

bobalkkagi 사용자 메뉴얼

TEAM 밥알까기

목차

| | |
|------------------|----|
| 1. bobalkkagi 소개 | 3 |
| 1.1. 기능 소개 | 3 |
| 1.2. 프로그램 구성 | 3 |
| 2. 사전 설치 사항 | 4 |
| 2.1. 프로그램 설치 | 4 |
| 3. 프로그램 사용 | 5 |
| 3.1. 사용 방법 | 6 |
| 4. 언패킹 기능 | 7 |
| 4.1. 기능 소개 | 7 |
| 4.2. 사용 방법 | 7 |
| 5. 디버깅 기능 | 10 |
| 5.1. 기능 소개 | 10 |
| 5.2. 사용 방법 | 10 |
| 6. 후킹 기능 | 14 |
| 6.1. 기능 소개 | 14 |
| 6.2. 사용 방법 | 14 |
| 7. 언래핑 기능 | 16 |
| 7.1. 기능 소개 | 16 |
| 7.2. 사용 방법 | 16 |

1. bobalkkagi 소개

Bobalkkagi 는 더미다라는 상용 패커를 통해 패킹된 프로그램을 기존 코드로 복구하기 위해 개발한 더미다 3.x 버전 언패킹 툴이다. TIGER RED 로 패킹된 프로그램에 초점을 잡아 분석을 진행하였고, 해당 옵션에서의 Anti-Debugger Detection, Entry Point Obfuscation, Advanced API-Wrapping 기능을 적용하여 해당 기능에 맞추어 언패킹 및 언래핑에 대한 자동화 도구를 개발하였다. 본 프로그램은 디버깅 및 후킹 기능까지 지원한다.

1.1. 기능 소개

| | 기능 | 설명 |
|---|------------|--------------------------|
| 1 | Unpacking | 패킹 되기 전 상태로 복구 |
| 2 | Debugging | Anti-Debugging 을 우회하여 분석 |
| 3 | Hooking | 특정 영역 Hooking |
| 4 | Unwrapping | 호출하는 API 의 복구 |

[표 1] 기능 소개

1.2. 프로그램 구성

Python 의 pip 를 이용하여 간단히 설치가 가능하며, github 에서 자세한 코드를 확인할 수 있다.

- 메모리에 로드할 DLL 목록 (win10_v1903)
- 다양한 옵션을 제공하여 많은 기능 활용

2. 사전 설치 사항

- Python3

Pip 를 통한 bobalkkagi 를 설치하기 위해 python3 의 설치가 필요하다. Python 설치 페이지()에서 설치 후, 환경 변수 등록 절차를 진행한다.



[그림 1]

- DLL 폴더

DLL 폴더를 만들어 메모리에 올라갈 DLL 목록을 저장하거나 bobalkkagi 도구에서 제공하는 DLL 파일()인 win10_v1903 폴더를 다운받아 진행한다.

2.1. 프로그램 설치

- pip 를 통한 설치

Python3 가 설치되어 있는 환경에서 [pip install bobalkkagi]를 통해 더미다 패키징 해제 자동화 프로그램인 bobalkkagi 를 설치한다.

```
pip install bobalkkagi
```

[그림 2] pip install

3. 프로그램 사용

- Default

기본적인 사용법으로 bobalkkagi PROTECTEDFILE 명령을 통해 패킹된 프로그램을 bobalkkagi 로 언패킹 및 언래핑을 진행할 수 있다.

- 옵션 (FLAGS)

--mode=MODE

에뮬레이션의 모드를 뜻하며, fast(f), hook_code(c), hook_block(b)로 나뉜다.

fast 모드는 함수 영역을 0x20 만큼 rip 와 비교하는 모드이며, hook_code 와 hook_block 은 매핑되는 모든 DLL 메모리와 rip 를 비교하는 모드이다.

--verbose=VERBOSE

Hooking 되는 API CALL 정보를 보여주는 옵션이다.

--dllPath=DLLPATH

윈도우 버전에 맞게 메모리에 로드할 DLL 목록을 지정하여 후킹할 DLL 을 불러온다.

--oep=OEP

oep 를 찾는 옵션으로 Hook_code 와 hook_block 에서만 해당 옵션을 종료할 수 있다.

--debugger=DEBUGGER

다른 버전이나 패커의 사용을 위한 옵션으로 hook_code 에서만 사용가능하다.

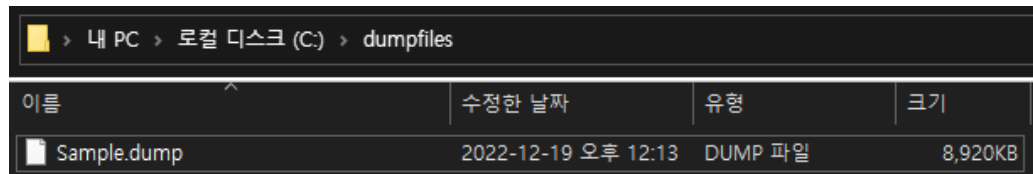
3.1. 사용 방법

- 명령 프롬프트(CMD)에서 위 옵션에 맞춰 bobalkkagi 를 실행한다.

```
c:\W>bobalkkagi Sample.exe --dllPath=win10_v1903
```

