

Krypton

Author : Karthik Selvaraj at Symantec Corporation, contact : neofx at gmail dot com

Krypton is an IDA Plugin that assists one in reverse engineering x86 binary executables, by executing a function from IDB (IDA database) using IDA's powerful Appcall feature.

krypton takes xrefs from a given function (say a possible decoder) to find all function calls to it and then parses and finds the parameters use (including prototype, no of arguments, and the arguments themselves) from instructions and uses them to execute the function using Appcall, this is most useful in analyzing a malware binary with encryption.

Features

- lists top referenced functions list to start analysis.
- Assists in identifying encryption/decryption sub-routine.
- Decrypts the encrypted strings/contents in a binary without a need to understand encryption method or an invasive debugging. thus helping in overall analysis of the binary being reverse engineered.

Installation

- To Install, Copy the plugin to IDA Plugin folder,
- Std path: "C:\Program Files\IDA\plugins\"
- NOTE: plugin requires IDA >= 5.6

Quick Start Guide

Issue **Ctrl+F8**

To Decrypt Strings

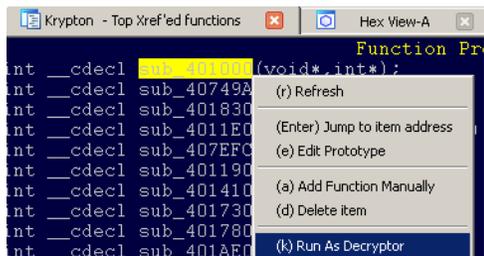
Say you see a function that looks like a string decryptor call, e.g.

```
mov     edx, [ebp+lpProcName]
push   edx
push   offset aTrgguernqpbagrkg ; "TrgGuernqPbagrkg"
call   sub_401000
add    esp, 8
mov    eax, [ebp+lpProcName]
push   eax ; lpProcName
mov    ecx, [ebp+hModule]
push   ecx ; hModule
call   ds:GetProcAddress
mov    dword_40E94C, eax
mov    edx, [ebp+lpProcName]
push   edx
push   offset aFrgguernqpbagrkg ; "FrgGuernqPbagrkg"
call   sub_401000
```

Hit Ctrl+F8

Krypton would list the top xref'ed functions, select suspected function, right click and say "(k) run as decryptor"

NOTE: If the analyzed file is a DLL, then IMAGE_FILE_DLL flag must be cleared from the characteristics entry in the PE header prior to running krypton. That is, bit 13 must be cleared in the characteristics WORD at offset 0x18 of the PE header. Be sure to reload the DLL in IDA prior to trying to run Krypton once the flag is cleared.



krypton decrypts it, you can right click and have it written as comment near call

```
0x00401520 | VirtualAllocEx
0x004014A1 | advapi32
0x00401483 | ntdll
0x00401544 | WriteProcessMemory
0x00401465 | kernel32
0x00401568 | (r) Refresh ext
0x0040161C | (w) write these to IDB sh
0x0040158C | SetInreadContext
0x004015B0 | ResumeThread
0x004015F8 | CryptAcquireContextA
0x004015D4 | GetModuleFileNameA
0x004014D8 | CreateProcessA
0x00401664 | CryptGetHashParam
0x004014FC | NtUnmapViewOfSection
0x00401640 | CryptHashData
```

