most expert class of users: the one that needs to do **a manual analysis** in the firmware.

Cybersecurity Defense Analyst

The CDA - **Cybersecurity Defense Analyst** uses data collected from a variety of cyber defense tools to analyze events that occur within their environments for the purposes of mitigating threats. This role is associated to PR - PROTECT and DEFEND, it identifies, analyzes, and mitigates threats. This is what describes it:

ID	CDA description
T0470	**Analyze and report system security posture trends.**
T0503	**Monitor external data sources to maintain currency of cyber defense threat condition and determine which security issues may have an impact on the enterprise.**
S0169	Skill in conducting trend analysis.
K0160	Knowledge of the common attack vectors on the network layer.
K0070	Knowledge of system and application security threats and vulnerabilities (e.g., buffer overflow, return-oriented attacks, malicious code..)

Table 3.6. Main Tasks, Skills and Knowledge description of a Cybersecurity Defense Analysis

Now that we defined a **fine grained users populations** we can aggregate their task based on the class of user of reference in order to be used in the next section, having:

Class	Role	ID
A	TSS, ECL	**T0125, T0263, T0468, T0509**
B	SSA, TWA	**T0085, T0243, T0748, T0708**
C	CDA	**T0470,T0526**

Table 3.7. User design summary

For which concern the context technologies and where the data analysis will be performed, we selected the desktop because it's the only device which can be at the same time simple, showing a generic overview and complex, showing very detailed information.

3.1.2 Requirements collection

After the profiling of the users, we toke care about what were the main tasks that those users have to make based on their levels of skills and we tried to organize them. The requirements have been elicited by the domain experts. The first requirement that we considered was **R1: Capability to see the overview of the most interesting file objects**, creating a way to define when a file object is *"interesting"*, ranking it. R1 satisfies the tasks:

- T0125 : The firmware's file are maintained and installed
- T0468 : Incidents, problems and events linked to the firmware security are diagnosed
- T0509 : Various security assessment can be made

Talking with the domain experts, this way must be completely customizable because the needs of each analysts change based on their goals. To R1 follows immediately the requirement **R2: Capability to see the overview of the firmware's status** , finding a way such that the user can easily and immediately check if the firmware is vulnerable or not. R2 instead, satisfies the possible tasks:

- T0263 : The overview identifies security requirements
- T0503 : The overview is proved thanks to the monitoring of external data sources

As consequence of R2, the requirement **R3: Capability to analyze the firmware** shows up, giving the opportunity to **R3.1: Capability to see the overview of the firmware composition** and **R3.2: Capability to maintain the focus only on certain aspects during the navigation of the firmware**. In general, R3 provides to solve the tasks:

- T0748 : Monitors the threat disposition

Now, we must go deeper, so we need to focus on the **R4: Capability to analyze a generic file object**, looking at every single plugin which FACT uses: crypto, users

and password, software components, cve lookup, exploit mitigations. During the meeting with the domain expert, we understood that is a key point to provide **R4.1 Capability to analyze the packed file objects**, creating an aggregation system which still can lead to the atomicity. Then, we must care about **R4.2: Capability to analyze an aggregation of files objects**, providing systems to filter and selecting based on mime types, mitigations and software components. R4.2 provides to solve the tasks:

- T0708: Identify threat tactics, and methodologies hidden inside the aggregation
- T0470: Aggregating the right files and their characteristics, some system security posture trends could emerg

As last, during the meeting with the domain experts, we got that the **R5: Capability to provide a concrete summary of the analysis done** was a very high necessity since FACT does not provides a detailed report [27]. R5 satisfies the tasks:

- T0085 : Documentation about what are the securities and insecurities
- T0243 : Documentation about what are the firmware components

ID	TaskID	Requirements description
R1	T0125, T0468, T0509	Capability to see the overview of the most interesting file objects
R2	T0263, T0503	Capability to see the overview of the firmware's status
R3	T0748	Capability to analyze the firmware
R3.1	T0748	Capability to see the overview of the firmware composition
R3.2	T0748	Capability to maintain the focus only on certain aspects during the navigation of the firmware
R4	T0708, T0470	Capability to analyze generic file objects
R4.1	T0708, T0470	Capability to analyze packed file objects
R4.2	T0708, T0470	Capability to analyze an aggregation of files objects
R5	T0085, T0243	Capability to provide a concrete summary of the analysis done

Table 3.8. Requirements description.

3.2 Abstraction

In this section we move to the next step of the *nested layer* approach, trying to abstract the tasks and the data. This is necessary because makes us capable to have clear view about the organizations and choice of the visual analytic techniques. We started designing a generic and abstract process usable by all kind of users and then we modeled the data to have a more generic overview of the FACT-Vis input.

3.2.1 Analysis process

We tried to build some generic analysis process, exploiting what we have done before and designing conceptually relevant aspects.
As starting point, the analyst **must familiarize with the environment**. So, the system should provide a general overview, dealing with requirements **R1** and **R2** :

- Through the *Rank Danger Algorithm* and its customizability, the system provides an overview of the most interesting file objects (**R1**) .

- Through the *aggregate information* of how many weaknesses are present in the critical and suspicious sections, the system provides an overview of the firmware's status (**R2**). Moreover, it's crucial to show the different software which compose the firmware and their possible vulnerabilities. Also, still the *Rank Danger Algorithm* helps to satisfy this requirement.

After the first look on what's going on, the analyst has the needs to explore the main characteristics of the firmware. Here, FACT-Vis must be designed to provide solutions to the requirements **R3, R3.1** and **R3.2** :

- Through the *high visibility* of the firmware's topology and the types of which it is composed, the system provides an overview of the firmware composition (**R3.1**).

- Through the *interaction* of the firmware's topology based on different metrics, the system provides the focus only on certain aspects during the navigation (**R3.2**), highlighting the firmware's morphologies most suited for its tasks.

- The combinations of the previous requirements, jointed to external resources linked by the system, provides the analysis of the firmware as it is **R3**.

After these steps, an user could stop here, concluding a first analysis. In that case, the system must provide a report of all information about the firmware and what the user discovered, satisfying **R5**.

Now, if the analysts wants to go deeper and carve more specific information, it has the needs to deal with the file objects. For this reason, FACT-Vis must be equipped with solutions acted to satisfy the **R4, R4.1** and **R4.2** requirements:

- Through a *technical sheet* which is composed by the collection of all file object's details, the system provides a fine grained view of the file object composition, providing support to analysis of a generic file objects (**R4**).

- Through an ad-hoc *aggregate error collection* of the unpacker plugin, the system provides a way to analyze the packed file objects (**R4.1**)

- The analysis of an files object's aggregation (**R4.2**) is satisfied using two specific subsystems: one for **the software components** and one for the **exploit mitigations** plugins.

Ended the process, FACT-Vis proposed to the analyst how the firmware is made, the "worst" file object with their (customizable) vulnerability order and a final report that encapsulated all the work done.

3.2.2 Cardinality

As second part of the abstraction, we focus on the data. In order to maintain a homogeneous management of all responses provided by FACT and NVD for a single file object , we organized the relationships between the outputs and the file objects . FACT-Vis offers a **File Object's centric** approach, meaning that every step made it is connected to a single component (considering from the global overview of the firmware and their subcomponents).

Single to Single

Trough a **single** instance of the these sources corresponds a **single File Object**:

- From *file_type (1) to FO (1)*
- From *cpu_architecture (1) to FO (1)*
- From *unpacker (1) to FO (1)*

These plugins will be considered as the main discriminant of the single file object, which will be characterized mainly by these plugins. The *file_type* explicitly says what is its origin and can help the user into retrieving specific properties related to its type. *cpu_architecture* (when present) inform the user about what is behind the file type and reveals some info about the systems that supports it and probably where they have been developed. The *unpacker* inform us about what happened during the pecurial phase of the unpacking, revealing personal characteristics of the file.

Multiple to Single

Trough **multiple** instance of the these sources corresponds a **single File Object**:

- From *users_and_passwords (n) to FO (1)*
- From *crypto_material (n) to FO (1)*
- From *exploit_mitigations (n) to FO (1)*

Various information came from these sources and characterize the file object in a unique way. The *users_and_passwords* will sign the file as a container of several sensible and insecure information. The *crypto_material* is similar to the previous, infact it label the files as a storage of several poor protected information. In this case, the information are related to hardcoded and plains private keys or certificates. The *exploit_mitigations* plugin instead will be the only one that (if presents) will reinforce the “image" of the file object. The fact that a file contains one or more mitigations means that the developer (or who is the deputy) had the will to protect the file.
To conclude this phase, we have to cite *software_components*, a particular plugins which is not directly connected to the file object:

- From *cve_lookup (n) to software_components (1)*
- From *NVD (n) to software_components (1)*
  - From *software_components (1) to FO (n)*

These two FACT plugins are strictly dependent if you want to do a file centric system. In fact multiple *cve_lookup (n)* items are connected to the software components which presents these vunerabilities and not directly to the file object. The same if for *NVD* will complete the description of the file object by providing the most up-to-date information about its CVE. A single *software_components* could be composed by several file objects and this is modeled putting n as the cardinality of *FO*.

