



c:\users\dilbert\edk2\UefiCpuPkg\Library\BaseXApicX2ApicLib\BaseXApicX2ApicLib.c. These paths reference the OVMF project (<https://github.com/tianocore/tianocore.github.io/wiki/OVMF>), which is a popular open-source implementation of UEFI firmware. This confirms that bios.bin is indeed a UEFI firmware volume.

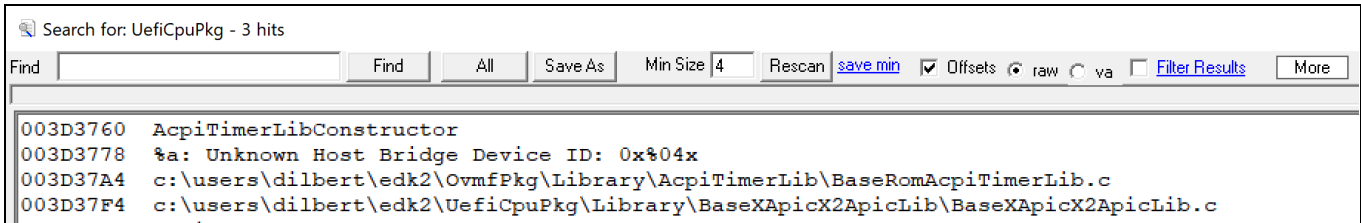


Figure 2: Strings hit for UefiCpuPkg

The other file, disk.img, is more straightforward to identify. Its hexadecimal data reveals the string FAT12 and the initial bytes EB 3F 90, which are characteristic of a FAT12-formatted disk image (<https://www.win.tue.nl/~aeb/linux/fs/fat/fat-1.html>).

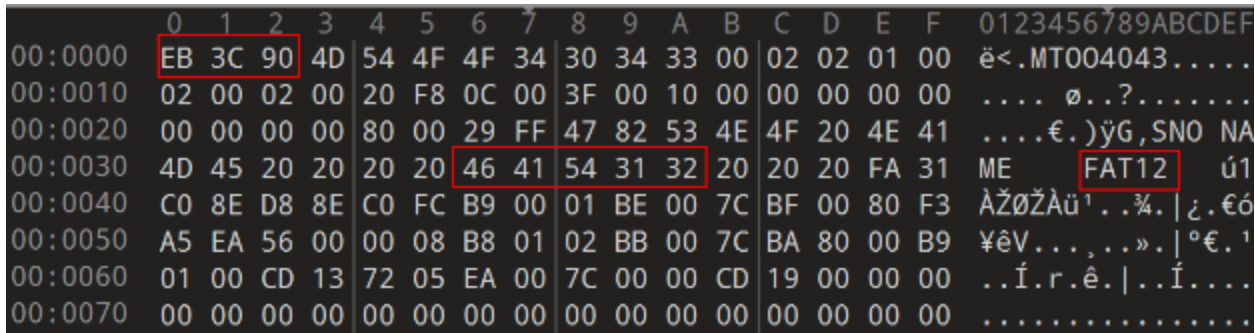


Figure 3: Disk image showing FAT12 signature

To verify this, we can open the FAT12 image using a file compression tool like 7-Zip. Inside, we find several files:

- catmeme1.jpg.c4tb
- catmeme2.jpg.c4tb
- catmeme3.jpg.c4tb
- DilbootApp.efi.enc

All these files appear to be encrypted, as evident from their extensions and their high entropy contents. We should also make note of the proprietary format they have which is identified by the magic string C4TB, that is part of a header appearing at the beginning of each of the first three files, that is then followed by the seemingly encrypted blob.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0:0000	43	34	54	42	54	15	01	00	70	15	01	00	59	02	00	00	C4TBT...p...Y...
0:0010	8C	F7	B3	F3	2A	C7	A5	44	72	4E	75	A6	38	D4	D9	EF	€÷³ó*Ç¥DrNu!8ÔUi
0:0020	EC	01	D0	15	79	58	F0	C6	BB	80	97	E0	82	60	2E	35	ì.Đ.yXðÆ»€-à,`.5
0:0030	F4	45	1C	1A	D7	72	80	49	6E	08	2C	F2	B9	A9	97	1D	ôE..xrcIn.,ò'©-
0:0040	01	36	CF	28	FE	9E	D1	89	7B	9B	0D	BD	23	70	55	39	.6İ(þžÑ%{>.½#pU9
0:0050	C1	77	DB	59	29	FA	BF	C4	FC	DB	61	DF	AF	DE	3B	63	ÁwÛY)ú¿ÄüÛaß`þ;c
0:0060	0C	B6	C9	0B	58	81	A4	A5	44	23	04	CA	6D	AE	92	76	.¶É.X.¤¥D#.Êm@'v
0:0070	AF	4C	08	2E	47	1B	7E	02	CE	7C	44	3C	E5	94	72	C0	┌L..G.-.Î D<ä"rÀ
0:0080	7A	B3	15	C0	AC	7C	44	33	8D	2B	DA	7F	DC	D4	B7	C9	z³.Å- D3.+Ú.ÛÔ·É
0:0090	70	FF	AB	FF	2B	B5	1E	21	DE	D8	81	36	DE	19	C7	2A	pÿ«ÿ+µ. !þø.6þ.Ç*

Figure 4: C4TB header example

## Basic Dynamic Analysis

Building upon our static analysis, we identified the `bios.bin` file as an OVMF UEFI firmware image designed for QEMU and KVM virtualization environments. For further analysis, we can run this image within QEMU by following these instructions: <https://github.com/tianocore/tianocore.github.io/wiki/How-to-run-OVME>. Remember to mount the `disk.img` file as the virtual disk during this process. Here's an example command for an Ubuntu system with QEMU installed:

Unset

```
qemu-system-x86_64 -drive format=raw,file=disk.img -biosbios.bin
```

Executing this command launches the virtual machine, presenting a startup logo:

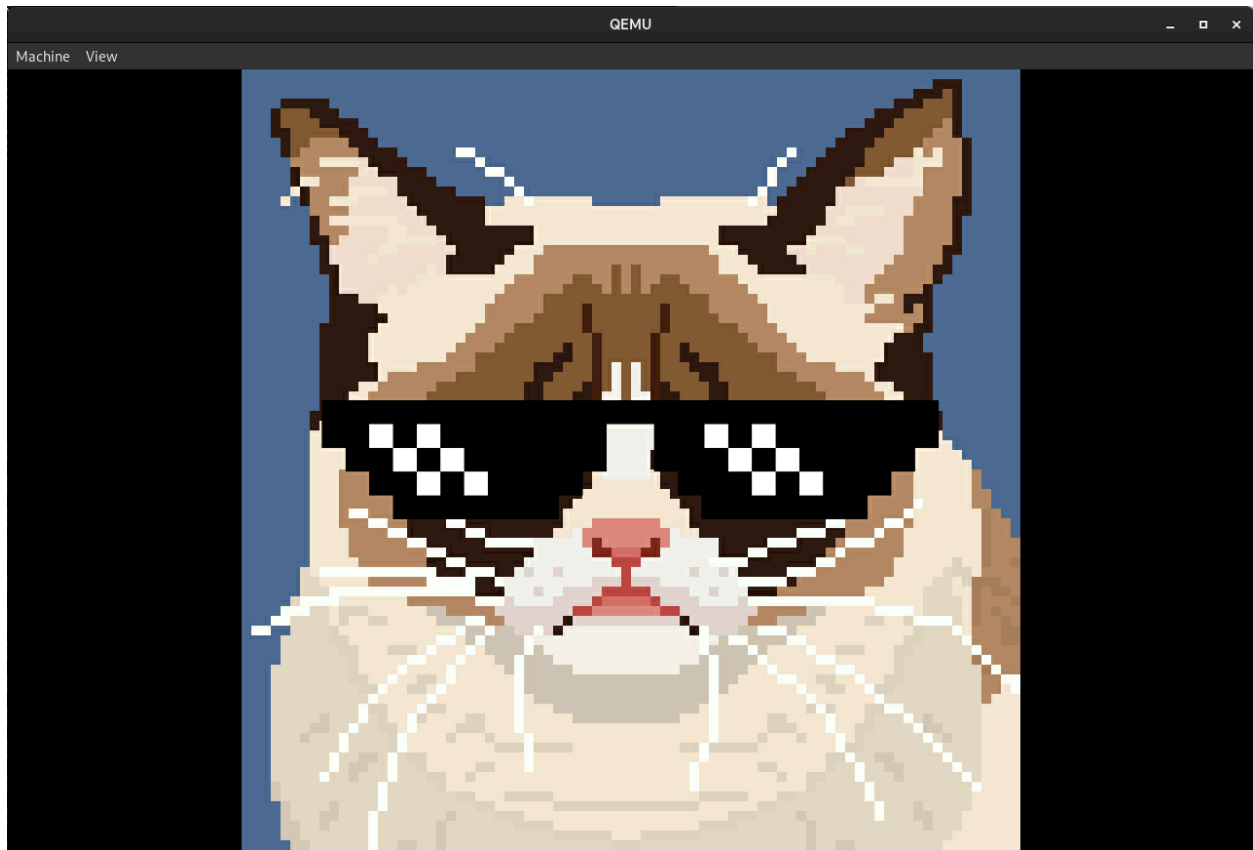


Figure 5: CATBERT Startup Logo

Following the logo, we encounter a modified EFI shell. This shell displays a ransom message and instructions (use PgUp and PgDown to navigate the text):

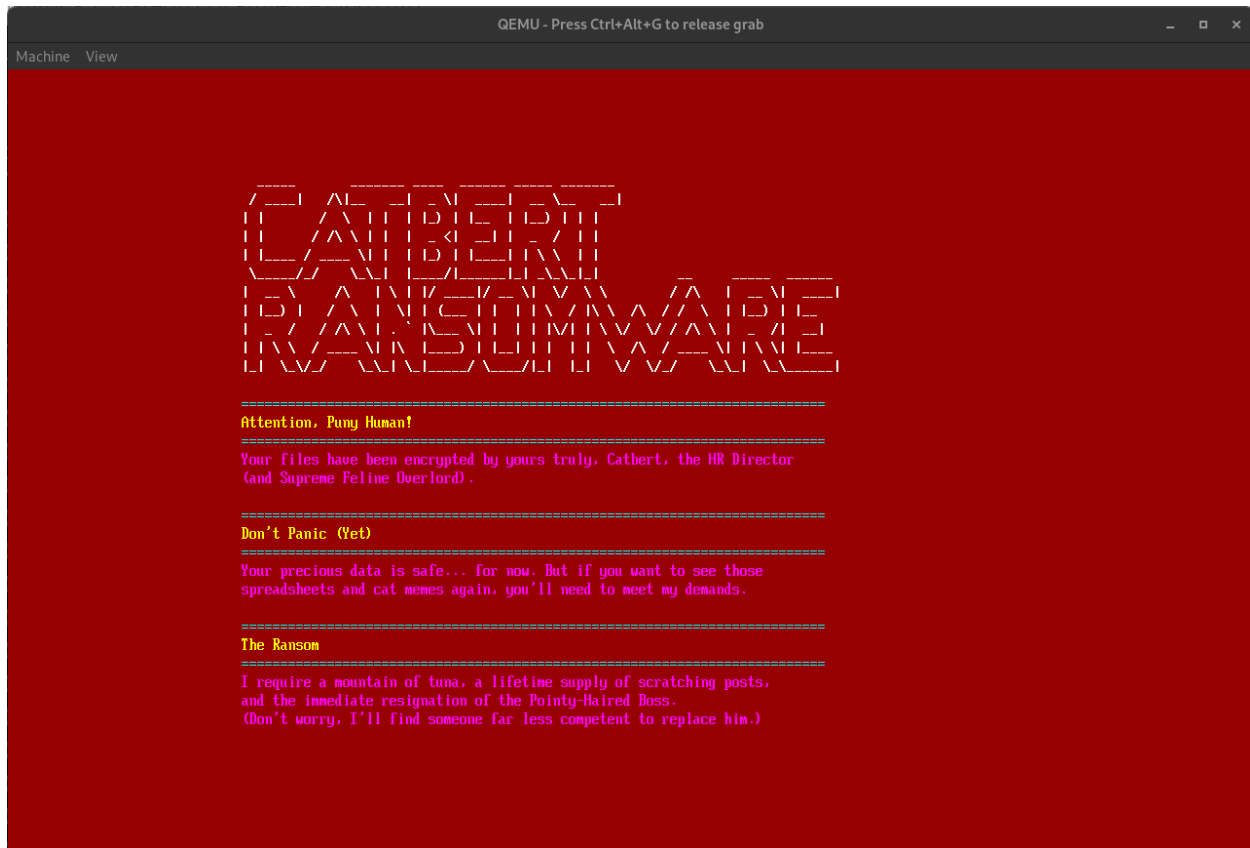
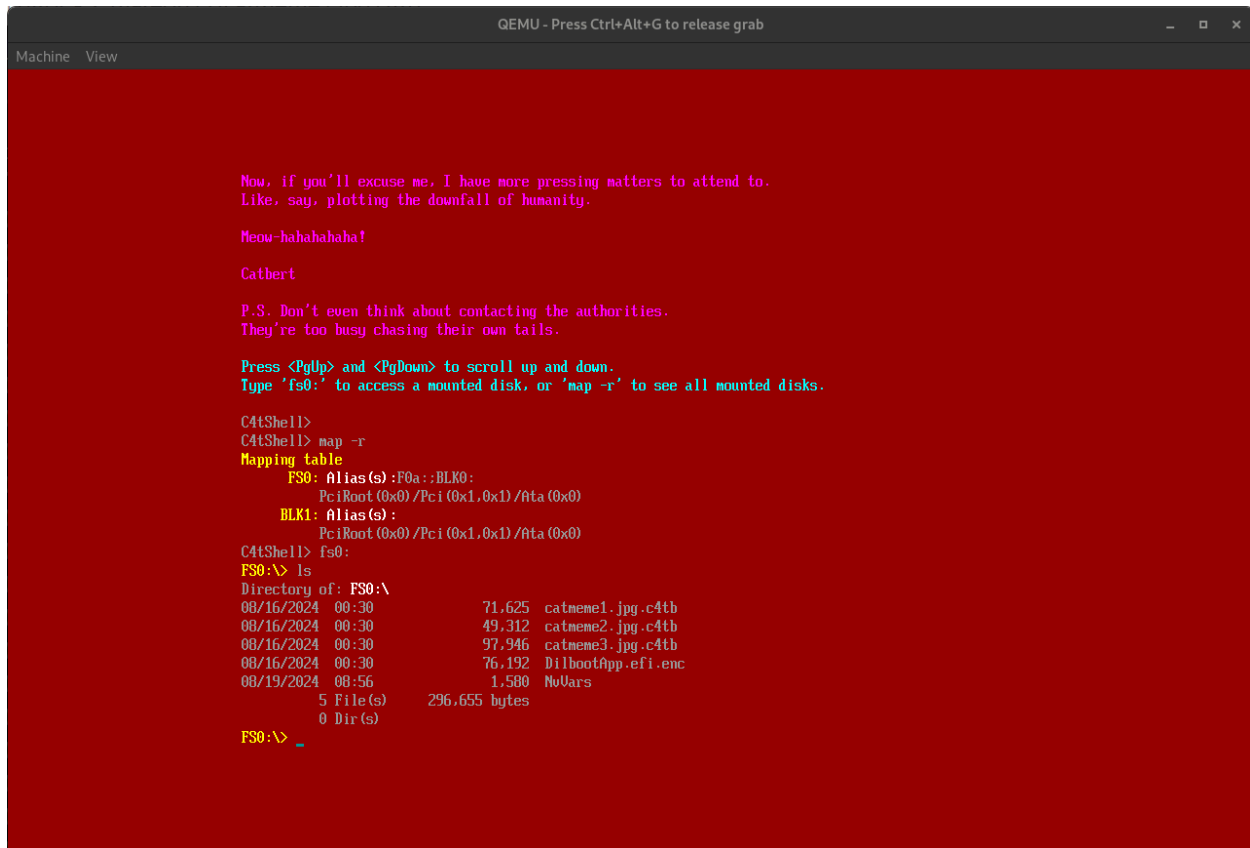


