Journal of Korea Multimedia Society Vol. 13, No. 2, February 2010(pp. 161-170)

정적 링크된 ELF 파일에서의 외부 심볼 정보 복구 기법

김 정 인†

요 약

ELF는 실행과 링킹 형식(Executable and Linkable Format)의 약어로서, 리눅스 시스템에서 사용하는 공유 라이브러리와 실행 파일을 위한 기본 파일 형식이다. 한편 링커는 정적 링킹을 통한 목적 파일 생성 시, 정적 공유 라이브러리의 심볼 정보를 목적 파일 내 심볼 테이블에 복사한다. 이 때, 심볼 테이블은 공유 라이브러리가 제공하는 함수 이름을 포함하여 여러 가지 디버깅 관련 정보를 포함하는데, 프로그램 실행에 직접적인 영향을 미치지 않는다는 점을 이용하여 보안상 디버깅되는 것을 방지하기 위해 삭제될 수 있다. 본 논문은 심볼 테이블이 삭제된 ELF 목적 파일에서 정적 공유 라이브러리의 심볼 정보를 복구하는 방법을 제안하고 실제로 실험을 행하여 복구된 심볼 정보를 확인한다.

A Recovery Method of External Symbol Information in Statically-Linked ELF Files

Jung-In Kim†

ABSTRACT

ELF, an abbreviation for Executable and Linkable Format, is the basic file format for shared libraries and executable files used in the Linux system, whereas 'Linker' copies the symbol information of static shared libraries into the symbol table in the target file generated by way of static linking. At this time, the symbol table keeps various pieces of debugging-related information including function names provided by the shared libraries, and it can be deleted to avoid debugging for security reasons by utilizing the fact that it does not directly affect the program execution. This paper proposes a method for restoring the symbol information of static shared libraries from the ELF object file in which the symbol table is deleted, and confirms that the symbol information is restored by conducting practical experiments.

Key words: Static Linking(정적링킹), Dynamic Linking(동적링킹), Symbol Information(심볼 정보), Recovery(복구), ELF(실행과 링킹형식 파일)

1. 서 론

파일로 된 바이너리가 운영체제에 의해 실행되기 위해서는 규격화된 형식의 정보가 필요하다. 리눅스 시스템에서는 공유 라이브러리와 실행 파일을 위한 기본 파일 형식으로 ELF를 사용한다. ELF는 실행과 링킹 형식(Executable and Linkable Format)의 약어로서, 다양한 아키텍처와 운영체제 상에서 프로그램을 실행하기 위한 여러 가지 정보를 포함한다. ELF 정보는 컴파일러에 의해 자동으로 생성되며, 실행 시 로더를 통해 커널에 전달된다[1,2].

공유 라이브러리나 실행 파일을 생성하기 위한 방법에는 크게 정적 링킹과 동적 링킹이 있다. 그 중 정적 링킹은 여러 목적 파일을 저장하는 편리한 방법

※ 교신저자(Corresponding Author) : 김정인, 주소 : 부산광역시 남구 용당동 535(608-711), 전화 : 051)629-1174, FAX : 051)629-1169, E-mail : jikim@tu.ac.kr

접수일 : 2009년 3월 11일, 수정일 : 2009년 7월 25일
완료일 : 2009년 10월 30일

† 종신회원, 동명대학교 컴퓨터공학과 부교수

으로, 링커는 정적 공유 라이브러리에 저장된 심볼 색인을 사용하여 ELF 목적 파일에 있는 심볼을 찾는다. 정적 공유 라이브러리의 내용과 색인 정보는 결과로 만들어질 목적 파일의 본체로 복사된다. 목적 파일에 저장된 색인 정보는 디버깅을 돕기 위한 것으로, 프로그램 내부에서 정적 공유 라이브러리의 함수 이름을 찾아 출력하는 스택 역추적 기능을 구현하는데 사용된다. 색인 정보가 저장되는 심볼 테이블은 프로그램 실행에 직접적인 영향을 미치지 않는다는 점을 이용, 보안상 디버깅되는 것을 방지하기 위해 삭제될 수 있다. 하지만 소스 코드가 분실된 프로그램이나 소스코드 없이 배포된 프로그램을 유지보수해야 할 경우, 심볼 테이블이 삭제되었다면 역공학 기법을 이용하더라도 디버깅이 매우 어려워진다는 단점도 존재한다.

본 연구에서는 ELF 목적 파일 내 재배치 정보를 이용하여 심볼 테이블이 삭제된 파일에서 심볼 정보를 복구하는 방법을 제안하고자 한다. 연구 결과, 목적 파일 내 재배치 정보를 이용하여 파일 본체에 복사된 정적 공유 라이브러리 코드를 새롭게 가공한 뒤, 원본 정적 공유 라이브러리 코드와 비교하는 방법으로 심볼 정보를 복구할 수 있었다. 또한 IDA Pro SDK를 이용하여 플러그인 형식으로 구현, 연구 결과를 시각적으로 표시하였다.

본 논문은 2장에서 관련연구, 3장에서는 본 연구에 대상이 되는 ELF파일의 형식과 재배치, 동적 바인딩 시의 심볼 테이블 상태에 대해서 기술하고, 4장에서는 심볼정보를 복구하는 방법을 제안하며, 5장에서 실제로 심볼정보 복구 시스템을 구현하고 그 결과를 분석하였으며 6장에서 결론을 맺는다.

2. 관련연구

주어진 실물로 부터 공학적 개념이나 형상 모델을 추출해 내는 과정을 역공학이라 한다. 전통적인 순공학은 개념으로부터 실물을 만드는 과정이라 한다면 역공학은 실물로부터 추상적 개념을 얻는 과정이라 할 수 있다[3]. 본 연구도 역공학에 기반한 심볼 복구 방법의 제안이라 볼 수 있다.