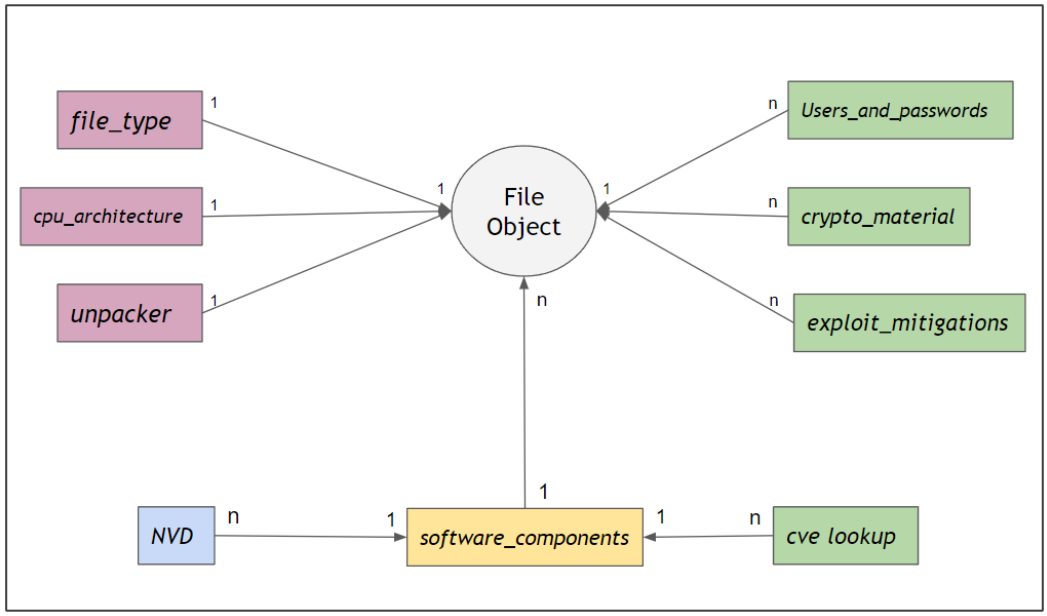

Figure 3.2. Visual representation of the cardinality designed.

3.3 Conceptualization

At this point, the next *nested layer* step is to find the appropriate techniques to **bind the users, the requirements and the data all together**. In this section, we will explore the design conceptualization using the requirements order. During this phase, we kept as mantra : *"Analyse First – Show the Important – Zoom, Filter and Analyse Further – Details on Demand"* .

3.3.1 The Rank Danger

In order to satisfy **R1**, we designed The Rank Danger algorithm **customizable and capable of offers a satisfying first look**. Here the notion of "interesting" file object has been deeply studied. The result was that a file object needs to be complained when it:

- Shows some crypto material like SSH keys or SSL certificates (**CRY**)
- Miss some exploit mitigations (**EXM**)
- Shows plain Users and Passwords (**UPW**)

- Suffers of common vulnerabilities or exposures(**CVE**)
- Has not been unpacked (**PCKD**)
- Is weak to vulnerabilities detected by some YARA rules (**KVU**)

All these elements must be aggregated in order to be comparable to each other, so we transformed them into metrics, each one with a certain value. To catalogue all the file objects we used three different ranks: **critical, suspicious** and **others**. Above all, we put the rule that a file object is always in the rank **suspicious** when it's packed (*PCKD*). This is because the file has not been opened by FACT and we don't know if it can or cannot contains some "interesting" elements. Keeping in mind the flexibility of the algorithm, we put the main rule to use a **threshold** in order to create a delimitation between the critical and the suspicious ranks. All those files which have at least one "interesting" element are evaluated and if their **score is above the threshold, it's ranked as *critical*, otherwise as *suspicious***. Then we choose to put in the rank *other* all those file object for which the algorithm didn't find any metrics. In order to properly aggregate the metrics we choose to weight every metrics with a different (and customizable) value based on empirical results during the implementation. In general, the **formula used to rank each file object is**:

$$W_{cry}*|CRY|+W_{upw}*|UPW|+W_{kvu}*|KVU|+W_{cve}\sum_{i=1}^{|CVE|} score_i+W_{exm}*(EXM_t-|EXM|) \tag{3.1}$$

So every "interesting" element is counted and weighted. The cve metric is also weighted by the *score* given to that specific vulnerability. This formula match different metrics and no other studies has been found that approached this specific problem. So, even if it may not be the optimal one, it still has some interesting implications. Moreover, **it is fully customizable** , permitting the user not only to **change the threshold, but also all the weights and even the score base of the cve**. In this way, every user will highlights the elements that it personally defines "interesting" overshadowing the formula's optimality.

3.3.2 Firmware overview

In order to satisfy **R2**, a way to aggregate information about the weaknesses of the firmware is to **show the components of which it's made of**. The first visualization considered used was a **Heatmap** where each row was a component and 10 columns represented the score of the vulnerabilities. This solution was not too much appreciated for two main reasons. The first was that the if a software component has no vulnerabilities, then it is not well represented. The second was that the cve presents were not well ordered and the result was confusing. The domain experts made us understand that they needed a more simple view like **histograms**. So we decided to build it and we designed it as follows: the view is composed by glyphs which have a histogram based on the count of how many vulnerabilities they have in a scale from 1 to 10. If no flaws are present, still remain the axis with name and the file objects which is composed by. When the user interact with the components, it shows the ordered list of the vulnerabilities. Moreover, the height of the histogram changes based on what the user needs. In fact some components could have a very high number of vulnerabilities with respect to other and the histogram height can be changed to help in visualizing a global or a local spike of flaws.

As last but not least, we decided to **sum up the same metrics of every file object ranked** and showing them through a **histogram**. Moreover, we decided to encode the danger level using the **red** for the critical files, the **yellow** for the suspicious files and the neutral **black** for the other ones. Moreover, the sum of all the Rank Danger Algorithm scores, the number of files object ranked and a list of the mime types composing the firmware, manage to complete and achieve the intended goal.

3.3.3 Firmware analysis

The solution modeled to solve **R3, R3.1** and **R3.2** are strictly connected. So, in this section we will explore all three. For the firmware's composition we need a visualization technique which permits the user to **localize, compare**, and **identify file objects and directories** inside probably huge hierarchies. A first solution was the usage of a Treemap, but some experiments [6] [32] showed up that the

performance of the tasks we needed are equivalent to the ones with a Sunburst visualization, but the last one is easier to learn and more pleasant than a Treemap. Given also the fact the we have to deal with *user A - The Amateur* which is not required to be an expert, we decided to build a **Sunburst**. Moreover, contrary to the Treemap, a Sunburst can add more information to the leaves[1], and this will be exploited by equipping it with a **double visual system** based on the leaves dimensions: one proportional to the **size** and one proportional to its number of **children**.

We define as **composition** of a firmware the conjunction of its **tree structure** and its **tree semantic**. So, the add of the sunburst intrinsically solves at least some aspects of both **R3.1** and **R3.2**. R3.1 is satisfied because we can observe the whole firmware without even interacting. We can **encode the leaves colors with the mime types lists and providing a good tree semantics of the firmware and the double visual system satisfies the tree structure needs**. Now we shift on the resolution of **R3.2**.

