Walkthrough

This file contains the walkthrough for the challenge `Infinite Free Conference Tickets`.

Part I

This challenge focuses heavily on obfuscation. It employs a wide variety of classic forms of junk code. For instance, this is one of the junk code patterns used in the challenge:

```
db 0xEB, 0xFF, 0xC0
```

When the CPU executes these instructions, it first reaches `0xEB` which is a two-byte jump short command. It jumps by `0xFF` which is -1 byte from the end of the instruction, so the RIP register lands before `0xFF`. It then interprets `0xFF, 0xC0` as another two-byte instruction and performs `inc eax`. Linear disassemblers might correctly interpret `0xEB, 0xFF` as `jmp -1`, then incorrectly start with `0xC1` instead of `0xFF` as the next opcode. This results in the following bytes being interpreted incorrectly and either produces incorrect disassembly or get displayed as raw data because the disassembler cannot understand the following bytes.

In practice, such junk code will result in linear disassemblers producing incorrect results (e.g., IDA Pro). Non-linear disassemblers like Radare2 and Binary Ninja do not suffer the same issues as much. Disassembly after junk code may be incorrect, or represented as raw data because the disassembler cannot interpret the bytes.

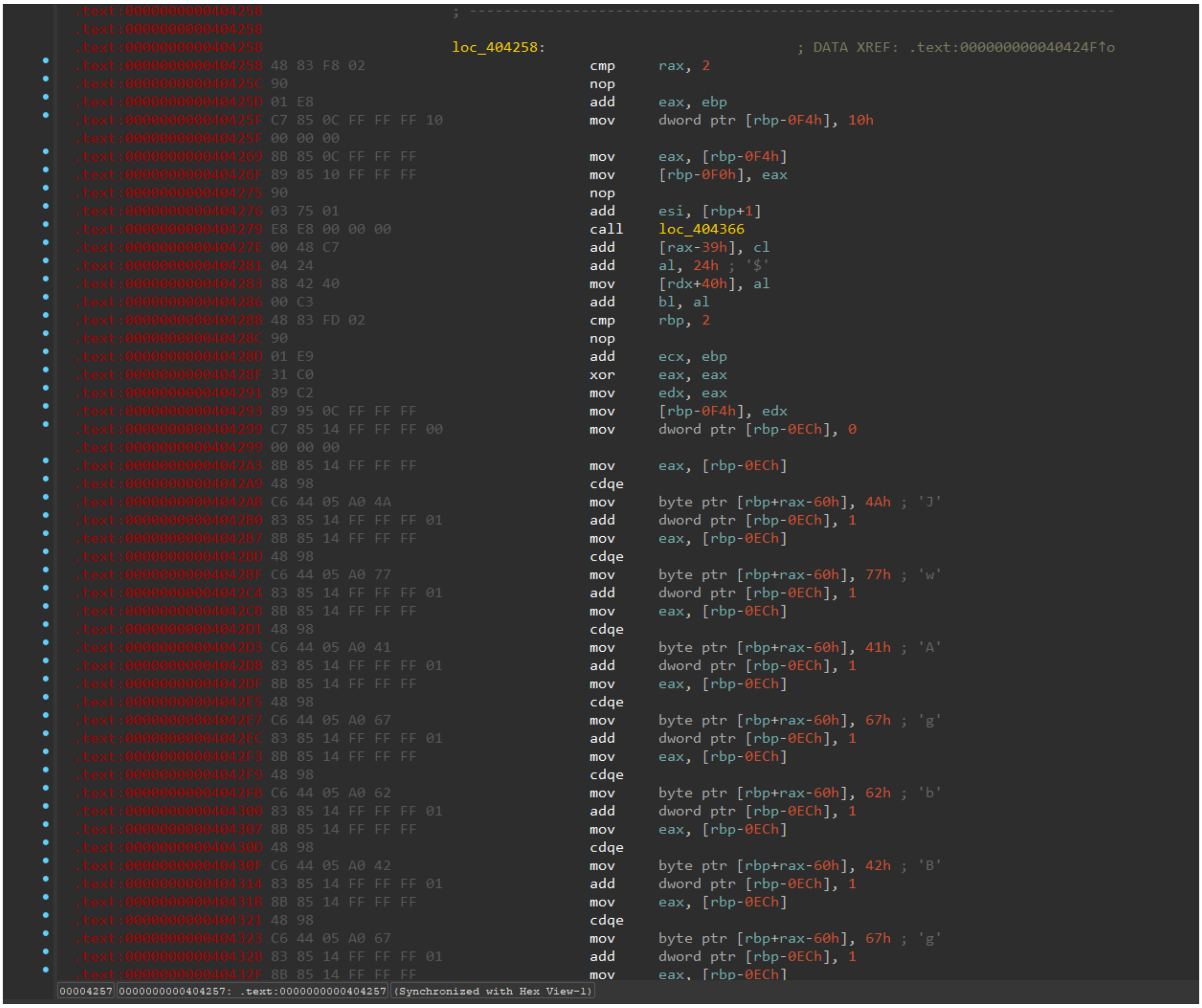
![[Pasted image 20240615143936.png]]