역공학은 소프트웨어 재사용에도 기여한다. 그 예로 C 및 C++로 개발된 원시 코드를 대상으로 역공학을 이용하여 재사용 가능한 부품을 추출하고 이 부품을 저장 및 검색, 합성하여 사용할 수 있는 재사용 시스템 CSORUS(C and C++ SOurce ReUse System)를 설계 및 구현한 연구가 있다[3]. 역공학을 이용하여 재사용 부품을 구축할 경우, 실무 분야에서 적용되어 정확성을 검증받은 신뢰도가 높은 부품이므로 재사용 부품의 신뢰도와 새로운 시스템의 유지보수를 쉽게 할 수 있다.

또한, 소프트웨어 역공학에서의 추상화를 위한 연구가 진행되었다. 위의 연구에서는 두 종류의 추상화 방법-문제영역적과 구조적 방법-이 논의되어졌다. 문제 영역 추상화는 프로그램이 쓰여지는 적용 분야에서의 개념과 일치하며 대개의 경우 프로그램 내에 직접 나타나지 않으며, 구조적 추상화는 불필요한 함수적 기술요소를 제거하고 루프문의 역할에 대한 요약을 생성해냄으로써 원시코드의 기능에 대한 추상화된 기술을 위해 사용된다. 문제 영역 추상화는 형식 코멘트언어를 사용하였고 이 정보는 코드 내에 삽입되어 구조적 추상화를 사용함에 더욱 간략화 된 기능명세서의 생성을 가능하게 한다. 이 접근 방법은 여러 종류의 COBOL 프로그램에 적용되어 여러 차례의 실험 연구를 통해 원시코드로부터 유용한 추상화 명세서를 생성함이 확인되었다[4].

한편, 정보보호 분야에도 역공학이 활용된다. 역공학에 기초한 악성코드 분석 시스템의 설계[5]는 역공학 기술을 기반으로 바이러스, 웜, 이질 및 분실코드 등의 소프트웨어 수행에 영향을 주는 악성코드를 분석하여 복구 및 문서화하는 방안을 제안하였다. 이 시스템은 구현을 통해 실행코드의 고급언어 프로그램과 사례에 대한 문서를 얻을 수 있으며, 악성코드 검출을 위한 해결방안을 제공한다.

또한, 암호화 모듈에 대한 역공학 공격을 방어하기 위한 연구가 있다[6]. 각종 정보통신 서비스를 제공하기 위한 소프트웨어는 정보보호 기술의 핵심요소인 암호화 모듈을 탑재하고 있는데, 여기서 암호화 모듈은 사용자 인증, 콘텐츠보호, 프라이버스 보호 등의 여러 가지 정보보호 기능을 구현하기 위한 핵심 모듈이다. 그러나, 악의적인 공격자에게 암호화 모듈을 사용하는 소프트웨어는 기밀한 정보를 다룬다는 점에서 공격의 대상이 될 수 있다. 또한 사람이 제작하는 소프트웨어는 크기가 커지고 복잡해질수록 위협 요소는 증가하기 마련이다. 따라서, 위의 연구에서는 암호화 모듈이 탑재된 소프트웨어가 역공학 분

석측면에서 악의적인 공격자에게 어떠한 위협 요소를 노출하는지 살펴보고, 이러한 역공학 분석 공격으로부터 보호하기 위해 소프트웨어 제작에서 고려해야 할 사항을 제시하였다.

오브젝트코드에서 잃어버린 심볼의 복구 방법을 제시한 본 연구는 위의 관련연구들과 직접 비교될 수는 없지만, 역공학적인 측면에서 거슬러 올라가면서 필요한 정보를 찾아내는 방법은 유사하다고 하겠다.

3. 본 논문에서 다루는 ELF 파일 형식

3.1 ELF 헤더형식

ELF 파일은 세그먼트 집합과 섹션 집합이라는 두 가지 방식으로 해석이 가능하다. 섹션은 기계어와 심볼 테이블 같은 구체적인 정보를 포함하는 작은 조각이며, 세그먼트는 동일한 메모리 속성을 갖는 섹션을 하나 이상 포함한 더 큰 단위이다[7].

그림 1에 나타낸 것과 같이 모든 공유 라이브러리와 실행 파일은 텍스트 세그먼트와 데이터 세그먼트를 각각 하나씩 가진다. 텍스트 세그먼트는 읽기 전용으로 ELF 헤더, 텍스트 섹션, 문자열 테이블, 읽기 전용 자료 심볼을 포함하며 데이터 세그먼트는 읽기 쓰기 속성으로 초기화된 쓰기 가능한 자료 심볼, 전역 오프셋 테이블, 초기화되지 않은 메모리를 포함한다. 또한 추가 정보로 심볼 테이블 섹션과 섹션 헤더 테이블이 제공되는데, 앞서 언급한 심볼 색인은 바로 심볼 테이블 섹션에 저장된다.

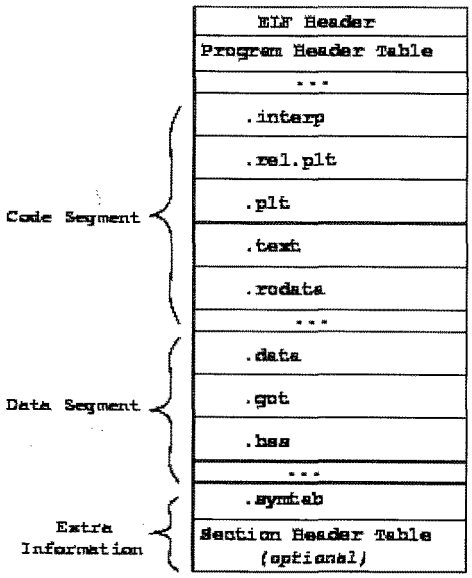


그림 1. ELF 파일의 구조

3.2 링킹

링킹은 공유 라이브러리나 실행파일에 목적 파일의 심볼을 바인딩하는 작업이다[2,8]. 다른 공유 라이브러리가 공개한 심볼이나 바인딩 예정으로 있는 목적 파일에 들어있는 함수와 변수를 사용하여 정의되지 않은 심볼을 찾아 결정하며, 공유 라이브러리나 실행파일이 메모리에 어떻게 올라오는지 관련 정보를 포함한 프로그램 헤더를 생성한다. 또한 정적 링킹 시 각 목적 파일의 텍스트 섹션은 목적 파일의 본체에 직접 복사되는데, 이는 삭제된 심볼 테이블을 복구하는데 있어 주요한 수단을 제공한다. 그림 2는 독립적인 목적 파일 네 개가 바인딩한 공유 라이브러리 상태를 보여준다.

