SkyPenguinLabs

How understanding & quickly identifying file types assist you during RE tasks

SPL RExC

Powered by SkyPenguinLabs

SPL-RExC Introduction

> Knowing File Types in RE Helps You, Trust....

Welcome to one of the first free lessons apart from the SkyPenguinLabs course page! This is a demonstrative course meant to both introduce and attract new learners and even professionals by giving you a good taste of what our content looks like, to see if it's worth it.

After all, the aim is to be better than companies who sell basic courses, such as Open Source Intelligence Fundamentals for more than $30 :D

In this lesson, we will be discussing the importance of being able to identify file extensions during a reverse engineering operation, and how to figure it out if you do not know where to go. It seems silly, as it's such a basic concept, however, the most fundamental concepts are often missed by people.

You may even be asking right now - how can you reverse engineer something without even knowing the file format of the app you are reversing?

Truly, you can, does not make things easy to be honest, but not every single file you are going to be looking at requires file analysis down to the file structure.

Hope you enjoy the content!s

Powered by SkyPenguinLabs

Table of Contents

> Lesson section outline

For this lesson, we are trying to stay minimal given this is a free resource, however, each section tries to cover as much as it can for something beginner focused.

- **Section 0x00 | Prerequisites** - *(Each lesson/course will have this, even paid, includes what you will need before reading the course.)*
- **Section 0x01 | File Formats & Reverse Engineering** - *(a section that covers some things that beginners may not be prepared for when reversing apps, such as having to repair files, or repair headers for a resource loaded by an app. This also discovers the importance of being certain)*
- **Section 0x02 | How to Recognize Them** - *(a section which covers how to recognize and fix file headers and types)*
- **Section 0x03 | Next Steps Forward** - *(Next steps forward when it comes to applying this knowledge elsewhere, automasting it with specific tools, and more)*
- **Section 0x04 | Conclusion** - *(Ending & Concluding the first free lesson of SkyPenguinLabs!)*

Powered by SkyPenguinLabs

Section 0x00

> Prerequisites

Understanding this lesson requires a bare minimum technical background in just IT. If you do not know what reverse engineering is, then that might be good to understand. However, for the mass majority of people here for reversing tips as beginners, this should be generally easy to comprehend.

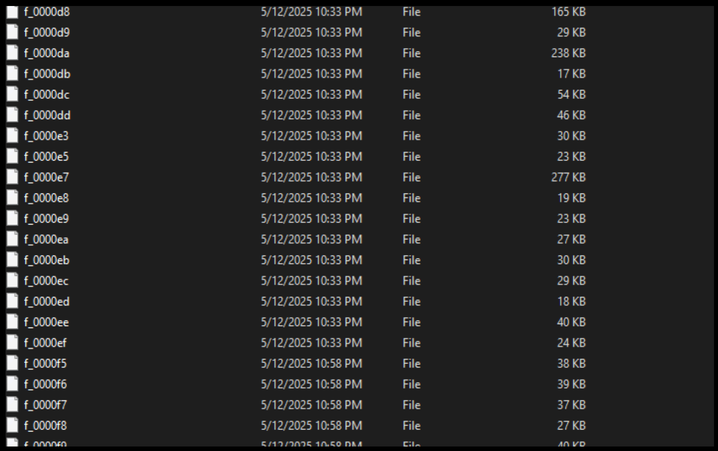
Since this is not covering any technical or practical content, we will not need any fancy environments. If you want to follow along, there will be minimal demonstrations using HxD to understand file metadata.

Powered by SkyPenguinLabs

Section 0x01

> File Formats and Reverse Engineering

Have you ever been reverse engineering an application, and come across funky files that never rendered or seemed like bunches of text and content that you could not make much of? Like the file below?



```
f_0000d8    5/12/2025 10:33 PM    File    165 KB
f_0000d9    5/12/2025 10:33 PM    File     29 KB
f_0000da    5/12/2025 10:33 PM    File    238 KB
f_0000db    5/12/2025 10:33 PM    File     17 KB
f_0000dc    5/12/2025 10:33 PM    File     54 KB
f_0000dd    5/12/2025 10:33 PM    File     46 KB
f_0000e3    5/12/2025 10:33 PM    File     30 KB
f_0000e5    5/12/2025 10:33 PM    File     23 KB
f_0000e7    5/12/2025 10:33 PM    File    277 KB
f_0000e8    5/12/2025 10:33 PM    File     19 KB
f_0000e9    5/12/2025 10:33 PM    File     23 KB
f_0000ea    5/12/2025 10:33 PM    File     27 KB
f_0000eb    5/12/2025 10:33 PM    File     30 KB
f_0000ec    5/12/2025 10:33 PM    File     29 KB
f_0000ed    5/12/2025 10:33 PM    File     18 KB
f_0000ee    5/12/2025 10:33 PM    File     40 KB
f_0000ef    5/12/2025 10:33 PM    File     24 KB
f_0000f5    5/12/2025 10:58 PM    File     38 KB
f_0000f6    5/12/2025 10:58 PM    File     39 KB
f_0000f7    5/12/2025 10:58 PM    File     37 KB
f_0000f8    5/12/2025 10:58 PM    File     27 KB
f_0000f9    5/12/2025 10:58 PM    File     40 KB
```

Sometimes, developers can either push breaking changes to a feature which results in resources on an app client being downloaded in chunks and corrupted, or times like above where app's cache data but do not properly categorize the header. Leaving us to do the work ourselves!

