Analysis Techniques of Executable and Linkable Format (ELF)

MISP-LEA project

Co-funded by the European Union

Team CIRCL
Gérard Wagener

TLP:CLEAR

https://www.circl.lu

20 January, 2025

Who is behind MISP-LEA?

- Proposal submitted to the ISF call **ISF-2022-TF1-AG-CYBER**[1]
- Consortium between **Shadowserver** and **CIRCL**
- Project start date: **June 1, 2023**
- Project duration: **24 months**
- Objective: Create a sharing hub bridging existing sharing communities and **Law Enforcement Agencies (LEA)**

[1] https://ec.europa.eu/info/funding-tenders/opportunities/docs/2021-2027/isf/wp-call/2021-2022/call-fiche_isf-2022-tf1-ag-cyber_en.pdf

MISP-LEA

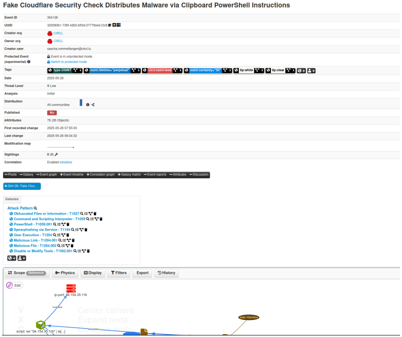
Objectives

- Operational **MISP** & **AIL** platforms for LEAs.
- Operational data feeds from **CIRCL** & **Shadowserver**.
- Bridging connections with other operational sharing communities and the private sector.
- Platforms are operated by **CIRCL**.
- Main benefit for LEAs: Bootstrap investigations.
- Enable seamless information sharing with non-EU members.

Key Figures for 2025

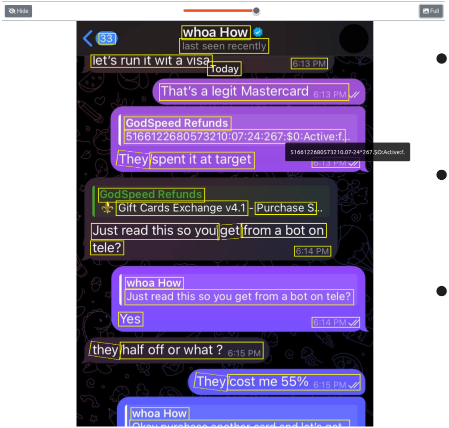
- Access provided to more than 40 LEA agencies. 121 users.
- 2 years of historical data from crawled onion sites, chats,...

MISP



- Foster automated sharing among Law Enforcement Agencies (LEAs).
- Establish connections with other sharing communities, such as ISACs and CTI communities.
- Share crime indicators that fall outside the scope of CSIRT activities.

AIL



- AIL platform enables the analysis of collected information from various sources.
- Focuses on processing data from onion sites, darknet forums, and social media.
- Key benefit: Facilitates automated information extraction for investigations.

What is ELF?

- ELF stands for Executable and Linkable Format[2].
- It is a common standard file format for executables, object code, shared libraries, and core dumps.
- Originally developed by Unix System Laboratories and now widely used in Unix-like operating systems.

[2] `https://refspecs.linuxfoundation.org/elf/elf.pdf`

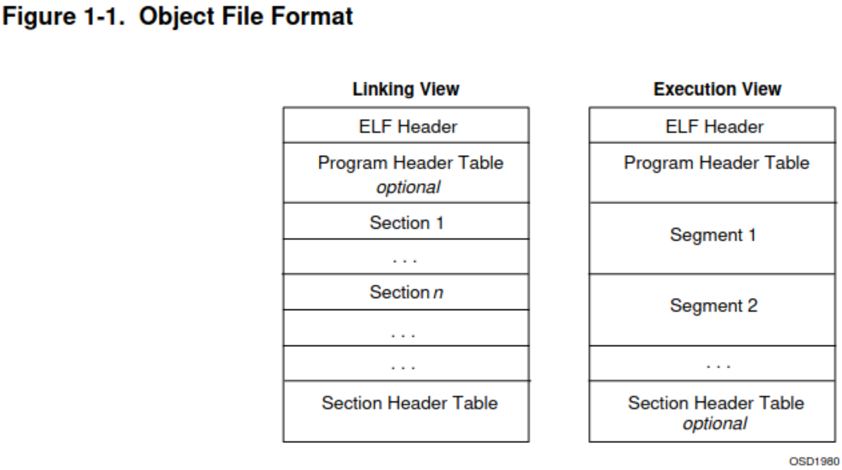
Structure of an ELF File

- An ELF file consists of three main parts:
  - **Header:** Contains metadata about the file type, architecture, and entry point.
  - **Program Header Table:** Describes how the file should be loaded into memory.
  - **Section Header Table:** Provides information about the sections in the file.
- ELF files are designed to be flexible and extensible.

Benefits of ELF

- Platform-independent format, enabling portability.
- Simplifies the linking and loading process.
- Supports dynamic linking, reducing redundancy.
- Extensively used in modern development environments.

