Stack String Decrypt con Ghidra Emulator (Orchard)

malverse.it/stack-string-decryptor-con-ghidra-emulator-orchard

Introduzione

Ciao a tutti! Oggi vedremo come realizzare uno script sfruttando le API di **Ghidra** per decifrare le stringhe di **Orchard**. In particolare analizzeremo il sample V3 (MD5: **cb442cbff066dfef2e3ff0c56610148f**) sviluppato in C++. Questo malware sfrutta inoltre un'interessante tecnica **DGA** il cui seed non è deterministico e dipende dal balance del genesis block; l'analisi tecnica di questo aspetto si può trovare sui blog bin.re e 360 Netlab.

Orchard per la decryption delle stringhe memorizza nello stack l'**offset del carattere all'interno dell'alfabeto**; vedremo quindi come è possibile realizzare un semplice script che sfrutta l'**EmulatorHelper** di Ghidra per la decifratura.

Iniziamo!

Analisi

Il malware inserisce nello stack gli offset dell'alfabeto e viene costruito ottenendo degli int da variabili globali e valori immediati; successivamente vengono chiamate due funzioni, la prima che si occupa di creare un oggetto e la seconda di fare la decryption della stringa.

La tecnica utilizzata dal malware è molto semplice, tuttavia è possibile complicare maggiormente l'analisi facendo sì che i valori nello stack dipendando dal risultato di funzioni, come ad esempio spiegato qui; in questo caso non potremmo escludere dall'esecuzione tutte le chiamate a funzione come fatto successivamente per Orchard, in quanto non ci permetterebbe di ottenere il valore corretto della stringa.

```
0071da04 8d 54 24 74     LEA        EDX=>funName,[ESP + 0x74]
0071da08 8b cc           MOV        ECX,ESP
0071da0a 0f 28 05        MOVAPS     XMM0,xmmword ptr [DAT_0075a6d0]          = 49h    I
         d0 a6 75 00
0071da11 0f 11 84        MOVUPS     xmmword ptr [ESP + local_40[0]],XMM0
         24 84 00
         00 00
0071da19 51              PUSH       ECX
0071da1a 0f 28 05        MOVAPS     XMM0,xmmword ptr [DAT_00759fb0]          = 27h    '
         b0 9f 75 00
0071da21 0f 11 84        MOVUPS     xmmword ptr [ESP + local_30[0]],XMM0
         24 98 00
         00 00
0071da29 50              PUSH       EAX
0071da2a 0f 28 05        MOVAPS     XMM0,xmmword ptr [DAT_0075a3d0]          = 45h    E
         d0 a3 75 00
0071da31 52              PUSH       EDX
0071da32 0f 11 84        MOVUPS     xmmword ptr [ESP + local_20[0]],XMM0
         24 b0 00
         00 00
0071da3a c7 84 24        MOV        dword ptr [ESP + local_c],'N'
         c4 00 00
         00 4e 00 ...
0071da45 e8 46 59        CALL       CreateObject                             void * * Create
         fe ff
0071da4a 8d 4c 24 34     LEA        ECX=>local_90,[ESP + '4']
0071da4e e8 3d d5        CALL       DecryptStackStrings                      undefined4 * De
         fe ff
```

Costruzione dello stack inserendo gli offset e chiamata delle due funzioni

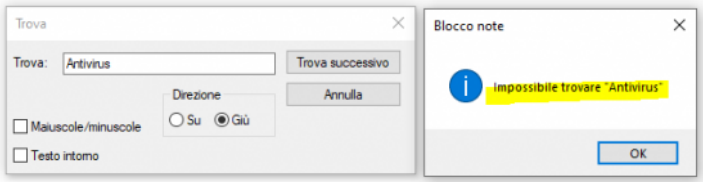
```
                        DAT_0075a6d0
    0075a6d0 49             ??          49h     I
    0075a6d1 00             ??          00h
    0075a6d2 00             ??          00h
    0075a6d3 00             ??          00h
    0075a6d4 56             ??          56h     V
    0075a6d5 00             ??          00h
    0075a6d6 00             ??          00h
    0075a6d7 00             ??          00h
    0075a6d8 45             ??          45h     E
    0075a6d9 00             ??          00h
    0075a6da 00             ??          00h
    0075a6db 00             ??          00h
    0075a6dc 52             ??          52h     R
    0075a6dd 00             ??          00h
    0075a6de 00             ??          00h
    0075a6df 00             ??          00h

                        DAT_0075a6e0
    0075a6e0 45             ??          45h     E
    0075a6e1 00             ??          00h
    0075a6e2 00             ??          00h
    0075a6e3 00             ??          00h
    0075a6e4 44             ??          44h     D
    0075a6e5 00             ??          00h
    0075a6e6 00             ??          00h
    0075a6e7 00             ??          00h
    0075a6e8 49             ??          49h     I
    0075a6e9 00             ??          00h
    0075a6ea 00             ??          00h
```

