한국정보전자통신기술학회논문지(jkiiect)'18-12, Vol.11 No.6
논문 18-11-06-724 http://dx.doi.org/10.17661/jkiiect.2018.11.6.724

가상화 기반 난독화 및 역난독화를 위한 코드 자동 분석 기술

김 순 곤*

Code Automatic Analysis Technique for Virtualization-based Obfuscation and Deobfuscation

Soon-Gohn Kim*

요 약 코드 난독화는 프로그램을 해석하거나 위조 또는 변조 방지를 목적으로 프로그램을 쉽게 이해할 수 없도록하는 기술이다. 역난독화는 난독화된 프로그램을 입력으로 받아 원 소스의 의미를 역공학 기술을 통해 분석하는 기술이다. 본 논문은 가상화 기반 환경에서 바이너리 코드에 대한 난독화 및 역난독화 기술에 대한 분석 연구이다. VMAttack를 기반으로 정적 코드분석, 동적 코드분석, 최적화 기법에 대한 구체적인 분석을 통해 난독화 및 역난독화 기술을 구체적으로 분석한 후 실레 바이너리 코드에 대해 난독화와 역난독화 기술을 실험하였다. 본 논문를 통하여 다양한 가상화, 난독화에 대한 연구를 진행할 수 있을 것으로 기대된다. 특히, 스택-기반 가상 머신에서 연구한 것을 레지스터-기반 가상 머신에서 실행될 수 있게끔 기능을 추가하여 연구를 시도해볼 수 있을 것이라 기대된다.

Abstract Code obfuscation is a technology that makes programs difficult to understand for the purpose of interpreting programs or preventing forgery or tampering. Inverse reading is a technology that analyzes the meaning of origin through reverse engineering technology by receiving obfuscated programs as input. This paper is an analysis of obfuscation and reverse-toxicization technologies for binary code in a virtualized-based environment. Based on VMAttack, a detailed analysis of static code analysis, dynamic code analysis, and optimization techniques were analyzed specifically for obfuscation and reverse-dipidization techniques before obfuscating and reverse-dipulation techniques. Through this thesis, we expect to be able to carry out various research on virtualization and obfuscation. In particular, it is expected that research from stack-based virtual machines can be attempted by adding capabilities to enable them to run on register-based virtual machines.

Key Words : Virtialization, Obfuscation, Deobfuscation, Static code Analysis, Dynamic Code Analysis

1. 서론

상용화 프로그램을 난독화(Obfuscation) 가상화된 프로그램을 역난독화(Deobfuscation) 역공학을 하여 크랙 파일을 생성하고, 라이센스를 무효화 시키는 일이 자주 발생하고 있다[1]. 또한, 온라인에서 유포되는 악성코드는 난독화- 가상화되어 스스로를 보호하는 강력한 코드로 발전되어 악용되고 있다. 최근에는 이를 방지하기 위해 IDA(Interactive DisAssenbler) 프로그램을 사용하여 분석하는 기술이 사용되고 있다. IDA 프로그램[3]과 VMAttack[3] 플러그인을 함께 사용하여 더욱 정확하고 다양한 분석을 시도하고 있다.

소프트웨어 기반 보호란 실행가능한 섹션에 대해서 암호화 및 압축하는 것이며 이 섹션을 반대로 해독 및 압축을 해제하면 새로운 코드가 생성된다. 이때, 진입 지점은 새로운 코드로 리다이렉

This Paper was supported by Joongbu University Research & Development Fund in 2018.
* Corresponding Author : School of Software Engineering, Joongbu University (sgkim@joongbu.ac.kr)
Received December 04, 2018 Revised December 13, 2018 Accepted December 15, 2018

트된다. 실행이 끝난 후, 진입 지점은 원래의 진입 지점으로 돌아가게 된다[4]. 예를 들면, Anti-Debug, Anti-VM[5,6]등이 있다.

본 논문에서는 가상화 기반으로 난독화된 바이너리 파일에 대해 정적 코드 분석, 동적 코드 분석, 최적화를 통해 역난독화 기술 방법에 대한 분석 내용과 실험 내용을 제시한다.