ELF



3

[3] Reference: https://refspecs.linuxfoundation.org/elf/elf.pdf

Binwalk Output

```
Sample: 6420
   f5d7d48b75d687b8356e93c82721bb536c633d773f8985f
```

```
binwalk sample
```

Decimal	Hexadecimal	Description
0	0x0	ELF, 32-bit LSB executable, Intel 80386, version 1 (SYSV)
13111	0x3337	Boot section Start 0x58028941 End 0x5A41
13115	0x333B	Boot section Start 0x5A41 End 0x0

→ matched signatures → false positives

Using Binwalk

Sample:
9e70725640c4284e2049e4b25c9cc46cca496053cebf69855ec25acc9bd63e05

Decimal	Hexadecimal	Description
0	0x0	ELF, 64-bit LSB executable, AMD x86-64, version 1 (GNU/Linux)
600864	0x92B20	Unix path: /usr/share/locale
612774	0x959A6	Unix path: /usr/lib/getconf
620336	0x97730	Unix path: /usr/lib/locale
622368	0x97F20	Unix path: /usr/lib/locale/locale-archive
674903	0xA4C57	Unix path: /usr/lib/x86_64-linux-gnu/
778039	**0xBDF37**	**mcrypt 2.2 encrypted data, algorithm: blowfish-448, mode: CBC, keymode: 8bit**

Using Binwalk

- **Encrypted Data:**
  - The file contains data encrypted using **mcrypt 2.2**.
- **Encryption Algorithm:**
  - Algorithm: **Blowfish-448**, a symmetric block cipher with a 448-bit key size.
- **Cipher Mode:**
  - Mode: **CBC (Cipher Block Chaining)** for enhanced security via block interdependency.
- **Key Mode:**
  - Key processed in **8-bit mode**, possibly a default for mcrypt configurations.
- **Implications:**
  - Decryption requires the encryption key and potentially an initialization vector (IV).
  - Indicates sensitive or protected data within the file.
  - Poses a reverse engineering challenge without the key.

Extracting the content

Sample:
9e70725640c4284e2049e4b25c9cc46cca496053cebf69855ec25acc9bd63e05

```
dd if=sample of=extracted_data bs=1 skip
   =778039
```

- Binwalk uses signatures to identify and extract data from files.
- Determine the size of the detected block for further analysis.
- Evaluate whether the detection is a false positive by inspecting the data manually or using additional tools.

ELF Symbols from Binary Analysis

Extract symbols from binary excluding GBLIBC references
Sample:
6420f5d7d48b75d687b8356e93c82721bb536c633d773f8985f74c8977425f04

```
nm sample | grep -v GBLIBC
```

```
08048bfd t p4tch_sel1nux_codztegfaddczda
08048e9c t parse_cred
8050bb3 T prepare_fops_lsm_shellcode
08049215 t put_your_hands_up_hooker
0804b220 D r1ngrrrrrrr
0804988e t rey0y0code
0804b2c0 d ruujhdbgatrfe345
```

ELF Symbols from Binary Analysis

- Interpretation of the output of tool `nm`
- `man page is your friend`

Symbol Type	Explanation
a	The symbol's value is absolute and will not be changed by further linking.
b	The symbol is in the BSS data section.
d	The symbol is in the initialized data section.
r	The symbol is in the read-only data section.
t	The symbol is in the text (code) section.
w	The symbol is a weak symbol that has not been specifically tagged as a weak object symbol.

Using objdump to View ELF Sections

```
objdump -h sample
```

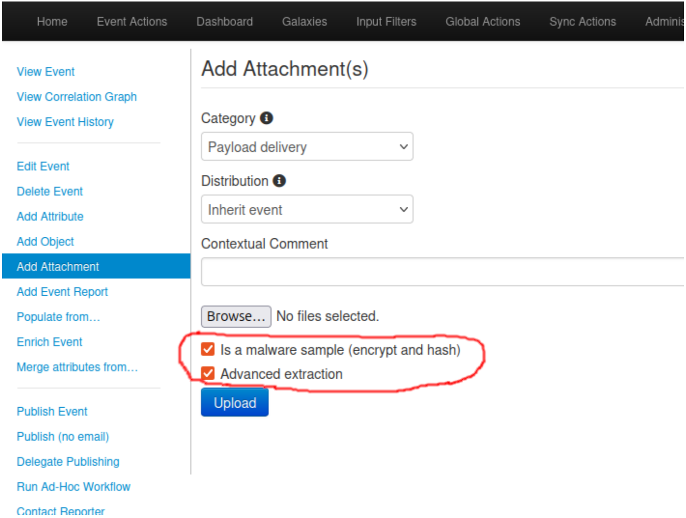
- **Output Structure:**
  - Lists all sections in the ELF file, including their attributes.
  - Provides information such as:
    - **Idx**: Section index in the ELF file.
    - **Name**: Name of the section (e.g., '.text', '.data').
    - **Size**: Size of the section in bytes.
    - **VMA (Virtual Memory Address)**: Where the section is loaded in memory.
    - **File Off**: Offset of the section in the binary file.
    - **Attributes**: Flags indicating section properties (e.g., 'ALLOC', 'LOAD', 'READONLY').
- **Use Case:**
  - Identify key sections like '.text' (code), '.data' (initialized data), '.bss' (uninitialized data), and '.dtor' (destructors).
  - Useful to identify the type of binary, such as a C program, C++, Go (Golang), etc.

ELF Section Details

Idx	Name	Size	VMA	LMA	File Off
0	.interp	00000013	08048134	08048134	00000134
1	.note.ABI-tag	00000020	08048148	08048148	00000148
2	.gnu.hash	00000030	08048168	08048168	00000168
3	.dynsym	00000290	08048198	08048198	00000198
11	.text	00001788	080489b0	080489b0	000009b0

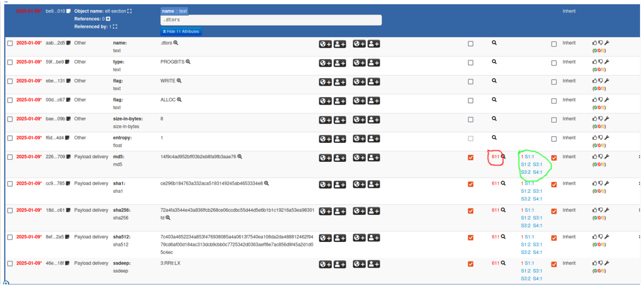
Idx	Attributes
0	CONTENTS, ALLOC, LOAD, READONLY, DATA
1	CONTENTS, ALLOC, LOAD, READONLY, DATA
2	CONTENTS, ALLOC, LOAD, READONLY, DATA
3	CONTENTS, ALLOC, LOAD, READONLY, DATA
11	CONTENTS, ALLOC, LOAD, READONLY, **CODE**

