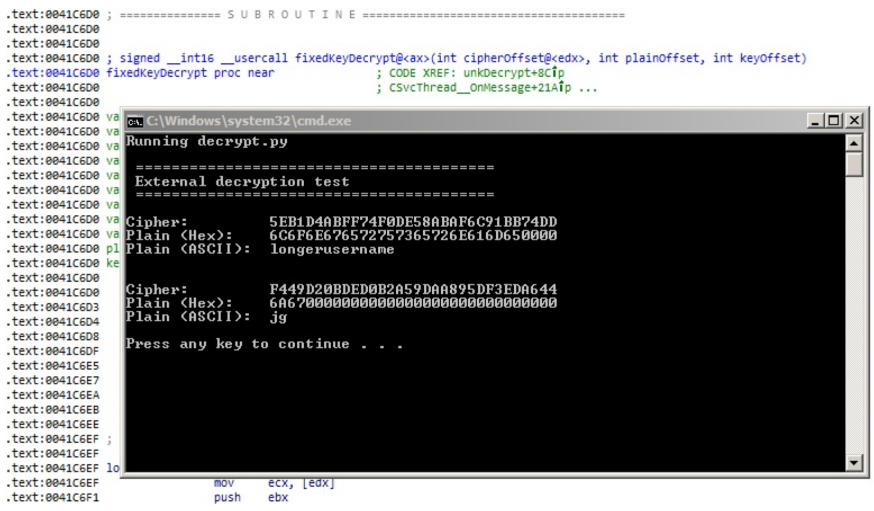
Ripping functions from PE binaries and reusing them elsewhere

Sat, Jun 2, 2018



```
.text:0041C6D0 ; =============== S U B R O U T I N E =======================================
.text:0041C6D0
.text:0041C6D0
.text:0041C6D0 ; signed __int16 __usercall fixedKeyDecrypt@<ax>(int cipherOffset@<edx>, int plainOffset, int keyOffset)
.text:0041C6D0 fixedKeyDecrypt proc near               ; CODE XREF: unkDecrypt+8C↑p
.text:0041C6D0                                         ; CSvcThread__OnMessage+21A↑p ...
.text:0041C6D0
.text:0041C6D0 va
.text:0041C6D0 va
.text:0041C6D0 va
.text:0041C6D0 va
.text:0041C6D0 va
.text:0041C6D0 va
.text:0041C6D0 va
.text:0041C6D0 va
.text:0041C6D0 va
.text:0041C6D0 pl
.text:0041C6D0 ke
.text:0041C6D0
.text:0041C6D0
.text:0041C6D3
.text:0041C6D4
.text:0041C6D8
.text:0041C6DF
.text:0041C6E5
.text:0041C6E7
.text:0041C6EA
.text:0041C6EB
.text:0041C6EE
.text:0041C6EF ;
.text:0041C6EF
.text:0041C6EF lo
.text:0041C6EF                 mov     ecx, [edx]
.text:0041C6F1                 push    ebx
```

```
Running decrypt.py

 ========================================
 External decryption test
 ========================================

Cipher:         5EB1D4ABFF74F0DE58ABAF6C91BB74DD
Plain (Hex):    6C6F6E676572757365726E616D650000
Plain (ASCII):  longerusername


Cipher:         F449D20BDED0B2A59DAA895DF3EDA644
Plain (Hex):    6A670000000000000000000000000000
Plain (ASCII):  jg

Press any key to continue . . .
```

Accessing a PE's function via Python!

There may be occasions where a PE binary contains helpful functions that could be reused elsewhere. If the function's implementation is not critical, it may be possible to *rip* the code out as-is. The function will then be used as a black box where input data is processed and returned as an output without knowledge of its internal workings.

There are some applications where this technique could be especially useful:

- Using features from legacy binaries such as proprietary parsers
- Implementing encryption/decryption functions
- Building keygens

Recently, I was disassembling a binary that had a function to decrypt a block of data. At over a thousand lines of x86 instructions, the function proved to be tricky to re-implement, hence I thought of trying this process as an experiment.

