Lab-Manual

Digital Forensics

CY-334L

Prepared By: Ms.Memoona Sadaf

Instructor: Ms.Memoona Sadaf

Lab/Teaching Assistant: Muhammad Ahmad Ali Qureshi

Air University Islamabad

AIR UNIVERSITY

Department of Cyber Security

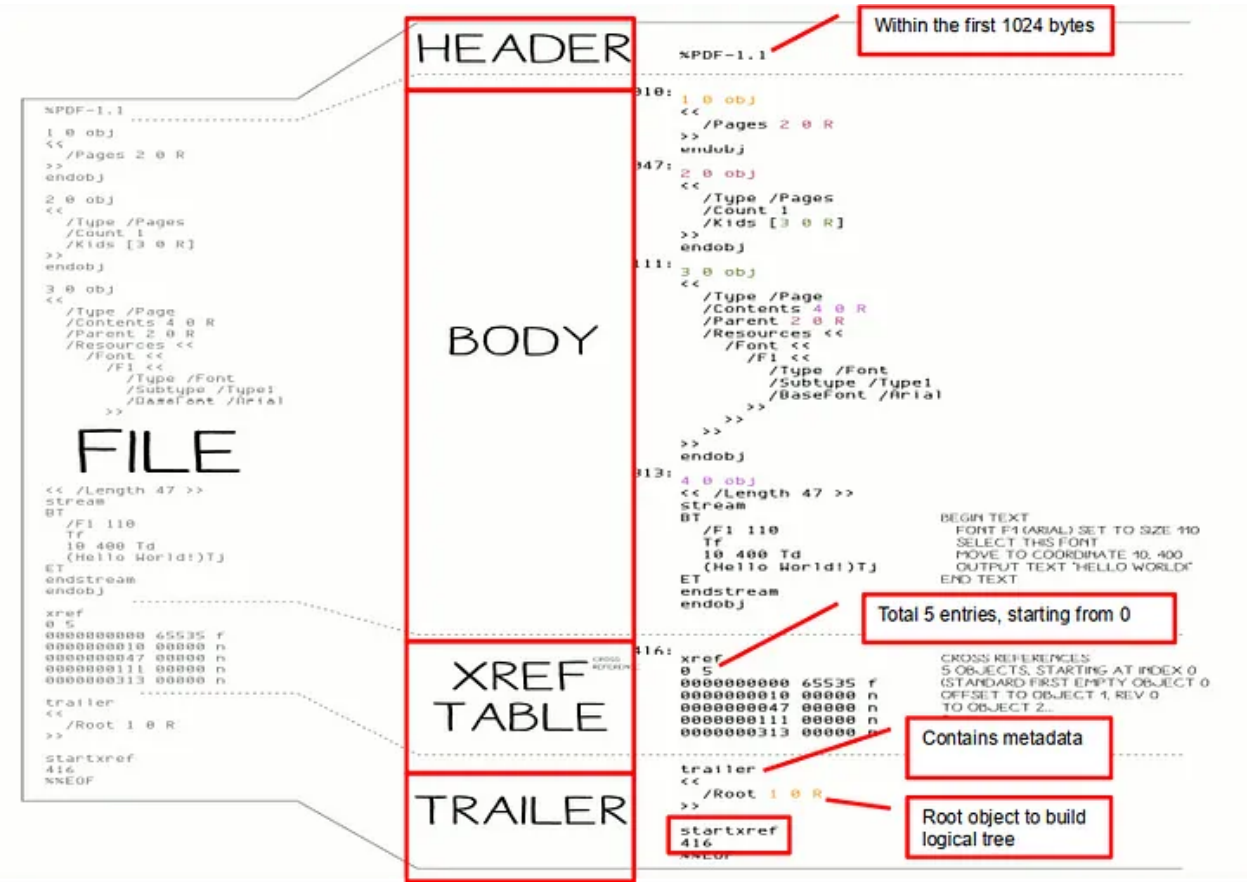
Lab Schedule

Week	Lab Topics
Week 6	**Common File Type Analysis Lab**

PDF Structure

In order to perform PDF forensics, it is essential to understand the structure of PDF. Luckily, it is fairly simple. I find the figure below from *zbetcheckin* quite representative for all what we need to know. PDF contains 4 parts:

- **Header:** starts with %PDF (e.g. %PDF-1.1 for PDF version 1.1) within the first 1024 bytes.
- **Boby:** contains the main content of a PDF which is actually just a list of objects (PDF object will be covered in more details in the **PDF syntax** part).
- **Xref (Cross Reference) table**: contain offset addresses of objects in the Body and their status ("n" stands for not free or in use and "f" stands for free). In the figure below, the first row "0 5" indicates that there are 5 objects in the PDF file and starting from object 0. So there are 5 more rows contain offset addresses and statuses for object 0, 1, 2, 3 and 4 respectively.
- **Trailer:** contains three important information: (1) the root object which indicates the starting of the PDF logical view. For further information, I suggest to check out this article from Didier Stevens. (2) the offset address of the Xref table so that PDF viewer can refer to objects. (3) other metadata such as author and created date.



PDF Syntax

Adobe provides a portable reference for PDF which is about 1,000 pages, so creating a PDF is fairly complex. Fortunately, we don't need to know all of those details to perform malware analysis for PDF. The figure below illustrates most (if not all :)) of what we need to know.