Collaborative Malware Analysis Using MISP



Uploading your sample to MISP

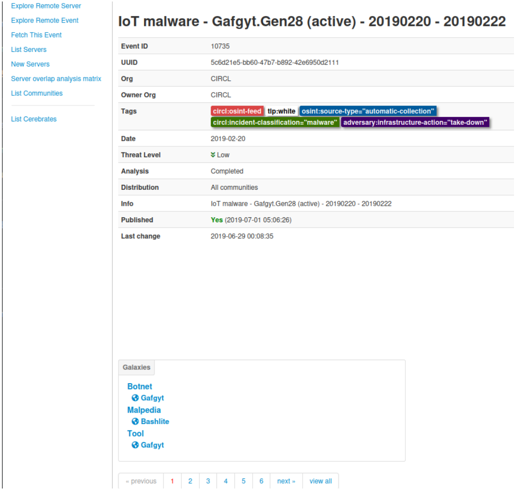
Collaborative Malware Analysis Using MISP



- Explore correlations between events and indicators.
- Analyze results from threat intelligence feeds.
- Review hits from synchronization caches.
- Watch out for false positives. Check the size of the section, as smaller sizes are more susceptible to false positives.

Collaborative Malware Analysis Using MISP

Exploring connected MISP instances within MISP-LEA



Kunai: what is it?

Kunai[4] is a security monitoring tool focusing on **threat detection** and **threat hunting tasks**. For those familiar with **Microsoft Sysmon**[5] you can view **Kunai** as its alter-ego for **Linux** systems.

It allows the monitoring of several system-related events:

- binary / script execution
- shared objects loaded
- drivers loaded
- eBPF programs loaded
- ...

List of **events**: `https://why.kunai.rocks/docs/events/`

[4] `https://github.com/kunai-project`

[5] https://learn.microsoft.com/en-us/sysinternals/downloads/sysmon

Example: execve event


```json
{
  "data": {
    "ancestors": "kernel|kernel",
    "parent_exe": "kernel",
    "command_line": "/sbin/modprobe -q -- net-pf-10",
    "exe": {
      "path": "/usr/bin/kmod",
      "md5": "08220eec2f1a1f3690a2d6b2a634d255",
      "sha1": "4dd4f7a269c9d18d755176bcf44bcef86abe2633",
      "sha256": "cc064683b03c958347f2a7d13ee9d4523434674e2599c2ca710f923dc44b0a5b",
      "sha512":
"87d3057d6881b5256bf1ae93386d9b615f1afe11c3c90ae2e71eb68d9cf4f550205135ffd5cf24ca6fa72e08edf56110bd70a9ca5c5448
283b5939384ff64813",
      "size": 166080,
      "error": null
    }
  },
  "info": {
    "host": "...",
    "event": {
      "source": "kunai",
      "id": 1,
      "name": "execve",
      "uuid": "e97b8ca5-f6bd-c206-afbd-701c0d61a9d9",
      "batch": 605
    },
    "task": "...",
    "parent_task": "...",
    "utc_time": "2024-10-29T12:47:58.834535124Z"
  }
}
```


NB: parts with **"..."** are elided for sake of space, please read documentation to understand the full event format.

How can it be used for binary analysis?

Spoiler alert: the primary goal of **Kunai** is not to be a binary analysis tool. Therefore it does not contain any advanced anti-analysis countermeasure some malware may implement.

Yet we believe it can be useful to achieve the following:

- Get a quick overview of the capacities of a malware sample
- It is monitoring **system-wide** events, so it catches some execution indirections:
  - cronjobs
  - services
  - dynamic linker tricks (example: LD_PRELOAD trick)
  - ...
- **Kunai output** can be directly **shared**, **used** as **IoC**, or to create **detection rules**.

Analysis process, in theory

1. Run **Kunai** on a machine dedicated to **dynamic malware analysis** (ideally a Virtual Machine).
2. Run the malware sample you want to look at.
3. Let the malware run for some time so that you can capture the maximum of its activity.
4. Collect the **Kunai** traces and analyze them.
5. **Optional**: build **detection rules**[6], extract **IoCs**, and share them.

[6] `https://why.kunai.rocks/docs/advanced/rule_configuration#detection-rules`

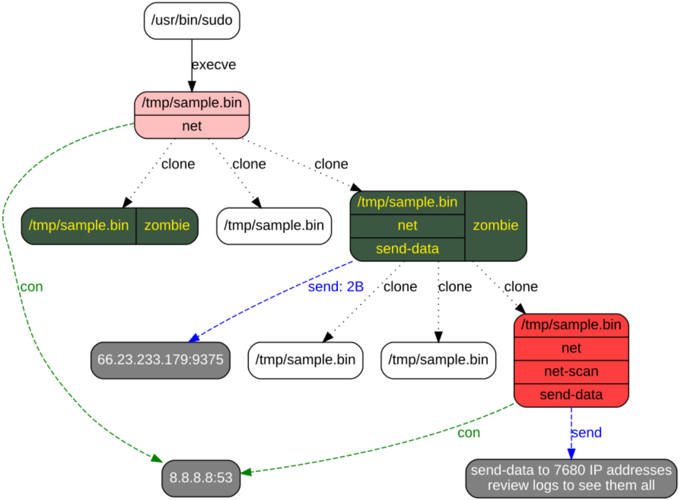
Analysis process in practice

Use our **Kunai sandbox** project:
`https://github.com/kunai-project/sandbox`

- It automates the procedure explained in the previous slide.
- It can be used to analyze samples from different architectures (currently **x86_64** and **amd64**) → can be used to analyze **IoT and mobile devices** malware.

