

API Deobfuscator: Resolving Obfuscated API Functions in Modern Packers



2015.8.6.

Seokwoo Choi

- Introduction
- API Deobfuscation Method
 - Memory Access Analysis for Dynamic Obfuscation
 - Iterative Run-until API method for Static Obfuscation
- Implementation
- Demo
- Conclusion

- Malwares hide functionalities by API obfuscation
 - Commercial packers obfuscate API functions
 - Malware authors have their own API obfuscator
- No deobfuscation tools for some modern packers
 - x64 packers
 - Custom packers

API obfuscation techniques in modern packers

- Dynamic API Obfuscation
 - API functions are obfuscated during runtime
 - Instructions and addresses changes every run

```
helloworldmsgbox32_tiger_red.exe+0000100e call 0x68a0000
068a0000 mov edi, edi
068a0002 pushad
068a0003 pushad
068a0004 pushfd
068a0005 jmp 0x68a001a
068a001a xor ecx, 0x2bb296b6
068a0020 jmp 0x68a0031
....
068a032e pop eax
068a032f call 0x777de9ed
user32.dll+0005e9ed mov edi, edi
user32.dll+0005e9ef push ebp
```

Branch into a newly allocated block during execution time
(obfuscated User32.dll :MessageBox)

API obfuscation techniques in modern packers

- Static API Obfuscation
 - API functions are obfuscated compile(packaging) time
 - Instructions and addresses are the same

```
hw64_tmd233_tr.exe+0000000000001017 call 0x7ff6bbef63a0
hw64_tmd233_tr.exe+000000000002c63a0 sub rsp, 0x8
hw64_tmd233_tr.exe+000000000002c63a4 jmp 0x7ff6bbf00b79
hw64_tmd233_tr.exe+000000000002d0b79 push r10
hw64_tmd233_tr.exe+000000000002d0b7b push r10
hw64_tmd233_tr.exe+000000000002d0b7d add qword ptr [rsp], 0x72384261
hw64_tmd233_tr.exe+000000000002d0b85 jmp 0x7ff6bbf01d47
hw64_tmd233_tr.exe+000000000002d1d47 mov r10, qword ptr [rsp]
hw64_tmd233_tr.exe+000000000002d1d4b add rsp, 0x8
hw64_tmd233_tr.exe+000000000002d1d4f push rcx
hw64_tmd233_tr.exe+000000000002d1d50 jmp 0x7ff6bbf01efc
hw64_tmd233_tr.exe+000000000002d1efc mov rcx, 0x72384261
```

Branch into other section

.....

```
hw64_tmd233_tr.exe+000000000002ccf29 push r13
hw64_tmd233_tr.exe+000000000002ccf2b mov r13, 0x8
hw64_tmd233_tr.exe+000000000002ccf32 sub rax, 0x5f6f805b
hw64_tmd233_tr.exe+000000000002ccf38 add rax, r13
hw64_tmd233_tr.exe+000000000002ccf3b jmp 0x7ff6bbf01830
hw64_tmd233_tr.exe+000000000002d1830 add rax, 0x5f6f805b
hw64_tmd233_tr.exe+000000000002d1836 pop r13
hw64_tmd233_tr.exe+000000000002d1838 xchg qword ptr [rsp], rax
hw64_tmd233_tr.exe+000000000002d183c jmp 0x7ff6bbf022a8
hw64_tmd233_tr.exe+000000000002d22a8 mov rsp, qword ptr [rsp]
hw64_tmd233_tr.exe+000000000002d22ac ret
```

API Call by 'ret' instruction

API Deobfuscation Goal

- After deobfuscation, we have
 - (Near) original entry point
 - Recovered API function calls at OEP
- With the deobfuscated image, we can do
 - Static analysis with disassembled and decompiled code
 - Dynamic analysis with debuggers

- How to deobfuscate API obfuscated binaries?
 - Dynamic API Obfuscation
 - Memory Access Analysis
 - Static API Obfuscation
 - Iterative Run-until-API Method
- How to evade anti-debugging?
 - Dynamic binary instrumentation (Intel Pin)
 - Anti-anti-debugger plugin in debuggers
 - Emulators

