

Table of Contents

1. INTRODUCTION.....	3
2. INSTALLATION.....	3
3. BASIC CONCEPTS OF THE PLUGIN IDADISCOVER.....	3
3.1. INTERNAL VARIABLES.....	4
3.2. WILDCARDS.....	4
3.2.1. %IDANAMESADDRESS1%.....	4
3.2.2. %IDANAMESCONTENT1%size%.....	6
3.2.3. %IDANAMESCONTENT1%-1%.....	8
3.2.4. %IDANAMESCONTENT1%-2%.....	8
3.3.5. %IMMOPERANDS1%.....	9
4. PLUGIN OPTIONS.....	10
4.1. CONFIG.....	10
4.1.1. CONFIG – CONFIGURATION.....	11
4.1.2. CONFIG – CALL TARGETS.....	12
4.1.3. CONFIG – CALL API NAMES.....	13
4.1.4. CONFIG – ENCRYPTED TEXT.....	13
4.2. SELECT.....	14
4.2.1. SELECT – SELECT KEY 1 (from a visual selection in IDA segments).....	15
4.2.2. SELECT – SELECT KEY 2 (from a visual selection in IDA segments).....	16
4.2.3. SELECT – SELECT CONTENT 1 (from a visual selection in IDA segments).....	16
4.2.4. SELECT – SELECT KEY 1 (from a file).....	16
4.2.5. SELECT – SELECT KEY 2 (from a file).....	16
4.2.6. SELECT – SELECT CONTENT 1 (from a file).....	16
4.2.7. SELECT – SELECT KEY 1 (from IDA segments, specify a range of addresses).....	17
4.2.8. SELECT – SELECT KEY 2 (from IDA segments, specify a range of addresses).....	17
4.2.9. SELECT – SELECT CONTENT 1 (from IDA segments, specify a range of addresses).....	17
4.2.10. SELECT – SEARCH BLOCK SIZE 1.....	17
4.2.11. SELECT – SEARCH STRING (accepts hexadecimal).....	17
4.2.12. SELECT – SELECT KEY 1 (introduce a string into a dialog box).....	18
4.2.13. SELECT – SELECT KEY 2 (introduce a string into a dialog box).....	18
4.2.14. SELECT – SELECT CONTENT 1 (introduce a string into a dialog box).....	18
4.2.15. SELECT – ZEROS TO CONTENT 1.....	18
4.2.16. SELECT – WILDCARD TO KEY 1.....	18
4.2.17. SELECT – WILDCARD TO KEY 2.....	19
4.2.18. SELECT – WILDCARD TO CONTENT 1.....	19
4.2.19. SELECT – PRINT CURRENT SELECTIONS.....	19
4.2.19. SELECT – EDIT CURRENT SELECTIONS.....	20
4.3. SIGNATURES.....	20
4.3.1. SIGNATURES – ANALYSIS YARA RULES.....	20
4.3.2. SIGNATURES – API CRC32 USAGE.....	21
4.4. CRYPTO.....	21
4.4.1. CRYPTO – SEARCH ENCRYPTED TEXT.....	22
4.4.2. CRYPTO – SEARCH STRINGS BUILT IN STACK.....	23
4.4.3. CRYPTO – CALCULATE RC4.....	24
4.4.4. CRYPTO – CALCULATE AES CBC.....	24
4.4.5. CRYPTO – CALCULATE AES ECB.....	25

4.4.6. CRYPTO – CALCULATE XOR.....	25
4.4.7. CRYPTO – CALCULATE MD5.....	25
4.4.8. CRYPTO – CALCULATE SHA256.....	26
4.4.9. CRYPTO – SEARCH STRING ENCRYPTED WITH RC4.....	26
4.4.10. CRYPTO – SEARCH STRING ENCRYPTED WITH AES CBC.....	26
4.4.11. CRYPTO – SEARCH STRING ENCRYPTED WITH AES ECB.....	27
4.4.12. CRYPTO – SEARCH STRING ENCRYPTED WITH XOR.....	27
4.4.13. CRYPTO – WILDCARDS IN CRYPTO OPTIONS.....	28
4.5. LOOPS.....	31
4.6. FUNCTIONS.....	31
4.6.1. FUNCTIONS – STRUCTURES HEURISTIC IDENTIFICATION.....	32
4.6.2. FUNCTIONS – MOST USED FUNCTIONS.....	32
4.6.3. FUNCTIONS – UP-NAME FUNCTIONS.....	33
4.6.4. RESET – UP-NAME FUNCTIONS.....	36
4.6.5. CREATE – FUNCTIONS FROM UNREFERENCED CODE BLOCKS.....	36
4.6.6. SEARCH – SET CANDIDATE WINDOWS API FOR DWORDS.....	36
4.7. EMULATOR.....	37
4.7.1. EMULATOR - CONTENT1 TO EMU MEMORY ADDRESS.....	39
4.7.2. EMULATOR - DWORD TO REGISTER.....	39
4.7.3. EMULATOR - DWORD TO EMU MEMORY ADDRESS.....	40
4.7.4. EMULATOR - INIT EMULATOR STACK.....	40
4.7.5. EMULATOR - SET REGISTER TO RECOVER (RESULTS AFTER EMULATION).....	40
4.7.6. EMULATOR - SET MEMORY ADDRESS TO RECOVER (RESULTS AFTER EMULATION).....	41
4.7.7. EMULATOR - SET START ADDRESS.....	41
4.7.8. EMULATOR - SET END ADDRESS.....	41
4.7.9. EMULATOR - SET MAXIMUM NUMBER OF INSTRUCTIONS TO EMULATE (DEFAULT 1000).....	41
4.7.10. EMULATOR - ADD TYPES OF INSTRUCTION TO SKIP.....	41
4.7.11. EMULATOR - ADD RECOMMENDED INSTRUCTIONS TYPES TO SKIP.....	42
4.7.12. EMULATOR - SET FLAG: MAP INVALID ADDRESSES ON ACCESS VIOLATION?.....	42
4.7.13. EMULATOR - SET FLAG: MAP FULL IDA SEGMENTS CODE IN EMULATOR ADDRESS SPACE?.....	42
4.7.14. EMULATOR - SET FLAG: ENABLE DEBUG MODE?.....	43
4.7.15. EMULATOR - SET FLAG: ENABLE VERBOSE OUTPUT?.....	43
4.7.16. EMULATOR - SET FLAG: EMULATOR RESULTS TO IDA COMMENTS?.....	43
4.7.17. EMULATOR - INTRODUCE WILDCARD TO EMULATOR MEMORY ADDRESS... ..	44
4.7.18. EMULATOR - INTRODUCE WILDCARD TO EMULATOR REGISTER.....	45
4.7.19. EMULATOR - START EMULATION.....	46
4.7.20. EMULATOR - RESET EMULATION CONFIG.....	46
4.7.21. EMULATOR - EDIT EMULATION CONFIG.....	46
4.7.22. EMULATOR - SHOW CURRENT EMULATION CONFIG.....	47
4.7.23. EMULATOR - EMULATE FROM CURRENT ADDRESS WITH DEFAULT CONFIG (CTRL-SHIFT-ALT-E).....	47
4.7.24. EMULATOR - EMULATE FROM WILDCARD MATCHES WITH DEFAULT CFG... ..	47
4.7.25. EMULATOR – WILDCARDS IN EMULATOR OPTIONS.....	51

1. INTRODUCTION

IdaDiscover is an IDA python plugin whose functionality is mostly focused on analysis of malware, but not necessarily restricted to this, it could be useful for other kind of tasks.

I started to implement this plugin long time ago, and all the options and functionalities of the plugin were born of my own necessity while I was analyzing malware samples. Every time I needed to automatize a task that is frequently performed in analysis of malware, I automatized this task as an option of the plugin.

Because of this, I feel that the plugin implements options and functionalities that are going to be really useful in the process of analyzing a malware, because it was born from my own necessity during years as malware analyst. It has made my life much easier and for this reason I decided to share it with other malware analysts.

2. INSTALLATION

IdaDiscover needs some external libraries to work properly:

- future (should be installed automatically by the plugin if it is not already installed)
- yara-python (should be installed automatically by the plugin if it is not already installed)
- py_aho_corasick (should be installed automatically by the plugin if it is not already installed)
- pycrypto (should be installed automatically by the plugin if it is not already installed)
- unicorn (should be installed automatically by the plugin if it is not already installed)
- capstone (should be installed automatically by the plugin if it is not already installed)
- revealPE (git submodule from <https://github.com/vallejocc/RevealPE>)
- pefile (git submodule from <https://github.com/erocarrera/pefile>)

One of the options of idaDiscover could be used to search patterns in the code or data with yara rules. You could use your own yara rules. By default, the yaras' folder is a git submodule from <https://github.com/Yara-Rules/rules>.

So, to install the plugin, you would only need to clone the git repository and its submodules, and load the plugin on IDA (it should install the python modules that it will need).

3. BASIC CONCEPTS OF THE PLUGIN IDADISCOVER

3.1. INTERNAL VARIABLES

IdaDiscover uses a set of internal variables, depending on the option that you choose, it will use the content of a subset of these internal variables. For example, if you choose the option 'Crypto' → 'Calculate Xor', the variables Key1 and Content1 should have been filled, because that option will xor both buffers (introduced into these variables with options of the 'Select' menu) and will keep the result in a file.

In the section 4.2 it is explained how to fill the values for these variables.

In the next sections the idaDiscover options and functionalities will be explained and for each option that uses these internal variables, it will be explained what they should contain.

List of internal variables:

- Key1
- Key2
- Content1
- SearchSize
- SearchString

3.2. WILDCARDS

Another important mechanism in idaDiscover are the wildcards. They are special values that can be introduced into the internal variables previously explained or other configuration fields like memory ranges in the emulator memory address space or the emulator registers.

Wildcards work with some options of idaDiscover, and basically they mean that the chosen idaDiscover operation (the chosen option) should be repeated N times, replacing the wildcard by a set of values from the IDA disassembly (a different set values of depending on the introduced wildcard).

In the next sub-sections, each type of wildcard is described.

3.2.1. %IDANAMESADDRESS1%

As IDA dissassembler's user probably knows, IDA identifies objects in the analyzed binary, and sets names for these objects. For example, these two buffers:

```

db 0
db 0
db 0
db 0
db 0
unk_1545970 db 22h ; " ; DATA XREF: sub_150B350+11fo
; sub_1510DD0+20F3fo
db 23h ; #
db 0Fh
db 39h ; 9
db 3Ch ; <
db 50h ; P
db 12h
db 08h
db 79h ; y
db 4
db 1
db 0Eh
db 50h ; P
db 26h ; &
db 0Eh
db 5Fh ; _
db 31h ; 1
db 51h ; Q
unk_1545982 db 35h ; 5 ; DATA XREF: sub_150B350+42fo
; sub_1510DD0+10E5fo
db 39h ; 9
db 18h
db 39h ; 9
db 24h ; $
db 53h ; S
db 13h
db 70h ; ..

```

If the wildcard %IDANAMESADDRESS1% is introduced into some IDA internal variables, the chosen option will be repeated and executed multiple times, and for each time that the option is executed, the wildcard will be replaced by addresses of names of objects identified by IDA disassembler.

For example, in the previous example, imagine that we have started an emulation (emulator options will be explained later) and register EAX is configured with %IDANAMESADDRESS1% wildcard. In this case, the emulation will be launched N times (one for each name in IDA), and for each emulation, the register EAX will be filled with the address of a different name of the IDA disassembly. In one of the emulations, EAX value will be 0x1545970 (because of the name unk_1545970), and in the next one EAX value will be 0x1545982 (because of unk_1545982), etc...

It exists two more addresses wildcards: %IDANAMESADDRESS2%, %IDANAMESADDRESS3%, just in case you would need to bruteforce more than one parameter.

For example, lets imagine a function that receives two arguments, and a piece of code calls this function in this way:

```
addr_my_code:
    push eax
    push ebx
    call my_function
```

Imagine that we know that this function's parameters are an encrypted buffer and a key, but we don't know where is the key and the encrypted buffer in the disassembly. So we decide to emulate the function and for each emulation we want that the addresses of a name of the IDA disassembly was introduced into EAX, and the address of another name in EBX.

If we want to do that, this won't work:

```
EAX = %IDANAMESADDRESS1%
EBX = %IDANAMESADDRESS1%
```

Because in this case, the same address will be pushed two times for the function, because EAX and EBX will be replaced by the same address.

However, if we do this:

```
EAX = %IDANAMESADDRESS1%
EBX = %IDANAMESADDRESS2%
```

In this case, idaDiscover will loop over the full list of IDA names in the disassembly two times, one for %IDANAMESADDRESS1% and another one for %IDANAMESADDRESS2%. Something like this:

```
for address_name1 in GetIDANames():
    for address_name2 in GetIDANames():
        emuconfig.registers.EAX = address_name1
        emuconfig.registers.EBX = address_name2
        emulate_code(addr_my_code, emuconfig)
```

It is important to be careful when we are working with wildcards. Imagine that we start an emulation with a limit of 0x10000 instructions (we will talk about the emulator and its options later) and two wildcards, and imagine that IDA disassembly contains 300 names for example. The emulation would be launched 300*300 times, emulating 0x10000 instructions each time, and it will take long time to finish.

3.2.2. %IDANAMESCONTENT1%size%

Before reading this section, it is recommendable to understand well the wildcard described in the previous section 3.2.1.

This wildcard is similar to %IDANAMESADDRESS1%, idaDiscover will loop over all the names that IDA dissassembler has set on all the objects that it has identified, however instead of replacing the wildcard with the address of the object, it will read the content of that address (the size of data to be read is specified in the own wildcard) and the wildcard will be replaced by that content. For example, if we have the same buffers of the section 3.2.1:

```

unk_1545970    uu      0
               db  22h ; "                ; DATA XREF: sub_150B350+11↑o
                                           ; sub_1510DD0+20F3↑o
               db  23h ; #
               db  0Fh
               db  39h ; 9
               db  3Ch ; <
               db  50h ; P
               db  12h
               db  0Bh
               db  79h ; y
               db   4
               db   1
               db  0Eh
               db  50h ; P
               db  26h ; &
               db  0Eh
               db  5Fh ; _
               db  31h ; 1
               db  51h ; Q
unk_1545982    db  35h ; 5                ; DATA XREF: sub_150B350+42↑o
                                           ; sub_1510DD0+10E5↑o
               db  39h ; 9
               db  1Bh
               db  39h ; 9
               db  24h ; $
               db  53h ; S
               ..  --

```

Lets suppose we have introduced this wildcard into the variable Content1:

%IDANAMESCONTENT1%s20%:

And lets suppose that we have read another buffer into the variable Key1 (using the ‘Select’ menu that will be explained in the next sections).

And now we execute the option ‘Crypto’ → ‘Calculate Xor’.

If we do this, the option ‘Calculate Xor’ will be executed N times (one time for each IDA disassembly name), and it will xor the content introduced into Key1 with 20 bytes read from each IDA name address. For example, when the name unk_1545970 is reached while idaDiscover loops over the IDA names, this buffer will be introduced into Content1:

“\x22\x23\x0F\x39\x3C\x50\x12\x0B\x79\x04\x01\x0E\x50\x26\x0E\x5F\x31\x51\x35\x39”

And this buffer will be xored with the content of Key1.

3.2.3. %IDANAMESCONTENT1%-1%

Before reading this section, it is recommendable to understand well the wildcards described in the previous section 3.2.2.

This wildcard is very similar to the previous one, %IDANAMESCONTENT1%size%.

Sometimes, we would like to loop over all the IDA names and replace the wildcard with the contents at that addresses, but we don't want a specific fixed size.

For this reason it was born this wildcard. For each IDA name, it will read the content at the current name, and it will read all the bytes until reaching the next IDA name. For example, in the capture of the previous sections:

```
unk_1545970  db  22h ; "                ; DATA XREF: sub_150B350+11↑o
                                     ; sub_1510DD0+20F3↑o
db  23h ; #
db  0Fh
db  39h ; 9
db  3Ch ; <
db  50h ; P
db  12h
db  0Bh
db  79h ; y
db   4
db   1
db  0Eh
db  50h ; P
db  26h ; &
db  0Eh
db  5Fh ; _
db  31h ; 1
db  51h ; Q
unk_1545982  db  35h ; 5                ; DATA XREF: sub_150B350+42↑o
                                     ; sub_1510DD0+10E5↑o
```

In this case, when idaDiscover loops over the IDA names, for the name unk_1545970, it will read the buffer until reaching the next name unk_1545982:

“\x22\x23\x0F\x39\x3C\x50\x12\x0B\x79\x04\x01\x0E\x50\x26\x0E\x5F\x31\x51”

3.2.4. %IDANAMESCONTENT1%-2%

Before reading this section, it is recommendable to understand well the wildcards described in the previous section 3.2.3.