It may be a rare case scenario, but when it happens, you best know how exactly to identify file formats, broken formats and structures, as well as other unique characteristics of files which can make your reversing bumps smoother.

> *Note: While we will not cover every single portion of the venture in this lesson, I will be teaching you the basics of applying specific methodologies to repair broken files, and understand resource formats.*

Before we dive straight into it, I think it is also important that I cover the exact type of scenarios and files that may be related to such scenarios so we have a good baseline to represent our skill.

Sections start on the next page!

Scenario 1 - Application Cache's Broken & Non-Broken Files

In one scenario that happened to me personally recently, I was dealing with an ElectronJS application which had LOADS of assets. And when I mean loads I do mean LOADS! This application would stuff all of the images (*over 60-100 images*), all of its font files, all of its configuration files, and etc in the same cache.

In this scenario, when the files were cached to the application, they had no names, and each of the files upon first look, did not seem like they had much data in them. Until…doing a simple header check, changing the file format and being able to see the data that was in there.

Scenario 2 - Application Downloads Broken Module

Some hobbyist reverse engineers will often take legacy versions of software for the pure purpose of breaking their auth systems because previous versions were weaker. In some scenarios, legacy versions of software may download modules that do not work anymore, or are not fully kept up leading to issues with compatibility when picking the exact legacy version, this also depends on the software and for this scenario is entirely hypothetical.

In such cases, because the module is broken, if possible, static or dynamic patching may be used to repair very specific portions of the library, such as the header, or a specific section if it is broken, or corrupted.

Fixing files is very trivial, and sometimes may not be possible because most modern applications that care have enough integrity to detect modifications or broken changes and force you to update.

With those two scenarios in mind, we can head into our next section which tackles both scenarios with different methods.

Powered by SkyPenguinLabs

Section 0x02

> How to Recognize and Repair Broken File Formats

There are so many different methods of working to identify file formats whos true identity is hidden. Sometimes, however, it can be either extremely easy, or extremely hard depending on if the file was broken unintentionally versus intentionally. Intentionally being somebody who wants to hide data inside of a resource, and purposefully mask its identity to bypass restrictions on resource loading (*in a hypothetical scenario*)

We will only be using two basic techniques for this lesson to keep it short, however, the general list goes to be the following:

- Signature Analysis - Something most people know about and every system uses today is signature analysis. When analyzing a file's metadata and structure, there is usually a header, which contains a set amount of bytes, which converts to a unique symbol relevant to that file format. For example, JPEG has \xFF\xD8\xFF in it

- Entropy Analysis - Something that is really unique is using entropy analysis as a form of file identification. Ideally this is only going to be used in very large massive and niche scenarios, but I wanted to bring this up because more so for identifying the encryption, packing and compression behind a file, which alludes to resources such as images, being tested against a form of authenticity if you will.

- Structure Analysis - Some files, such as Atom Shell Archive (*ASAR*) files, APK (*Android Package Kit*) files, and BPLIST (*Binary Property List*) have unified structures which can also share their own unique signatures. In unique scenarios, you can hash structures or patterns of structures to validate files as a form of second-step-validation.

Note that structure analysis is used after signature analysis. It's a way of validating that the file is actually what it says it is. This also includes reading further into metadata tags and comparing it with official verified data.

