바이너리 분석 시작하기

제 29회 해킹캠프

2024.08.17

POC SECURITY

발표자 소개

- Demon팀 소속
- Best of the Best 7기 수료
- SW Maestro 10기 수료
- 컴퓨터과학 전공 / 정보보호학 석사

Agenda

- 바이너리란 무엇인가
- 바이너리 분석의 필요성과 어려움
- 초도 분석: 바이너리 정보 수집
- 정적 분석과 동적 분석
- Instrumentation, Taint analysis, Symbolic execution
- 바이너리 분석 방해하기
- 결론
- 앞으로 공부해야 할 것들

바이너리란 무엇인가

- 바이너리(binary) : 0과 1 이진수
- 컴퓨터는 모든 것을 0과 1 이진수로 표현한다
- 파일, 명령어, 데이터 등 모든 게 이진수다
- 프로그램: 실행 가능한 바이너리 (executable binary) ← 우리의 목표

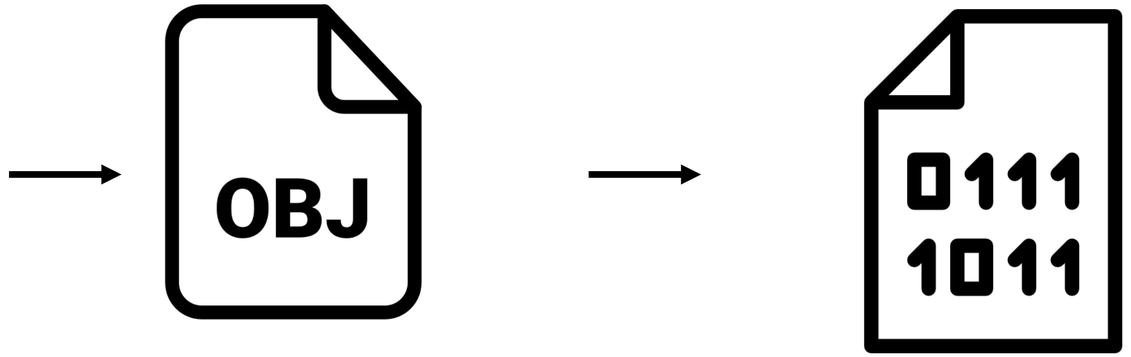
바이너리란 무엇인가

```c
int main() {

   printf("%s", "Hello Hackingcamp 29 !\n");

   return 0;

}
```



main.c

main.o

a.out

Compiler
Assembler

Linker

POC SECURITY

바이너리란 무엇인가

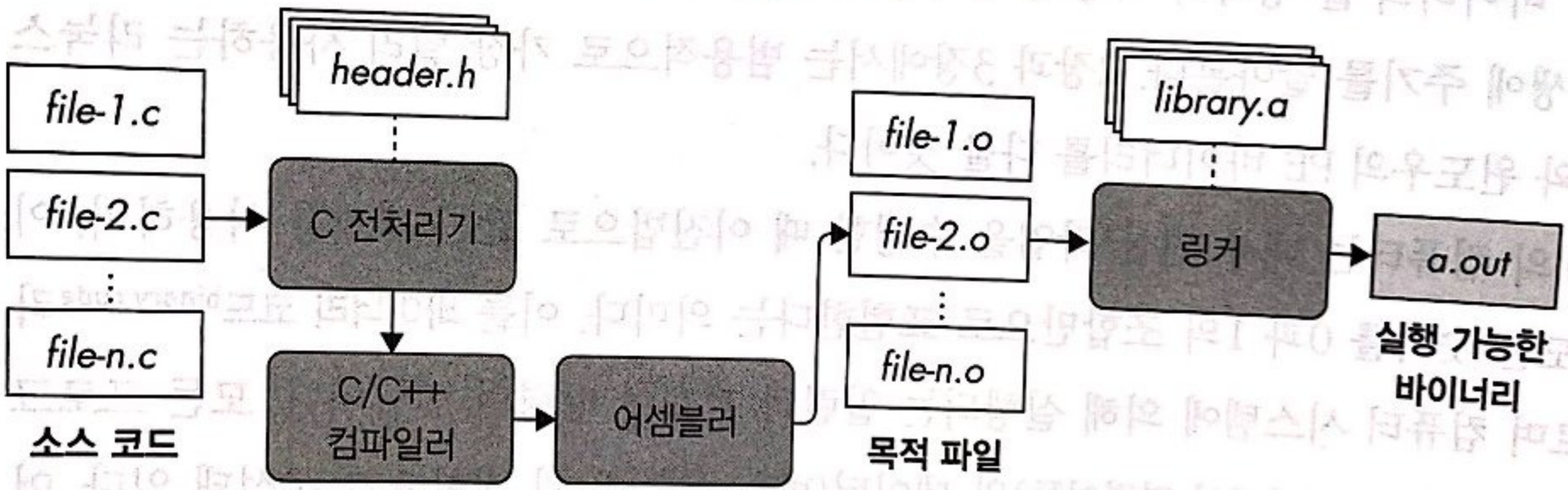
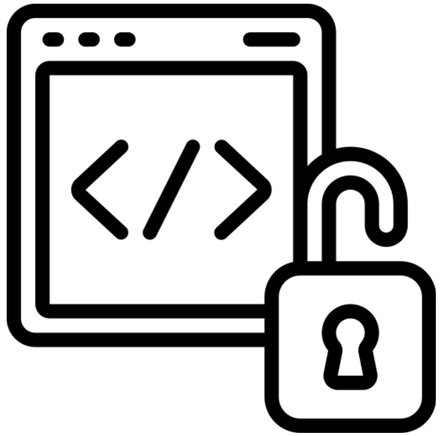