Variabili globali

che contengono gli offset

Interessante notare come floss non riesca a decifrare questo tipo di stringhe:

```
FLARE FLOSS RESULTS (version v2.1.0-0-gbf2bf1c)
+-----------------------+-------------------------------------------------------------------------------------------------------------------+
| file path             | C:\Users\malw\Desktop\e\Orchard\ce468c155dfb627612cc2cc6fff55e6c854415e07277fbc3c45f5ace08437a7e.bin |
| extracted strings     |                                                                                                                   |
|  static strings       | 2751                                                                                                              |
|  stack strings        | 8                                                                                                                 |
|  tight strings        | 3                                                                                                                 |
|  decoded strings      | 2                                                                                                                 |
+-----------------------+-------------------------------------------------------------------------------------------------------------------+

---------------------------
| FLOSS STACK STRINGS (8) |
---------------------------
1WBP
0WBh
false
null
true
false
null
true

---------------------------
| FLOSS TIGHT STRINGS (3) |
---------------------------
1096175631
16843010
1096200207

-----------------------------
| FLOSS DECODED STRINGS (2) |
-----------------------------
4278124287
4278124286


-------------------------------
| FLOSS STATIC STRINGS (2751) |
-------------------------------
-------------------------------
| FLOSS ASCII STRINGS (2185) |
```