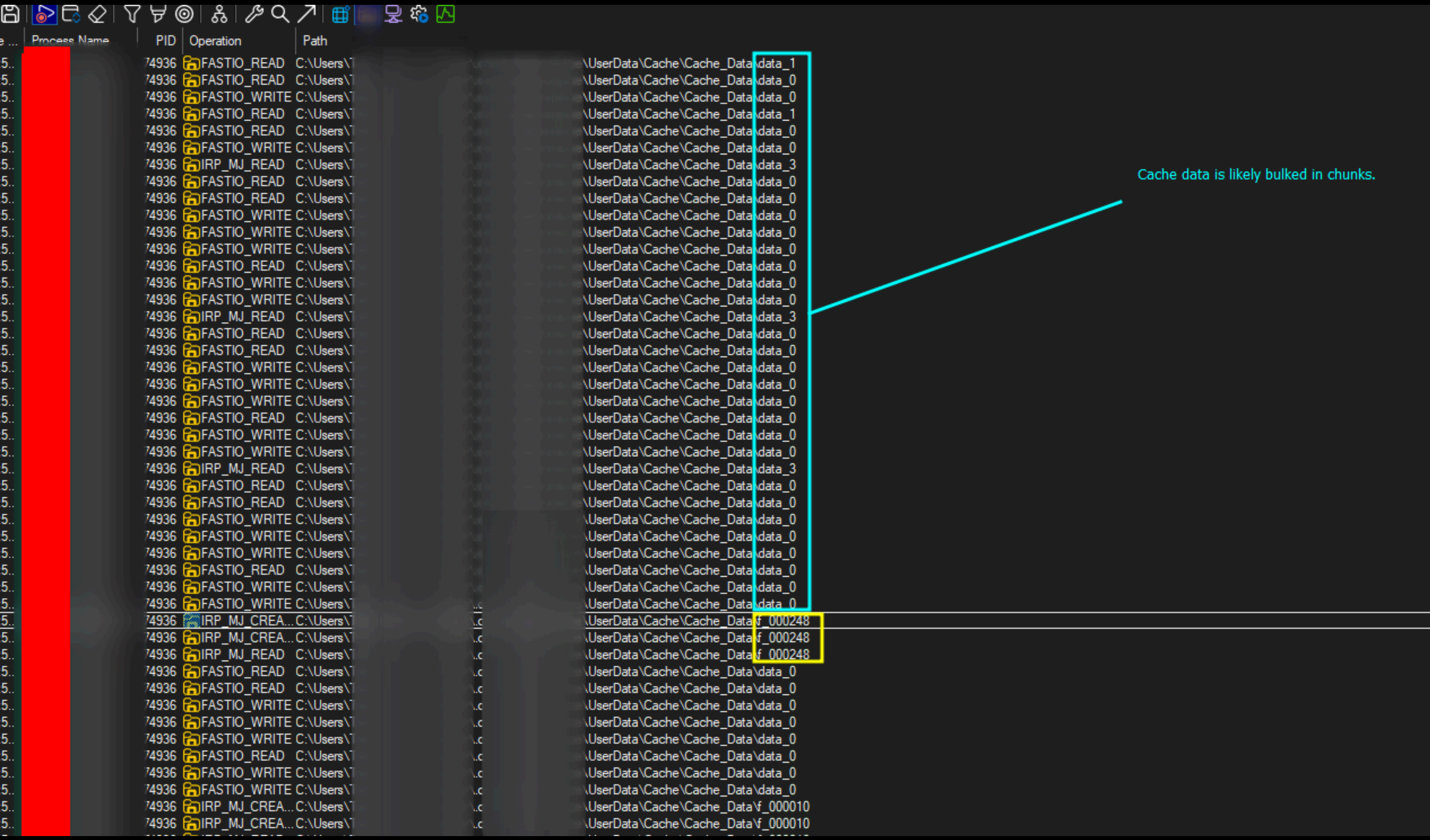
For this lesson, we will be using signature analysis, and structure analysis to identify files.

Scenario 1 - Using HxD To View File Headers

One of the most basic ways to understand a file header is to use a hex editor to grab its header. Or, if it's a file which contains no symbol based header, then a hex editor will give you a good direct road to the headers bytes which may give you some information if you are able to correlate with a file loader, or software which accepts the file by adjusting the bytes.

That being said, the scenario I have for us to explore is wanting to locate an asset loaded inside of a client side Desktop application for Windows. The idea was to use procmon to listen in on the process and see what file calls it makes, however, it was discovered when the app loads the target image we are looking for, it loads hundreds of other files and writes the content to them with no intention of naming the files or validating their extensions.

An example of this scenario in procmon is demonstrated in the screenshot below.