그림 1-1 C 언어로 작성된 소스 코드의 컴파일 과정

바이너리 분석의 필요성과 어려움

바이너리 분석의 필요성



소스가 없기 때문에 😰

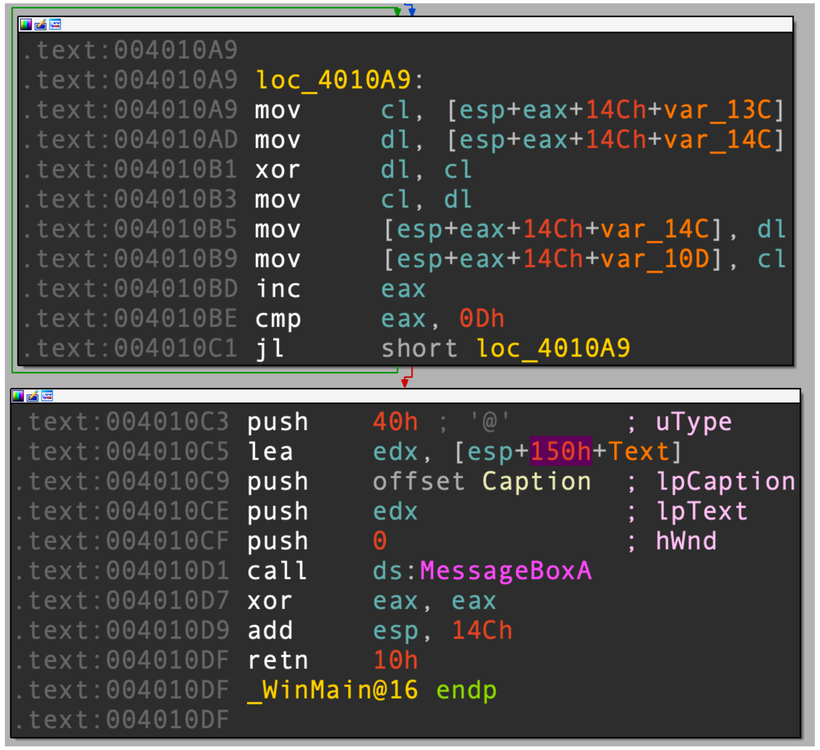
- 악성코드 동작 방식 분석 파악
- 악성코드 모사 구현 (나쁜 짓)