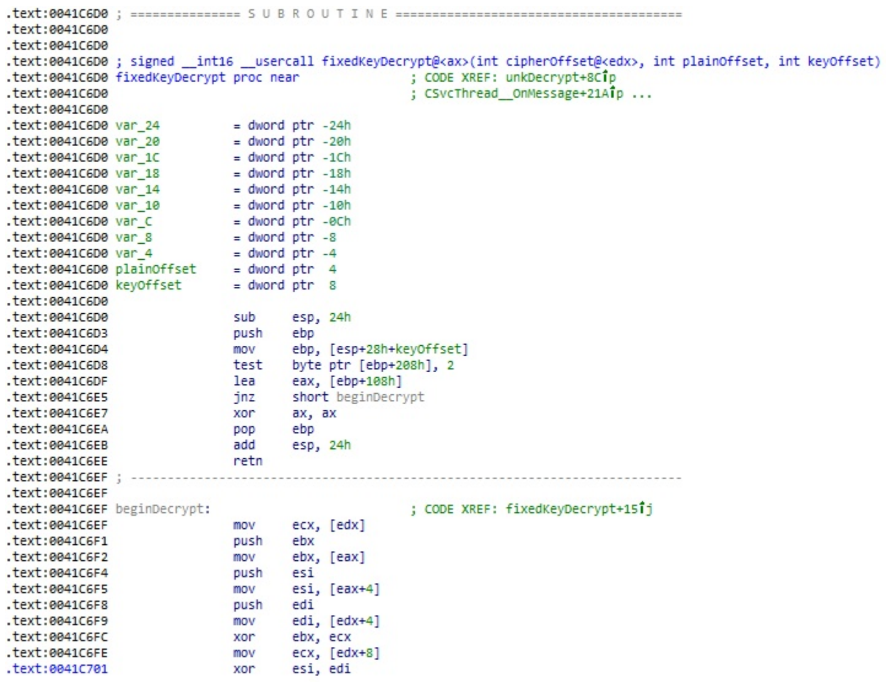
The general outline of the process is as follows:

- **Feasibility analysis**
  - The subroutine's inputs and outputs are identified; it should have minimal external calls, no indirect branches, and contain clear entry points and return paths.

- **Ripping the code**
    - The desired subroutine is ripped - the disassembly is copied as-is into an ASM file, and data references along with external calls are fixed.
- **Packaging the code**
    - A barebones PE is built around the ASM file, and helper functions are added to read prerequisite data and print the processed output.

      If everything works, the function should live as a standalone executable, receive data via the command line, and return processed data via stdout.

Feasibility analysis



```
.text:0041C6D0 ; =============== S U B R O U T I N E =======================================
.text:0041C6D0
.text:0041C6D0
.text:0041C6D0 ; signed __int16 __usercall fixedKeyDecrypt@<ax>(int cipherOffset@<edx>, int plainOffset, int keyOffset)
.text:0041C6D0 fixedKeyDecrypt proc near               ; CODE XREF: unkDecrypt+8C↑p
.text:0041C6D0                                         ; CSvcThread__OnMessage+21A↑p ...
.text:0041C6D0
.text:0041C6D0 var_24          = dword ptr -24h
.text:0041C6D0 var_20          = dword ptr -20h
.text:0041C6D0 var_1C          = dword ptr -1Ch
.text:0041C6D0 var_18          = dword ptr -18h
.text:0041C6D0 var_14          = dword ptr -14h
.text:0041C6D0 var_10          = dword ptr -10h
.text:0041C6D0 var_C           = dword ptr -0Ch
.text:0041C6D0 var_8           = dword ptr -8
.text:0041C6D0 var_4           = dword ptr -4
.text:0041C6D0 plainOffset     = dword ptr  4
.text:0041C6D0 keyOffset       = dword ptr  8
.text:0041C6D0
.text:0041C6D0                 sub     esp, 24h
.text:0041C6D3                 push    ebp
.text:0041C6D4                 mov     ebp, [esp+28h+keyOffset]
.text:0041C6D8                 test    byte ptr [ebp+208h], 2
.text:0041C6DF                 lea     eax, [ebp+108h]
.text:0041C6E5                 jnz     short beginDecrypt
.text:0041C6E7                 xor     ax, ax
.text:0041C6EA                 pop     ebp
.text:0041C6EB                 add     esp, 24h
.text:0041C6EE                 retn
.text:0041C6EF ; ---------------------------------------------------------------------------
.text:0041C6EF
.text:0041C6EF beginDecrypt:                           ; CODE XREF: fixedKeyDecrypt+15↑j
.text:0041C6EF                 mov     ecx, [edx]
.text:0041C6F1                 push    ebx
.text:0041C6F2                 mov     ebx, [eax]
.text:0041C6F4                 push    esi
.text:0041C6F5                 mov     esi, [eax+4]
.text:0041C6F8                 push    edi
.text:0041C6F9                 mov     edi, [edx+4]
.text:0041C6FC                 xor     ebx, ecx
.text:0041C6FE                 mov     ecx, [edx+8]
.text:0041C701                 xor     esi, edi
```