2. VMProtect

VMProtect는 2가지 의미로로 정의할 수 있다. VMProtect를 제작한 회사에서는 가상 머신에서 실행하므로 코드를 보호하며, 이 가상 머신은 일반적인 가상 머신과는 다른 구조를 가지고 있다. VMProtect는 내장(built-in) 디스어셈블러와 스크립트 언어를 가지고 있다. 내장 디스어셈블러를 통해 운영체제에 제한이 없어지며, 스크립트 언어를 통해서 플랫폼에 대해 제한이 없어진다. VMProtect를 분석한 자료인 Inside VMProtect에서는, VMProtect는 코드를 해독하지 않으며, 자연어 코드가 적정한 폴리몰픽(polymorphic) 바이트코드로 컴파일 된다고 한다. 어떠한 바이너리 파일에서 다른 바이너리 파일로 변경이 될 때, 사용되는 가상 머신은 똑같지 않는다. 또는 다른 가상 머신이 사용된다 하더라도 내부의 바이너리가 같을 수 있다고 한다. 그림1은 시스템구조를 보여준다.

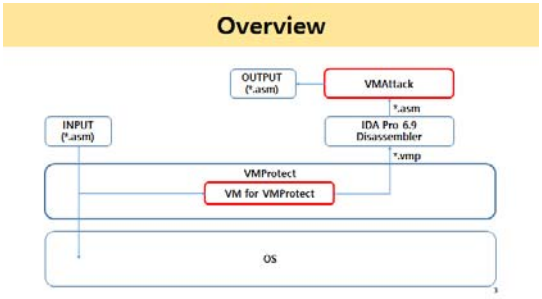

그림 1. 전체 시스템 구조
Fig.1 Overal System Structure

2.1 VMProtect 기능

VMProtect는 Memory Protection, Import Protection, Pack the Output File, Debugger Detection, Resource Protection, License Manager 등의 기능이 있다. 그림2의 Memory Protection은 Cyclic Redundancy Check 기능을 통해서 메모리의 이미지 파일을 보호하는 것이다. 원래의 진입 지점으로 흐름이 변경되기 전에 절대성이 검사되며, 만약 내부의 변화가 있을 경우 에러 알림창을 띄운다.

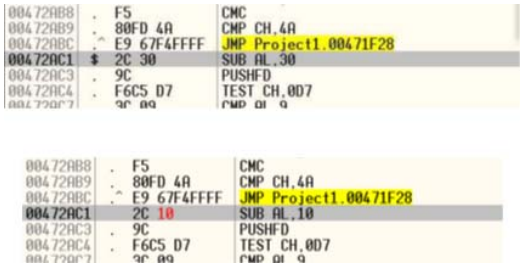

그림 2. 내부 코드 구조
Fig. 2. Internal Code Structure

그림3의 Import Protection은 Import Table(IAT, Import Address Table)로부터 모든 진입점들이 삭제된 것이다. API 호출을 위해서 코드 리다이렉션이 추가되며 DLL 파일의 내부 프로그램을 실행시키는 API 정보를 모두 숨긴다. Pack the Output File은 보호되는 목적 파일의 기능을 압축하여 축소시키는 것이다. 압축된 실행 프로그램은 런타임 동안에 실행되고 압축이 해제된다. Debugger Detection은 사용자 모드 디버거와 커널 모드 디버거 2가지 종류의 디버거를 찾는다. Resource Protection은 바이너리의 자원을 암호화하는 것이다. Licence Manager는 온라인에서 구매한 것을 확인하며, 시리얼 넘버를 관리하는 것이다.