In this case, both the encodings and the interaction with the sunburst helps a lot in focusing the attention on certain group or single file object. But this was not enough. In fact, in order to support some search analysis on specific folder or file objects, we used two methods. For the search of a single file object by the hid, we added a **search bar** to find a specific file object. To support the search analysis based on some folder instead, we relied on a classical file directory tree exploration which still remains a valid visualization technique [20]. We boosted it adding a color encoding and the hiding or expanding interactions with the desired folders. This helps in giving the focus only on certain portions of the firmware, supporting **R3.2**. At last, in order to fully satisfy **R3.1** and **R3.2**, we will use an element we introduced in the previous point: the lists of mime types. If enhanced, this list can help us a little bit even for this problem. In fact, in making the aggregation of the same mime types,the systems gives an hint about the firmware's *composition.* respecting the mantra [19], we decided to group the mime types by their prefix and only interacting with them they are expanded, showing all the mimes of a supergroup. To conclude this part and completely satisfy **R3**, we added the cve details and the software components

links to the official websites

3.3.4 Files object analysis

Here we will talk about the solutions used to satisfy **R4, R4.1** and **R4.2**. For what about concern **R4**, we preferred to aggregate all possible information of the specific file object into a single view. Due to the different variable nature of the file objects the best way to expose their data was to provide (always on demand) a **technical sheet** for each file. Here is showed everything about it, more specifically:

- **Hid** - the identifier connected to the file object's virtual file path
- **Uid** - the unique identifier, a unique hex code for every file object
- **Size** - file object's byte dimension
- **Mime** - the MIME type
- **Unpacker** - log of the plug-in (detailed on demand)

Moreover, this sheet is completed adding the possibility to **download** the file object and if it suffers of **crypto material or plain credentials**, it is showed with an expandable system to request the log.
Now we deal with **R4.1**. During the meetings with the domain's expert, we got that a packed file object must be treated in a way different with respect to the others. We already ranked the PCKD files as suspicious but the system must provide more useful information. So we analyzed what are all the possibles output that FACT provides when a file is not unpacked. We found **3 different cases**:

1. The unpacking process has been skipped because it's a **blacklisted** file type
2. The unpacking process raised a **error**
3. The unpacking process stopped because **maximum unpacking depth** was reached

So we decided that the best way to permit the user to understand what the unpacker did (if it didn't unpack anything) was to create an **expandable tree** of always 3 level: The root, the 3 different cases (if present) and the file objects connected to those cases. This visualization is enhanced with a verbose log of the plug-in (if asked). Obviously, this visualization will spawn **only if has been found some packed file** .
Now we have to deal with **R4.2**. In order to do so, we have to comprehend when it make sense talking about **files object's aggregation**. Two or more file objects could be aggregated basically in 3 cases:

1. When they belongs to the same mime types
2. When they are present in the same software component
3. When they share the same exploiting mitigations

For the case 1 we simply count how many mime types exists of a specific kind and show them in the already designed lists of mime type. For the case 2 we add a symbol (depending on the rank) of the file inside the **software component's histograms** visualization. Then, for case 3 we decided to build a dedicated visualization. This was not easy to design because the high number of file objects and the 12 possible mitigations checked by *checksec.sh* leads to a huge explosion of relationships between them. So our solution was to build a **bipartite graph** which connects the file objects to the **mitigations presents in that file**. This was the only way to contain the graph explosion, in fact this way of though is "against" the semantics of the whole system because it shows to the user what the file objects have rather than what they miss.

3.3.5 Conclusion of the analysis

To fulfill the last requirement **R5**, we needed a way to:

1. Flag when a file object is good or not
2. Create a final summary

For what about concerns the first point, we designed a simple icon system to target a file object as **safe** or **dangerous** and store it in the **technical sheet**. For the final summary, we designed to build a **final report** which sums up all the work done by the system and by the user, showing: the metadata, the file types, the exploit mitigations, the software components and the results of the Rank Danger Analysis connected to the flags set by the user.

Now that we ended this step, the next and the last *nested layer* is the solution implementation, which will be described in the next chapter.

Chapter 4

FACT-VIS

The last step from the *nested layer* is the solution implementation. Here, we will illustrate how we managed the building blocks identified in the previous layers, which perform the computations and transformations necessary for the data analysis and how we implemented them. Moreover, we will discuss the discrepancies between what we designed and what we implemented, the reasons behind that and the trade-off bewtween design choices and the implementation needs. The development environment was the first theme that we approached. Both FACT and the NIST offers a very powerful and complete **REST API** service, the first permits to extract all the useful information about the performed analysis, the second provides the complete database of the CVEs. Moreover, Javascript is equipped with **D3.js**, a library for manipulating documents based on data. Its emphasis on web standards gives us the full capabilities of modern browsers without tying building an own framework, combining powerful visualization components and a data-driven approach to DOM manipulation[1]. Once defined the technologies, we explore the main development steps.

[1] https://d3js.org/

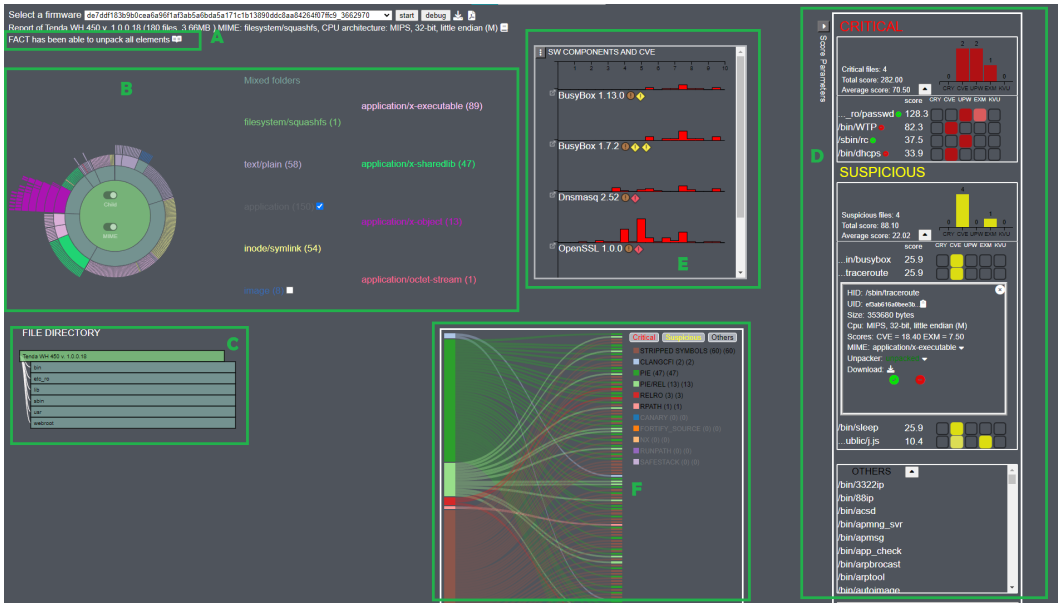

Figure 4.1. FACT-Vis dealing with a real firmware: The unpacker log **(A)** says something on the success ratio of FACT; both the Sunburst with the MIME GUI **(B)** and the File directory **(C)** provide together semantic and syntactic exploration of the firmware loaded; the Rank Danger view **(D)** let the user to understand the details (on several levels of depth) of the analysis; the Software component **(E)** and the Exploit Mitigation **(F)** provide information related to both atomic and aggregated files which complete the analysis.

4.1 The Unpacker Plug-in

The first step of the analysis focuses on the output of the **unpacking** process. What we implemented is an expandable tree of always 3 levels (root, error type and file objects). Moreover we added the log of the unpacker (always *on demand*) in case the analysts needs to manual check the process output.

Figure 4.2. Expandable tree implementation

This implementation satisfies the **R4.1** requirement. Due to the fact that firmware is distributed as binary files and given that these binary files are commonly closed source, users who want to assess the security of the software need to rely on reverse engineering. Once the firmware has been obtained, the process of carving the file system is often completely automated and prone to errors. Sometimes the firmware is encrypted and/or is compressed, some vendors pack their firmware using proprietary packers so even the most powerful unpacker tool can commits mistakes or lacks (Costin et al. [12] successfully unpacked 8,617 firmware out of 23,035 collected firmware images). Using FACT we noticed that some firmware were automatically unpacked, but not in a proper way. **Some corrupted file were produced** during this process. After several analysis, we observed that FACT uses the *generic_carver*

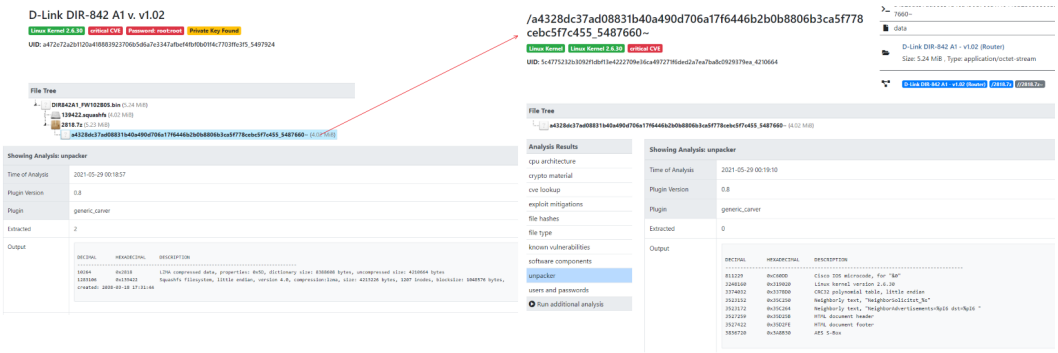

Figure 4.3. An example of a uncorrect unpacking. While FACT return a correct unpacked file, the result produced is a uncromprensible file object which remains not properly packed

unpacking plugin which exploits **Binwalk** and is only intended as fallback if no dedicated plugin is available for unpacking this file type or the unpacking fails. Since Binwalk tries to find the start and end of files in blobs of binary data and this procedure is generally **prone to errors, it can also produce false positives or files with some garbage attached at the end.** In those case FACT was unable to unpack the firmware cleanly, since no unpacking plugin was available for this image format. Has already said, many vendors use proprietary archive or image formats and we would need a separate unpacker for most ones. It seems that no distinct files could be unpacked from that and this **result is rather common**. A

firmware image like that probably contains e.g. a bootloader apart from the file system which was unpacked successfully, so this may be data related to that.
Once got this error and kept in mind, we decided to alert the user when this particular unpacker has been used in order to inform it of a certain probability that this event happened.