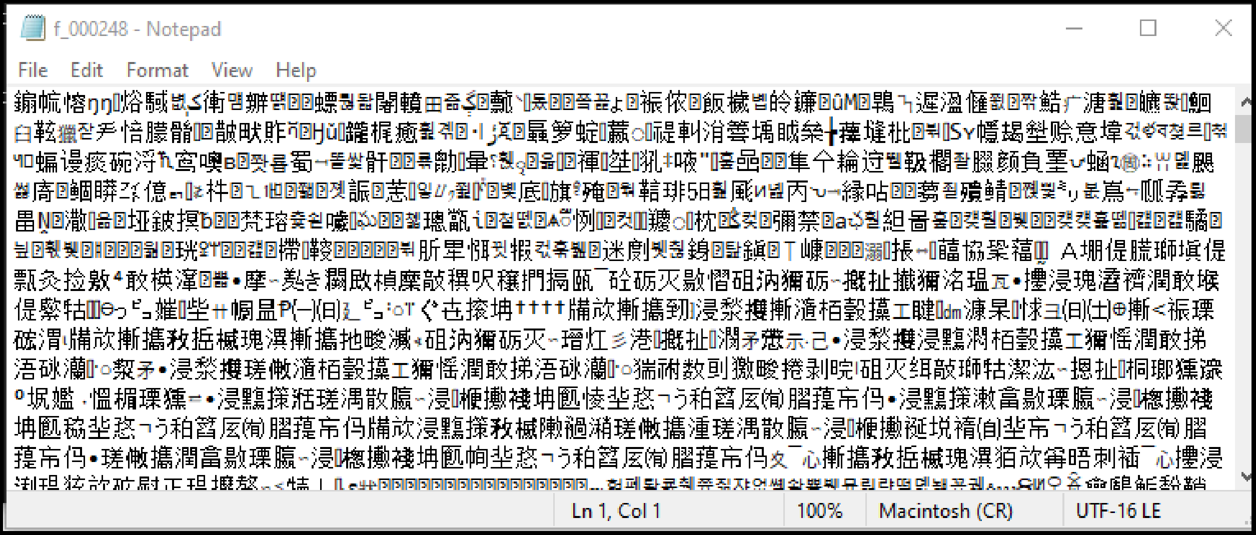
```
Process Name   PID    Operation     Path
[redacted]     74936  FASTIO_READ   C:\Users\[redacted]\UserData\Cache\Cache_Data\data_1
               74936  FASTIO_READ   C:\Users\[redacted]\UserData\Cache\Cache_Data\data_0
               74936  FASTIO_WRITE  C:\Users\[redacted]\UserData\Cache\Cache_Data\data_0
               74936  FASTIO_READ   C:\Users\[redacted]\UserData\Cache\Cache_Data\data_1
               74936  FASTIO_READ   C:\Users\[redacted]\UserData\Cache\Cache_Data\data_0
               74936  FASTIO_WRITE  C:\Users\[redacted]\UserData\Cache\Cache_Data\data_0
               74936  IRP_MJ_READ   C:\Users\[redacted]\UserData\Cache\Cache_Data\data_3
               74936  FASTIO_READ   C:\Users\[redacted]\UserData\Cache\Cache_Data\data_0
               74936  FASTIO_READ   C:\Users\[redacted]\UserData\Cache\Cache_Data\data_0
               74936  FASTIO_WRITE  C:\Users\[redacted]\UserData\Cache\Cache_Data\data_0
               74936  FASTIO_READ   C:\Users\[redacted]\UserData\Cache\Cache_Data\data_0
               74936  FASTIO_WRITE  C:\Users\[redacted]\UserData\Cache\Cache_Data\data_0
               74936  FASTIO_READ   C:\Users\[redacted]\UserData\Cache\Cache_Data\data_0
               74936  FASTIO_WRITE  C:\Users\[redacted]\UserData\Cache\Cache_Data\data_0
               74936  FASTIO_WRITE  C:\Users\[redacted]\UserData\Cache\Cache_Data\data_0
               74936  IRP_MJ_READ   C:\Users\[redacted]\UserData\Cache\Cache_Data\data_3
               74936  FASTIO_READ   C:\Users\[redacted]\UserData\Cache\Cache_Data\data_0
               74936  FASTIO_READ   C:\Users\[redacted]\UserData\Cache\Cache_Data\data_0
               74936  FASTIO_WRITE  C:\Users\[redacted]\UserData\Cache\Cache_Data\data_0
               74936  FASTIO_WRITE  C:\Users\[redacted]\UserData\Cache\Cache_Data\data_0
               74936  FASTIO_WRITE  C:\Users\[redacted]\UserData\Cache\Cache_Data\data_0
               74936  FASTIO_READ   C:\Users\[redacted]\UserData\Cache\Cache_Data\data_0
               74936  FASTIO_WRITE  C:\Users\[redacted]\UserData\Cache\Cache_Data\data_0
               74936  FASTIO_WRITE  C:\Users\[redacted]\UserData\Cache\Cache_Data\data_0
               74936  IRP_MJ_READ   C:\Users\[redacted]\UserData\Cache\Cache_Data\data_3
               74936  FASTIO_READ   C:\Users\[redacted]\UserData\Cache\Cache_Data\data_0
               74936  FASTIO_READ   C:\Users\[redacted]\UserData\Cache\Cache_Data\data_0
               74936  FASTIO_WRITE  C:\Users\[redacted]\UserData\Cache\Cache_Data\data_0
               74936  FASTIO_WRITE  C:\Users\[redacted]\UserData\Cache\Cache_Data\data_0
               74936  FASTIO_WRITE  C:\Users\[redacted]\UserData\Cache\Cache_Data\data_0
               74936  FASTIO_READ   C:\Users\[redacted]\UserData\Cache\Cache_Data\data_0
               74936  FASTIO_WRITE  C:\Users\[redacted]\UserData\Cache\Cache_Data\data_0
               74936  FASTIO_WRITE  C:\Users\[redacted]\UserData\Cache\Cache_Data\data_0
               74936  IRP_MJ_CREA...C:\Users\[redacted]\UserData\Cache\Cache_Data\f_000248
               74936  IRP_MJ_CREA...C:\Users\[redacted]\UserData\Cache\Cache_Data\f_000248
               74936  IRP_MJ_READ   C:\Users\[redacted]\UserData\Cache\Cache_Data\f_000248
               74936  FASTIO_READ   C:\Users\[redacted]\UserData\Cache\Cache_Data\data_0
               74936  FASTIO_READ   C:\Users\[redacted]\UserData\Cache\Cache_Data\data_0
               74936  FASTIO_WRITE  C:\Users\[redacted]\UserData\Cache\Cache_Data\data_0
               74936  FASTIO_WRITE  C:\Users\[redacted]\UserData\Cache\Cache_Data\data_0
               74936  FASTIO_WRITE  C:\Users\[redacted]\UserData\Cache\Cache_Data\data_0
               74936  FASTIO_READ   C:\Users\[redacted]\UserData\Cache\Cache_Data\data_0
               74936  FASTIO_WRITE  C:\Users\[redacted]\UserData\Cache\Cache_Data\data_0
               74936  FASTIO_WRITE  C:\Users\[redacted]\UserData\Cache\Cache_Data\data_0
               74936  IRP_MJ_CREA...C:\Users\[redacted]\UserData\Cache\Cache_Data\f_000010
               74936  IRP_MJ_CREA...C:\Users\[redacted]\UserData\Cache\Cache_Data\f_000010
```

