DNS research proposal - A Smart Approach to Infection Analysis and Infected ELF Binaries Disinfection

Pietro Mazzini

2021–2022

Abstract

The aim of this research proposal is to study and theorize a machine learning approach to ELF binaries structure anomaly detection. The neural network proposed should detect crafted or infected executable files and visually underline anomalies found to help researchers understand viruses behaviors and newly exploited binary crafting techniques. The second part of the research aim is restoring infected ELF binaries to their initial, harmless, stage.

1 Introduction

1.1 Research area

About a month ago (April 2021) the first edition of the e-zine **TMP.out** [19] was published; this e-zine focuses on ELF binary study and research. Some of the papers are *"Dead Bytes" by xcellerator* [9], containing references to *"ELF Binary Mangling" by netspooky* [4][5][6], *"Implementing the PT_NOTE Infection Method in x64 Assembly" by sblip* [8][2][1] and *"PT_NOTE Disinfector" by manizzle* [3].

Summarizing, these papers are about elaborate and artistic methods to embed in ELF binaries malicious code that gets eventually executed, substantially what a parasite virus does, and advanced crafting techniques to create malformed executable files.

1.2 General idea

In this research proposal is presented a machine learning model able to, given in input an ELF binary file and nothing more:

1. Detect if the ELF has been infected by a virus (or if it is malformed)

2. If the ELF is infected determine which portions and bytes have been modified, providing a rich and handy structure overview

3. Restore the ELF file content and behavior to the original state

1.3 Background

ELF binary structure The ELF binary format (Executable and Linkable Format) is a standard, cross platform, file format for executable files, object code, shared libraries, and core dumps. The ELF starts with the ELF header (Figure 1), this holds a road map describing the file's organization.

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x5b20
  Start of program headers:          64 (bytes into file)
  Start of section headers:          140208 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         11
  Size of section headers:           64 (bytes)
  Number of section headers:         27
  Section header string table index: 26
```

Figure 1: Example ELF header of `ls` ELF binary.

The ELF header contains references to the program header table (Figure 2) and the section header table (Figure 3). The program header table holds the offsets of the various segments of which the ELF is composed; there are different types of segments, the main function of these is containing loadable code that will be used to build the process memory image (Figure 4). The section header table is instead used to list sections, these hold information about linking and relocation.

```
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x0000000000000268 0x0000000000000268  R      0x8
  INTERP         0x00000000000002a8 0x00000000000002a8 0x00000000000002a8
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000003510 0x0000000000003510  R      0x1000
  LOAD           0x0000000000004000 0x0000000000004000 0x0000000000004000
                 0x00000000000133d1 0x00000000000133d1  R E    0x1000
  LOAD           0x0000000000018000 0x0000000000018000 0x0000000000018000
                 0x00000000000008cc0 0x00000000000008cc0  R      0x1000
  LOAD           0x0000000000020fd0 0x0000000000021fd0 0x0000000000021fd0
                 0x0000000000001298 0x0000000000002588  RW     0x1000
  DYNAMIC        0x0000000000021a58 0x0000000000022a58 0x0000000000022a58
                 0x0000000000000200 0x0000000000000200  RW     0x8
  NOTE           0x00000000000002c4 0x00000000000002c4 0x00000000000002c4
                 0x0000000000000044 0x0000000000000044  R      0x4
  GNU_EH_FRAME   0x000000000001d324 0x000000000001d324 0x000000000001d324
                 0x0000000000000954 0x0000000000000954  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000020fd0 0x0000000000021fd0 0x0000000000021fd0
                 0x0000000000001030 0x0000000000001030  R      0x1
```

Figure 2: Example Program header of `ls` ELF binary.