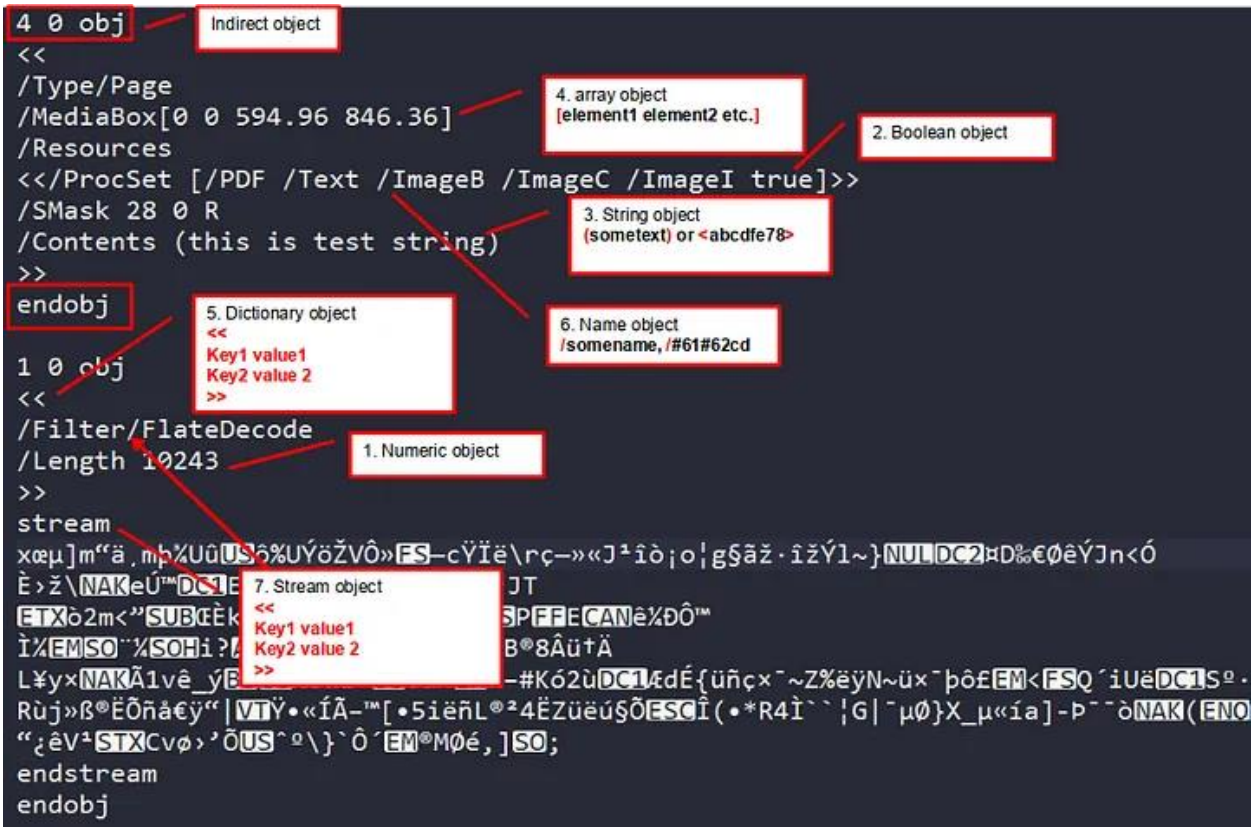
PDF content is actually just a list of objects which are linked together to build a logical tree. An **Object** starts with ***obj_num obj_rev obj*** (e.g. 4 0 obj) and ends with ***endobj.*** *An object can be referenced by another object, so it can also be called an* ***indirect object.*** *Particularly, in the figure below, object 4 refers to object 28 (28 0 R).*

There are 7 basic types of objects (examples are shown in figure 2)**:**

1. **Numeric object**: is for numeric value such as length in this example.
2. **Boolean object:** is simply just true and false.
3. **String object:** is for string. However, it can support multiple forms : **(1) ASCII text** in parentheses () such as ***(****some text here****)*** and **(2) hex string** in angle

brackets **<>** such as *<abcdfe78>*.

4. **Array object**: is enclosed in a square bracket [].
5. **Dictionary object**: is a set of key-value pairs inside **<< …>>. As** shown in figure 2, there are 2 key-values pairs: key1 — /Filter with value1 — /FlateDecode and key2 – /Length with value2-10243.
6. **Name object:** is a unique symbol defined by a slash (/) and has no internal structure. Example are /somename or /#61#62cd (# to indicate 2-digit hexadecimal, so in this example, the name object is /abcd)
7. **Stream object**: is a sequence of bytes and can contain other objects/files/images etc. Furthermore, a stream can be encoded by one or multiple schemes. Figure 2 illustrates object 1 as a stream with the format:**<< *Dictionary***/Filter /FlateDecode (encode by FlateDecode)/Length 10243 (length of this stream is 10243)***>>stream***some bytes***endstream***



Some notable suspicious objects:

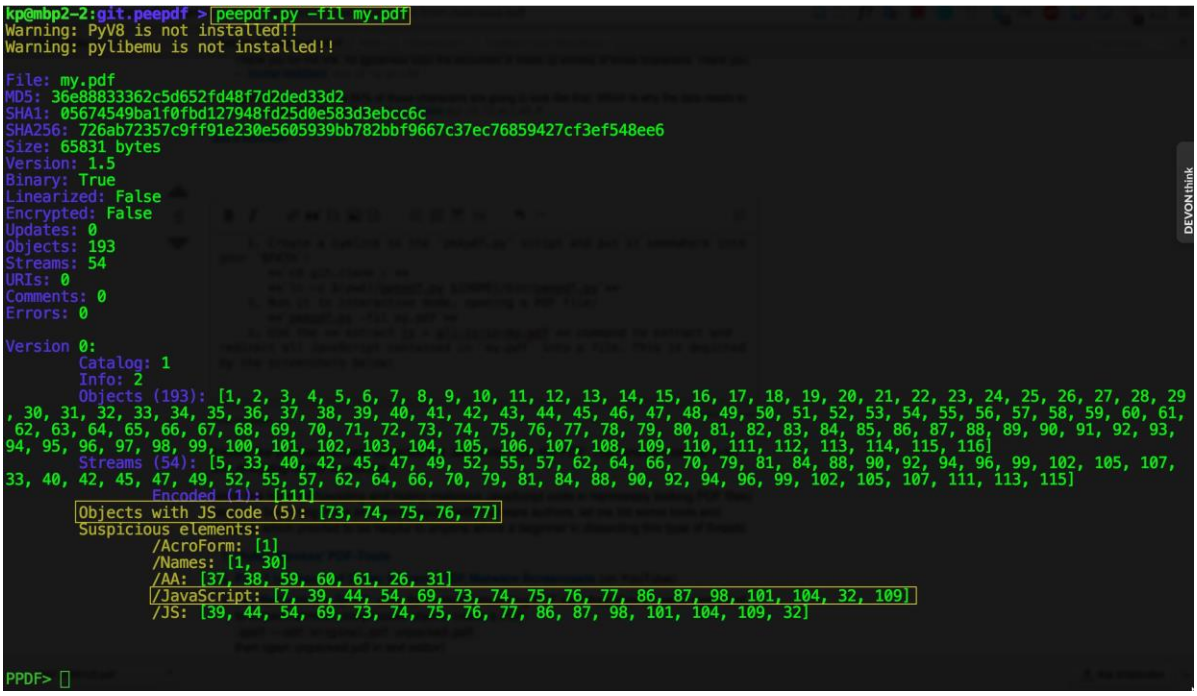
Below are some suspicious objects which are frequently strong indicators for further PDF analysis.