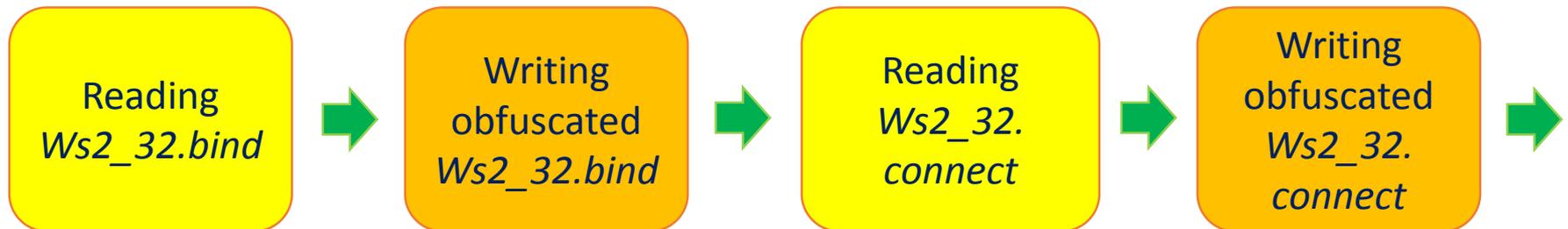


API Deobfuscation for Dynamic Obfuscators

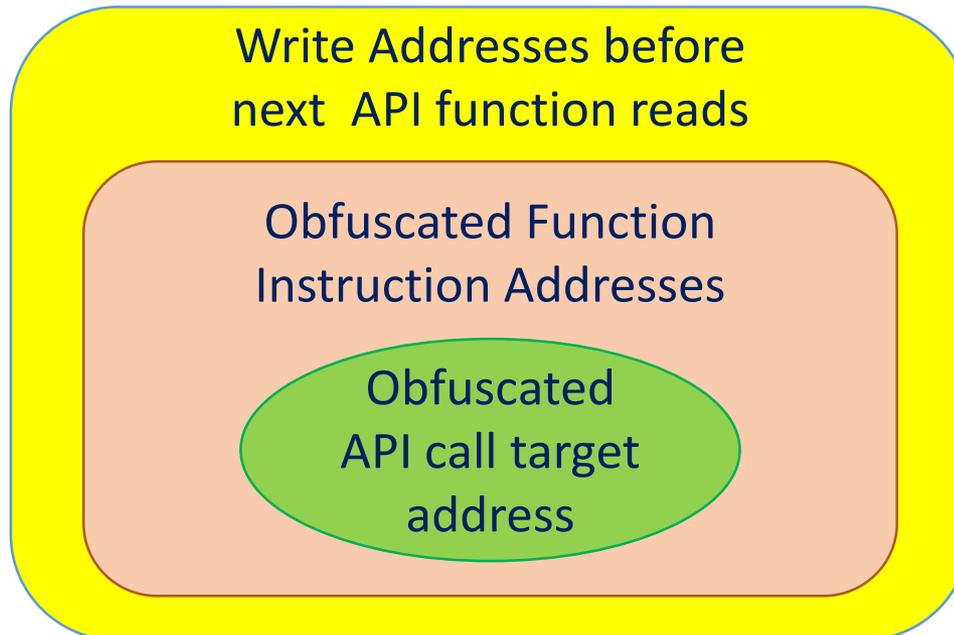
API Deobfuscation for Dynamic Obfuscators

- Memory Access Analysis
 - Relate *memory reads on API function code* and corresponding *memory writes on obfuscated code*
 - Instruction addresses of obfuscated API function
→ Original API function
 - Recover original API function by the obfuscated call target address