```
Section Headers:
  [Nr] Name              Type            Address          Offset
       Size              EntSize         Flags  Link  Info  Align
  [ 0]                   NULL            0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS        00000000000002a8  000002a8
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.gnu.bu[...] NOTE            00000000000002c4  000002c4
       0000000000000024  0000000000000000   A       0     0     4
  [ 3] .note.ABI-tag     NOTE            00000000000002e8  000002e8
       0000000000000020  0000000000000000   A       0     0     4
  [ 4] .gnu.hash         GNU_HASH        0000000000000308  00000308
       00000000000000b0  0000000000000000   A       5     0     8
  [ 5] .dynsym           DYNSYM          00000000000003b8  000003b8
       0000000000000c00  0000000000000018   A       6     1     8
  [ 6] .dynstr           STRTAB          0000000000000fb8  00000fb8
       00000000000005bc  0000000000000000   A       0     0     1
  [ 7] .gnu.version      VERSYM          0000000000001574  00001574
       0000000000000100  0000000000000002   A       5     0     2
  [ 8] .gnu.version_r    VERNEED         0000000000001678  00001678
       0000000000000080  0000000000000000   A       6     1     8
  [ 9] .rela.dyn         RELA            00000000000016f8  000016f8
       0000000000001e00  0000000000000018   A       5     0     8
  [10] .rela.plt         RELA            00000000000034f8  000034f8
       0000000000000018  0000000000000018  AI       5    22     8
  [11] .init             PROGBITS        0000000000004000  00004000
       000000000000001b  0000000000000000  AX       0     0     4
  [12] .plt              PROGBITS        0000000000004020  00004020
       0000000000000020  0000000000000010  AX       0     0     16
  [13] .text             PROGBITS        0000000000004040  00004040
       0000000000013382  0000000000000000  AX       0     0     16
  [14] .fini             PROGBITS        00000000000173c4  000173c4
       000000000000000d  0000000000000000  AX       0     0     4
  [15] .rodata           PROGBITS        0000000000018000  00018000
       0000000000005321  0000000000000000   A       0     0     32
  [16] .eh_frame_hdr     PROGBITS        000000000001d324  0001d324
       0000000000000954  0000000000000000   A       0     0     4
  [17] .eh_frame         PROGBITS        000000000001dc78  0001dc78
       0000000000003048  0000000000000000   A       0     0     8
  [18] .init_array       INIT_ARRAY      0000000000021fd0  00020fd0
       0000000000000008  0000000000000008  WA       0     0     8
  [19] .fini_array       FINI_ARRAY      0000000000021fd8  00020fd8
       0000000000000008  0000000000000008  WA       0     0     8
  [20] .data.rel.ro      PROGBITS        0000000000021fe0  00020fe0
       0000000000000a78  0000000000000000  WA       0     0     32
  [21] .dynamic          DYNAMIC         0000000000022a58  00021a58
       0000000000000200  0000000000000010  WA       6     0     8
  [22] .got              PROGBITS        0000000000022c58  00021c58
       0000000000000398  0000000000000008  WA       0     0     8
  [23] .data             PROGBITS        0000000000023000  00022000
       0000000000000268  0000000000000000  WA       0     0     32
  [24] .bss              NOBITS          0000000000023280  00022268
       00000000000012d8  0000000000000000  WA       0     0     32
```

Figure 3: Example Section header (truncated) of `ls` ELF binary.

```
key:
    [...]   A complete page
    T   Text
    D   Data
    P   Padding

Page Nr
#1      [TTTTTTTTTTTTTTTT]              <- Part of the text segment
#2      [TTTTTTTTTTTTTTTT]              <- Part of the text segment
#3      [TTTTTTTTTTTTPPPP]              <- Part of the text segment
#4      [PPPPDDDDDDDDDDDD]              <- Part of the data segment
#5      [DDDDDDDDDDDDDDDD]              <- Part of the data segment
#6      [DDDDDDDDDDDDPPPP]              <- Part of the data segment