- **Embedded Javascript**: /JS, /JavaScript,/MacroForm and /XFA (the last two items can contain Javascript as well as other data used by Javascript in XML)
- **Embedded Flash**: /RichMedia (Flash supports action script)
- **Launching**: /AA, /Launch and /OpenAction (perform a defined action upon opening a PDF)
- **Internet access**: /URI and /SubmitForm
- **Embedded file**: /EmbeddedFiles
- **Others**: /Goto etc.

PeepPDF:

Installation and usage:

1. Clone the GitHub repository: `git clone https://github.com/jesparza/peepdf git.peepdf`.
2. Create a symlink to the `peepdf.py` script and put it somewhere into your `$PATH`: `cd git.clone ; ln -s $(pwd)/peepdf.py ${HOME}/bin/peepdf.py`
3. Run it in interactive mode, opening a PDF file: `peepdf.py -fil my.pdf`
4. Use the `extract js > all-js-in-my.pdf` command to extract and redirect all JavaScript contained in `my.pdf` into a file. This is depicted by the screenshots below:


kp@mbp2-2:git.peepdf > peepdf.py -fil my.pdf
Warning: PyV8 is not installed!!
Warning: pylibemu is not installed!!
File: my.pdf
MD5: 36e88833362c5d652fd48f7d2ded33d2
SHA1: 05674549ba1f0fbd127948fd25d0e583d3ebcc6c
SHA256: 726ab72357c9ff91e230e5605939bb782bbf9667c37ec76859427cf3ef548ee6
Size: 65831 bytes
Version: 1.5
Binary: True
Linearized: False
Encrypted: False
Updates: 0
Objects: 193
Streams: 54
URIs: 0
Comments: 0
Errors: 0
Version 0:
Catalog: 1
Info: 2
Objects (193): [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116]
Streams (54): [5, 33, 40, 42, 45, 47, 49, 52, 55, 57, 62, 64, 66, 70, 79, 81, 84, 88, 90, 92, 94, 96, 99, 102, 105, 107, 33, 40, 42, 45, 47, 49, 52, 55, 57, 62, 64, 66, 70, 79, 81, 84, 88, 90, 92, 94, 96, 99, 102, 105, 107, 111, 113, 115]
Encoded (1): [111]
Objects with JS code (5): [73, 74, 75, 76, 77]
Suspicious elements:
/AcroForm: [1]
/Names: [1, 30]
/AA: [37, 38, 59, 60, 61, 26, 31]
/JavaScript: [7, 39, 44, 54, 69, 73, 74, 75, 76, 77, 86, 87, 98, 101, 104, 32, 109]
/JS: [39, 44, 54, 69, 73, 74, 75, 76, 77, 86, 87, 98, 101, 104, 109, 32]
PPDF>
DEVONthink

```
PPDF> help

Documented commands (type help <topic>):
========================================
bytes           exit         js_jjdecode       open          search
changelog       extract      js_join           quit          set
create          filters      js_unescape       rawobject     show
decode          hash         js_vars           rawstream     stream
decrypt         help         log               references    tree
embed           info         malformed_output  replace       vtcheck
encode          js_analyse   metadata          reset         xor
encode_strings  js_beautify  modify            save          xor_search
encrypt         js_code      object            save_version
errors          js_eval      offsets           sctest

PPDF> help extract

Usage: extract uri|js [$version]

Extracts all the given type elements of the specified version after being decoded and decrypted (if necessary)

PPDF> extract js > all-js-in-my.pdf
PPDF> exit

Leaving the Peepdf interactive console...Bye! ;)

kp@mbp2-2:git.peepdf > cat all-js-in-my.pdf
function Motion(msg, n) {
    var f = new String(msg);
    return f.substr(n) + f.substr(0, n);
}
function checkField(aField) {
    if (aField.value == "") { // empty
        var msg = "No fields can be left empty!";
        app.alert(msg);
        return 0;
    }
}
function goNext(item, event, cName) {
    AFNumber_Keystroke(0, 0, 0, 0, "", true);
    if (event.rc && AFMergeChange(event).length == event.target.charLimit) item.getField(cName).setFocus();
}
var f = this.getField("message.1");

if (global.ttIsRunning == 1) {
    app.clearInterval(global.run);
    global.ttIsRunning = 0;
}
var f = this.getField("message.1");

var code = new String("this.getField('message.1').value =  Motion(this.getField('message.1').value,2);");

global.ttIsRunning = 1;

kp@mbp2-2:git.peepdf >
```


File carving

This section of the Forensics Lab introduces you to

file carving

File carving is an incredibly useful skill to have in the world of computer forensics. It basically means recovering files from a physical storage device after the files have been deleted, the device has been erased, or the device has been partially destroyed. At this point, the data on the device just looks like a sequence of "raw bytes" — meaning a sequence of bytes without any information as to where any file(s) begins or ends in this sequence.

- **How a storage device ("drive") is formatted**A storage device (hard drive, thumb drive, etc) is nothing more than a huge sequence of bytes. We can refer to a specific byte by its *offset*, i.e. it's distance from the initial byte. So the initial byte has offset 0, the next offset 1, and so forth. A formatted drive has
    - a *filesystem* — which means a record for each file of its name, the byte offset at which it begins, and the byte offset at which it ends; as well as records that indicate what directories (folders) there are, and which directory each file belongs in. Some of the bytes on the drive are used to represent the filesystem.
    - *files* — the filesystem only contains information about the files, the actual files themselves are (usually) nothing more than a chunk of consecutive bytes on the drive.
    - free or *unallocated* space — these are the bytes on the drive that are not currently being used to store information either as part of the *filesystem* or as part of a *file*. When new files are created, bytes from the unallocated space are commandeered to store the new file.

The Windows "Recycle Bin"When a user "deletes" a file in Windows, the file is simply moved to a new directory, called the "Recycle Bin". When the user "empties" the Recycle Bin, the files in that directory are deleted as described to the left.