Figure 6: CATBERT Ransome Message

Based on the message, we can identify the mounted disk image volume by running `map -r`. We can then enter this volume with a command like `fs0:.` Listing the files using `ls` reveals the same files identified previously using 7-Zip:



```
QEMU - Press Ctrl+Alt+G to release grab
Machine View

Now, if you'll excuse me, I have more pressing matters to attend to.
Like, say, plotting the downfall of humanity.

Meow-hahahahaha!

Catbert

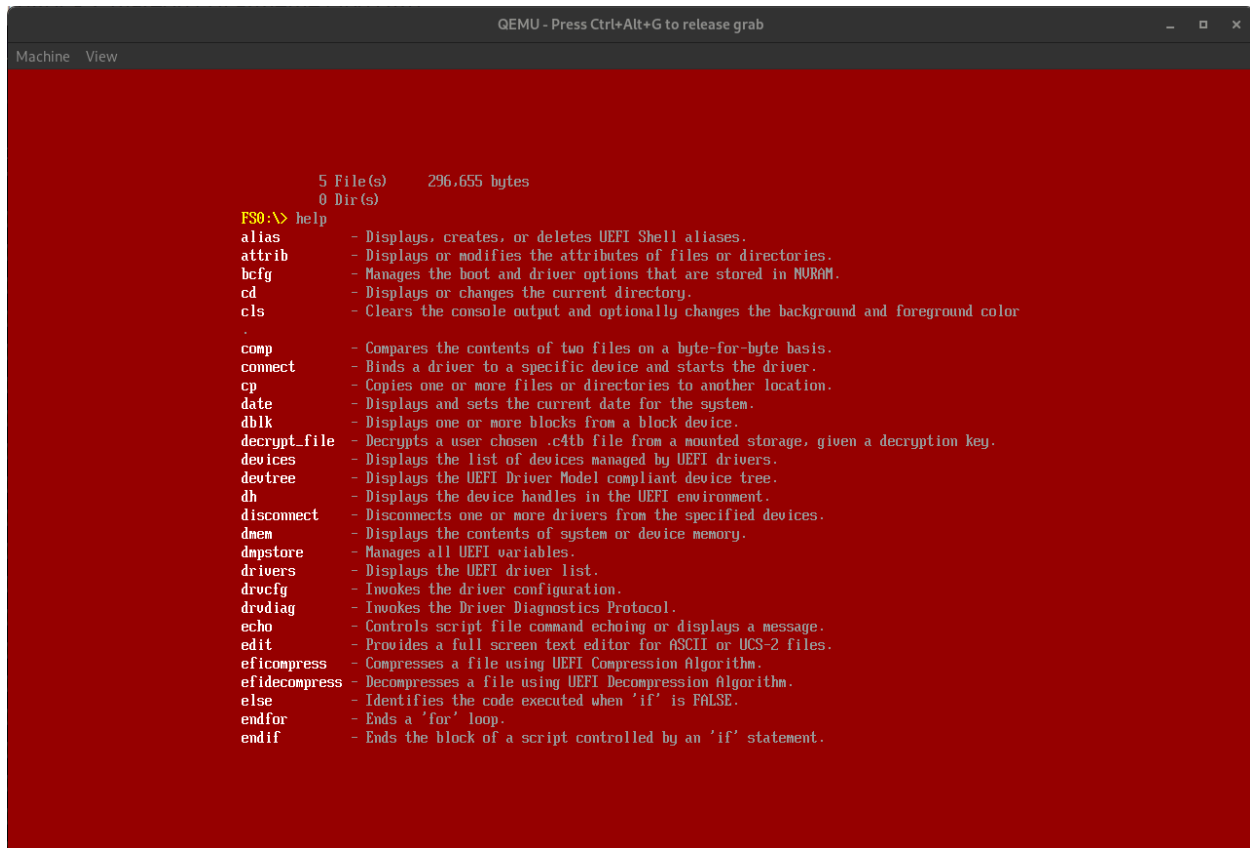
P.S. Don't even think about contacting the authorities.
They're too busy chasing their own tails.

Press <PgUp> and <PgDown> to scroll up and down.
Type 'fs0:' to access a mounted disk, or 'map -r' to see all mounted disks.

C4tShell>
C4tShell> map -r
Mapping table
  FS0: Alias(s):F0a:;BLK0:
        PciRoot(0x0)/Pci(0x1,0x1)/Pta(0x0)
  BLK1: Alias(s):
        PciRoot(0x0)/Pci(0x1,0x1)/Pta(0x0)
C4tShell> fs0:
FS0:\> ls
Directory of: FS0:\
08/16/2024 00:30          71,625  catmeme1.jpg.c4tb
08/16/2024 00:30          49,312  catmeme2.jpg.c4tb
08/16/2024 00:30          97,946  catmeme3.jpg.c4tb
08/16/2024 00:30          76,192  DilbootApp.efi.enc
08/19/2024 08:56           1,580  NoVars
          5 File(s)    296,655 bytes
          0 Dir(s)
FS0:\> _
```

Figure 8: Running map and ls in CATBERT shell

To explore available commands further, type `help`. One intriguing command discovered through this exploration is `decrypt_file`.



```
Machine View
QEMU - Press Ctrl+Alt+G to release grab

5 File(s) 296,655 bytes
0 Dir(s)
FS0:\> help
alias - Displays, creates, or deletes UEFI Shell aliases.
attrib - Displays or modifies the attributes of files or directories.
bcfg - Manages the boot and driver options that are stored in NVRAM.
cd - Displays or changes the current directory.
cls - Clears the console output and optionally changes the background and foreground color.
.
comp - Compares the contents of two files on a byte-for-byte basis.
connect - Binds a driver to a specific device and starts the driver.
cp - Copies one or more files or directories to another location.
date - Displays and sets the current date for the system.
dblk - Displays one or more blocks from a block device.
decrypt_file - Decrypts a user chosen .c4tb file from a mounted storage, given a decryption key.
devices - Displays the list of devices managed by UEFI drivers.
devtree - Displays the UEFI Driver Model compliant device tree.
dh - Displays the device handles in the UEFI environment.
disconnect - Disconnects one or more drivers from the specified devices.
dmem - Displays the contents of system or device memory.
dmpstore - Manages all UEFI variables.
drivers - Displays the UEFI driver list.
drvcfg - Invokes the driver configuration.
drvdiag - Invokes the Driver Diagnostics Protocol.
echo - Controls script file command echoing or displays a message.
edit - Provides a full screen text editor for ASCII or UCS-2 files.
efidecompress - Compresses a file using UEFI Compression Algorithm.
efidecompress - Decompresses a file using UEFI Decompression Algorithm.
else - Identifies the code executed when 'if' is FALSE.
endfor - Ends a 'for' loop.
endif - Ends the block of a script controlled by an 'if' statement.
```

Figure 9: CATBERT help screen

Based on its description, it requires a filename and password (which we currently don't know). Next, let's experiment with `decrypt_file` on an arbitrary file within the shell just to observe the output:





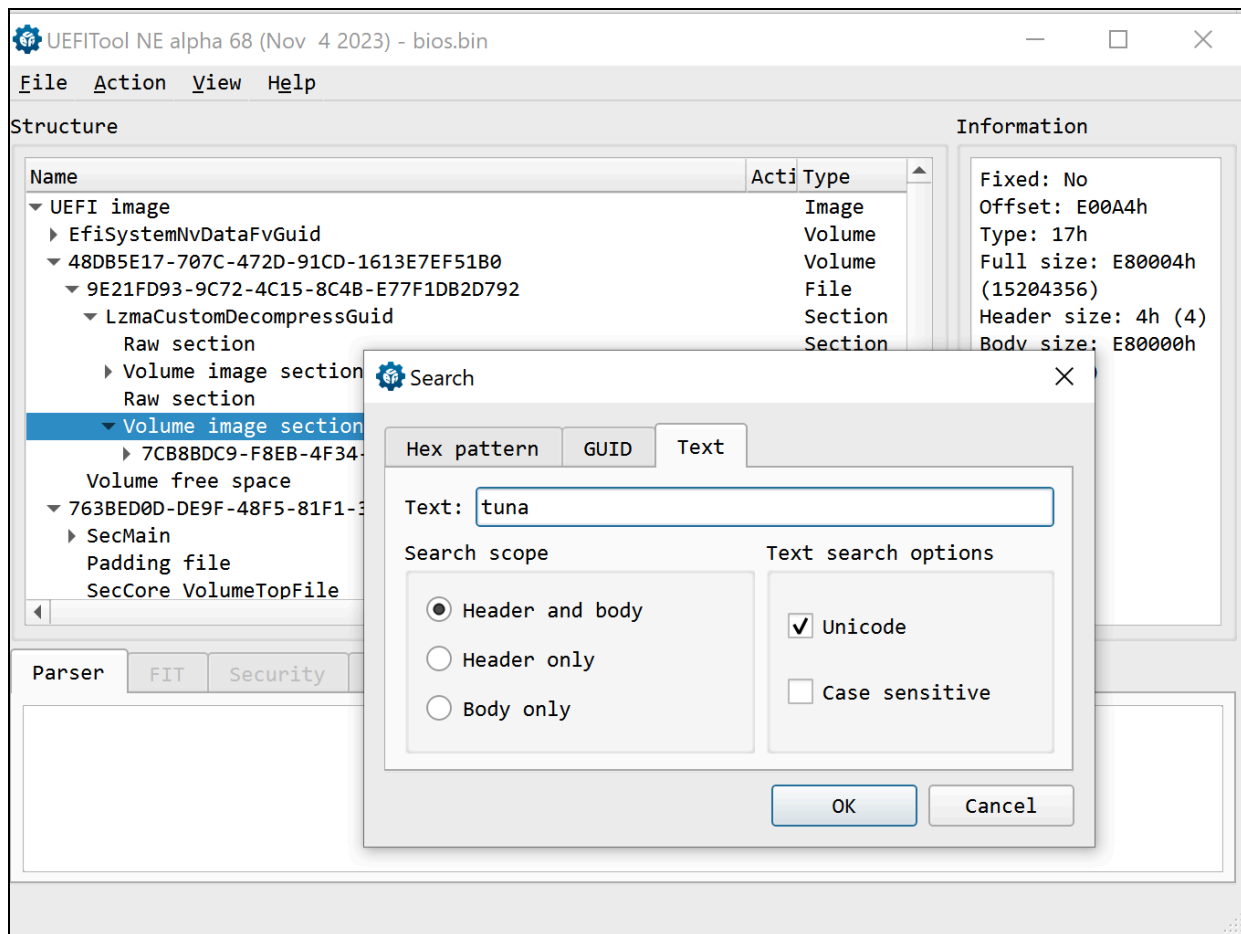


Figure 11: UEFITool

This search should lead us to the `FullShell` module. We can then extract this module by right-clicking the `PE32 Image Section` entry and selecting `Extract Body`. This will yield the underlying Portable Executable (PE) file, accessible for analysis in IDA.

Given that the code for the EFI shell is publicly available and our compiled image contains a lot of debug strings, we have various paths to further our analysis. For example, we can compile a version of the shell with symbols and diff it against our module. Alternatively, we can examine the source code directly and use the debug prints to find the associated code lines in the EDKII repository. Namely, we can look at the code for `ShellPkg` (<https://github.com/tianocore/edk2/blob/master/ShellPkg/Application/Shell/Shell.c>) and shell commands (<https://github.com/tianocore/edk2/tree/master/ShellPkg/Library/UefiShellLevel3CommandsLib>) and try to map functions from it to functions in our modified shell module.

The following function names, identified within the analysis, provide valuable insights into the shell's behavior and will help us better understand the code moving forward:

- `sub_16578 (ShellCommandLineParseEx)`: Parses command-line arguments, providing information about the input commands and their parameters.