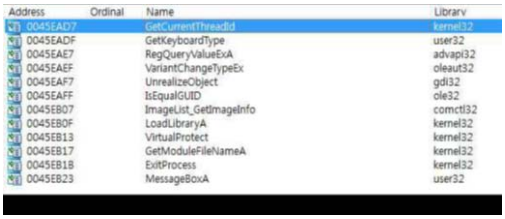

그림 3. API를 숨긴 IAT구조
Fig. 3. IAT Structure with Hidden API

2.2 VMProtect Architecture

VMProtect는 Reduced Instruction Set Computing(RISC)구조를 사용하며, 스택 기반 가상 머신을 사용한다. 스택 기반 가상 머신의 대표적인 예로는 Java Virtual Machine이 있다. 코드 보호 메커니즘으로는 가상화, 돌연변이, 복합 보호가 있다. 코드의 가상화된 조각은 결과 코드에 포함이 되어있다. 몇 개의 다른 가상 머신이 사용되어 분석에 복잡하다. VMProtect는 인터프리터, 가상 프로그램 카운터, 바이트 코드 핸들러 테이블, 바이트 코드 핸들러로 구성되어 있다. 가상 레지스터는 스택의 일부분이다. 다른 프로텍션 프로그램과 다른점으로는, 다른 부분을 다른 방법으로 보호할 수 있다. 다른 방법이란, 가상화, 난독화, 복합을 뜻한다. VMProtect의 바이트 코드와 IR은 유사하며, 이는 어셈블리와 다른 차이점을 가지고 있다. 차이점으로는, 스택 기반의 가상 머신이 있으며, IR은 임시 값을 가지고 있다. 또한, 스크래치 영역을 가지고 있다. 스크래치 영역이란, 가상 레지스터와 임시 값을 저장하며, 이는 레지스터보다 빠르다. 가상화 난독화 기능으로는, 폴리몰픽(polymorphic), 더미 삽입, 상수 값 난독화, 바이트코드를 디스어셈블과 IR로 변경하는 방법이 있다. 폴리몰픽은 다른 명령어로 표현하는 것이다. 더미 삽입은 중요한 명령어 사이에 더비 명령어를 삽입하는 것이다.

2.3 VMProtect Context

VM Context란, 호스트의 레지스터와 플래그 값을 저장하는 공간이다. 가상화 난독화되기 전에 저장이 되며, 가상화 난독화가 끝난 후 다시 호출된다. VM 레지스터를 초기화하는데 실제 머신에서 가상 머신으로 흐름이 넘어가게 되고, 스택에 실제 머신의 레지스터를 저장한다. 실제 머신의 레지스터를 가상 머신의 레지스터로 저장하므로 초기화가 된다. eax, ebx, ecx, edx, esi, edi, ebp, eflags 8가지의 레지스터를 사용하며, 시작 지점으로부터 4바이트씩 총 32바이트를 가상 레지스터가 할당한다. 레지스터의 순서는 난독화된 후 랜덤으로 바뀐다. 그림4의 핸들러 주소 테이블은 4바이트의 주소로 구성된다. 각각의 pcode로 핸들러의 주소가 저장되며, 핸들러의 개수는 256개이다.

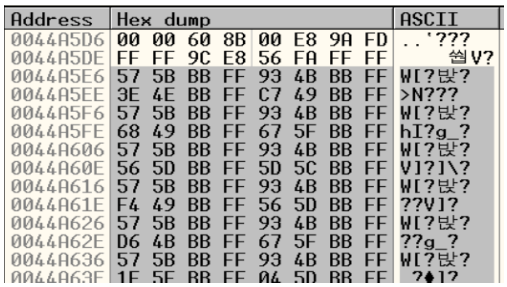

그림 4. 핸들러 주소 테이블
Fig. 4. Handler Address Table

2.4 VMProtect Loop

VM Loop는 가상 코드를 실행하기 위한 것이다. 명령어 포인터로 바이트 코드를 읽어 들이고, opcode 핸들러를 계산한다. 핸들러를 호출한다. 2개의 변수를 가지게 되는데, Down-Read VM-Byte, Up-Read VM-Byte를 가지게 된다. 레지스터 ESI와 RSI는 VM 명령어 포인터를 담당한다.

2.5 VMProtect Flow

VMProtect Flow는 실제 머신과 가상 머신 사이의 코드의 흐름을 의미한다. 텍스트 섹션(.text)은 난독화되기 전의 원래 코드 부분이며 실제 머신에서 수행이 된다. 가상화 섹션(.vmp)

은 VMProtect를 통해 난독화된 후의 결과 코드 섹션이다. 가상화된 코드는 .vmp 섹션에서 수행된다. 텍스트 섹션과 가상화 섹션의 흐름을 통해 VM 진입 부분을 알 수 있다.