This wildcard works exactly like the wildcard described at section 3.2.3, the only difference is that once a buffer between names is read, trailing zeros are removed. For example:

```
unk_15459F0    db      0
               db    15h                ; DATA XREF: sub_15083D0+23F↑o
               db    3Eh ; >
               db      0
               db    23h ; #
               db    2Ah ; *
               db    58h ; X
               db      3
               db    54h ; T
               db      7
               db      4
               db    0Ch
               db    18h
               db    4Dh ; M
               db    2Bh ; +
               db    13h
               db    48h ; H
               db    7Bh ; {
               db      1
               db      1
               db    3Bh ; ;
               db    26h ; &
               db    55h ; U
               db    0Eh
               db    79h ; y
               db      0
               db      0
               db      0
               db      0
               db      0
               db      0
               db      0
unk_1545A10    db    0Eh                ; DATA XREF: sub_15083D0+279↑o
               db    7Ch ; |
```

In this case, if the wildcard %IDANAMESCONTENT1%-2% is used, when idaDiscover loops over the IDA names and reaches the name unk_15459F0, it will replace the wildcard with the following buffer:

```
"\x15\x3E\x00\x23\x2A\x58\x03\x54\x07\x04\x0C\x18\x4D\x2B\x13\x48\x7B\x01\x01\x3B\x26\x55\x0E\x79"
```

3.3.5. %IMMOPERANDS1%

Before reading this section, it is recommendable to understand well the wildcard described in the previous section 3.2.1.

Maybe this wildcard is not going to be used so frequently like the other wildcards, however sometimes it could be interesting.

When this wildcard is set, idaDiscover will loop all the instructions of the IDA disassembly. When an instruction with an immediate value is found, the wildcard is replaced by that immediate value and the chosen option is executed.

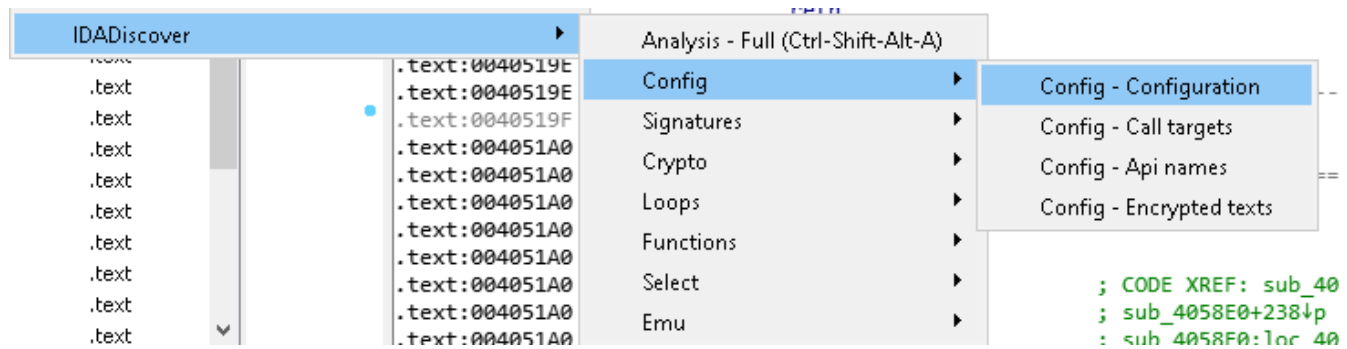
For example, this piece of code:

```
-----
.text:01541390                                     ; -----
.text:01541390                                     ; sub_153E370+39↑p
.text:01541390                                     ; sub_153EF40+32↑p
.text:01541390                                     ;
.arg_0 = dword ptr 8
.arg_4 = dword ptr 0Ch
.text:01541390 55                                push    ebp
.text:01541391 89 E5                            mov     ebp, esp
.text:01541393 8B 45 08                            mov     eax, [ebp+arg_0]
.text:01541396 B9 00 01 00 00                            mov     ecx, 100h
.text:01541398 23 0D 3C 79 54 01                        and     ecx, dword_154793C
.text:015413A1 3B 45 0C                            cmp     eax, [ebp+arg_4]
.text:015413A4 0F 94 C0                            setz    al
.text:015413A7 83 C1 80                            add     ecx, 0FFFFFFF80h
.text:015413AA 89 0D 30 7E 54 01                        mov     dword_1547E30, ecx
.text:015413B0 5D                                pop     ebp
.text:015413B1 C3                                retn
.text:015413B1                                sub_1541390 endp
.text:015413B1
.text:015413B1                                ; -----
.text:015413B2 90 90 90 90 90 90 90 90 90 90+                align 10h
.text:015413C0
.text:015413C0                                ; ===== S U B R O U T I N E =====
.text:015413C0
.text:015413C0                                ; Attributes: bp-based frame
.text:015413C0
```

When idaDiscover loops the instructions and reaches the instruction at 0x1541396, the wildcard will be replaced by 0x100, and when the instruction at 0x15413A7, the wildcard will be replaced by 0xFFFFFFFF80. And in both cases, the chosen option will be executed after replacing the wildcard in the variable or the emulator memory or register, or whatever...

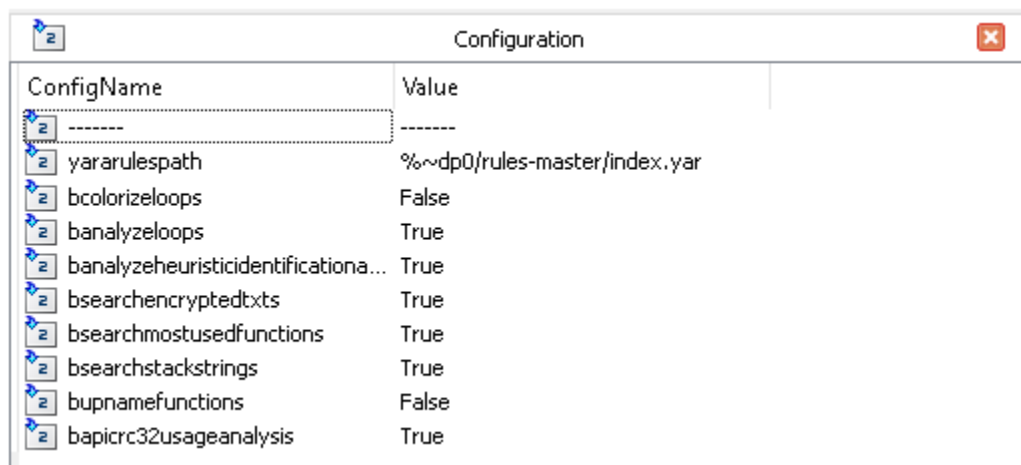
4. PLUGIN OPTIONS

4.1. CONFIG



4.1.1. CONFIG – CONFIGURATION

When this menu option is clicked, a IDA window is open where it is possible to configure some general parameters of the plugin:



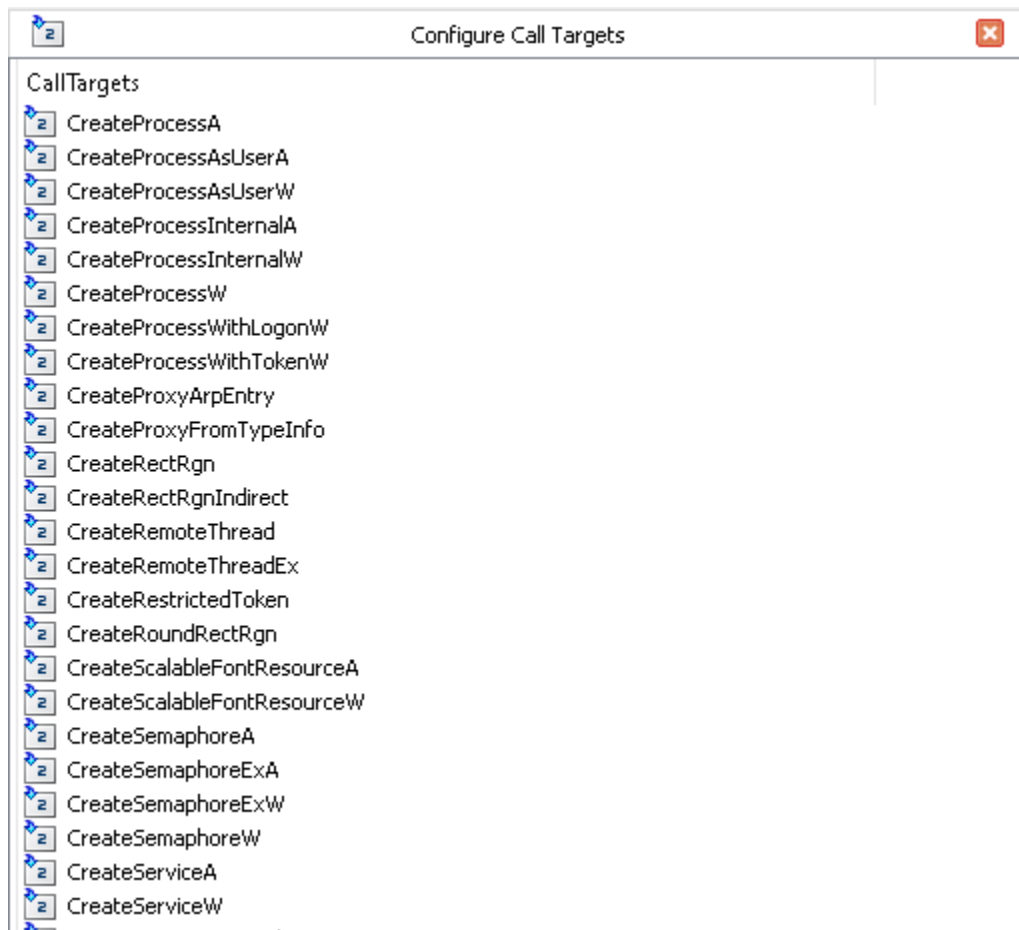
- yararulespath: path to the yara rules file (the string %~dp0 would be replaced by the plugin directory path).
- bcolorizeloops: enable loops coloring in the analysis of loops.
- banalyzeloops: configure if full analysis option should include analysis of loops.
- banalyzeheuristicidentificationalgorithms: configure if full analysis option should include heuristic analysis and identification of well-known algorithms.
- bsearchencryptedtxts: configure if full analysis option should include x-raying analysis of encrypted strings with revealPE tools (<https://github.com/vallejoc/RevealPE>).
- bsearchmostusedfunctions: configure if full analysis option should include analysis of functions to sort them based on number of times that have been used.
- bsearchstackstrings: configure if full analysis option should include analysis of strings built on stack.

- `bupnamefunctions`: configure if full analysis option should include up-name functionality (it will be explained later).
- `bapirc32usageanalysis`: configure if full analysis option should include analysis of usage of `crc32` of API names.

4.1.2. CONFIG – CALL TARGETS

Internally, `idaDiscover` uses a list of well-known names where disassembly's calls could be calling. Usually api names, for example, "`call ds:RegSetValueExW`" (but it could be any other name). Mainly, this list of names is used by the option `Up-Names` that will be explained later.

With this configuration option, a window will appear where it is possible to customize the list of call-targets to be used by `idaDiscover`:

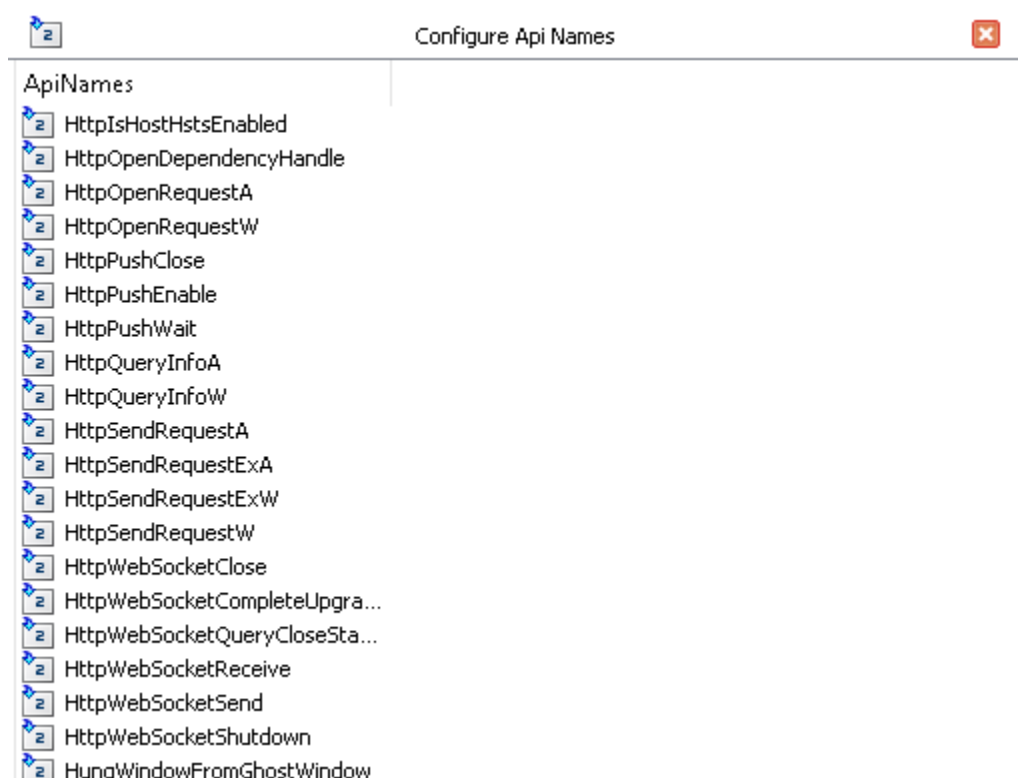


By default, a long list of `apinames` is configured here. But maybe you could be interested in introduce a custom name of your disassembly (for example, "`strings_decryptor`" if you want that calls of the type "`call strings_decryptor`" are considered for the `Up-Names` option or other options that could be implemented in the future).

4.1.3. CONFIG – CALL API NAMES

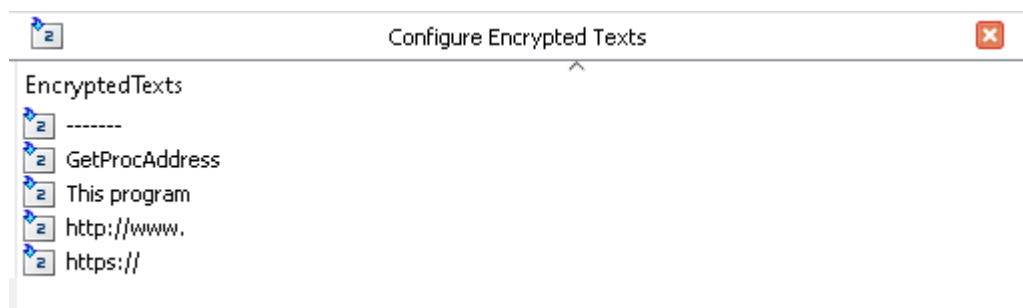
This list could look similar to the Call-Targets list (because Call-Targets list is composed mainly by names of apis), however Call-Targets list is a list of names where call instructions could be calling (for example, string_decryptor for “call string_decryptor” or crc32 for “call crc32”) meanwhile the list of api names are only api names, and this list is used in different options of idaDiscover (for example, it is used in the option “Apis Crc32 usage”, that search for instructions using values that are crc32 of api names).

In this configuration option a window is open where it is possible to customize the list of api names used by idaDiscover:



4.1.4. CONFIG – ENCRYPTED TEXT

Another list that is used internally by idaDiscover is the list of encrypted texts. With this option, a window is open where it is possible to customize this list of strings:

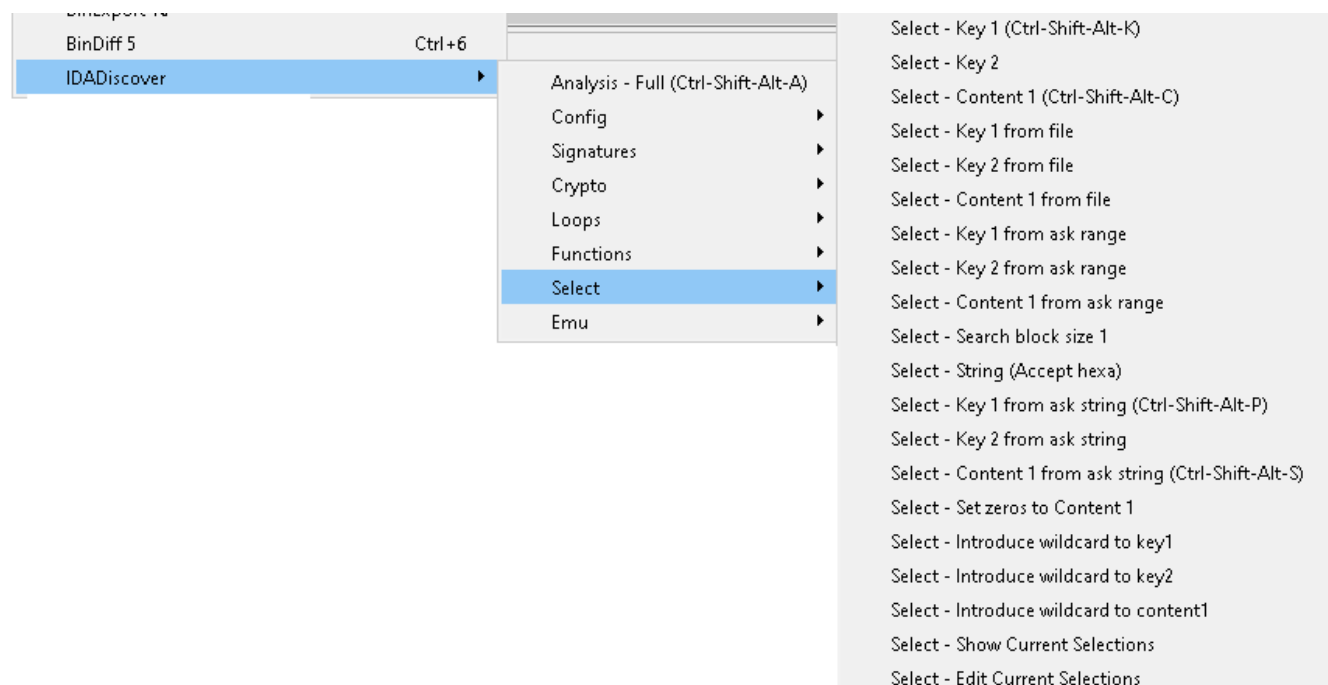


For the moment, these strings are used in the option Analysis - Encrypted text, that performs a veery simple cryptoanalysis (under the hood, it is using [RevealPE tool](#)) to try to find candidate encrypted strings that could be strings of the list once they are encrypted (with simple encryption algorithms), and try to get the decryption method and key. It will be explained later.