- 취약점 분석
- 익스플로잇 작성 (나쁜 짓)
- 등등 각자의 목적에 따라

바이너리 분석의 필요성과 어려움

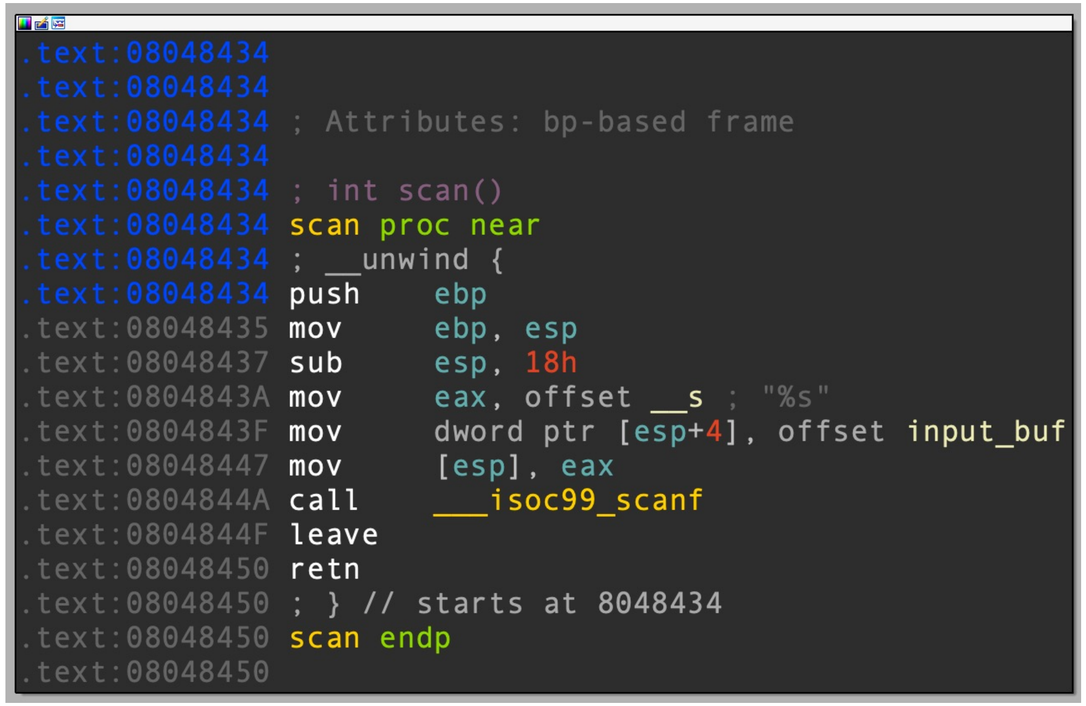
바이너리 분석의 어려움

- 무슨 말인지도 모르겠는 걸 하루 종일 보고 있어야 한다



```
.text:004010A9
.text:004010A9 loc_4010A9:
.text:004010A9 mov     cl, [esp+eax+14Ch+var_13C]
.text:004010AD mov     dl, [esp+eax+14Ch+var_14C]
.text:004010B1 xor     dl, cl
.text:004010B3 mov     cl, dl
.text:004010B5 mov     [esp+eax+14Ch+var_14C], dl
.text:004010B9 mov     [esp+eax+14Ch+var_10D], cl
.text:004010BD inc     eax
.text:004010BE cmp     eax, 0Dh
.text:004010C1 jl      short loc_4010A9
```

```
.text:004010C3 push    40h ; '@'       ; uType
.text:004010C5 lea     edx, [esp+150h+Text]
.text:004010C9 push    offset Caption  ; lpCaption
.text:004010CE push    edx             ; lpText
.text:004010CF push    0               ; hWnd
.text:004010D1 call    ds:MessageBoxA
.text:004010D7 xor     eax, eax
.text:004010D9 add     esp, 14Ch
.text:004010DF retn    10h
.text:004010DF _WinMain@16 endp
.text:004010DF
```



```
.text:08048434
.text:08048434
.text:08048434 ; Attributes: bp-based frame
.text:08048434
.text:08048434 ; int scan()
.text:08048434 scan proc near
.text:08048434 ; __unwind {
.text:08048434 push    ebp
.text:08048435 mov     ebp, esp
.text:08048437 sub     esp, 18h
.text:0804843A mov     eax, offset __s ; "%s"
.text:0804843F mov     dword ptr [esp+4], offset input_buf
.text:08048447 mov     [esp], eax
.text:0804844A call    ___isoc99_scanf
.text:0804844F leave
.text:08048450 retn
.text:08048450 ; } // starts at 8048434
.text:08048450 scan endp
.text:08048450
```


POC SECURITY

바이너리 분석의 필요성과 어려움

바이너리 분석의 어려움

- 무슨 말인지도 모르겠는 걸 하루 종일 보고 있어야 한다
- 각종 방해 기법 존재
- 모르는 아키텍처에 대해 새로운 학습 필요 (PPC, ARM, RISC-V, MIPS 등)
- 어디를 핵심으로 잡고 깊게 분석할지 결정 ← 이걸 모르면 하루 종일 삽질
- 등등 그냥 어렵다 !!