Junk codes like these are spread out throughout the entire file. They make the disassembly unreadable. Participants need to successfully identify these obfuscation patterns and make use of IDAPython or other similar automated tools to NOP-patch these junk codes in bulk to make the file readable. For instance, the following IDAPython script can be used to patch the `0xEB, 0xFF, 0xC0` pattern:

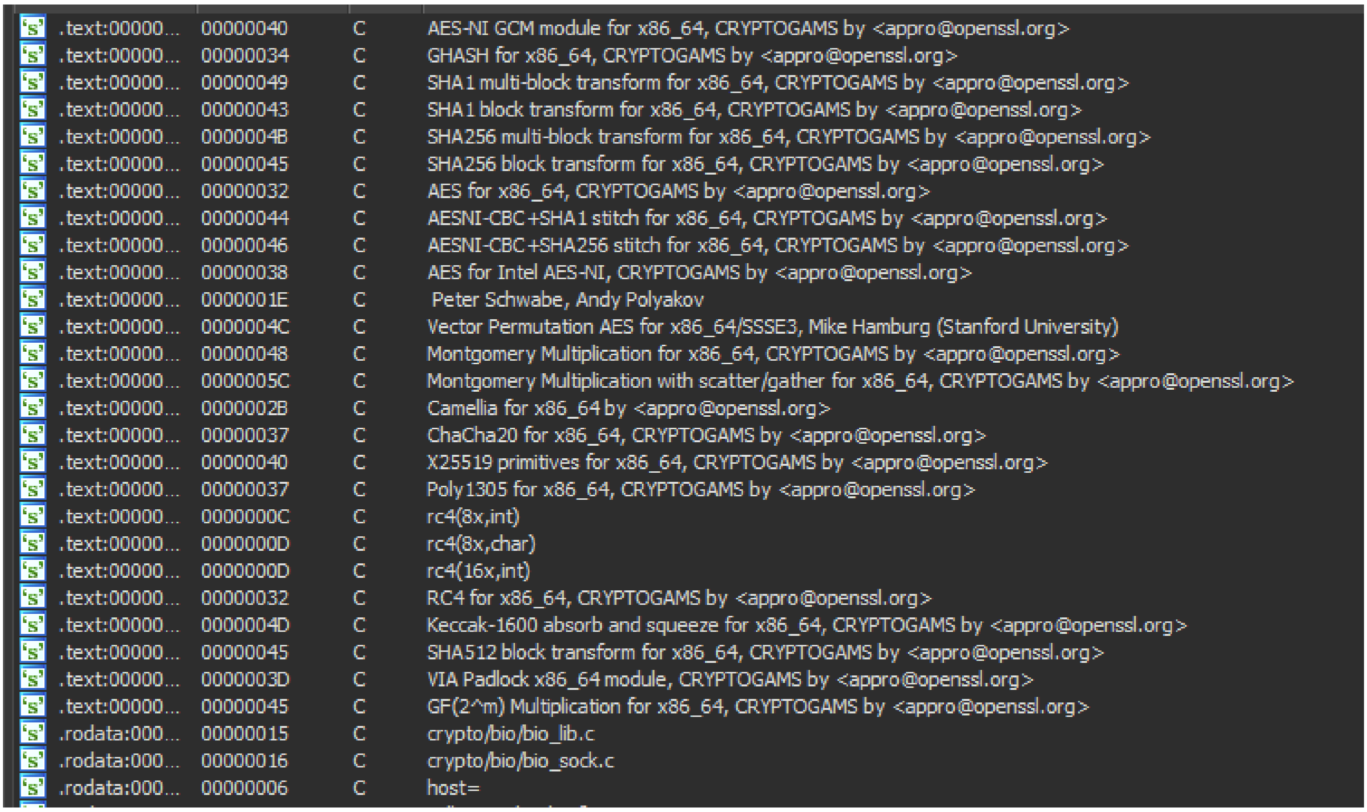
```
#!/usr/bin/python
# -*- utf-8 -*-
from idaapi import *
from idautils import *
from idc import *

def nops_out_junk_bytes():
    # Iterate through all segments
    for seg in Segments():
        seg_start = get_segm_start(seg)
        seg_end = get_segm_end(seg)

        ea = seg_start
        while ea < seg_end:
            # Read 3 bytes from the current address
            bytes = get_bytes(ea, 3)

            if bytes is None or len(bytes) < 3:
```

```
                ea += 1
                continue

            # If the bytes match 0xEB 0xFF 0xC0
            # Replace 0xEB with 0x90 (NOP)
            if bytes[0] == 0xEB and bytes[1] == 0xFF and bytes[2] == 0xC0:
                patch_byte(ea, 0x90)

            ea += 1

nops_out_junk_bytes()
```