With this configuration option you could customize the list of target strings that are going to be used as plaintext in the cryptoanalysis.

4.2. SELECT

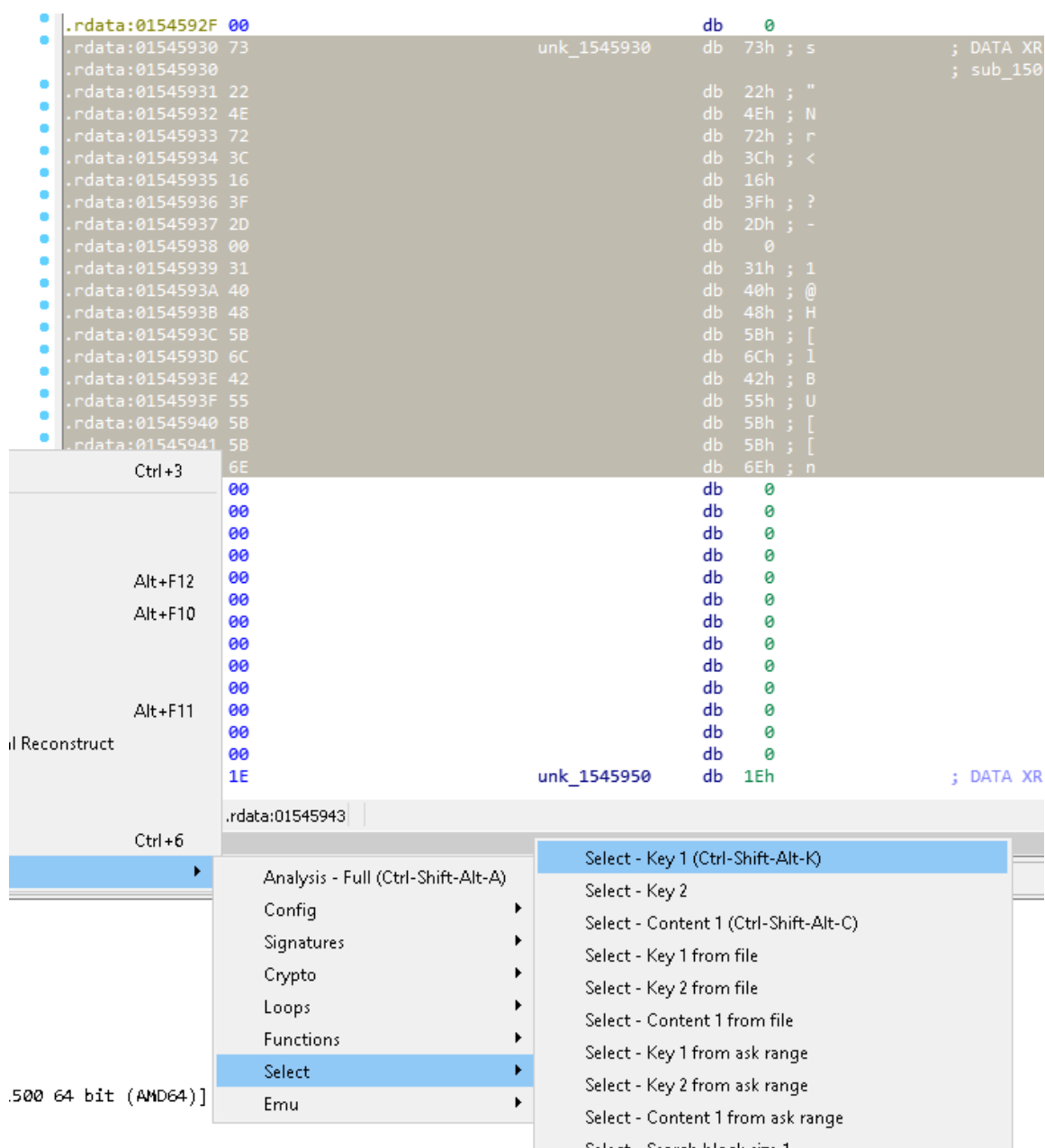
Select menu is very important, because with the options of this menu you can fill some different internal variables used internally by idaDiscover (content1, key1, key2,...):



And these variables will be very important because they are used to introduce data for the other options (crypto, emulator, etc...), as we will see later. For the moment, in this section let's learn how to select (fill) these variable from different sources.

4.2.1. SELECT – SELECT KEY 1 (from a visual selection in IDA segments)

This menu's option can be used to store in the idaDiscover's internal variable Key1, the content selected in IDA disassembly or hexa dump:



The shortcut for this option is Ctrl+Shift+Alt+K.

4.2.2. SELECT – SELECT KEY 2 (from a visual selection in IDA segments)

This option works like the option explained in the previous section 4.2.1, but it stores the selected content into Key2.

4.2.3. SELECT – SELECT CONTENT 1 (from a visual selection in IDA segments)

This option works like the option explained in the previous section 4.2.1, but it stores the selected content into Content1.

4.2.4. SELECT – SELECT KEY 1 (from a file)

This option works like the option explained in the previous section 4.2.1, but it asks for a path to a file, then the content of the file is read, and it is stored into the variable Key1.

4.2.5. SELECT – SELECT KEY 2 (from a file)

This option works like the option explained in the previous section 4.2.1, but it asks for a path to a file, then the content of the file is read, and it is stored into the variable Key2.

4.2.6. SELECT – SELECT CONTENT 1 (from a file)

This option works like the option explained in the previous section 4.2.1, but it asks for a path to a file, then the content of the file is read, and it is stored into the variable Content1.

4.2.7. SELECT – SELECT KEY 1 (from IDA segments, specify a range of addresses)

This option works like the option explained in the previous section 4.2.1, however an initial addresses and an ending address is asked (instead of getting the selected area in IDA disassembly), and the block of code or data between both addresses is stored into Key1.

4.2.8. SELECT – SELECT KEY 2 (from IDA segments, specify a range of addresses)

This option works like the option explained in the previous section 4.2.1, however an initial addresses and an ending address is asked (instead of getting the selected area in IDA disassembly), and the block of code or data between both addresses is stored into Key2.

4.2.9. SELECT – SELECT CONTENT 1 (from IDA segments, specify a range of addresses)

This option works like the option explained in the previous section 4.2.1, however an initial addresses and an ending address is asked (instead of getting the selected area in IDA disassembly), and the block of code or data between both addresses is stored into Content1.

4.2.10. SELECT – SEARCH BLOCK SIZE 1

Dispite the most used idaDiscover internal variables are Key1, Key2 and Content1, another internal variables are used for some specific options. This is the case of the 'Search Block Size 1' variable, that is used for the moment in the four 'Search' options under 'Crypto' menu (they will be explained later).

With this select option, a menu asking for a size will appear and the size will be stored in this internal variable.

4.2.11. SELECT – SEARCH STRING (accepts hexadecimal)

Again, this option fills another internal variable of idaDiscover, 'Search String 1', that is used for the moment in the four 'Search' options under 'Crypto' menu (they will be explained later).

4.2.12. SELECT – SELECT KEY 1 (introduce a string into a dialog box)

This option works like the option explained in the previous section 4.2.1, but it asks for a string (this string accepts hexadecimal values encoded in this way: \x00,\x01,...), then the content of the string is stored into the variable Key1.

4.2.13. SELECT – SELECT KEY 2 (introduce a string into a dialog box)

This option works like the option explained in the previous section 4.2.1, but it asks for a string (this string accepts hexadecimal values encoded in this way: \x00,\x01,...), then the content of the string is stored into the variable Key2.

4.2.14. SELECT – SELECT CONTENT 1 (introduce a string into a dialog box)


This option works like the option explained in the previous section 4.2.1, but it asks for a string (this string accepts hexadecimal values encoded in this way: \x00,\x01,...), then the content of the string is stored into the variable Content1.

4.2.15. SELECT – ZEROS TO CONTENT 1

This option works like the option explained in the previous section 4.2.1, but it asks for a number of zeros, and it fills the variable Content1 with that number of zeros.

4.2.16. SELECT – WILDCARD TO KEY 1

With this option it is possible to introduce an idaDiscover wildcard (wildcards have been introduced in the section 3 and it will be explained in detail for each option where a wildcard can be set) in idaDiscover internal Key1:

 Please enter a string

Enter text value -

- Wildcard -

If you insert wildcards, some operations (for example crypto options) will be launched N times.

For each time, key1 will be filled with IDA data at different addresses (where IDA names were set) or imm values, depending on the inserted wildcard.

CAREFUL setting multiple wildcards could cause the operation to take loooong time.

Accepted values

%IDANAMESCONTENT1%size% (the content at names of the IDA disassemble, size indicates the amount of bytes to be copied)

%IDANAMESCONTENT2%size% (identical like %IDANAMESCONTENT1%size%)

%IDANAMESCONTENT1%-1% (identical to %IDANAMESCONTENT1%size%, but the size is automatically set to the size of the block)

%IDANAMESCONTENT1%-2% (identical to %IDANAMESCONTENT1%-1%, but trailing zeroes are removed from the read data)

%IMMOPERANDS1% (imm operands among all the instructions of the IDA disassembly)

%IMMOPERANDS2% (identical like %IMMOPERANDS1%)

4.2.17. SELECT – WILDCARD TO KEY 2

With this option it is possible to introduce an idaDiscover wildcard (wildcards have been introduced in the section 3 and it will be explained in detail for each option where a wildcard can be set) in idaDiscover internal Key2.

4.2.18. SELECT – WILDCARD TO CONTENT 1

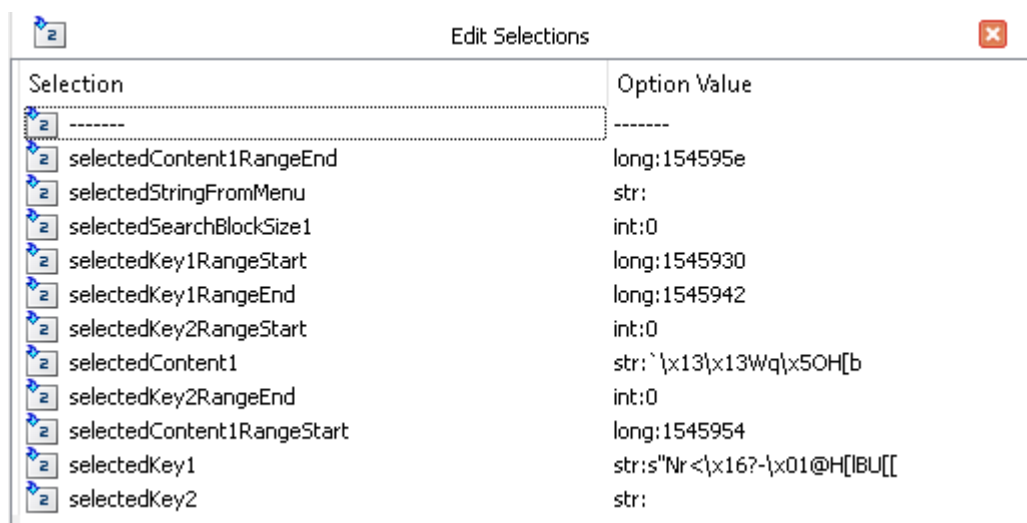
With this option it is possible to introduce an idaDiscover wildcard (wildcards have been introduced in the section 3 and it will be explained in detail for each option where a wildcard can be set) in idaDiscover internal Content1.

4.2.19. SELECT – PRINT CURRENT SELECTIONS

With this option it is possible to print the values of the idaDiscover internal variables.

4.2.19. SELECT – EDIT CURRENT SELECTIONS

This option opens a new window where the list of idaDiscover internal variables is shown with their values. In this window it is possible to edit the values of the variables:



4.3. SIGNATURES

4.3.1. SIGNATURES – ANALYSIS YARA RULES

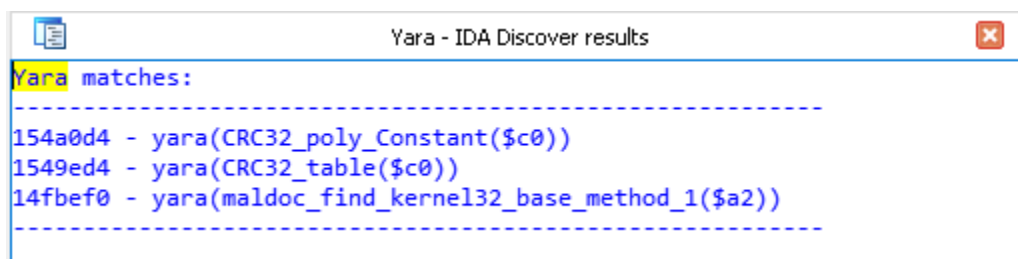
As it was explained in the section 4.1.1, in the main config of idaDiscover is stored a parameter `yararulespath`.

By default, this parameter is pointing to `<idaDiscover directory>/rules-master/index.yar`, and this folder `rules-master` is a submodule from this wellknown yara collection github repository:

<https://github.com/Yara-Rules/rules>

However this parameter could be customized to point to the own yaras of the idaDiscover's user.

When this option of idaDiscover is used, patterns of the yara rules are searched in the code and data of the segments of the IDA disassembly. A new window with the results is open when the analysis finishes. For example:



Additionally, comments are set at addresses of code or data where the matches happened:

.data:01549ED0 00 00 00 00	dword_1549ED0 dd 0	; DATA XREF: crc32+6C↑w
.data:01549ED0		; sub_14F4590+14↑r
.data:01549ED0		; sub_14F4590+1C↑w
.data:01549ED4 00 00 00 00	dword_1549ED4 dd 0	; DATA XREF: crc32+4F↑w
.data:01549ED4		; sub_14F4600+3C↑r
.data:01549ED4		; yara(CRC32_table(\$c0))
.data:01549ED8 96	db 96h ; -	
.data:01549ED9 30	db 30h ; 0	
.data:01549EDA 07	db 7	
.data:01549EDB 77	db 77h ; w	
.data:01549EDC 2C	db 2Ch ; ,	
.data:01549EDD 61	db 61h ; a	
.data:01549EDE 0E	db 0Eh	
.data:01549EDF EE	db 0EEh ; i	
.data:01549EE0 BA	db 0BAh ; e	
.data:01549EE1 51	db 51h ; Q	
.data:01549EE2 09	db 9	
.data:01549EE3 99	db 99h ; ™	
.data:01549EE4 19	db 19h	

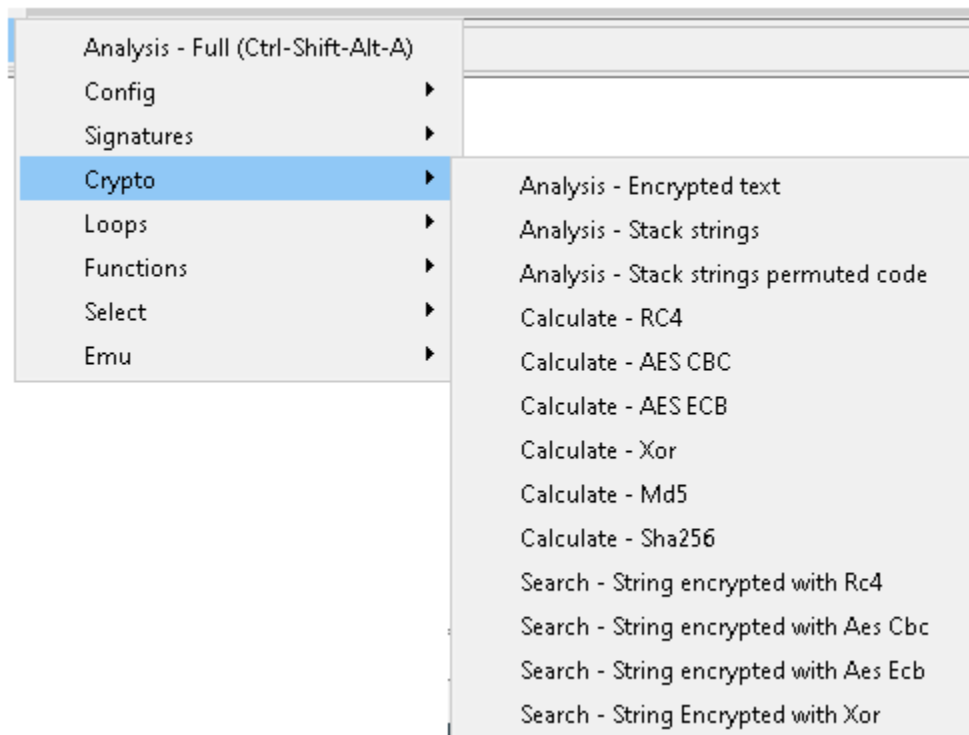
4.3.2. SIGNATURES – API CRC32 USAGE

We have already talked about this option in section 4.1.2 (read this section to know how to configure api names used to search crc32).