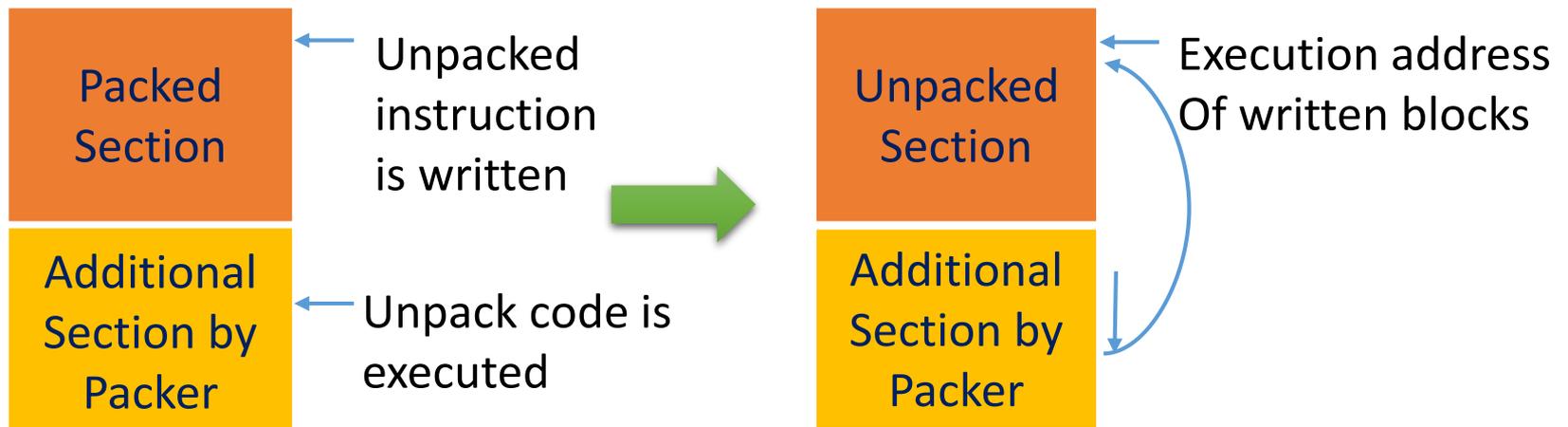
- What happens during runtime obfuscation?
 - Runtime obfuscator reads each function, obfuscates each instruction, writes the obfuscated code into a newly allocated memory
 - Each function is obfuscated in sequence



- How can we identify the original API function?
 - Record every memory write before the next API function or DLL reads
 - Limit the number of memory write for the last API function



- Find OEP
 - Record every memory write and execute
 - OEP is the Last written address that is executed
 - Check written memory blocks (1 block = 4 Kbytes) to save memory
 - OEP is in the original executable file sections



- Search for intermodular calls at OEP by pattern matching
 - Matched patterns may contain false positives
 - After target address resolution, misinterpreted instruction disappears

The screenshot shows the Immunity Debugger interface for a process named '*G.P.U*' in the main thread of the 'HelloWor' module. The assembly window displays instructions from address 012C1000 to 012C1014, including PUSH, CALL, and OR instructions. A 'Registers (FPU)' window shows the state of registers like EAX, ECX, EDX, EBX, ESP, and EBP. A 'Found intermodular calls' window is open, listing various calls with their addresses, disassemblies, and destinations. The destinations include 'kernel32.IsDebuggerPresent', 'kernel32.GetCurrentThreadId', and 'kernel32.GetTickCount64'. The disassembly for the call at 012C15F4 is highlighted as 'PUSH EBX'.

Address	Value	Comment
012C2000	011F02FB	
012C2004	011F06C2	
012C2008	011F07A0	
012C200C	011F07A5	
012C2010	011F0840	
012C2014	011F0997	
012C2018	011F099C	
012C201C	011F002E	
012C2020	00000000	

Address	Disassembly	Destination
012C100E	CALL 011F0000	
012C1077	CALL 663E70AE	
012C107F	CALL 011F099C	
012C10C2	CALL 66436E10	
012C10D9	CALL 66460F0F	
012C1113	CALL 663E8568	
012C11DC	CALL HelloWor.012C17C2	
012C1268	CALL 663F0F2B	
012C1282	CALL 663E6E8C	
012C12A3	CALL HelloWor.012C146C	
012C12C0	CALL 66437164	
012C12D0	CALL 663F0D9B	
012C12EA	CALL 011F0997	
012C132F	DB E8	
012C15F4	PUSH EBX	
012C15F5	CALL 011F06C2	
012C1604	CALL 011F07A0	
012C160D	CALL 011F002E	
012C161A	CALL 011F07A5	
012C1670	CALL 663D043F	
012C167A	CALL 011F099C	
012C16B2	CALL ESI	
012C16BF	CALL 663E6C84	
012C16FB	CALL ESI	
012C1738	CALL HelloWor.012C1870	
012C17A2	CALL HelloWor.012C1882	
012C1842	CALL HelloWor.012C1888	
012C205F	CALL 00005D62	