초도 분석: 바이너리 정보 수집

- 바이너리를 받았다.. 뭐부터 해야 하나..?

```
alkyne@dba82f865a13:~/hackingcamp29$ ls -al
total 24
drwxrwxr-x 2 alkyne alkyne  4096 Jul 31 09:33 .
drwxr-xr-x 3 alkyne alkyne  4096 Jul 31 09:32 ..
-rw-r--r-- 1 alkyne alkyne 13485 Jul 31 09:33 binary
```

- 기본적인 정보 수집을 통해 향후 분석 방향 수립

초도 분석: 바이너리 정보 수집

- 바이너리의 종류, 특성에 따라 앞으로 분석 방향 결정
- 기본 도구만 사용해도 충분한 정보 수집할 수 있다
- file: 파일 유형, 각종 정보 수집 가능

```
alkyne@dba82f865a13:~/hackingcamp29$ file binary
binary: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.24, BuildID[sha1]=aa0
6e585338b14500d42e148bf5b1346c010cc5, not stripped
```

- strings: 파일 내 (모든) 문자열 식별 가능 (심볼, 데이터 등)

```
alkyne@dba82f865a13:~/hackingcamp29$ strings binary
/lib64/ld-linux-x86-64.so.2
nXS8
__gmon_start__
libc.so.5
exit
sprintf
```

POC SECURITY

초도 분석: 바이너리 정보 수집

- ldd: 참조하는 공유 라이브러리 식별 가능 (바이너리를 실행하기 때문에 주의)
- xxd: 파일의 내용을 바이트와 아스키로 동시에 확인 가능

```
alkyne@dba82f865a13:~/hackingcamp29$ xxd binary  | more
00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF............
00000010: 0200 3e00 0100 0000 d008 4000 0000 0000  ..>.......@.....
00000020: 4000 0000 0000 0000 b821 0000 0000 0000  @........!......
00000030: 0000 0000 4000 3800 0900 4000 1e00 1b00  ....@.8...@.....
00000040: 0600 0000 0500 0000 4000 0000 0000 0000  ........@.......
00000050: 4000 4000 0000 0000 4000 4000 0000 0000  @.@.....@.@.....
00000060: f801 0000 0000 0000 f801 0000 0000 0000  ................
00000070: 0800 0000 0000 0000 0300 0000 0400 0000  ................
00000080: 3802 0000 0000 0000 3802 4000 0000 0000  8.......8.@.....
00000090: 3802 4000 0000 0000 1c00 0000 0000 0000  8.@.............
000000a0: 1c00 0000 0000 0000 0100 0000 0000 0000  ................
000000b0: 0100 0000 0500 0000 0000 0000 0000 0000
```

초도 분석: 바이너리 정보 수집

- readelf: 리눅스 바이너리 ELF 분석 (헤더 파싱을 다 해준다 !)

```
alkyne@dba82f865a13:~/hackingcamp29$ readelf -h ./binary
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x4008d0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          8632 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
  Size of section headers:           64 (bytes)
  Number of section headers:         30
  Section header string table index: 27
```