[그림 3] 기본명령

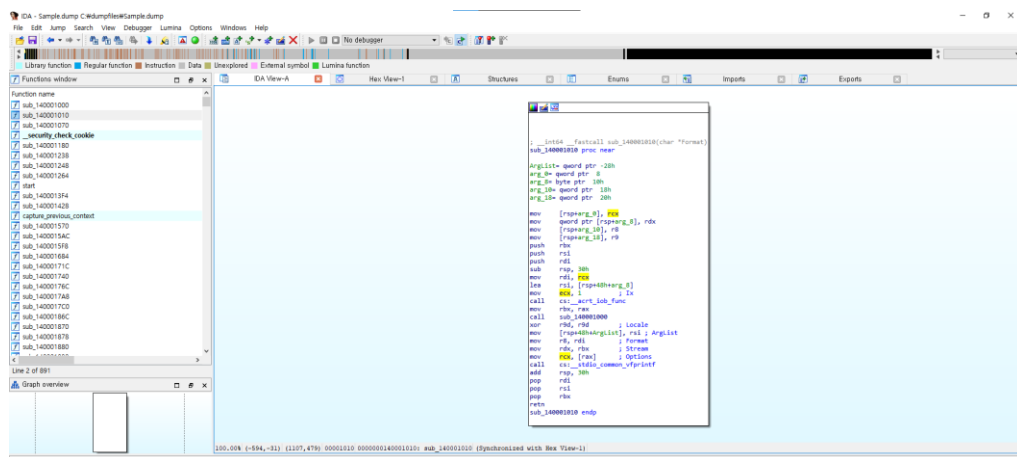
- 언패킹과 언래핑이 종료된 후 덤프 파일의 생성을 확인한다.



| 이름 | 수정한 날짜 | 유형 | 크기 |
|-------------|---------------------|---------|---------|
| Sample.dump | 2022-12-19 오후 12:13 | DUMP 파일 | 8,920KB |

[그림 4] 덤프파일

- 분석 프로그램을 활용하여 분석을 진행한다



[그림 5] 분석

4. 언패킹 기능

4.1. 기능 소개

더미다 3.x 버전으로 패킹된 x64 바이너리를 언패킹하는 기능이며, 언패킹 루팅을 실행하여 Original Entry Point 를 찾은 후 실행 직전의 dump 파일을 생성한다.

code, block, fast 모드가 존재하면 각 모드의 상세는 다음과 같다.

code: 모든 Instruction 정보를 기록하며 언패킹을 진행한다. Debugging 시 가장 적절한 모드이며 실행 옵션 중 mode = c 를 통해 실행한다.

block: jmp, call 등 block 단위의 Instruction 정보를 기록하며 언패킹을 진행한다. Debugging 시 해당 모드를 권장하지 않으며 옵션 중 mode = b 를 통해 실행한다.

fast: Instruction 정보를 기록하지 않고 언패킹을 진행하며, 실행 옵션 중 mode = f 를 통해 실행한다.

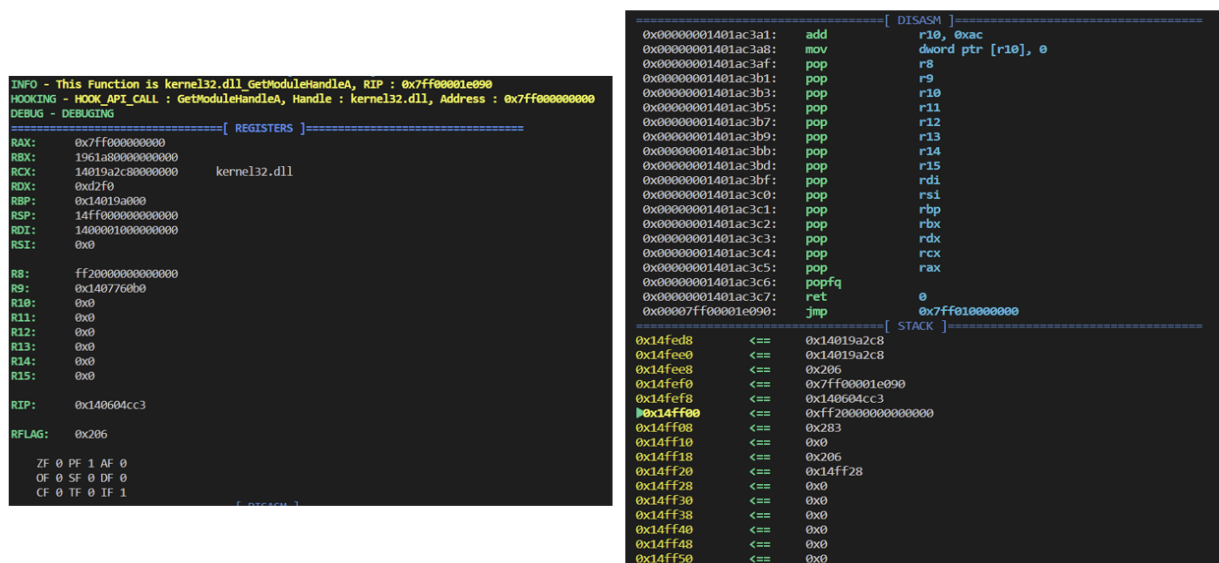
4.2. 사용 방법

bobalkkagi tool 실행시 원하는 mode 를 적용하여 실행한다.

```
\bobalkkagi> python main.py ..\testfiles\putty_protected.exe c T ..\win10_v1903 t t
```

[그림 6] hook_code mode 선택

- code 모드를 적용하게 되면 Register, Instruction, Stack 등의 상세정보를 확인할 수 있다.