Example: Mirai

Malware activity graph built from **Kunai** logs:



Going Further

- Read the documentation: `https://why.kunai.rocks/`
- Do hands-on exercises: `https://github.com/kunai-project/workshops`
- Check out some malware traces: `https://helga.circl.lu/NGSOTI/malware-dataset`
- Contribute:
  - Join the Discord channel.
  - Open issues for bugs, feature requests, ...
  - Give feedback: what you like and what you don't like.

Disassembly of <main> (Part 1) with objdump

```
8049d05 <main>:
8049d05: 8d 4c 24 04            lea    0x4(%esp),%ecx
8049d09: 83 e4 f0               and    $0xfffffff0,%esp
8049d0c: ff 71 fc               push   -0x4(%ecx)
8049d0f: 55                     push   %ebp
8049d10: 89 e5                  mov    %esp,%ebp
8049d12: 51                     push   %ecx
8049d13: 83 ec 34               sub    $0x34,%esp
8049d16: 89 4d e4               mov    %ecx,-0x1c(%ebp)
8049d19: c7 04 24 cc a5 04 08   movl   $0x804a5cc,(%esp)
8049d20: e8 37 ec ff ff         call   804895c <puts@plt>
8049d25: e8 82 eb ff ff         call   80488ac <getuid@plt>
8049d2a: 85 c0                  test   %eax,%eax
```

Disassembly of <main> (Part 2) with objdump

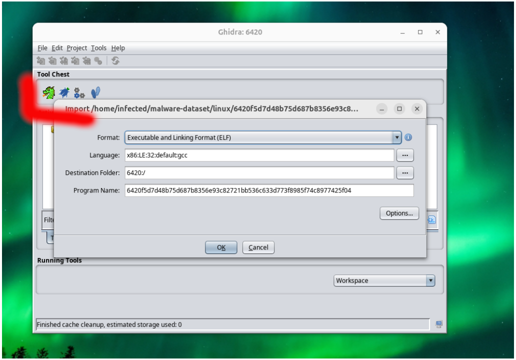
```
8049d47: c7 04 24 42 a6 04 08   movl  $0x804a642,(%esp)
8049d4e: e8 89 eb ff ff         call  80488dc <fwrite@plt>
8049d53: c7 45 e8 01 00 00 00   movl  $0x1,-0x18(%ebp)
8049d5a: e9 1c 03 00 00         jmp   804a07b <main+0x376>
8049d5f: 8b 55 e4               mov   -0x1c(%ebp),%edx
8049d62: 8b 42 04               mov   0x4(%edx),%eax
8049d65: 89 44 24 04            mov   %eax,0x4(%esp)
8049d69: 8b 55 e4               mov   -0x1c(%ebp),%edx
8049d6c: 8b 02                  mov   (%edx),%eax
8049d6e: 89 04 24               mov   %eax,(%esp)
8049d71: e8 e2 f8 ff ff         call  8049658 <env_prepare>
8049d76: e8 59 fa ff ff         call  80497d4 <y0y0stack>
8049d7b: e8 b1 fa ff ff         call  8049831 <y0y0code>
```

Introduction Ghidra

- **Disassembly and Decompilation:**
  - Transforms binary code into human-readable assembly.
  - Generates high-level language representations (C-like pseudocode).
- **Cross-Platform Support:**
  - Analyzes binaries for multiple architectures (x86, ARM, MIPS, etc.).
  - Compatible with various operating systems (Windows, Linux, macOS).
- **Collaboration:**
  - Supports multi-user reverse engineering projects.
  - Version-controlled changes for shared analysis.
- **Scriptability:**
  - Customize and automate analysis with Python and Java.
- **Extensibility:**
  - Add plugins and extend functionality for specific needs.
- **Data Flow Analysis:**
  - Tracks variables, functions, and references for better insight.

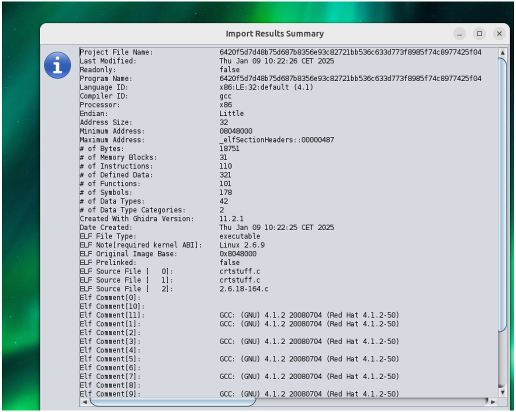
Static Analysis Using Ghidra

- Creating a project in Ghidra.
- Importing and analyzing a binary file.



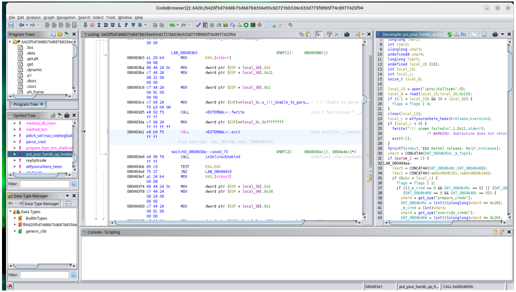
Static Analysis Using Ghidra

- Determine the type of binary (e.g., ELF, PE).
- Analyze the binary's metadata for key attributes such as architecture, endianness, and sections.