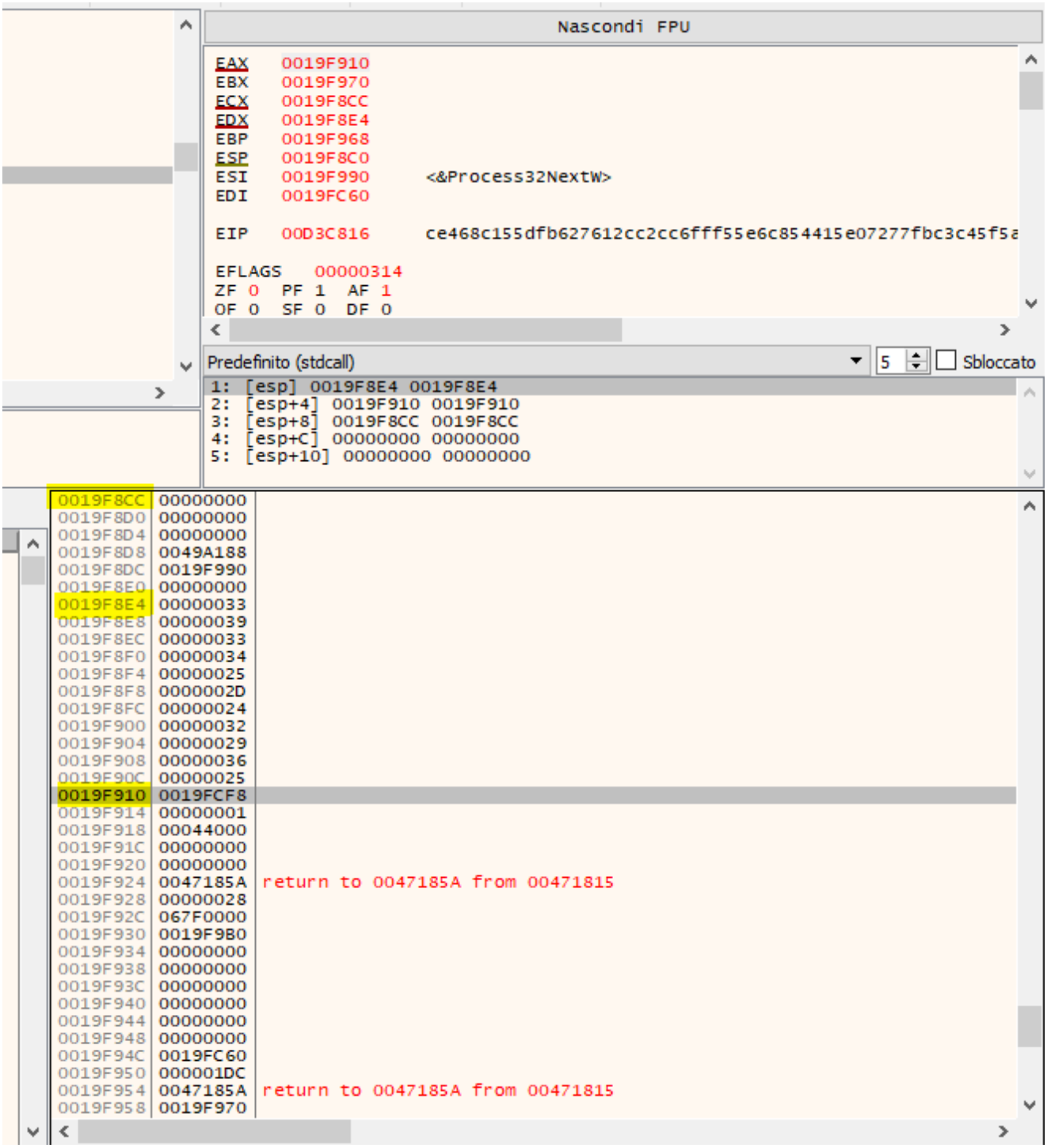


Risultato di floss

Per l'emulation successiva ci interessa come vengono utilizzati i registri e in particolare per la prima funzione:

- **EDX**: puntatore al primo offset all'interno dello stack.
- **EAX**: contiene il valore che sottratto all'indirizzo del primo offset e diviso per 4 permette di ottenere la lunghezza della stringa.
- **ECX**: puntatore this.

Ad esempio in questo caso EDX punta a **0x19F8E4** e quindi 33,39,33,34,25,2D,24,32,29,36,25 che equivale alla stringa **SYSTEMDRIVE**. In EAX invece abbiamo 0x19F910, quindi (0x19F910 – 0x19F8E4)/ 4 = **11** che è il numero di offset presenti.