목적 파일 네 개에서 나온 각 섹션은 공유 라이브러리에서 더 큰 섹션으로 결합된다. 이 섹션들은 더 큰 연속적인 영역인 세그먼트에 포함되어 있으며, 실행 시 읽기, 쓰기, 실행과 같은 특정 메모리 속성을 띠고 메모리로 적재된다.

3.3 심볼 테이블

그림 3은 심볼 구조체를 표현한 것이다. 심볼 테이블은 ELF 심볼 구조체의 배열이며 함수, 변수, 다른

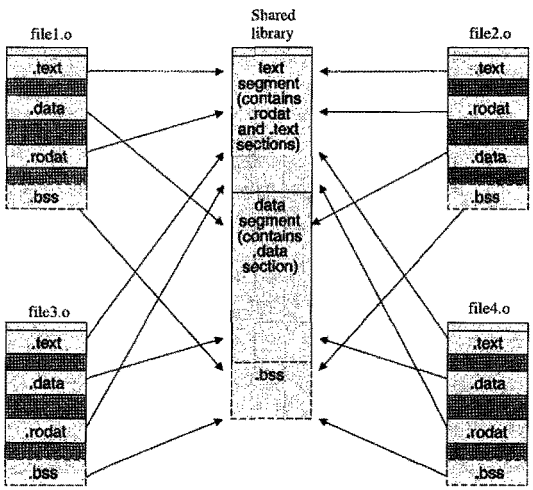


그림 2. 공유 라이브러리와 목적 파일 링킹

```
typedef struct
{
  Elf32_Word    st_name;      /* Symbol name (string tbl index) */
  Elf32_Addr    st_value;     /* Symbol value */
  Elf32_Word    st_size;      /* Symbol size */
  unsigned char st_info;      /* Symbol type and binding */
  unsigned char st_other;     /* Symbol visibility */
  Elf32_Section st_shndx;     /* Section index */
} Elf32_Sym;
```

그림 3. 심볼 구조체

심볼 유형을 기술한다. ELF 파일은 기본적으로 두 개의 심볼 테이블을 가진다. 하나는 동적 심볼 테이블로 불리며 ELF 실행 파일에서 다양한 심볼을 실행 중에 찾기 위해 사용된다. 나머지 하나는 주 심볼 테이블로서 실행 중에 사용하지 않는 정적 심볼 테이블을 포함하여 ELF 목적 파일에 들어있는 다른 모든 심볼을 가진다[7,9].

st_name은 심볼 이름에 대한 문자열 테이블 색인이며, st_value는 ELF 파일에서 변위를 나타내거나 메모리에 올릴 경우 심볼 주소를 포함한 섹션에서 변위를 나타낸다. 이 값은 공유 라이브러리에서 ELF 파일 내부 변위를 나타내는 반면, 실행 파일에서는 실제 주소를 나타낸다.

3.4 재배치 테이블

재배치는 메모리에 올라온 ELF 섹션에 들어있는 주소들을 그에 대응하는 함수나 변수의 현재 주소로 바꾸는 과정이다. 그림 4에 재배치된 결과를 나타낸다.

재배치 테이블은 공유 라이브러리가 어느 주소 공간에나 올라올 수 있도록 하는데 필요하다. 그림 5는 재배치에 필요한 정보를 담은 구조체를 나타낸다.

재배치 구조체는 재배치가 필요한 메모리 주소와 재배치 유형과 심볼 색인에 대한 정보를 포함한다.

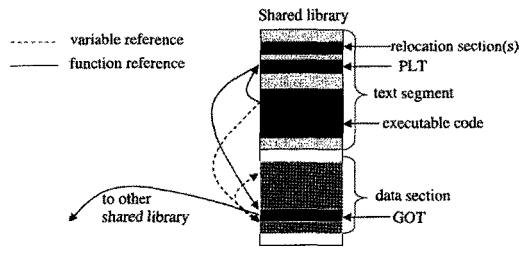


그림 4. 재배치

```
typedef struct
{
  Elf32_Addr    r_offset;  /* Address */
  Elf32_Word    r_info;    /* Relocation type and symbol index */
} Elf32_Rel;
```

그림 5. 재배치 구조체

r_offset 필드는 재배치에 의해 바뀌어야 하는 목표 주소나 변위이다. 목적 파일일 경우 변위는 영향을 받는 섹션 내부이며, 공유 라이브러리나 실행 파일일 경우 변위는 심볼의 주소이다. r_info 필드는 재배치 유형과 심볼 색인을 나타낸다[10].

3.5 동적 링킹에서의 바인딩

동적 링킹에서의 바인딩은 실행 중 정의되지 않은 심볼을 찾아 연결시키는 과정으로, 동적 공유 라이브러리를 사용하는 모든 바이너리는 로더를 통한 바인딩 과정을 거친다. 리눅스에서 사용하고 있는 재배치 방식은 지연 바인딩이며, 이는 실제로 함수를 처음 호출할 때 심볼을 링킹하는 방식이다[11]. 앞으로 논의될 정적 링킹에서의 바인딩과의 차이점을 인지하기위해 동적 링킹에서의 바인딩 방법과 역공학 기법을 통한 함수 정보 습득 방법을 우선 기술한다.

```
/* example.c */
#include <stdio.h>
int main()
{
    printf("Hello, ELF!\n");
    return 0;
}
```

위 코드는 실질적인 비교를 위해 만든 간단한 예제 프로그램이다. 본 예에서는 동적 링킹 방식으로 컴파일 하도록 한다.

```
$> gcc example.c -o example
$> cp example example_stripped
$> strip -S exmaple_stripped
```