4.2 Rankdanger Algorithm

For what about concern the **rank danger algorithm**, the idea was to apply the formula seen in 3.1. Here the pseudocode used to rank every file object:

```
global var THRESHOLD, W_CRY, W_UPM, W_EXM, W_KNV, W_CVE_CRIT,W_CVE_CRIT
function RDA(fw) {//
  var Critical,Suspicious,Other
  for each fo in fw{
    var score = RSingle(fo)
    if(score > THRESHOLD)
      Critical.add(fo)
    else if (score == "packed" or (score > 0 and score < THRESHOLD) )
      Suspicious.add(fo)
    else
      Other.add(fo)
  }
}
function RSingle(fo){
  var cry,upw,exm,cve,w_cve,knv,score = 0
  #packed case
  if (fo is packed)
    return "packed"
  #crypto, user_n_password and known_vulnerabilities case
  for each crypto in fo.cry
    cry += W_CRY
  for each plain in fo.upw
    upw += W_UPM
  for each flaws in fo.knv
    knv += W_KNV

  #cve case
```

```
  if(fo.sw_comp is critical)
    w_cve = W_CVE_CRIT
  else:
    w_cve = W_CVE_N_CRIT
  for each flaws in fo.sw_comp
    cve += flaws[SCORE_TYPE] * w_cve

  #exploit mitigation case
  for each mitigation in fo.exm
    exm++
  exm = (11-exm) * W_EXM

  score = cry+cve+upw+knv+exm
  return score
}
```

In order to **prioritize the files**, we built three lists : *Critical* ,*Suspicious* and *Other*. If a file is **packed, it is automatically labeled as suspicious** and added to the relative list. Otherwise, the score evaluation starts and for *crypto, users_n_password* and *known_vulnerabilities* it is evaluated simply adding the related weight for how many flaws are detected. For what about concern the Cve, once got the right weight (based on the critical level) and the right score type (base_score, impact_score, exploitability_score), we perform the sum. Then, after got the sum of all mitigations found, the formula (3.1) returns the score. In order to aggregate all metrics, we used a matrix-based approach, where every row represent a file object and every column the relative metric. If a metric is not present the cell will not be filled. Otherwise, the the cell will have the color intensity (red if critical and yellow if suspicious) proportional to the value of the metric: higher is the score, more intense is the color. For both the critical and the suspicious, we added a bar chart which aggregate the flaws found and shows the average and the total score of the ranks. We made this to satisfy the **R2** requirements. The visual solution of this result, jointed with the one considered to satisfy the **R2** requirement, is composed by two subsystems.

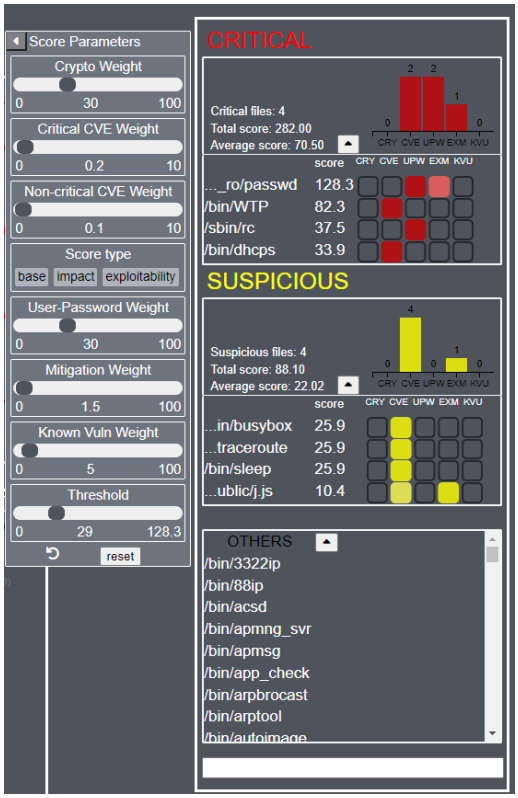

Figure 4.4. How is visualized the RDA result

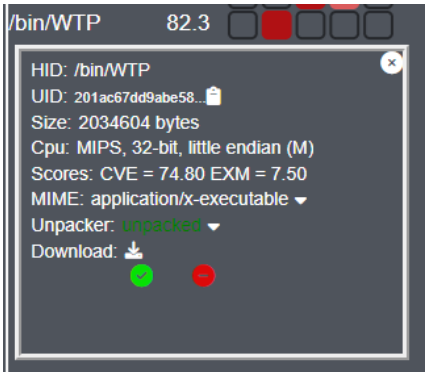

Figure 4.5. The technical data sheet

As shown in Figure 4.4, with a side expandable menu the user can change the algorithm's parameters in real time and see what are the changes. At the bottom is present a search bar in order to find the file object by its hid. Moreover, clicking on the file object name, the user can access to the personal file information through the **technical data sheet** (as shown in Figure 4.5). Specifically, the information are:

- Hid;
- Uid;
- Bytes dimension;
- Architecture;
- Score composition
- MIME type;
- Crypto material (if present);

- Cracked material (if present);
- Unpacker output (expandible on demand);
- Download option output;

Moreover, this view is provided of two icons deputed to flag the file as dangerous or safe clicking on the respective icon. This was necessary to satisfy the **R5** requirement.

4.3 Firmware navigation

If the user wants to contextualize the file objects and their flaws or if it wants to freely explore the firmware, the **Sunburst** view let the user be capable to do it. Specifically, the sunburst permits to **localize, compare, and identify file objects and directories** .

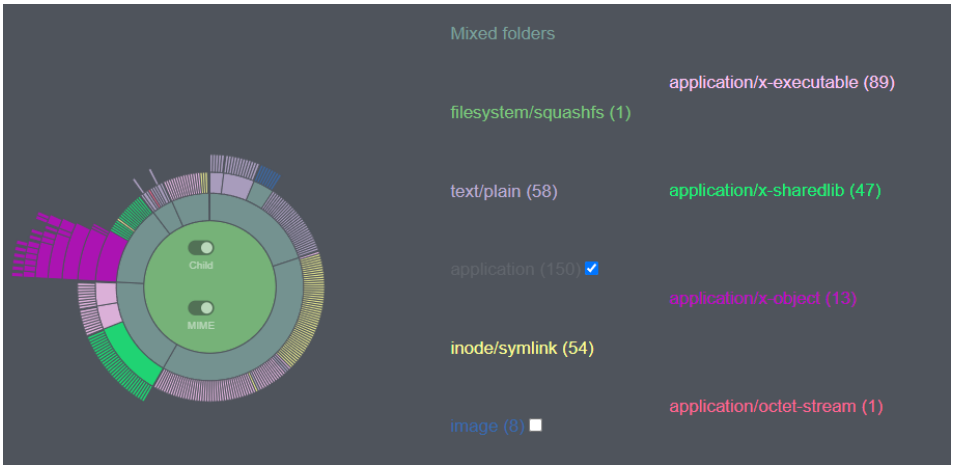

Figure 4.6. the Sunburst and the mime list. The toggle on top permits to switch the double visual system and the bottom one change the color encoding between mime and rank

The sunburst is **completely interactive**, with the possibility to focus only on certain kind of files based on the type, which can be expanded (as shown in Figure 4.6) or removed (to filter and have a more clear view). These two elements solve the **R3.1** and partially **R3.2** requirements. In the figure, the sunburst node dimensions are proportional to the number of children, except for the leaves which have a unit dimension. Hovering each node, some general information like the rank, the name

and other information will rise. Clicking on the “leaves” checkbox, the sunburst changes in a way such that the node are proportional to the size. This feature implements the **double visual system** which we discussed previously and help to satisfy the requirements taken in subject. Moreover, we decided to add a new feature which consist into encode the color of the sunburst’s node with the one related to their **rank** exception made for the folder which are encoded with a neutral color (the same for the mime list). This permits the user to get an immediate overview on what are the problematics of the firmware studied. As already said in the design chapter, the sunburst is not enough in supporting some search analysis on specific folder or file objects. For this reason we implemented a **file directory tree** in order to provide better exploration capabilities.