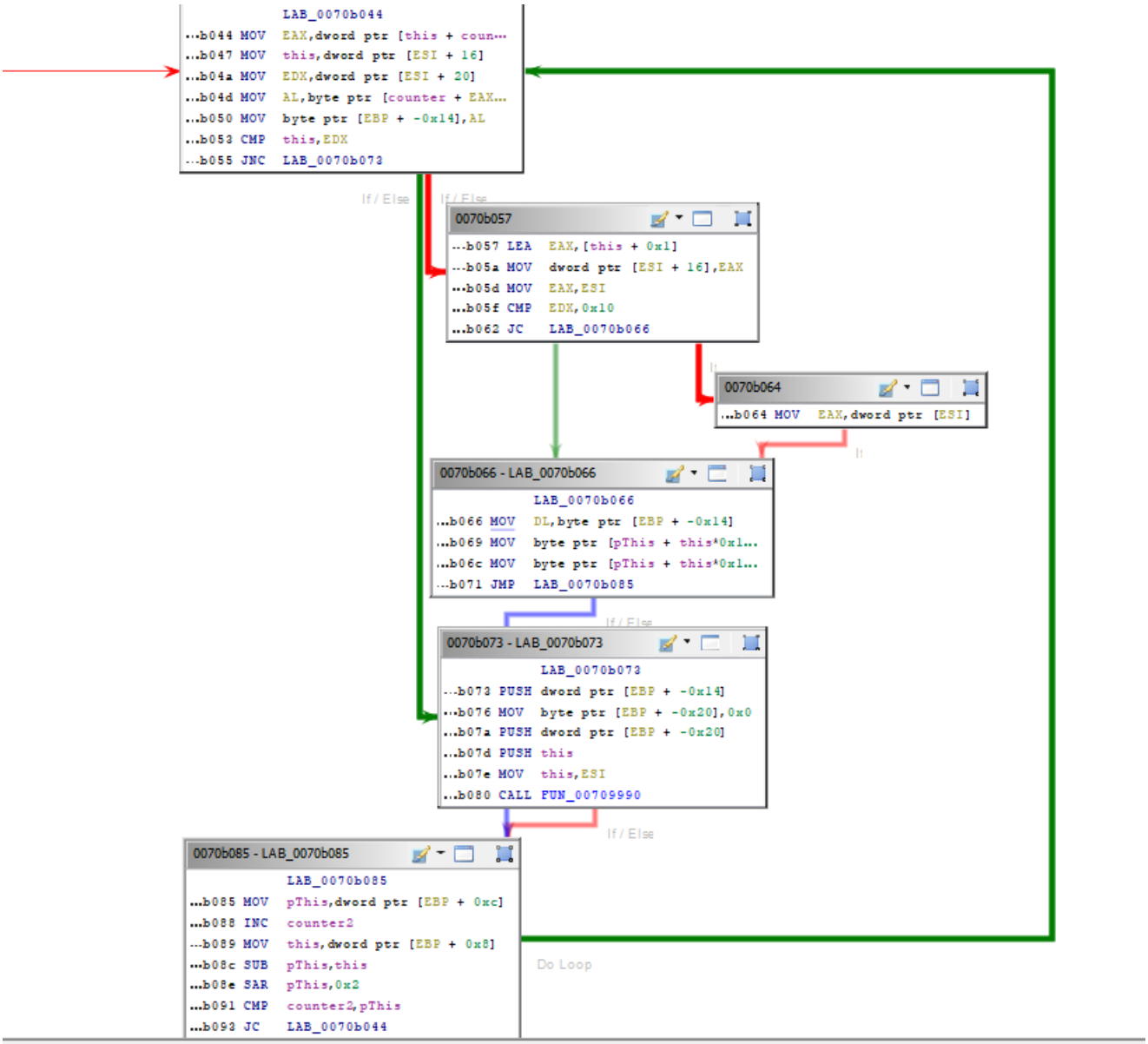
Utilizzo dei registri EAX, EDX e ECX

La prima funzione si occupa di creare l'oggetto allocando una nuova area di memoria e copiando gli offset dallo stack.

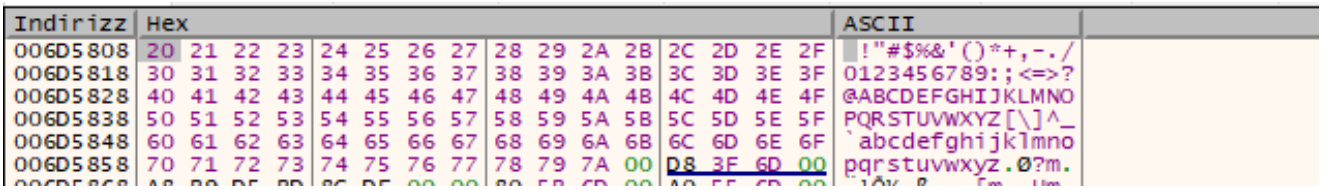
```
26  SetMemory:
27        *(void **)this = newMemory;
28        *(void **)((int)this + 4) = newMemory;
29        *(void **)((int)this + 8) = (void *)(size + (int)newMemory);
30        memcpy(newMemory,pStackCharacter,pStackSize - (int)pStackCharacter);
31        *(void **)((int)this + 4) = (void *)(size + (int)newMemory);
32        return (void **)this;
33      }
34      if (size < size + 0x23) {
35        newMemory2 = operator_new(size + 0x23);
36        if (newMemory2 != (void *)0x0) {
37          newMemory = (void *)((int)newMemory2 + 0x23U & 0xffffffe0);
38          *(void **)((int)newMemory - 4) = newMemory2;
39          goto SetMemory;
40        }
41        goto LAB_0070344d;
42      }
43    }
```

Creazione di un nuovo oggetto

La seconda funzione invece si occupa di decifrare la stringa, ottenendo gli offset che servono come indici per l'alfabeto:



Funzione di decifratura



Indirizz	Hex																ASCII
006D5808	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F	!"#$%&'()*+,-./
006D5818	30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F	0123456789:;<=>?
006D5828	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F	@ABCDEFGHIJKLMNO
006D5838	50	51	52	53	54	55	56	57	58	59	5A	5B	5C	5D	5E	5F	PQRSTUVWXYZ[\]^_
006D5848	60	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F	`abcdefghijklmno
006D5858	70	71	72	73	74	75	76	77	78	79	7A	00	D8	3F	6D	00	pqrstuvwxyz.Ø?m.