With automated and manual junk code patching, the program's bytes can be gradually converted into normal readable disassembly:



From this point onward, the participants can read the assembly or, optionally, patch the program to a state where IDA can correctly identify the subroutine boundaries and run decompilers on the functions. The latter could be difficult. The participants will find that this program contains strings hinting that OpenSSL is statically linked, indicating that one of the algorithms in OpenSSL may be used to sign the token.



.text:00000...	00000040	C	AES-NI GCM module for x86_64, CRYPTOGAMS by <appro@openssl.org>
.text:00000...	00000034	C	GHASH for x86_64, CRYPTOGAMS by <appro@openssl.org>
.text:00000...	00000049	C	SHA1 multi-block transform for x86_64, CRYPTOGAMS by <appro@openssl.org>
.text:00000...	00000043	C	SHA1 block transform for x86_64, CRYPTOGAMS by <appro@openssl.org>
.text:00000...	0000004B	C	SHA256 multi-block transform for x86_64, CRYPTOGAMS by <appro@openssl.org>
.text:00000...	00000045	C	SHA256 block transform for x86_64, CRYPTOGAMS by <appro@openssl.org>
.text:00000...	00000032	C	AES for x86_64, CRYPTOGAMS by <appro@openssl.org>
.text:00000...	00000044	C	AESNI-CBC+SHA1 stitch for x86_64, CRYPTOGAMS by <appro@openssl.org>
.text:00000...	00000046	C	AESNI-CBC+SHA256 stitch for x86_64, CRYPTOGAMS by <appro@openssl.org>
.text:00000...	00000038	C	AES for Intel AES-NI, CRYPTOGAMS by <appro@openssl.org>
.text:00000...	0000001E	C	Peter Schwabe, Andy Polyakov
.text:00000...	0000004C	C	Vector Permutation AES for x86_64/SSSE3, Mike Hamburg (Stanford University)
.text:00000...	00000048	C	Montgomery Multiplication for x86_64, CRYPTOGAMS by <appro@openssl.org>
.text:00000...	0000005C	C	Montgomery Multiplication with scatter/gather for x86_64, CRYPTOGAMS by <appro@openssl.org>
.text:00000...	0000002B	C	Camellia for x86_64 by <appro@openssl.org>
.text:00000...	00000037	C	ChaCha20 for x86_64, CRYPTOGAMS by <appro@openssl.org>
.text:00000...	00000040	C	X25519 primitives for x86_64, CRYPTOGAMS by <appro@openssl.org>
.text:00000...	00000037	C	Poly1305 for x86_64, CRYPTOGAMS by <appro@openssl.org>
.text:00000...	0000000C	C	rc4(8x,int)
.text:00000...	0000000D	C	rc4(8x,char)
.text:00000...	0000000D	C	rc4(16x,int)
.text:00000...	00000032	C	RC4 for x86_64, CRYPTOGAMS by <appro@openssl.org>
.text:00000...	0000004D	C	Keccak-1600 absorb and squeeze for x86_64, CRYPTOGAMS by <appro@openssl.org>
.text:00000...	00000045	C	SHA512 block transform for x86_64, CRYPTOGAMS by <appro@openssl.org>
.text:00000...	0000003D	C	VIA Padlock x86_64 module, CRYPTOGAMS by <appro@openssl.org>
.text:00000...	00000045	C	GF(2^m) Multiplication for x86_64, CRYPTOGAMS by <appro@openssl.org>
.rodata:000...	00000015	C	crypto/bio/bio_lib.c
.rodata:000...	00000016	C	crypto/bio/bio_sock.c
.rodata:000...	00000006	C	host=