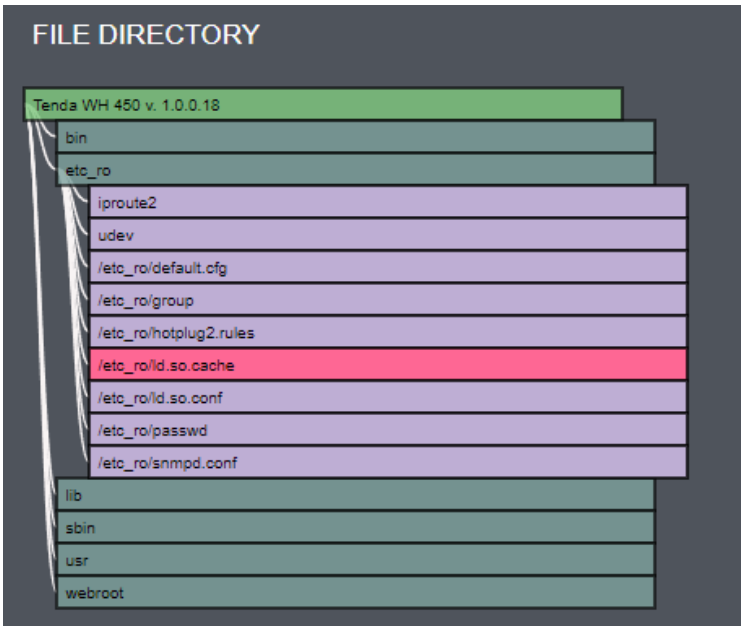

Figure 4.7. the file directory

Similarly to the classic paradigm used to navigate the file system, each node is expandable and retractable and every leaf is connected with all the other views. Moreover as shown in Figure 4.7, in order enhance this kind of view, we added a color encoding, which follow the same of the sunburst. In this way the user can explore and navigate the file directory keeping always in mind the typology of the folder which is traversing. This view is synchronized with the sunburst in order to

help into performing a better exploration and a more fluid analysis.

4.4 Software components and CVE

Due to the peculiar **cardinality** between cve, software components and the file objects which is unique, we decided to build a specific view. This consists into a list of software components and one or more icon which represent the file(s) composing the item (encoded with the rank colors). For the components with at least one vulnerability, the view is composed by a histogram.

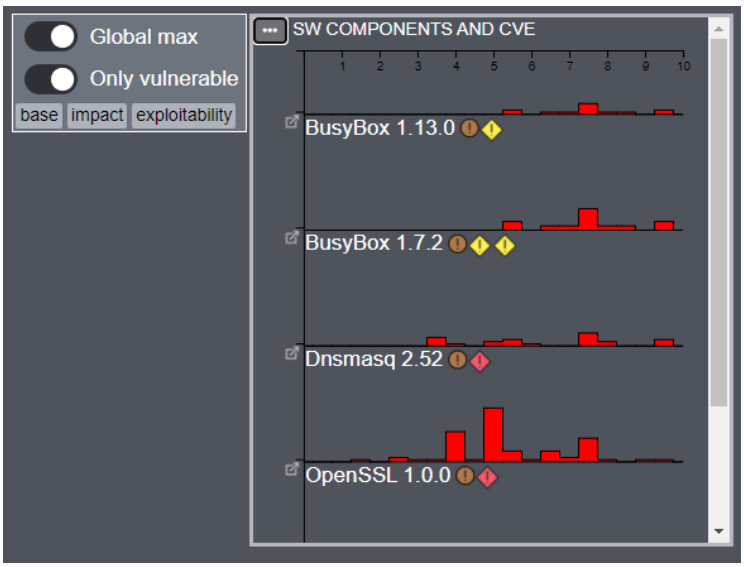

Figure 4.8. The view dedicated to the software components and its relative menu, as always, *on demand*

As shown in figure 4.8, the bars height is proportional to the number of CVE that a certain component has. Those CVEs are collected through the REST APIs provided by the NVD, searching the collection of CVEs for a certain component. **The collection of all CVEs is made every time the user starts an analysis**. This means that if a new collection of vulnerabilities are published on the NVD system, it is always **up-to-date**.

During the meetings with the domain experts raised the needs to permit the user to change some parameters in order to focus only on some elements. A first feature added was the change in real time (as the rank danger algorithm parameter) of the

score type. Then, we added a filter to **hide the non vulnerable** components.

We notice that sometimes this was not enough, in fact some components had so much flaws that overshadowed the others, so we decided to implement a new function for which change the logic behind the height of the histograms and let the user to support both global and local component analysis. To conclude, we added the possibility to click on the component to remove it, the critical icon when a component has this property and the link to the website, in order to completely satisfy **R3**. In a user want to explore the components, it must only hover the mouse on the interested part of the histogram and all the cve will appears. If the user want to see them on details, it have to click in order to lock the tooltip and then can explore all cve which are clickable and are redirected to the **its specific NVD page** which provides all the details.

4.5 Exploit mitigation

In order to complete the **R4.2** : Capability to analyze an aggregation of files objects, we needed a way to show the exploit mitigations data found in the application file type and we decided to use a **bipartite graph**.

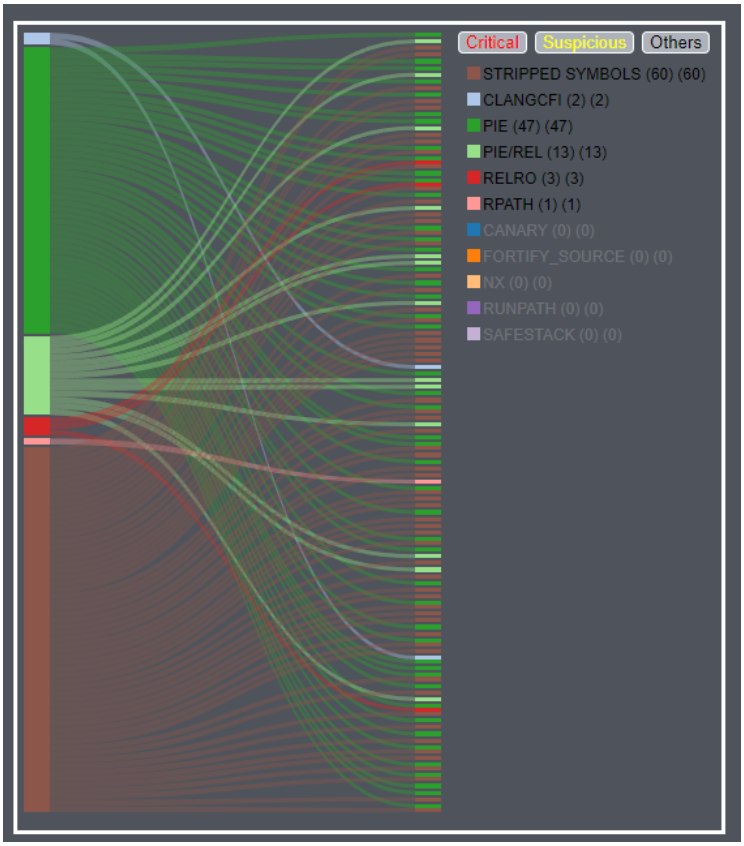

Figure 4.9. The view dedicated to the exploit mitigations

As visible in Figure4.9 , on the left we aggregated the mitigations which are encoded following the colors set in the legenda located in the far right. The legenda shows the number of items with that specific mitigation and provide a semantic zoom when mouseovered. The file object are encoded accumulating the mitigation e.g. a file with 3 mitigation will be represented with 3 colored bars each one related to its mitigation's color. On top of the legenda, the user can choose the level of aggregation by selecting the rank. It's important to notice that this visual solution suffers of a particular problem which is related to the number of file object that are viewable. In fact, if a firmware contains too much file objects with mitigations and the user want to analyzie all three ranks, the view will explode providing some computational delay and a impossibility to proceed further in the analysis. For this reason we recommend to show both three rank only when a not so much file objects mitigation-provided are present. Notice also that the semantics of this view

is slightly detached from the others: the visualization shows what are the mitigations that the file has and not the ones which miss. This is because we noticed empirically that a file object tends to miss lots of mitigations and a bipartite graph with this semantic should have degraded the whole system in terms of performaces.