3. VMAttack

VMAttack이란 IDA 프로그램의 플러그인이며, 역난독화를 위해 정적 동적 분석을 지원한다. VMAttack의 구조는 그림 5와 같다. 라이브러리에 기반한 정적, 동적, 최적화 모듈을 가지고 있으며, 이들을 자동화하는 시스템을 가지고 있다. 정적 모듈은 가상 변환 및 디스 어셈블리 모듈로 구성되있으며 VMAttack의 정적접근을 가능하게 하고 동적 모듈은 동적 슬라이싱 및 클러스터링으로 세분화 될 수 있다. 최적화 모듈은 정적 분석 IR 의사 코드 명령어에 대한 실행 트레이스 필터링 기능 및 개선 사항을 제공한다. 이러한 모듈 위에 자동화 시스템을 사용하면 사용 가능한 모든 모듈을 자동으로 조합 할 수 있다.

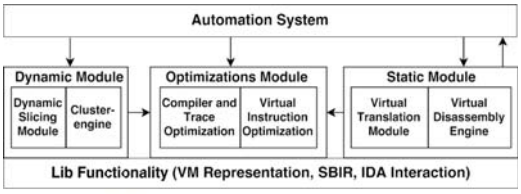

Figure:Software Architecture of VMAttak.

그림 5. VMAttack의 구조
Fig. 5.VM Attack Structure

3.1 동적분석

압축 된 바이너리의 실행 흔적을 기록하고 VM 인터프리터 논리에만 속한 명령어를 필터링하여 최적화는 것이 동적 분석이다. 동적 분석은 동적 슬라이싱 및 클러스터링 중심의 최적화를 통해 원래 기능을 복구하는 데 중점을 둔다.

동적분석은 바이너리를 실행 시 트레이스의 정보를 얻고 스택의 레지스터 값을 포함하여 분석 시간을 단축할 수 있다. 실행 중에는 명령 트레이스를 생성하게 되고 명령 트레이스는 4개의 필수요소와 2개의 옵션요소로 구성되어 있다. 최적화에는 Propagation과 Folding 방법으로 구성이 된다. Propagation 최적화에는 레지스터가 값으로 전환되고 오프셋을 계산하는 Constant Propagation과 스택 주소를 읽을 때마다 스택 주석으로 사용해주는 Stack Address Propagation이 있다. Folding 최적화에는 불필요한 Trace를 교체해주는 peephole optimization과 나중에 실행단계에서 사용되지 않는 피연산자를 Trace에서 제거하는 Unused Operand Folding과 특정 작업을 표준화하는 것으로 peephole Optimization과 비슷한 operation Standardization이 있다.

클러스터링 하면서 불필요한 명령어를 제거하고 고유명령어를 파악 할 수 있게 해주며 명령 트레이스를 고유명령어와 반복명령어로 분류해주고 주소가 두 번 이상 발생하면 기본값을 클러스터로 선언한다. 분석이 성공적으로 끝나게 되면 리버스 엔지니어링에게 고유명령어 및 클러스터가 있는 명령어 트레이스를 제공한다. 가상머신이 입력 순서를 해당 스택 주소로 이동하기 위해 동일한 순서의 명령을 사용하는 경우, 반복되기 때문에 클러스터로 선언이 된다. 결과로 바이너리의 동일한 주소를 실행하는 두 개의 클러스터를 보여준다.