- Direct call resolution
 - If the call targets are in the constructed map from obfuscated addresses to API function, modify call targets to the original API function address
 - Generate a text file that contains resolved API function calls and OEP

- Indirect call resolution
 - Original segments (.text, .idata, ...) are merged into one segment by packing
 - Identify a memory block that contains successive obfuscated API function addresses
 - Modify obfuscated call addresses in the IAT candidate with the original API function

- Example: API Deobfuscation Information

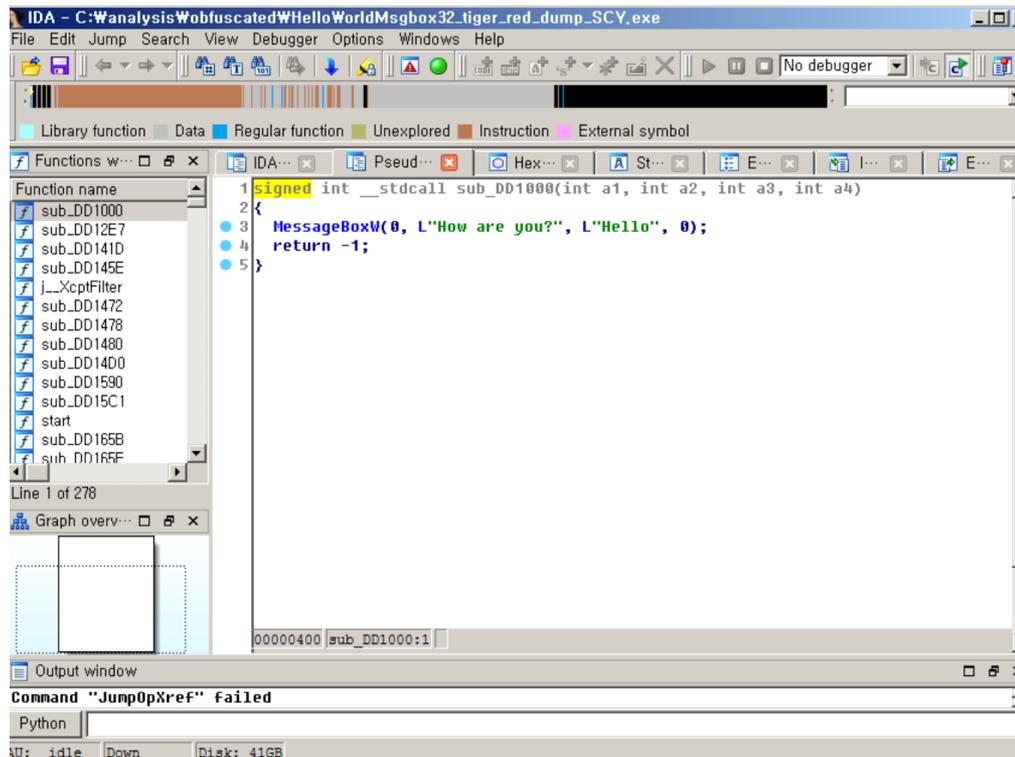
```
OEP:0000112d
00002000    addr ntdll.dll    RtlDecodePointer
00002004    addr kernel32.dll    GetSystemTimeAsFileTime
00002008    addr kernel32.dll    GetCurrentThreadId
0000200c    addr kernel32.dll    QueryPerformanceCounter
00002010    addr kernel32.dll    IsProcessorFeaturePresent
00002014    addr kernel32.dll    IsDebuggerPresent
00002018    addr ntdll.dll    RtlEncodePointer
0000201c    addr kernel32.dll    GetTickCount64
0000203c    addr ntdll.dll    RtlFreeHeap
0000209c    addr user32.dll    MessageBoxW
0000100e    call user32.dll    MessageBoxW
0000107f    call ntdll.dll    RtlEncodePointer
000012ea    call kernel32.dll    IsDebuggerPresent
000015f5    call kernel32.dll    GetSystemTimeAsFileTime
00001604    call kernel32.dll    GetCurrentThreadId
0000160d    call kernel32.dll    GetTickCount64
0000161a    call kernel32.dll    QueryPerformanceCounter
0000167a    call ntdll.dll    RtlEncodePointer
```