4.6 Usage scenarios

With different levels of expertise, FACT-Vis can have various goals and so be used by different type of users. Thus, we have built different hypothetical user analysis based on the NICE Framework; in addition, in the simplest scenario we consider a completely newbie starts the system and checks if some file objects are ranked as critical. If so, the system suggests that the firmware contains some problems and need to be seen by a more expert user.

4.6.1 Personal router safety

For this last case, we do not consider the NICE framework anymore. Instead we suppose that a private citizen which is passionate of cyber security, has been capable of extract his **Tenda WH 450's firmware** and now he wants to analyze it in order to check if his router has serious flaws. So, after gave the firmware to FACT, he **boots FACT-Vis**. The first thing that he notices is that the firmware has been **completely unpacked**. This information means that the firmware has been completely elaborated by FACT-Vis and there **will not be any shadow on the analysis**.

After briefly **explored** the firmware structure by using the sunburst and the mime types which is composed by, he wants to **check its weaknesses**. FACT-Vis ranked **four file objects as critical**, so they require the utmost attention. He notices that both */etc_ro/passwd* and */sbin/rc* have been targeted as critical because **plain credentials** has been found. The user decides that it's only about user passwords which they are **simple replaceable** with some stronger ones, and decides to flag the file objects has **safe**, keeping in mind that he needs to change them.

Then he remembered reading on internet that sometimes it's useful to check file with

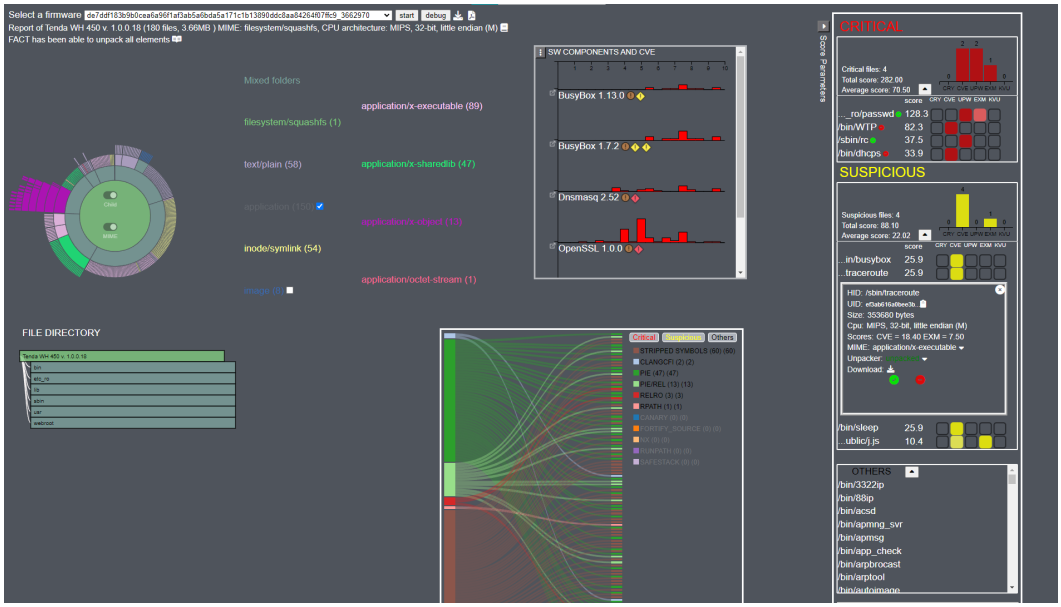

Figure 4.10. What Fact-Vis shows

big dimensions. So, swapping the sunburst visualization with the one that highlights the **size**, he notices that the biggest file is not been considered as dangerous. Instead, the second biggest file **/bin/WTP** is a **Critical** one! So, after checking that its vulnerable to various and several **vulnerabilities** and the software component from which belongs, it flags it as **dangerous**. The same does with the other critical file: */bin/dhcps*. For him, this is enough to conclude that **its firmware is vulnerable** and he can now download the report and spread it through internet or send it to the router vendor. This scenario has been made to highlight the wide range spectrum of the users provided by FACT-Vis: the layered design permits to cohabit together novices (which can access to some useful information through a superficial examination), with a skilled expert (which can track trends, tag file objects and perform a more intense work of analysis).

4.6.2 An Executive Cyber Leader reviews a router

We suppose that an ECL - Executive Cyber Leader needs to perform the analysis on a **Netgear R8300 firmware** in order to review it and **establishing a vision and a direction for this resource**. We recall that this kind of user needs to perform these two taskes:

Figure 4.11. The Unpacker log

- T0263: Identify security requirements specific to an IT system in all phases of the system life cycle.
- T0509: Perform an information security risk assessment.

So, after gave the firmware to FACT, he **boots FACT-Vis**. The system reports that it could not properly analyze the system because **FACT has not been able to unpack 6 elements**. So he starts analyzing the log of the unpacker plugin which reports:

```
! WARNING: Unknown squashfs version.
```

Moreover, opening the tree log, he notices that an **undefined error** raised up. The file for which this error raised is */usr/sbin/rule.trf* (see Figure 4.11) So, this concludes that the analysis made by the system **will contains some shadows** and the only way to deal with the packed files is to **download** them, performing **T0509**. However, the ECL wants to study more deeply the firmware in order to do a valid review and send it to the vendor which will fix the issues.
So, after exploring the morphology with the **mime type** list, he starts browsing the **File directory**, in order to check in the deepest way the topology of the firmware. From this chart, he notices that the folder */segment1* contains a **strange couple of file object**: */34b9529fa009b11ae83527bbec6869f14c6328cb46fb7b26739533299afe16762206580* and */22000.cpio*. Hovering the cursor on the sunburst (as shown in Figure 4.12 he notices that both are **ranked as Critical** from the system, so the user moves the attention on the rank danger view and notices that the file and its child :

- Are the ones with the highest score (*1628.5*), which is about 20 times more than */usr/local/sbin/openssl*, the third file with the highest score.

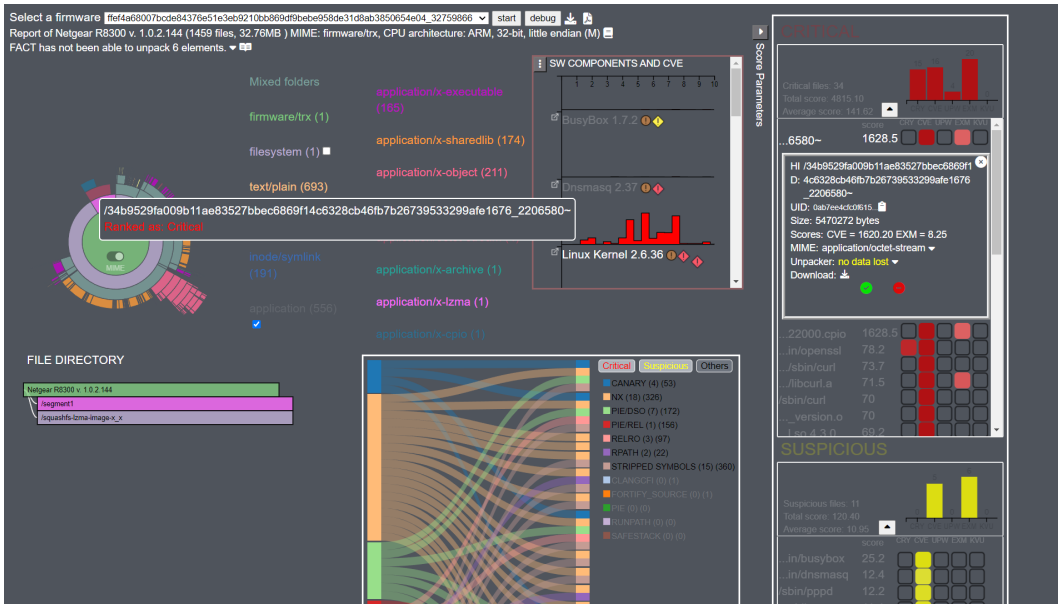

Figure 4.12. The step described in the scenario. Notice how FACT-Vis tries to focus the attention of the user only on the targeted file object

- The score is only due to a **CVE** problem.
- The unpacker used is a *generic carver* and the log is verbose wrt other files.
- *.cpio* is a compressed extension.
- Are not provided by any mitigation
- Both are the components of the *Linux Kernel*

As said in the unpacker section, the usage of a generic carver could suggest a production of some indistinct files during the unpacking process for what about concern the folder */segment1*, in fact, the **yellow label** of the unpacker, once expanded suggests this kind of event. These 2 elements are the components of a **Linux Kernel** which is full of vulnerabilities with **a spike of 56 vulnerabilities** in the range 9-10 and 758 in the range 7-8, this means that the risk assessment to do to accomplish **T0509** is that the **firmware is extremely dangerous**.