- `sub_170FC` (**ShellPrintHiEx**): Handles the printing of messages to the console, potentially displaying error messages or status updates.
- `sub_168D4` (**ShellCommandLineGetRawValue**): Retrieves raw values from the command line, allowing for direct access to user-provided input.
- `sub_13050` (**AllocatePool**): Allocates memory for dynamic data structures, such as buffers or strings.
- `sub_FDD0` (**StrCpyS**): Copies strings, ensuring proper handling of buffer boundaries.
- `sub_17CC0` (**ShellFileExists**): Checks for the existence of files on a given volume, used for file-related operations.
- `sub_15038` (**ShellOpenFileByName**): Opens files based on their names, enabling access to read, write or create new files.
- `sub_170CC` (**ShellPrintEx**): Prints messages to the console, offering flexibility in formatting and customization.
- `sub_F9BC` (**CopyMem**): Copies memory blocks.

In the UEFI environment, several global structures and function tables play crucial roles. These structures are typically accessed through the `EFI_SYSTEM_TABLE`, which is a central data structure providing pointers to essential system services. Here is where we can find some of this data types in our shell's image:

- `EFI_SYSTEM_TABLE`: Located at offset `0xE8460` in our image, this table contains pointers to other important structures, including the boot and runtime services tables.
- `EFI_BOOT_SERVICES`: Residing at offset `0xE8470`, this table provides functions for interacting with the system during the boot process.
- `EFI_RUNTIME_SERVICES`: Found at offset `0xE8490`, this table contains functions that can be invoked both during the boot process and after the system has started.

Another significant structure, `FILE_HANDLE_FUNCTION_MAP`, is populated within the `sub_14C28` (**ShellLibConstructorWorker**) function. This map contains function pointers derived from an `EFI_SHELL_PROTOCOL` instance, providing a mechanism for handling file-related operations within the shell environment.

```

if ( efiShellProtocol )
{
    FileFunctionMap.GetFileInfo = efiShellProtocol->GetFileInfo;
    FileFunctionMap.SetFileInfo = efiShellProtocol->SetFileInfo;
    FileFunctionMap.ReadFile = efiShellProtocol->ReadFile;
    FileFunctionMap.WriteFile = efiShellProtocol->WriteFile;
    FileFunctionMap.CloseFile = efiShellProtocol->CloseFile;
    FileFunctionMap.DeleteFile = efiShellProtocol->DeleteFile;
    FileFunctionMap.GetFilePosition = efiShellProtocol->GetFilePosition;
    FileFunctionMap.SetFilePosition = efiShellProtocol->SetFilePosition;
    FileFunctionMap.FlushFile = efiShellProtocol->FlushFile;
    GetFileSize = efiShellProtocol->GetFileSize;
}
else
{
    FileFunctionMap.GetFileInfo = (EFI_FILE_INFO *(__fastcall *)(void *))FileHandleGetInfo;
    FileFunctionMap.SetFileInfo = (unsigned __int64 (__fastcall *)(void *, const EFI_FILE_INFO *))FileHandleSetInfo;
    FileFunctionMap.ReadFile = (unsigned __int64 (__fastcall *)(void *, unsigned __int64 *, void *))FileHandleRead;
    FileFunctionMap.WriteFile = (unsigned __int64 (__fastcall *)(void *, unsigned __int64 *, void *))FileHandleWrite;
    FileFunctionMap.CloseFile = (unsigned __int64 (__fastcall *)(void *))FileHandleClose;
    FileFunctionMap.DeleteFile = (unsigned __int64 (__fastcall *)(void *))FileHandleDelete;
    FileFunctionMap.GetFilePosition = (unsigned __int64 (__fastcall *)(void *, unsigned __int64 *))FileHandleGetPosition;
    FileFunctionMap.SetFilePosition = (unsigned __int64 (__fastcall *)(void *, unsigned __int64))FileHandleSetPosition;
    FileFunctionMap.FlushFile = (unsigned __int64 (__fastcall *)(void *))FileHandleFlush;
    GetFileSize = (unsigned __int64 (__fastcall *)(void *, unsigned __int64 *))FileHandleGetSize;
}
FileFunctionMap.GetFileSize = GetFileSize;
return 0LL;

```

Figure 12: Population of the FileFunctionMap

## File Decryption Logic

To understand the decryption process, we'll focus on the `decrypt_file` command. This command is implemented within the same image as the shell itself. By searching for the string `decrypt_file` we can locate the relevant code, specifically the function `sub_31BC4`.

This function handles the decryption process, starting by validating the input arguments: a filename to decrypt and a password. It then opens the specified file and checks for the magic value `C4TB` to confirm that it's a valid encrypted file. If the file doesn't begin with this magic, a message is displayed to indicate that it's not a supported file format.

```

QEMU - Press Ctrl+Alt+G to release grab
Machine View
FS0:\> decrypt_file DilbootApp.efi.enc ATestPassword
Successfully read 76192 bytes from DilbootApp.efi.enc
Oh, you thought you could just waltz in here and decrypt ANY file, did you?
NewsFlash: Only .c4tb encrypted JPEGs are worthy of my decryption powers.
FS0:\> _

```

Figure 13: decrypt\_file second attempt

The function then goes on to parse the encrypted file's header. This header comprises 16 bytes, with the first 4 bytes representing the `C4TB` magic signature mentioned previously. The subsequent 4 bytes contain a little-endian integer, likely denoting the size of the encrypted data that follows. The remaining two 4-byte fields appear to function as follows: one is a little-endian integer with an offset pointing to an additional,

yet-to-be-determined buffer located after the encrypted blob, while the last one represents the little-endian size of this undetermined buffer. The structure of the header can be organized as follows:

```
C/C++
typedef struct _encrypted_file_header {
    unsigned char magic[4];
    uint32_t blob_size;
    uint32_t unk_buffer_offset;
    uint32_t unk_buffer_size;
}encrypted_file_header;
```

In our code, this header structure is nested within a larger structure. This larger structure is allocated at the address `0x31E6E` and its fields are populated after parsing the aforementioned header. The layout of the larger structure can be illustrated as follows:

```
C/C++
typedef struct _encrypted_file_data
{
    encrypted_file_header file_hdr;
    unsigned char *encrypted_blob; // pointer to a buffer that is populated with the
    contents of the encrypted blob from the .c4tb file
    unsigned char *unk_buffer; // pointer to a buffer that is populated with the
    undetermined contents that follow the encrypted blob
}encrypted_file_data;
```

After parsing the encrypted file, the function verifies the password length, ensuring it contains 16 characters. This is a crucial requirement for the decryption process.

Subsequently, the password's bytes are copied into specific, predefined locations within the `unk_buffer` of the `encrypted_file_data` structure. Following this, the `sub_31274` function is invoked, operating on a global structure residing at address `0xE86C0`. The outcome of this function call is then compared to zero. If the result is non-zero, the decryption process continues. It is highly likely that the actual password validation occurs within this `sub_31274` function, which we will explore in greater detail in the following section.

```
inputPassword = (BYTE *)gInputPassword;
unkBuff = gUnkBuff;
gUnkBuff[5] = *(_BYTE *)gInputPassword;
unkBuff[4] = inputPassword[2];
unkBuff[12] = inputPassword[4];
unkBuff[11] = inputPassword[6];
unkBuff[19] = inputPassword[8];
unkBuff[18] = inputPassword[10];
unkBuff[26] = inputPassword[12];
unkBuff[25] = inputPassword[14];
unkBuff[33] = inputPassword[16];
unkBuff[32] = inputPassword[18];
unkBuff[40] = inputPassword[20];
unkBuff[39] = inputPassword[22];
unkBuff[47] = inputPassword[24];
unkBuff[46] = inputPassword[26];
unkBuff[54] = inputPassword[28];
unkBuff[53] = inputPassword[30];
sub_31274(unkBuff);
```

Figure 14: password copy

If the result of `sub_31274` is non-zero, the CRC32 hash of the password is calculated and compared against a predefined list of hashes:

- `0x8AE981A5`
- `0x92918788`
- `0x80076040`

If the computed hash aligns with one of these predefined values, the associated password is stored in a buffer referenced by a global variable. Essentially, the code appears to systematically store each successfully decrypted password in a designated buffer. These buffers play a role in a subsequent key generation logic, which we will delve into later in our analysis.

Assuming the password is correct, it's used as an RC4 key to decrypt the `encrypted_blob` data within the encrypted file. The RC4 decryption function is implemented in `sub_31934`. After decryption, the decrypted data is processed to ensure it's a valid JPEG image by checking for the presence of `JFIF` magic value. If the decryption is successful, the decrypted data is written to a new file without the `.c4tb` extension, and the original encrypted file is deleted.

```
PrintYayCat();
fileHdr = gFileHdr;
ARC4(gAsciiPassword, v43, (char *)gFileHdr->blob, gFileHdr->blobSize, gFileHdr->blob);
blob = fileHdr->blob;
if ( blob[6] != 'J' || blob[7] != 'F' || blob[8] != 'I' || blob[9] != 'F' )
{
    ShellPrintEx(-1, -1, L"is that what you think you're doing? Trying to crack something?\r\n");
    ShellPrintEx(-1, -1, L"Well, let me tell you, you're wasting your time.\r\n");
    return 2LL;
}
```

Figure 15: JFIF Check

## Exploring the Password Validation Logic

Let's take a closer look at the `sub_31274` function, which we previously identified as the core component responsible for validating the password used to decrypt each `.c4tb` file. Upon closer inspection, this function takes the `unk_buffer` data located after each encrypted blob in the `.c4tb` file (once it has been populated with the password's bytes) and assigns it to a field within a globally defined structure. It then proceeds to evaluate the bytes within this `unk_buffer` utilizing a switch-case construct to execute specific code segments depending on the value of each processed byte from the buffer. Within each case of this switch statement, we observe an action being performed, followed by the buffer pointer being incremented by two. Starting to sound familiar?

Indeed, we are witnessing a virtual machine implementation here. This implies that the processed `unk_buffer` contains bytecode interpreted by this virtual machine. The buffer pointer essentially serves as an instruction pointer, while the structure containing it likely represents the virtual machine's context. The switch-case construct houses the opcode handlers responsible for executing each supported opcode.

By delving into these handlers, we can glean some characteristics of both the bytecode and the underlying virtual machine:

- The instruction pointer (or buffer pointer) is incremented by one before dispatching to the corresponding opcode handler. This means that opcode values are of 1 byte length.
- In certain handlers, we observe a further increment of the instruction pointer by 2. Subsequently, operations are performed on the 2 bytes that were skipped over during this increment, as part of the handler's logic. This suggests that bytecode instruction arguments, when present, are 16 bits long.
- The argument bytes processed by some opcode handlers are assigned to a pointer that is often decremented by one. This pointer also updates a field within the virtual machine's context structure. This behavior strongly hints at a stack-based mechanism, where certain operations involve stack pops or pushes, and a stack top pointer is maintained within the context structure.

Furthermore, recall that we previously observed a field within the virtual machine's context structure that is checked in the `sub_31BC4` function. This check determines whether the underlying `.c4tb` file should be

decrypted using the evaluated password. Based on our current understanding, we can confidently conclude that this field represents the final outcome or result of the virtual machine's execution.

## Building a Bytecode Disassembler

With these insights in hand, we're ready to commence the construction of our bytecode disassembler. It's important to acknowledge that the virtual machine we're dealing with in this challenge is rooted in a publicly available implementation known as PigletVM (<https://github.com/vkazanov/bytecode-interpreters-post/blob/master/pigletvm.c>). However, it's evident that this implementation has undergone several modifications, necessitating careful attention as we develop our disassembler.

As our initial step, let's formally define the structure of the virtual machine context located at address `0xE86C0`. This structure largely aligns with the one found in the PigletVM repository, with the primary difference being adjustments to the sizes of the stack and memory buffers.

```
C/C++
typedef struct _vm_context{
    /* instruction pointer */
    uint8_t *ip;

    /* stack */
    uint64_t stack[256];
    uint64_t *stack_top;

    uint64_t memory[65536];

    /* the VM execution result */
    uint64_t result;
} vm_context;
```

Now, let's examine all available opcodes by analyzing their respective handlers within the switch-case construct. Again, these handlers largely mirror those found in the PigletVM repository, albeit with some notable additions. These additions primarily encompass opcodes related to arithmetic and logical operations, along with those responsible for displaying values on the screen:

### Data Manipulation and Memory Operations

- **OP\_PUSH1 (0x1)**: Pushes an immediate argument (a constant value embedded in the instruction) onto the stack.
- **OP\_LOAD1 (0x2)**: Loads a value from a memory cell addressed by an immediate argument onto the stack.
- **OP\_LOADADD1 (0x3)**: Loads a value from a memory cell addressed by an immediate argument, then adds it to the value at the top of the stack.

- **OP\_STOREI (0x4)**: Pops a value from the stack and stores it into a memory cell addressed by an immediate argument.
- **OP\_LOAD (0x5)**: Pops an address from the stack and uses it to retrieve a value from the corresponding memory cell, pushing the retrieved value onto the stack.
- **OP\_STORE (0x6)**: Pops an address from the stack, then pops a value from the stack, and stores this value into the memory cell at the popped address.
- **OP\_DUP (0x7)**: Duplicates the value at the top of the stack, effectively creating a copy and pushing it onto the stack.

## Arithmetic and Logical Operations

- **OP\_ADD (0x9)**: Pops two values from the stack, adds them together, and pushes the result back onto the stack.
- **OP\_ADDI (0xA)**: Adds an immediate value to the value at the top of the stack.
- **OP\_SUB (0xB)**: Pops two values from the stack, subtracts the second value from the first, and pushes the result onto the stack.
- **OP\_DIV (0xC)**: Pops two values from the stack, divides the first value by the second, and pushes the result onto the stack.
- **OP\_MUL (0xD)**: Pops two values from the stack, multiplies them together, and pushes the result onto the stack.
- **OP\_XOR (0x1A)**: Pops two values from the stack, performs a bitwise XOR operation on them, and pushes the result back onto the stack.
- **OP\_OR (0x1B)**: Pops two values from the stack, performs a bitwise OR operation on them, and pushes the result back onto the stack.
- **OP\_AND (0x1C)**: Pops two values from the stack, performs a bitwise AND operation on them, and pushes the result back onto the stack.
- **OP\_MOD (0x1D)**: Pops two values from the stack, performs a modulo operation (first value modulo second value), and pushes the result back onto the stack.
- **OP\_SHFTL (0x1E)**: Pops two values from the stack, performs a left shift operation on the first value by the number of bits specified in the second value, and pushes the result back onto the stack.
- **OP\_SHFTR (0x1F)**: Pops two values from the stack, performs a right shift operation on the first value by the number of bits specified in the second value, and pushes the result back onto the stack.
- **OP\_ROT32 (0x20), OP\_ROT16 (0x21), OP\_ROT8 (0x22), OP\_ROT32 (0x23), OP\_ROT16 (0x24), OP\_ROT8 (0x25)**: Perform left and right rotations on 32-bit, 16-bit, and 8-bit values, respectively. The first value popped from the stack is the value to be rotated, and the second value specifies the number of bits to rotate.