The image contains two screenshots from a debugger. The left screenshot shows the 'DEBUG - DEBUGGING' window with the following information:

```
INFO - This Function is kernel32.dll.GetModuleHandleA, RIP : 0x7ff00001e990
HOOKING - HOOK_API_CALL : GetModuleHandleA, Handle : kernel32.dll, Address : 0x7ff000000000
DEBUG - DEBUGGING

===== [ REGISTERS ] =====
RAX: 0x7ff000000000
RBX: 1961a80000000000
RCX: 14019a2c00000000 kernel32.dll
RDX: 0xd2f0
RBP: 0x14019a000
RSP: 14ff000000000000
RDI: 1400001000000000
RSI: 0x0
R8: ff20000000000000
R9: 0x1407760b0
R10: 0x0
R11: 0x0
R12: 0x0
R13: 0x0
R14: 0x0
R15: 0x0
RIP: 0x140604cc3
RFLAG: 0x206
ZF 0 PF 1 AF 0
OF 0 SF 0 DF 0
CF 0 TF 0 IF 1
```

The right screenshot shows the 'DISASM' window with the following assembly code:

```
0x00000001401ac3a1: add     r10, 0xac
0x00000001401ac3a8: mov     dword ptr [r10], 0
0x00000001401ac3af: pop     r8
0x00000001401ac3b1: pop     r9
0x00000001401ac3b3: pop     r10
0x00000001401ac3b5: pop     r11
0x00000001401ac3b7: pop     r12
0x00000001401ac3b9: pop     r13
0x00000001401ac3bb: pop     r14
0x00000001401ac3bd: pop     r15
0x00000001401ac3bf: pop     rdi
0x00000001401ac3c0: pop     rsi
0x00000001401ac3c1: pop     rbp
0x00000001401ac3c2: pop     rbx
0x00000001401ac3c3: pop     rdx
0x00000001401ac3c4: pop     rcx
0x00000001401ac3c5: pop     rax
0x00000001401ac3c6: popfq
0x00000001401ac3c7: ret
0x00007fff00001e990: jmp     0x7ff010000000

===== [ STACK ] =====
0x14fed8 <== 0x14019a2c8
0x14fee0 <== 0x14019a2c8
0x14fee8 <== 0x206
0x14fef0 <== 0x7ff00001e990
0x14fef8 <== 0x140604cc3
0x14ff00 <== 0xff20000000000000
0x14ff08 <== 0x283
0x14ff10 <== 0x0
0x14ff18 <== 0x206
0x14ff20 <== 0x14ff28
0x14ff28 <== 0x0
0x14ff30 <== 0x0
0x14ff38 <== 0x0
0x14ff40 <== 0x0
0x14ff48 <== 0x0
0x14ff50 <== 0x0
```

[그림 7] 상세정보 확인 가능

- block 모드를 적용하면 언패킹시 실행된 함수들 전부를 확인할 수 있다.

```
INFO - This Function is ntdll.dll_ZwQueryInformationProcess, RIP : 0x7ff00014e3e0
HOOKING - HOOK_API_CALL : ZwQueryInformationProcess
INFO - This Function is ntdll.dll_RtlAcquireSRWLockExclusive, RIP : 0x7ff0000eb4a0
INFO - This Function is ntdll.dll_RtlReleaseSRWLockExclusive, RIP : 0x7ff0000e7cb0
INFO - This Function is ntdll.dll_RtlAcquireSRWLockExclusive, RIP : 0x7ff0000eb4a0
INFO - This Function is ntdll.dll_RtlReleaseSRWLockExclusive, RIP : 0x7ff0000e7cb0
INFO - This Function is ntdll.dll_RtlAcquireSRWLockExclusive, RIP : 0x7ff0000eb4a0
INFO - This Function is ntdll.dll_RtlReleaseSRWLockExclusive, RIP : 0x7ff0000e7cb0
INFO - This Function is ntdll.dll_LdrControlFlowGuardEnforced, RIP : 0x7ff0000ccf10
INFO - This Function is kernel32.dll_SetCurrentDirectoryW, RIP : 0x7ff00001f1d0
HOOKING - HOOK_API_CALL : SetCurrentDirectoryW
INFO - This Function is kernel32.dll_SetCurrentDirectoryW, RIP : 0x7ff00001f1d0
HOOKING - HOOK_API_CALL : SetCurrentDirectoryW
INFO - This Function is ntdll.dll_ZwSetInformationThread, RIP : 0x7ff00014e260
HOOKING - HOOK_API_CALL : ZwSetInformationThread
INFO - This Function is ntdll.dll_ZwQueryInformationProcess, RIP : 0x7ff00014e3e0
HOOKING - HOOK_API_CALL : ZwQueryInformationProcess
INFO - This Function is ntdll.dll_ZwQueryInformationProcess, RIP : 0x7ff00014e3e0
HOOKING - HOOK_API_CALL : ZwQueryInformationProcess
INFO - This Function is kernel32.dll_VirtualProtect, RIP : 0x7ff00001af90
HOOKING - HOOK_API_CALL : VirtualProtect, Address : 0x7ff00017f490, Size : 0x1000, Privilege : 0x40
INFO - This Function is kernel32.dll_IsBadWritePtr, RIP : 0x7ff00005e280
INFO - This Function is kernel32.dll_VirtualProtect, RIP : 0x7ff00001af90
HOOKING - HOOK_API_CALL : VirtualProtect, Address : 0x7ff00017f490, Size : 0x1000, Privilege : 0x20
INFO - This Function is kernel32.dll_VirtualProtect, RIP : 0x7ff00001af90
HOOKING - HOOK_API_CALL : VirtualProtect, Address : 0x7ff000151ae0, Size : 0x1000, Privilege : 0x40
```