생성된 example 파일과 example_stripped 파일은 심볼 테이블의 유무에만 차이가 날뿐 그 외 다른 부분에서는 완전히 동일하다. 그림 6은 동적링킹 결과를 gdb 디버거로 보여준 것이다.

```
(gdb) disas main
Dump of assembler code for function main:
0x0000000000400498 <main+0>:    push   %rbp
0x0000000000400499 <main+1>:    mov    %rsp,%rbp
0x000000000040049c <main+4>:    mov    $0x400596,%edi
0x00000000004004a1 <main+9>:    callq  0x4003c0 <puts@plt>
0x00000000004004a6 <main+14>:   mov    $0x0,%eax
0x00000000004004ab <main+19>:   leaveq
0x00000000004004ac <main+20>:   retq
End of assembler dump.
```

그림 6. gdb 디버거로 확인한 동적 링킹에서의 바인딩

gdb 디버거로 example_stripped 프로그램을 실행시켜보면, 심볼 테이블이 삭제되었음에도 함수들의 이름이 올바로 출력되는 것을 알 수 있다. 그림 6에서 printf() 함수가 아닌, puts() 함수가 호출된 이유는 컴파일러의 판단에 의해 최적화가 이루어졌기 때문이다. 하지만 여기서는 함수 이름의 표시 여부가 관건이므로 불필요한 설명은 생략한다.

로더에는 동적 공유 라이브러리 내 함수의 실제 주소를 얻기 위한 함수들이 존재한다. 리눅스의 경우 해당 함수의 이름은 _dl_runtime_resolve()와 _dl_fixup()이다. 프로그램에서 처음으로 공유 라이브러리의 함수를 호출하면, _dl_runtime_resolve() 함수가 실행된다. 더 자세히 알아보기 위해 역공학 기법으로 해당 함수를 분석하여 그림 7에 나타내었다.

%eax 레지스터에는 _dl_runtime_resolve() 함수를 부르기 전 저장했던 link_map 구조체 변수 1의 주소가 들어있고, %edx 레지스터에는 _dl_runtime_resolve() 함수 plt를 호출하기 전에 프로그램 내의 함수 plt에서 저장한 reloc_offset 값이 들어있다. 이 값들이 무엇인지 다음 절에서 설명한다.

3.6 link_map 구조체

link_map 구조체는 로더가 참조하는 링크 지도이다. 바인딩되는 동적 공유 라이브러리의 정보를 포함하며 그림 8과 같은 구조를 갖는다.

실제 link_map 구조체의 값을 확인하려면, 디버거

```
00012ec0 <_dl_runtime_resolve>:
   12ec0:       50                      push   %eax
   12ec1:       51                      push   %ecx
   12ec2:       52                      push   %edx
   12ec3:       8b 54 24 10             mov    0x10(%esp),%edx
   12ec7:       8b 44 24 0c             mov    0xc(%esp),%eax
   12ecb:       e8 60 ab ff ff          call   da30 <_dl_fixup>
   12ed0:       5a                      pop    %edx
   12ed1:       59                      pop    %ecx
   12ed2:       87 04 24                xchg   %eax,(%esp)
   12ed5:       c2 08 00                ret    $0x8
   12ed8:       90                      nop
   12ed9:       8d b4 26 00 00 00 00    lea    0x0(%esi),%esi
```

그림 7. _dl_runtime_resolve() 함수

```
struct link_map
  {
    /* These first few members are part of the protocol with the debugger
       This is the same format used in SVR4.  */

    ElfW(Addr) l_addr;          /* object가 로드된 기본 주소.  */
    char *l_name;               /* object가 발견된 절대 파일 이름.  */
    ElfW(Dyn) *l_ld;            /* 공유 object의 동적 섹션.  */
    struct link_map *l_next, *l_prev; /* 로드된 object의 체인.  */
  };
```

그림 8. link_map 구조체

```
(gdb) x $eax
0x98d6c8:       0x00000000 // l_addr (1) <--+
(gdb)                                        |
0x98d6cc:       0x0098992b // l_name        |
(gdb)                                        |
0x98d6d0:       0x08049474 // l_ld          |
(gdb)                                        |
0x98d6d4:       0x0098d9a8 // l_next ------>|
(gdb)                                        ||
0x98d6d8:       0x00000000 // l_prev       ||
(gdb) x/x 0x0098d9a8                         ||
0x98d9a8:       0x002a6000 // l_addr (2)<--+| <--+
(gdb)                                        |   |
0x98d9ac:       0x0098992b // l_name        \   \
(gdb)                                        |   |
0x98d9b0:       0x002a6590 // l_ld          |   |
(gdb)                                        |   |
0x98d9b4:       0xb7f85000 // l_next-------->|   |
(gdb)                                        ||  |
0x98d9b8:       0x0098d6c8 // l_prev: 이전 (1) link_map 주소를 가리킴
(gdb) x 0xb7f85000                           |   |
0xb7f85000:     0x004af000 // l_addr (3)<--+   |<----------+
(gdb)                                        |           |
0xb7f85004:     0x0098ded8 // l_name: /lib/libc.so.6      |
(gdb)                                        |           |
0xb7f85008:     0x005e7d9c // l_ld           |           |
(gdb)                                        |           |
0xb7f8500c:     0x0098d300 // l_next ------>+           |
(gdb)                                        |           |
0xb7f85010:     0x0098d9a8 // l_prev: 이전(2) link_map 주소를 가리킴
(gdb) x/s 0x0098ded8                         |           |
0x98ded8:       "/lib/libc.so.6"             |           |
(gdb) x/x 0x0098d300                         v           |
0x98d300 <_rtld_local+768>:     0x00973000 // l_addr (4)  |
(gdb)                                                    |
0x98d304 <_rtld_local+772>:     0x08048114 // l_name: /lib/ld-linux.so.2
(gdb)                                                    |
0x98d308 <_rtld_local+776>:     0x0098cef8 // l_ld        |
(gdb)                                                    |
0x98d30c <_rtld_local+780>:     0x00000000 // l_next      |
(gdb)                                                    |
0x98d310 <_rtld_local+784>:     0xb7f85000 // l_prev: 이전 (3) link_map 주소를 가리킴
(gdb)
```

그림 9. link_map 구조체들의 구조

를 이용하여 _dl_fixup() 함수에서 정지한 후, 인자로 넘어온 %eax 레지스터의 내용을 확인하면 된다.