## Control Flow

- **OP\_JUMP (0xE)**: Unconditionally jumps to an absolute bytecode address specified by the immediate argument.
- **OP\_JUMP\_IF\_TRUE (0xF)**: Pops the top value from the stack. If it evaluates to true (non-zero), jumps to an absolute bytecode address specified by the immediate argument.



- **OP\_JUMP\_IF\_FALSE (0x10)**: Pops the top value from the stack. If it evaluates to false (zero), jumps to an absolute bytecode address specified by the immediate argument.

## Comparison Operations

- **OP\_EQUAL (0x11), OP\_LESS (0x12), OP\_LESS\_OR\_EQUAL (0x13), OP\_GREATER (0x14), OP\_GREATER\_OR\_EQUAL (0x15), OP\_GREATER\_OR\_EQUAL\_I (0x16)**: Pop two values from the stack, compare them according to the specific operator, and push the result (1 for true, 0 for false) back onto the stack.

## Miscellaneous

- **OP\_POP\_RES (0x17) & OP\_SET\_RES (0x19)**: Pop the top value from the stack and set it as the execution result in the VM context.
- **OP\_DONE (0x18)**: Terminates the execution of the virtual machine.
- **OP\_PRINTC (0x26)**: Pops a value from the stack and prints it as an ASCII character.
- **OP\_DISCARD (0x8)**: Pops two values from the stack and discards them - this operation seems to have no effect other than adjusting the stack.

There are two paths we can take from here. If you are already aware that the code utilizes a modified PigletVM, you could extend the original PigletVM with the additional opcodes, compile it, and then leverage its built-in disassembler, which can be found here:

<https://github.com/vkazanov/bytecode-interpreters-post/blob/master/pigletvm-exec.c#L127>

Alternatively, let's assume no prior knowledge of PigletVM and develop our own disassembler from scratch using Python. Although our approach will share similarities with the logic employed in the PigletVM repository, the resulting code turns out to be quite simple:

Python

```
# The dictionary maps an opcode value to a tuple of the format (ins_name, has_arg)
opcode_dict = {0x1: ("PUSHI", 1), 0x2: ("LOADI", 1), 0x3: ("LOADADDI", 1), 0x4: ("SOTREI",
1), 0x5: ("LOAD", 0), 0x6: ("STORE", 0), 0x7: ("DUP", 0), 0x8: ("DISCARD", 0), 0x9: ("ADD",
0), 0xA: ("ADDI", 1), 0xB: ("SUB", 0), 0xC: ("DIV", 0), 0xD: ("MUL", 0), 0xE: ("JUMP", 1),
0xF: ("JUMP_IF_TRUE", 1), 0x10: ("JUMP_IF_FALSE", 1), 0x11: ("EQUAL", 0), 0x12: ("LESS", 0),
0x13: ("LESS_OR_EQUAL", 0), 0x14: ("GREATER", 0), 0x15: ("GREATER_OR_EQUAL", 0), 0x16:
("GERATER_OR_EQUAL_I", 1), 0x17: ("POP_RES", 0), 0x18: ("DONE", 0), 0x19: ("SET_RES", 0), 0x1A:
("XOR", 0), 0x1B: ("OR", 0), 0x1C: ("AND", 0), 0x1D: ("MOD", 0), 0x1E: ("SHFTL", 0), 0x1F:
("SHFTR", 0), 0x20: ("ROTL32", 0), 0x21: ("ROTR32", 0), 0x22: ("ROTL16", 0), 0x23: ("ROTR16",
0), 0x24: ("ROTL8", 0), 0x25: ("ROTR8", 0), 0x26: ("PRINTC", 0)}

# This function disassembles bytecode, iterating over each instruction. It prints the
corresponding human-readable opcode and any arguments associated with the instruction.
# The instruction pointer is advanced based on the size of the current instruction and its
arguments, ensuring accurate tracking of the bytecode's execution flow.
```

```
def disassemble(bytecode, len):
    ip = 0
    disassembly = []
    while (ip < len):
        ins_ip = ip
        ins = opcode_dict[bytecode[ip]][0]
        # check if the instruction has an argument or not
        if (opcode_dict[bytecode[ip]][1]):
            arg_int = ( bytecode[ip + 1] << 8 ) | (bytecode[ip + 2])
            arg = "0x%x" % arg_int
            ip += 3
        else:
            ip += 1
            arg = ""

        disassembly += ["0x%x: %s %s" % (ins_ip, ins, arg)]
    return "\n".join(disassembly)
```

Now, we have the ability to extract the bytecode from each encrypted .c4tb file and disassemble it to examine the inner workings of its password verification mechanism. To facilitate this analysis, we can employ a straightforward script, such as the following:

```
Python
for i in range(1,4):
    with open("catmeme%d.jpg.c4tb" % i, "rb") as f:
        file_data = f.read()
        bytecode_offset = int.from_bytes(file_data[8:12], "little")
        bytecode_len = int.from_bytes(file_data[12:16], "little")
        bytecode = file_data[bytecode_offset:]
        with open("catmeme%d_disassembly.txt" % i, "wb") as fd:
            fd.write(bytes(disassemble(bytecode, bytecode_len), "UTF-8"))
```

## Bytecode 1: Byte Rotation & Comparison

First, let's analyze the bytecode extracted from `catmeme1.jpg.c4tb`. The initial segment of the bytecode can be dissected into distinct fragments. It commences with two sequences of `STORE` operations, where hardcoded values are placed into consecutive indices. This pattern aligns with the initialization of two arrays:

```
C/C++
0x0: PUSHI 0x0
0x3: PUSHI 0xbbaa
```

0x6: STORE  
0x7: PUSHI 0x1  
0xa: PUSHI 0xddcc  
0xd: STORE  
0xe: PUSHI 0x2  
0x11: PUSHI 0xffee  
0x14: STORE  
0x15: PUSHI 0x3  
0x18: PUSHI 0xadde  
0x1b: STORE  
0x1c: PUSHI 0x4  
0x1f: PUSHI 0xefbe  
0x22: STORE  
0x23: PUSHI 0x5  
0x26: PUSHI 0xfeca  
0x29: STORE  
0x2a: PUSHI 0x6  
0x2d: PUSHI 0xbeba  
0x30: STORE  
0x31: PUSHI 0x7  
0x34: PUSHI 0xcdab  
0x37: STORE  
0x38: PUSHI 0xa  
0x3b: PUSHI 0x6144  
0x3e: STORE  
0x3f: PUSHI 0xb  
0x42: PUSHI 0x7534  
0x45: STORE  
0x46: PUSHI 0xc  
0x49: PUSHI 0x6962  
0x4c: STORE  
0x4d: PUSHI 0xd  
0x50: PUSHI 0x6c63  
0x53: STORE  
0x54: PUSHI 0xe  
0x57: PUSHI 0x3165  
0x5a: STORE  
0x5b: PUSHI 0xf  
0x5e: PUSHI 0x6669  
0x61: STORE  
0x62: PUSHI 0x10  
0x65: PUSHI 0x6265  
0x68: STORE  
0x69: PUSHI 0x11  
0x6c: PUSHI 0x6230  
0x6f: STORE

Upon closer inspection of the bytecode buffer initialization code within `sub_31274` that we pointed out earlier, we observe a revealing detail: the bytes of the validated user password (expected to be 16 bytes in length) are assigned in specific indices so as to replace the placeholder values `0xbbaa`, `0xddcc`, `0xffee`, `0xadde`, `0xefbe`, `0xfeca`, `0xbeba`, and `0xcdab`. In other words, we see logic that is used to integrate user input into the bytecode's data structures. The second initialization sequence seemingly constructs another array populated with predefined hard-coded values. We can provide a rough outline of both initialization processes using C-like pseudocode for clarity:

```
C/C++
wchar_t userInputArr[8];
wchar_t hardcodedBuffer[8];

userInputArr[0] = 0xbbaa;
userInputArr[1] = 0xddcc;
userInputArr[2] = 0xffee;
userInputArr[3] = 0xadde;
userInputArr[4] = 0xefbe;
userInputArr[5] = 0xfeca;
userInputArr[6] = 0xbeba;
userInputArr[7] = 0xcdab;

hardcodedBuffer[0] = 0x6144; // "aD"
hardcodedBuffer[1] = 0x7534; // "u4"
hardcodedBuffer[2] = 0x6962; // "ib"
hardcodedBuffer[3] = 0x6c63; // "lc"
hardcodedBuffer[4] = 0x3165; // "le"
hardcodedBuffer[5] = 0x6669; // "fi"
hardcodedBuffer[6] = 0x6265; // "be"
hardcodedBuffer[7] = 0x6230; // "b0"
```

Analyzing the second buffer, we find the string `Da4ubiclelifeb0b` initialized within it in reverse. However, attempts to decrypt `catmeme1.jpg.c4tb` with this string are unsuccessful, suggesting it's not the correct password.

Proceeding with our bytecode analysis, we uncover a recurring pattern repeated four times. This pattern involves a series of logical operations performed on bytes from each of the previously mentioned buffers. The pattern concludes with a `STORE` instruction, saving the result of these operations in memory:

```
C/C++
// First sequence of operations on userInputArr elements at indices 3,2,1,0, result is stored
// in memory index 8
0x70: PUSHI 0x8 // Memory index in which the result of the actions below is saved
```

```
0x73: PUSHI 0x3 // Load the value of userInputBuffer[3]
0x76: LOAD
0x77: PUSHI 0x30
0x7a: SHFTL // userInputBuffer[3] << 0x30
0x7b: PUSHI 0x2 // Load the value of userInputBuffer[2]
0x7e: LOAD
0x7f: PUSHI 0x20
0x82: SHFTL // userInputBuffer[2] << 0x20
0x83: OR // (userInputBuffer[3] << 0x30) | (userInputBuffer[2] << 0x20)
0x84: PUSHI 0x1 // Load the value of userInputBuffer[1]
0x87: LOAD
0x88: PUSHI 0x10
0x8b: SHFTL // userInputBuffer[1] << 0x10
0x8c: OR // (userInputBuffer[3] << 0x30) | (userInputBuffer[2] << 0x20) |
(userInputBuffer[1] << 0x10)
0x8d: PUSHI 0x0 // Load the value of userInputBuffer[0]
0x90: LOAD
0x91: OR // (userInputBuffer[3] << 0x30) | (userInputBuffer[2] << 0x20) |
(userInputBuffer[1] << 0x10) | userInputBuffer[0]
0x92: STORE // Store the result of the previous instruction into memory index 8

// Second sequence of operations on userInputArr elements at indices 7,6,5,4, result is
stored in memory index 9.
0x93: PUSHI 0x9
0x96: PUSHI 0x7
0x99: LOAD
0x9a: PUSHI 0x30
0x9d: SHFTL
0x9e: PUSHI 0x6
0xa1: LOAD
0xa2: PUSHI 0x20
0xa5: SHFTL
0xa6: OR
0xa7: PUSHI 0x5
0xaa: LOAD
0xab: PUSHI 0x10
0xae: SHFTL
0xaf: OR
0xb0: PUSHI 0x4
0xb3: LOAD
0xb4: OR
0xb5: STORE

// First sequence of operations on hardcodedBuffer elements at indices 3,2,1,0 (correspond
to memory indices 0xd,0xc,0xb,0xa), result is stored in memory index 0x12.
0xb6: PUSHI 0x12
0xb9: PUSHI 0xd
0xbc: LOAD
```

```
0xbd: PUSHI 0x30
0xc0: SHFTL
0xc1: PUSHI 0xc
0xc4: LOAD
0xc5: PUSHI 0x20
0xc8: SHFTL
0xc9: OR
0xca: PUSHI 0xb
0xcd: LOAD
0xce: PUSHI 0x10
0xd1: SHFTL
0xd2: OR
0xd3: PUSHI 0xa
0xd6: LOAD
0xd7: OR
0xd8: STORE

// Second sequence of operations on hardcodedBuffer at indices 7,6,5,4 (correspond to memory
indices 0x11,0x10,0xf,0xe), result is stored in memory index 0x13.
0xd9: PUSHI 0x13
0xdc: PUSHI 0x11
0xdf: LOAD
0xe0: PUSHI 0x30
0xe3: SHFTL
0xe4: PUSHI 0x10
0xe7: LOAD
0xe8: PUSHI 0x20
0xeb: SHFTL
0xec: OR
0xed: PUSHI 0xf
0xf0: LOAD
0xf1: PUSHI 0x10
0xf4: SHFTL
0xf5: OR
0xf6: PUSHI 0xe
0xf9: LOAD
0xfa: OR
0xfb: STORE
```

These operations seem to be transforming the buffers initialized at the start of the bytecode into four 64-bit integers, as illustrated in the following pseudocode:

```
C/C++
uint64_t userInputInt1;
uint64_t userInputInt2;
```

```

uint64_t hardcodedBufferInt1;
uint64_t hardcodedBufferInt2;

userInputInt1 = (userInputBuffer[3] << 0x30) | (userInputBuffer[2] << 0x20) |
(userInputBuffer[1] << 0x10) | userInputBuffer[0]; // userInput1 = 0xaddeffeeddccbaa
userInputInt2 = (userInputBuffer[7] << 0x30) | (userInputBuffer[6] << 0x20) |
(userInputBuffer[5] << 0x10) | userInputBuffer[4]; // userInput1 = 0xcdabbebafeafebe

hardcodedBufferInt1 = (hardcodedBuffer[3] << 0x30) | (hardcodedBuffer[2] << 0x20) |
(hardcodedBuffer[1] << 0x10) | hardcodedBuffer[0]; // hardcodedBufferInt1 = "lcibu4aD"
hardcodedBufferInt2 = (hardcodedBuffer[7] << 0x30) | (hardcodedBuffer[6] << 0x20) |
(hardcodedBuffer[5] << 0x10) | hardcodedBuffer[4]; // hardcodedBufferInt2 = "b0befile"