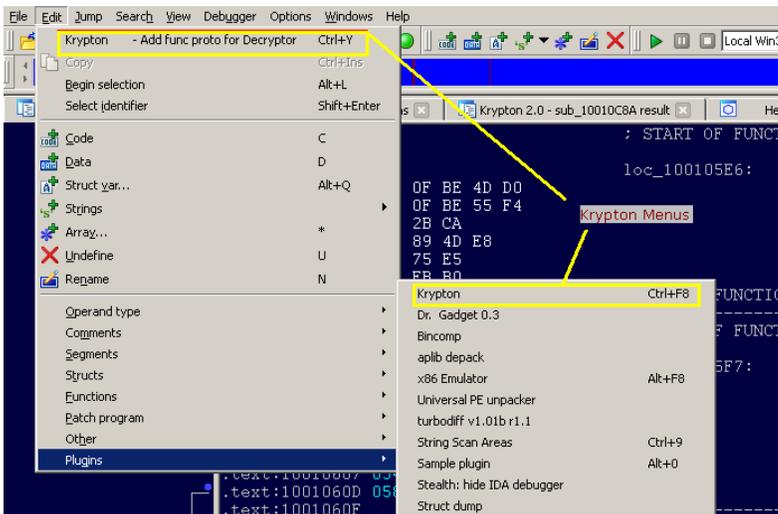
result of decryption is written as comment in respective places

```
mov     edx, [ebp+lpProcName]
push   edx
push   offset aTrgguernqpbagrkg ; "TrgGuernqPbagrkg"
call   sub_401000 ; GetThreadContext
add    esp, 8
mov    eax, [ebp+lpProcName]
push  eax ; lpProcName
mov    ecx, [ebp+hModule]
push  ecx ; hModule
call   ds:GetProcAddress
mov    dword_40E94C, eax
mov    edx, [ebp+lpProcName]
push  edx
push  offset aFrgguernqpbagrkg ; "FrgGuernqPbagrkg"
call   sub_401000 ; SetThreadContext
```

decryption result as comment

Detailed Usage Guide - Decryption

Once installed, krypton adds the below menus and HotKeys, to launch the Plugin use hotkey **Ctrl+F8**



Krypton waits for IDA Auto-Analysis to finish and then brings up a Krypton Plugin prototype view, which lists top Cross referenced functions in IDB

Usually decryptor functions are called in multiple places thus increasing their cross reference numbers, so one might possibly find the decryptor functions at the start of the list and this is also a good starting point for analysis, the more one marks top referenced functions, the more clearer IDB gets.

Virtual Addr	Function Prototype	Argument Array	Type	xrefs
0x10010C8A	int __cdecl sub_10010C8A(void*,int,int*);	[(5, 4), (2, 1), ('BUFFER', 1024)]	push	(296)
0x10002669	int __cdecl sub_10002669(int*);	[('BUFFER', 1024)]	push	(039)
0x10007CAA	int __cdecl sub_10007CAA(int*);	[('BUFFER', 1024)]	push	(038)
0x100036C6	int __cdecl sub_100036C6(int*);	[('BUFFER', 1024)]	push	(035)
0x10005571	int __cdecl sub_10005571(int*,int*);	[('BUFFER', 1024), ('BUFFER', 1024)]	push	(024)
0x1000C83C	int __cdecl sub_1000C83C(int*);	[('BUFFER', 1024)]	push	(022)
0x10002D12	int __cdecl sub_10002D12(void);	[('BUFFER', 1024)]	push	(012)
0x10002F97	int __cdecl sub_10002F97(int*);	[('BUFFER', 1024)]	push	(011)
0x10008F4F	int __cdecl sub_10008F4F(int*,int,int,int);	[('BUFFER', 1024), (2, 1), (5, 4), (2, 1)]	push	(011)
0x10004AB1	int __cdecl sub_10004AB1(int*);	[('BUFFER', 1024)]	push	(010)
0x1000D00C	int __cdecl sub_1000D00C(int*);	[('BUFFER', 1024)]	push	(010)
0x10005ADB	int __cdecl sub_10005ADB(int*);	[('BUFFER', 1024)]	push	(008)
0x10009914	int __cdecl sub_10009914(int*);	[('BUFFER', 1024)]	push	(008)
0x100029C0	int __cdecl sub_100029C0(int*);	[('BUFFER', 1024)]	push	(006)
0x10003695	int __cdecl sub_10003695(int*);	[('BUFFER', 1024)]	push	(005)
0x1000895E	int __cdecl sub_1000895E(void);	[('BUFFER', 1024)]	push	(005)
0x1000935B	int __cdecl sub_1000935B(int*);	[('BUFFER', 1024)]	push	(005)
0x1000C3BD	int __cdecl sub_1000C3BD(int*);	[('BUFFER', 1024)]	push	(005)
0x10001599	int __cdecl sub_10001599(int*);	[('BUFFER', 1024)]	push	(004)
0x10003DCC	int __cdecl sub_10003DCC(int*);	[('BUFFER', 1024)]	push	(004)
0x1000627F	int __cdecl sub_1000627F(void);	[('BUFFER', 1024)]	push	(004)
0x10009E76	int __cdecl sub_10009E76(int*);	[('BUFFER', 1024)]	push	(004)
0x1000A1D1	int __cdecl sub_1000A1D1(int*,int);	[('BUFFER', 1024), (5, 4)]	push	(004)
0x1000CBA8	int __cdecl sub_1000CBA8(int*);	[('BUFFER', 1024)]	push	(004)
0x1000F080	int __cdecl sub_1000F080(int*);	[('BUFFER', 1024)]	push	(004)