pages 1, 2, 3 constitute the text segment
pages 4, 5, 6 constitute the data segment
```

Figure 4: Example process image [12].

An high level point of view of how an ELF binary is structured is reported in Figure 5; this image points to the fact that sections and segments often overlap, so it can be said that segments represent the binary at a structural, coarse-grained level, while sections are more fine-grained and focus on the semantic of the contained bytes. Sections not associated with a segment typically contain such thing as debugging information, symbol tables etc.

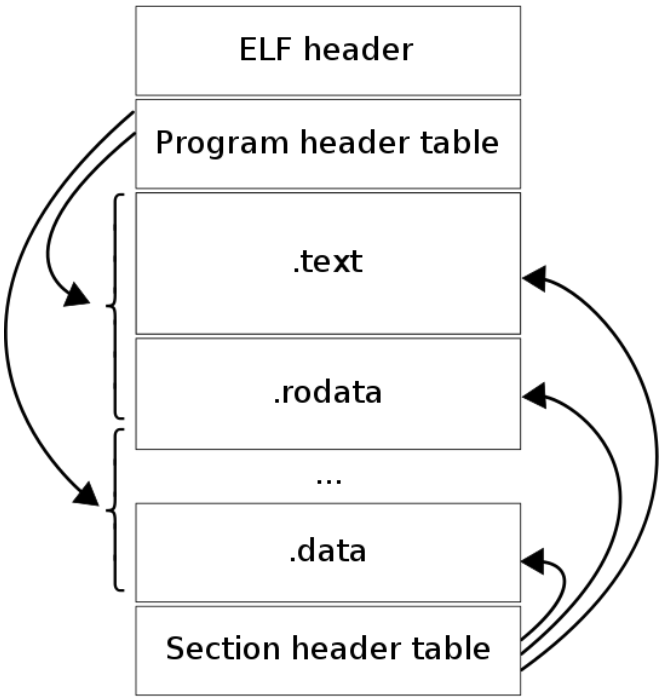

Figure 5: ELF general structure [13].

Malformed ELF binary A malformed ELF binary [7][4][5][6] is a file that contains unexpected values in its header; these ELF files do not run on all Linux distributions and they could also break across different versions of the same distribution. A high number of software that work with ELF binaries like debuggers (`GNU Debugger`, also known as `gdb`) and analysis tools (`readelf`, tool that displays information about ELF format object files), can't handle this type of files and break during execution. An example of how `readelf` behaves given a malformed ELF is shown in Figure 6.

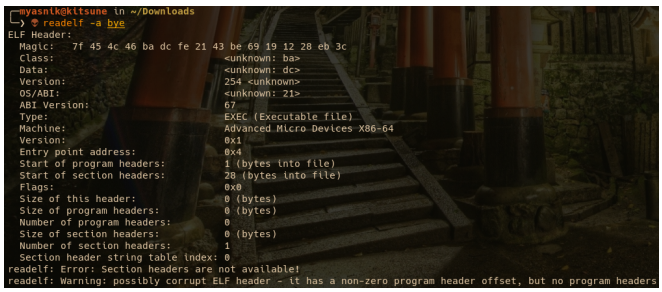

Figure 6: How `readelf` parses `bye.asm` [7], a malformed (yet executable) ELF file.

Malformed ELF binaries are usually used to contain viral code and are well known for their tiny dimensions; they can appear tough to understand and analyze as they usually break code debuggers.

Parasite viruses To insert parasite code means that the process image must load it so that the original code and data is still intact.

There are plenty of infection techniques that could be employed to achieve this; in general the aim is to manipulate the text section and the program header of the infected binary. The pivotal techniques used by parasitic viruses are briefly explained in the next bullet point list [15].

- **.note Section Overwriting**: the `.note` section is a standard section of the ELF format. It is primarily used by compilers and other tools to give information about the object. The goal is to overwrite an existing `.note` section as it is not essential for the file with a loadable and executable section [8].
- **Section Adding**: in order to have an unlimited payload size available, a new section can be created and executed.
- **Segment Padding**: segment addresses are subject to padding (Figure 4); the viral code can so be injected in padding areas.
- **Section Padding**: same as segment padding but applied to sections.
- **Code Cave**: a code cave is an area of bytes in the `.data` segment of a binary that contains a null byte pattern (`x00`) greater than two bytes; those code caves can be chained to contain a split payload.

The common point between these categories is that the binary entry point, the address at which executable code is placed, is always modified to point to the new injected code.

2 Proposed Work

Infection detection This part of the process could be achieved instructing a machine learning autoencoder model [11]. The aim of an autoencoder is to learn a representation (encoding) for a set of data, typically for dimensionality reduction, by training the network to ignore signal "noise" (Figure 7). Along with the reduction side, a reconstructing side is learned, where the autoencoder tries to generate from the reduced encoding a representation as close as possible to its original input. Feeding a large data set of unharmed ELF binaries to the autoencoder this will generate a neural layer containing sort of a summary of the input data, this is then used in the working phase to detect anomalies in the analyzed binaries.