```

This is followed by several memory initializations of what could be later used as variables:

```

C/C++
0xfc: PUSHI 0x14 // memory index 0x14 (we'll refer to it as variable 'i')
0xff: PUSHI 0x0 // value 0
0x102: STORE // i = 0
0x103: PUSHI 0x18 // memory index 0x18 (we'll refer to it as variable 'boolFlag')
0x106: PUSHI 0x1 // value 1
0x109: STORE // boolFlag = 1
0x10a: PUSHI 0x17 // memory index 0x17 (we'll refer to it as variable 'j')
0x10d: PUSHI 0x0 // value 0
0x110: STORE // j = 0
0x111: PUSHI 0x19 // memory index 0x19 (we'll refer to it as variable 'k')
0x114: PUSHI 0x0 // value 0
0x117: STORE // k = 0

```

The following bytecode utilizes these initialized variables to extract individual characters from `userInputInt1` or `userInputInt2`, and `hardcodedBufferInt1` or `hardcodedBufferInt2`. It employs the variable `i` as an incrementing index, shifting each of these integers right by multiples of 8 bits (depending on the processed character's index), and then performing a logical AND operation with `0xFF`. In addition, it appears to perform this code only if `boolFlag` variable is set:

```

C/C++
// Check if boolFlag is true
0x118: PUSHI 0x18
0x11b: LOAD // Load the value of 'boolFlag' from memory
0x11c: PUSHI 0x1
0x11f: EQUAL

```

```
0x120: JUMP_IF_FALSE 0x241          // Skip the following block if boolFlag is false

// Check if i is less than 8
0x123: PUSHI 0x14
0x126: LOAD
0x127: PUSHI 0x8
0x12a: LESS
0x12b: JUMP_IF_FALSE 0x150 // Skip the following block if i is not less than 8

// If i < 8, extract a character from userInputInt1
0x12e: PUSHI 0x15          // We'll treat the memory at index 0x15 as inputChr
0x131: PUSHI 0x8
0x134: LOAD                // Load userInputInt1
0x135: PUSHI 0x8
0x138: PUSHI 0x14
0x13b: LOAD                // Load the value of 'i'
0x13c: MUL
0x13d: SHFTR              // The extracted character is obtained via userInputInt1 >>
(8 * i)
0x13e: STORE              // Store the extracted character in inputChr

// If i < 8, extract a character from hardcodedBufferInt1
0x13f: PUSHI 0x16          // We'll treat the memory at index 0x15 as hardcodedChr
0x142: PUSHI 0x12
0x145: LOAD                // Load hardcodedBufferInt1
0x146: PUSHI 0x8
0x149: PUSHI 0x14
0x14c: LOAD                // Load the value of 'i'
0x14d: MUL
0x14e: SHFTR              // The extracted character is obtained via
hardcodedBufferInt1 >> (8 * i)
0x14f: STORE              // Store the extracted character in hardcodedChr

// If i >= 8, extract a character from userInputInt2
0x150: PUSHI 0x14
0x153: LOAD                // Load the value of 'i'
0x154: PUSHI 0x7
0x157: GREATER
0x158: JUMP_IF_FALSE 0x17d // Skip the following block if i is not greater than or
equal to 8

0x15b: PUSHI 0x15
0x15e: PUSHI 0x9
0x161: LOAD                // Load userInputInt2
0x162: PUSHI 0x8
0x165: PUSHI 0x14
0x168: LOAD                // Load the value of 'i'
0x169: MUL
```



```
0x16a: SHFTR // The extracted character is obtained via userInputInt2 >>
(8 * i)
0x16b: STORE // Store the extracted character in inputChr

// If i >= 8, extract a character from hardcodedBufferInt2
0x16c: PUSHI 0x16
0x16f: PUSHI 0x13
0x172: LOAD // Load hardcodedBufferInt2
0x173: PUSHI 0x8
0x176: PUSHI 0x14
0x179: LOAD // Load the value of 'i'
0x17a: MUL
0x17b: SHFTR // The extracted character is obtained via
hardcodedBufferInt2 >> (8 * i)
0x17c: STORE // Store the extracted character in hardcodedChr

// // Mask the lower 8 bits of inputChr
0x17d: PUSHI 0x15
0x180: PUSHI 0x15
0x183: LOAD // Load inputChr
0x184: PUSHI 0xff
0x187: AND // InputChr & 0xff
0x188: STORE // Store the result back in inputChr

// Mask the lower 8 bits of hardcodedChr
0x189: PUSHI 0x16
0x18c: PUSHI 0x16
0x18f: LOAD // Load hardcodedChr
0x190: PUSHI 0xff
0x193: AND // hardcodedChr & 0xff
0x194: STORE // Store the result back in hardcodedChr
```

After extracting characters from both the password and hardcoded buffers, the bytecode proceeds to check specific indices within the hardcoded buffer. For each of these indices, there's logic in place to modify the corresponding character within the hardcoded buffer itself:

```
C/C++
// Go to the next block if i == 2
0x195: PUSHI 0x14
0x198: LOAD // Load the value of 'i'
0x199: PUSHI 0x2
0x19c: EQUAL
0x19d: JUMP_IF_FALSE 0x1ac
```

```
// If i == 2, rotate the corresponding hardcodedChr byte left by 4 bits
0x1a0: PUSHI 0x16
0x1a3: PUSHI 0x16
0x1a6: LOAD // Load hardcodedChr
0x1a7: PUSHI 0x4
0x1aa: ROTL8 // Bitwise rotation left of hardcodedChr as a byte by 4
bits
0x1ab: STORE // Store the result back in hardcodedChr

// Go to the next block if i == 9
0x1ac: PUSHI 0x14
0x1af: LOAD
0x1b0: PUSHI 0x9
0x1b3: EQUAL
0x1b4: JUMP_IF_FALSE 0x1c3

// If i == 9, rotate the corresponding hardcodedChr byte right by 2 bits
0x1b7: PUSHI 0x16
0x1ba: PUSHI 0x16
0x1bd: LOAD // Load hardcodedChr
0x1be: PUSHI 0x2
0x1c1: ROTR8 // Bitwise rotation right of hardcodedChr as a byte by 2
bits
0x1c2: STORE // Store the result back in hardcodedChr

// Go to the next block if i == 0xd
0x1c3: PUSHI 0x14
0x1c6: LOAD
0x1c7: PUSHI 0xd
0x1ca: EQUAL
0x1cb: JUMP_IF_FALSE 0x1da

// If i == 0xd, rotate the corresponding hardcodedChr byte left by 7 bits
0x1ce: PUSHI 0x16
0x1d1: PUSHI 0x16
0x1d4: LOAD // Load hardcodedChr
0x1d5: PUSHI 0x7
0x1d8: ROTL8 // Bitwise rotation left of hardcodedChr as a byte by 7 bits
0x1d9: STORE // Store the result back in hardcodedChr

// Go to the next block if i == 0xf
0x1da: PUSHI 0x14
0x1dd: LOAD
0x1de: PUSHI 0xf
0x1e1: EQUAL
0x1e2: JUMP_IF_FALSE 0x1f1

// If i == 0xf, rotate the corresponding hardcodedChr byte left by 7 bits
```

```
0x1e5: PUSHI 0x16
0x1e8: PUSHI 0x16
0x1eb: LOAD // Load hardcodedChr
0x1ec: PUSHI 0x7
0x1ef: ROTL8 // Bitwise rotation left of hardcodedChr as a byte by 7 bits
0x1f0: STORE // Store the result back in hardcodedChr
```

Once certain characters in the hardcoded buffer have been modified, the bytecode compares the input characters against their corresponding counterparts in the hardcoded buffer. If any mismatch is detected, the `boolFlag` is set to 0:

```
C/C++
0x1f1: PUSHI 0x15
0x1f4: LOAD // Load inputChr
0x1f5: PUSHI 0x16
0x1f8: LOAD // Load hardcodedChr
0x1f9: EQUAL
0x1fa: PUSHI 0x0
0x1fd: EQUAL
0x1fe: JUMP_IF_FALSE 0x208 // Skip the next block if inputChr == hardcodedChr

0x201: PUSHI 0x18 // Load boolFlag
0x204: PUSHI 0x0
0x207: STORE // boolFlag = 0
```

The subsequent code reveals that the entire logic described above constitutes the body of a loop, iterating 16 times (corresponding to the number of characters in the input buffer). The earlier check at address `0x11f` confirms that the loop's body executes only if `boolFlag` is set to 1. Consequently, if the count index `i` reaches 16 or higher, `boolFlag` is set to 0, terminating the loop.

Furthermore, the initial block of the following bytecode includes a check to see if `inputChr` matches `hardcodedChr`. The variable `j` is incremented solely when these two characters are equal, suggesting that `j` likely serves as a counter for matching characters:

```
C/C++
// Check again if the input character matches the corresponding hardcoded buffer character
0x208: PUSHI 0x15 // Load inputChr
0x20b: LOAD
0x20c: PUSHI 0x16 // Load hardcodedChr
0x20f: LOAD
```

```
0x210: EQUAL
0x211: JUMP_IF_FALSE 0x220 // Skip the next block if inputChr != hardcodedChr

// j++
0x214: PUSHI 0x17
0x217: PUSHI 0x17 // Load 'j'
0x21a: LOAD
0x21b: PUSHI 0x1
0x21e: ADD
0x21f: STORE

// i++
0x220: PUSHI 0x14
0x223: PUSHI 0x14
0x226: LOAD // Load 'i'
0x227: PUSHI 0x1
0x22a: ADD
0x22b: STORE

// Check if i > 0xf
0x22c: PUSHI 0x14
0x22f: LOAD // Load 'i'
0x230: PUSHI 0xf
0x233: GREATER
0x234: JUMP_IF_FALSE 0x23e // if i >= 16

0x237: PUSHI 0x18
0x23a: PUSHI 0x0
0x23d: STORE

// Next iteration of the loop
0x23e: JUMP 0x118
```

In conclusion, the bytecode verifies to check if there are precisely 16 matching characters. If this condition is met, it sets the virtual machine's execution result to 1 using the `SET_RES` instruction. This instruction will return the value of the variable `k`, which is initially set to 0 and only changed to 1 if a perfect match of 16 characters occurs:

```
C/C++
// Check if 'j' indicates that 16 characters are matching
0x241: PUSHI 0x17 // Load 'j'
0x244: LOAD
0x245: PUSHI 0x10
0x248: EQUAL
```

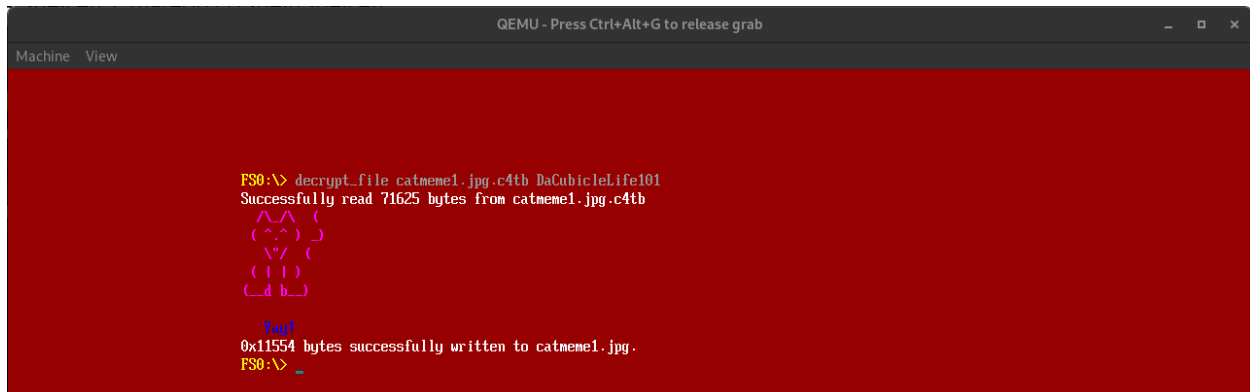
```
0x249: JUMP_IF_FALSE 0x253

// If j == 16, Set variable 'k' to 1
0x24c: PUSHI 0x19
0x24f: PUSHI 0x1
0x252: STORE

// Load the variable 'k' (which is 0 by default, unless set in the previous block)
// and use it as the virtual machine's execution result
0x253: PUSHI 0x19
0x256: LOAD // Load 'k'
0x257: SET_RES // Set 'k' as the VM's execution result
0x258: DONE
```

From the analysis and the specific character modifications at indices 2 ('4' -> 'C'), 9 ('l' -> 'L'), 13 ('b' -> '1'), and 15 ('b' -> '1') of the hardcoded buffer (via bitwise rotation), we can deduce that the correct password is **DaCubicleLife101**.