Cache data is likely bulked in chunks.

As this image shows, an application is making some really large sets of calls to open cache data upon running the app.

Note: In the scenario, this was not mentioned, but will be added as a tip to the lesson. When trying to see what files or resources apps may require, attempting to do a few runs on *initialization / on destruction* is really useful for spotting the way applications logically require resources.

So what do we do? Of course we can try to open this file with a notepad and see where it goes.

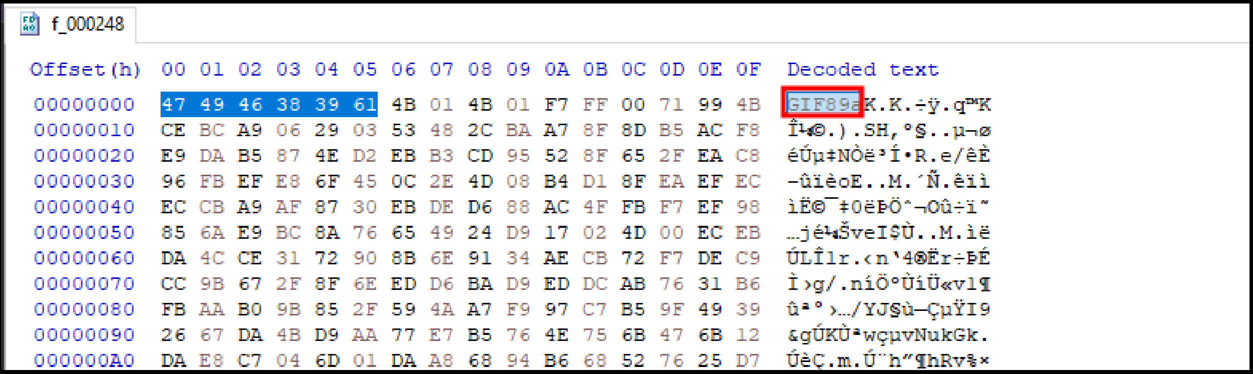


Ah yes, we run into the typical windows issue regarding the use of notepad as a text editor and how it uses code pages. When you open a GIF file (*or any binary file*) in Notepad, it tries to interpret the binary data as text, using a default character encoding (*like UTF-8 or ANSI*). Since the data is not text, and instead a stream of bytes, this results in garbled output after attempting to parse & translate often resulting in this or what looks like random linguistic characters because some binary values map to characters in Asian language code pages.

So we need to use tools like HxD to view the file. For those who are new and are unaware, HxD is a hex editor, which can take in any file, and process its data as a form of 'hex' which is useful for representing data streams as ASCII (*American Standard Code for Information Interchange code page*), which when converted to this will output useful strings of information that can be used to identify specific relics apart of the file.

In many scenarios where you come across wacky files, HxD is going to be one of the first few ways you may notice a files format. Unless you are on Linux, in which case you pipe xxd into grep or other various tools such as 'head' to sort through specific segments of a file.

Opening this file in HxD results in the following output.