그림 9는 메모리 상의 link_map 구조체들의 구조이다. link_map 구조체들은 더블 링크드 리스트 구조로 연결되어 있는데, l_next와 l_prev는 각각 다음과 같이 이전 구조체를 포인터로 가르킨다.

재배치 테이블 시작점인 JMPREL과 심볼 테이블 시작점인 SYMTAB, 문자열 테이블 시작점인 STRTAB 등을 구하기 위해 선언된 테이블 인덱스는 그림 10과 같다. _dl_fixup() 함수 시작 직후, 이 내용들을 참고하여 각 테이블 위치를 구해온다.

앞서 link_map 구조체 변수 1과 reloc_offset을 인자로 받아왔다. 이 link_map 구조체를 통해 문자열 테이블, 심볼 테이블, 재배치 테이블 등의 주소를 구할 수 있다.

dl_fixup() 함수는 재배치테이블을 기준으로 심볼 테이블을 구하고, 심볼테이블 내용을 통해 문자열테이블에 있는 실제 함수의 이름을 찾는다.

_dl_lookup_symbol_x() 함수는 elf/dl-lookup.c 파일에 있으며, 로드된 객체의 심볼 테이블들을 검사하

```
#define DT_STRTAB       5               /* Address of string table */
#define DT_SYMTAB       6               /* Address of symbol table */
#define DT_JMPREL       23              /* Address of PLT relocs */
```

그림 10. elf.h 헤더에 선언된 테이블 인덱스

```
(gdb) x $eax
0x110000:       0x464c457f  ; return->l_addr: 라이브러리 시작 주소
(gdb) x $edx
0x11c074:       0x00002a7e  ; sym->st_name
(gdb)
0x11c078:       0x00015e50  ; sym->st_value
(gdb) x $eax + *($edx + 4)          ; return->l_addr + sym->st_value
0x125e50 <__libc_start_main>:   0x89d23155
(gdb)
```

그림 11. gdb 디버거로 확인한 함수 이름

여 심볼 undef_name의 정의를 찾는 역할을 한다. 예로, 첫 번째 인자로 들어가는 strtab + sym->st_name 계산을 통해 함수 이름을 구할 수 있다.

그림 11에서 실제 함수의 주소가 나오는 것을 볼 수 있다. 결론적으로 %eax에는 실제 함수 주소가 저장된다.

4. 심볼 복구 시스템 설계

4.1 심볼 복구 개요

정적 링크된 ELF 실행 파일에서 심볼 테이블이 삭제되었을 경우에도 올바른 실행을 위해 정적 공유 라이브러리의 코드는 파일의 본체에 존재한다. 반대로 생각해보면 이를 기반으로 원본 정적 공유 라이브러리 코드와 비교하여, 삭제된 심볼 테이블의 정보를 유추해 낼 수 있음을 의미한다. 하지만, 정적 공유 라이브러리의 코드 원본이 그대로 존재하는 것은 아니다. 라이브러리 내 함수나 변수들에 접근하는 코드는 재배치 정보를 참조하여 상대적 메모리 주소들로 수정되기 때문이다. 수정되는 조건과 방법을 알아내기 위해 실행 파일에 복사된 정적 공유 라이브러리 코드와 원본 정적 공유 라이브러리 코드를 비교하는 실험을 진행하였다.

4.2 실행 파일에 복사된 정적 공유 라이브러리 코드와 원본 정적 공유 라이브러리 코드의 비교

```
/* example.c */
#include <stdio.h>
int main()
{
    printf("Hello, ELF!\n");
    return 0;
}
```

이 실험에서는 printf() 함수에 대한 심볼 정보를 비교의 대상으로 선정하였다. 만약 정적 링킹을 통한 컴파일로 생성된 실행 파일의 printf() 함수 코드와 원본 정적 공유 라이브러리의 printf() 함수 코드에 존재하는 차이점에서 어떠한 법칙을 알아낼 수 있다면 역으로 원본 정적 공유 라이브러리의 심볼 정보를 유추해 낼 수 있다.

먼저 해당 코드를 리눅스 시스템에서 libc.a 정적 공유 라이브러리와의 정적 링킹을 통한 컴파일을 수행하여 실행 파일을 생성한 뒤, strip 도구를 이용하여 심볼 테이블을 삭제하였다.

```
$> gcc example.c -o example -static
$> cp example example_stripped
$> strip -S exmaple_stripped
```

다음으로 비교의 대상이 되는 원본 정적 공유 라이브러리에서 printf() 함수 코드를 추출하였다. 정적 공유 라이브러리는 해당 라이브러리가 포함하는 수많은 목적 파일들을 archive 방식을 이용하여 하나의 파일에 담고 있다. printf() 함수는 stdio.h 헤더에 선언되어 있으며, stdio.h 헤더는 표준 C 라이브러리의 헤더 파일이다. 리눅스에서 표준 C 라이브러리 파일은 libc.so(동적 공유 라이브러리)와 libc.a(정적 공유 라이브러리)에 해당된다. 본 실험은 정적 공유 라이브러리에 대한 것이므로 libc.a 파일을 분석하도록 한다. 앞서 언급하였듯이 libc.a 파일은 archive(이하 ar로 표기) 방식으로 묶어져있으므로 ar 도구를 이용하여 원래의 파일들로 풀어낸다.

```
$> cp /usr/lib/libc.a /tmp
$> ar /tmp/libc.a
```

풀어낸 파일들 중에 printf.o 파일이 printf() 함수 코드를 담고 있는 목적 파일이다. printf.o 파일의 ELF 헤더 정보를 분석해보면, 파일 오프셋 기준 0×34부터 printf() 함수의 코드가 존재함을 알아낼 수 있다. 그 크기는 26바이트로 해당 영역은 0×34부터 0×4d까지이다.