In order to identify all firmware's software components, he interacts with the relative menu and after changing the max into local (in order to have a better overview because the linux kernel prevailed too much above the others), sees all the

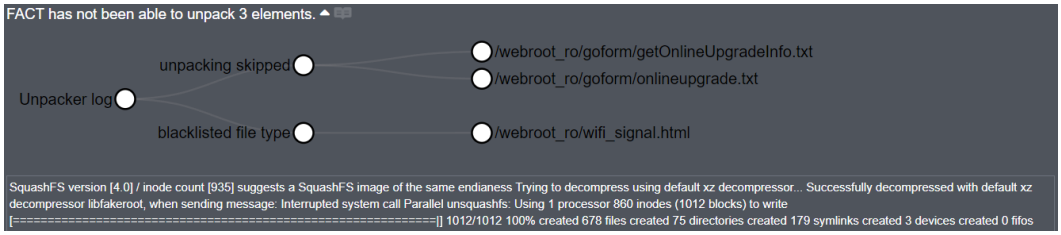

Figure 4.13. The unpacker log

components. After this examination, the user set the **parameters** and filters all CVE, finds out that the only suspicious files are the one packed and above *19 critical files*, 4 suffers of cracked credentials, and 15 of exposed crypto material. At last he analyzes the mitigations in order to have a general overview of their frequency and accomplish the whole **T0263** To conclude the review, he generates the report and send it to the vendor with all these conclusions attached in order to be represented and reanalyzed once these flaws are fixed.

4.6.3 A firmware analysis from a Cybersecurity Defense expert

We suppose now that a CDA - Cybersecurity Defense Analyst needs to study the new version of the **Tenda AC18 AC1900** router's firmware developed from its company in order to **identifies, analyzes, and mitigates threats**. We recall that a CDA has the goals of:

- T0470: Analyze and report system security posture trends.
- T0503: Monitor external data sources to maintain currency of cyber defense threat condition and determine which security issues may have an impact on the enterprise

So these tasks requires a more deep examination. After the FACT elaboration, the user boots FACT-Vis and the first thing notices is that the unpacking have gone relatively well. As seen in Figure 4.13, FACT has not been able to unpack only 3 elements over 658 and they are only readable files. This means that the **whole analysis will not contains shadows** provided by the packed elements. The first view provided by the sunburst is very poor of information. In fact, contrarily to the previous firmware, due to the atomic files which are not aggregated, the leaves

of the visualization are **not completely distinguished** and doesn't let the user to understand the topology. For this reason the user starts exploring the firmware through the file directory in search of some hints. Once he got how is the structure, he switch the sunburst visual system and gets new information about **how the file types are spread across the system** . He decides to change the encoding and seeing the one ranked but before doing that, he **changes the parameters** and filtered Users and password and crypto, in order to find information only about CVE and other vulnerabilities together with the mitigations. Then, the CDA changes the encoding and notices a very interesting fact: **two of the biggest files results ranked as critical**. The Figure 4.14 shows all the steps.

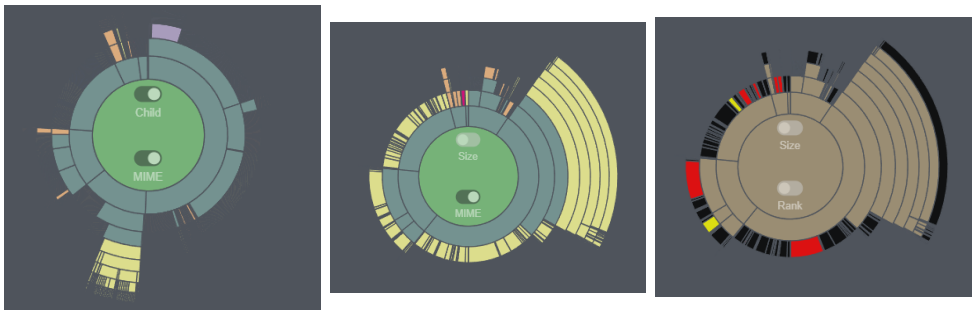

Figure 4.14. The sunburst shows different information based on the encoding and the double visual system

So to partially accomplish **T0470** a first trend, which links the size to the CVEs has been found. Then he focus the attention on the exploit mitigation views and the visualization suggests that

- **NX and STRIPPED SYMBOLS are the most frequent mitigations among all files**.
- **The STRIPPED SYMBOLS is second the most common mitigation found**
- **Among 54 files found with the PIE-REL mitigation, no one has been ranked between the critical and suspicious files**

as visible in Figure 4.15 All these consideration, jointed with the report, will satisfy the CDA which will proceed into describing the these steps done and present them

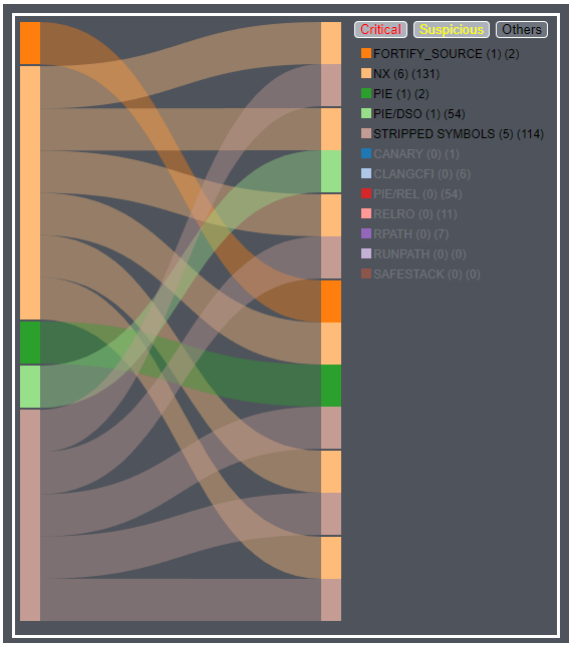

Figure 4.15. How the exploit mitigation view appears

to whom it may concern. At last, to completely satisfy the task **T0503** he can emulate the firmware following the OWASP steps, using for example the FAT tool which we described in the state of the art chapter.

Chapter 5

Conclusions

We presented **FACT-Vis: a visual tool for the analysis and security of firmware**. We started describing the field of action and the domain, trying to explain what are the difficulties of this subject and moreover the needs that the firmware analysis requires. Then we exposed the current guidelines present in the state of art and what tools these guidelines suggests, focusing on the very detailed examination of FACT - the Firmware Analysis and Comparison Tool used as starting point for our work. Then, in order to put the reader in the right context and let him know the domain as its best, we presented spurious works done both in technical projects and visual techniques. Following the *nested model*, we designed our tool and, with the help of domain's experts, we described this process starting from the profiling and the requirement collection. Then we abstracted the problem in order to have a more margin to compare the various tasks to deal with and then we conceptualized the work, concocting some tangible solutions aimed to bind what we did before. At the end we presented some phases of the implementation and some difficulties encountered during this process, closing with a possible usage scenario. **FACT-Vis** is a transverse project, which connect two different fields of study: **the firmware analysis**, which lives in a low level context concerning the security of both hardware and software domain and the **visual analytics** which manipulate data in order to be shown in the most efficient way. The tool proposed tries at his best to accomplish its purposes and facilitate the user to analyzes the flaws of a firmware. This has been made because the security of our systems has to be preserved for the

good of the freedom and the transparency. The future of **FACT-Vis** is aimed to maintain its stability and boost its performances but moreover, it's aimed to be the most user-friendly possible, without losing its core goal: provide balance between the different users in performing firmware analysis. FACT-Vis works only on some steps of the OWASP methodology and will be interesting to do a future research which will cover the other steps and will exploit the information provided in order to be comprehensive for the next steps. Also, a study about how to enrich the dynamic firmware analysis exploiting the static information and integrate them using other external sources will be interesting. Basing on the feedback, some new features are ready to be deployed such as a custom server to manual analyze the file objects rather than a different layout disposition. FACT-Vis born to be a user tool and not a user barrier in the already full of obstacles firmware analysis field of study.