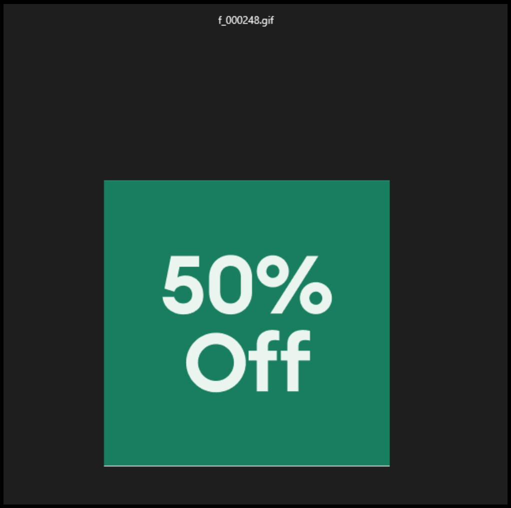


```
f_000248

Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text

00000000  47 49 46 38 39 61 4B 01 4B 01 F7 FF 00 71 99 4B  GIF89aK.K.÷ÿ.q™K
00000010  CE BC A9 06 29 03 53 48 2C BA A7 8F 8D B5 AC F8  Î¼©.).SH,º§..µ¬ø
00000020  E9 DA B5 87 4E D2 EB B3 CD 95 52 8F 65 2F EA C8  éÚµ‡NÒë³Í•R.e/êÈ
00000030  96 FB EF E8 6F 45 0C 2E 4D 08 B4 D1 8F EA EF EC  –ûïèoE..M.´Ñ.êïì
00000040  EC CB A9 AF 87 30 EB DE D6 88 AC 4F FB F7 EF 98  ìË©¯‡0ëÞÖˆ¬Oû÷ï˜
00000050  85 6A E9 BC 8A 76 65 49 24 D9 17 02 4D 00 EC EB  …jé¼ŠveI$Ù..M.ìë
00000060  DA 4C CE 31 72 90 8B 6E 91 34 AE CB 72 F7 DE C9  ÚLÎ1r.‹n‘4®Ër÷ÞÉ
00000070  CC 9B 67 2F 8F 6E ED D6 BA D9 ED DC AB 76 31 B6  Ì›g/.níÖºÙíÜ«v1¶
00000080  FB AA B0 9B 85 2F 59 4A A7 F9 97 C7 B5 9F 49 39  ûª°›…/YJ§ù—ÇµŸI9
00000090  26 67 DA 4B D9 AA 77 E7 B5 76 4E 75 6B 47 6B 12  &gÚKÙªwçµvNukGk.
000000A0  DA E8 C7 04 6D 01 DA A8 68 94 B6 68 52 76 25 D7  ÚèÇ.m.Ú¨h”¶hRv%×
```

And like that, we now understand that this file is a GIF file. But not only is it just any GIF, it is a GIF version 8 which contains very specific structures alluding to more features in the standard and structure. This will also change how the application renders or parses the resources content.



Now, we can take this file and fix it up and make sure the file renders as a GIF. Simply change f_000248 in this case to f_000248.gif and the OS will attempt to load this as a GIF. The image on the left side of this page showcases the success of the rename using the command line utility 'ren' to properly rename files by their extensions.

Scenario 2 - Similar but Unknown Structures

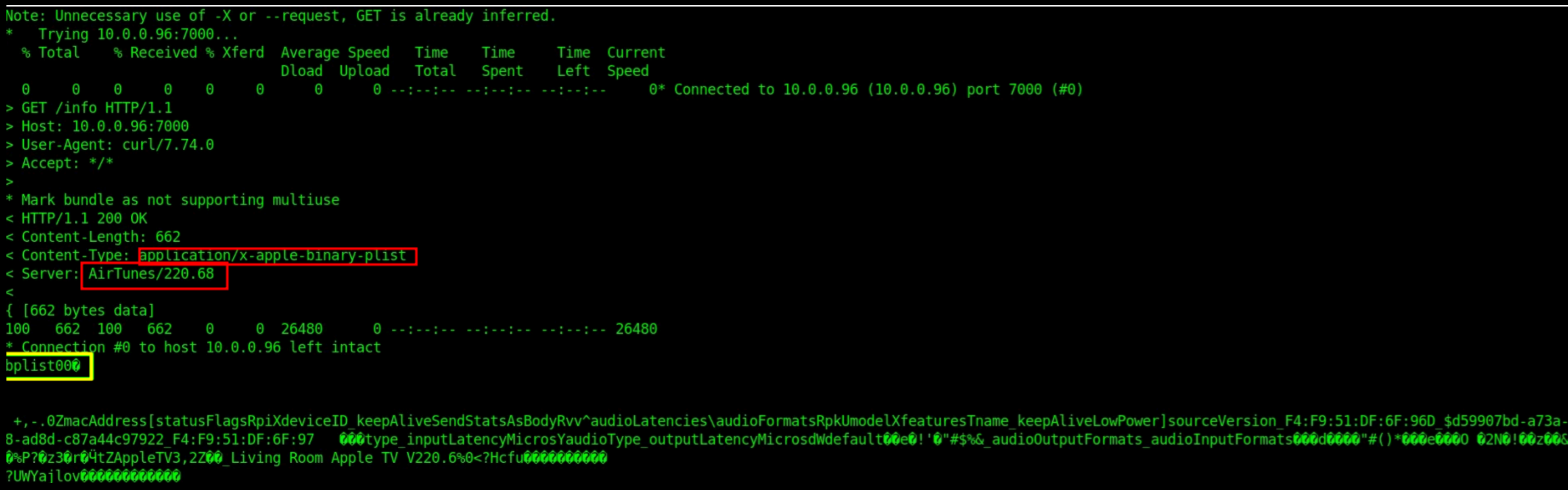
Sometimes, you may come across scenarios, such as reverse engineering proprietary software or IoT devices where you find or stumble across some random configuration file that looks like one file but really renders as another.