- **What does "delete" a file really mean?**When you tell the operating system to delete a file, all it really means is that
    - the filesystem structure's record of that file (its name, the byte offset it starts at and the byte offset it ends at) is destroyed, and
    - the bytes that constitute the file itself are simply reclassified as "unallocated space".

How to truly delete a file So what if you want to delete a file so that it truly cannot be recovered? To do that you have to not only "delete" the file in the sense of removing its record from the filesystem, you must also overwrite the the bytes of that file with zeros or with random values. There are utilities that will do that for you.It is possible that a sophisticated forensics analysis could analyze the magnetic patterns on a drive and determine not only the current bit pattern of a byte, but also previously stored bit patterns. Fear of this has led many people to consider a file to only truly be deleted if its bytes have been overwritten many times.

- **Recovering a file that has been "deleted"**Notice that after a file is "deleted", all of its bytes are still sitting there on the drive ... they are simply categorized as "unallocated", which means they are available for use in representing new files. So, a file that has been deleted is recoverable up until the time that its bytes are commandeered for other purposes. However, the file's name and the offsets at which it begins and ends are no longer available. So the trick is finding where the file begins and ends, and that is what "file carving" is all about.

With computers, "deleting" a file doesn't necessarily mean the data stored in the file (the bytes that comprise the file) are gone. It means that the filesystems record of the file's name and its connection to that area of the hard drive are gone. Those bytes become "unallocated space".

To carve a file from a block of bytes, you'll need to look for the header (and, depending on the file type, the footer) of the file. For example, the header (in hex) for a PNG file is `89 50 4e 47` and the footer is `49 45 4e 44 ae 42 60 82`. Below we have an example of a

chunk of unallocated space from a drive. Looking carefully, we spot a PNG header (starting at offset 10) and, following it, a PNG footer (ending at offset 42) and thus we can deduce that the from offset 10 to 42 is a PNG file.

Document Analysis and Steganography

Documents are a common way of sending or storing information like messages, reports, videos, or ideas. MS Office documents, Images, and audio files are some commonly used formats in our day-to-day lives. However, beyond what we see written in a word document or hear in an audio file, these documents can also contain hidden messages or malicious code that may execute when we open them. In this lab, we'll explore some techniques to analyze and examine these documents.

Microsoft Office Documents

There are two main file formats used by Microsoft Office documents:

- OLE (Object Linking and Embedding)
- OOXML (Office Open XML)

OLE

OLE (Object Linking and Embedding) was the file format used in early versions of Microsoft Office between 1997 and 2003. It defined a “file within a file” structure which allowed other files to be embedded into a file. For example, an Excel spreadsheet could be embedded within a Microsoft Word document.

It supported file extensions like `.rtf`, `.doc`, `.ppt`, and `.xls`, among others.

OOXML

OOXML (Office Open XML) is the current file format used in Microsoft Office, which relies on an XML-based format for office documents.

The extensions for these documents include `.docx`, `.pptx`, and `.xlsx`, among others.

The OOXML format stores Office documents as ZIP containers. This means that the documents such as Word, Excel, and PowerPoint files, are actually just ZIP files. By renaming the extension from `.docx`, `.xlsx`, or `.pptx` to `.zip`, you can extract the

contents of the archive and view the individual XML files. This is a useful feature for digital forensics, as it allows investigators to examine the contents of a document without modifying it.

Anatomy of an OOXML document

To take an example, let’s create a word document with the text `Hello World!` in it, transfer it over to our Kali machine and then unzip it.

```
$ unzip Doc1.docx
Archive:  Doc1.docx
  inflating: [Content_Types].xml
  inflating: _rels/.rels
  inflating: word/document.xml
  inflating: word/_rels/document.xml.rels
  inflating: word/theme/theme1.xml
  inflating: word/settings.xml
  inflating: word/styles.xml
  inflating: word/webSettings.xml
  inflating: word/fontTable.xml
  inflating: docProps/core.xml
  inflating: docProps/app.xml
```

Here’s how the file structure of a Word document looks like:

```
.
├── [Content_Types].xml
├── docProps
│   ├── app.xml
│   └── core.xml
├── _rels
│   └── .rels
└── word
    ├── document.xml
    ├── fontTable.xml
    ├── _rels
    │   └── document.xml.rels
    ├── settings.xml
    ├── styles.xml
    ├── theme
    │   └── theme1.xml
    └── webSettings.xml
```

[Content_Types].xml

This file contains information about the content types that are present in the document and their corresponding file extensions.

📂 docProps

This folder contains two files, `app.xml` and `core.xml`.

- `app.xml` – contains information about the application that was used to create the document.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Properties    xmlns="<http://schemas.openxmlformats.org/officeDocument/2006/extended-pr
operties>"    xmlns:vt="<http://schemas.openxmlformats.org/officeDocument/2006/docPropsV
Types>">
  <Template>Normal.dotm</Template>
  <TotalTime>0</TotalTime>
  <Pages>1</Pages>
  <Words>1</Words>
  <Characters>12</Characters>
  <Application>Microsoft Office Word</Application>
  <DocSecurity>0</DocSecurity>
  <Lines>1</Lines>
  <Paragraphs>1</Paragraphs>
  <ScaleCrop>false</ScaleCrop>
  <Company/>
  <LinksUpToDate>false</LinksUpToDate>
```

```
  <CharactersWithSpaces>12</CharactersWithSpaces>
  <SharedDoc>false</SharedDoc>
  <HyperlinksChanged>false</HyperlinksChanged>
  <AppVersion>16.0000</AppVersion>
</Properties>
```

- `core.xml` – contains metadata of the document, such as the author's name, creation date, and modification date.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<cp:coreProperties xmlns:cp="<http://schemas.openxmlformats.org/package/2006/metadat
a/core-properties>" xmlns:dc="<http://purl.org/dc/elements/1.1/>" xmlns:dcterms="<htt
p://purl.org/dc/terms/>" xmlns:dcmitype="<http://purl.org/dc/dcmitype/>" xmlns:xsi="<
http://www.w3.org/2001/XMLSchema-instance>">
  <dc:title/>
  <dc:subject/>
  <dc:creator>Saad Javed</dc:creator>
  <cp:keywords/>
  <dc:description/>
  <cp:lastModifiedBy>Saad Javed</cp:lastModifiedBy>
  <cp:revision>2</cp:revision>
  <dcterms:created xsi:type="dcterms:W3CDTF">2023-02-04T15:44:00Z</dcterms:created>
  <dcterms:modified xsi:type="dcterms:W3CDTF">2023-02-04T15:44:00Z</dcterms:modified>
</cp:coreProperties>
```

_rels

This folder contains one file named `.rels`.