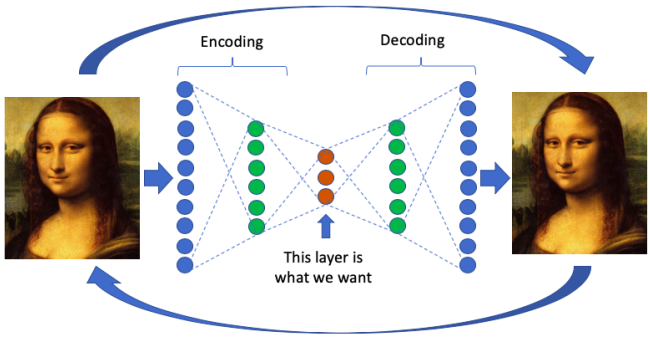

Figure 7: Autoencoder model workflow [10].

Using entire binaries for the training phase could lead to a noisy model, as ELF binaries can vary much between each other; thus, the data set must be modified in order to distinguish a patched binary form a legitimate one. The proposed solution is to only consider the ELF header, the program header, and the section header for each analyzed ELF binary instead of the whole binary; these portions are the ones that are most likely modified by parasitic viruses (other than the code section added/modified). This can be proved comparing the infected and legitimate version of a simple ELF file: in Figures 8 and 9 the *Midrashim* [1][2] virus has been used; this implements the PT_NOTE to PT_LOAD infection [8] which is part of the **.note Section Overwriting** category.

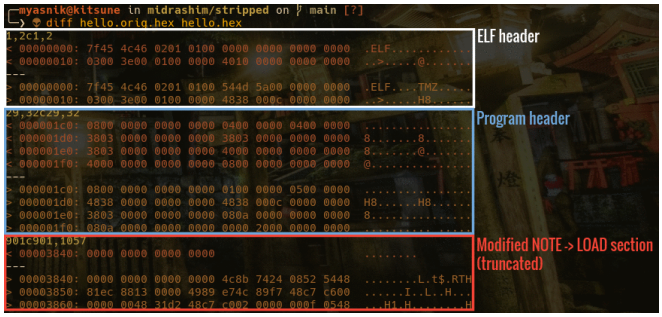

Figure 8: Example of `diff` between a simple *Hello World* ELF binary (legitimate vs infected by *Midrashim* virus [1][2]) using the hex dumper `xxd`.

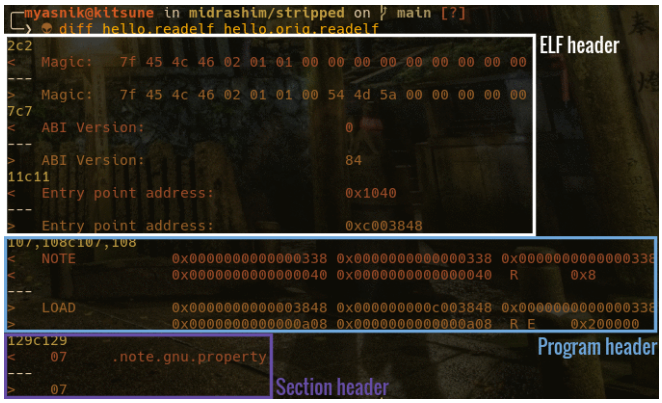

Figure 9: Example of `diff` between a simple *Hello World* ELF binary (legitimate vs infected by *Midrashim* virus [1][2]) using `readelf`.

Infected portions highlighting Merging the just trained machine learning model with two well known GNU utilities, `readelf` and the hex dumper `xxd`, will lead to the development of a wrapper software that is able to identify patched portions of binaries and highlight them to malware analysts. How this software is supposed to work is synthesized in Figures 10 and 11.

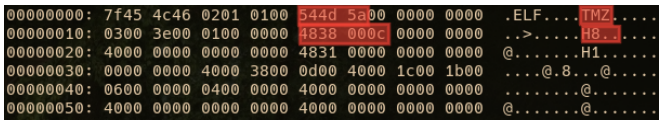

Figure 10: How the model should wrap `xxd` to highlight the suspected infection bytes.

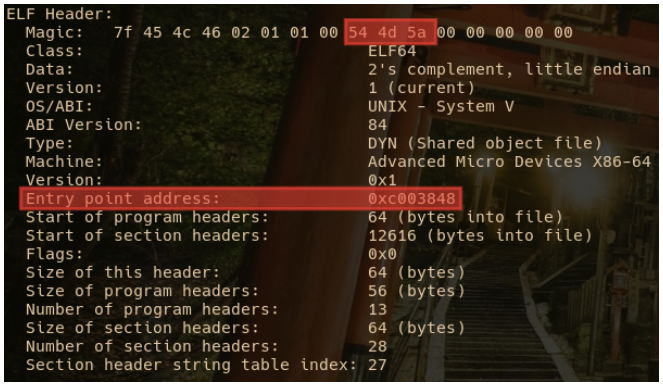

Figure 11: How the model should wrap `readelf` to highlight the suspected infection bytes.

ELF binary restoration The last step consists of restoring the ELF file to its unharmed state parsing and analyzing the highlighted infected portions. To accomplish this, three options could be adopted. The first idea is to try to achieve data recovery through machine learning, this will require the training of a new model; "Development of machine learning solutions and their applications on data recovery related problems requires collection of statistical data from raw data samples as well as from previously sorted/resolved cases." [16] As the just cited paper points to, this method will require a lot of work for constructing a large data set containing already restored binaries. The second idea is based on machine learning as well, but provides a slightly different result; a new model could be trained to categorize infection types and suggest action to restore infected binaries. This solution isn't comparable to the former one because it's just a sheer categorization model, but together with the detection model it could be useful to security analysts. The last method adoptable involves developing traditional software to restore binaries; this is possible as it has already been achieved for some type of infections [3] but the feasibility for all types of infections is not foregone.

One could also think about mixing the just explained approaches; this should be definitely the best idea but also the most time-consuming and complex one. Merging different paradigms can be very challenging but is seems a widely used procedure nowadays.