Above figure shows top referenced functions. The 3rd column named "argument array" is a series of (argument_instruction_size, data_size) pairs

So for below example call

```

push 0x04010010    (68 10 00 01 04)
push 3              (6A 03)
push 0x03b09000    (68 00 90 b0 03)
call 0x10010C8A

```

there are 3 arguments, whose argument array pair would be, [(5,4),(2,1),(5,4)] since the 3rd argument is a memory buffer we have to pass allocated buffer address.

To accomplish this, Plugin allows a few "Keywords" to be specified in Argument array

To pass a allocated buffer, use **"BUFFER"**

- this instructs Plugin to allocate memory and pass the address as argument, and buffer size is the second value within parenthesis. As in ('BUFFER', 0x400)

To pass a const value in place of an argument, use **"CONST"**

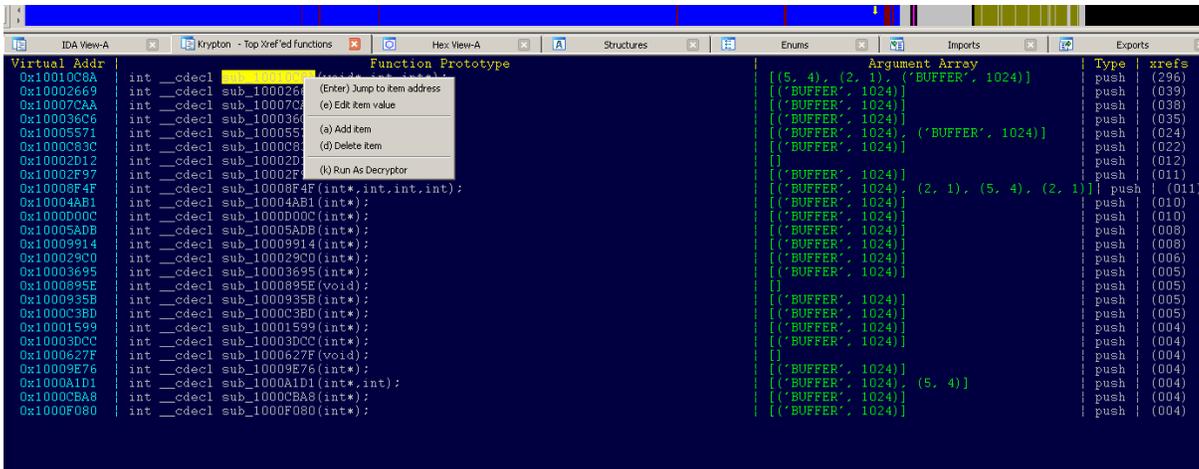
- this instructs Plugin to pass a constant value as a parameter, const value is the second value within parenthesis. As in ('CONST', 0x11).

In our example, first function sub_10010C8A has 296 references, when analyzed we can confirm that it is a decryptor. Hovering over these functions will bring up hint that shows, 10 disassembly lines from a first cross reference of that function. This helps one to verify and fix the argument array.