- `.rels` – contains information about the relationships between the different parts of the document such as for `app.xml` and `core.xml`.

word

This folder contains the actual content of the document.

- `document.xml` — contains the actual text of the document.

```
<!-- SNIP -->
  <w:body>
    <w:p w14:paraId="68F74602" w14:textId="442B891B" w:rsidR="00D473D4" w:rsidRPr="00
B20485" w:rsidRDefault="00B20485">
      <w:pPr>
        <w:rPr>
          <w:lang w:val="en-US"/>
```

```
        </w:rPr>
      </w:pPr>
      <w:r>
        <w:rPr>
          <w:lang w:val="en-US"/>
        </w:rPr>
        <w:t>Hello World!</w:t>
      </w:r>
    </w:p>
    <w:sectPr w:rsidR="00D473D4" w:rsidRPr="00B20485">
      <w:pgSz w:w="11906" w:h="16838"/>
      <w:pgMar w:top="1440" w:right="1440" w:bottom="1440" w:left="1440" w:header="70
8" w:footer="708" w:gutter="0"/>
      <w:cols w:space="708"/>
      <w:docGrid w:linePitch="360"/>
    </w:sectPr>
  </w:body>
</w:document>
```

- `fontTable.xml` – contains information about the fonts used in the document.
- 📁 **_rels** — contains one file `ocument.xml.rels`.
  - `document.xml.rels` — contains information about the relationships between the different parts of the document, such as styles, themes, settings, as well as the URIs for external links.
- `settings.xml` — contains document settings and configuration information.
- `styles.xml` — contains information about the styles used in the document.
- 📁 **theme** – contains files about the theme used in the document.
  - `theme1.xml` — contains the actual theme content.
- `webSettings.xml` — contains information about the web-specific settings of the document, such as HTML frameset settings as well as how the document is handled when saved as HTML.

The information about any additional files that may be present in a document can be found on the link http://officeopenxml.com/anatomyofOOXML.php.

Macro-Enabled Documents

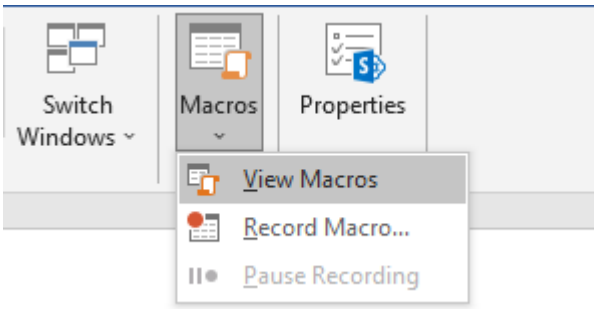
Macro-Enabled documents are documents that contain macros, which are sets of instructions that automate tasks. Macros can be written in Visual Basic for Applications

(VBA) and can be used to perform a wide range of tasks, such as formatting text, performing calculations, and automating complex processes. However, attackers often utilize this functionality of Office documents with a phishing attack and embed malicious macros to perform malicious actions and install malware on the system.

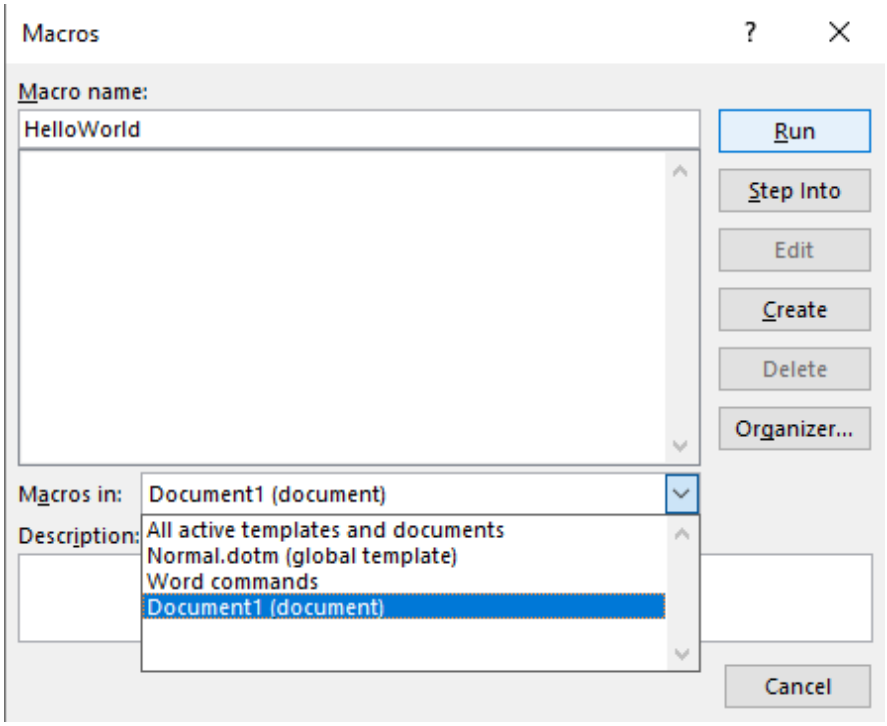
The extensions for these documents include `.docm`, `.pptm`, and `.xlsm`, among others.