The desired function is first identified, along with its inputs and outputs.

In my case, the decryption function is located at `0x41C6D0`, taking 3 parameters: cipher (edx register), plaintext output (stack), key (stack).
These are addresses to byte arrays located in `.data`.

As the desired instructions will be copied as-is, some issues may arise from this process:

- Ripping utilities may not understand indirect branches (`jmp eax`) which are only determined at runtime, and hence might not properly follow and extract the entire disassembly.
- External calls must be manually resolved (such as `printf`) and additional imports might be required in the ASM file.

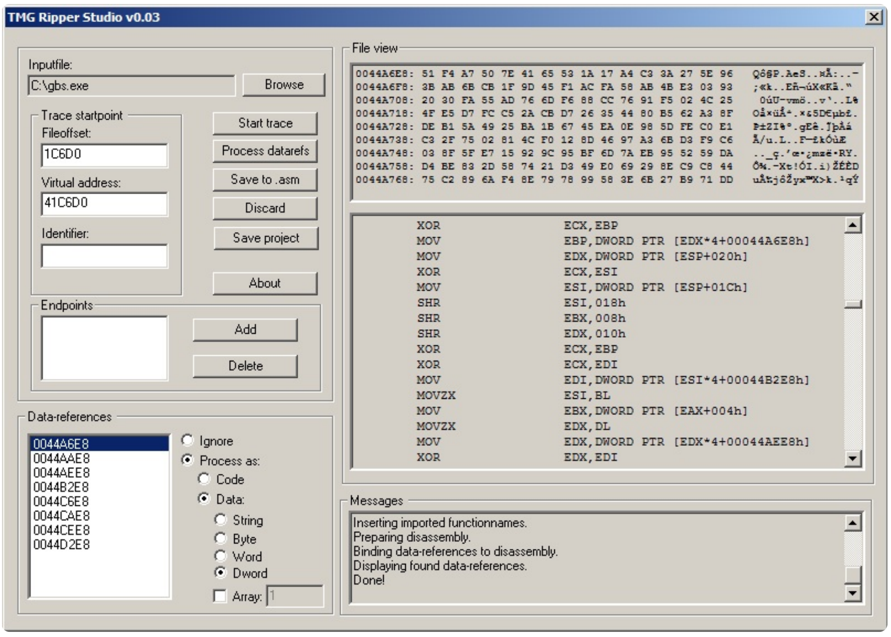
If the function does not have much hiccups above, it should be possible to try ripping it!

Ripping the code

There are apparently two popular utilities to accomplish this:

- Ollydbg with a code-ripping plugin (Ziggy)
- TMG Ripper studio (The Millenium Group)

I chose to use the latter as the setup process was simply downloading, unarchiving and running it.


TMG Ripper Studio v0.03

Inputfile: C:\gbs.exe

Trace startpoint
Fileoffset: 1C6D0
Virtual address: 41C6D0
Identifier:

Endpoints

Data-references

```
0044A6E8
0044AAE8
0044AEE8
0044B2E8
0044C6E8
0044CAE8
0044CEE8
0044D2E8
```