←

Addresses are in RVA

- Generating a debugger script to resolve API calls
 - The text file generated by the memory access analyzer contains OEP, resolved obfuscated addresses
 - Implemented a python script to generate a debugger script that execute until OEP and resolve obfuscated addresses

- Decompiled code with dumped file



```
IDA - C:\analysis\Wobfuscated\HelloWorldMsgbox32_tiger_red_dump_SCY.exe
File Edit Jump Search View Debugger Options Windows Help
Library function Data Regular function Unexplored Instruction External symbol
Functions window
Function name
sub_DD1000
sub_DD12E7
sub_DD141D
sub_DD145E
j_XcptFilter
sub_DD1472
sub_DD1478
sub_DD1480
sub_DD14D0
sub_DD1590
sub_DD15C1
start
sub_DD165B
sub_DD16FF
Line 1 of 278
Graph overview
00000400 sub_DD1000:1
Output window
Command "JumpOpXref" failed
Python
U: idle Down Disk: 41GB
```

```
1 signed int __stdcall sub_DD1000(int a1, int a2, int a3, int a4)
2 {
3     MessageBoxW(0, L"How are you?", L"Hello", 0);
4     return -1;
5 }
```



black hat[®]
USA 2015

API Deobfuscation for Static Obfuscators

API Deobfuscation for Static Obfuscators

- Static obfuscation pattern at OEP
 - Obfuscated call pattern
 - “Call qword ptr [____]” is changed into “Call rel32” when obfuscated
 - Obfuscated call run into API function
 - Stack shape is preserved
 - API call instruction and the first few instructions in the API function are obfuscated
 - After executing obfuscated instructions, execution reaches an instruction in the original API function

- Search obfuscated call by pattern
 - CALL rel32 – is a candidate
 - Check whether the address is in another section of the process

```
1000 48 83 EC 28      sub rsp,28
1004 4C 8D 05 BD 11 00 00  lea r8,qword ptr ds:[7FF688C321C8]
1008 48 8D 15 C6 11 00 00  lea rdx,qword ptr ds:[7FF688C321D8]
1012 45 33 C9         xor r9d,r9d
1015 33 C9          xor ecx,ecx
1017 E8 84 53 2C 00    call hw64_tmd233_tr.7FF688EF63A0
101C 00 83 C8 FF 48 83  add byte ptr ds:[rbx-7CB70038],al
1022 C4            ???
1023 28 C3         sub bl,al
```

```
10B9 E8 EF 11 2D 00    call hw64_tmd233_tr.7FF688F022AD
10BE 00 48 83         add byte ptr ds:[rax-7D],cl
10C1 C9            leave
10C2 FF E8         jmp far EA:0
10C4 01 B7 2C 00 00 48  add dword ptr ds:[rdi+4800002C],esi
```

Call rel32; db 00

'00' after call break alignment

so that a few incorrect disassembled code occur

.....

Obfuscated Call Resolution

- Obfuscated code is executed until API function
- Run-until-API method
 - Change RIP into candidate API call address
 - Run until API function

```
hw64_tmd233_tr.exe+000000000001017 call 0x7ff6bbbf63a0
hw64_tmd233_tr.exe+00000000002c63a0 sub rsp, 0x8
hw64_tmd233_tr.exe+00000000002c63a4 jmp 0x7ff6bbbf00b79
hw64_tmd233_tr.exe+00000000002d0b79 push r10
hw64_tmd233_tr.exe+00000000002d0b7b push r10
hw64_tmd233_tr.exe+00000000002d0b7d add qword ptr [rsp], 0x72384261
hw64_tmd233_tr.exe+00000000002d0b85 jmp 0x7ff6bbbf01d47
hw64_tmd233_tr.exe+00000000002d1d47 mov r10, qword ptr [rsp]
hw64_tmd233_tr.exe+00000000002d1d4b add rsp, 0x8
hw64_tmd233_tr.exe+00000000002d1d4f push rcx
hw64_tmd233_tr.exe+00000000002d1d50 jmp 0x7ff6bbbf01efc
hw64_tmd233_tr.exe+00000000002d1efc mov rcx, 0x72384261
```