To take an example, let's create an empty word document, and follow the steps below:

1. Click View → Macros → View Macros.



2. Type a name such as HelloWorld, select Document1 (current document) under Macros in, and click create.

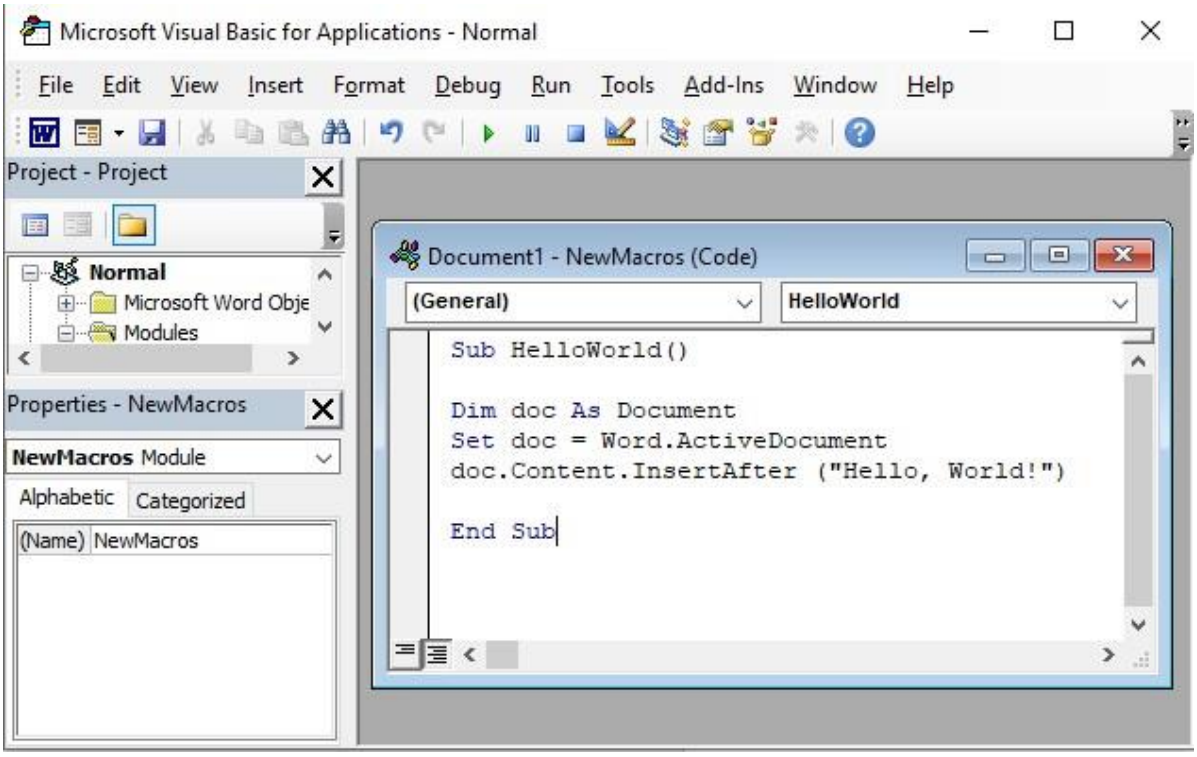


3. Paste the following code in the text box.

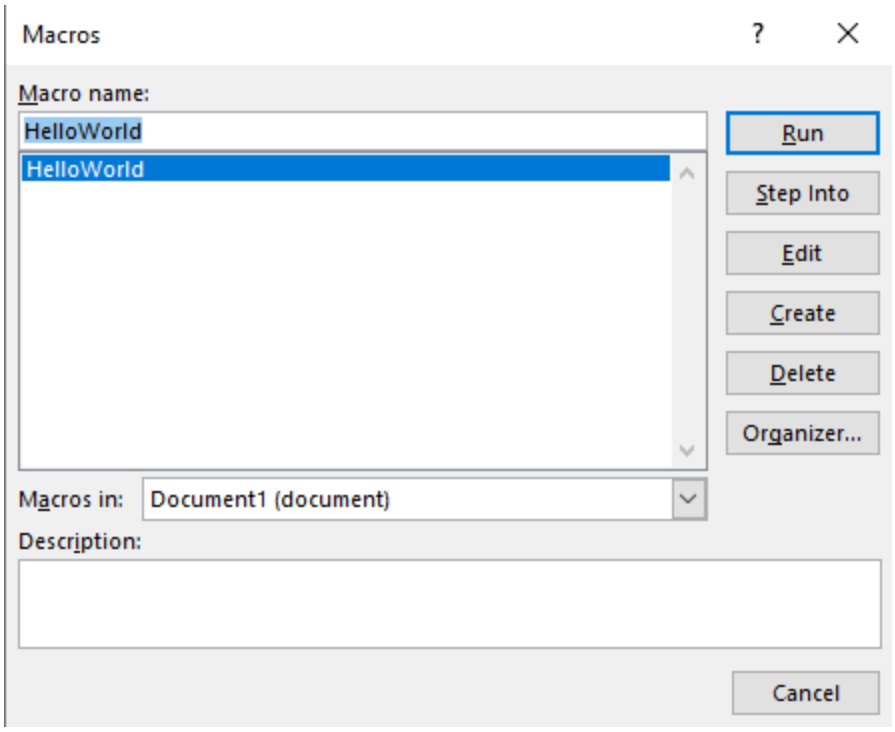
```
Sub HelloWorld()

Dim doc As Document
Set doc = Word.ActiveDocument
doc.Content.InsertAfter ("Hello, World!")

End Sub
```



4. Close the Microsoft Visual Basic for Application tab.
5. Repeat step 1, select the HelloWorld macro and click Run.



6. Observe that `Hello, World!` is now written in the document.
7. Save the document as `.docm`.

As per the anatomy of OOXML files, the macro is now stored inside `word/vbaProject.bin`, however, we won't be able to read it as it's in binary form. But, we can use a collection of tools called `oletools` to analyze and extract macros from OLE files such as Microsoft Office Documents.

To install `oletools`, use the command: `sudo -H pip install -U oletools`.