POC SECURITY

초도 분석: 바이너리 정보 수집

- nm: 심볼 목록 출력 (static symbol table 추출)
- nm -D -demangle 명령어 사용하면 C++ 함수 내용 복원 가능

```
_ZNSsD1Ev
_ZNSt8ios_base4InitD1Ev
_ZNSsaSEPKc
_ZSt3cin
_ZNSirsERi
_ZNSirsERj
_ZTVN10__cxxabiv117__class_type_infoE
__gxx_personality_v0
_ZSt4cout
_ZNSolsEi
_Znwm
_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc
_ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_
_ZNSsC1Ev
_ZStrsIcSt11char_traitsIcEERSt13basic_istreamIT_T0_ES6_PS3_
_ZNSolsEPFRSoS_E
_ZStlsIcSt11char_traitsIcESaIcEERSt13basic_ostreamIT_T0_ES7_RKSbIS4_S5_T1_E
_ZTVN10__cxxabiv120__si_class_type_infoE
_ZNSt8ios_base4InitC1Ev
_ZdlPv
```

```
 U std::istream::operator>>(int&)
 U std::istream::operator>>(unsigned int&)
 U std::ostream::operator<<(std::ostream& (*)(std::ostream&))
 U std::ostream::operator<<(int)
 U std::basic_string<char, std::char_traits<char>, std::allocator<char> >::basic_string()
 U std::basic_string<char, std::char_traits<char>, std::allocator<char> >::~basic_string()
 U std::string::operator=(char const*)
 U std::ios_base::Init::Init()
 U std::ios_base::Init::~Init()
 U std::cin
 U std::cout
 U std::basic_ostream<char, std::char_traits<char> >& std::endl<char, std::char_traits<char>
d::char_traits<char> >&)
 U std::basic_ostream<char, std::char_traits<char> >& std::operator<< <std::char_traits<char
cd::char_traits<char> >&, char const*)
 U std::basic_ostream<char, std::char_traits<char> >& std::operator<< <char, std::char_trait
>(std::basic_ostream<char, std::char_traits<char> >&, std::basic_string<char, std::char_trait
> const&)
 U std::basic_istream<char, std::char_traits<char> >& std::operator>><char, std::char_traits
ar, std::char_traits<char> >&, char*)
```

초도 분석: 바이너리 정보 수집

- strace, ltrace: 시스템콜 및 라이브러리 호출 파악 가능
- objdump: 각 섹션별 데이터 추출 가능 (.rodata, .bss, .text 등)
  - 프로그램 헤더, 각종 섹션 정보, 심볼 테이블 등 확인 가능 !

```
Program Header:
    PHDR off    0x0000000000000040 vaddr 0x0000000000400040 paddr 0x0000000000400040 align 2**3
         filesz 0x00000000000001f8 memsz 0x00000000000001f8 flags r-x
  INTERP off    0x0000000000000238 vaddr 0x0000000000400238 paddr 0x0000000000400238 align 2**0
         filesz 0x000000000000001c memsz 0x000000000000001c flags r--
    LOAD off    0x0000000000000000 vaddr 0x0000000000400000 paddr 0x0000000000400000 align 2**21
         filesz 0x00000000000011c4 memsz 0x00000000000011c4 flags r-x
    LOAD off    0x0000000000001e28 vaddr 0x0000000000601e28 paddr 0x0000000000601e28 align 2**21
```

```
Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .interp       0000001c  0000000000400238  0000000000400238  00000238  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .note.ABI-tag 00000020  0000000000400254  0000000000400254  00000254  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
```

정적 분석과 동적 분석

- 정적 분석: 소스코드 실행하지 않으며 분석
  - 전체 코드 구조 파악
  - 실행 되지 않는 코드도 파악 가능
  - 디컴파일러 등을 활용한 바이너리 오디팅
  - 앞서 살핀 초도 분석 과정이 대부분 정적분석에 포함

정적 분석과 동적 분석

- 동적 분석: 바이너리를 실행하며 동작 행위 관찰
  - 시스템 콜, 메모리 릭 등 측정 가능
  - 어떤 파일을 생성하는지, 암호화 하는지 추적 가능 (특히 랜섬웨어)
  - 실제 호출 함수들 실행 흐름 파악 가능 (전달 인자, 실행 순서, 변수 값 등)
  - 디버깅을 통해 실제 변수 값, 힙 레이아웃 등 메모리 구조 파악 가능
  - 아무 바이너리나 실행하면 위험하므로 가상 환경에서 주로 분석

정적 분석과 동적 분석

- 서로 장단점이 있다
- 가능하다면 둘 다 함께 사용하는 것이 좋다
- 중요한 점은 [정적 + 동적] 분석으로 내가 필요한 부분 빠르게 파악 필요
- 해당 부분 집중 공략

고급 기법

- Taint analysis
- 특정 부분이 다른 부분에 어떻게 영향을 미칠까?
- 나의 인풋이 어떤 레지스터, 어떤 변수들을 오염시키는지 분석

```
0002356 mov     [rbp-80h], rbx
000235A mov     rax, [rbp-88h]
0002361 mov     rax, [rax]
0002364 add     rax, 8
0002368 mov     rdx, [rax]
000236B mov     rax, [rbp-88h]
0002372 mov     rdi, rax
0002375 call    rdx
```

- rdx를 조작할 수 있을까? (나쁜 생각...)
- rdx는 어디에서 왔을까?
- rax는 어디서 가져올까?
- 스택에 원하는 값을 쓸 수 있을까?