Basically, this option search code and data of the IDA disassembly for values (in data dwords, instructions' immediates, etc...) matching crc32 of api names. A window is open with the searching results and a comment will be set at the address of the code or data where the match happened.

4.4. CRYPTO

Crypto menu contains some options related to cryptography algorithms:

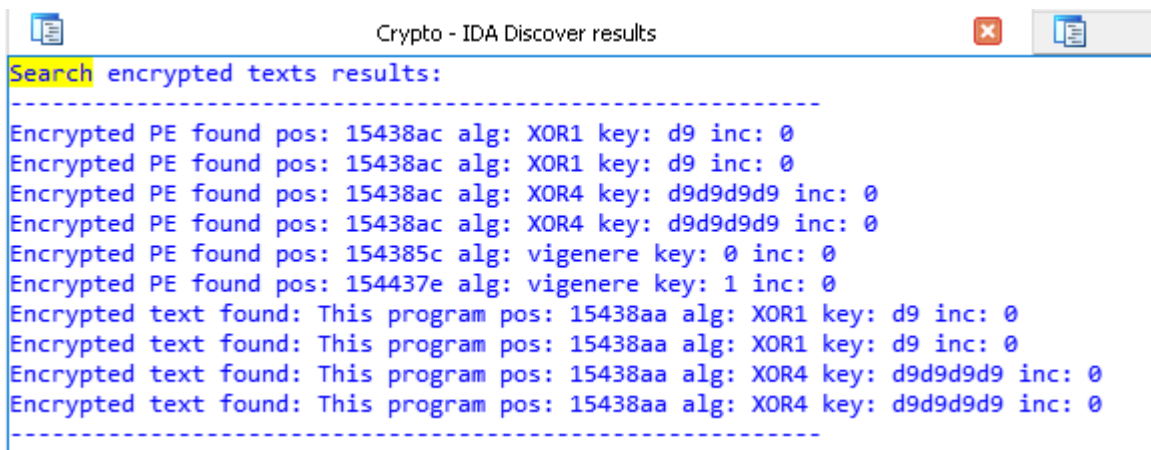


4.4.1. CRYPTO – SEARCH ENCRYPTED TEXT

This option was already introduced in the section 4.1.4, where it was explained how to modify the default configuration and add o remove encrypted strings to search.

When we execute this option, the set of configured string is searched, trying to find encrypted forms (with simple encryption algorithms) of these strings. For this purpose it is using veeery simple cryptanalysis methods (my [RevealPE tool](#) is used under the hood, I recommend you to read [this article](#) to understand the methods used to find the encrypted strings).

Once this option is executed and finished, a new window is open with the results:



4.4.2. CRYPTO – SEARCH STRINGS BUILT IN STACK

This option will search for strings built in the stack. For example, the following piece of code:

```
mov     ecx, [ebp+var_4]
call    unknown_libname_54 ; BDS 2005-20
test    eax, eax
jz      loc_414CE8
mov     [ebp+var_24], 786F6276h
mov     [ebp+var_20], 76726573h
mov     [ebp+var_1C], 2E656369h
mov     [ebp+var_18], 657865h
lea     eax, [ebp+var_10]
lea     edx, [ebp+var_24]
call    unknown_libname_41 ; BDS 2005-20
mov     [ebp+var_24], 68747970h
mov     [ebp+var_20], 652E6E6Fh
mov     [ebp+var_1C], 6578h
lea     eax, [ebp+var_14]
lea     edx, [ebp+var_24]
call    unknown_libname_41 ; BDS 2005-20
mov     eax, [ebp+var_4]
```

If we run this option with the sample owning this code, a new window will be open with these strings built in stack and some comments will be added to the lines where the strings are build:

Search encrypted texts results:

Strings constructed in stack:

sub_414BF0 - 414c2b - vboxservice.
sub_414BF0 - 414c52 - python.e
sub_414E40 - 414ebf - myapp.ex
sub_414E40 - 414eec - self.exe

All the candidate pieces for strings constructed in stack:

414c2b - vbox
414c32 - serv
414c39 - ice.
414c52 - pyth
414c59 - on.e
414e99 - t.ex
414ebf - myap
414ec6 - p.ex
414eec - self
414ef3 - .exe

```
call    unknown_libname_54 ; BDS 2005-21
test    eax, eax
jz       loc_414CE8
mov     [ebp+var_24], 786F6276h ; vbox
mov     [ebp+var_20], 76726573h ; serv
mov     [ebp+var_1C], 2E656369h ; ice.
mov     [ebp+var_18], 657865h
lea     eax, [ebp+var_10]
lea     edx, [ebp+var_24]
call    unknown_libname_41 ; BDS 2005-21
mov     [ebp+var_24], 68747970h ; pyth
mov     [ebp+var_20], 652E6E6Fh ; on.e
mov     [ebp+var_1C], 6578h
lea     eax, [ebp+var_14]
lea     edx, [ebp+var_24]
call    unknown_libname_41 ; BDS 2005-21
```

4.4.3. CRYPTO – CALCULATE RC4

Before reading this section, it is important to understand the idaDiscover internal variables, explained in section 3.1, and the Select menu, explained in the section 4.2.

This option will decrypt with RC4 the content of the internal variable Content1 with the key stored in the internal variable Key1. Then, a file path must be chosen where the decrypted content will be kept.

It is possible to use wildcards with this option. Take a look at the section 4.4.13 where it is explained how to use wildcards with the crypto options.

4.4.4. CRYPTO – CALCULATE AES CBC

Before reading this section, it is important to understand the idaDiscover internal variables, explained in section 3.1, and the Select menu, explained in the section 4.2.

This option will decrypt with AES CBC the content of the internal variable Content1 with the key stored in the internal variable Key1 and IV stored in Key2. Then, a file path must be chosen where the decrypted content will be kept.

It is possible to use wildcards with this option. Take a look at the section 4.4.13 where it is explained how to use wildcards with the crypto options.

4.4.5. CRYPTO – CALCULATE AES ECB

Before reading this section, it is important to understand the idaDiscover internal variables, explained in section 3.1, and the Select menu, explained in the section 4.2.

This option will decrypt with AES ECB the content of the internal variable Content1 with the key stored in the internal variable Key1. Then, a file path must be chosen where the decrypted content will be kept.

It is possible to use wildcards with this option. Take a look at the section 4.4.13 where it is explained how to use wildcards with the crypto options.

4.4.6. CRYPTO – CALCULATE XOR

Before reading this section, it is important to understand the idaDiscover internal variables, explained in section 3.1, and the Select menu, explained in the section 4.2.

This option will decrypt with XOR the content of the internal variable Content1 with the key stored in the internal variable Key1. Then, a file path must be chosen where the decrypted content will be kept.

It is possible to use wildcards with this option. Take a look at the section 4.4.13 where it is explained how to use wildcards with the crypto options.

4.4.7. CRYPTO – CALCULATE MD5

Before reading this section, it is important to understand the idaDiscover internal variables, explained in section 3.1, and the Select menu, explained in the section 4.2.

This option will calculate MD5 of the content of the internal variable Content1. Then, a file path must be chosen where the calculated hash.

It is possible to use wildcards with this option. Take a look at the section 4.4.13 where it is explained how to use wildcards with the crypto options.

4.4.8. CRYPTO – CALCULATE SHA256

Before reading this section, it is important to understand the idaDiscover internal variables, explained in section 3.1, and the Select menu, explained in the section 4.2.

This option will calculate SHA256 of the content of the internal variable Content1. Then, a file path must be chosen where the calculated hash.

It is possible to use wildcards with this option. Take a look at the section 4.4.13 where it is explained how to use wildcards with the crypto options.

4.4.9. CRYPTO – SEARCH STRING ENCRYPTED WITH RC4

Before reading this section, it is important to understand the idaDiscover's Select menu, explained in the section 4.2.

For this option, it is necessary to configure these variables: Content1, Key1, SearchString and SearchSize.

When this option is executed, idaDiscover will loop on the content of Content1, getting blocks of size = SearchSize, it will decrypt each block with RC4 and Key1 as key, then it will check if SearchString appears into the decrypted block.

If the SearchString appears in the decrypted block, idaDiscover will log the address where the decrypted block started and will continue.

When the operation finishes, all the matches will be shown.

4.4.10. CRYPTO – SEARCH STRING ENCRYPTED WITH AES CBC

Before reading this section, it is important to understand the idaDiscover's Select menu, explained in the section 4.2.

For this option, it is necessary to configure these variables: Content1, Key1, Key2, SearchString and SearchSize.

When this option is executed, idaDiscover will loop on the content of Content1, getting blocks of size = SearchSize, it will decrypt each block with AES CBC and Key1 as key and Key2 as IV, then it will check if SearchString appears into the decrypted block.

If the SearchString appears in the decrypted block, idaDiscover will log the address where the decrypted block started and will continue.

When the operation finishes, all the matches will be shown.

4.4.11. CRYPTO – SEARCH STRING ENCRYPTED WITH AES ECB

Before reading this section, it is important to understand the idaDiscover's Select menu, explained in the section 4.2.

For this option, it is necessary to configure these variables: Content1, Key1, SearchString and SearchSize.

When this option is executed, idaDiscover will loop on the content of Content1, getting blocks of size = SearchSize, it will decrypt each block with AES ECB and Key1 as key, then it will check if SearchString appears into the decrypted block.

If the SearchString appears in the decrypted block, idaDiscover will log the address where the decrypted block started and will continue.

When the operation finishes, all the matches will be shown.

4.4.12. CRYPTO – SEARCH STRING ENCRYPTED WITH XOR

Before reading this section, it is important to understand the idaDiscover's Select menu, explained in the section 4.2.

For this option, it is necessary to configure these variables: Content1, Key1, SearchString and SearchSize.


When this option is executed, idaDiscover will loop on the content of Content1, getting blocks of size = SearchSize, it will decrypt each block with XOR and Key1 as key, then it will check if SearchString appears into the decrypted block.

If the SearchString appears in the decrypted block, idaDiscover will log the address where the decrypted block started and will continue.

When the operation finishes, all the matches will be shown.

4.4.13. CRYPTO – WILDCARDS IN CRYPTO OPTIONS

‘Calculate’ crypto options explained in sections 4.4.3 to 4.4.8 accept the following wildcards. It is possible to use the options of the select menu to configure these wildcards into the idaDiscover internal variables, as explained in the sections 4.2.16, 4.2.17 and 4.2.18:

 Please enter a string

Enter text value -

- Wildcard -

If you insert wildcards, some operations (for example crypto options) will be launched N times.

For each time, key1 will be filled with IDA data at different addresses (where IDA names were set) or imm values, depending on the inserted wildcard.

CAREFUL setting multiple wildcards could cause the operation to take loooong time.

Accepted values

%IDANAMESCONTENT1%size% (the content at names of the IDA disassemble, size indicates the amount of bytes to be copied)

%IDANAMESCONTENT2%size% (identical like %IDANAMESCONTENT1%size%)

%IDANAMESCONTENT1%-1% (identical to %IDANAMESCONTENT1%size%, but the size is automatically set to the size of the block)

%IDANAMESCONTENT1%-2% (identical to %IDANAMESCONTENT1%-1%, but trailing zeroes are removed from the read data)

%IMMOPERANDS1% (imm operands among all the instructions of the IDA disassembly)

%IMMOPERANDS2% (identical like %IMMOPERANDS1%)

OK Cancel

Basically, when you introduce wildcards in the crypto operations, they will be executed N times, replacing the wilcard by the content associated to the wildcard (this is explained in section 3.2), and all the results will be kept in the specified directory with the following format for the name of the written files:

dec_content1_<replaced wildcard offset>_<replaced wildcard size>_key1_<replaced wildcard offset>_<replaced wildcard size>_key2_<replaced wildcard offset>_<replaced wildcard size>.bin

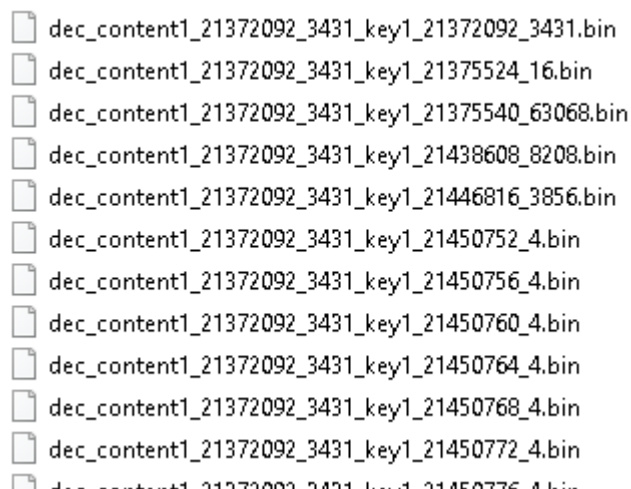
(only the variables with wildcards will appear in the name, for example if only content1 is configured with a wildcard, the name will be: dec_content1_<replaced wilcard offset>_<replaced wildcard size>.bin).

Lets see an example to understand this better. We have an unpacked sample of trickbot. We know it contains an ECS30 key, encrypted with a simple xor. The xor key is stored in some part of the binary, but we don't know where. Same thing with the encrypted ECS30 key, the encrypted ECS30 key is stored in the binary and we don't know where.

Ok, so we are going to configure %IDANAMESCONTENT1%-2% into the internal variable content1, and %IDANAMESCONTENT2%-2% into the internal variable key1. In this way, the operation will be repeated for all the contents of the objects identified by IDA (where IDA have set names), a loop similar to this:

```
for name1 in GetIDANames():
    for name2 in GetIDANames():
        content1 = GetContent(name1)
        key1 = GetContent(name2)
        content1 = removeTrailingZeroes(content1)
        key1 = removeTrailingZeroes(key1)
        XorAndKeep(content1, key1)
```

All the results will be kept in the target directory:



A list of generated binary files, each preceded by a small icon representing a file. The files are named as follows:

- dec_content1_21372092_3431_key1_21372092_3431.bin
- dec_content1_21372092_3431_key1_21375524_16.bin
- dec_content1_21372092_3431_key1_21375540_63068.bin
- dec_content1_21372092_3431_key1_21438608_8208.bin
- dec_content1_21372092_3431_key1_21446816_3856.bin
- dec_content1_21372092_3431_key1_21450752_4.bin
- dec_content1_21372092_3431_key1_21450756_4.bin
- dec_content1_21372092_3431_key1_21450760_4.bin
- dec_content1_21372092_3431_key1_21450764_4.bin
- dec_content1_21372092_3431_key1_21450768_4.bin
- dec_content1_21372092_3431_key1_21450772_4.bin
- dec_content1_21372092_3431_key1_21450776_4.bin

Have in mind that if we set two wildcards, it will cause a lot of operations (number_of_IDA_names * number_of_IDA_names) and will generate a lot of result files, and will take a long time to finish. In the example that it is being explained, 172000 files were generated, most of them small files, and it took around 45 minutes to finish. In spite of this fact, it is still acceptable and useful, because it was done automatically while you can be doing another things, and probably trying to find the xor key and the encrypted ECS30 by manual reversing, could take you a time too (specially if you don't know the malware in depth) and you will spend your effort, instead of doing it in an automatic way.

Frequently, it is not necessary to use two or more wildcards. If, for example, you know where a xor (or rc4, or other algorithm,...) key is stored, you could use one wildcard only, to decrypt all the data blocks identified by IDA, and search for the decrypted contents that you are interested in.