Virtual Addr	Function Prototype	Argument Array	Type	xrefs
0x10010C8A	int __cdecl decryptor_func(void*,int,int*);	[(5, 4), (2, 1), ('BUFFER', 1024)]	push	(296)
0x10002669	int __cdecl sub_10002669(int*);	[('BUFFER', 1024)]	push	(039)
0x10007CAA	int __cdecl sub_10007CAA(int*);	[('BUFFER', 1024)]	push	(038)
0x100036C6	int __cdecl sub_100036C6(int*);	[('BUFFER', 1024)]	push	(035)
0x10005571	int __cdecl sub_10005571(int*,int*);	[('BUFFER', 1024), ('BUFFER', 1024)]	push	(024)
0x1000C83C	int __cdecl sub_1000C83C(int*);	[('BUFFER', 1024)]	push	(022)
0x10002D12	int __cdecl sub_10002D12(void);	[('BUFFER', 1024)]	push	(012)
0x10002F97	int __cdecl sub_10002F97(int*);	[('BUFFER', 1024)]	push	(011)
0x10008F4F	int __cdecl sub_10008F4F(int*,int,int,int);	[('BUFFER', 1024), (2, 1), (5, 4), (2, 1)]	push	(011)
0x10004AB1	int __cdecl sub_10004AB1(int*);	[('BUFFER', 1024)]	push	(010)
0x1000D00C	int __cdecl sub_1000D00C(int*);	[('BUFFER', 1024)]	push	(010)
0x10005ADB	int __cdecl sub_10005ADB(int*);	[('BUFFER', 1024)]	push	(008)
0x10009914	int __cdecl sub_10009914(int*);	[('BUFFER', 1024)]	push	(008)
0x100029C0	int __cdecl sub_100029C0(int*);	[('BUFFER', 1024)]	push	(006)
0x10003695	int __cdecl sub_10003695(int*);	[('BUFFER', 1024)]	push	(005)
0x1000895E	int __cdecl sub_1000895E(void);	[('BUFFER', 1024)]	push	(005)
0x1000935B	int __cdecl sub_1000935B(int*);	[('BUFFER', 1024)]	push	(005)
0x1000C3BD	int __cdecl sub_1000C3BD(int*);	[('BUFFER', 1024)]	push	(005)
0x10001599	int __cdecl sub_10001599(int*);	[('BUFFER', 1024)]	push	(004)
0x10003DCC	int __cdecl sub_10003DCC(int*);	[('BUFFER', 1024)]	push	(004)
0x1000627F	int __cdecl sub_1000627F(void);	[('BUFFER', 1024)]	push	(004)
0x10009E76	int __cdecl sub_10009E76(int*);	[('BUFFER', 1024)]	push	(004)
0x1000A1D1	int __cdecl sub_1000A1D1(int*,int);	[('BUFFER', 1024), (5, 4)]	push	(004)
0x1000CBA8	int __cdecl sub_1000CBA8(int*);	[('BUFFER', 1024)]	push	(004)
0x1000F080	int __cdecl sub_1000F080(int*);	[('BUFFER', 1024)]	push	(004)

Right clicking on the Protolist view will bring a Popup Menu as shown in the below screen shot, bringing options to edit/delete/add function prototypes.

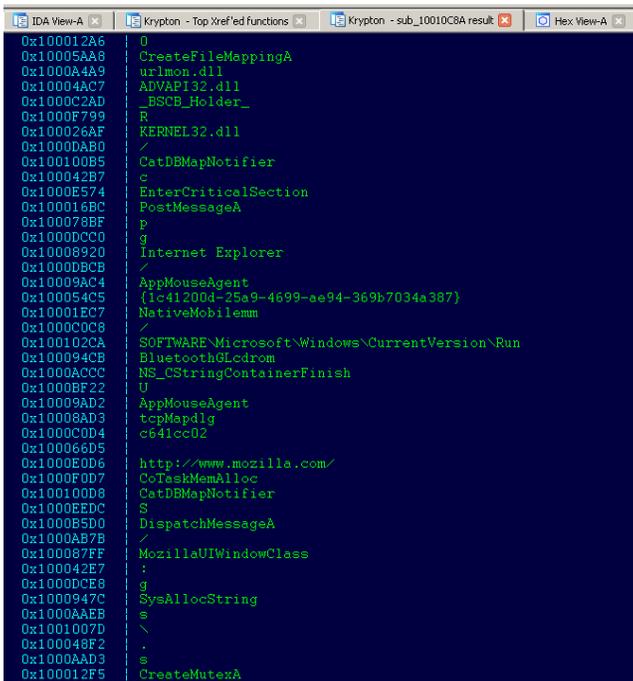
One can add new function to the prototype list, by placing cursor in the interested function from IDA dis-assembly view and issuing **Ctrl+Y** shortcut.