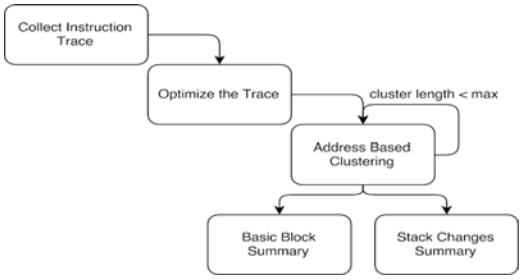

그림 6. 클러스터링 알고리즘
Fig. 6. Clustering Algorithm

그림 6은 클러스터링의 알고리즘을 보여준다. 먼저 동적분석을 위해 바이너리를 실행하여 생성

된 명령 트레이스를 수집하여 최적화를 시켜준다. 주소 기반 클러스터링에는 명령어 트레이스에서 연속적인 클러스터링 라운드로 구성이 되고 각 클러스터링 라운드에서 모든 주소와 그 이웃에 대해 동일한 주소 순서가 트레이스에서 검색된다. 이와 동일한 순서가 존재하면 클러스터가 생성된다. 클러스터링 라운드가 모든 클러스터에 길이 증가가 없는 경우 클러스터링 루프는 중지가 된다. 그 후 기본 블록 요약과 스택 변경요약으로 나뉘는데 기본 블록은 클러스터 내부의 기본 블록위치, 시작 및 종료 주소, 명령어, 스택변경요약으로 구성 되어있으며 스택 변경요약에는 기본블록에서 발생한 모든 스택 변경 내용이 포함되어 있다. 그림7에서 보여주는 동적 슬라이싱(dynamic slicing) 알고리즘은 직간접적으로 특정변수의 값을 계산하는 모든 명령문을 찾을 수 있는 기법을 말한다. 동적 모듈의 일부로서 명령어 트레이스를 예상하고 Propagation Optimization을 적용한다. 동적 슬라이싱에는 두 가지 접근방법이 있다. 값의 원점이 계산인 경우와 주소인 경우로 접근할 수 있는데, 계산인 경우는 계산과 관련된 모든 트레이스 행이 계산에 사용된 값의 원점까지 추적이 되고 주소인 경우 이전 메모리 주소를 찾을 수 없을 때까지 추적이 된다. 먼저 동적 분석을 위해 바이너리를 실행하여 생성된 명령 트레이스를 수집하여 최적화를 시킨다. 트레이스에서 가상 머신 입력 및 출력 매개 변수가 식별되어 추출된다. 각 출력 레지스터에 대해 분석은 결과 값의 출처를 추적하여 동적 슬라이스를 만들고 결과 값이 트레이스에서 마지막 스택 주소까지 이어지기 때문에 주소에 저장된 값이 무엇인지 알 수 있다.

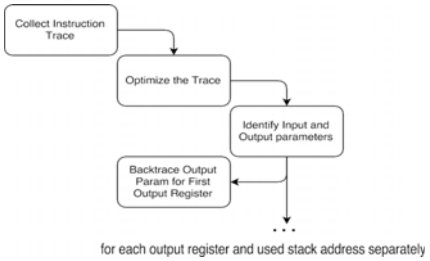

그림 7. 동적 분할 알고리즘
Fig. 7. Dynamic Slicing Algorithm

3.2 정적분석

가상화된 바이트 코드를 중간 표현의 의사 코드 명령어로 변환하는 것이 정적분석이다. 스택 기반 IR에서 가상머신의 자동화된 분석을 통한 바이트 코드의 역난독화에 중점을 둔다.

SVMC이란, 정적분석을 위해서는 SVMC의 4개 입력값(바이트코드 시작 및 끝, 점프 테이블 기본주소, 가상 머신 기능의 시작 주소)가 필요하다. SBIR이란, 스택 기반 VM의 사용 변수, 반환 매개 변수 및 제어 흐름 종속성을보다 명확하게 파악할 수 있습니다.

정적분석을 위해서는 가상머신 Context의 4개 입력값이 필요하여 SVMC에서 찾아주고 인터프리터의 점프 테이블은 바이트 코드를 해당 어셈블리 명령어에 매핑하는데 사용하여 오버헤드를 포함하고 있지만 어셈블리 명령으로 나타낼 수 있는 바이트 코드를 생성해주고 SBIR로 변환하여 어셈블리 명령의 오버헤드를 제거한다. 마지막으로 가상 명령 최적화를 사용하여 최상의 결과를 얻어낸다.

3.3 자동화

VMAttack은 자동화 모델을 제공하며 가상머신 명령어 트레이스를 제거하고 기능 명령을 보존한다. 이 알고리즘은 8단계로 구성되어 있으며 모든 분석 기능이 순서대로 적용되어 단계 마다 가중치를 가지고 있습니다. 분석단계의 가중치는 점수에 영향을 미치는 정도를 결정한다. Initial Scoring에서 시작하며 두 가지 경로가 있다. 기

본적인 방법의 경로는 Compute Register Mappings이며 정적 및 동적분석의 경로는 Propagation Optimizations하는 경로가 있다. 별표 부분은 각 분석의 가중치에 대한 점수를 부여한다. VMAttack관련 논문에서는 기본적인 방법을 통해 평균 89.86%의 실행 트레이스를 줄일 수 있었으며 동적 및 정적 분석 방법에 대해 평균 96.67%의 실행 트레이스 감소한다.