Bibliography

[1] Marco Angelini et al. “SymNav: Visually Assisting Symbolic Execution”. In: *2019 IEEE Symposium on Visualization for Cyber Security (VizSec)*. Vancouver, BC, Canada: IEEE, Oct. 2019, pp. 1–11. ISBN: 9781728138763. DOI: 10.1109/VizSec48167.2019.9161524. URL: https://ieeexplore.ieee.org/document/9161524/ (visited on 02/23/2022).

[2] Marco Angelini et al. “Vulnus: Visual Vulnerability Analysis for Network Security”. In: *IEEE Transactions on Visualization and Computer Graphics* 25.1 (Jan. 2019), pp. 183–192. ISSN: 1077-2626, 1941-0506, 2160-9306. DOI: 10.1109/TVCG.2018.2865028. URL: https://ieeexplore.ieee.org/document/8443131/ (visited on 03/07/2022).

[3] Hala Assal, Sonia Chiasson, and Robert Biddle. “Cesar: Visual representation of source code vulnerabilities”. In: *2016 IEEE Symposium on Visualization for Cyber Security (VizSec)*. Oct. 2016, pp. 1–8. DOI: 10.1109/VIZSEC.2016.7739576.

[4] *binvis.io*. URL: https://binvis.io/#/.

[5] Jonathan Braley. *Increased CVE Counts: A Positive Indicator of a Maturing Security Ecosystem*. en-US. URL: https://www.icasi.org/increased-cve-counts-a-positive-indicator-of-a-maturing-security-ecosystem/.

[6] Pierre Caserta and Olivier Zendra. “Visualization of the Static Aspects of Software: A Survey”. In: *IEEE Transactions on Visualization and Computer Graphics* 17.7 (July 2011), pp. 913–933. ISSN: 1941-0506. DOI: 10.1109/TVCG.2010.110.

[7] *Challenges — FACT documentation*. URL: https://fkie-cad.github.io/FACT_core/main.html.

[8] *checksec.sh*. URL: https://www.trapkit.de/tools/checksec/.

[9] Daming D. Chen et al. "Towards Automated Dynamic Analysis for Linux-based Embedded Firmware". In: *NDSS*. 2016. DOI: 10.14722/NDSS.2016.23415.

[10] Luke Chen. *Creating relocatable Linux executables by setting RPATH with $ORIGIN*. en. Nov. 2021. URL: https://nehckl0.medium.com/creating-relocatable-linux-executables-by-setting-rpath-with-origin-45de573a2e98.

[11] *Control Flow Integrity — Clang 15.0.0git documentation*. URL: https://clang.llvm.org/docs/ControlFlowIntegrity.html.

[12] Andrei Costin et al. "A {Large-Scale} Analysis of the Security of Embedded Firmwares". en. In: 2014, pp. 95–110. ISBN: 9781931971157. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/costin.

[13] A. Cui and S.J. Stolfo. "A quantitative analysis of the insecurity of embedded network devices: Results of a wide-area scan". In: cited By 114. 2010, pp. 97–106. DOI: 10.1145/1920261.1920276. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-78751540482&doi=10.1145%2f1920261.1920276&partnerID=40&md5=dd015e9d09289b9eaba04dd243963a25.

[14] *FindBugs™ - Find Bugs in Java Programs*. URL: http://findbugs.sourceforge.net/.

[15] *Firmwalker – Craig Smith*. en-US. URL: https://craigsmith.net/firmwalker/.

[16] *Ghidra*. URL: https://ghidra-sre.org/.

[17] Eric Gustafson et al. "Toward the Analysis of Embedded Firmware through Automated Re-hosting". en. In: 2019, pp. 135–150. ISBN: 9781939133076. URL: https://www.usenix.org/conference/raid2019/presentation/gustafson (visited on 03/07/2022).

[18] *Hardening ELF binaries using Relocation Read-Only (RELRO)*. en. URL: https://www.redhat.com/en/blog/hardening-elf-binaries-using-relocation-read-only-relro.

[19] D.A. Keim et al. "Challenges in Visual Data Analysis". In: *Tenth International Conference on Information Visualisation (IV'06)*. ISSN: 2375-0138. July 2006, pp. 9–16. DOI: 10.1109/IV.2006.31.

[20] A. Kobsa. "User Experiments with Tree Visualization Systems". In: *IEEE Symposium on Information Visualization*. ISSN: 1522-404X. Oct. 2004, pp. 9–16. DOI: 10.1109/INFVIS.2004.70.

[21] Jingyue Li et al. "An Empirical Study on Off-the-Shelf Component Usage in Industrial Projects". In: *Product Focused Software Process Improvement*. Ed. by Frank Bomarius and Seija Komi-Sirviö. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 54–68. ISBN: 978-3-540-31640-4.

[22] Peter Mell, Karen Scarfone, and Sasha Romanosky. "Common Vulnerability Scoring System". In: *IEEE Security Privacy* 4.6 (Nov. 2006), pp. 85–89. ISSN: 1558-4046. DOI: 10.1109/MSP.2006.145.

[23] *New Supply Chain Vulnerabilities Impact Medical and IoT Devices*. en-US. Mar. 2022. URL: https://www.forescout.com/blog/access-7-vulnerabilities-impact-supply-chain-component-in-medical-and-iot-device-models/ (visited on 03/14/2022).

[24] *NVD - Home*. URL: https://nvd.nist.gov/ (visited on 02/28/2022).

[25] *Offensive IoT Exploitation*. URL: https://www.attify-store.com/products/offensive-iot-exploitation.

[26] *OWASP Internet of Things | OWASP Foundation*. en. URL: https://owasp.org/www-project-internet-of-things/.

[27] *PDF report with details · Issue #498 · fkie-cad/FACT_core*. en. URL: https://github.com/fkie-cad/FACT_core/issues/498.

[28] *PIE and RELRO*. URL: https://www.trapkit.de/articles/relro/.

[29] *QEMU*. URL: https://www.qemu.org/ (visited on 03/01/2022).

[30] *Quick Start Guide · ReFirmLabs/binwalk Wiki*. en. URL: https://github.com/ReFirmLabs/binwalk.

[31] *SafeStack — Clang 15.0.0git documentation*. URL: https://clang.llvm.org/docs/SafeStack.html.

[32] JOHN Stasko et al. "An evaluation of space-filling information visualizations for depicting hierarchical structures". en. In: *International Journal of Human-Computer Studies* 53.5 (Nov. 2000), pp. 663–694. ISSN: 1071-5819. DOI: 10.1006/ijhc.2000.0420. URL: https://www.sciencedirect.com/science/article/pii/S1071581900904208 (visited on 02/23/2022).

[33] Pengfei Sun et al. "Hybrid Firmware Analysis for Known Mobile and IoT Security Vulnerabilities". In: *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. ISSN: 1530-0889. June 2020, pp. 373–384. DOI: 10.1109/DSN48063.2020.00053.

[34] Sam L. Thomas, Flavio D. Garcia, and Tom Chothia. "HumIDIFy: A Tool for Hidden Functionality Detection in Firmware". en. In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by Michalis Polychronakis and Michael Meier. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 279–300. ISBN: 9783319608761. DOI: 10.1007/978-3-319-60876-1_13.

[35] Christian Tominski and Heidrun Schumann. *Interactive visual data analysis*. 1st ed. Boca Raton: CRC Press, 2020. ISBN: 9780367898755 9781498753982.

[36] Philipp Trinius et al. "Visual analysis of malware behavior using treemaps and thread graphs". In: *2009 6th International Workshop on Visualization for Cyber Security* (2009). DOI: 10.1109/VIZSEC.2009.5375540.

[37] Aiko Yamashita. "Experiences from performing software quality evaluations via combining benchmark-based metrics analysis, software visualization, and expert assessment". In: *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Bremen, Germany: IEEE, Sept. 2015, pp. 421–428. ISBN: 9781467375320. DOI: 10.1109/ICSM.2015.7332493. URL: http://ieeexplore.ieee.org/document/7332493/.

[38] *YARA - The pattern matching swiss knife for malware researchers.* URL: http://virustotal.github.io/yara/.

[39] Xiaojin Zhu and Andrew B. Goldberg. "Introduction to Semi-Supervised Learning". In: *Synthesis Lectures on Artificial Intelligence and Machine Learning* 3.1 (Jan. 2009), pp. 1–130. ISSN: 1939-4608. DOI: 10.2200/S00196ED1V01Y200906AIM006. URL: https://www.morganclaypool.com/doi/abs/10.2200/S00196ED1V01Y200906AIM006 (visited on 03/07/2022).