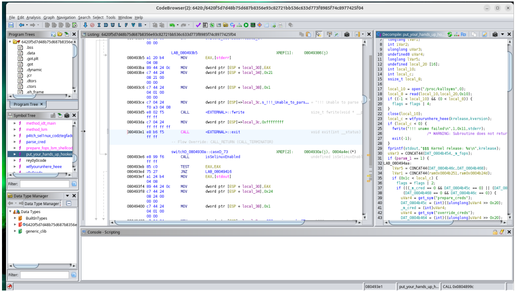
Static Analysis Using Ghidra

- Explore the functions defined within the binary.
- Analyze the disassembly view to examine low-level instructions.
- Utilize the decompiled view for a high-level representation of the code.



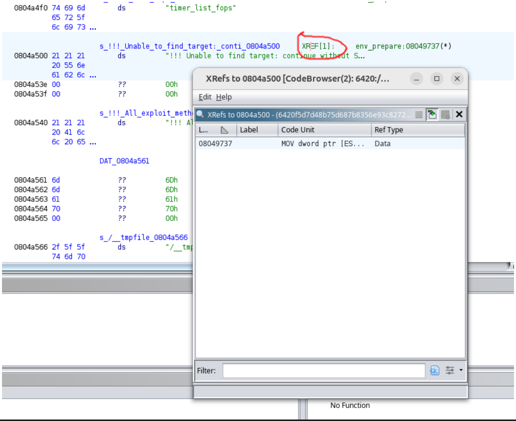
Static Analysis Using Ghidra

- **Benefits of Ghidra's Decompiled View:**
  - Provides a high-level, human-readable representation of the code.
  - Simplifies understanding of complex binaries.
- **Avoid Manual Pattern Matching:**
  - Eliminates the need to manually match patterns in assembly code.
  - Speeds up the reverse engineering process.



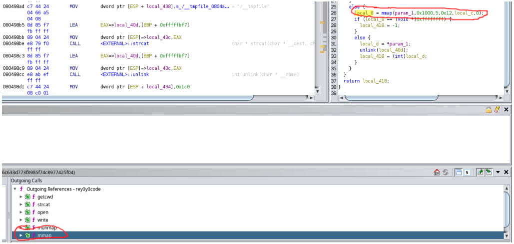
String Analysis and Cross-References in Ghidra

- Identify interesting strings, such as filenames, hardcoded paths, or error messages.
- Use the cross-references (Xrefs) feature to determine which functions or code sections utilize these strings.



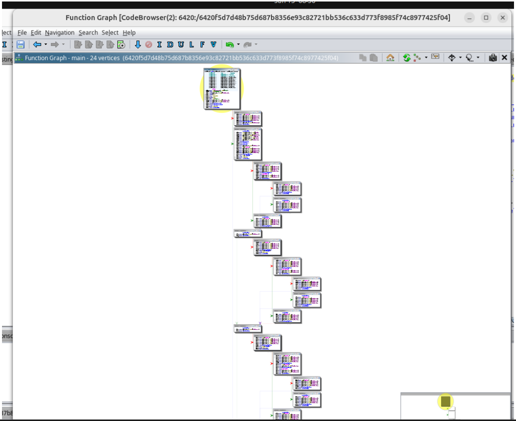
String Analysis and Function Call Trees in Ghidra

- Certain functions are known to generate forensic artifacts, such as ‘fopen‘ and ‘mmap‘.
- Locate these functions in the function call tree to identify which functions use them.
- Determine the artifacts that can be leveraged for detection and analysis.



Static Analysis and Function Call Graphs in Ghidra

- Visual representations of function call graphs provide valuable insights into program behavior.
- Insights include identifying parsing activities, code execution loops, and function relationships.



Core Dumps on Ubuntu

- **What is a Core Dump?**
  - A core dump is a snapshot of a program's memory at the moment it crashes.
  - Used for debugging to analyze the cause of the crash.
- **Where to Find Core Dumps in Ubuntu?**
  - Default location: `/var/lib/apport/coredump`.
  - When using `systemd`, they may be in `/var/lib/systemd/coredump`.
  - Core dumps may also be written to the program's working directory or as specified by `/proc/sys/kernel/core_pattern`.
- **Configuring Core Dumps:**
  - Set unlimited size: `ulimit -c unlimited`.
  - Check core pattern: `cat /proc/sys/kernel/core_pattern`.
  - Enable or configure core dumps in `/etc/security/limits.conf`.

Analyzing Crash Reports

Problem Type: Crash
Architecture: amd64
Crash Counter: 1
Date: Thu Jan 9 15:51:49 2025
Dependencies:

- adduser 3.137ubuntu1
- adwaita-icon-theme 46.0-1
- apt 2.7.14build2
- apt-utils 2.7.14build2
- at-spi2-common 2.52.0-1build1
- at-spi2-core 2.52.0-1build1
- base-passwd 3.6.3build1
- ca-certificates 20240203
- dbus 1.14.10-4ubuntu4.1
- ...

Analyzing a Base64-Encoded Core Dump

Crash Report Details:

- **Source Package:** zoom
- **System Info:** Linux 6.8.0-51-generic x86_64
- **User Groups:** adm, cdrom, dip, kvm, libvirt, lpadmin, plugdev, sudo, users
- **Core Dump Format:** Base64 Encoded

Base64 Blob (Partial):

H4sICAAAAAAC/0NvcmVEdW1wAA==
7J0HgBPV2v5nYUGaGhuiog5WLECogqJEEQVBjCiKlSy7C6y0sLsgYIsFxZ5r
74169drQ2LnW2LvGjj2Wq

Note: Decode the Base64 blob to retrieve the original core dump using the following command:

```
echo "H4sICAAAAAAC/0NvcmVEdW1wAA==" | base64 -d > coredump.g
gunzip coredump.gz
```

Extracting Core Dumps from Crash Files

- Unzipping the core dump creates a file such as `_opt_zoom_ZoomWebviewHost.1000.crash`.
- Decoding and decompressing the binary blob will produce a core dump.