고급 기법

- Symbolic execution
- 특정 프로그램 상태나 영역에 도달할 수 있는 방법 모색
- 제약 조건(constraint)을 풀어내주는 도구를 솔버라고 한다
- 명령어 도달 가능성 증명
- 바이너리를 분석하다 취약점을 찾았다 → 어떻게 트리거 할 수 있을까?

고급 기법

- Z3 솔버 소개

```python
x = input()
y = input()

z = x + y
if (x >= 5):
    foo()
    y = y + z
    if (y < z):
        vuln() # we want to call this
    else:
        bar()
else:
    qoo()
```

```
z3@hackingcamp29:~$ z3 -in
(declare-const x Int)
(declare-const y Int)
(declare-const z Int)
(declare-const y2 Int)
(assert (= z (+ x y)))
(assert (>= x 5))
(assert (= y2 (+ y z)))
(assert (< y2 x))
(check-sat)
sat
(get-model)
(model
  (define-fun y () Int
    (- 1))
  (define-fun x () Int
    5)
  (define-fun y2 () Int
    3)
  (define-fun z () Int
    4)
)
```

- 특정 조건을 만족하는 수식을 풀어낼 수 있다
- 수식 단순화가 중요

바이너리 분석 방해하기

- 분석의 허들을 높이기 위해 사용
- 소프트웨어 구조 파악 방해 위해 사용
- 다양한 기법이 있다
  - 난독화 (Obfuscation)
  - 안티 디버깅
  - 패킹
  - 코드 가상화
  - 등등..

바이너리 분석 방해하기

- 난독화 (Obfuscation): 코드나 데이터가 읽기 어렵게 변형된다
  - 변수 이름, 함수 이름을 의미 없게 바꿈
  - 의미 없는 instruction을 끼워 넣어 디컴파일이 안되게 함
  - 코드 제어 흐름을 복잡하게 만듦
- 패킹: 바이너리를 압축시켜 원본 코드를 찌그려(?) 뜨린다
  - 코드 크기 줄어듦
  - 바이너리 암호화

바이너리 분석 방해하기

- 안티 디버깅: 디버깅 안 되게 한다
  - 디버거를 감지한다
  - 바이너리가 디버거 내에서 실행 되는 것을 방지한다
  - 디버거가 감지되면 바이너리 강제 종료 or 특정 코드 블록으로 점프
  - API 사용: **IsDebuggerPresent()** 등
  - PEB.BeingDebugged

```
mov eax, fs:[30h] // PEB를 가리키는 레지스터
mov al, [eax+2] // BeingDebugged 플래그
```

  - 자체 무결성 검사: 코드가 변경되었나 감시
  - 타이밍 체크: 디버거를 사용하면 코드 실행 시간이 늘어나는 것을 이용

바이너리 분석 방해하기

- 코드 가상화: 코드를 가상 머신(VM) 위에서 실행
  - 원본 코드를 가상머신 코드로 변환
  - 가상 머신 Interpreter가 이를 해석하며 실행

```
int add(int a, int b) {
    return a + b;
}
```

원본 코드

```
PUSH a
PUSH b
ADD
POP result
```

변환된 코드

```
void interpret(bytecode *code) {
    while (*code != END) {
        switch (*code) {
            case PUSH:
                // push to stack
                stack_push(*(++code));
                break;
            case ADD:
                // pop two values from stack
                int b = stack_pop();
                int a = stack_pop();
                // push sum result
                stack_push(a + b);
                break;
            case POP:
                // pop from stack
                result = stack_pop();
                break;
        }
        code++;
    }
}
```

바이너리 분석 방해하기

- 어셈블리 난독화
- 디컴파일러가 힘들어한다
- 중요한 값을 숨김

```
lea rcx, [0xdead]
```

⟶

```
lea rdx, [0x1ce54]
sub rdx, 0xefa8
push rdx
pop r9
mov rcx, r9
add rcx, 1
```