The idea is that by understanding how to recognize key differences in specific structured formats, such as JSON and XML, you will be able to recognize when a file format builds on top of that standard.

In the case of BPLIST files, or Binary Property List files, developed by Apple as their custom proprietary file format, BPLIST builds on top of the existing PLIST file format, built on top of XML. BPLIST transforms the PLIST functionalities into a binary-like setting by creating BPLIST, which stores the same key-value pairs in a binary format for performance purposes.

Additionally, this file format hides a lot of information really well from people who do not understand files from a broad perspective. One sample of this is how the information such as the padding, size of objects in the compiled file, and even the offset table for the data is stored at the very end of the file.

The following example is a demonstration of what querying a raw Apple AirTunes Server which hosts a BPLIST file for configuration on a local IoT device looks like.



```
Note: Unnecessary use of -X or --request, GET is already inferred.
*   Trying 10.0.0.96:7000...
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0* Connected to 10.0.0.96 (10.0.0.96) port 7000 (#0)
> GET /info HTTP/1.1
> Host: 10.0.0.96:7000
> User-Agent: curl/7.74.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Content-Length: 662
< Content-Type: application/x-apple-binary-plist
< Server: AirTunes/220.68
<
{ [662 bytes data]
100   662  100   662    0     0  26480      0 --:--:-- --:--:-- --:--:-- 26480
* Connection #0 to host 10.0.0.96 left intact
bplist00�


 +,-.0ZmacAddress[statusFlagsRpiXdeviceID_keepAliveSendStatsAsBodyRvv^audioLatencies\audioFormatsRpkUmodelXfeaturesTname_keepAliveLowPower]sourceVersion_F4:F9:51:DF:6F:96D_$d59907bd-a73a-
8-ad8d-c87a44c97922_F4:F9:51:DF:6F:97   ���type_inputLatencyMicrosYaudioType_outputLatencyMicrosdWdefault��e�!'�"#$%&_audioOutputFormats_audioInputFormats���d����"#()*���e���O �2N�!��z��&
�%P?�z3�r�ЧtZAppleTV3,2Z��_Living Room Apple TV V220.6%0<?Hcfu�����������
?UWYajlov�������������
```

Since this file is compiled, there are very minimal tools for understanding what exactly the file is off of the start. Notice that yes, there is a header specifying BPLIST00, however, when running the file through tools doing research about the file header (*using Google to get ahold of devtools*) the file reading is incorrect and the information never gets decompiled.

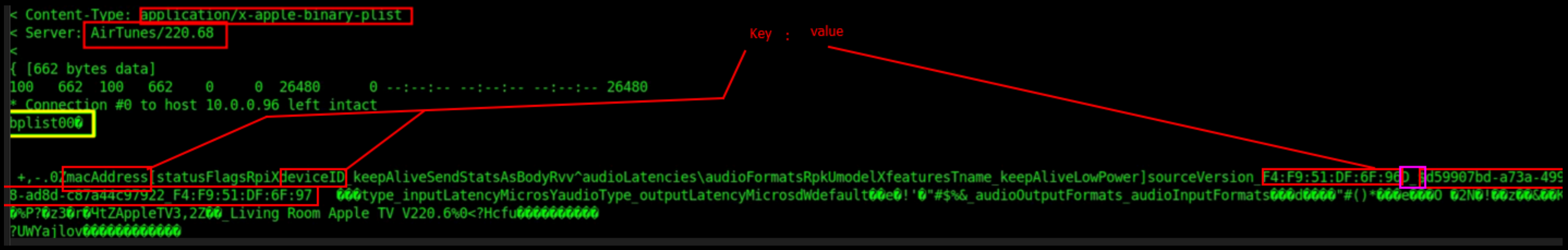
This indicates a few things

- A - Some information for the file is necessary before decoding it.

- B - The version of the file has very very minute changes when used on internal software versus external software. As I have noticed fundamental differences when using tools with this file format as a dev versus taking it and looking at it in other applications.

This kind of stuff may not happen all the time, but when it does, it can often be confusing. And in this case, there is nothing to repair until we actually come up with something that's 'wrong' with the file, more so why it is not working in ways it should.