3.4 동적 분석 코드

static_deobfuscate.py는 라이브러리 호출 부분 포함 총 4개의 부분으로 나누어진다. 그림 8에서는 베이직 블록의 색을 지정해주는 리스트가 정의된다. 그림 9에서 get_instruction_list를 통해서 주어지는 가상 명령어를 위한 x86 명령어의 리스트를 생성한다. clear_comment는 모든 주석을 부분적으로 삭제한다. first_deobfuscate는 가상화 코드의 시작 주소와 끝 주소 사이를 VmInstruction으로 변환하여 준다. 즉, 일부만 지정한 부분에서 변환이 이루어진다. deobfuscate는 전체에서 바꿀 수 있는 모든 가상화 코드를 VmInstruction으로 변환하도록 한다. display_ps_inst는 의사 명령어를 보여주며, display_vm_inst는 가상 명령어를 보여준다.

```
from idaapi import *
from idautils import *
from idc import *
from lib.Instruction import Instruction
from lib.Optimize import *
from lib.Register import (get_reg_by_size,
                          get_reg_class)
from lib.VmInstruction import VmInstruction
from lib.VmInstruction import (add_ret_pop,
                               to_vpush)
import distorm3
import lib.PseudoInstruction as PI
import lib.StartVal as SV
from ui.BBGraphViewer import show_graph
from lib.VMRepresentation import VMContext, get_vmr

bb_colors = [0xdddddff, 0xffdddd, 0xddffdd, 0xffddff, 0xffffdd, 0xddffff]
```

그림 8. static_deobfuscate.py의 소스 코드(1)
Fig. 8. static_deobfusccater.py Source Code(1)

```
def calc_code_addr(instr, base):
def get_instruction_list(vc, base):
def clear_comments(ea, endaddr):
def get_start_push(vm_addr):
def get_catch_reg(reg, length):
def first_deobfuscate(ea, base, endaddr):
def deobfuscate_all(base):
def display_ps_inst(ps_inst_lst):
def display_vm_inst(vm_inst_lst):
def read_in_comments(start, end):
def find_start(start, end):
def get_jaddr_from_comments(pp_lst, comment_lst):
def get_jmp_input_found(cjmp_addrs, jmp_addrs):
def change_comments(pp_lst, cjmp_addrs):
def get_jmp_addr(bb):
def has_ret(bb):
def get_jmp_loc(jmp_addr, jmp_addrs):
```

그림 9. static_deobfuscate.py의 소스 코드(2)
Fig. 9. static_deobfusccater.py Source Code(2)

그림 10의 deobfuscate는 코드 시작 주소와 끝 주소 사이에서 역난독화를 수행한다. 여기서 반환값은 최저 분기 주소이다. 앞의 그림 9에서 정의된 함수들이 여기에서 많이 사용이 된다. 그림 11의 start 함수는 역난독화의 진입 지점이며, 최소의 시작 주소 코드가 발견되기 전까지 계속해서 역난독화를 수행한다. 반환 값은 함수의 진입 지점의 실제 시작 주소가 된다. 그림 12에서는 베이직 블록에 대한 함수들이 수행된다. 색상을 처리하는 color_basic_blocks 함수는 베이직 블록을 생성하고 리스트 형태로 반환해주는 make_bb_lists, 베이직 블록의 시작과 끝을 출력해주는 print_bb가 정의된다. get_distorm_info라는 함수는 distorm3의 정보를 출력한다. 여기서 나오는 정보는 instruction bytes, opcode, opcode type, flag, raw flag, instruction class, flow control, address, size, dt, valid, segment, unused prefixes mask, mnemonic이 있다. 마지막 static_deobfuscate 함수는 부분 선택적 역난독 자동화를 할 것인지, 전체 역난독 자동화를 할 것인지를 선택하여 수행한다.

```
def deobfuscate(code_saddr,  base_addr, code_eaddr, vm_addr, display=4, real_start=0):
    """
    @brief This function does the deobfuscation between code_saddr and code_eaddr
    @param code_saddr Address of the start of obfuscated code
    @param base_addr Address of the jumptable of the virtual machine
    @param code_eaddr Address of the end of obfuscated code
    @param vm_addr Address of the virtual machine function
    @param display Set output type
        * 0 : VirtualInstruction
        * 1 : PseudoInstruction Push/Pop representation
        * 2 : PseudoInstruction full optimized
    @param real_start Address of entry point of the function
    @return The lowest found jump address
    """
```