← Obfuscated Call Start

.....

```
hw64_tmd233_tr.exe+00000000002ccf3b jmp 0x7ff6bbbf01830
hw64_tmd233_tr.exe+00000000002d1830 add rax, 0x5f6f805b
hw64_tmd233_tr.exe+00000000002d1836 pop r13
hw64_tmd233_tr.exe+00000000002d1838 xchg qword ptr [rsp], rax
hw64_tmd233_tr.exe+00000000002d183c jmp 0x7ff6bbbf022a8
hw64_tmd233_tr.exe+00000000002d22a8 mov rsp, qword ptr [rsp]
hw64_tmd233_tr.exe+00000000002d22ac ret
user32.dll+0000000000083b30 sub rsp, 0x38
user32.dll+0000000000083b34 xor r11d, r11d
user32.dll+0000000000083b37 cmp dword ptr [rip+0x155c6], r11d
```

← Execute until API address is met

Obfuscated Call Resolution

- Integrity check
 - We need to check whether the stack pointer and the stack content is preserved after executing obfuscate call

```

RIP → 00007FFA99493830 48 83 EC 38 sub rsp,38
00007FFA99493834 45 33 DB xor r11d,r11d
00007FFA99493837 44 39 1D C6 55 01 00 cmp dword ptr ds:[7FFA994A9104],r11d
00007FFA9949383E 74 2E je user32.7FFA9949386E
00007FFA99493840 65 48 88 04 25 30 00 00 mov rax,qword ptr gs:[30]
00007FFA99493849 4C 88 50 48 mov r10,qword ptr ds:[rax+48]
00007FFA9949384D 33 C0 xor eax,ebx
00007FFA9949384F F0 4C 0F B1 15 E8 6D 01 lock cmpxchg qword ptr ds:[7FFA994AA940]
00007FFA99493858 4C 88 15 E9 6D 01 00 mov r10,qword ptr ds:[7FFA994AA948]
00007FFA9949385F 41 8D 43 01 lea eax,dword ptr ds:[r11+1]
00007FFA99493863 4C 0F 44 D0 cmov r10,rax
00007FFA99493867 4C 89 15 DA 6D 01 00 mov qword ptr ds:[7FFA994AA948],r10
00007FFA9949386E 83 4C 24 28 FF or dword ptr ss:[rsp+28],FFFFFFFF
00007FFA99493873 66 44 89 5C 24 20 mov word ptr ss:[rsp+20],r11w
00007FFA99493879 E8 CE FE FF FF call <user32.MessageBoxTimeoutW>
00007FFA9949387E 48 83 C4 38 add rsp,38
00007FFA99493882 C3 ret
00007FFA99493883 90 nop
00007FFA99493884 90 nop
00007FFA99493885 90 nop
00007FFA99493886 90 nop
00007FFA99493887 90 nop

rsp=FC8B82F758
user32.d11[83830] | ".text":00007FFA99493830 <MessageBoxW>

Address Address Comments
00007FFA998B1000 0001493000014550
00007FFA998B1008 0001481000014A80
00007FFA998B1010 00014D0F000014DD0
00007FFA998B1018 0001521000014F40
00007FFA998B1020 0001541000015220
00007FFA998B1028 0001596000015780
00007FFA998B1030 00015F0000015C60
00007FFA998B1038 0001610000015F60
00007FFA998B1040 00016620000164F0

000000FC8B82F758 00007FF6B8C3101D ke4_tmd233_tr
000000FC8B82F760 000000FC8B82F768 000000FC8B88BD70
000000FC8B82F770 000000FC8B889920
000000FC8B82F778 0000000000000001
000000FC8B82F780 0000000000000000
000000FC8B82F788 00007FF6B8C312EB return to hw64
000000FC8B82F790 000000FC8B8A2AED
000000FC8B82F798 000000000000000A
000000FC8B82F7A0 01E6D7C6F0EDA448
    
```

Check Stack Pointer

Check Stack & Return Address

- Apply run-until API method repeatedly on candidate obfuscated calls
 - Save context & Restore

```
0000000000001017 call user32.dll MessageBoxW
00000000000010b9 call msvcr110.dll __set_app_type
00000000000010c3 call ntdll.dll RtlEncodePointer
000000000000110f call msvcr110.dll __setusermatherr
0000000000001121 call msvcr110.dll __configthreadlocale
0000000000001174 call msvcr110.dll __getmainargs
00000000000012fc call msvcr110.dll exit
000000000000130b call msvcr110.dll _cexit
000000000000132e call msvcr110.dll _ismbblead
000000000000135e call msvcr110.dll _exit
000000000000136e call msvcr110.dll _cexit
0000000000001395 call kernel32.dll IsDebuggerPresent
00000000000014fc goto msvcr110.dll XcptFilter
```

....