헥사 에디터를 이용하여 해당 영역의 코드를 추출하였다. 그림 12는 원본 공유라이브러리의 printf() 함수를 헥사값으로 표현한 것이고 그림 13은 실행파일에서 추출한 printf()함수를 헥사값으로 표현한 것

```
00000030                55 89 E5 8D  45 0C 83 EC 0C 50 FF 75
00000040  08 FF 35 00 00 00 00 E8  FC FF FF FF C9 C3
```

그림 12. 원본 공유 라이브러리의 printf() 함수

```
00001720                55 89 E5 8D  45 0C 83 EC 0C 50 FF 75
00001730  08 FF 35 3C ED 05 08 E8  18 3C 00 00 C9 C3
```

그림 13. 실행 파일에서 추출한 printf() 함수

이다.

그리고 정적 링킹을 통한 컴파일로 생성된 example_stripped 파일에서의 printf() 함수 코드를 추출한 결과와 비교하였다.

그림 14에서 붉은 색으로 마킹된 부분이 두 파일 간의 차이점이다. 마킹된 부분이 정확히 어떤 코드인지 알아보기 위해 해당 코드를 역어셈블하였다.

그림 15는 gdb 디버거로 역어셈블한 printf() 함수의 코드이다. 밑줄 친 부분은 그림 8의 마킹된 부분에 해당한다. 실행 파일에서 추출한 코드와 원본 정적 공유 라이브러리의 코드에서 변경된 부분은 함수 오프셋 0×0f부터 0×12까지 4바이트, 0×14부터 0×17까지 4바이트 두 곳이다. 위 결과로 보아 두 명령은 모두 메모리를 직접 참조하며, 변경된 8바이트 코드는 직접 메모리 주소를 가리키는 코드로서 각각 printf 함수의 지역 변수인 __stdoutp의 주소와 vfprintf 함수를 호출하기 위한 주소를 나타냄을 알 수 있다. 링크되기전의 공유 라이브러리에는 각각 0×00 0×00 0×00 0×00과 0×fc 0×ff 0×ff 0×ff로 채워져 있던 공간이 링커가 계산한 특정 메모리 주소로 변경되었다.

4.3 심볼 복구의 예

링커는 공유 라이브러리의 재배치 테이블에 들어

```
00000030                55 89 E5 8D  45 0C 83 EC 0C 50 FF 75
00000040  08 FF 35 [redacted] E8 [redacted] C9 C3
```

그림 14. 원본 공유 라이브러리의 printf() 함수 코드와 실행 파일에서 추출한 printf() 함수의 비교

```
0x8049724 <printf>:    push   %ebp
0x8049725 <printf+1>:  mov    %esp,%ebp
0x8049727 <printf+3>:  lea    0xc(%ebp),%eax
0x804972a <printf+6>:  sub    $0xc,%esp
0x804972d <printf+9>:  push   %eax
0x804972e <printf+10>: pushl  0x8(%ebp)
0x8049731 <printf+13>: pushl  0x805ed3c
0x8049737 <printf+19>: call   0x804d354 <vfprintf>
0x804973c <printf+24>: leave
0x804973d <printf+25>: ret
```

그림 15. printf() 함수의 어셈블리 코드

있는 정보를 토대로 실행파일에서 공유 라이브러리의 함수들을 호출할 때의 메모리 주소를 계산한다. 공유 라이브러리의 재배치 테이블에는 상대적 메모리 주소로 수정해야 하는 코드에 대한 정보가 들어있다. libc.a에서 추출한 printf.o 파일의 재배치 테이블 내용을 확인하기위해 readelf 도구를 사용하였다.

```
$> readelf -r printf.o
```

그림 16은 printf.o 파일의 재배치 테이블을 조회한 결과이다. Offset 필드는 .text 섹션의 시작 위치를 기준으로 수정해야 할 오프셋을 의미한다. printf.o 파일에서 그 값은 각각 0×0000000f와 0×00000014로, 앞서 수행한 실험의 결과와 동일함을 알 수 있다. Type은 어떤 종류의 수정인지 구별하는데 사용된다. 여기서 R_386_32는 절대 주소 지정 방식을 나타내며 R_386_PC32는 PC 유효 주소 지정 방식을 나타낸다. Sym. Name 필드는 수정되는 재배치 항목에 대한 심볼 이름으로 각각 __stdoutp와 vfprintf에 대한 것임을 의미한다.

이 결과를 통해 링커가 링킹을 수행할 때 어떤 정보를 토대로 실행 파일에 포함되는 정적 공유 라이브러리의 코드를 수정하는지 알아낼 수 있다. 이는 링킹이 수행되기 전이나 후 어떠한 경우에도 유효하다. 또한, 그림 17에 나타낸 것처럼 수정될 부분을 제외한 나머지 코드는 여전히 원본 정적 공유 라이브러리의 코드와 완전히 동일하다는 사실을 발견할 수 있는데, 이 사실은 본 논문에서 제안하는 복구 알고리즘에 주요한 수단을 제공한다.

정적 공유 라이브러리에 포함된 모든 재배치 테이블을 참조하여 시그니처 패턴으로 생성한 뒤, 실행 파일의 복사된 정적 공유 라이브러리 코드와 비교한다면 각각의 코드가 원본 정적 공유 라이브러리의

```
  Offset     Info    Type            Sym.Value  Sym. Name
0000000f  00000701 R_386_32          00000000   __stdoutp
00000014  00000802 R_386_PC32       00000000   vfprintf
```

그림 16. printf.o 파일의 재배치 테이블

```
00000030                55 89 E5 8D  45 0C 83 EC 0C 50 FF 75
00000040  08 FF 35 [redacted] E8 [redacted] C9 C3
```

수정 될 부분을 제외하여 비교

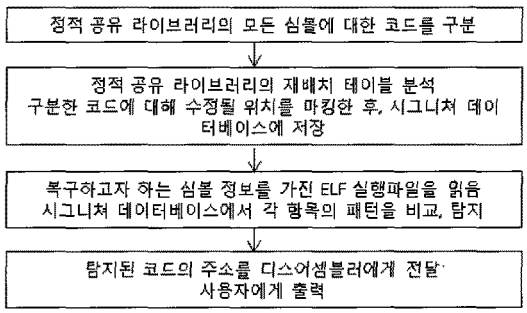
그림 17. 탐지 시그니쳐 생성 방법

어떤 부분인지 알아낼 수 있고, 이 정보를 토대로 삭제된 심볼 테이블을 복구할 수 있다.