Once we verified/corrected that the function Prototype and Argument Array listed, we can run the function for all its cross references, by issuing **"k"** key in keyboard or by right clicking and selecting menu item **"(k) Run as Decryptor"**

When issued, Plugin automatically sets up necessary breakpoints and runs the selected function in default debugger and returns the result in another view as shown in the below figure.

Note: Krypton does not execute the binary like any other debugger, it only runs the selected function by setting up necessary stack with the arguments it found at the respective reference points of the function.



When the decryption results are satisfactory one can have the results written back to IDB as comments at their appropriate call references by issuing **"w"** key in keyboard or by right clicking and selecting menu item **"(w) write these to IDB"** as shown below.

```

IDA View-A x Krypton - Top Xref'ed functions x Krypton - sub_10010C8A result x
0x100012A6 | 0
0x10005AA8 | CreateFileMappingA
0x1000A4A9 | urlmon.dll
0x10004AC7 | ADVAPI32.dll
0x1000C2AD | _BSCB_Holder_
0x1000F799 | R
0x100026AF | KERNEL32.dll
0x1000DAB0 | /
0x100100B5 | CatDBMapNotifier
0x100042B7 | c (r) Refresh
0x1000E574 | EnterCriticalSection (w) write these to IDB
0x100016BC | PostMessageA
0x100078BF | p
0x1000DCC0 | g
0x10008920 | Internet Explorer
0x1000DBC8 | /
0x10009AC4 | AppMouseAgent
0x100054C5 | {1c41200d-25a9-4699-ae94-369b7034a387}
0x10001EC7 | NativeMobilemm
0x1000C0C8 | /
0x100102CA | SOFTWARE\Microsoft\Windows\CurrentVersion\Run
0x100094CB | BluetoothGLcdrom
0x1000ACCC | NS_CStringContainerFinish
0x1000BF22 | U
0x10009AD2 | AppMouseAgent
0x10008AD3 | tcpMapdlg
0x1000C0D4 | c641cc02
0x100066D5 |
0x1000E0D6 | http://www.mozilla.com/
0x1000F0D7 | CoTaskMemAlloc
0x100100D8 | CatDBMapNotifier
0x1000EEDC | S
0x1000B5D0 | DispatchMessageA
0x1000AB7B | /
0x100087FF | MozillaUIWindowClass
0x100042E7 | ;
0x1000DCE8 | g
0x1000947C | SysAllocString
0x1000AAEB | s
0x1001007D | \
0x100048F2 | .
0x1000AAD3 | s
0x100012F5 | CreateMutexA

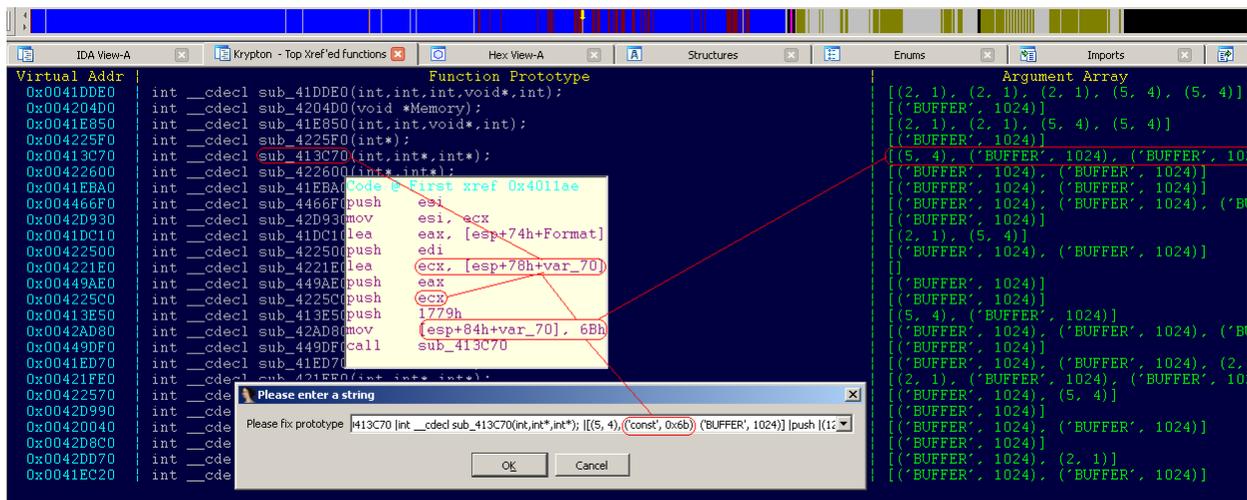
```