- Iterative run-until-API method can be applied to various packers
 - VMP: API function call is virtualization-obfuscated
 - Themida64: API function call is mutated
 - Obsidium: The first few instructions in an API function are obfuscated
 - Custom packers
 - But, at last, execution is redirected into a real API function

- Debugging x64 binary with x64DBG after deobfuscation

The screenshot shows the x64DBG debugger interface. The main window displays assembly code for the file 'hw64_tmd233_tr.exe'. The CPU window shows the following assembly instructions:

```
00007FF68BC31000 48 83 EC 28 sub rsp,28
00007FF68BC31004 4C 8D 05 BD 11 00 00 lea r8,qword ptr ds:[7FF68BC321C8]
00007FF68BC31008 48 8D 15 C6 11 00 00 lea rdx,qword ptr ds:[7FF68BC321D8]
00007FF68BC31012 45 33 C9 xor r9d,r9d
00007FF68BC31015 33 C9 xor ecx,ecx
00007FF68BC31017 FF 15 03 11 00 00 call qword ptr ds:[<&MessageBox>]
00007FF68BC3101D 83 C8 FF or eax,FFFFFFFF
00007FF68BC31020 48 83 C4 28 add rsp,28
00007FF68BC31024 C3 ret
00007FF68BC31025 CC int3
00007FF68BC31026 CC int3
00007FF68BC31027 CC int3
00007FF68BC31028 CC int3
00007FF68BC31029 CC int3
00007FF68BC3102A CC int3
00007FF68BC3102B CC int3
00007FF68BC3102C CC int3
00007FF68BC3102D CC int3
00007FF68BC3102E CC int3
00007FF68BC3102F CC int3
00007FF68BC31030 CC int3
00007FF68BC31031 CC int3
```

The register window shows 'eax=C128A22B'. The command window shows 'hw64_tmd233_tr.exe[101D] | "":00007FF68BC3101D'. The API call resolution window shows the following table:

Address	Address	Comments
00007FF68BC32000	00007FFA992D1340	kernel32.GetCurrentThreadId
00007FF68BC32008	00007FFA992D1610	kernel32.GetSystemTimeAsFileTime
00007FF68BC32010	00007FFA992D168C	kernel32.GetTickCount64
00007FF68BC32018	00007FFA992D2D60	kernel32.IsDebuggerPresent
00007FF68BC32020	00007FFA992D9004	kernel32.IsProcessorFeaturePresent
00007FF68BC32028	00007FFA992D1350	kernel32.QueryPerformanceCounter
00007FF68BC32030	00007FFA97115308	msvcrt.void __cdecl terminate(void)
00007FF68BC32038	00007FFA97123310	msvcrt._XcptFilter
00007FF68BC32040	00007FFA99328000	ntdll.C_specific_handler

Run correctly with resolved API call

- Dumping x86/64 binary and static analysis with IDA Pro

```

; Segment type: Pure code
; Segment permissions: Read/Write/Execute
___ segment para public 'CODE' use64
assume cs:___
;org 7FF6BBC31000h
assume es:nothing, ss:nothing, ds:___, fs:noth

; int __stdcall WinMain(HINSTANCE hInstance, H
WinMain proc near
sub    rsp, 28h
lea   r8, Caption    ; "Hello"
lea   rdx, Text      ; "How are you?"
xor   r9d, r9d       ; uType
xor   ecx, ecx       ; hWnd
call  cs:MessageBoxW
or   eax, 0FFFFFFFh
add   rsp, 28h
retn
WinMain endp
  
```

Line 1 of 48

Graph ov...