[그림 8] 언패킹 시 실행된 함수 정보 출력

- fast 모드를 적용하면 언패킹에 필요한 후킹 함수 정보만이 출력된다.

```
HOOKING - HOOK_API_CALL : ZwQueryInformationProcess
HOOKING - HOOK_API_CALL : ZwQueryInformationProcess
HOOKING - HOOK_API_CALL : VirtualProtect, Address : 0x7ff00017f490, Size : 0x1000, Privilege : 0x40
HOOKING - HOOK_API_CALL : VirtualProtect, Address : 0x7ff00017f490, Size : 0x1000, Privilege : 0x20
HOOKING - HOOK_API_CALL : VirtualProtect, Address : 0x7ff000151ae0, Size : 0x1000, Privilege : 0x40
HOOKING - HOOK_API_CALL : RtlAllocateHeap, handle : 0x1e9e3850000, RAX : 0x1e9e3850000
HOOKING - HOOK_API_CALL : RtlFreeHeap, handle : 0x1e9e3850000,
HOOKING - HOOK_API_CALL : NtUserGetForegroundWindow
HOOKING - HOOK_API_CALL : GetWindowTextA
HOOKING - HOOK_API_CALL : OpenThreadToken
HOOKING - HOOK_API_CALL : OpenProcessToken, token : 0x1f8
HOOKING - HOOK_API_CALL : ZwQueryInformationToken, token : 0x159
HOOKING - HOOK_API_CALL : ZwAllocateVirtualMemory, Address : 0x20000000000, Size : 0x159, Privilege : 0x4
HOOKING - HOOK_API_CALL : ZwQueryInformationToken, token : 0x1a4
HOOKING - HOOK_API_CALL : ZwSetInformationProcess
HOOKING - HOOK_API_CALL : ZwClose, handle : 0x1f8
HOOKING - HOOK_API_CALL : RtlAllocateHeap, handle : 0x1e9e3850000, RAX : 0x1e9e3850000
HOOKING - HOOK_API_CALL : RtlFreeHeap, handle : 0x1e9e3850000,
HOOKING - HOOK_API_CALL : VirtualFree, Address : 0x20000000000
HOOKING - HOOK_API_CALL : GetProcAddress, shell32.dll_IsUserAnAdmin: 0x7ff0012f6d10
HOOKING - HOOK_API_CALL : RtlAllocateHeap, handle : 0x1e9e3850000, RAX : 0x1e9e385a000
HOOKING - HOOK_API_CALL : ZwOpenThreadTokenEx
HOOKING - HOOK_API_CALL : ZwOpenProcessTokenEx, token : 0x13d
HOOKING - HOOK_API_CALL : ZwDuplicateToken
HOOKING - HOOK_API_CALL : ZwClose, handle : 0x13d
HOOKING - HOOK_API_CALL : ZwAccessCheck
HOOKING - HOOK_API_CALL : RtlFreeHeap, handle : 0x1e9e3850000,
HOOKING - HOOK_API_CALL : ZwAllocateVirtualMemory, Address : 0x20000001000, Size : 0x8, Privilege : 0x4
HOOKING - HOOK_API_CALL : ZwGetContextThread
HOOKING - HOOK_API_CALL : VirtualFree, Address : 0x20000001000
```

[그림 9] 필요한 후킹 함수 정보 출력

언패킹 실행 시 다음과 같은 Error 가 발생하는 경우 DLL 로드와 관련이 있다. 더미다로 패킹한 파일의 IAT 정보들이 일부 지워지기 때문에 언패킹에 필요한 dll 이 로드되지 않는다.

```
address =DLL_SETTING.DllFuncs[key]
KeyError: 'win32u.dll_NtUserGetForegroundWindow'
```

[그림 10] dll 로드 error

해당 Error 를 해결하기 위해서는 unpacking.py 에 필요한 dll 을 다음과 같은 방식으로 추가하면 된다.

```
PE_Loader(uc, program, GLOBAL_VAR.ImageBaseStart, oep)
PE_Loader(uc, "user32.dll", GLOBAL_VAR.DllEnd, False) # dll 추가
```

[그림 11] 수동 dll 로드

로드해야할 dll 이 win10_v1903 폴더에 존재하지 않는 경우 다음과 같은 Error 메시지가 출력된다.

wsock32.dll 이 로드되지 않아 해당 dll 의 Address 가 0x0 인걸 확인할 수 있다. 해당 Error 는 필요한 dll 을 win10_v1903 폴더에 추가하거나 해당 dll 이 존재하는 폴더를 dllPath 로 지정하면 해결할 수 있다.

```
ERROR - wsock32.dll is not loaded. Please check the following 2 things.
1. Check if wsock32.dll exists in ..\win10_v1903
2. Check if wsock32.dll is included in the default dll.

HOOKING - HOOK_API_CALL : GetModuleHandleA, Handle : wsock32.dll, Address : 0x0
INFO - This Function is kernel32.dll_GetModuleHandleA, RIP : 0x7ff00001e090
```

[그림 12] dll 목록 폴더 error

5. 디버깅 기능

5.1. 기능 소개

언패킹 실행 중 메모리 접근, 핸들, 권한 등의 문제 발생시 언패킹 기능이 제대로 동작하지 않는다. 따라서 디버깅 기능을 통해 오류 원인 파악 및 수정 기능을 제공해야 한다.