바이너리 분석 방해하기

- 미티게이션 함수 난독화 (_security_init_cookie)

```
void __cdecl _security_init_cookie()
{
  uintptr_t v0; // rax
  unsigned __int64 v1; // [rsp+30h] [rbp+10h] BYREF
  struct _FILETIME SystemTimeAsFileTime; // [rsp+38
  LARGE_INTEGER PerformanceCount; // [rsp+40h] [rbp

  v0 = _security_cookie;
  if ( _security_cookie == 0x2B992DDFA232i64 )
  {
    SystemTimeAsFileTime = 0i64;
    GetSystemTimeAsFileTime(&SystemTimeAsFileTime);
    v1 = (unsigned __int64)SystemTimeAsFileTime;
    v1 ^= GetCurrentThreadId();
    v1 ^= GetCurrentProcessId();
    QueryPerformanceCounter(&PerformanceCount);
    v0 = ((unsigned __int64)&v1 ^ v1 ^ PerformanceC
    if ( v0 == 0x2B992DDFA232i64 )
      v0 = 0x2B992DDFA233i64;
    _security_cookie = v0;
  }
  qword_140003000 = ~v0;
}
```

```
__int64 sub_1400092B2()
{
  __int64 v4; // rax
  __int64 v5; // rbx
  __int64 v7; // rcx
  __int64 result; // rax
  unsigned __int64 v9; // [rsp+30h] [rbp+10h] BYREF
  struct _FILETIME SystemTimeAsFileTime; // [rsp+38h] [rbp+18h] BYREF
  LARGE_INTEGER PerformanceCount; // [rsp+40h] [rbp+20h] BYREF

  _CF = 0;
  _ZF = 0;
  _OF = 0;
  v4 = _security_cookie;
  __asm { pushf }
  v5 = __ROL8__(
         (1780018988i64 - __ROL8__((1686453381i64 - __ROL8__(0xE40009D721F0B75Fui64, 180)) ^
         189);
  __asm { popf }
  if ( _security_cookie == v5 )
  {
    SystemTimeAsFileTime = 0i64;
    GetSystemTimeAsFileTime(&SystemTimeAsFileTime);
    v9 = (unsigned __int64)SystemTimeAsFileTime;
    v9 ^= GetCurrentThreadId();
    v9 ^= GetCurrentProcessId();
    QueryPerformanceCounter(&PerformanceCount);
    v4 = __ROL8__(
           (1377977522i64
           - __ROL8__(
               (2011552730i64 - __ROL8__((1596978736i64 - __ROL8__(0x1AF9B2403FEFD2DFi64, 83)
               18)) ^ 0x58FD8472,
             186) & ((unsigned __int64)&v9 ^ v9 ^ PerformanceCount.QuadPart ^ ((unsigned __int
    _CF = 0;
    _OF = 0;
    _ZF = v4 == 0;
    _SF = v4 < 0;
    __asm { pushf }
    v7 = __ROL8__((1588557972i64 - __ROL8__(0x5DCCFFD8F989182i64, 123)) ^ 0x7227789E, 217);
```


POC SECURITY

바이너리 분석 방해하기

- 위 기법들은 독립적으로 적용 가능
- 바이너리 분석의 난이도를 대폭 올리는 ~~나쁜~~ 상용 도구들이 많다
- VMProtect, Themida 등이 대표적인 도구들
  - 깃허브에 오픈소스 도구들도 많이 있다
- 안티 디버깅, 난독화, 패킹, 코드 가상화 등 각종 기법 동시 다발적 적용 🤬
- 파일 섹션 랜덤화, vmp0, vmp1과 같은 (이상한) 섹션 명

결론

바이너리 분석은 너무 어렵다 !

앞으로 공부해야 할 것들

- 여러 아키텍처에 대한 이해 (PPC, ARM, RISC-V, MIPS 등) 필요하다면...
- 디버깅 도구 사용법 습득 (gdb, windbg, IDA Pro)
- 분석 방해 기법 파악 및 우회법 학습
- 고급 기법도 공부해보기 (Symbolic execution, Taint analysis 등등)
- 여러 분석 보고서 보기
- 대학원 가기

Let’s enjoy CTF…

감사합니다.

QnA

POC SECURITY