Building the dataset To obtain a large amount of benign ELF binaries to build the dataset the easiest way is collecting files found in `/bin`-like folders in Linux based systems. Using binaries of most famous Linux distributions will grant a great variety with very little effort. To achieve this `Docker` could be used: the idea is pulling official images of famous software and operative systems and extracting from the generated `Docker` container their binaries. To further enrich the dataset, as the ELF file format is cross-platform and cross-architecture, adopting binaries of different architectures can be considered.

About the pre-processing phase the plan is, for each binary, extracting the ELF, program and section headers as `readelf` plain text output (`readelf -e <binary-file>`), and also the hex dump of these using `xxd` or `python`. This will be useful for tying together the *Infection detection* and the *Infected portions highlighting* phases.

The proposed dataset building method doesn't apply to the *ELF binary restoration* phase as in this case the dataset would be a lot more complex; this is one of the reasons why the adoption of machine learning isn't really suitable for this part of the project.

3 Final thoughts

Related works Linux viruses world is incredibly vast but, except from the famous classification and analysis study done in [12], academic research on this matter still has to move forward. The proposed work could spur researchers to deepen this subject and to study new methods to counteract viral infections in the Linux ecosystem.

Said that, analogies could be found in [17] with the Windows' PE (Portable Executable) format features. In this paper are analyzed machine learning techniques to perform malware detection; one of these methods consists of helping the model to spot potential threats instructing it on how a PE is structured, more or less what is explained in this proposal but studying a different type of executable file. In the above cited paper are collected other interesting methods for malware detection but their usage in the proposed case wouldn't be so useful. This mainly because the focus of the research project isn't spotting malicious code but the general detection of patched binaries. Whether the patched binary has been injected with a simple `printf("Hello World!")` or with malicious code, the trained model will attempt to detect and restore this patch regardless. In the proposal the infection detection is stressed more than the general patching of an ELF simply for practical and real world reasons. So, adopting techniques cited in [17] like *String features* (analysis of strings collocated in the executable file), *Function based features* (extract functions and use them to produce various attributes representing the file), in general Dynamic analysis etc. wouldn't be a profitable choice as these techniques would be useless in this research context.

Impact Comparing the proposed work paradigm with the current state of the art of malware detection ([14][17][18]) some considerations arises: the depicted approach aims are really different from what the traditional malware detection focuses on. It's hard to talk about pros and cons, they're more likely conceptual differences. In malware detection, as the name points to, the end is perceiving if a file contains viral code, while in this research the point is simply detecting and restoring corrupted binaries. The concept of reparation in malware detection doesn't exist; that's the strong feature of this research.

The restoration model proposed could certainly be useful in industrial fields where a viral attack took place to restore the infected binaries without the need of recompiling, rebuilding or reinstalling software. Nevertheless, this can be an edge case, but in some environments this work could be very effective and useful; for instance let's think about Internet of Things devices: having a way to detect viral infections and restoring the device to a unharmed state without the need of a full reset could save a lot of time and work.