Alfabeto utilizzato per la decryption

Emulation con Ghidra

Per effettuare Emulation ci sono diverse soluzioni come Unicorn, Flare-Emu, Qiling e Dumpulator. In questo caso utilizzeremo le API di Ghidra attraveso la classe EmulatorHelper . Per approfondire l'argomento: QUI, QUI e QUI.

In particolare, i passaggi effettuati per l'emulazione sono:

- **Ottenere tutte le chiamate alla funzione di Decryption**
  Ottenibile facilmente con le API Ghidra getReferencesTo e getFromAddress
- **Determinare quale istruzioni devono essere emulate**
  - Funzione getStartAndEndAddress
  - Una stessa funzione può contenere più chiamate alla funzione di Decryption, quindi l'inizio dell'emulation è il prologo della funzione chiamante oppure l'indirizzo dell'istruzione successiva a una precedente chiamata di Decryption
- **Emulare solamente le istruzioni che ci interessano**
  - Escludere chiamate che possono modificare il flusso (call, jmp, ecc)
  - L'API Ghidra getFlowType permette di ottenere il FlowType dell'istruzione
- **Leggere l'output dopo l'emulation**
  - Indirizzo contenuto in EDX permette di ottenere il primo valore dell'offset nello stack
  - (Valore contenuto in EAX – valore contento in EDX) / 4 = size delle stringa
  - Le API Ghidra readRegister e readMemory permettono di leggere i valori memorizzati nei registri e nel range di memoria specificato

Il codice per effettuare l'emulation:

```
from ghidra.app.emulator import EmulatorHelper
from ghidra.program.model.symbol import SymbolUtilities

def decrypt(data):
    alphabet = "!\"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyzw"

    decrypted = ""

    for b in data:
       if(b != 0):
            decrypted += alphabet[b-1]

    return decrypted


class Emulator(object):

    def __init__(self):
        self.emulator = EmulatorHelper(currentProgram)

    def run(self):

        references = getReferencesTo(toAddr("DecryptStackStrings"))

        decryptOk = 0
        numCall = len(references)

        # Ottengo l'indirizzo delle chiamate alla funzione di Decryption
        for ref in references:

            callAddr = ref.getFromAddress()

            # Ottengo l'indirizzo di partenza e finale per l'emulation
            start, end = self.getStartAndEndAddress(callAddr)

            print("-----------------------------------------------------------------")
            print("Start: " + start.toString() )
            print("End: " + end.toString() )
            print("-----------------------------------------------------------------")

            # Imposto i registri e avvio l'emulation con gli indirizzi
precedentemente trovati
            self.setupRegister(start)
            self.runEmulation(start, end)

            # Ottengo i valori dei registri EBP, EDX, EAX per calcolare la size e
l'indirizzo di partenza degli offset
            valueEBP = self.emulator.readRegister("EBP")
            valueEDX = self.emulator.readRegister("EDX")
            valueEAX = self.emulator.readRegister("EAX")
```

```
            # Calcolo la dimensione della stringa
            size = valueEAX - valueEDX

            addr = toAddr(self.emulator.readRegister("EDX"))
            code = bytes(self.emulator.readMemory(addr, size))

            decryptedString = decrypt(code)

            if(len(decryptedString) != 0):
                #hexdump.hexdump(code)
                #print("{%s} Decrypted: %s " % (callAddr, decryptedString) )

                print("{%s} - %s " % (callAddr, decryptedString) )

                decryptOk += 1

                # Imposto il commento su Ghidra
                codeUnit = currentProgram.getListing().getCodeUnitAt(callAddr)
                codeUnit.setComment(codeUnit.PLATE_COMMENT, decryptedString)



        print("Decryption done:" + str(decryptOk))
        print("Call number: " + str(numCall))

        self.emulator.dispose()


    def setEIPNextInstruction(self):

        nextInstr = getInstructionAt(self.emulator.getExecutionAddress()).getNext()
        self.emulator.writeRegister(self.emulator.getPCRegister(),
nextInstr.getAddress().getOffset())
        return nextInstr


    # Funzione che permette di ottenere l'indirizzo di partenza e termine
dell'emulation
    def getStartAndEndAddress(self, fromAddr):

        # Ottengo il prologo della funzione attuale
        functionAddr = getFunctionContaining(fromAddr).getEntryPoint()

        while True:

            instr = getInstructionBefore(fromAddr)
            addr = instr.getAddress()

            mnemonic = instr.getMnemonicString().lower()

            if(len(instr.getOpObjects(0)) > 0):
                op1 = instr.getOpObjects(0)[0]
```