Once the files are generated, we search for the file containing the plaintext that we know:

```

idadiscover_output$ find . -type f -print | xargs grep ECS30
Binary file ./dec_content1_21438608_8208_key1_21458828_8.bin matches
Binary file ./dec_content1_21458828_8_key1_21438608_8208.bin matches
idadiscover_output$ xxd ./dec_content1_21438608_8208_key1_21458828_8.bin
00000000: 4543 5333 3000 0000 f320 86db 204d f073  ECS30.... .. M.s
00000010: 37b5 fb18 b0c0 af80 bbf3 fbf1 4ac0 3bc6  7.....J.;.
00000020: 001f 23ef 1c4c 0654 a38f a619 7c41 57eb  ..#...L.T....|AW.
00000030: 0bbc 7f41 a158 7970 0dc3 a138 1c5e e27a  ...A.Xyp...8.^z
00000040: d129 fbb6 5541 d58e c7c7 3e1e f3b4 6763  .)..UA....>...gc
00000050: d250 f55b 5fd1 c056 b828 87db b544 d7e1  D [ V 8 D

```

So, now we know the xor key is located at hex(21458828):

```

lata:01476F8C C3          xor_key      db 0C3h ; Ã          ; DATA XREF: decrypt_config_do_xor+19to
lata:01476F8C          db 0C3h ; Ã          ; decrypt_config_do_xor+1Eto
lata:01476F8D 54          db 54h ; T
lata:01476F8E 87          db 87h ; ‡
lata:01476F8F F0          db 0F0h ; ð
lata:01476F90 A1          db 0A1h ; ¡
lata:01476F91 80          db 80h ; €
lata:01476F92 6C          db 6Ch ; l
lata:01476F93 59          db 59h ; Y
lata:01476F94 ..          db .. ; ..

```

And the encrypted ECS30 key is located at hex(21438608):

```

.text:01472090 86          ecc_key      db 86h ; +          ; DATA XREF: decrypt_config+19to
.text:01472091 17          db 17h
.text:01472092 D4          db 0D4h ; Ô
.text:01472093 C3          db 0C3h ; Ã
.text:01472094 91          db 91h ; ‘
.text:01472095 80          db 80h ; €
.text:01472096 6C          db 6Ch ; l
.text:01472097 59          db 59h ; Y
.text:01472098 30          db 30h ; 0
.text:01472099 74          db 74h ; t
.text:0147209A 01          db 1
.text:0147209B 2B          db 2Bh ; +
.text:0147209C 81          db 81h
.text:0147209D 0D          db 0Dh ; ð

```

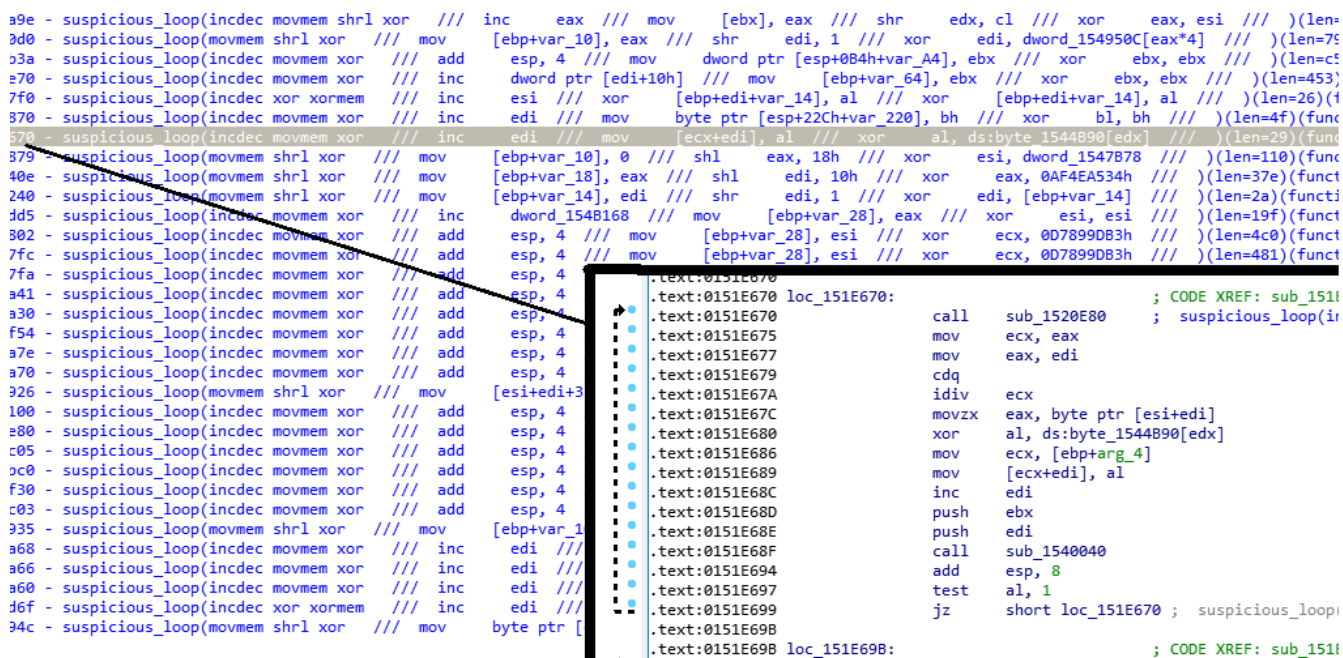
So, in this way, we have found the xor key, that is the same xor key to decrypt the xor layer of the config (following the references of the code using this xor key, we can find easily the pointer to the encrypted config). In a very automatic way and only with static analysis, we have been able to find the key of the xor layer, the ecc key, and the trickbot config.

Pending work: it is pending to add the possibility to configure a filter for the results: only the result files containing a given string will be written to disk. In disk way, we avoid to generate a big amount of files. For example in this case, we would add a filter ECS30, and only two files would have been written to disk and we would have needed to search with find and grep later.

4.5. LOOPS

From my experience, this option has been quite useful for me during malware analysis, because it helped me frequently to find decryption algorithms, decryptors of strings, etc...

Based on some heuristic rules, this option will enumerate the more suspicious loops of the IDA disassembly:



```
a9e - suspicious_loop(incdec movmem shr1 xor /// inc eax /// mov [ebx], eax /// shr edx, cl /// xor eax, esi /// )(len=
3d0 - suspicious_loop(movmem shr1 xor /// mov [ebp+var_10], eax /// shr edi, 1 /// xor edi, dword_154950C[eax*4] /// )(len=7
03a - suspicious_loop(incdec movmem xor /// add esp, 4 /// mov dword ptr [esp+0B4h+var_A4], ebx /// xor ebx, ebx /// )(len=c
e70 - suspicious_loop(incdec movmem xor /// inc dword ptr [edi+10h] /// mov [ebp+var_64], ebx /// xor ebx, ebx /// )(len=453)
7f0 - suspicious_loop(incdec xor xormem /// inc esi /// xor [ebp+edi+var_14], al /// xor [ebp+edi+var_14], al /// )(len=26)(f
370 - suspicious_loop(incdec movmem xor /// inc edi /// mov byte ptr [esp+22Ch+var_220], bh /// xor bl, bh /// )(len=4f)(func
570 - suspicious_loop(incdec movmem xor /// inc edi /// mov [ecx+edi], al /// xor al, ds:byte_1544890[edx] /// )(len=29)(func
879 - suspicious_loop(movmem shr1 xor /// mov [ebp+var_10], 0 /// shl eax, 18h /// xor esi, dword_1547B78 /// )(len=110)(func
40e - suspicious_loop(movmem shr1 xor /// mov [ebp+var_18], eax /// shl edi, 10h /// xor eax, 0AF4EA534h /// )(len=37e)(func
240 - suspicious_loop(movmem shr1 xor /// mov [ebp+var_14], edi /// shr edi, 1 /// xor edi, [ebp+var_14] /// )(len=2a)(functi
dd5 - suspicious_loop(incdec movmem xor /// inc dword_1548168 /// mov [ebp+var_28], eax /// xor esi, esi /// )(len=19f)(func
302 - suspicious_loop(incdec movmem xor /// add esp, 4 /// mov [ebp+var_28], esi /// xor ecx, 0D7899DB3h /// )(len=4c0)(func
7fc - suspicious_loop(incdec movmem xor /// add esp, 4 /// mov [ebp+var_28], esi /// xor ecx, 0D7899DB3h /// )(len=481)(func
7fa - suspicious_loop(incdec movmem xor /// add esp, 4
a41 - suspicious_loop(incdec movmem xor /// add esp, 4
a30 - suspicious_loop(incdec movmem xor /// add esp, 4
f54 - suspicious_loop(incdec movmem xor /// add esp, 4
a7e - suspicious_loop(incdec movmem xor /// add esp, 4
a70 - suspicious_loop(incdec movmem xor /// add esp, 4
326 - suspicious_loop(movmem shr1 xor /// mov [esi+edi+3
100 - suspicious_loop(incdec movmem xor /// add esp, 4
e80 - suspicious_loop(incdec movmem xor /// add esp, 4
c05 - suspicious_loop(incdec movmem xor /// add esp, 4
0c0 - suspicious_loop(incdec movmem xor /// add esp, 4
f30 - suspicious_loop(incdec movmem xor /// add esp, 4
c03 - suspicious_loop(incdec movmem xor /// add esp, 4
335 - suspicious_loop(movmem shr1 xor /// mov [ebp+var_1
a68 - suspicious_loop(incdec movmem xor /// inc edi ///
a66 - suspicious_loop(incdec movmem xor /// inc edi ///
a60 - suspicious_loop(incdec movmem xor /// inc edi ///
d6f - suspicious_loop(incdec xor xormem /// inc edi ///
34c - suspicious_loop(movmem shr1 xor /// mov byte ptr [
; CODE XREF: sub_1511E670
loc_151E670:
call sub_1520E80 ; suspicious_loop(ir
mov ecx, eax
mov eax, edi
cdq
ecx
idiv ecx
movzx eax, byte ptr [esi+edi]
xor al, ds:byte_1544890[edx]
mov ecx, [ebp+arg_4]
mov [ecx+edi], al
inc edi
push ebx
push edi
call sub_1540040
add esp, 8
test al, 1
jz short loc_151E670 ; suspicious_loopi
loc_151E698:
; CODE XREF: sub_1511E670
```

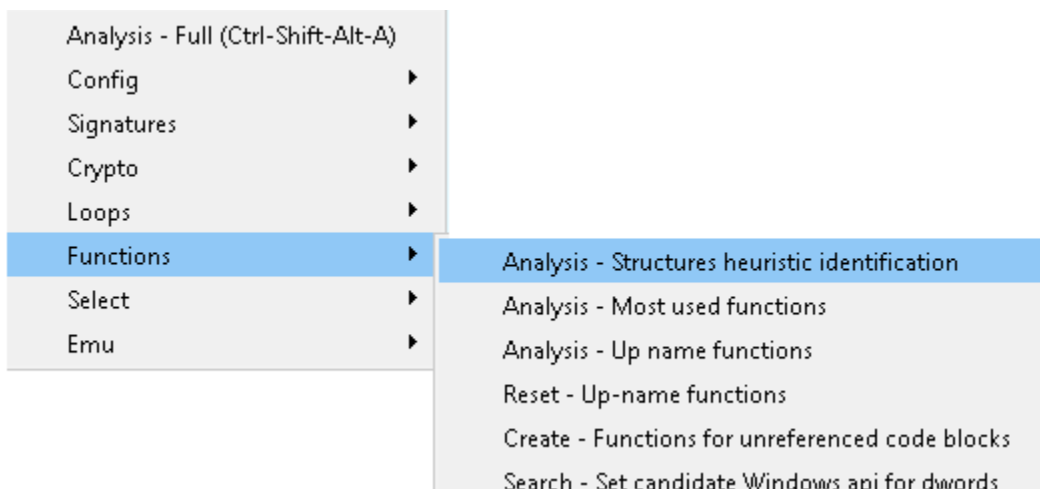
In addition, a comment will be added in the disassembly, at the address of the loop. The suspicious loop located in the previous image is a decryptor of strings.

The format of the results:

<loop address> - suspicious_loop(<suspicious instructions summary> /// ...<list of specific suspicious instructions>... ///)(len=<len of the loop in bytes>)(function = <name of the function owning the loop>)

4.6. FUNCTIONS

This menu's options are related to disassembly's functions: identification of algorithms, statistics about the functions, renaming of functions to better understanding of the code, etc...



4.6.1. FUNCTIONS – STRUCTURES HEURISTIC IDENTIFICATION

This option will try to identify well-known algorithms in a heuristic way (not yara patterns or ids, it performs a more heuristic and less restrictive identification).

For the moment only RC4 algorithm identification is implemented, but new algorithms will be supported soon.

```
Heuristic identification algorithms results:
-----
sub_40657C - candidate rc4 SBox_initialize found but SBox_Scramble not found
-----
```

4.6.2. FUNCTIONS – MOST USED FUNCTIONS

This option was useful for me frequently too, to find, for example, string decryptors. When this option is executed, a new window is open, where the list of functions is shown, ordered by the number of references to it, and a small summary of suspicious instructions in the function:


```

Function 40c720 - references 31 ( sub_40C720 ) - ( )
Function 40c7ac - references 2d ( sub_40C7AC ) - ( )
Function 41ec68 - references 2c ( sub_41EC68 ) - ( )
Function 405634 - references 2a ( @System@@CheckAutoResult$qqr1 ) - ( )
Function 40338c - references 28 ( @System@@LStrFromWStr$qqrr17System@AnsiStri
Function 40486c - references 24 ( sub_40486C ) - ( )
Function 40603c - references 24 ( GetProcAddress_0 ) - ( )
Function 4063e4 - references 24 ( @Windows@ZeroMemory$qqrpvui ) - ( )
Function 41e594 - references 20 ( sub_41E594 ) - ( )
Function 411270 - references 1f ( @Webscript@THtmlItem@GetHTML$qqrv ) - (
Function 405f44 - references 1d ( CloseHandle_0 ) - ( )
Function 406124 - references 1c ( Sleep ) - ( )
Function 426374 - references 1c ( sub_426374 ) - ( )
Function 403678 - references 1b ( sub_403678 ) - ( )
Function 4037d8 - references 1a ( @System@@WStrFromPWCharLen$qqrr17System@Wid
Function 403cd4 - references 1a ( sub_403CD4 ) - ( incdec )
Function 403878 - references 19 ( @System@@WStrFromLStr$qqrr17System@WideStri
Function 40269c - references 18 ( sub_40269C ) - ( )
Function 4033ec - references 18 ( @System@@LStrCat3$qqrv ) - ( )
Function 403238 - references 16 ( @System@@LStrFromPCharLen$qqrr17System@Ansi
Function 41326c - references 16 ( sub_41326C ) - ( )
Function 4061f4 - references 15 ( SelectObject ) - ( )
Function 41f708 - references 15 ( sub_41F708 ) - ( )
Function 41f760 - references 15 ( sub_41F760 ) - ( )
Function 427754 - references 15 ( sub_427754 ) - ( )
Function 4277e0 - references 15 ( sub_4277E0 ) - ( )
Function 40bbb0 - references 14 ( @System@UTF8Encode$qqrx17System@WideString
Function 4261c0 - references 14 ( sub_4261C0 ) - ( )
Function 4061d4 - references 13 ( DeleteObject ) - ( )
Function 40df20 - references 13 ( sub_40DF20 ) - ( incdec xor xormem )
Function 403570 - references 12 ( sub_403570 ) - ( )
Function 40412c - references 12 ( sub_40412C ) - ( )

```

The format of the results is:

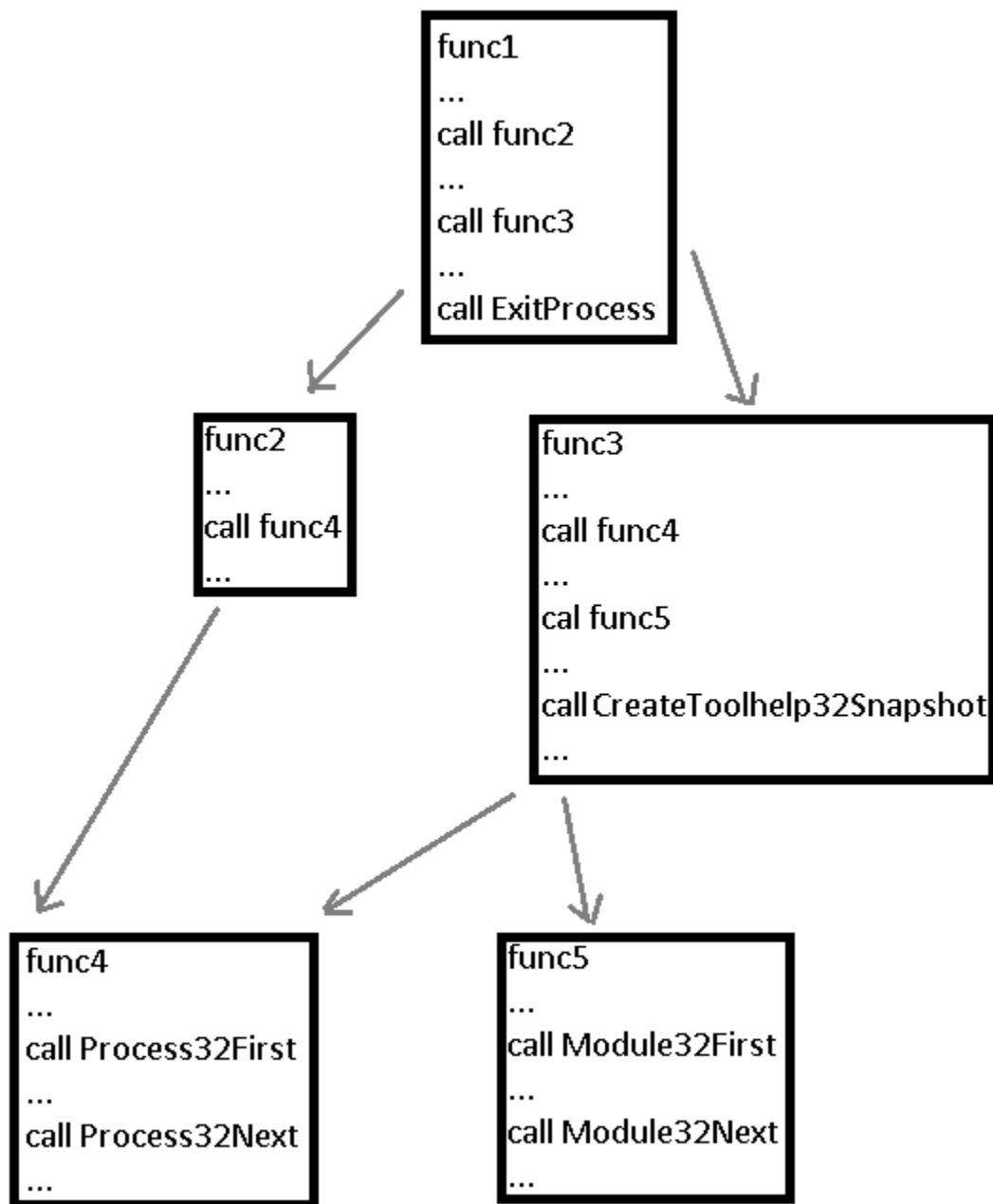
Function <address> - references <number of references> (<name of the function>) - (<summary suspicious instructions>)

4.6.3. FUNCTIONS – UP-NAME FUNCTIONS

This option is a bit more complicated to explain. When this option is executed, the functions are renamed trying to add some information about the function itself, into the own name that is set. In addition, the names ‘spread up’ to the higher level functions, so the names of the higher level functions contains information about the functions that are called under them (a kind of tree). It is going to be much easier to explain and to understand this functionality with an example, that we will see in the following lines.

Before continuing the reading of this section, it is important to read and understand the section 4.1.2. It talks about the list of names that this functionality will have in mind when it rename functions.

Lets imagine a tree of functions (based on the references between them) like this:



ExitProcess, CreateToolhelp32Snapshot, Process32First, Process32Next, Module32First and Module32Next, are names in the list of targets explained in the section 4.1.2.

So if we are in func4 and we execute the option described in this section, each function is renamed having in mind the calls to names in the list of section 4.1.2. And in addition, as we said, that names will spread up.

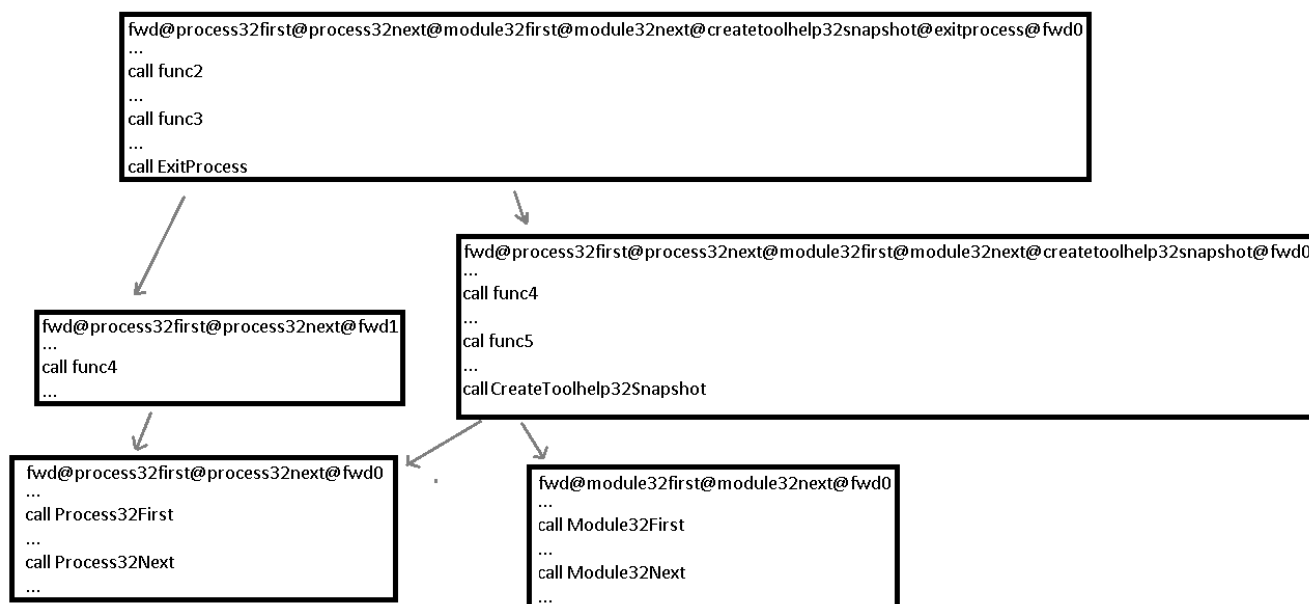
The format of the names of the renamed functions is like this:

fwd@<target1>@<target2>@...@<targetN>@fwd<X>

The last value <X> is used simply to avoid naming two functions in the same way (in the moment when the function is renamed, if the name exists, X is increased until the name didn't exist).

So, if we execute this option, the previous functions will be renamed in this way:

func4	fwd@process32first@process32next@fwd0
func5	fwd@module32first@module32next@fwd0
func2 (func4 spreads up)	fwd@process32first@process32next@fwd1
func3 (func4 and func5 spread up)	fwd@process32first@process32next@module32first@module32next@createtoolhelp32snapshot@fwd0
func1 (func2 and func3 spread up)	fwd@process32first@process32next@module32first@module32next@createtoolhelp32snapshot@exitprocess@fwd0



This functionality could look crazy and useless, but sometimes it has been very helpful for me. Sometimes you have big binaries, with a lot of code, and it is difficult to trace what is happening, what parts of the code are using other parts of the code, etc... Using this option, when you reach func1 (once it is renamed with the new name), you know under this function, at some point, it is called ExitProcess, CreateToolhelp32Snapshot, Process32First, Process32Next, Module32First and Module32Next.

I know with the super useful IDA option "Xrefs graph from" you can get similar results, and much more powerful, because you see a graph of calls. I didn't pretend to repeat functionality here. This

idaDiscover's option renames the functions and, while you explore the code, it is easy to know at a glance what is doing a function inside.

By the way, un-renaming all the functions that this option renames, it is easy, you only need to use the next idaDiscover option: Reset up-name functions.

4.6.4. RESET – UP-NAME FUNCTIONS

Already explained in the previous section, un-renames the names created by the option explained in section 4.6.3.

4.6.5. CREATE – FUNCTIONS FROM UNREFERENCED CODE BLOCKS

Frequently this option is not necessary, because IDA usually covers and identifies perfectly each part of a binary. But sometimes, specially when analyzing arbitrary dumped areas of memory (shellcodes without PE headers or other well-known headers etc...), some parts of the binary could have no references and appear as data non identified data. Sometimes this options is useful to convert that unknown data into functions.

4.6.6. SEARCH – SET CANDIDATE WINDOWS API FOR DWORDS

IdaDiscover's directory has a subfolder named WinDlls. This subfolder must contain a set of Windows dll (by default, some dlls, from a win7SP1 6.1 build 7601 32, are into this directory).

Using the offset of the exports of the dlls into the WinDlls directory, this option search for each DWORD in the IDA disassembly, if they match the low WORD of that DWORD with an export of a dll of the WinDlls directory, it renames the DWORD with the name of the export.

This option can be really useful when you dump a memory region from a running process to analyze it.

When I analyze a packed malware, I frequently let it run, and then I stop the execution with WinDbg or another debugger, and I dump the suspicious regions of the memory, such as RWE regions containing PEs, regions that seem shellcodes, etc...

What happens when you analyze a PE or a shellcode dumped from memory? Usually, the malware would have searched for APIs that it is going to use, and pointers to that APIs are kept in DWORDs. With this option, it is possible to rename that DWORDs to the name of the API.

Because of that, to get this option working, **it is important to copy the most important Windows' dlls** (kernel32, ntdll, wininet, user32, etc...) **from the machine where we run the malware** (and where we have dumped memory regions from) **into the idaDiscover's subfolder, WinDlls.**

Example:

UPX0:0046A200	dword_46A200	dd	/bA6BA46h
UPX0:0046A200			
UPX0:0046A204	dword_46A204	dd	76AAF67Bh
UPX0:0046A204			
UPX0:0046A208	dword_46A208	dd	76A71DA4h
UPX0:0046A208			
UPX0:0046A20C	dword_46A20C	dd	76A62301h
UPX0:0046A20C			
UPX0:0046A210	dword_46A210	dd	76A73EA2h
UPX0:0046A210			
UPX0:0046A214	dword_46A214	dd	76A58CB9h
UPX0:0046A214			
UPX0:0046A218	dword_46A218	dd	76A72F81h
UPX0:0046A218			
UPX0:0046A21C	dword_46A21C	dd	76A72F99h
UPX0:0046A21C			
UPX0:0046A220	dword_46A220	dd	76A71DC3h
UPX0:0046A220			
UPX0:0046A224	dword_46A224	dd	76A71DBC h
UPX0:0046A224			
UPX0:0046A228	dword_46A228	dd	76A79911h
UPX0:0046A228			
UPX0:0046A22C	dword_46A22C	dd	76A775A5h
UPX0:0046A22C			
UPX0:0046A230	dword_46A230	dd	76A6BB9Fh
UPX0:0046A230			
UPX0:0046A234	dword_46A234	dd	76A739AAh
UPX0:0046A234			
UPX0:0046A238	dword_46A238	dd	76A765D4h
UPX0:0046A238			
UPX0:0046A23C	dword_46A23C	dd	76A58A3Bh
UPX0:0046A23C			
UPX0:0046A240	dword_46A240	dd	76A72C8Ah
UPX0:0046A240			
UPX0:0046A244	dword_46A244	dd	76A72412h
UPX0:0046A244			
UPX0:0046A248	dword_46A248	dd	76A7D53Dh
UPX0:0046A248			
UPX0:0046A24C	dword_46A24C	dd	76A713D0h
UPX0:0046A24C			
UPX0:0046A250	dword_46A250	dd	76A5689Fh
UPX0:0046A250			
UPX0:0046A254	dword_46A254	dd	76A767C8h
UPX0:0046A254			
UPX0:0046A258	dword_46A258	dd	76A8F771h
UPX0:0046A258			



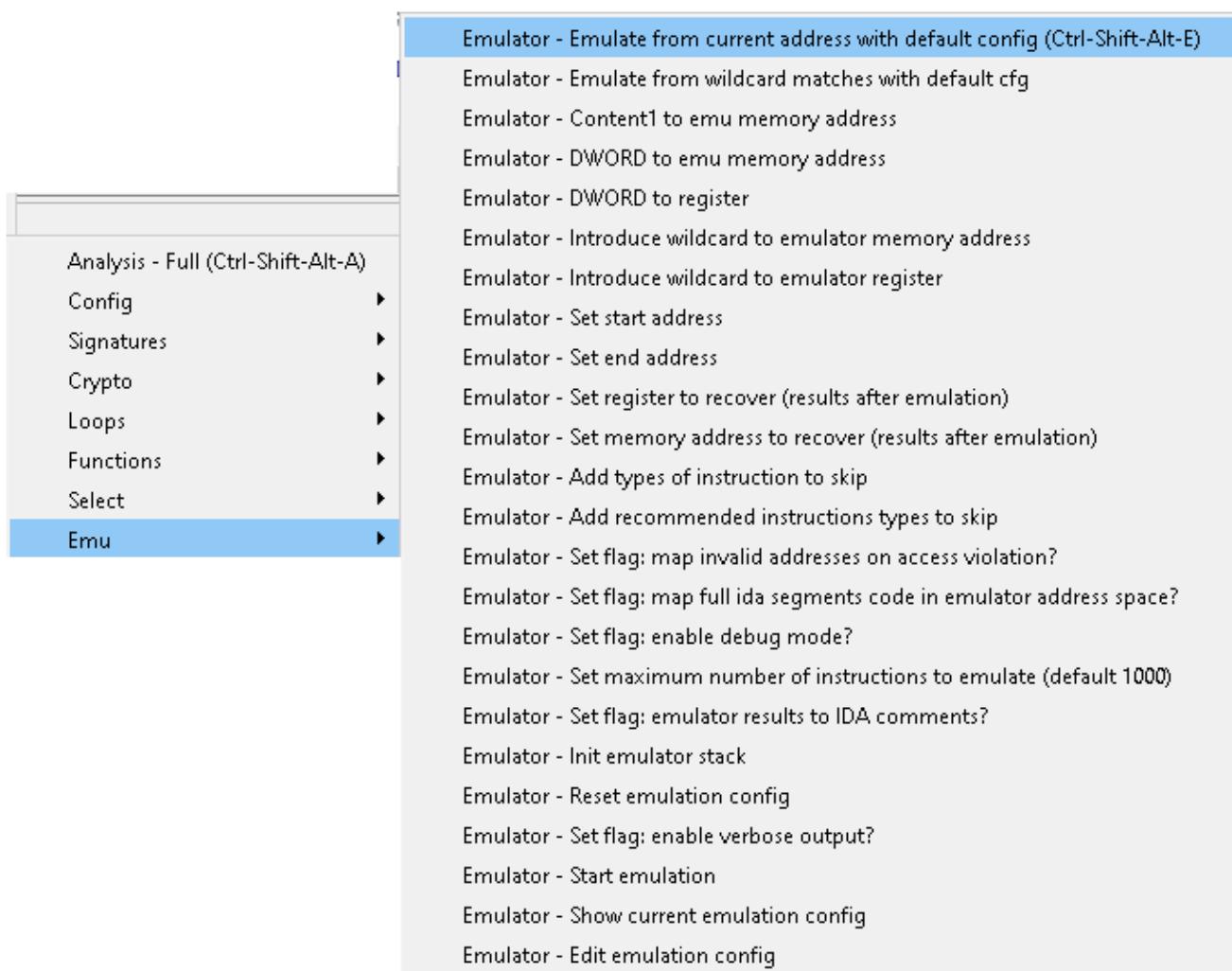
UPX0:0046A200	kernel32_dll_Sleep	dd	/bA6BA46h
UPX0:0046A200			
UPX0:0046A200			
UPX0:0046A200			
UPX0:0046A204	kernel32_dll_FatalAppExitA	dd	76AAF67Bh
UPX0:0046A204			
UPX0:0046A204			
UPX0:0046A204			
UPX0:0046A208	kernel32_dll_VirtualFree	dd	76A71DA4h
UPX0:0046A208			
UPX0:0046A208			
UPX0:0046A208			
UPX0:0046A20C	kernel32_dll_HeapDestroy	dd	76A62301h
UPX0:0046A20C			
UPX0:0046A20C			
UPX0:0046A20C			
UPX0:0046A210	kernel32_dll_HeapCreate	dd	76A73EA2h
UPX0:0046A210			
UPX0:0046A210			
UPX0:0046A210			
UPX0:0046A214	kernel32_dll_SetFileAttributesA	dd	76A58CB9h
UPX0:0046A214			
UPX0:0046A214			
UPX0:0046A214			
UPX0:0046A214			
UPX0:0046A218	kernel32_dll_FreeEnvironmentStringsA	dd	76A79911h
UPX0:0046A218			
UPX0:0046A218			
UPX0:0046A218			
UPX0:0046A21C	kernel32_dll_GetEnvironmentStrings	dd	76A775A5h
UPX0:0046A21C			
UPX0:0046A21C			
UPX0:0046A21C			
UPX0:0046A220	kernel32_dll_FreeEnvironmentStringsW	dd	76A72C8Ah
UPX0:0046A220			
UPX0:0046A220			
UPX0:0046A220			
UPX0:0046A224	kernel32_dll_GetEnvironmentStringsW	dd	76A72412h
UPX0:0046A224			
UPX0:0046A224			
UPX0:0046A224			

4.7. EMULATOR

An emulator is a powerful tool. Among the existing emulators, [Unicorn](#) is probably one of the most powerful.