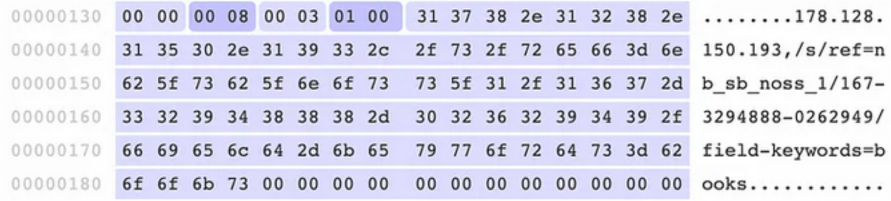
Extracted Core Dump Details

Format: ELF 64-bit LSB core file, x86-64
Details: SVR4-style, from `/opt/zoom/ZoomWebviewHost --type=utility --utility-sub-type=screen_ai.mojom.Scr`
User Info: real uid: 1000, effective uid: 1000, real gid: 1000, effective gid: 1000
Exec Path: `/opt/zoom/ZoomWebviewHost`
Platform: x86_64

Note: Use tools like `gdb`, `readelf`, or `objdump` to analyze the extracted core dump.

Analyzing Unknown Formats

- Threat actors often use customized binary formats for encoding.
- Malware configuration parsing[7].
- Beacons of remote access tools, such as Cobalt Strike.



```
00000130  00 00 00 08 00 03 01 00  31 37 38 2e 31 32 38 2e  ........178.128.
00000140  31 35 30 2e 31 39 33 2c  2f 73 2f 72 65 66 3d 6e  150.193,/s/ref=n
00000150  62 5f 73 62 5f 6e 6f 73  73 5f 31 2f 31 36 37 2d  b_sb_noss_1/167-
00000160  33 32 39 34 38 38 38 2d  30 32 36 32 39 34 39 2f  3294888-0262949/
00000170  66 69 65 6c 64 2d 6b 65  79 77 6f 72 64 73 3d 62  field-keywords=b
00000180  6f 6f 6b 73 00 00 00 00  00 00 00 00 00 00 00 00  ooks............
```

Config field 0×8 showing an example blob structure in the sample data

Image source:[8].

[7] https://github.com/TeamT5/MalCfgParser

[8] https://sixdub.medium.com/using-kaitai-to-parse-cobalt-strike-beacon-configs-f5f0552d5a6e

Setting up Kaitai Struct

- The latest release (as of 2022) is available on GitHub: `https://github.com/kaitai-io/kaitai_struct`.
- To build Kaitai Struct from sources:
  - Ensure you have **Scala SBT (sbt)** installed.
  - Clone the repository and run the build commands.
- Command sequence for building Kaitai Struct.

```
git clone --recursive \
    https://github.com/kaitai-io/kaitai_struct.git
sbt compile
sbt compilerJVM/universal:packageBin
unzip unpack the zip file in \
    kaitai_struct/compiler/jvm/target/universal/kaitai

kaitai-struct-compiler -h
```

Setting up Kaitai Struct Python Environment

To set up the Python environment for Kaitai Struct, follow these steps:

```
python3 -m venv venv
source venv/bin/activate
pip3 install kaitaistruct
python3 parse.py
```

Important: Ensure you stay within the virtual environment. Exiting the virtual environment may prevent your script from running as expected.

Custom Format used in Kaitai Struct Example

The following is an example of a '.ksy' file for Kaitai Struct:

Offset (Bytes)	Field Name	Description
0x00–0x03	Header	4-byte unsigned integer (u4)
0x04–0x0A	Body	8 bytes of data

Table: Structure of the Example Data Format

Offset	00	01	02	03	04	04	05	06	07	08	09	A
Content	02	d2	49	96	62	61	64	63	66	65	68	67

Table: Visualization of the Example File

Description of custom binary format in YAML

Create an example.ksy file

```yaml
meta:
  id: example
  title: Example Binary Format
  endian: le
seq:
  - id: header
    type: u4
  - id: body
    size: 8
```

Transform it into python code

```
kaitai-struct-compiler -t python example.ksy
```

Generated Python File

```python
# This is a generated file! Please edit source .ksy file
    and use kaitai-struct-compiler to rebuild

import kaitaistruct
from kaitaistruct import KaitaiStruct, KaitaiStream,
    BytesIO

class Example(KaitaiStruct):
    def __init__(self, _io, _parent=None, _root=None):
        self._io = _io
        self._parent = _parent
        self._root = _root if _root else self
        self._read()

    def _read(self):
        self.header = self._io.read_u4le()
        self.body = self._io.read_bytes(8)
```

Using your generated python class

```python
from example import Example

# Open the binary file
with open("data.bin", "rb") as f:
    data = Example.from_io(f)

# Access parsed fields
print(f"Header: {data.header}")
print(f"Body: {data.body}")
```

Kaitai Struct Formats - Overview

- The Kaitai Struct community actively **publishes** formats that can be parsed using Kaitai Struct.
- Explore available formats:
  - Community repository: `https://github.com/kaitai-io/kaitai_struct_formats/`
  - Example: Parsing ELF files: `https://github.com/kaitai-io/kaitai_struct_formats/blob/master/executable/elf.ksy`
- Formats cover a wide range of applications, including:
  - Databases
  - Windows-related formats
  - Serialization
  - Security
  - Networking
  - Media
  - MacOS
  - Filesystems

Kaitai Struct Formats - Categories (1/2)

- **Databases**:
  - SQLite3
- **Windows**:
  - LNK files
  - Minidump
  - Shell items
  - System time
  - Registry
- **Serialization**:
  - BSON
  - Chrome
  - Google Protobuf
  - Microsoft CFB
  - MGSPack
  - PHP serialized
  - Python CPickle
  - Ruby Marshal

Kaitai Struct Formats - Categories (2/2)

- **Security**:
  - EFI variable signature
  - SSH public key
- **Networking**:
  - Bitcoin transaction key
  - WebSocket
- **Media**:
  - Android OpenGL shaders cache
  - WAV
- **MacOS**:
  - DS_Store
  - Mac OS resource
- **Filesystems**:
  - LUKS
  - VDI

Decrypting files without access to a tool

Problem Statement

- Faced with a large number of encrypted files.
- Encryption uses a custom implementation.
- No available command-line tool for decryption.
- The key and/or IV has been recovered.
- Debugging and manual decryption of each file is time-consuming and inefficient.