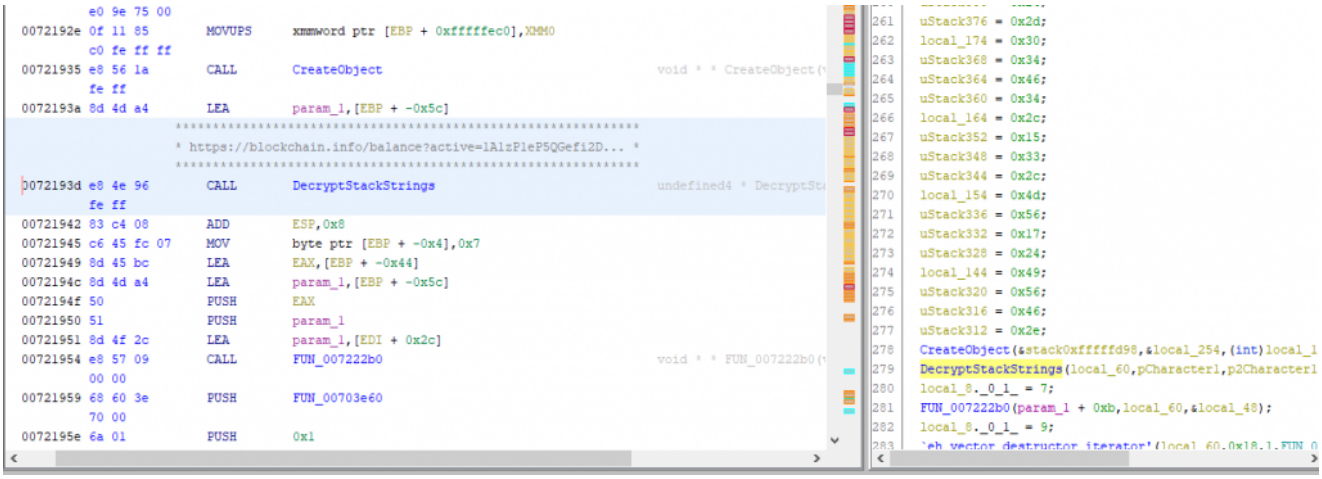
```
                op1_str = op1.toString().lower()

                op2 = instr.getOpObjects(0)[1]
                op2_str = op2.toString().lower()

            # Chiamata funzione CreateObject, indirizzo dove l'emulation deve
terminare
            if mnemonic == "call" and op1_str == "00703390":
                end = addr


            # Se siamo arrivati al prologo oppure a un'altra chiamata di decryption,
quello è l'indirizzo di partenza dell'emulation
            if functionAddr == addr or (mnemonic == "call" and op1_str ==
"0070af90"):
                return addr, end

            fromAddr = instr.getAddress()



    def setupRegister(self, start):

        emulator = self.emulator

        # Ottengo la memoria per lo stack e inizializzo ESP e EBP
        stack_address = (start.getAddressSpace().getMaxAddress().getOffset() >> 1) -
0x7fff

        emulator.writeRegister("ESP", stack_address)
        emulator.writeRegister("EBP", stack_address)
        emulator.writeRegister(emulator.getPCRegister(), start.getOffset())


    def runEmulation(self, start, end):

        emulationDone = False

        while not monitor.isCancelled():

            executionAddr = self.emulator.getExecutionAddress()
            currentInstruction = getInstructionAt(executionAddr)


            # Eseguo solamente istruzioni di tipo FALL_THROUGH escludendo quelle che
possono modificare il flusso (call, jmp, ja, ecc)
            op = str(currentInstruction).lower()
            flowType = currentInstruction.getFlowType().toString()
            prefixes = ["lock"]

            while(flowType != "FALL_THROUGH" or op in prefixes):
```

```
                newInstruction = self.setEIPNextInstruction()
                flowType = newInstruction.getFlowType().toString()
                op = newInstruction.toString()

            if self.emulator.step(monitor) == False:
                raise Exception("Emulation Error:
'{}'".format(self.emulator.getLastError()))


            if (executionAddr == end):
                emulationDone = True
                break


        if not emulationDone:
            raise Exception("[EMULATION] Error Emulation! ")

        print("[EMULATION] Done!")


if __name__=="__main__":

    print(" [EMULATION] Starting.. ")

    Emulator().run()
```