Ignore [✓ Process as:] Code [✓ Data:] String Byte Word [✓ Dword] Array: 1

File view

```
0044A6E8: 51 F4 A7 50 7E 41 65 53 1A 17 A4 C3 3A 27 5E 96   Qô§P.AeS..¤Ã:'.–
0044A6F8: 3B AB 6B CB 1F 9D 45 F1 AC FA 58 AB 4B E3 03 93   ;«kË..Eñ¬úX«Kã."
0044A708: 20 30 FA 55 AD 76 6D F6 88 CC 76 91 F5 02 4C 25    0úU-vmö..v'..L%
0044A718: 4F E5 D7 FC C5 2A CB D7 26 35 44 80 B5 62 A3 8F   Oå×üÅ*.×&5D€µb£.
0044A728: DE B1 5A 49 25 BA 1B 67 45 EA 0E 98 5D FE C0 E1   Þ±ZI%º.gEê.]þÀá
0044A738: C3 2F 75 02 81 4C F0 12 8D 46 97 A3 6B D3 F9 C6   Ã/u.L..F—£kÓùÆ
0044A748: 03 8F 5F E7 15 92 9C 95 BF 6D 7A EB 95 52 59 DA   .._ç.'œ•¿mzë•RY.
0044A758: D4 BE 83 2D 58 74 21 D3 49 E0 69 29 8E C9 C8 44   Ô¾.-Xt!ÓI.i)ŽÉÈD
0044A768: 75 C2 89 6A F4 8E 79 78 99 58 3E 6B 27 B9 71 DD   uÂ‰jôŽyx™X>k.¹qÝ
```

```
          XOR              ECX,EBP
          MOV              EBP,DWORD PTR [EDX*4+00044A6E8h]
          MOV              EDX,DWORD PTR [ESP+020h]
          XOR              ECX,ESI
          MOV              ESI,DWORD PTR [ESP+01Ch]
          SHR              ESI,018h
          SHR              EBX,008h
          SHR              EDX,010h
          XOR              ECX,EBP
          XOR              ECX,EDI
          MOV              EDI,DWORD PTR [ESI*4+00044B2E8h]
          MOVZX            ESI,BL
          MOV              EBX,DWORD PTR [EAX+004h]
          MOVZX            EDX,DL
          MOV              EDX,DWORD PTR [EDX*4+00044AEE8h]
          XOR              EDX,EDI
```

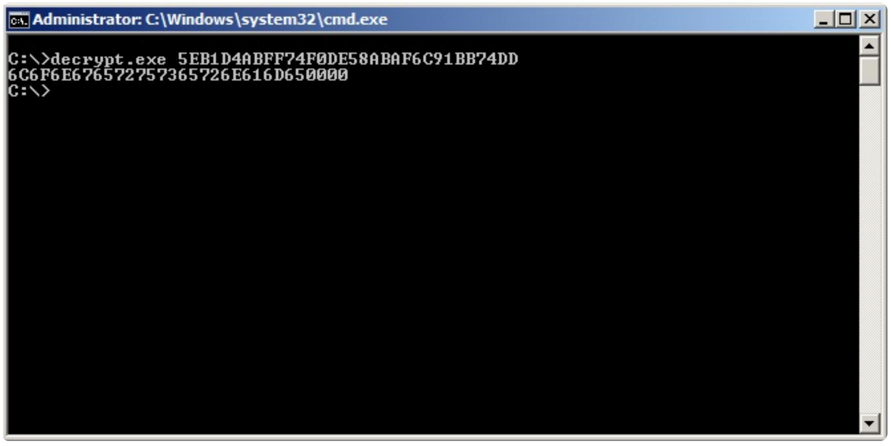
Messages

```
Inserting imported functionnames.
Preparing disassembly.
Binding data-references to disassembly.
Displaying found data-references.
Done!
```

Using TMGRS is straightforward:

1. Specify the path to your binary, and the virtual address of the function to be ripped. (`0x41C6D0` in my case). When *Start trace* is clicked, TMGRS automatically determines the file offset, traces the disassembly and displays data references used by that region of instructions.
2. Review the data references to ensure that the data types are correct (IDA usually makes accurate guesses). Clicking on *Process datarefs* will save those changes.
3. Add an identifier of your choice. This allows the function to be accessed via `call yourIdentifier` later. When you are ready, *Save to asm* and you will have a MASM-compatible rip!

Packaging the code


Administrator: C:\Windows\system32\cmd.exe

```
C:\>decrypt.exe 5EB1D4ABFF74F0DE58ABAF6C91BB74DD
6C6F6E676572757365726E616D650000
C:\>
```

At this point, a (M)ASM file is ready with the desired instructions, although it is incapable of doing anything on its own. Again, there are many ways to use this data, and I chose to use it like a command line utility: by building a wrapper to take in command-line parameters, and return the result via the standard output.