Decrypting Files Without Access to a Tool

Traditional Approach

- Write a loader program to execute the code.
- Read the code into a buffer.
- Cast the buffer to a function pointer.
- Execute the function pointer.
- **Challenges:**
  - Buffers are often protected against code execution.
  - Requires fiddling with `mmap` and `mprotect`.
  - The code might include malicious instructions that went unidentified.
  - The decryptor may be designed for another CPU architecture (e.g., MIPS, RISC-V).

Decrypting Files Without Access to a Tool

- The Unicorn Engine[9] is a CPU emulator based on QEMU.
- Supports multiple architectures:
  - ARM, ARM64 (ARMv8), m68k, MIPS, PowerPC, RISC-V, S390x (SystemZ), SPARC, TriCore, and x86 (including x86_64).
- Provides bindings for various programming languages:
  - Pharo, Crystal, Clojure, Visual Basic, Perl, Rust, Haskell, Ruby, Python, Java, Go, D, Lua, JavaScript, .NET, Delphi/Pascal, and MSVC.
- Offers hooking capabilities for:
  - **Memory access**, **executed instructions**, and **interrupts**.
- Thread-safe[10]
- Works without modifying code (e.g., no need to insert instructions such as `INT3` or `0xCC`).

[9] `https://www.unicorn-engine.org/`

[10] Multithreading is often used as an anti-debugging technique.

Building Unicorn Engine

Prerequisites

Install the required tools:

- `cmake`
- `pkg-config`

Command:

```
sudo apt install cmake pkg-config
```

Building Unicorn Engine

Build Steps

Follow these steps to build Unicorn:

1. Create and navigate to the build directory:

   ```
   mkdir build; cd build
   ```

2. Run `cmake` with the release build type:

   ```
   cmake .. -DCMAKE_BUILD\_TYPE=Release
   ```

3. Compile the project and install it:

   ```
   make & make install
   ```

XOR Cipher Example in C 1/2

```c
#include <stdio.h>
#include <string.h>
// Function to encrypt/decrypt a string using XOR cipher
void xor_cipher(char *data, char key) {
    for (int i = 0; i < strlen(data); i++) {
        data[i] ^= key; // XOR each character with the key
    }
}
```

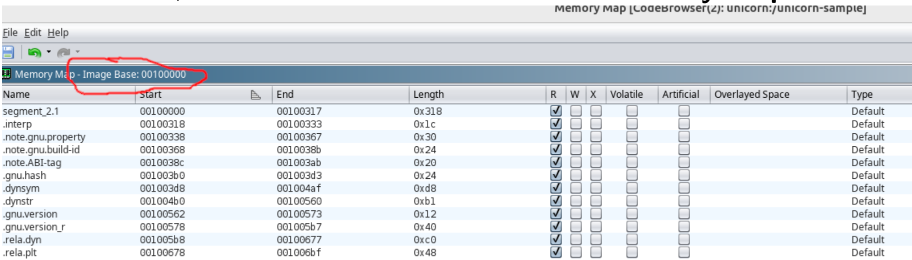
XOR Cipher Example in C 2/2

```
int main() {
    char data[] = "Hello, World!";  // Message to encrypt
    char key = 'K';                 // Encryption key

    printf("Original: %s\n", data);

    // Encrypt the data
    xor_cipher(data, key);
    printf("Encrypted: %s\n", data);

    // Decrypt the data
    xor_cipher(data, key);  // Apply XOR again with the same
        key to decrypt
    printf("Decrypted: %s\n", data);

    return 0;
}
```

```
    gcc -o sample sample.c
```

Determining Base Address

In Ghidra, click on **Window** and then select **Memory Map**.



Name	Start	End	Length	R	W	X	Volatile	Artificial	Overlayed Space	Type
segment_2.1	00100000	00100317	0x318	✓						Default
.interp	00100318	00100333	0x1c	✓						Default
.note.gnu.property	00100338	00100367	0x30	✓						Default
.note.gnu.build-id	00100368	0010038b	0x24	✓						Default
.note.ABI-tag	0010038c	001003ab	0x20	✓						Default
.gnu.hash	001003b0	001003d3	0x24	✓						Default
.dynsym	001003d8	001004af	0xd8	✓						Default
.dynstr	001004b0	00100560	0xb1	✓						Default
.gnu.version	00100562	00100573	0x12	✓						Default
.gnu.version_r	00100578	001005b7	0x40	✓						Default
.rela.dyn	001005b8	00100677	0xc0	✓						Default
.rela.plt	00100678	001006bf	0x48	✓						Default