Entering this password into the shell using the `decrypt_file` command successfully decrypts the file.



```
QEMU - Press Ctrl+Alt+G to release grab
Machine View

FS0:\> decrypt_file catmeme1.jpg.c4tb DaCubicleLife101
Successfully read 71625 bytes from catmeme1.jpg.c4tb
  ^ ^ (
 ( ^ ^ )
  v v (
 ( | | )
 ( d b )

  flag
0x11554 bytes successfully written to catmeme1.jpg.
FS0:\> _
```

Figure 16: first successful `decrypt_file`

Finally, we can extract the decrypted JPG image using 7z and reveal the first part of the final flag - **th3\_r04d\_:**



Figure 17: catmeme1.jpg

For reference, we can represent the entire bytecode using the following refined pseudocode, where variable `i` is renamed to `index`, `j` to `matchCount`, `boolFlag` to `loopFlag` and `k` to `finalFlag`:

```
C/C++
bool bytecodeCheck1()
{

    // ===== Init input =====
    wchar_t userInputArr[8];
    wchar_t hardcodedBuffer[8];

    uint64_t userInputInt1;
    uint64_t userInputInt2;
```

```
uint64_t hardcodedBufferInt1;
uint64_t hardcodedBufferInt2;

bool finalFlag = 0;           // formerly denoted 'k'
bool loopFlag = 1;           // formerly denoted 'boolFlag'
uint64_t index = 0;           // formerly denoted 'i'
uint64_t matchCount;         // formerly denoted 'j'
uint64_t inputChr;
uint64_t hardcodedChr;

// Placeholder for user input, modified by the native code calling the bytecode
userInputArr[0] = 0xbbaa;
userInputArr[1] = 0xddcc;
userInputArr[2] = 0xffee;
userInputArr[3] = 0xadde;
userInputArr[4] = 0xefbe;
userInputArr[5] = 0xfeca;
userInputArr[6] = 0xbeba;
userInputArr[7] = 0xcdab;

hardcodedBuffer[0] = 0x6144; // "aD"
hardcodedBuffer[1] = 0x7534; // "u4"
hardcodedBuffer[2] = 0x6962; // "ib"
hardcodedBuffer[3] = 0x6c63; // "lc"
hardcodedBuffer[4] = 0x3165; // "le"
hardcodedBuffer[5] = 0x6669; // "fi"
hardcodedBuffer[6] = 0x6265; // "be"
hardcodedBuffer[7] = 0x6230; // "b0"

userInputInt1 = (userInputBuffer[3] << 0x30) | (userInputBuffer[2] << 0x20) |
(userInputBuffer[1] << 0x10) | userInputBuffer[0]; // userInput1 = 0xaddeffeeddccbaa
userInputInt2 = (userInputBuffer[7] << 0x30) | (userInputBuffer[6] << 0x20) |
(userInputBuffer[5] << 0x10) | userInputBuffer[4]; // userInput2 = 0xcdabbebafecefebe

hardcodedBufferInt1 = (hardcodedBuffer[3] << 0x30) | (hardcodedBuffer[2] << 0x20) |
(hardcodedBuffer[1] << 0x10) | hardcodedBuffer[0]; // hardcodedBufferInt1 = "lcibu4aD"
hardcodedBufferInt2 = (hardcodedBuffer[7] << 0x30) | (hardcodedBuffer[6] << 0x20) |
(hardcodedBuffer[5] << 0x10) | hardcodedBuffer[4]; // hardcodedBufferInt2 = "b0befile"

// ===== Compare loop =====

while(loopFlag == 1)
{
    if (index < 8)
    {
        inputChr = userInputInt1 >> (8 * index);
```

```
        hardcodedChr = hardcodedBufferInt1 >> (8 * index);
    }
    if (index >= 8)
    {
        inputChr = userInputInt2 >> (8 * (index % 8));
        hardcodedChr = hardcodedBufferInt2 >> (8 * (index % 8));
    }

    inputChr = inputChr & 255;
    hardcodedChr = hardcodedChr & 255;

    if (index == 2)
    {
        hardcodedChr = ROTL8(harcodedChr, 4);
    }

    if (index == 9)
    {
        hardcodedChr = ROTR8(harcodedChr, 2);
    }

    if (index == 13)
    {
        hardcodedChr = ROTL8(harcodedChr, 7);
    }

    if (index == 15)
    {
        hardcodedChr = ROTL8(harcodedChr, 7);
    }

    if (inputChr != harcodedChr)
    {
        loopFlag = 0;
    }

    if (inputChr == harcodedChr)
    {
        matchCount = matchCount + 1;
    }

    index = index + 1;

    if (index >= 16)
```



```
        {
            loopFlag = 0;
        }
    }

    // ===== Final test =====

    if(matchCount == 16)
    {
        finalFlag = 1;
    }

    return finalFlag;
```

## Bytecode 2: Enter RNG

Let's proceed to examine the bytecode extracted from `catmeme2.jpg.c4tb`. Although the previous analysis might have seemed laborious, it has significantly aided our understanding of this bytecode as well. This new bytecode shares a substantial amount of boilerplate code with the first one.

In fact, the code up until address `0x103`, which primarily handles buffer initialization, is identical in both bytecodes. The sole difference lies in the contents of `hardcodedBuffer`, which are initialized with distinct values, as shown in the following pseudo-code pertaining to the bytecode of `catmeme2.jpg.c4tb`:

C/C++

```
hardcodedBuffer[0] = 0xa059;
hardcodedBuffer[1] = 0x6a4d;
hardcodedBuffer[2] = 0xde23;
hardcodedBuffer[3] = 0x24c0;
hardcodedBuffer[4] = 0x64e2;
hardcodedBuffer[5] = 0x59b1;
hardcodedBuffer[6] = 0x7207;
hardcodedBuffer[7] = 0x7f5c;
```

Upon closer inspection of this bytecode, we see that the loop structure and most of its associated variables are consistent with the previous bytecode. The primary distinction lies within the body of the comparison loop itself. Let's begin by examining the initialization of the variables used in the loop, which serve the same purpose but have slightly different memory indices:

```
C/C++
// index = 0
0xfc: PUSHI 0x14
0xff: PUSHI 0x0
0x102: STORE

// loopFlag = 1
0x103: PUSHI 0x15
0x106: PUSHI 0x1
0x109: STORE

// matchCount = 0
0x10a: PUSHI 0x16
0x10d: PUSHI 0x0
0x110: STORE

// finalFlag
0x111: PUSHI 0x17
0x114: PUSHI 0x0
0x117: STORE
```

If you follow the code after this you'll also note that `inputChr` is stored in memory index `0x18` and `hardcodedChr` is stored in memory index `0x19`. Following that, we observe the calculation of several constants and their subsequent storage in new variables:

```
C/C++
// Calculation and storage of 17405 | (3 << 16), which is the constant 214013 (0x343fd)
0x118: PUSHI 0x1c // STORE index, we'll denote the underlying variable const1
0x11b: PUSHI 0x43fd
0x11e: PUSHI 0x3
0x121: PUSHI 0x10
0x124: SHFTL
0x125: OR
0x126: STORE

// Calculation and storage of 40643 | (38 << 16), which is the constant 2531011 (0x269ec3)
0x127: PUSHI 0x1d // STORE index, we'll denote the underlying variable const2
0x12a: PUSHI 0x9ec3
0x12d: PUSHI 0x26
0x130: PUSHI 0x10
0x133: SHFTL
0x134: OR
0x135: STORE

// Calculation and storage of 1 << 31, which is the constant 2147483648 (0x80000000)
```

```
0x136: PUSHI 0x1b          // STORE index, we'll denote the underlying variable const3
0x139: PUSHI 0x1
0x13c: PUSHI 0x1f
0x13f: SHFTL
0x140: STORE

// Storage of 0x1337
0x141: PUSHI 0x1e          // STORE index, we'll denote the underlying variable 'lcgVal'
0x144: PUSHI 0x1337
0x147: STORE
```

These constants should be familiar to you (or readily discoverable via a quick search) as components used within an LCG algorithm. Specifically, `0x343fd` serves as the multiplier, `0x269ec3` functions as the increment, and `0x80000000` acts as a modulus. Based on these observations, we can hypothesize that `0x1337` likely serves as an initial seed, provided to an LCG based random number generator during the bytecode's execution.

The bytecode spanning addresses `0x148` to `0x1c4` functions similarly to the previous code, extracting individual bytes (characters) from the runtime-generated 64-bit integers: `userInputInt1`, `userInputInt2`, `hardcodedBufferInt1`, and `hardcodedBufferInt2`. The first significant difference between this bytecode and the earlier one appears at address `0x1c5`. Here, we observe an explicit LCG round, confirming our suspicion that `0x1337` is indeed the seed for the LCG random number generator:

```
C/C++
...
// LCG round: (seed * 214013 + 2531011) % 0x80000000
// Result is stored back to 'lcgVal'
0x1c5: PUSHI 0x1e          // STORE arg ('lcgVal')
0x1c8: PUSHI 0x1c
0x1cb: LOAD                // Load const1 (214013)
0x1cc: PUSHI 0x1e
0x1cf: LOAD                // Load lcgVal (initially, 0x1337)
0x1d0: MUL
0x1d1: PUSHI 0x1d
0x1d4: LOAD                // Load const2 (2531011)
0x1d5: ADD
0x1d6: PUSHI 0x1b          // Load const3 (0x80000000)
0x1d9: LOAD
0x1da: MOD
0x1db: STORE
```

The heart of the password verification lies in a byte-by-byte comparison, wherein the LCG generated integers are used to generate a stream of XOR key bytes.

Here's how it works - for each character in the provided password:

- A new random number is produced using the Linear Congruential Generator (LCG).
- A corresponding byte from the LCG-generated sequence is extracted. The index for this extraction is calculated as the password byte's position modulo 4. The LCG-generated value is then bitwise shifted right by this index times 8 (i.e., by the number of bytes specified in this index).
- The extracted byte is masked with 0xff to retain only the least significant 8 bits. Finally, this masked byte is XORed with the original password byte.
- The resulting byte from the XOR is compared against the corresponding byte in the hardcodedBuffer.

```
C/C++
```

```
...
// Extracts an LCG generated byte from a corresponding index
// shiftedLcg = lcgVal >> (8 * (index % 4))
0x1e4: PUSHI 0x1a // STORE arg ('shiftedLcg')
0x1e7: PUSHI 0x1e
0x1ea: LOAD // Load 'lcgVal'
0x1eb: PUSHI 0x8
0x1ee: PUSHI 0x14
0x1f1: LOAD // Load 'index'
0x1f2: PUSHI 0x4
0x1f5: MOD
0x1f6: MUL
0x1f7: SHFTR
0x1f8: STORE

// keyChr = shiftedLcg & 0xff
0x1f9: PUSHI 0x1f // STORE arg, we'll denote it 'keyChr'
0x1fc: PUSHI 0x1a
0x1ff: LOAD // Load 'shiftedLcg'
0x200: PUSHI 0xff
0x203: AND
0x204: STORE

// xorRes = keyChr ^ inputChr
0x205: PUSHI 0x20 // STORE arg, we'll denote it 'xorRes'
0x208: PUSHI 0x18
0x20b: LOAD // Load 'inputChr'
0x20c: PUSHI 0x1f
0x20f: LOAD // Load 'keyChr'
0x210: XOR
0x211: STORE

// Compare xorRes to hardcodedChr
```

```
0x212: PUSHI 0x20
0x215: LOAD // Load 'xorRes'
0x216: PUSHI 0x19
0x219: LOAD // Load 'hardcodedChr'
0x21a: EQUAL
0x21b: PUSHI 0x0
0x21e: EQUAL
0x21f: JUMP_IF_FALSE 0x229
...
```

Armed with this knowledge, we can craft a straightforward Python script. This script will XOR the bytes of the `hardcodedBuffer` with the bytes generated from 4 consecutive rounds of the LCG, revealing the password:

```
Python
def lcg(seed):
    i = 0
    xorKey = []
    hardcodedBuffer = [0x59, 0xa0, 0x4d, 0x6a, 0x23, 0xde, 0xc0, 0x24, 0xe2, 0x64, 0xb1,
0x59, 0x07, 0x72, 0x5c, 0x7f]
    flag = []

    while i < 16:
        seed = (214013 * seed + 2531011) % 2147483648
        keyChr = (seed >> (8 * (i % 4))) & 0xff
        flag += [keyChr ^ hardcodedBuffer[i]]
        i += 1
    return "".join([chr(c) for c in flag])

print(lcg(0x1337))
```

With the password `G3tDaJ0bD0neM4te` in hand, we can proceed to decrypt the `catmeme2.jpg.c4tb` file. Let's use the `decrypt_shell` command and provide the password as an argument, this should successfully decrypt the file and reveal the second part of the flag - `t0_succ3ss`:



Figure 18: catmeme2.jpg

### Bytecode 3: Hash Function Cocktail

The bytecode for `catmeme3.jpg.c4tb` appears to follow a familiar pattern. We can anticipate input processing and initialization routines, coupled with validation checks likely embedded within a loop structure - reminiscent of the bytecodes we've previously encountered. However, a key distinction is apparent: this bytecode doesn't seem to contain a `hardcodedBuffer` with predefined values.

First we'll see an initialization of the constant `0xffffffff` which will likely be used as a bitmask later on:

```
C/C++
// mask = 0xffff
0x7e: PUSHI 0x1e           // 'mask' at index 0x1e
0x81: PUSHI 0xffff
0x84: STORE

0x85: PUSHI 0x1d
0x88: PUSHI 0x1e
0x8b: LOAD
0x8c: STORE
```

```
mask = (mask << 16) | mask
0x8d: PUSHI 0x1e
0x90: PUSHI 0x1e
0x93: LOAD
0x94: PUSHI 0x10
0x97: SHFTL
0x98: PUSHI 0x1d
0x9b: LOAD
0x9c: OR
0x9d: STORE
```

Next we find the following interesting piece of code:

```
C/C++
// hashVal = 0x1505
0xb3: PUSHI 0x13
0xb6: PUSHI 0x1505
0xb9: STORE // 'hashVal' at index 0x13 is initialized with 0x1505

// Check if i < 4
0xba: PUSHI 0x1b
0xbd: LOAD // Load 'i' from index 0x1b in memory
0xbe: PUSHI 0x4
0xc1: LESS
0xc2: JUMP_IF_FALSE 0x10f

// If i < 4, extract single character from 'userInputInt1'
0xc5: PUSHI 0x1c
0xc8: PUSHI 0x8
0xcb: LOAD // Load 'userInputInt1' from index 0x8 in memory
0xcc: PUSHI 0x8
0xcf: PUSHI 0x1b
0xd2: LOAD // Load 'i'
0xd3: MUL
0xd4: SHFTR
0xd5: STORE // Store the extracted 'inputChr' in index 0x1c

// inputChr = inputChr & 0xff
0xd6: PUSHI 0x1c
0xd9: PUSHI 0x1c
0xdc: LOAD
0xdd: PUSHI 0xff
0xe0: AND
0xe1: STORE
```

```
// hashValCopy = hashVal
0xe2: PUSHI 0x1d
0xe5: PUSHI 0x13
0xe8: LOAD
0xe9: STORE // 'tempVal' at index 0x1d in memory

// hashValCopy = (hashValCopy << 5) + hashVal + inputChr
0xea: PUSHI 0x13
0xed: PUSHI 0x13
0xf0: LOAD // Load 'hashVal'
0xf1: PUSHI 0x5
0xf4: SHFTL
0xf5: PUSHI 0x1d
0xf8: LOAD
0xf9: ADD
0xfa: PUSHI 0x1c
0xfd: LOAD
0xfe: ADD
0xff: STORE

// Increment i
0x100: PUSHI 0x1b
0x103: PUSHI 0x1b
0x106: LOAD
0x107: PUSHI 0x1
0x10a: ADD
0x10b: STORE

// Next loop iteration
0x10c: JUMP 0xba

// After loop ends: hashValCopy = hashValCopy & 0xffffffff
0x10f: PUSHI 0x13
0x112: PUSHI 0x13
0x115: LOAD
0x116: PUSHI 0x1e
0x119: LOAD
0x11a: AND
0x11b: STORE
```