Output window

7FF6BBC322F0: using guessed type __int64 qword_7FF6BBC322F0[2];

Python

API call recovered

Address	Ordinal	Name	Library
00007FF6...		GetCurrentThreadId	kernel32
00007FF6...		GetSystemTimeAsFileTime	kernel32
00007FF6...		GetTickCount64	kernel32
00007FF6...		IsDebuggerPresent	kernel32
00007FF6...		IsProcessorFeaturePresent	kernel32
00007FF6...		QueryPerformanceCounter	kernel32
00007FF6...		?terminate@@@VAXXZ	msvcrt
00007FF6...		_XcptFilter	msvcrt
00007FF6...		_C_specific_handler	msvcrt
00007FF6...		_crtCapturePreviousContext	msvcrt110
00007FF6...		_crtGetShowWindowMode	msvcrt110
00007FF6...		_crtSetUnhandledExceptionFilter	msvcrt110
00007FF6...		_crtTerminateProcess	msvcrt110
00007FF6...		_crtUnhandledException	msvcrt110
00007FF6...		_crt_debugger_hook	msvcrt110
00007FF6...		_dllonexit	msvcrt
00007FF6...		_acmdln	msvcrt110
00007FF6...		_fmode	msvcrt110
00007FF6...		_commode	msvcrt110
00007FF6...		_getmainargs	msvcrt

IAT recovered

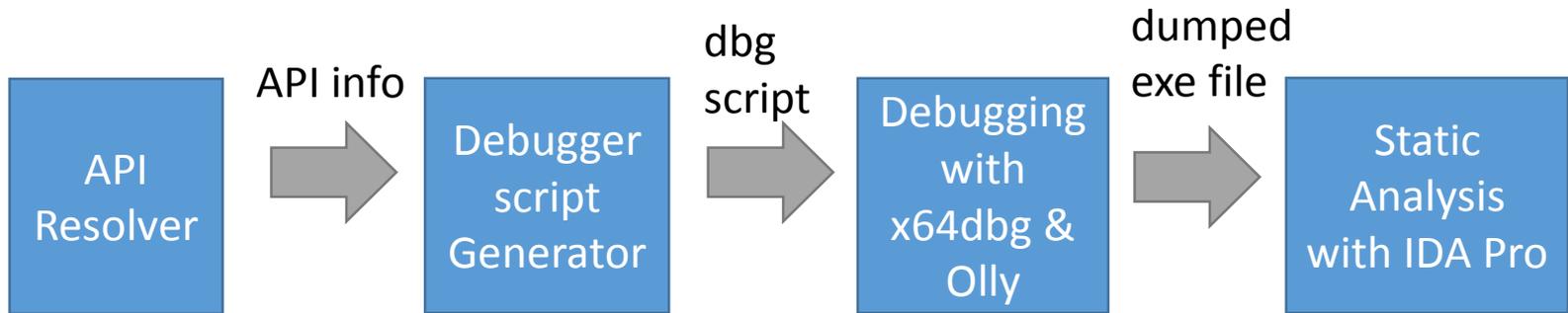


black hat[®]
USA 2015

Implementation

- Pin tool to resolve API Address
 - Windows 8.1/7 – 32/64 bit (on VMWare)
 - Visual Studio 2013
 - Intel Pin 2.14
- Python script to patch obfuscated call
- Reversing tools
 - X64dbg
 - IDA

Deobfuscation Process





black hat[®]
USA 2015

Demo

Reversing Packed Binary with API Deobfuscator

- Packed 32/64 bit samples
- Commercial packer packed 32bit malware

A large, stylized lightning bolt graphic in shades of blue and white, positioned on the right side of the slide. The bolt is jagged and appears to be striking downwards.

The Black Hat logo, featuring a white silhouette of a fedora hat inside a white circle.
black hat[®]
USA 2015

Conclusion

- Suggested two methods for API deobfuscation
 - Memory access analysis for dynamic obfuscation
 - Run-until-API method for static obfuscation
- Commercial packer protected binary can be analyzed using API deobfuscator
 - Using debugger
 - Using disassembler & decompiler

- Depending on DBI tools
 - Packers can detect DBI tools
 - Defeating the transparency feature of DBI (BH US'14)
 - Ex) Obsidium detect Intel Pin as a debugger
 - DBI tools crash in some applications
- Static whole function obfuscated code cannot be deobfuscated
 - No instructions in the original API function is executed when the whole function is obfuscated

- Anti-anti-debugging
 - Building x86/64 emulator for unpacking
- API function resolution
 - Code optimization and binary diffing for static whole function obfuscation
 - Backward dependence analysis for custom packers