Determining the Start Address of the Function

- **Challenge:** Identify the function or code block responsible for encryption.
- **Approach:** Look for functions containing numerous arithmetic and bitwise operations:
  - Arithmetic operations: `ADD`, `SUB`, `MUL`, `DIV`.
  - Bitwise operations: `XOR`, `SHR`, `SHL`.
- Check for these operations grouped in blocks or loops.

```
00101189 f3 0f 1e fa     ENDBR64
0010118d 55              PUSH      RBP
0010118e 48 89 e5        MOV       RBP,RSP
00101191 53              PUSH      RBX
00101192 48 83 ec 28     SUB       RSP,0x28
00101196 48 89 7d d8     MOV       qword ptr [RBP + local_30],RDI
0010119a 89 f0           MOV       EAX,ESI
0010119c 88 45 d4        MOV       byte ptr [RBP + local_34],AL
0010119f c7 45 ec        MOV       dword ptr [RBP + local_1c],0x0
         00 00 00 00
001011a6 eb 26           JMP       LAB_001011ce
```

Determining the End of a Function

- Functions often end with the **RET** instruction.

```
             ff ff
00101le0 48 39 c3       CMP      RBX,RAX
00101le3 72 c3          JC       LAB_00101la8
00101le5 90             NOP
00101le6 90             NOP
00101le7 48 8b 5d f8    MOV      RBX,qword ptr [RBP + local_10]
00101leb c9             LEAVE
00101lec c3             RET

                 **************************************************************
                 *                          FUNCTION                          *
                 **************************************************************
```

Python Code Example: Hooking in Unicorn Engine

Listing 1: Hooking Example with Unicorn Engine

```python
from unicorn import *
from unicorn.x86_const import *

import struct

def hook_mem_access(uc, access, address, size, value, user_data):
    print(f"[*] Memory access: {access:x} at 0x{address:x}, data size = {size}, data value = 0x{value:x}")

def hook_code(uc, address, size, user_data):
    print(f"[*] Current RIP: {address:x}, instruction size = {size}")
```

- Create a virtual environment and install the Python bindings.
- Import the necessary methods.
- Set up the hooking functions.

Python Code Example: Configuring the Engine

Listing 2: Unicorn Engine Example with Hooks

```
1  with open("sample", "rb") as f:
2      binary = f.read()
3
4  ADDRESS = 0x1000000
5
6  uc = Uc(UC_ARCH_X86, UC_MODE_64)
7  uc.hook_add(UC_HOOK_MEM_WRITE, hook_mem_access)
8  uc.hook_add(UC_HOOK_MEM_READ, hook_mem_access)
9  uc.hook_add(UC_HOOK_CODE, hook_code, None, ADDRESS + 0x1189,
       ADDRESS + 0x12AC)
10 uc.mem_map(ADDRESS, 2 * 1024 * 1024)  # 2 MB
11 uc.mem_write(ADDRESS, binary)
```

- Define the CPU architecture (line 6)
- Install the hooks (line 7 to 9)
- Configure memory layout (line 10)
- Load the ELF file (line 11)

Function Parameter Passing

Listing 3: Unicorn Engine Example: Setting Arguments

```
input_str = b".''$gk$9'/j"
input_key = 75 # 'K'

# Write the input string to memory
uc.mem_write(ADDRESS + 0x4000, input_str)  # Address where input_str is stored (binary 16K, string at 17K)

# Set up registers
uc.reg_write(UC_X86_REG_RDI, ADDRESS + 0x4000)  # Set the first argument (address of the string)
uc.reg_write(UC_X86_REG_RSI, input_key)        # Set the second argument (offset)
uc.reg_write(UC_X86_REG_RSP, ADDRESS + 0x6000) # Set the stack (RSP)
```

Pay attention to the operating system's calling convention.

Python Code Example: Emulating with Unicorn

Listing 4: Unicorn Engine Emulation Example

```python
try:
    # Start emulation from the specified range
    uc.emu_start(ADDRESS + 0x1189, ADDRESS + 0x11EC)  # Start and end addresses recovered from Ghidra

    # Read the result from memory and decode it
    result = uc.mem_read(ADDRESS + 0x4000, len(input_str)).decode("utf-8")
    print(f"Encoded string: {result}")

except UcError as e:
    # Handle errors during emulation
    print(f"Unicorn error: {e}")
```

Unicorn Engine Trouble Shooting

```
[*] Current RIP: 1001189,
    instruction size = 4
[*] Current RIP: 100118d,
    instruction size = 1
[*] Memory access: 11 at 0
    x1005ff8, data size = 8,
    data value = 0x0
[*] Current RIP: 100118e,
    instruction size = 3
[*] Current RIP: 1001191,
    instruction size = 1
[*] Memory access: 11 at 0
    x1005ff0, data size = 8,
    data value = 0x0
[*] Current RIP: 1001192,
    instruction size = 4
[*] Current RIP: 1001196,
    instruction size = 4
[*] Memory access: 11 at 0
    x1005fd0, data size = 8,
    data value = 0x1004000
```

```
1 00101189 f3 0f 1e fa    ENDBR64
2 0010118d 55             PUSH    RBP
3 0010118e 48 89 e5       MOV     RBP,RSP
  00101191 53             PUSH    RBX
4 00101192 48 83 ec 28    SUB     RSP,0x28
  00101196 48 89 7d d8    MOV     qword ptr [RE
  0010119a 89 f0          MOV     EAX,ESI
  0010119c 88 45 d4       MOV     byte ptr [RBF
  0010119f c7 45 ec       MOV     dword ptr [RE
           00 00 00 00
  001011a6 eb 26          JMP     LAB_001011ce
```

References and Outlook

- **Malware Samples Used:** https://helga.circl.lu/NGSOTI/malware-dataset
- **AIL Training:** 4th February at 122 Rue Adolphe Fischer, 1521 Luxembourg
- **Registration Link:** https://pretix.eu/circl/fkq78/
- **Note:** Subject to a vetting process.
- **Join the MISP-LEA Initiative:**
- Training material for LEA
  - https://github.com/neolea
  - https://github.com/MISP/misp-training-lea

  Contact us at info@misp-lea.org.