그림 10. static_deobfuscate.py의 소스 코드(3)
Fig. 10. static_deobfusccater.py Source Code(3)

```
def start(code_saddr, base_addr, code_eaddr, vm_addr, display=4, real_start=0):
    """
    @brief Entrypoint of the deobfuscation; starts
    deobfuscate until the minimal code start address is found
    @param code_saddr Address of the start of obfuscated code
    @param base_addr Address of the jumptable of the virtual machine
    @param code_eaddr Address of the end of obfuscated code
    @param vm_addr Address of the virtual machine function
    @param display Set output type
        * 0 : VirtualInstruction
        * 1 : PseudoInstruction Push/Pop representation
        * 2 : PseudoInstruction full optimized
    @param real_start Address of entry point of the function
    """
```

그림 11. static_deobfuscate.py의 소스 코드(4)
Fig. 11. static_deobfusccater.py Source Code(4)

```
def color_basic_blocks(basic_lst):
def make_bb_lists(pp_lst, basic_lst):
def has_locals(bb_lsts):
def print_bb(bb_lsts):
def get_distorm_info(inst_addr):
def jmp_to_orig(address, base):
def static_vmctx(manual=False):
def static_deobfuscate(display=0, userchoice=False):
```

그림 12. static_deobfuscate.py의 소스 코드(5)
Fig. 12. static_deobfusccater.py Source Code(5)

3.5 정적 분석 코드

dynamic_deobfuscate.py는 라이브러리 호출 부분을 포함하여 4개의 부분으로 구성되어 있다. 그림 13의 라이브러리 부분은 UI 라이브러리, 디버거 라이브러리, 트레이스 분석 라이브러리, 가상 머신레지스터 호출 라이브러리를 호출하여 사용한다.

```
from threading import Thread
from ui.UIManager import GradingViewer
from ui.UIManager import OptimizationViewer
from ui.UIManager import StackChangeViewer
from ui.UIManager import VMInputOuputViewer
from DebuggerHandler import load, save, get_dh
from lib.TraceAnalysis import *
from lib.VMRepresentation import get_vmr
from ui.NotifyProgress import NotifyProgress
from ui.UIManager import ClusterViewer
```

그림 13. dynamic_deobfuscate.py의 소스 코드(1)
Fig. 13. dynamic_deobfuscate.py의 Source Code(1)

```
### DEBUGGER LOADING STRATEGIES ###
# IDA Debugger
def load_idadbg(self):
# OllyDbg
def load_olly(self):
# Bochs Dbg
def load_bochsdbg(self):
# Win32 Dbg
def load_win32dbg(self):
# Immunity Dbg
def load_immunitydbg(self):
# Working with Win32Dbg, BochsDbg, OllyDbg
available_debuggers = [load_idadbg, load_olly, load_bochsdbg, load_win32dbg, load_immunitydbg]
```

그림 14. dynamic_deobfuscate.py의 소스 코드
Fig. 14. dynamic_deobfuscate.py의 Source Code(2)

```
### INIT AND LOAD CONTEXT ###
def prepare_trace():
def prepare_vm_ctx():
def prepare_vm_operands():
def load_dbg(choice):
def load_trace():
def save_trace():
```

그림 15. dynamic_deobfuscate.py의 소스 코드(2)
Fig. 15. dynamic_deobfuscate.py의 Source Code(2)

그림 14은 디버거에 대한 함수를 정의하며, 어떤 디버거가 사용 가능한지도 나타내고 있다. 디버거는 IDA Debugger, OllyDbg, Bochs Dbg, Win32 Dbg, Immunity Dbg를 정의하고 있다. 그림 15는 가상 머신의 Context 부분을 초기화 및 호출을 정의하고 있다. 앞서 설명한 흐름 부분의 가에 해당이 되는 소스 코드이다.

```
def gen_instruction_trace(choice):
    """
    Generate instruction trace
    :param choice: which debugger to use
    """
class DynamicAnalyzer(Thread):
    def __init__(self, func, trace, **kwargs):
    def run(self):
    def get_result(self):

def address_heuristic():
    """
    Compute the occurrence of every address in the instruction trace.
    """
manual_func = [find_output, find_input, find_virtual_regs, follow_virt_reg]
def manual_analysis(choice):
def input_output_analysis(manual=False):
def clustering_analysis(visualization=0, grade=False, trace=None):
def optimization_analysis():
def dynamic_vmctx(manual=False):
def init_grading(trace):
def grading_automaton(visualization=0):
```