This code can be rewritten as the following pseudocode, revealing that we are looking at a djb2 hash algorithm that operates on the first 4 bytes of the input:



```
C/C++
uint64_t hashVal = 5381;

while (i < 4)
{
    inputChr = (userInputInt1 >> (8 * i)) & 0xff;
    hashVal = (hashVal >> 5) + hashVal + inputChr;
    i++;
}

hashVal &= 0xffffffff;
```

The next snippet that follows this calculation appears to generate the hash constant `0x7c8df4cb` and compare our calculated input hash against it:

```
C/C++
// hardcodedHash1 = (0x7c8d << 16) | 0xf4cb
// Or 0x7c8df4cb
0x11c: PUSHI 0x14          // 'hardcodedHash1' at index 0x14
0x11f: PUSHI 0x7c8d
0x122: PUSHI 0x10
0x125: SHFTL
0x126: PUSHI 0xf4cb
0x129: OR
0x12a: STORE

// Compare hardcodedHash1 with the formerly calculated hashValCopy
0x12b: PUSHI 0x14
0x12e: LOAD
0x12f: PUSHI 0x13
0x132: LOAD
0x133: EQUAL
0x134: JUMP_IF_FALSE 0x143

// If a match found, hashMatchCount++
0x137: PUSHI 0x1f          // 'hashMatchCount' at index 0x1f
0x13a: PUSHI 0x1f
0x13d: LOAD
0x13e: PUSHI 0x1
0x141: ADD
0x142: STORE
```

Next, if this check passes successfully, we see another hash calculation:

```
C/C++
// Check if i < 8
0x155: PUSHI 0x1b
0x158: LOAD
0x159: PUSHI 0x8
0x15c: LESS
0x15d: JUMP_IF_FALSE 0x1a5

// If i < 8, extract the processed character from userInputInt1 to inputChr
0x160: PUSHI 0x1c
0x163: PUSHI 0x8
0x166: LOAD
0x167: PUSHI 0x8
0x16a: PUSHI 0x1b
0x16d: LOAD
0x16e: MUL
0x16f: SHFTR
0x170: STORE
0x171: PUSHI 0x1c
0x174: PUSHI 0x1c
0x177: LOAD
0x178: PUSHI 0xff
0x17b: AND
0x17c: STORE

// hashVal2 = ROTR32(hashVal2, 13)
0x17d: PUSHI 0x15          // 'hashVal2' at index 0x15
0x180: PUSHI 0x15
0x183: LOAD
0x184: PUSHI 0xd
0x187: ROTR32
0x188: STORE

// hashVal2 += inputChr
0x189: PUSHI 0x15
0x18c: PUSHI 0x15
0x18f: LOAD
0x190: PUSHI 0x1c
0x193: LOAD
0x194: ADD
0x195: STORE

// i++
0x196: PUSHI 0x1b
0x199: PUSHI 0x1b
0x19c: LOAD
0x19d: PUSHI 0x1
0x1a0: ADD
0x1a1: STORE
```

```
// Next loop iteration
0x1a2: JUMP 0x155

// After loop ends, hashVal2 ^= 0xffffffff
0x1a5: PUSHI 0x15
0x1a8: PUSHI 0x15
0x1ab: LOAD
0x1ac: PUSHI 0x1e
0x1af: LOAD
0x1b0: AND
0x1b1: STORE
```

This translates to the following pseudocode, which implements a ROT13-based hashing algorithm applied to the input characters specifically at positions 4 through 7 (inclusive):

```
C/C++
uint64_t hashVal2 = 0;

while (i < 8)
{
    inputChr = (userInputInt1 >> (8 * i)) & 0xff;
    hashVal2 = ROTR32(hashVal2) + inputChr;
    i++;
}

hashVal2 ^= 0xffffffff;
```

Once again, the calculated hash is compared against a constant, in this case `0x8b681d82`:

```
C/C++
// hardcodedHash2 = (0x8b68 << 0x10) | 0x1d82
// Or 0x8b681d82
0x1b2: PUSHI 0x16
0x1b5: PUSHI 0x8b68
0x1b8: PUSHI 0x10
0x1bb: SHFTL
0x1bc: PUSHI 0x1d82
0x1bf: OR
0x1c0: STORE

// Check if equal to calculated hash
```

```
0x1c1: PUSHI 0x16
0x1c4: LOAD
0x1c5: PUSHI 0x15
0x1c8: LOAD
0x1c9: EQUAL
0x1ca: JUMP_IF_FALSE 0x1d9

// If euqal, hashMatchCount++
0x1cd: PUSHI 0x1f
0x1d0: PUSHI 0x1f
0x1d3: LOAD
0x1d4: PUSHI 0x1
0x1d7: ADD
0x1d8: STORE
```

Moving on to the hash calculation for indices 8 through 15, the underlying algorithm appears to be adler32. While a deeper dive into the bytecode can confirm this, let's skip ahead to the calculation of the constant hash value used for comparison:

```
C/C++
// hardcodedHash3 = (0xf91 << 0x10) | 0x374
// Or 0x0f910374
0x277: PUSHI 0x18
0x27a: PUSHI 0xf91
0x27d: PUSHI 0x10
0x280: SHFTL
0x281: PUSHI 0x374
0x284: OR
0x285: STORE

0x286: PUSHI 0x18
0x289: LOAD
0x28a: PUSHI 0x17
0x28d: LOAD
0x28e: EQUAL
0x28f: JUMP_IF_FALSE 0x29e

0x292: PUSHI 0x1f
0x295: PUSHI 0x1f
0x298: LOAD
0x299: PUSHI 0x1
0x29c: ADD
0x29d: STORE
```

A quick Google search for the value `0f910374` reveals its the Adler32 hash of the commonly used keyword `password`.

Finally, let's look at the last hashing logic in this bytecode. First we'll note the calculation of some constants:

```
C/C++
// prime1 = 403
0x2a9: PUSHI 0xa
0x2ac: PUSHI 0x193
0x2af: STORE

// prime2 = 256
0x2b0: PUSHI 0xb
0x2b3: PUSHI 0x100
0x2b6: STORE

// prime = (prime2 << 16) | prime1
// Or 0x1000193
0x2b7: PUSHI 0xc
0x2ba: PUSHI 0xb
0x2bd: LOAD
0x2be: PUSHI 0x10
0x2c1: SHFTL
0x2c2: PUSHI 0xa
0x2c5: LOAD
0x2c6: OR
0x2c7: STORE
// hvalInit1 = 40389
0x2c8: PUSHI 0xd
0x2cb: PUSHI 0x9dc5
0x2ce: STORE

// hvalInit2 = 33052
0x2cf: PUSHI 0xe
0x2d2: PUSHI 0x811c
0x2d5: STORE

// hvalInit = (hvalInit2 << 16) | hvalInit1
// Or 0x811c9dc5
0x2d6: PUSHI 0xf
0x2d9: PUSHI 0xe
0x2dc: LOAD
0x2dd: PUSHI 0x10
0x2e0: SHFTL
0x2e1: PUSHI 0xd
0x2e4: LOAD
0x2e5: OR
0x2e6: STORE
```

```
// fnvsize = 1 << 32
0x2e7: PUSHI 0x10
0x2ea: PUSHI 0x1
0x2ed: PUSHI 0x20
0x2f0: SHFTL
0x2f1: STORE
```

The presence of the constants `0x1000193` and `0x811c9dc5` strongly suggest using the FNV1 hash algorithm. Let's examine the subsequent code to confirm this hypothesis:

C/C++

```
// fnvHash = hvalInit
0x2f2: PUSHI 0x19
0x2f5: PUSHI 0xf
0x2f8: LOAD
0x2f9: STORE

// index = 0
0x2fa: PUSHI 0x1b
0x2fd: PUSHI 0x0
0x300: STORE

// Check if index < 16
0x301: PUSHI 0x1b
0x304: LOAD
0x305: PUSHI 0x10
0x308: LESS
0x309: JUMP_IF_FALSE 0x37e

// Check if index < 8
0x30c: PUSHI 0x1b
0x30f: LOAD
0x310: PUSHI 0x8
0x313: LESS
0x314: JUMP_IF_FALSE 0x328

// If index < 8, extract a character corresponding to it from userInputChr1
// Store it in inputChr
0x317: PUSHI 0x1c
0x31a: PUSHI 0x8
0x31d: LOAD
0x31e: PUSHI 0x8
0x321: PUSHI 0x1b
```

```
0x324: LOAD
0x325: MUL
0x326: SHFTR
0x327: STORE

// else check if index >= 8
0x328: PUSHI 0x1b
0x32b: LOAD
0x32c: PUSHI 0x7
0x32f: GREATER
0x330: JUMP_IF_FALSE 0x344

// If index >= 8, extract a character corresponding to it from userInputChr2
// Store it in inputChr
0x333: PUSHI 0x1c
0x336: PUSHI 0x9
0x339: LOAD
0x33a: PUSHI 0x8
0x33d: PUSHI 0x1b
0x340: LOAD
0x341: MUL
0x342: SHFTR
0x343: STORE

// inputChr = inputChr & 0xff
0x344: PUSHI 0x1c
0x347: PUSHI 0x1c
0x34a: LOAD
0x34b: PUSHI 0xff
0x34e: AND
0x34f: STORE

// fnvHash = (fnvHash * prime) % fnvsize
0x350: PUSHI 0x19
0x353: PUSHI 0x19
0x356: LOAD
0x357: PUSHI 0xc
0x35a: LOAD
0x35b: MUL
0x35c: PUSHI 0x10
0x35f: LOAD
0x360: MOD
0x361: STORE

// fnvHash = fnvHash ^ inputChr
0x362: PUSHI 0x19
0x365: PUSHI 0x19
0x368: LOAD
```

```
0x369: PUSHI 0x1c
0x36c: LOAD
0x36d: XOR
0x36e: STORE

// index++
0x36f: PUSHI 0x1b
0x372: PUSHI 0x1b
0x375: LOAD
0x376: PUSHI 0x1
0x379: ADD
0x37a: STORE

// Next loop iteration
0x37b: JUMP 0x301
// After loop ends, fnvHash &= 0xffffffff
0x37e: PUSHI 0x19
0x381: PUSHI 0x19
0x384: LOAD
0x385: PUSHI 0x1e
0x388: LOAD
0x389: AND
0x38a: STORE
```

The bytecode clearly indicates the calculation of an FNV1-32 hash for the entire password. We can deduce this from the index range, which spans from 0 to 15, encompassing the full length of the password.

Now, let's identify the hash constant used for comparison with the calculated hash.

```
C/C++
// hardcodedHash4 = (0x31f0 << 16) | 0x9d2
// Or 0x31f009d2
0x38b: PUSHI 0x1a
0x38e: PUSHI 0x31f0
0x391: PUSHI 0x10
0x394: SHFTL
0x395: PUSHI 0x9d2
0x398: OR
0x399: STORE
```

Just as before, we observe a comparison between our calculated hash and the aforementioned constant. If they match, the `matchCount` is incremented.



```
C/C++
// Compare fnvHash with hardcodedHash4
0x39a: PUSHI 0x1a
0x39d: LOAD
0x39e: PUSHI 0x19
0x3a1: LOAD
0x3a2: EQUAL
0x3a3: JUMP_IF_FALSE 0x3b2

// If they match, matchCount++
0x3a6: PUSHI 0x1f
0x3a9: PUSHI 0x1f
0x3ac: LOAD
0x3ad: PUSHI 0x1
0x3b0: ADD
0x3b1: STORE
```

In essence, this bytecode processes the input password in distinct chunks:

- Indices 0 to 3: djb2 hash, compared against `0x7c8df4cb`
- Indices 4 to 7: ROT13-based hash, compared against `0x8b681d82`
- Indices 8 to 15: Adler32 hash, compared against `0x0f910374` (corresponding to the input keyword `password`, as evident in e.g., <https://md5calc.com/hash/adler32/password>)
- Entire password: FNV1-32 hash, compared against `0x31f009d2`

The bytecode systematically hashes each password chunk and compares it against a hardcoded value. A successful match increments an internal counter, `matchCount`. Crucially, subsequent chunk hashes are only calculated if the previous one matched. Finally, the bytecode compares the entire input with the above fnv1-32 hash and evaluates `matchCount` against 3. If all hash matches were successful, the execution result is set to 1, signifying a valid password.

Armed with the knowledge that the initial two hash algorithms operate on 4-byte inputs, and knowing the expected hash results for the correct ones, we can employ a simple brute-force approach. Keep in mind that djb2 and ROT13 hashes can produce collisions—multiple inputs resulting in the same hash value. Therefore, we'll need to verify potential password candidates by calculating the full FNV1-32 hash and comparing it against the constant extracted from the bytecode.