Modifications to the original ASM file include:

- Headers and basic imports
- Arrays in the `.data` segment, required by the ripped function, along with buffers and variables for other accessory functions such as storing and processing of command line parameters
- Helper functions to parse and convert between hex-strings and bytes
- Actual code to load and parse command line parameters, call the ripped function, and print the formatted output

The ASM file that I worked on can be downloaded here. If you intend to hack on it, take note that there is **no** bounds checking and it expects the data to be properly formatted.

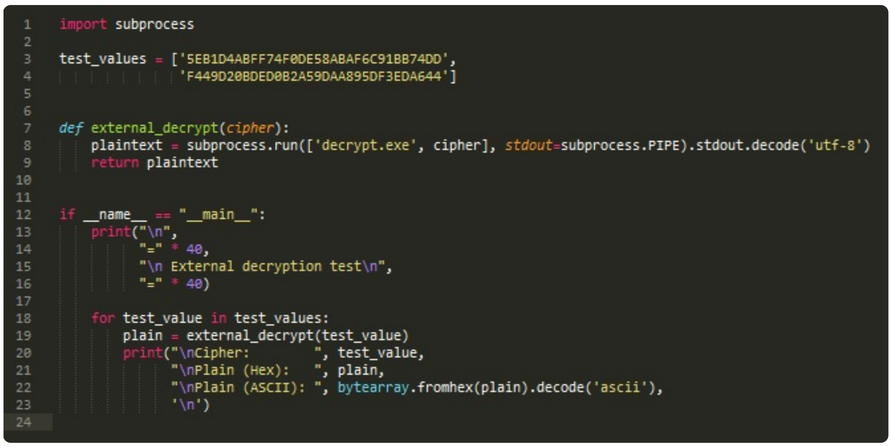
Building the ASM file requires MASM32. Running the below commands with `your_file` replaced with your filename should hopefully result in a working binary in the same folder.

```
c:\masm32\bin\ml /c /coff /Cp your_file.asm
c:\masm32\bin\link /SUBSYSTEM:CONSOLE /LIBPATH:c:\masm32\lib your_file.obj
```

Troubleshooting

Issues may arise from assuming incorrect data types during the ripping process. What I guessed as *DWORDs* turned out to be byte arrays. However the indices were determined at runtime, and I was unsure on how much data it was accessing. I got around this issue by ripping the a large chunk of the memory region which is effective (since unused contiguous data bytes do not affect the program flow) at the expense of a slightly larger binary.

Reusing the function elsewhere



```python
import subprocess

test_values = ['5EB1D4ABFF74F0DE58ABAF6C91BB74DD',
               'F449D20BDED0B2A59DAA895DF3EDA644']


def external_decrypt(cipher):
    plaintext = subprocess.run(['decrypt.exe', cipher], stdout=subprocess.PIPE).stdout.decode('utf-8')
    return plaintext


if __name__ == "__main__":
    print("\n",
          "=" * 40,
          "\n External decryption test\n",
          "=" * 40)

    for test_value in test_values:
        plain = external_decrypt(test_value)
        print("\nCipher:        ", test_value,
              "\nPlain (Hex):   ", plain,
              "\nPlain (ASCII): ", bytearray.fromhex(plain).decode('ascii'),
              '\n')

```

Running this Python script results in the output shown in the first image of this post

The ripped function can now be reliably accessed from other sources as a command line tool. In Python3 with the `subprocess` module, this one-liner runs the binary, then reads and returns the standard output as a string:

```
subprocess.run(['your_binary.exe', 'your_parameters'], stdout=subprocess.PIPE).stdout.decod
e('utf-8')
```

The performance is *could be better*, with each call taking about 9.36ms on my 2012 MacbookPro. It is likely that most of the delay stems from the process creation overhead. However, the goal of ripping and reusing a PE's function elsewhere has been achieved! ^_^

If you are working with Sublime Text, MasmAssembly is a nifty package to add code highlighting to MASM-formatted files

Nerd snipe: If you can figure out what's going on in the decryption function in the earlier ASM file, please let me know!