Emulation is a generic tool, it can be used in different ways to build more powerful tools. You could create, for example, a kind of emulator-based debugger. Emulators have been used for years in antivirus products. Etc...

There are IDA plugins using internally an emulator. That doesn't mean idaDiscover's 'Emulator' option is repeating the functionality of that other plugins. I have tried to build a set of emulator-based tools that, from my point of view, are really powerful in the task of analyzing malware.



The target of these options that use internally the emulator, it is not having a kind of emulator-based debugger for example. Some of the tasks that you could perform automatically with these emulator options could be:

- Bruteforce functions' parameters
- Massively emulate functions on specific arguments
- Emulate all the calls to strings decryptors, get the decryptor string and set comments foreach line calling the decryptor with the obtained decrypted string
- Enable verbose output or debug mode to deeply trace the emulated code
- Etc...

When an emulation is started, these steps are followed:

- A new instance of the emulator is created
- IDA disassembly segments are mapped into the emulator memory address space
- Stack is created
- Registers' values are set to the configured registers' values
- Buffers specified by the user are copied to the emulator's memory addresses specified for them
- Set starting address (EIP)
- Start emulation (if wildcards, emulate N times, replacing wildcards. It will be explained later)

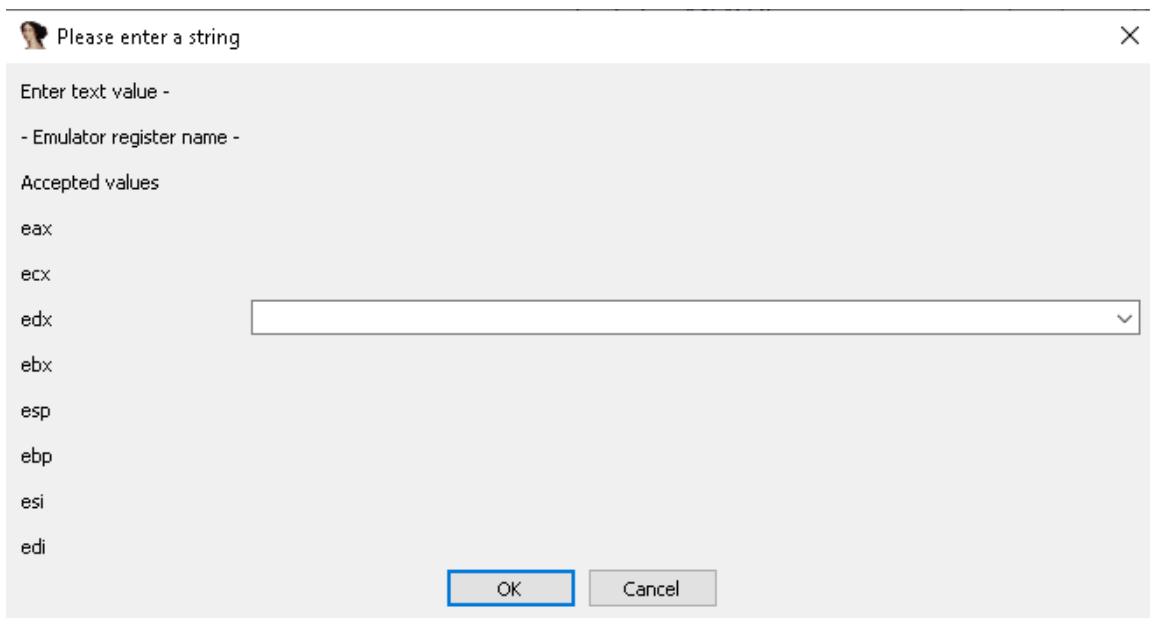
In the next section I will explain all the options that the emulator uses internally, and I will give some examples about how these options could be used in a useful way.

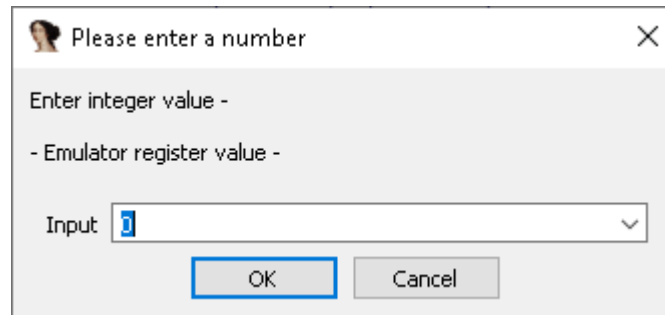
4.7.1. EMULATOR - CONTENT1 TO EMU MEMORY ADDRESS

When this option is used, idaDiscover will ask for a memory address into the emulator's address space where the current buffer selected in Content1 will be copied before starting the emulation.

4.7.2. EMULATOR - DWORD TO REGISTER

In this option, idaDiscover will ask for a register and a value to be introduced into that register.





The configured values for registers will be copied to the emulator's registers before starting the emulation.

4.7.3. EMULATOR - DWORD TO EMU MEMORY ADDRESS

Like the previous described option, this option will ask for a value, but in this case idaDiscover will ask for a memory address of the emulator's address space, and the 4 bytes of the specified DWORD value will be copied into the specified memory address.

4.7.4. EMULATOR - INIT EMULATOR STACK

Most of the cases, a correct emulation will need to create a valid stack. It would be possible to create the stack manually. For example, move 0x20000 zeros to content1 with the select options, copy that 0x20000 zeros to the address 0x0 of the address space of the emulator with the option explained in the section 4.7.1, set ESP = 0x10000 with the option explained in section 4.7.2, to point in the middle of the copied buffer. And usually with that it would be enough.

It is boring to have to create manually the stack each time that you want to emulate something. This option 'Init emulator stack' automatically creates a stack with this parameters:

- 0x20000 bytes (zeroes) copied to address 0x0
- ESP = 0x10000
- EBP = 0x10000

4.7.5. EMULATOR - SET REGISTER TO RECOVER (RESULTS AFTER EMULATION)

Emulator's options show a log of the most important events happened, however, frequently it is necessary to recover some specific registers or regions of memory.

For example, if we emulate a function just until the return of the call, we could be interested in setting the register EAX to be recovered and showed in the results after the emulation, to know the return value of the emulated function.

This is possible with this option.

4.7.6. EMULATOR - SET MEMORY ADDRESS TO RECOVER (RESULTS AFTER EMULATION)

Very similar to the option explained in the previous section, with this option it is possible to configure a memory address of the emulator's address space and a size, and the region of memory configured will be recovered and shown in the results when the emulation finishes.

4.7.7. EMULATOR - SET START ADDRESS

The value configured with this option will be moved to emulator's EIP register before starting the emulation.

4.7.8. EMULATOR - SET END ADDRESS

With this option it is possible to configure an address of the emulator's address space, once the emulation has started, it will finish when EIP reaches the configured address.

4.7.9. EMULATOR - SET MAXIMUM NUMBER OF INSTRUCTIONS TO EMULATE (DEFAULT 1000)

The name of the option is self-explanatory. Once the emulation has started, when the number of emulated instructions reach the value configured here (default is 1000), the emulation stops.

4.7.10. EMULATOR - ADD TYPES OF INSTRUCTION TO SKIP

Here it is possible to configure a list of instructions that should be skipped. This list really contains patterns that must match the disassembly of the instruction, and if it matches, the instruction is skipped.

For example, we could configure this type of instruction to skip:

- “call dword ptr [“

If we configure this pattern, we would be skipping, for example, typical calls to imports. When an instruction of this type is reached, it is simply not emulated and EIP is set to the next instruction after the “call dword ptr [...]”.

4.7.11. EMULATOR - ADD RECOMMENDED INSTRUCTIONS TYPES TO SKIP

This option is similar to the previous one, but in this case a set of predefined recommended instructions to skip are configured. For the moment only this pattern is added “call dword ptr [“ (the main reason to add this pattern is to skip calls to imports). In the future this predefined list of types of instructions to skip will be increased.

4.7.12. EMULATOR - SET FLAG: MAP INVALID ADDRESSES ON ACCESS VIOLATION?

This option activates or deactivates a flag in the emulator that defines if the page of an invalid memory address of the emulator’s address space should be mapped when an instruction accesses it. If this flag is true, the page will be mapped and filled with zeroes, and the emulation will continue. If this flag is false, the emulation will stop when an invalid address is accessed.

By default, this flag is true.

4.7.13. EMULATOR - SET FLAG: MAP FULL IDA SEGMENTS CODE IN EMULATOR ADDRESS SPACE?

This option activates or deactivates a flag in the emulator that defines if all the segments of the IDA disassembly should be mapped into the address space of the emulator.

Frequently it is highly recommended to set this flag = true, but sometimes, if we want to emulate a very specific part of the code that we know well, we could want to deactivate this flag, and then we should

use the option explained in the section 4.7.1 to copy the parts of code and data that we want to the emulator's address space.

By default this flag is true.

4.7.14. EMULATOR - SET FLAG: ENABLE DEBUG MODE?

If this flag is enabled debug messages will be printed.

By default this flag is false.

4.7.15. EMULATOR - SET FLAG: ENABLE VERBOSE OUTPUT?

If this flag is enabled, additional output messages will be printed.

By default this flag is false.

4.7.16. EMULATOR - SET FLAG: EMULATOR RESULTS TO IDA COMMENTS?

In previous sections (4.7.5 and 4.7.6) we have talked about the possibility of specifying registers and memory regions to be recovered when the emulation finishes.

With this option, it is possible to add comments in the IDA disassembly (exactly, at the address where the emulation started) with the contents of the registers and memory regions that were specified to be recovered after the emulation.

For example, with the following code:

```
.text:01518B98 8D 45 A0
.text:01518B9B 50
.text:01518B9C 68 3F 60 54 01
.text:01518BA1 E8 9A 5A 00 00
.text:01518BA6 83 C4 0C
.text:01518BA9 56
.text:01518BAA 8D 45 A0
.text:01518BAD 50

lea     eax, [ebp+var_60]
push    eax
push    offset unk_154603F
call    strings_decryptor
add     esp, 0Ch
push    esi
lea     eax, [ebp+var_60]
push    eax
```

We are going to emulate the function `strings_decryptor`, we are going to set `start_address = 0x1518b9c` and `end_address = 0x1518BA6`. We initialize the stack too.

The second parameter of the function is the address where the decrypted string will be copied. We have started the emulation when the first argument is pushed, so the function `string_decryptor` is going to find in the stack a zero for the second argument and is going to copy the decrypted string in the address `0x0`. Another option would be to copy a buffer in the address space of the emulator, move to `EAX` the address of that buffer, and start the emulation at the address of the instruction 'push `eax`', and in this way the decrypted string will be copied to that buffer. But really, it is not necessary, it is enough to let it to copy the decrypted string to the address `0x0`.

So now we use the option explained in section 4.7.6 to ask the emulator to recover the content at the address `0x0` when the emulation finishes (we specify for example a region of size 20 bytes). We start the emulation and when it finishes the comment with the decrypted string (hexa and text) is printed and added as comment:

```
.text:01518B98 8D 45 A0      lea     eax, [ebp+var_60]
.text:01518B9B 50           push    eax
.text:01518B9C 68 3F 60 54 01 push    offset unk_154603F ; 0x0:
.text:01518B9C                                     ; ->7365745f75726c006651755b2f16752e6d426437
.text:01518B9C                                     ; ->set_url.fQu[/ .u.mBd7
.text:01518B9C                                     ;
.text:01518BA1 E8 9A 5A 00 00 call    strings_decryptor
.text:01518BA6 83 C4 0C     add     esp, 0Ch
.text:01518BA9 56           push    esi
.text:01518BAA 8D 45 A0     lea     eax, [ebp+var_60]
.text:01518BAD 50           push    eax
```


4.7.17. EMULATOR - INTRODUCE WILDCARD TO EMULATOR MEMORY ADDRESS

Wildcards also works with some emulator menu's options and they are very powerful together with the emulator. In section 4.7.25 it is explained deeply how wildcards works together with the emulator.

With this option it is possible to introduce wildcards into a memory region of the emulator's address space, of this type:

- `%IDANAMESADDRESS1%`
- `%IDANAMESADDRESS2%`
- `%IDANAMESCONTENT1%size%`
- `%IDANAMESCONTENT2%size%`
- `%IDANAMESCONTENT1%-1%`
- `%IDANAMESCONTENT1%-2%`
- `%IMMOPERANDS1%`
- `%IMMOPERANDS2%`

It is important to read the section 3.2 to understand how to use each type of wildcard.

 Please enter a string

Enter text value -

- Wildcard -

If you insert wildcards, the emulation will be launched N times.

For each emulation, the target emulator address will be filled with IDA addresses, data or imm values, depending on the inserted wildcard.

CAREFUL setting multiple wildcards could cause the emulation to take loooong time.

Accepted values

%IDANAMESADDRESS1% (addreses of the names of the IDA disassembly)

%IDANAMESADDRESS2% (identical like %IDANAMESADDRESS1%)

%IDANAMESCONTENT1%size% (the content at names of the IDA disassemble, size indicates the amount of bytes to be copied)

%IDANAMESCONTENT2%size% (identical like %IDANAMESCONTENT1%size%)

%IDANAMESCONTENT1%-1% (identical to %IDANAMESCONTENT1%size%, but the size is automatically set to the size of the block)

%IDANAMESCONTENT1%-2% (identical to %IDANAMESCONTENT1%-1%, but trailing zeroes are removed from the read data)

%IMMOPERANDS1% (imm operands among all the instructions of the IDA disassembly)

%IMMOPERANDS2% (identical

OK

Cancel


4.7.18. EMULATOR - INTRODUCE WILDCARD TO EMULATOR REGISTER

Wildcards also works with some emulator menu's options and they are very powerful together with the emulator. In section 4.7.25 it is explained deeply how wildcards works together with the emulator.

With this option it is possible to introduce wildcards into emulator's registers, of this type:

- %IDANAMESADDRESS1%
- %IDANAMESADDRESS2%
- %IMMOPERANDS1%
- %IMMOPERANDS2%

It is important to read the section 3.2 to understand how to use each type of wildcard.

 Please enter a string

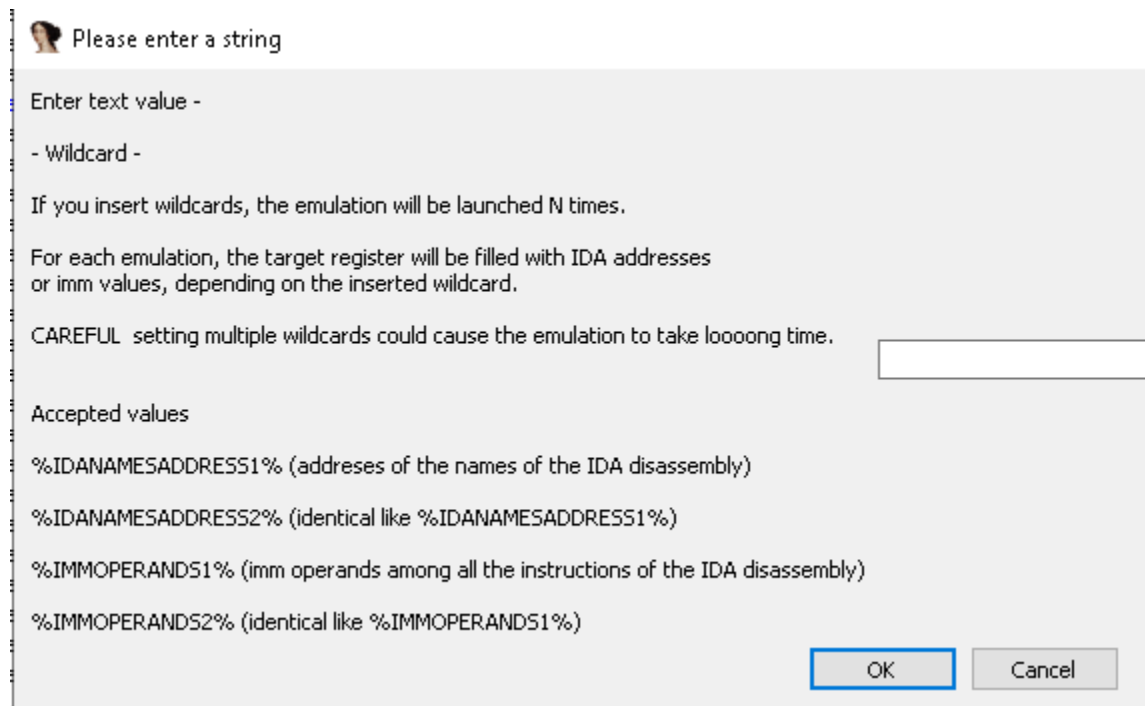
Enter text value -

- Emulator register name -

Accepted values

eax

ecx



4.7.19. EMULATOR - START EMULATION

With this option it is possible to start the emulation with the current config.