5.2. 사용 방법

bobalkkagi tool 실행시 debugger 옵션을 적용하여 실행한다.

```
\bobalkkagi> python main.py ..\testfiles\putty_protected.exe c T ..\win10_v1903 t t
```

[그림 13] 디버깅 옵션

디버깅 기능이 적용되면 h(help)명령어를 통해 사용가능한 명령어를 확인할 수 있다.

```
UNICORN DEBUG > h

===== help =====
q: quit
n: next
c: continue
s: search dll function by address
sf: search dll function by name
w: write memory
set: set register
bp: breakpoint
bl: breakpoint list
view: view memory
h: help
=====
```

[그림 14] 디버깅 기능

- n(next): 다음 Instruction 을 실행시키는 명령이다.



[그림 15] 디버깅 next

- s, sf: 주소와 이름으로 dll 의 함수를 찾는 명령어이다. s 는 주소로, sf 는 이름으로 해당 함수를 찾는다.

```
UNICORN DEBUG > s
address : 0x7ff00001a310
result: kernel32.dll_GetProcAddress
```

Search by address

```
UNICORN DEBUG > sf
dll_function: kernel32.dll_GetProcAddress
result: 0x7ff00001a310
```

Search by name

[그림 16] 디버깅 dll 함수 찾기

- view: 메모리에 쓰여있는 값을 확인하는 명령이며, 확인하고 싶은 주소와 사이즈를 입력하여 메모리의 값을 확인한다.

```
address(64bit size): 0x14fee0
size(ex. 0x1234): 0x30
00000000014fee0: 000000014019a2c8 0000000000000000
00000000014fef0: 0000000000000000 0000000000000206
00000000014ff00: ff20000000000000 0000000000000283
```

```
0x14fed8 <== 0x14019a2c8
0x14fee0 <== 0x14019a2c8
0x14fee8 <== 0x0
0x14fef0 <== 0x0
0x14fef8 <== 0x206
0x14ff00 <== 0xff20000000000000
0x14ff08 <== 0x283
0x14ff10 <== 0x0
```

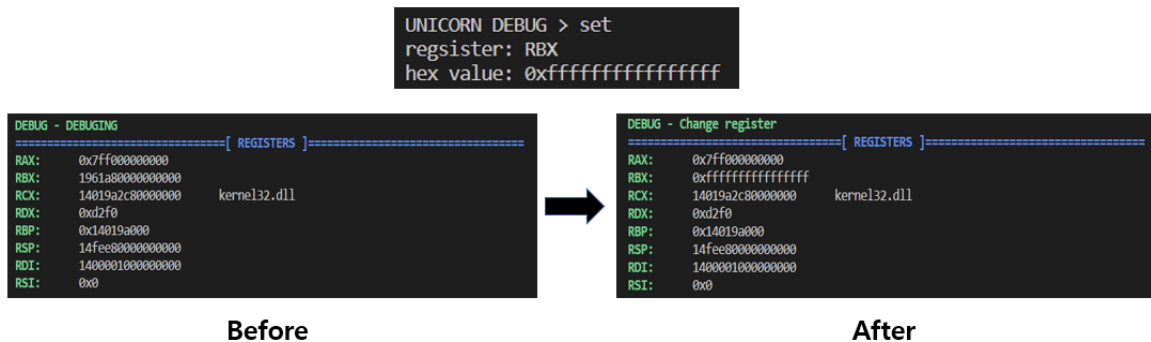
[그림 17] 디버깅 메모리 확인

- w(write): 메모리에 값을 쓰는 명령어이다. 쓰고싶은 주소와 값을 입력하여 메모리의 값을 변경하며, 바뀌기 이전값도 확인할 수 있다.

```
UNICORN DEBUG > w
address: 0x14fef0
Value: 0xffffffffffffffff
before: 0
Changed: 18446744073709551615
UNICORN DEBUG > view
address(64bit size): 0x14fee0
size(ex. 0x1234): 0x30
00000000014fee0: 000000014019a2c8 0000000000000000
00000000014fef0: ffffffffffffffffffff 0000000000000206
00000000014ff00: ff20000000000000 0000000000000283
```

[그림 18] 디버깅 메모리 작성

- set: Register 의 값을 변경하는 명령이다. Register 와 값을 입력하여 해당 Register 의 값을 변경한다.