If you look at a standard PLIST file, however, you can kind of grasp what this structure may look like at first before it gets compiled. Let me map it out again for you.



```
< Content-Type: application/x-apple-binary-plist
< Server: AirTunes/220.68
<
{ [662 bytes data]
100   662  100   662    0     0  26480      0 --:--:-- --:--:-- --:--:-- 26480
* Connection #0 to host 10.0.0.96 left intact
bplist00�


 +,-.0ZmacAddress[statusFlagsRpiXdeviceID_keepAliveSendStatsAsBodyRvv^audioLatencies\audioFormatsRpkUmodelXfeaturesTname_keepAliveLowPower]sourceVersion_F4:F9:51:DF:6F:96D_$d59907bd-a73a-499
8-ad8d-c87a44c97922_F4:F9:51:DF:6F:97   ���type_inputLatencyMicrosYaudioType_outputLatencyMicrosdWdefault��e�!'�"#$%&_audioOutputFormats_audioInputFormats���d����"#()*���e���O �2N�!��z��&��K
�%P?�z3�r�4tZAppleTV3,2Z��_Living Room Apple TV V220.6%0<?Hcfu����������
?UWYajlov������������
```

In the image on the previous page, we take the previous structure and analyze it a bit further, acknowledging the use of map-like structures within the files raw data stream. This is HUGE for us because it allows us to validate that the file most likely is the type of file it is if we continue to validate the way the structure gets parsed using remote sources.

While this is only one piece of the puzzle, it is important to understand that recognizing the files structure and being able to understand how files get created is something worth taking up for this exact scenario.

While we did not necessarily ‘fix’ or ‘patch’ any broken things here, the purpose of this section was to cover specific components that go into fixing files, such as properly identifying and looking at the files structure.

Conclusion for Section

Considering this section kept the beef of the content, I would like to assert that given this is a free lesson, we are trying to gauge readers' understanding of our content. Additionally, this is a way of giving readers the direct resemblance of how we expand on topics.

Section 0x03

> Next Steps Forward

When working with file formats and their information, it can be quite a pain to understand where to go once you get specific information, such as how the software you are looking at deals with ‘x’ files.

To end this course, I would like to give you a small section which covers some baseline ideas for taking the information you obtain with the two scenarios above, and how you can take them further by applying skills such as development to automate them.

Scenario 1 - Using HxD To View File Headers

Scenario 1 required a decent amount of manual analysis to understand how the application was downloading and saving data to files. Being able to recognize the GIF relic assisted us in knowing the fix to view the files graphical content was simply renaming the file rather than patching it as we would need to in other scenarios.

Since we know the application is trying to bundle information into the cache folders, and stores resources such as GIF in there without naming them, we can write a program that opens every single file in the directory recursively which then grabs the first 40 bytes of the files header and checks it against a local signature database/map which auto-renames the file according to the signature detected in the files header.

Taking it even a bit further includes validating specific sections, such as a PNG’s IDENT section before renaming. This prevents resources from being mistaken as something they truly aren’t. For example, GIF’s can be renamed as PNG’s yet still load as their structures relate just enough to create an initial render, but not enough to make them the full file.

Simply, taking this a bit further involves automating the process with scripts or programs built in language’s like Go, which we explore in other lessons.

Scenario 2 - Similar but Unknown Structures

Scenario 1 builds a tool to rename files and correct resource names for easy viewing by looking at the header. But what if the header specified a version of the file format, yet the file structure was not actually related to that version? In such a case, we need to try our best to make attempts at validating the structure, which can allow us to take the identification to another level and correct issues in files using various methods such as hex editing.

I know I brought this up a lot, but did not mention it: Hex editing for file formats involves dumping their contents using software such as HxD and modifying individual sets of bytes to result in specific outcomes, such as steganography which we explore in other lessons as well. “Dynamic” hex editing does not really exist for images that are static, but I use dynamic to represent the process being done by an automated program which was built to modify hex-contents while a file is in the process of being read or used.

That being said, we may take our understanding of the file structure and use it to build out custom utilities and parsers that can deal with the file in scenarios where we may not have access to tools that work, or tools such as developer tools which carry specific features consumer level tools do not.

This is something that is important to keep in mind for, as many companies claim they release the software for working with the files unique structures and features, however purposefully go out of their way to ensure the public never gets the correct functionality to operate outside of specific areas such as development.

This is a form of control that from a security standpoint makes sense, but then again, this is no different than assuming somebody won’t reverse engineer the proprietary one- just manages the attack surface.

Powered by SkyPenguinLabs

Section 0x04

> Lesson Conclusion

Working with file formats is a crucial part of the reversing process. I feel some people, surprisingly, leave out to the scientifical extent. While, reversing does not always call for knowing specific structures or how files work behind the scenes especially as in depth, in very specific scenarios including ones that are starting to become more modern (*such as reversing ElectronJS apps*), this understanding is required to make the processing and operational goals behind the task at hand finish with the correct information.

Additionally, if we want to make sure that we can go from reverse engineering to exploit development, due to the wide use of image, video, and general assets inside of applications, understanding the file format makes it really easy to build exploits (*if plausible*) for the app being reversed.

And that concludes it! Thank you for being a part of the SkyPenguinLabs lesson material and we definitely hope you picked something up from it- I know I definitely do.

~ Totally_Not_A_Haxxer