Let's use `oleid` to detect whether our document has any macros embedded in it.

```
$ oleid HelloWorld.docm
XLMMacroDeobfuscator: pywin32 is not installed (only is required if you want to use MS Exc
el)
oleid 0.60.1 - <http://decalage.info/oletools>
THIS IS WORK IN PROGRESS - Check updates regularly!
Please report any issue at <https://github.com/decalage2/oletools/issues>

Filename: HelloWorld.docm
WARNING  For now, VBA stomping cannot be detected for files in memory
--------------------+--------------------+----------+--------------------------
Indicator           |Value               |Risk      |Description
--------------------+--------------------+----------+--------------------------
File format         |MS Word 2007+ Macro-|info      |
```

```
                     |Enabled Document     |           |
                     |(.docm)              |           |
---------------------+---------------------+-----------+-----------------------------
Container format     |OpenXML              |info       |Container type
---------------------+---------------------+-----------+-----------------------------
Encrypted            |False                |none       |The file is not encrypted
---------------------+---------------------+-----------+-----------------------------
VBA Macros           |Yes                  |Medium     |This file contains VBA
                     |                     |           |macros. No suspicious
                     |                     |           |keyword was found. Use
                     |                     |           |olevba and mraptor for
                     |                     |           |more info.
---------------------+---------------------+-----------+-----------------------------
XLM Macros           |No                   |none       |This file does not contain
                     |                     |           |Excel 4/XLM macros.
---------------------+---------------------+-----------+-----------------------------
External             |0                    |none       |External relationships
Relationships        |                     |           |such as remote templates,
                     |                     |           |remote OLE objects, etc
---------------------+---------------------+-----------+-----------------------------
```

The result shows that the tool found VBA Macros and evaluated the risk as medium. We can now proceed with using `olevba` to extract the macros from the document.

```
$ olevba HelloWorld.docm
XLMMacroDeobfuscator: pywin32 is not installed (only is required if you want to use MS Exc
el)
olevba 0.60.1 on Python 3.10.8 - <http://decalage.info/python/oletools>
===============================================================================
FILE: HelloWorld.docm
Type: OpenXML
WARNING  For now, VBA stomping cannot be detected for files in memory
-------------------------------------------------------------------------------
VBA MACRO ThisDocument.cls
in file: word/vbaProject.bin - OLE stream: 'VBA/ThisDocument'
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
(empty macro)
-------------------------------------------------------------------------------
VBA MACRO NewMacros.bas
in file: word/vbaProject.bin - OLE stream: 'VBA/NewMacros'
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Sub HelloWorld()

Dim doc As Document
Set doc = Word.ActiveDocument
doc.Content.InsertAfter ("Hello, World!")
```

```
End Sub
No suspicious keyword or IOC found.
```

The output above shows that the extracted macro is exactly the same as we attached with the document.

Steganography

The word steganography comes from the Greek word Steganographia, made up of two words “steganos” meaning “covered or concealed” and “graphia” meaning “to write”. It involves hiding secrets in an otherwise seemingly innocent piece of information. An early example of steganography is the use of invisible ink made from lemon juice or vinegar to write on paper and then reveal the writing by heating the paper.

In today’s digital world, steganography is used to hide a message within another file, like an image or audio file, in such a way that it can not be seen or heard by anyone who doesn't know it's there. This technique can be used for both innocent purposes, like sending a secret message to a friend, or for malicious reasons, like concealing evidence or communicating without detection.

> Today, digital steganography is one of the important components in the toolboxes of spies and malicious hackers, as well as human rights activists and political dissidents.
>
> — Ben Dickson

The first recorded uses of steganography can be traced back to 440 BC in Greece, when Herodotus mentions two examples in his Histories. Histiaeus sent a message to his vassal, Aristagoras, by shaving the head of his most trusted servant, "marking" the message onto his scalp, then sending him on his way once his hair had regrown, with the instruction, "When thou art come to Miletus, bid Aristagoras shave thy head, and look thereon."

Source: Wikipedia

Some recent examples include:

- Malicious memes on Twitter – Link to blog.

- PDFs with malicious JavaScript code – Link to article.

Image Steganography

PNG and JPEG are two common image formats. They can also be used as a channel to hide messages inside them. Both these formats rely on different structures to construct an image, and therefore, different techniques are utilized to hide messages within them.

PNG

For PNG files, one of the commonly employed techniques to hide messages is by modifying the least significant bits (LSBs) of certain pixels such as those who have minimal impact on the quality of the image. The algorithm then knows which pixels to extract the embedded message from.

As an example, consider the below image of Van Gogh's Starry Night painting with a message already embedded in it.

One tool used for detecting PNG steganography is `zsteg`, which can be installed using the command `sudo gem install zsteg`.

Here's how we can use the tool to extract the hidden message:

```
$ zsteg starry_night.png
b1,rgb,lsb,xy       .. text: "148:The fishermen know that the sea is dangerous and the sto
rm terrible, but they have never found these dangers sufficient reason for remaining ashor
e."
b2,r,lsb,xy         .. file: OpenPGP Public Key
b4,g,lsb,xy         .. text: "*e~_^u|["
```

JPEG

For JPEG files, a commonly used steganography method involves finding pairs of positions inside an image such that exchanging their values has the effect of embedding the corresponding part of the secret message. If no such pairs are found, the pixels in the remaining positions are simply overwritten.

With `steghide` tool, we can hide and extract messages out of JPEG files.

> 💡 Another tool to add to your digital forensics toolkit is exiftool, which can be useful in extracting metadata from files such as an image or an audio file, which includes information such as the creation and modification dates, author, and location of where an image was captured (latitude and longitude).

Audio Steganography

Audio steganography is a fascinating area in the field of data security. It involves hiding data within audio files so that the presence of the data is concealed. This technique is used to protect information during transmission or to include watermarks for copyright protection.

Techniques in Audio Steganography

1. **Least Significant Bit (LSB) Encoding**:
   - This is one of the simplest methods where bits of the audio signal are replaced with bits of the secret message. It is relatively easy to implement but can be vulnerable to even slight modifications in the audio file.
2. **Phase Coding**:
   - This technique involves modifying the phase of the initial audio signal. The phase of the selected parts of the sound is adjusted to encode the message. This method is more robust against signal degradation than LSB.
3. **Spread Spectrum**:
   - In this method, the message is spread across the frequency spectrum of the audio signal. This makes the hidden data less susceptible to intentional or unintentional interference.
4. **Echo Hiding**:
   - This involves introducing an echo into the discrete signal. The information is embedded in the echo's delay and amplitude, making it imperceptible to the human ear but detectable with the right tools.

Tools for Analyzing Audio Steganography

1. **Audacity**:

    - A free, open-source, cross-platform audio software that can be used to analyze audio files for discrepancies that might suggest data has been embedded.
2. **DeepSound**:
    - A more specialized tool that provides audio steganography capabilities. It can encode and decode hidden files within audio files and supports various audio formats.
3. **Steghide**:
    - While primarily used for hiding data within image files, Steghide can also be applied to audio steganography. It supports compression, encryption, and embedding of checksums to verify the integrity of the data.
4. **Sonic Visualiser**:
    - This tool is used for viewing and analyzing the contents of audio files at a very detailed level. It's particularly useful for spotting the subtle changes that steganography might introduce to an audio file.