5. 심벌 복구 시스템 구현

5.1 grayResolve 구현

IDA Pro는 역어셈블러로서 ELF 실행 파일을 분석하여 어셈블리 코드로 변환해주며, SDK를 통해 사용자 확장 기능을 제공한다. 본 논문에서는 IDA Pro SDK를 이용하여 심볼 정보 복구 애플리케이션인 grayResolve를 구현하였다. 프로그램의 흐름은 다음과 같다.



먼저 정적 공유 라이브러리 파일로부터 각각의 목적 파일들을 구분할 필요가 있다. 정적 공유 라이브러리 파일은 ar 형식으로 되어있으며 그 내용은 그림 18과 같다.

grayResolve에서는 각각의 목적 파일을 추출해내기 위해 elf_parse_object() 함수를 구현하였다. elf_parse_object() 함수는 정적 공유 라이브러리 파일을 읽어 들여, ar 헤더의 내용 중 파일 이름과 파일 크기를 참조하여 각각의 목적 파일에 대한 포인터를 다음 함수에 넘겨준다.

Field Offset from	Field Offset to	Field Name	Field Format
0	15	File name	ASCII
16	27	File modification timestamp	Decimal
28	33	Owner ID	Decimal
34	39	Group ID	Decimal
40	47	File mode	Octal
48	57	File size in bytes	Decimal
58	59	File magic	\140\012

그림 18. ar 형식의 헤더

elf_search_signature() 함수는 elf_parse_object() 함수로부터 넘겨받은 포인터를 기준으로 ELF 헤더 내용을 분석하여 재배치 테이블의 정보를 조회한 뒤, 함수들에 대한 시그니쳐 패턴을 생성한다. 그리고 실행 파일에서 일치되는 패턴이 있는지 검색하여, 매칭이 이루어지면 매치된 함수의 이름으로 역어셈블 출력을 수정한다. 또한 역어셈블 출력을 수정하는 것과 별도로 디버깅 윈도우에 매치되는 함수들의 이름을 출력하도록 하여 그 내용을 자세히 알 수 있게 하였다.

마지막으로 시그니쳐 패턴에 매치된 함수 정보를 사용자에게 효과적으로 보여주기 위해 IDA Pro SDK가 제공하는 do_name_anyway() 함수를 이용하였다. 그림 19에 do_name-anyway()의 함수 설명을 보여준다.

do_name_anyway() 함수는 역어셈블 출력 결과에서 주어진 주소에 대해 IDA Pro가 관리하는 내부 데이터 베이스에 접근하여 특정 명칭을 부여하도록 한다. 이는 따로 심볼 테이블의 정보를 ELF 형식에 맞게 복구할 필요 없이 역어셈블러의 도움을 받아 즉시 사용자에게 그 결과를 나타낼 수 있는 장점을 가진다.

그림 20은 심볼 정보 복구 전, 그림 21은 심볼 정보 복구 후의 결과이다. 두 가지 상태를 살펴보면 밑줄 친 부분에서 IDA Pro가 임시로 지정한 sub_8049724 이름의 함수가 printf로 변경되었음을 알 수 있다. 이

```
// Give a name anyway
// This function tries to give the specified name the specified addresss
// and tries to modify the name with a postfix ("_00") if the name
// already exists.
// It will try 100 different postfixes and fail only if all such names
// are already in use. For example, if the specified name is "xxx",
// the function will try to give names "xxx","xxx_0","xxx_2",.... "xxx_99"
//      ea      - linear address
//      name    - new name
//                NULL: return 0
//                "" : return 0
//      maxlen - if specified, the name will be truncated to have the specified
//               maximum length
// returns: 1-ok, 0-failure

idaman bool ida_export do_name_anyway(ea_t ea, const char *name, size_t maxlen=0);
```

그림 19. do_name_anyway() 함수 설명

```
.text:080481EC sub_80481EC     proc near               ; CODE XREF: start+71↓p
.text:080481EC                 push    ebp
.text:080481ED                 mov     ebp, esp
.text:080481EF                 sub     esp, 8
.text:080481F2                 and     esp, 0FFFFFFF0h
.text:080481F5                 mov     eax, 0
.text:080481FA                 add     eax, 0Fh
.text:080481FD                 add     eax, 0Fh
.text:08048200                 shr     eax, 4
.text:08048203                 shl     eax, 4
.text:08048206                 sub     esp, eax
.text:08048208                 sub     esp, 0Ch
.text:0804820B                 push    offset aHelloElf ; "Hello, ELF!\n"
.text:08048210                 call    sub_8049724
.text:08048215                 add     esp, 10h
.text:08048218                 mov     eax, 0
.text:0804821D                 leave
.text:0804821E                 retn
.text:0804821E sub_80481EC     endp
```

그림 20. 심볼 정보 복구 전

```
.text:080481EC sub_80481EC     proc near               ; CODE XREF: start+71p
.text:080481EC                 push    ebp
.text:080481ED                 mov     ebp, esp
.text:080481EF                 sub     esp, 8
.text:080481F2                 and     esp, 0FFFFFFF0h
.text:080481F5                 mov     eax, 0
.text:080481FA                 add     eax, 0Fh
.text:080481FD                 add     eax, 0Fh
.text:08048200                 shr     eax, 4
.text:08048203                 shl     eax, 4
.text:08048206                 sub     esp, eax
.text:08048208                 sub     esp, 0Ch
.text:0804820B                 push    offset aHelloElf ; "Hello, ELF!\n"
.text:08048210                 call    printf
.text:08048215                 add     esp, 10h
.text:08048218                 mov     eax, 0
.text:0804821D                 leave
.text:0804821E                 retn
.text:0804821E sub_80481EC     endp
```