result of writing into IDB can be easily seen in the cross reference of the decrypt function as below.

Direction	Type	Address	Text
Up	p	sub_10001095+132	call decryptor_func ; c641cc02
Up	p	sub_10001095+145	call decryptor_func ; 1.0rc2
Up	p	sub_10001095+1AF	call decryptor_func ; 1
Up	p	sub_10001095+211	call decryptor_func ; 0
Up	p	sub_100012B7+3E	call decryptor_func ; CreateMutexA
Up	p	sub_1000153A-21	call decryptor_func ; CoInitialize
Up	p	sub_10001599+47	call decryptor_func ; ole32.dll
Up	p	sub_10001636+2E	call decryptor_func ; ObtainUserAgentString
Up	p	sub_10001693+29	call decryptor_func ; PostMessageA
Up	p	sub_10001693:loc_10001...	call decryptor_func
Up	p	sub_10001A33+4F	call decryptor_func ; PostQuitMessage
Up	p	sub_10001ABF-10	call decryptor_func ; TranslateMessage
Up	p	sub_10001ABF+92	call decryptor_func ; firefox
Up	p	sub_10001DE5+58	call decryptor_func ; Software\Classes\CLSID\{def18169-486d-436b-9349-6c712331df...}
Up	p	sub_10001DE5+E2	call decryptor_func ; NativeMobilemm
Up	p	sub_10001DE5+129	call decryptor_func ; SOFTWARE\Microsoft\MSLicensing\HardwareID
Up	p	sub_10001DE5+18F	call decryptor_func ; ClientHWID
Up	p	sub_10001DE5:loc_10002...	call decryptor_func
Up	p	sub_10001DE5+37D	call decryptor_func ; Software\Classes\CLSID\{def18169-486d-436b-9349-6c712331df...}
Up	p	sub_1000237D+9C	call decryptor_func ; exe
Up	p	sub_10002669+46	call decryptor_func ; KERNEL32.dll
Up	p	sub_1000274F:loc_10002...	call decryptor_func
Up	p	sub_1000286C+28	call decryptor_func ; DestroyWindow
Up	p	sub_10002988:loc_10002...	call decryptor_func
Up	p	sub_10002A72+21	call decryptor_func ; WaitForSingleObject
Up	p	sub_10002ABE+50	call decryptor_func ; ieCommonOffice
Up	p	sub_10002BBB+41	call decryptor_func ; OLEAUT32.dll
Up	p	sub_10002BBB+67	call decryptor_func ; UnmapViewOfFile
Up	p	sub_10002CBF+3F	call decryptor_func ; CreateDirectoryA
Up	p	sub_10002EE7+49	call decryptor_func ; GetMessageA

Line 1 of 296

Let see an example of a decryptor function where prototype editing can be used,



In the above figure, we can see that Plugin guessed the prototype to be `sub_413C70(int, int*, int*)`, and the argument array as `[(5,4), ('BUFFER', 1024), ('BUFFER', 1024)]`.

However, when looked at the disassembly in the hint, one can see that a constant value **0x6B** is passed through a local variable `var_70`.

So we can tell krypton to use a const value for second argument by editing the prototype listed to have argument array value `[(5,4), ('const', 0x6b), ('BUFFER', 1024)]`.

Now when run with the modified prototype, krypton executes correctly, decrypts and shows us result.