[그림 19] 디버깅 레지스터 변경

- bp(break point): 원하는 주소에 break point 를 거는 명령이다.
- bl(break point list): bp 가 걸린 리스트를 확인하는 명령이다.

```

UNICORN DEBUG > bp
address: 0x7ff00001a310
UNICORN DEBUG > bl
bp0: 00007ff00001a310
UNICORN DEBUG > bp
address: 0x140604ccf
UNICORN DEBUG > bl
bp0: 00007ff00001a310
bp1: 0000000140604ccf

```

[그림 20] 디버깅 bp

- c(continue): 다음 break point 까지 실행하는 명령어이다. bp 를 걸었던 0x7ff00001a310 주소인 kernel32.dll_GetProcAddress 까지 실행됨을 확인할 수 있다.

```

UNICORN DEBUG > c
INFO - This Function is kernel32.dll_LoadLibraryA, RIP : 0x7ff00001eb60
HOOKING - HOOK_API_CALL : LoadLibraryA, user32.dll: 0x7ff0008d6000
INFO - This Function is kernel32.dll_LoadLibraryA, RIP : 0x7ff00001eb60
HOOKING - HOOK_API_CALL : LoadLibraryA, advapi32.dll: 0x7ff000545000
INFO - This Function is kernel32.dll_LoadLibraryA, RIP : 0x7ff00001eb60
HOOKING - HOOK_API_CALL : LoadLibraryA, ntdll.dll: 0x7ff0000b2000
INFO - This Function is kernel32.dll_LoadLibraryA, RIP : 0x7ff00001eb60
HOOKING - HOOK_API_CALL : LoadLibraryA, shell32.dll: 0x7ff0010ab000
INFO - This Function is kernel32.dll_LoadLibraryA, RIP : 0x7ff00001eb60
HOOKING - HOOK_API_CALL : LoadLibraryA, shlwapi.dll: 0x7ff001875000
INFO - This Function is kernel32.dll_GetProcAddress, RIP : 0x7ff00001a310
HOOKING - HOOK_API_CALL : GetProcAddress, kernel32.dll_SetLastError: 0x7ff000016a60
DEBUG - DEBUGGING
===== [ REGISTERS ] =====
RAX:      0x7ff000016a60
RBX:      0xffffffffffffffff
RCX:      0x7ff000000000
RDX:      1401f8a3b0000000      SetLastError
RBP:      0x14019a000
RSP:      0x14ff00
RDI:      1400001000000000
RSI:      0x0

```

[그림 21] 디버깅 bp 까지 실행

- q(quit): 디버깅을 종료하는 명령어이다. 디버깅을 종료하고 언패킹을 계속 실행한다.

```

UNICORN DEBUG > q
FINISHED DEBUG
INFO - This Function is ntdll.dll_RtlInitializeCriticalSection, RIP : 0x7ff0001150b0
INFO - This Function is kernel32.dll_GetCurrentThreadId, RIP : 0x7ff000015e50
INFO - This Function is kernel32.dll_OpenThread, RIP : 0x7ff00001bcd0
INFO - This Function is kernelbase.dll_OpenThread, RIP : 0x7ff0002cfd90
INFO - This Function is ntdll.dll_ZwOpenThread, RIP : 0x7ff0001505d0
HOOKING - HOOK_API_CALL : ZwOpenThread, handle : 0x11d
INFO - This Function is kernel32.dll_GetUserDefaultUILanguage, RIP : 0x7ff00001ef10
HOOKING - HOOK_API_CALL : GetUserDefaultUILanguage, RCX : 0x14ff18
INFO - This Function is kernel32.dll_GetProcessHeap, RIP : 0x7ff000016a50
INFO - This Function is kernelbase.dll_GetProcessHeap, RIP : 0x7ff0002f2a50
INFO - This Function is ntdll.dll_RtlAllocateHeap, RIP : 0x7ff0000ed870
HOOKING - HOOK_API_CALL : RtlAllocateHeap, handle : 0x1e9e3850000, RAX : 0x1e9e3851000
INFO - This Function is kernel32.dll_GetProcessHeap, RIP : 0x7ff000016a50
INFO - This Function is kernelbase.dll_GetProcessHeap, RIP : 0x7ff0002f2a50

```

[그림 22] 디버깅 종료

6. 후킹 기능

6.1. 기능 소개

안티 디버깅 우회나 특정 함수 동작을 제어하기 위해서는 해당 함수를 후킹할 필요가 있다.

다음의 설명을 통하여 후킹할 함수를 bobalkkagi tool 에 추가할 수 있다.

6.2. 사용 방법

hookFuncs.py 폴더에 후킹하고자 하는 함수 정보를 추가한다. "dllName_funcName" : number 형식으로 추가한다.

```
HookFuncs={
    "kernel32.dll_GetModuleHandleA" : 0,
    "kernel32.dll_LoadLibraryA" : 1,
    "kernel32.dll_GetProcAddress" : 2,
    "ntdll.dll_ZwOpenThread" : 3,
    "kernel32.dll_GetUserDefaultUILanguage" : 4,
    "ntdll.dll_RtlAllocateHeap" : 5,
    "kernel32.dll_GetCurrentDirectoryW" : 6,
    "kernel32.dll_GetModuleFileNameW" : 7,
    "kernel32.dll_SetCurrentDirectoryW" : 8,
    "kernel32.dll_GetCommandLineA" : 9,
    "ntdll.dll_ZwQueryInformationProcess" : 10,
    "ntdll.dll_ZwAllocateVirtualMemory" : 11,
    "ntdll.dll_ZwGetContextThread" : 12,
    "kernel32.dll_VirtualFree" : 13,
    "ntdll.dll_ZwSetInformationThread" : 14,
    "advapi32.dll_OpenThreadToken" : 15,
    "advapi32.dll_OpenProcessToken" : 16,
```

[그림 23] 후킹 함수 정보

api_hook.py 폴더에 후킹한 함수의 동작을 설정한다. 다음은 후킹함수의 예시이다.

```
def hook_exampleFunction(uc, log, regs):  
    set_register(regs)  
  
    uc.reg_read(UC_X86_REG_RDX)  
    uc.reg_write(UC_X86_REG_RAX, 0x1)  
    uc.mem_read(REGS.rdx, 0x20)  
    uc.mem_write(REGS.r8, struct.pack('<Q', 0x1))  
  
    ret(uc, REGS.rsp)
```

[그림 24] 후킹 함수 예시

set_register(regs)는 Register 정보를 가져오는 함수이다. REGS.{Register} 로 원하는 Register 에 접근할 수 있다.

ret(uc, REGS.rsp)는 어셈블리어의 ret 명령어를 구현한 함수이다.