Applications of Audio Steganography

- **Confidential communication**: It allows sensitive information to be transmitted without drawing attention.
- **Watermarking**: Audio files can be watermarked to trace the origin of unauthorized copies.
- **Covert operations**: Used by law enforcement for covert operations requiring high security.

Audio steganography is a powerful tool for secure communication and copyright protection, with various techniques and tools available for both embedding and detecting hidden data in audio streams.

Hex Editor: Overview and Uses

A hex editor is a software tool that allows users to view and edit the binary data of a file. In a hex editor, data is represented as hexadecimal values, which are readable to humans and reflect the raw contents of a file, including its header, data section, and end.

Understanding File Headers

The file header is a crucial part of any file, containing metadata about the file, such as type, size, and how the file should be handled by software. Modifying a file's header can change how the file is recognized and processed by applications.

Common File Headers and Hex Values

1. **JPEG (Image File)**:
    - Header: FF D8 FF
    - These bytes are the starting markers for a JPEG file.
2. **PNG (Image File)**:
    - Header: 89 50 4E 47 0D 0A 1A 0A
    - This sequence identifies the file as a PNG.
3. **GIF (Image File)**:
    - Header: 47 49 46 38 39 61 or 47 49 46 38 37 61
    - These bytes signify that the file is a GIF, either 89a or 87a version.
4. **PDF (Document File)**:

    - Header: 25 50 44 46
    - This represents a PDF file starting with %PDF.
5. **ZIP (Compressed File)**:
    - Header: 50 4B 03 04
    - Zip files start with these bytes, indicating the file is a ZIP archive.

Modifying File Headers

Modifying a file header can serve several purposes:

- **Mimicking File Types**: By changing the header, you can make a file appear as a different type than it is, which can be used for security testing or penetration testing exercises.
- **Data Recovery**: If a file header is corrupted, editing the header to its correct form can make the file accessible again.
- **Bypassing Filters**: In security contexts, changing file headers can help test the robustness of file upload filters or email gateways that rely on header information to block unwanted file types.

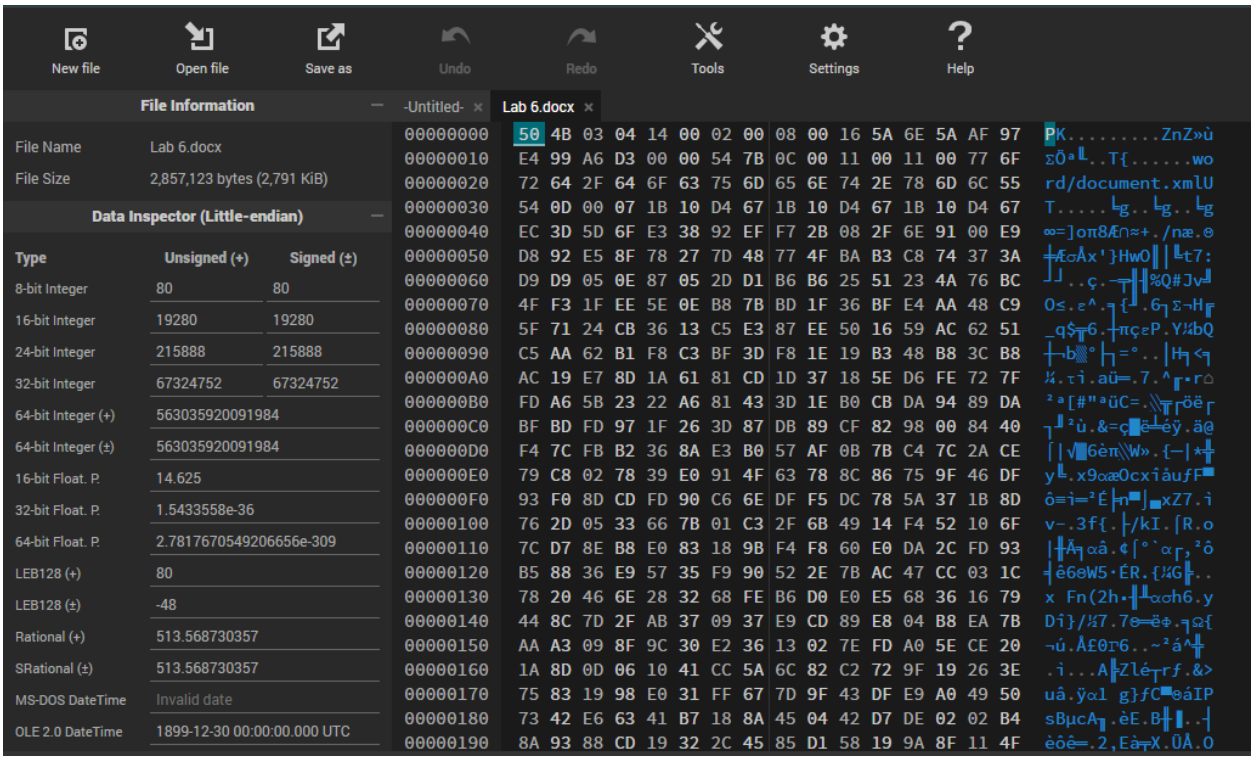
Tools for Hex Editing

1. **HxD**:
    - A freeware hex editor that allows users to edit hexadecimal data and binary files. It's widely used for data recovery, analysis, and forensics.
2. **Hex Fiend**:
    - A fast and clever open-source hex editor for macOS. It scales well for very large files and can handle binary diffs.
3. **010 Editor**:
    - A professional-grade hex editor with advanced binary editing and scripting capabilities, often used in professional environments for detailed analysis.

Security and Hex Editing

Modifying binary data can have security implications. It's essential for security professionals to understand hex editing to recognize tampered files or perform forensic analysis. Ethical use guidelines should always be followed when handling data, especially in sensitive or regulated environments.

Overall, hex editors provide a powerful interface for analyzing and manipulating files at their most fundamental level. Whether for troubleshooting, forensics, or security testing, understanding how to use a hex editor is a valuable skill in many technical fields.



Online Hex-editor Tool https://hexed.it/