From here, it may be helpful for the participants to use a debugger to step through the program to figure out the contents of the concatenated strings and decrypted keys in the memory. This program uses AES-256-CBC to encrypt the SHA-256 hash of the user's name. Participants can catch the IV and AES key used when the function `EVP_DecryptInit_ex` is called. The IV and the AES encryption key are stored in the same chunk of memory without null byte separation. They are passed to the `EVP_DecryptInit_ex` with offsets pointing to different portions of the memory chunk:

```
EVP_DecryptInit_ex(ctx, (const EVP_CIPHER *)cipher, NULL, MESSAGE_XOR_KEY +
26, MESSAGE_XOR_KEY);
```

The first half of the memory will get decrypted before the AES key and will be null-terminated. It will be used as a XOR decryption key to decrypt the Base64-encoded text strings. Once the program gets to the AES encryption part, the AES key will be appended after the message decryption key, with its first byte overwriting the message decryption key's last byte, the null byte. OpenSSL will take the first 16 bytes starting from the address of `MESSAGE_XOR_KEY` as the IV, and 32 bytes starting from the address of `MESSAGE_XOR_KEY + 26` as the AES encryption key. The string at the address of `MESSAGE_XOR_KEY + 26` is the flag for the first part of the challenge, which would be `flag{ReADiNg_AsM_aiNt_thAT_HarD}`.

It is also worth mentioning that this program also has countermeasures against dynamic analysis (debuggers). It achieves this through sabotaging the program's stack if `ptrace(PTRACE_TRACEME);` returns `-1`, indicating that a debugger is already attached. Attempting to debug the program without patching this check will result in a confusing segfault happening later in the program:

```
0x00007fffffffd368│+0x0000: 0x0000000000434c10  →   sub rsp, 0x8          ← $rsp
0x00007fffffffd370│+0x0008: 0x000000000041e620  →   jmp 0x41e600
0x00007fffffffd378│+0x0010: 0x0000000000918840  →  0x00000000009238f0  →  0x0000000000000000
0x00007fffffffd380│+0x0018: 0x00000000007313f5  →   xor r10d, r10d
0x00007fffffffd388│+0x0020: 0x00000000007312b0  →   endbr64
0x00007fffffffd390│+0x0028: 0x00000000009189c0  →   add al, BYTE PTR [rax]
0x00007fffffffd398│+0x0030: 0x0000000000000001
0x00007fffffffd3a0│+0x0038: 0x0000000000731270  →   endbr64

     0x4202be                  punpcklqdq xmm0, xmm3
     0x4202c2                  movq   xmm5, rax
     0x4202c7                  punpcklqdq xmm1, xmm4
 →   0x4202cb                  movaps XMMWORD PTR [rsp+0x10], xmm0
     0x4202d0                  punpcklqdq xmm2, xmm5
     0x4202d4                  movaps XMMWORD PTR [rsp+0x20], xmm1
     0x4202d9                  movaps XMMWORD PTR [rsp+0x30], xmm2
     0x4202de                  call   0x4325f0
     0x4202e3                  mov    rdi, QWORD PTR [rsp+0xa0]

[#0] Id 1, Name: "validate", stopped 0x4202cb in ?? (), reason: SIGSEGV

[#0] 0x4202cb → movaps XMMWORD PTR [rsp+0x10], xmm0
[#1] 0x41e5fa → add rsp, 0x18
[#2] 0x41f26f → test rax, rax
[#3] 0x41f79b → add rsp, 0x18
[#4] 0x403dd9 → cmp eax, 0x1
[#5] 0x40578b → mov DWORD PTR [rbp-0xe8], eax
[#6] 0x70dd98 → mov edi, eax
[#7] 0x70fe20 → call 0x76c4a0
[#8] 0x402c65 → hlt

gef➤  bt
#0  0x00000000004202cb in ?? ()
#1  0x000000000041e5fa in ?? ()
#2  0x000000000041f26f in ?? ()
#3  0x000000000041f79b in ?? ()
#4  0x0000000000403dd9 in ?? ()
#5  0x000000000040578b in ?? ()
#6  0x000000000070dd98 in ?? ()
#7  0x000000000070fe20 in ?? ()
#8  0x0000000000402c65 in ?? ()
gef➤
```