특정 레지스터 값 읽기: uc.reg_read(UC_X86_REG_{Register})

특정 레지스터에 값 쓰기: uc.reg_write(UC_X86_REG_{Register}, Value)

특정 메모리 값 읽기: uc.mem_read(Address, Size)

특정 메모리에 값 쓰기: uc.mem_write(Address, Bytes)

7. 언래핑 기능

7.1. 기능 소개

Advanced API-Wrapping 기능을 적용시킨 패킹 프로그램은 API 를 호출할 때 API 함수 분석을 어렵게 하기위해 여러 번 API 호출을 포장한다. 언래핑 기능은 이러한 Wrapping 패턴을 파악하여 Advanced API-Wrapping 이 적용된 파일인 경우 해당 API 를 복구하여 분석을 쉽게 할 수 있도록 도움을 주는 기능이다.

7.2. 사용 방법

bobalkkagi 의 언래핑 자동화 부분 실행 시 코드에서 적용한 Wrapping 패턴을 찾아 언래핑을 실행한다.

- wrapping 패턴을 찾는 함수는 pattern_target 으로 해당 영역에서 틀에 사용된 패턴 정보를 상세히 확인할 수 있다.

```
def pattern_target():
    assem = Decode(GLOBAL_VAR.text[0], origin_data[GLOBAL_VAR.text[0]:GLOBAL_VAR.text[0]+GLOBAL_VAR.text[1]], Decode64Bits)

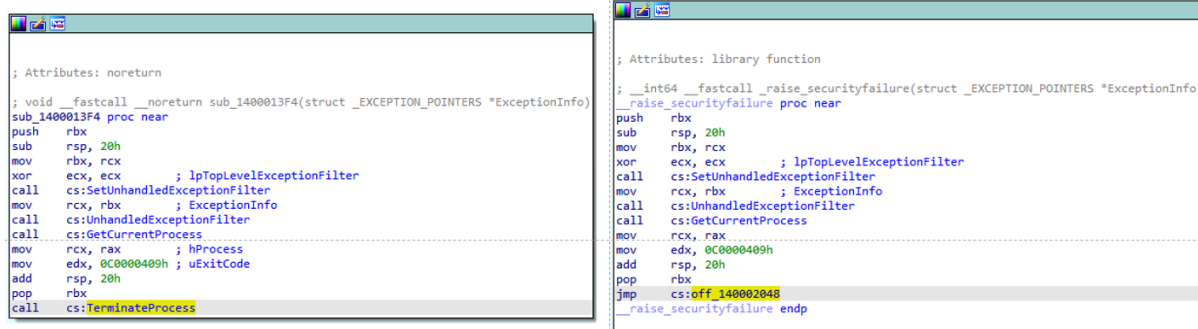
    for asm in assem:
        if asm[2].split(" ")[0] == "CALL":
            # instruction format
            if(asm[1] == 6):
                calloffset_pattern[asm[0]]=1
            elif(asm[1] == 5):
                calloffset_pattern[asm[0]]=2
            else:
                calloffset_pattern[asm[0]]=0

        if asm[2].split(" ")[0] == "MOV":
            mov_offset.append(asm[0])

        if asm[2].split(" ")[0] == "JMP":
            jmp_offset.append(asm[0])
```

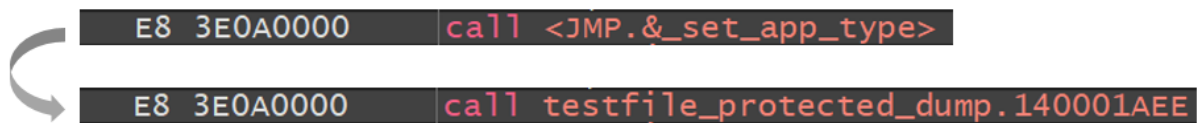
[그림 25] wrapping 패턴 함수

- 실제 wrapping 이 되어있지만 wrapping 해제 패턴이 적용되어 있지않으면 아래 오른쪽 그림과 같이 wrapping 이 그대로 남아 어떤 함수인지 확인이 안된다.



[그림 26]] 패턴을 파악하지 못한 Error

- 위에서 미리 설정된 패턴 이외의 패턴은 아래와 같이 패킹이 되지 않은 프로그램과 패킹이 된 프로그램의 분석을 통해 API 가 보이지 않는 부분을 확인한다.