Here's a naive Python implementation for brute-forcing the password. Despite its simplicity, this code effectively cracks the password in under a minute, demonstrating the feasibility of the brute-force approach in this scenario:

```
Python
import string
import itertools
```

```
djb2_candidates = []
rot13_candidates = []
combinations = []

FNV_32_PRIME = 0x01000193
FNV1_32_INIT = 0x811c9dc5

def fnv(data, hval_init, fnv_prime, fnv_size):
    hval = hval_init
    for byte in data:
        hval = (hval * fnv_prime) % fnv_size
        hval = hval ^ ord(byte)
    return hval

def hash_fnv1_32_test(data, hval_init=FNV1_32_INIT):
    return fnv(data, hval_init, FNV_32_PRIME, 2**32) == 0x31f009d2

def ror(dword, bits):
    return (dword >> bits | dword << (32 - bits)) & 0xFFFFFFFF

def hash_rot13_test(s):
    hash = 0
    for c in s:
        hash = ror(hash, 13)
        hash += ord(c)
    return hash == 0x8b681d82

def hash_djb2_test(s):
    hash = 5381
    for x in s:
        hash = (( hash << 5) + hash) + ord(x)

    return (hash & 0xFFFFFFFF) == 0x7c8df4cb

def generate_strings(length=4):
    chars = string.printable
    for item in itertools.product(chars, repeat=length):
        yield "".join(item)

for s in generate_strings():
    if hash_djb2_test(s) == True:
        djb2_candidates.append(s)

for s in generate_strings():
    if hash_rot13_test(s) == True:
        rot13_candidates.append(s)
```

```
for i in range(len(djb2_candidates)):
    for j in range(len(rot13_candidates)):
        combinations.append((djb2_candidates[i], rot13_candidates[j]))

for comb in combinations:
    if (hash_fnv1_32_test(("%%s%%spassword" % (comb[0], comb[1])))):
        print(("%%s%%spassword" % (comb[0], comb[1])))
```

From executing the above code, we get the password `VerYDumBpassword`. Entering this password into the shell yields the third part of the flag - `_1s_alw4ys_`:



Figure 19: catmeme3.jpg

## Final Round: DilbootApp.efi

As observed earlier in the `FullShell` module, each entered password is validated against a pre-computed CRC32 checksum. Successful matches are stored in memory. Once all three passwords are correctly provided, the RC4 encryption key for `DilbertApp.efi.enc` is derived from their combined bytes. Interestingly, this key is `BureaucracY4Life` although we don't need to explicitly know it. The code seamlessly constructs and utilizes this key to automatically decrypt the file.

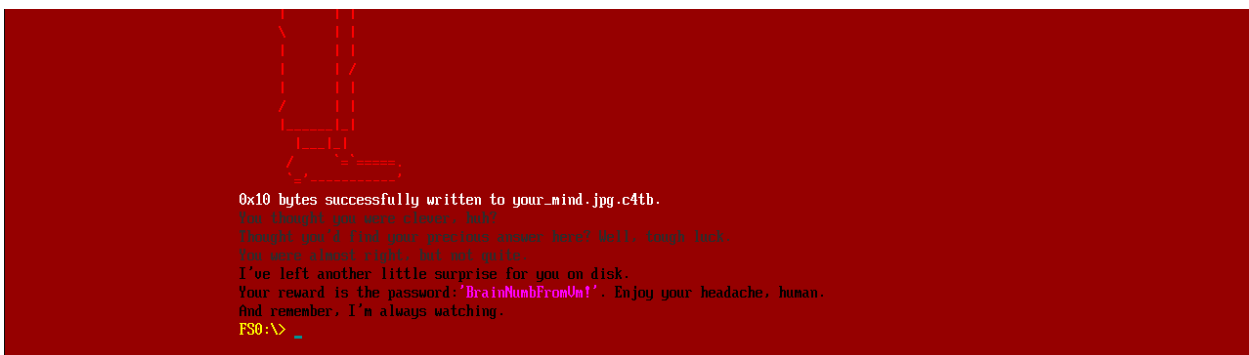
```

if ( gPassword1 && gPassword2 && gPassword3 && !gIsEfiAppDecrypted )
{
    gST->ConOut->SetAttribute(gST->ConOut, 64uLL);
    ShellPrintEx(-1, -1, L"Oh, you think you're so smart, huh? Decrypting JPEGs? Big deal.\r\n");
    ShellPrintEx(-1, -1, L"As a special favor, I'll let you enjoy the thrill of watching me\r\n");
    ShellPrintEx(-1, -1, L"decrypt the UEFI driver. Consider yourself lucky.\r\n");
    gST->ConOut->SetAttribute(gST->ConOut, 71uLL);
    dilbootEfiEncPassword = (char *)InternalAllocatePool(UNUSED_ARG(), 0x100uLL);
    gPassword4 = dilbootEfiEncPassword;
    if ( !dilbootEfiEncPassword )
        return 9LL;
    inputPassword3 = gPassword3;
    *dilbootEfiEncPassword = gPassword3[7];
    dilbootEfiEncPassword[1] = inputPassword3[5];
    dilbootEfiEncPassword[2] = inputPassword3[2];
    dilbootEfiEncPassword[3] = inputPassword3[1];
    inputPassword1 = gPassword1;
    dilbootEfiEncPassword[4] = gPassword1[1];
    dilbootEfiEncPassword[5] = inputPassword3[5];
    dilbootEfiEncPassword[6] = inputPassword1[6];
    dilbootEfiEncPassword[7] = inputPassword3[2];
    dilbootEfiEncPassword[8] = inputPassword1[1];
    dilbootEfiEncPassword[9] = inputPassword1[6];
    dilbootEfiEncPassword[10] = inputPassword3[3];
    dilbootEfiEncPassword[11] = inputPassword1[13] + 3;
    dilbootEfiEncPassword[12] = inputPassword1[9];
    dilbootEfiEncPassword[13] = inputPassword1[10];
    dilbootEfiEncPassword[14] = inputPassword1[11];
    dilbootEfiEncPassword[15] = inputPassword1[12];
}

```

Figure 20: Combining all three passwords to form RC4 key

We're almost there. Now, we'll simply run the decrypted `DilbootApp.efi` file from the shell and follow Catbert's on-screen instructions. It seems he's left us one last challenge—another encrypted file on the disk. Thankfully, he's also provided the password: `BrainNumbFromVm!`.



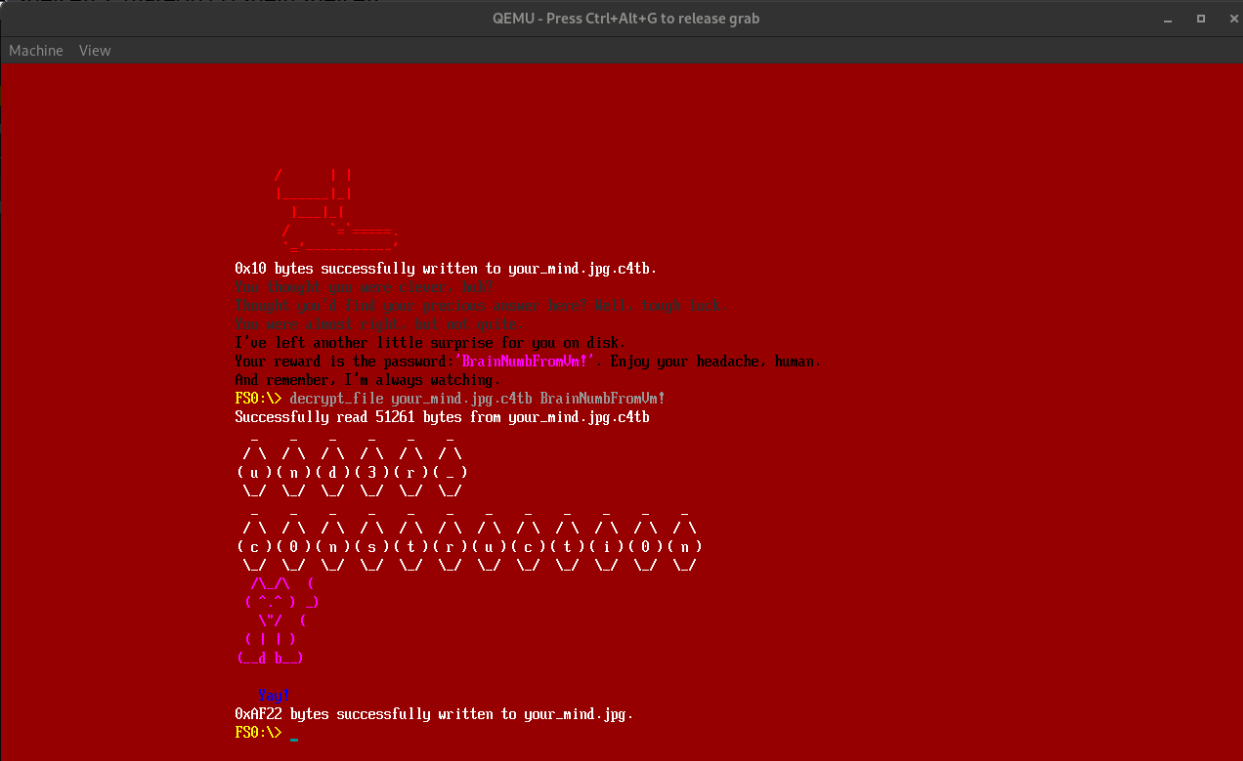
```

0x10 bytes successfully written to your_mind.jpg.c4tb.
You thought you were clever, huh?
Thought you'd find your precious answer here? Well, tough luck.
You were almost right, but not quite.
I've left another little surprise for you on disk.
Your reward is the password: 'BrainNumbFromVm!'. Enjoy your headache, human.
And remember, I'm always watching.
FS0:\>

```

Figure 21: Executed DilbootApp.efi

As expected, a new file, `your_mind.jpg.c4tb` awaits us on the disk (after being written to it by `DilbootApp.efi`). Let's run the `decrypt_file` command against it with the provided password, and witness the last part of the flag, `und3r_c0nstructi0n`, displayed triumphantly on our screen:



```

QEMU - Press Ctrl+Alt+G to release grab
Machine View

0x10 bytes successfully written to your_mind.jpg.c4tb.
the thought you were given, huh?
Thought you'd find your precious answer here? Well, tough luck.
You were almost right, but not quite.
I've left another little surprise for you on disk.
Your reward is the password: 'BrainNumbFromUm!'. Enjoy your headache, human.
And remember, I'm always watching.
FS0:\> decrypt_file your_mind.jpg.c4tb BrainNumbFromUm!
Successfully read 51261 bytes from your_mind.jpg.c4tb

(u)(n)(d)(3)(r)(_)
(c)(0)(n)(s)(t)(r)(u)(c)(t)(i)(0)(n)
(^)
(v)
(c)
(d b)

0x1F22 bytes successfully written to your_mind.jpg.
FS0:\> _

```

Figure 22: Decrypting your\_mind

The message unveiling the flag is embedded within more bytecode. It decodes and prints each character individually using the `PRINTC` instruction. While we won't delve into the intricacies of this bytecode here, consider it an exercise for the curious reader.

Additionally, a closer look at the disk reveals the decrypted `your_mind.jpg` file. Opening it unveils the final piece of the flag, embedded within a meme: [@flare-on.com](https://www.instagram.com/flare-on).

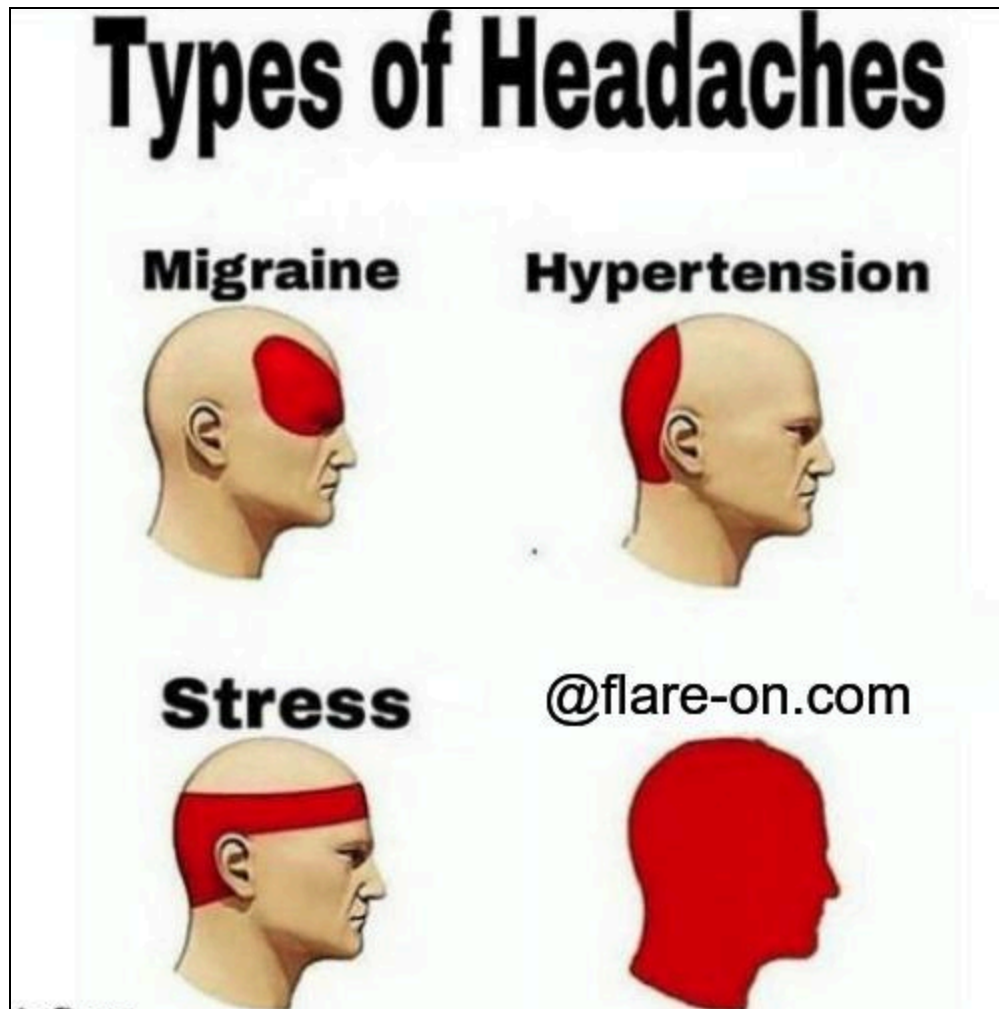


Figure 23: your\_mind.jpg

Bringing together all the flag parts we've gathered thus far, we can now construct the complete flag:

## Final Flag

Unset

`th3_r04d_t0_succ3ss_1s_alw4ys_und3r_c0nstructi0n@flare-on.com`