The `ptrace` call is obfuscated and called via a x86 syscall instead of a direct `ptrace` call. The syscall ID is calculated dynamically during the program's execution. The participants will need to patch these checks in order to be able to debug the program.

```
// manually do a ptrace syscall and write result to ptrace_result
// using x86 `int 0x80` to perform the syscall to make it slightly less obvious
__asm__ volatile(
    "movl %3, %%edx\n"
    "movl %1, %%ebx\n"
    "movl %0, %%eax\n"
    "movl %2, %%ecx\n"
    "int $0x80\n"
    "cmp %%ecx, %%eax\n"
    "jge 0f\n"
    // compiles to a single byte 0x58
```

```
    // will crash the program in a later function calls, likely within
OpenSSH
    "pop %%rax\n"
    "0:\n"
    :
    : "r"(syscall_id), "r"(ptrace_request), "r"(ptrace_request), "r"
((int)message[index])
    : "eax", "ebx", "ecx", "edx"
);
```

Part II

In this part of the challenge, the participant will need to analyze the file to uncover the algorithm used to check valid tokens. The algorithm is as follows:

- Let `AESDecrypt(T, K, IV)` represent the AES-256-CBC decryption of the token `T` using the key `K` and initialization vector `IV`.
- Let `SHA256(U)` represent the SHA-256 hash of the username `U`.

The algorithm checks if the decryption of the token is equal to the hash of the username:

```
AESDecrypt(T, K, IV) = SHA256(U)
```

Thus, the mathematical formula for the validation is:

```
Valid ⇔ AESDecrypt(T, K, IV) = SHA256(U)
```

If this equation holds true, the token is valid; otherwise, it is invalid. The participants can fill in the reversed algorithm in the provided `solver_template.py` file to solve the challenge. The algorithm will be:

- Let `AESEncrypt(P, K, IV)` represent the AES-256-CBC encryption of the plaintext `P` using the key `K` and initialization vector `IV`.
- Let `SHA256(U)` represent the SHA-256 hash of the username `U`.

```
token = AESEncrypt(SHA256(U), K, IV)
```

The completed `solver.py` will look like:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import base64
import hashlib
import os
```

```python
import requests
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives.padding import PKCS7

KEY = b"flag{ReADiNg_AsM_aiNt_thAT_HarD}"
IV = b"shELlnEverDaNCEwiThUsagAIN"[:16]
SERVER_BASE_URL = os.environ.get("SERVER_BASE_URL", "http://127.0.0.1:8080")


def generate_token(name: str) -> str:
    """
    Generate the token for the given name.

    :param name: the name of the participant
    :return: the token for the name
    """

    # use AES-256 in CBC mode
    cipher = Cipher(algorithms.AES(KEY), modes.CBC(IV), backend=default_backend())
    encryptor = cipher.encryptor()

    # hash the name with SHA-256
    name_hash = hashlib.sha256(name.encode("utf-8")).digest()

    # pad the name with PKCS#7
    padder = PKCS7(128).padder()
    padded_name_hash = padder.update(name_hash) + padder.finalize()

    # encrypt the padded name hash
    encrypted_hash = encryptor.update(padded_name_hash) + encryptor.finalize()

    # encode the encrypted hash with base64
    encoded_encrypted_hash = base64.b64encode(encrypted_hash).decode()

    # concatenate the base64-encoded name and the base64-encoded encrypted hash
    token = "{}.{}".format(
        base64.b64encode(name.encode("utf-8")).decode(), encoded_encrypted_hash
    )

    return token

# The rest of the file is omitted
```

The participant can then run the solver script against the challenge server to obtain the flag for part II of this challenge.