그림 16. dynamic_deobfuscate.py의 소스 코드(3)
Fig. 16. dynamic_deobfuscate.py의 Source Code(3)

그림 16에서의 gen_instruction_trace를 통하여 트레이스 명령어를 생성한다. 이때, 특정 디버거를 생성한다. DynamicAnalyzer 클래스는 실행

을 하여 결과 값을 가져온다. address_heuristic 함수는 트레이스 명령어에서 만들어지는 모든 주소를 계산한다. 마지막 manual_analysis부터 grading_automation 부분은 DynamicAnalyzer 클래스에 포함되며, 동적 분석의 기능들과 같다.

4. 결론

본 논문에서는 가상화, 난독화 프로그램에 대해서 분석해보고, 프로그램으로부터 생성된 결과물에 대한 분석하는 툴인 VMAttack을 분석했다. VMProtect는 가상화, 난독화 패킹을 하는 도구이며, 상용 프로그램이다. 상용 프로그램을 내부적으로 분석하지 못한 제한적인 요인으로 인해, 이에 관한 자료들을 찾아 참고하였다. VMAttack은 설치 과정과 기능 실행 부분에서 아직까지는 미비한 부분이 있는 것으로 보인다. 하지만, 스택-기반 가상 머신을 분석하기에는 충분히 매력적인 도구로 보인다.

본 논문를 통하여 다양한 가상화, 난독화에 대한 연구를 진행할 수 있을 것으로 기대된다. 특히, 스택-기반 가상 머신에서 연구한 것을 레지스터-기반 가상 머신에서 실행될 수 있게끔 기능을 추가하여 연구를 시도해볼 수 있을 것이라 기대된다.

REFERENCES

[1] Johannes Kinder, "Towards static analysis of virtualization obfuscated binaries," In 2012 19th Working Conference on Reverse Engineering. IEEE, pp.61-70, 2012

[2] Jasvir Nagra and Christian Collberg, " Surreptitious Software: Obfuscation, Watermarking, and Tamper proofing for Software Protection," Pearson Education, 2009.

[3] Rolf Rolles, "Unpacking Virtualization Obfuscators," In Proceedings of the 3rd USENIX Conference on Offensive Technologies(WOOT'09), USENIX, 2009.

[4] Monirul I. Sharif, Andrea Lanzi, Jonathon T. Giffin, and Wenke Lee, "Automatic Reverse Engineering of Malware Emulators," In 30th IEEE Symposium on Security and Privacy (S&P 2009), May 2009

[5] Hiralal Agrawal and Joseph R Horgan, "Dynamic program slicing," In ACM SIGPlan Notices, Vol. 25. ACM, pp.246-256, 2009.

[6] Bertrand Anckaert, Mariusz Jakubowski, and Ramarathnam Venkatesan, "Proteus: virtualization for diversified tamperresistance," In Proceedings of the ACM workshop on Digital rights management. ACM, pp.47-58, 2007.

[7] Choi Do-Hyeon, Jung Oh Park, ' A Study on Security Authentication Vector Generation of Virtualized Internal Environment using Machine Learning Algorithm', The Journal of The Institute of Internet, Broadcasting and Communication VOL. 16 No. 6, 2016

저자약력

김 순 곤(Soon-Gohn Kim) **[종신회원]**

- 전북대학교 일반대학원 전자계산기공학과 졸업(공학박사)
- 동아생명보험(주) 전자계산실 (DBA)
- 한국원자력연구소 핵전산연구부 (선임연구원)
- 중부대학교 소프트웨어공학부 (교수)

〈관심분야〉 데이터베이스시스템, 정보보호암호화응용프로토콜, 정보시스템감리, 데이터마이닝, 유비쿼터스컴퓨팅, 서버장애예측, 소프트웨어시스템분석설계