Il risultato della decryption:

```
{0072087c} - Message
{00720957} - Problem_Type
{00720a34} - Message_Type
{0071e62c} - .exe
{0071c81e} - SYSTEMDRIVE
{0071c8b2} - C:\
{0071cae8} - %08lX
{0071d047} - psapi.dll
{0071d176} - GetModuleFileNameExW
{00720546} - Serial_Number
{00720620} - Message_Type
{0070c08d} - kernel32.dll
{0070c18b} - Wow64DisableWow64FsRedirection
{0070c263} - Wow64RevertWow64FsRedirection
{0071d977} - nvcuda.dll
{0071da4e} - cuDriverGetVersion
{0071ed62} - Identity
{0071eeb5} - Operating_System
{0071f046} - System_Architecture
{0071f19b} - Elevated
{0071f292} - Threads
{0071f3b1} - Camera
{0071f49e} - Antivirus
{0071f5b5} - Version
{0071f72f} - Active_Window
{0071f864} - CPU_Model
{0071f9a1} - GPU_Models
{0071fa8a} - Ram_Size
{0071fb91} - Authenticate_Type
{0071a8a9} - ntdll.dll
{0071a9c9} - RtlGetVersion
{0071a02d} - MajorVersion
{0071a108} - MinorVersion
{0071a1e7} - BuildNumber
{0071a2e0} - ProductType
{0071abf7} - ROOT\SecurityCenter2
{0071ae48} - SELECTdisplayNameFROMAntiVirusProduct
{0071aee8} - WQL
{0071b1d4} - displayName
{0070b3de} - Name
{0070b4a2} - Type
{00718915} - nvapi.dll
{007189e1} - nvapi_QueryInterface
{0071902e} - OpenCL.dll
{007190ec} - clGetPlatformIDs
{007191ac} - clGetDeviceIDs
{00719278} - clGetDeviceInfo
{00721710} - %04d-%02d-%02d
{00721762} - .duckdns.org
{0072178c} - .com
{007217b6} - .net
{007217e0} - .org
```

```
{0072193d} - https://blockchain.info/balance?
active=1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa
{00722452} - CPU_Status
{00722538} - GPU_Status
{0072260b} - CPU_Hashrate
{007226e8} - GPU_Hashrate
{007227b2} - GPU_Type
{007228ac} - GPU_Algorithm
{0072297f} - Message_Type
{007195b1} - Host
{00719670} - Port
{00719741} - File_Name
{0071980e} - Handle
{00722ce8} - Domain
{00722dad} - Port
{00722e8b} - Process_ID
{00722f66} - Process_Name
{00723041} - Process_Path
{0072311f} - Is_Patched
{007231f6} - In_Memory
{007232d4} - Patch_Name
{007233b2} - Install_Path
{0072349a} - System_Idle
{00723581} - System_Uptime
{00723674} - Power_SaverMode
{00723751} - Message_Type
{00702716} - Required_Binary
{007027ea} - Cuda_Version
{0070c879} - *.exe
{0072393e} - Buffer
{007239cd} - Execute_Name
{00723a62} - Binary_Source
{00723af6} - Execute_Type
{00723cd6} - Execute_Name
{00723dbf} - Execute_Result
{00723ec0} - Execute_Result_Type
{00723f9d} - Message_Type
{00724146} - Option
{007241e8} - Message_Option
{007242d8} - Type
{00724373} - Transfer_Port
{00724401} - Message_Type
{00724539} - Buffer
{007245c0} - Handle
{0072465d} - Message_Option
```



Grazie per l'ascolto! Per qualunque commento e migliorie (sicuramente ce ne sono tante 🙂) fatemi sapere nei commenti!! A presto 😀

Share this content:

-
-
-
-
-