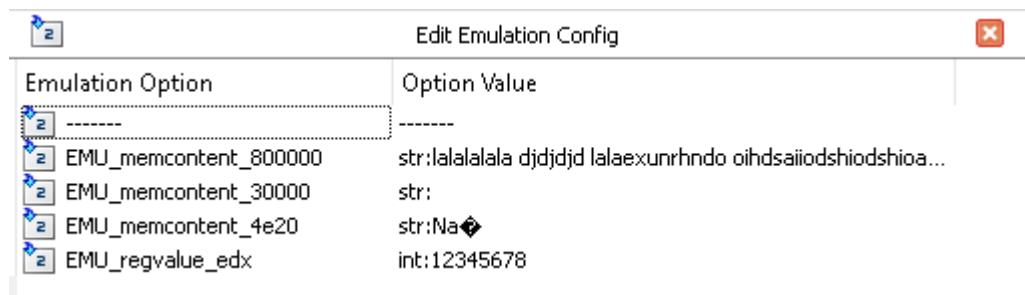
4.7.20. EMULATOR - RESET EMULATION CONFIG

With this option it is possible to reset the current emulator config and remove all the configuration parameters introduced.

4.7.21. EMULATOR - EDIT EMULATION CONFIG

With this option it is possible to edit the current introduced emulator config.

A new window is open where it is possible to edit them:



4.7.22. EMULATOR - SHOW CURRENT EMULATION CONFIG

With this option the current emulator config is printed in the output window.

4.7.23. EMULATOR - EMULATE FROM CURRENT ADDRESS WITH DEFAULT CONFIG (CTRL-SHIFT-ALT-E)

This option is very useful because you can start an emulation at the current address in the IDA dissassembly screen just with one click.

The current config is kept before starting the emulation with this option, but the following basic config is introduced:

- address 0x0 = 0x200000 bytes (zeroes)
- ESP = 0x100000
- EBP = 0x100000
- EIP = current screen EA
- endaddr = (until reaching the max number of instructions to emulate)
- flag map IDA segments = True

4.7.24. EMULATOR - EMULATE FROM WILDCARD MATCHES WITH DEFAULT CFG

In addition to the wildcards that can be configured into the emulator registers and memory regions (it was explained in sections 4.7.17 and 4.7.18 how to configure this wildcards, and it will be explained in the next section 4.7.25 how they work, with examples), it is possible to specify a pattern that should match a part of the disassembly of the instructions, and if the pattern matches the disassembly of an instruction, the emulation will start at that point.

Additionally, it is possible to configure a number of instructions previous to the match where the emulation should start (in this way, it will be possible to emulate, for example, the params to a call to a function whose name matches our pattern).

Lets see an example. We have a disassemble where we know that a function is a decryptor of strings. We would like to emulate all the calls to this function that is a decryptor of strings and create comments for each address where the emulation started with the decrypted string:

```

• • •
.text:0151A18C 8D 7D E8      lea     edi, [ebp+var_18]
.text:0151A18F 50            push    eax
.text:0151A190 57            push    edi
.text:0151A191 68 F1 4B 54 01 push    offset unk_1544BF1
.text:0151A196 E8 A5 44 00 00 call    strings_decryptor
.text:0151A19B 83 C4 0C      add     esp, 0Ch
.text:0151A19E 57            push    edi
.text:0151A19F FF D6        call    esi
.text:0151A1A1 89 C6        mov     esi, eax
.text:0151A1A3 E8 F8 F7 00 00 call    sub_15299A0
.text:0151A1A8 50            push    eax
.text:0151A1A9 6A 00        push    0
.text:0151A1AB E8 50 73 FD FF call    api_search
.text:0151A1B0 83 C4 08      add     esp, 8
.text:0151A1B3 89 C7        mov     edi, eax
.text:0151A1B5 8D 9D 48 FF FF FF lea     ebx, [ebp+var_B8]
.text:0151A1BB 6A 18        push    18h
.text:0151A1BD 53            push    ebx
.text:0151A1BE 68 80 61 54 01 push    offset unk_1546180
.text:0151A1C3 E8 78 44 00 00 call    strings_decryptor
.text:0151A1C8 83 C4 0C      add     esp, 0Ch
.text:0151A1CB 53            push    ebx
.text:0151A1CC 56            push    esi
.text:0151A1CD FF D7        call    edi
.text:0151A1CF A3 68 AC 54 01 mov     dword_154AC68, eax
.text:0151A1D4 E8 C7 F7 00 00 call    sub_15299A0
.text:0151A1D9 50            push    eax
.text:0151A1DA 6A 00        push    0
.text:0151A1DC E8 1F 73 FD FF call    api_search
.text:0151A1E1 83 C4 08      add     esp, 8
.text:0151A1E4 89 C7        mov     edi, eax
.text:0151A1E6 E8 95 E3 01 00 call    sub_1538580
.text:0151A1EB 8D 9D 27 FF FF FF lea     ebx, [ebp+var_D9]
.text:0151A1F1 50            push    eax
.text:0151A1F2 53            push    ebx
.text:0151A1F3 68 A0 61 54 01 push    offset unk_15461A0
.text:0151A1F8 E8 43 44 00 00 call    strings_decryptor
• • •

```

We are going to emulate at an instruction before each address where the word 'strings_decryptor' appears, to emulate also the push of the address of the encrypted string. The second parameter of the function is the address where the decrypted string will be copied, and the third argument is the size.

We don't know the exact size for each string, so we are going to keep in the third argument of the stack a fixed big size, and in the second argument any address, for example 0x60000000, where the decrypted string will be copied.

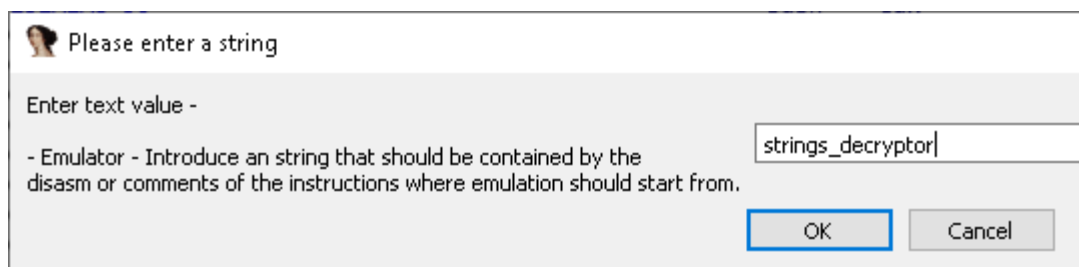
We are going to use the default stack described in section 4.7.23 (in the beginning ESP = 0x100000).

So, as explained in the section 4.7.3 (move a DWORD to a memory address of the emulator), we are going to move the value 0x60000000 to the address 0x100000 (the initial ESP) and a fixed value for the size, for example the value 0x2000 to the address 0x100004 (the third argument). Then, when the emulation starts, the address of the encrypted string will be pushed and the call to strings_decryptor will be emulated:


ESP	Content at the beginning of strings_decryptor
0xffff8	<return addr of call strings_decryptor>
0xffffc	<address of the encrypted string>
0x100000	0x60000000 <dest buffer decrypted string>
0x100004	0x2000 <size of the string>

Once we have set this configuration, we only need to specify the results that we want to recover with the option explained in section 4.7.6 (set memory address to recover after emulation). We will specify that we want to recover, for example, 0x20 bytes from the address 0x60000000. And, additionally, we are going to enable the flag described in section 4.7.16 to create comments for each emulation's results. In this way, the content of 0x60000000 after each emulation (the decrypted string) will be added as comment in the disassemble at the address where the emulation started.

So, with this configuration, we are going to use this option (emulate from wildcard matches) to emulate all the calls to strings_decryptor:



As we said, we want to emulate from the previous instruction to the call strings_decryptor:

 Please enter a number

Enter integer value -

- Emulator - Foreach match where the emulation should start, the emulation will start a number of instructions previous to the match (this number of instructions is configured here).

Input

OK Cancel

IdaDiscover will search for the matches of strings_decryptor and will start to emulate in the previous instruction, and, when the emulation finishes, comments will be added with the results:

```

...
.text:0151A18C 8D 7D E8      lea     edi, [ebp+var_18]
.text:0151A18F 50           push    eax
.text:0151A190 57           push    edi
.text:0151A191 68 F1 4B 54 01 push    offset unk_15448F1 ; 0x0:
.text:0151A191                                     ; ->637279707433322e646c6c001a7b6c0a75542b00
.text:0151A191                                     ; ->crypt32.dll..{l.uT..
.text:0151A191                                     ;
.text:0151A196 E8 A5 44 00 00 call    strings_decryptor
.text:0151A198 83 C4 0C     add     esp, 0Ch
.text:0151A19E 57           push    edi
.text:0151A19F FF D6       call    esi
.text:0151A1A1 89 C6       mov     esi, eax
.text:0151A1A3 E8 F8 F7 00 00 call    sub_15299A0
.text:0151A1A8 50           push    eax
.text:0151A1A9 6A 00       push    0
.text:0151A1AB E8 50 73 FD FF call    api_search
.text:0151A1B0 83 C4 08     add     esp, 8
.text:0151A1B3 89 C7       mov     edi, eax
.text:0151A1B5 8D 9D 48 FF FF FF lea     ebx, [ebp+var_B8]
.text:0151A1B8 6A 18       push    18h
.text:0151A1BD 53           push    ebx
.text:0151A1BE 68 80 61 54 01 push    offset unk_1546180 ; 0x0:
.text:0151A1BE                                     ; ->4365727447657443657274696669636174654368
.text:0151A1BE                                     ; ->CertGetCertificateCh
.text:0151A1BE                                     ;
.text:0151A1C3 E8 78 44 00 00 call    strings_decryptor
.text:0151A1C8 83 C4 0C     add     esp, 0Ch
.text:0151A1CB 53           push    ebx
.text:0151A1CC 56           push    esi
.text:0151A1CD FF D7       call    edi
.text:0151A1CF A3 68 AC 54 01 mov     dword_154AC68, eax
.text:0151A1D4 E8 C7 F7 00 00 call    sub_15299A0
.text:0151A1D9 50           push    eax
.text:0151A1DA 6A 00       push    0
.text:0151A1DC E8 1F 73 FD FF call    api_search
.text:0151A1E1 83 C4 08     add     esp, 8
.text:0151A1E4 89 C7       mov     edi, eax
.text:0151A1E6 E8 95 E3 01 00 call    sub_1538580
.text:0151A1EB 8D 9D 27 FF FF FF lea     ebx, [ebp+var_D9]
.text:0151A1F1 50           push    eax
.text:0151A1F2 53           push    ebx
.text:0151A1F3 68 A0 61 54 01 push    offset unk_15461A0 ; 0x0:
.text:0151A1F3                                     ; ->4365727456657269667943657274696669636174
.text:0151A1F3                                     ; ->CertVerifyCertificat
.text:0151A1F3                                     ;
.text:0151A1F8 E8 43 44 00 00 call    strings_decryptor
...

```

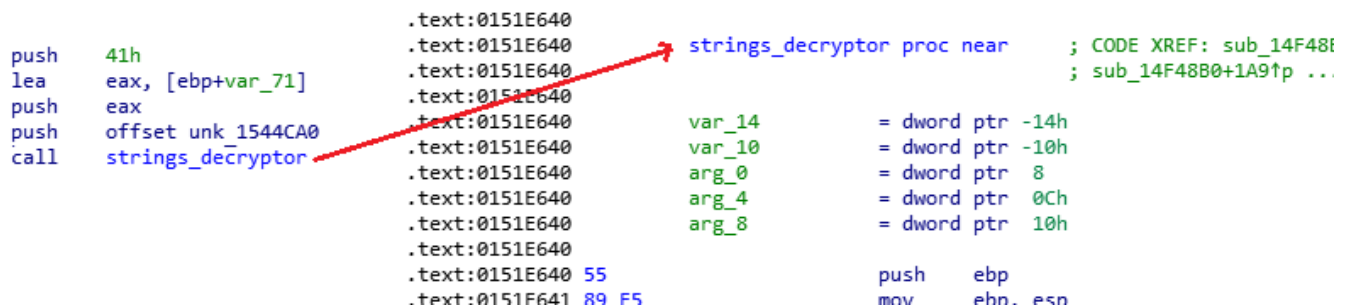
4.7.25. EMULATOR – WILDCARDS IN EMULATOR OPTIONS

We have already explained how to introduce wildcards into the emulator registers and memory regions in sections 4.7.17 and 4.7.18.

In this section we are going to explain, with an example, how to use the wildcards together with the emulator.

It is very similar to the way that wildcards were used with crypto options, that was explained in section 4.4.13. Basically, when a register contains a wildcard or a memory region contains a wildcard, N emulations will be launched, replacing for each emulation the wildcard by the associated content (associated content for each wildcard was explained in section 3.2).

As example, lets analyze a sample with a decryptor of strings:



The screenshot shows a disassembly view of a function call. On the left, assembly instructions are listed: `push 41h`, `lea eax, [ebp+var_71]`, `push eax`, `push offset unk_1544CA0`, and `call strings_decryptor`. A red arrow points from the `call` instruction to the function definition on the right. The function definition is `strings_decryptor proc near` with a comment `; CODE XREF: sub_14F48B0+1A9↑p ...`. Below the function name, arguments are listed: `var_14 = dword ptr -14h`, `var_10 = dword ptr -10h`, `arg_0 = dword ptr 8`, `arg_4 = dword ptr 0Ch`, and `arg_8 = dword ptr 10h`. At the bottom, there are two more instructions: `push ebp` and `mov ebp, esp`.

The first argument of this function ‘strings_decryptor’ is the pointer to the encrypted string. The second argument is a buffer where the decrypted string will be copied. And the third argument is the size of the string.

We want to get the full list of decrypted string, so we could start our emulation at 0x151e640 (the strings_decryptor function) introducing a wildcard to emulate N times the function, pushing a different encrypted string each time.

First, we are going to use the option described in the section 4.7.17 (introduce a wildcard to emulator memory address) to write the wildcard %IDANAMESCONTENT1%-2% (you can read about this wildcard in the section 3.2) to the address 0x20000000 of the emulator’s address space. In this way, all the data with name identified by IDA will be copied to that address for each emulation.

Having in mind that with the default stack ESP = 0x100000, now, as explained in the section 4.7.3 (move a DWORD to a memory address of the emulator), we are going to move the value 0x20000000 to the address 0x100004 (this is going to be the encrypted string address, 0x100000 would contain the ret address of strings_decryptor), the value 0x60000000 to the address 0x100008 (this is going to be the address where the decrypted string is going to be copied) and a fixed value for the size, for example the value 0x2000 to the address 0x10000c (the third argument).

Once we have set this configuration, we only need to specify the results that we want to recover with the option explained in section 4.7.6 (set memory address to recover after emulation). We will specify that we want to recover, for example, 0x20 bytes from the address 0x60000000.

And we launch the emulation (with default config to get the stack in the default region, and we increase the maximum number to 20000, to avoid that the emulation finishes before decrypting the strings). Once the emulation finishes, the emulation will be launched N times, trying to decrypt each buffer with name identified by IDA. Because of this, the decryption will be applied on a lot of buffer that won't be encrypted strings, we will need to search the results for the decrypted strings that we are interested on.

Some of the decrypted strings that are shown in the results of the example:

```
• • •
0x60000000L:content-encoding.QnW06wyTaom?Bg1VQnW06wyTaom?Bg1VQnW06wyTaom?Bg1VQnW06w
Output memory configured by user:
0x60000000:
->636f6e746556e742d656e636f64696e6700516e57
->content-encoding.QnW
• • •
0x60000000L:audio/.yTaom?Bg1VQnW06wyTaom?Bg1VQnW06wyTaom?Bg1VQnW06wyTaom?Bg1VQnW06w
Output memory configured by user:
0x60000000:
->617564696f2f007954616f6d3f42673156516e57
->audio/.yTaom?Bg1VQnW
• • •
0x60000000L:video/.yTaom?Bg1VQnW06wyTaom?Bg1VQnW06wyTaom?Bg1VQnW06wyTaom?Bg1VQnW06w
Output memory configured by user:
0x60000000:
->766964656f2f007954616f6d3f42673156516e57
->video/.yTaom?Bg1VQnW
• • •
0x60000000L:model/.yTaom?Bg1VQnW06wyTaom?Bg1VQnW06wyTaom?Bg1VQnW06wyTaom?Bg1VQnW06w
Output memory configured by user:
0x60000000:
->6d6f64656c2f007954616f6d3f42673156516e57
->model/.yTaom?Bg1VQnW
• • •
0x60000000L:multipart/form-data.06wyTaom?Bg1VQnW06wyTaom?Bg1VQnW06wyTaom?Bg1VQnW06w
Output memory configured by user:
0x60000000:
->6d756c7469706172742f666f726d2d6461746100
->multipart/form-data.
• • •
```