그림 21. 심볼 정보 복구 후

```
grayResolver> Matched: __sys_read at 0x08052d48
grayResolver> Matched: read at 0x08052d48
grayResolver> Matched: _read at 0x08052d48
grayResolver> Matched: lseek at 0x08052d5c
grayResolver> Matched: __sys___syscall at 0x08052d80
grayResolver> Matched: __syscall at 0x08052d80
grayResolver> Matched: ___syscall at 0x08052d80
grayResolver> Matched: __error at 0x0804d558
grayResolver> Matched: ___error_unthreaded at 0x0804d558
grayResolver> Number of matched symbols: 504 / Total: 2838
```

그림 22. 심볼 정보 복구 결과

는 초기 실험에 사용되었던 example.c의 내용과 정확히 일치한다. 뿐만 아니라 실행 파일에서 직접 호출하지 않는 함수라도 정적 공유 라이브러리로부터 실행 파일에 복사되었으면 그 정보를 알아낼 수 있는데, 이는 바이트 단위 패턴 매칭 기반이기 때문에 가능한 것이다. 바이트 단위 패턴 매칭은 속도가 느리다는 단점이 있지만, 오탐 발생률이 거의 없다는 장점이 있다.

그림 22의 디버깅 윈도우에 표시되는 심볼 정보 복구 결과를 보면 libc.a 정적 공유 라이브러리의 총 2838개의 함수 중 실행 파일에 포함된 504개의 함수에 대한 정보를 복구했음을 알 수 있다. 나머지 2334개의 함수는 비능률적으로 실행 파일의 크기가 늘어나는 문제를 방지하기 위한 링커의 판단에 의해 실행 파일에 복사되지 않은 영역에 해당하는 함수들로서, 실행 파일의 수행과는 무관한 비-호출 함수이므로 복구 알고리즘의 성공률과는 무관하다.

5.2 향후 과제

본 논문에서 제시한 복구 알고리즘은 반드시 원본 정적 라이브러리 파일을 알고 있다는 가정 하에 올바른 동작을 보장한다. 만약 사용자 함수들을 정적 공유 라이브러리로 만든 뒤 목적 프로그램에 정적 링크하여 컴파일 한 경우, 정적 공유 라이브러리 파일을 배포하지 않는다면 본 논문에서 제시하는 방법을 적용할 수 없다. 하지만, 이러한 문제점은 일반적인 동적 링크된 실행 파일에도 존재하는 부분으로, 이를 전문적으로 연구하는 역공학이라는 분야가 존재할 정도로 그 내용과 범위가 방대하다. 따라서 본 논문에서 다루는 범위 밖의 내용으로 분류하여 언급을 생략하였다. 또한 디버깅을 방지하기 위한 목적으로, 본 논문에서 언급한 심볼 테이블을 삭제하는 방법 외에 안티-디버깅 코드 삽입 방법이 존재한다. 프로그램 스스로 디버거가 실행되어있는지를 탐지하는 루틴을 삽입하여, 탐지 시 자동으로 실행을 종료하거나 비정상적인 동작을 유발함으로써 디버깅을 막는 방법이다. 안티-디버깅을 자동으로 우회할 수 있는 방법은 아직 제시되지 않고 있는데, 차후 많은 연구와 논의가 이루어져야 할 부분이다.

6. 결 론

본 논문에서는 정적 링크된 ELF 파일에서 삭제된 심볼 테이블을 복구 하는 기법에 대해 다루었다. 또한 디스어셈블러와 연동되는 심볼 복구 애플리케이션인 grayResolve를 구현하여, 복구를 자동화하였다. 해당 애플리케이션을 이용하면 역공학을 하는데 장애물중 하나인 심볼 테이블의 삭제를 극복할 수 있으며, 디버깅을 방지하기위해 심볼 테이블을 삭제하는 것이 최선의 방법이 아님을 알 수 있다.

참 고 문 헌

[1] Alessandro Rubini, Jonathan Corbet, and Greg Kroah-Hartman, *Linux Device Drivers 3/E,* O'REILLY, 2005.

[2] Arthur Griffith, *GCC: The Complete Reference,* McGrew-Hill, 2002.

[3] 최은만, "역공학을 이용한 소프트웨어 재사용 시스템에 관한 연구," 정보처리학회 논문지, 4권 1호, pp. 97-106, 1997.

[4] 박수희, "소프트웨어 역공학에서의 문제 영역 및 구조적 초상화 방법," 동덕여자대학교 학술저널, Vol.27 No.1, pp. 331-358, 1997.

[5] 송진국, 오염덕, 강형우, 이진석, "역공학에 기초한 악성코드 분석 시스템의 설계," 한국인터넷정보학회 학술발표대회 논문집, 1권 2호, pp. 273-276, 2000.

[6] 김권엽, 최재민, 이상진, 임종인, "소프트웨어에 적용된 암호화 모듈의 역공학 분석에 관한 고찰," 한국방송공학회 학술발표대회 논문집, pp. 142-145, 2007.

[7] Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2, TIS Committee, 1995.

[8] Mark Wilding and Dan Behman, *Self-Service Linux : Mastering the Art of Problem Determination,* PRENTICE HALL, 2006.

[9] Mike Loukides and Andy Oram, *Programming with GNU Software,* O'REILLY, 1998.

[10] Julien Vanegue, *ELF relocation into non-relocatable objects,* Phrack, 2003.

[11] Daniel Plerre Boet and Marco Cesati, *Understanding the Linux Kernel 3/E,* O'REILLY, 2005.

김 정 인

1991년 4월~1993년 3월, 게이오대학 계산기과학전공 공학석사

1993년 4월~1996년 3월, 게이오대학 계산기과학전공 공학박사

1996년 5월~2월 포항공과대학교 정보통신연구소 연구원, 기계번역시스템 개발

1998년 3월~현재, 동명대학교 컴퓨터공학과 부교수

관심분야 : 기계번역, 기계학습, 시멘틱웹