[그림 27] Wrapping 패턴 파악

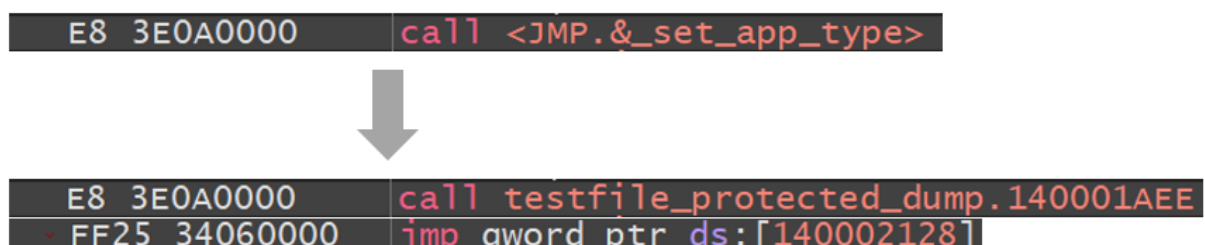
- wrapping 패턴이 확인을 찾지 못하는 경우 아래와 같이 프로텍트된 파일에서 존재하지 않는 API 를 원본 파일과 비교하여 offset 을 찾아 해당 주소에 있는 명령을 위와 같은 방법으로 확인하여 wrapping 패턴을 추가한다.

| | | | | | |
|-------|--------------|---|-------|---|---|
| E9AEE | kernel32.dll | 1 | FALSE | 0 | 0 |
| E9B02 | GDI32.dll | 1 | FALSE | 0 | 0 |
| E9B16 | IMM32.dll | 1 | FALSE | 0 | 0 |
| E9B2A | ole32.dll | 1 | FALSE | 0 | 0 |
| E9B3E | USER32.dll | 1 | FALSE | 0 | 0 |
| E9B52 | SHELL32.dll | 1 | FALSE | 0 | 0 |
| E9B66 | COMDLG32.dll | 1 | FALSE | 0 | 0 |
| E9B7A | ADVAPI32.dll | 1 | FALSE | 0 | 0 |

| kernel32.dll [1 entry] | | | | | |
|--------------------------|-----------------|---------|----------------|--------|-----------|
| Call via | Name | Ordinal | Original Thunk | Thunk | Forwarder |
| 1961A8 | GetModuleHan... | - | - | 19600D | - |

[그림 28] api 개수 분석

- wrapping 패턴이 덮고자 하는 명령어보다 크기가 크다면, 아래와 같이 wrapping 특성을 활용하여 패커에서 생성한 특정 부분에 원하는 명령을 넣는 방식으로 wrapping 을 해제한다.



[그림 29] wrapping 특성을 이용한 명령어 크기 Error 해결방안

- wrapping 패턴 중 복구가 되지않는 API 가 있어 확인하였는데 GetExitCodeProcess 함수를 사용하여 프로세스 종료값을 검색하는 동작이기에 해당 명령을 실행하는 부분이 에러가 터지지 않아 후킹을 걸지 못하는 문제가 발생하였다.


ExitProcess function (processthreadsapi.h)

Article • 11/02/2022 • 2 minutes to read

 Feedback

Ends the calling process and all its threads.

Syntax

```
C++  Copy
```

```
void ExitProcess(  
    [in] UINT uExitCode  
);
```

Parameters

```
[in] uExitCode
```

The exit code for the process and all threads.

Return value

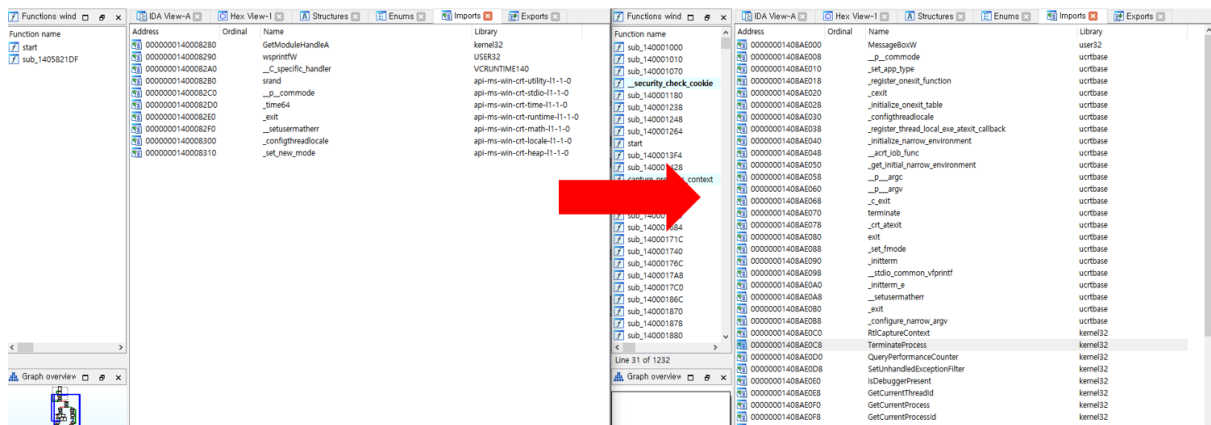
None

Remarks

Use the `GetExitCodeProcess` function to retrieve the process's exit value. Use the `GetExitCodeThread` function to retrieve a thread's exit value.

[그림 30] ExitProcess function

- 모든 패턴을 찾고 wrapping 을 해제한다면 아래 그림과 같이 IAT 가 복구되어 API 를 어떠한 것을 가져오는지 쉽게 알 수 있다.



[그림 31] 언래핑 완료, API 복구