Integrating this new concept to modern malware de-

tection approaches could further enrich the work provided by those services.

Expected results The final result of this work will be a complex tool-set for ELF binaries analysis and restoration; machine learning models can be theoretically useful in the detection and highlighting phase, while in the restoration phase this approach could be cumbersome. The best scenario will probably be implementing restoration as a modular software that could be expanded adding rules (for example as `yaml` files) for each disinfection technique. This path is still very valuable even without the adoption of a neural network because an ELF restoration engine doesn't exist yet.

However, the alternative of trying to adopt a machine learning approach in the restoration phase shouldn't be discarded as it could lead to interesting and unexplored paths.

References

[1] guitmz. *Linux.Midrashim: Assembly x64 ELF virus*. Jan. 2021. URL: https://www.guitmz.com/linux-midrashim-elf-virus/.

[2] guitmz. *TMP.out - Linux.Midrashim.asm*. Apr. 2021. URL: https://tmpout.sh/1/Linux.Midrashim.asm.

[3] manizzle. *TMP.out - PT_NOTE Disinfector*. Apr. 2021. URL: https://tmpout.sh/1/4.html.

[4] netspooky. *Elf Binary Mangling - Part 1*. Aug. 2018. URL: https://n0.lol/ebm/1.html.

[5] netspooky. *Elf Binary Mangling - Part 2*. Dec. 2018. URL: https://n0.lol/ebm/2.html.

[6] netspooky. *Elf Binary Mangling - Part 3*. Dec. 2018. URL: https://n0.lol/ebm/3.html.

[7] netspooky. *GitHub - bye.asm*. URL: https://gist.github.com/netspooky/dd750e7ced85fb1861780a90be71053d.

[8] sblip. *TMP.out - Implementing the PT_NOTE Infection Method in x64 Assembly*. Apr. 2021. URL: https://tmpout.sh/1/2.html.

[9] xcellerator. *TMP.out - Dead Bytes*. Apr. 2021. URL: https://tmpout.sh/1/1.html.

[10] *Anomaly Detection with Autoencoders Made Easy - Image*. URL: https://towardsdatascience.com/anomaly-detection-with-autoencoder-b4cdce4866a6.

[11] Pierre Baldi. "Autoencoders, Unsupervised Learning and Deep Architectures". In: *Proceedings of the 2011 International Conference on Unsupervised and Transfer Learning Workshop - Volume 27*. UTLW'11. Washington, USA: JMLR.org, 2011, pp. 37–50.

[12] Silvio Cesare. *UNIX VIRUSES*. 1998. URL: https://ivanlef0u.fr/repo/madchat/vxdevl/vdat/tuunix01.htm.

[13] *ELF, Wikipedia - Image*. URL: https://en.wikipedia.org/wiki/Executable_and_Linkable_Format.

[14] Dragoş Gavriluţ et al. "Malware detection using machine learning". In: *2009 International Multiconference on Computer Science and Information Technology*. 2009, pp. 735–741. DOI: 10.1109/IMCSIT.2009.5352759.

[15] Pierre Graux, Aymeric Mouillard, and Mounir Saoud. "Backdooring ELF Using Unused Code". In: (2016).

[16] David Edwards Igor Sestanj. "Advanced Data Recovery Techniques: Using machine learning in Data Recovery". In: (2019).

[17] Asaf Shabtai et al. "Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey". In: *Information Security Technical Report* 14.1 (2009). Malware, pp. 16–29. ISSN: 1363-4127. DOI: https://doi.org/10.1016/j.istr.2009.03.003. URL: https://www.sciencedirect.com/science/article/pii/S1363412709000041.

[18] Alireza Souri and Rahil Hosseini. "A State-of-the-Art Survey of Malware Detection Approaches Using Data Mining Techniques". In: *Hum.-Centric Comput. Inf. Sci.* 8.1 (Dec. 2018). ISSN: 2192-1962. DOI: 10.1186/s13673-018-0125-x. URL: https://doi.org/10.1186/s13673-018-0125-x.

[19] *TMP.out e-zine*. URL: https://tmpout.sh/1/.