Cheating the ELF

+--+
원문서 제목 : **Cheating the ELF**
원문서 저자 : the grugq
원문서 작성일 : ?
문 서 번역일 : 2003. 08. 00
문 서 번역자 : 광주광역쉬에서 세벌쉭
코 멘 트 : 본 번역본은 순전히 개인적인 호기심에서 번역한 것이다.
조금이라도 의심스러운 해석은 원문서를 참조하기 바란다.
+--+

Cheating the ELF

Subversive Dynamic Linking to Libraries

the grugq

Abstract:

특징적이고 다양한 유닉스 기생충들의 개발은 주 파일에서 외부 함수들로 믿을만한 접근을 할수 없음으로 제한되어져왔다. 지금까지, 기생 코드내부에서 라이브러리를 사용하는 것은 금지된 복잡한 작업이라는 사실로 받아들여졌다. 우리는 ELF 포멧의 동적 링킹 메커니즘들을 조사해 볼것이다. 기생코드가 공유 오브젝트에 접근하도록 허락하기위해서 어떻게 통과하는지 또는 어떻게 가로채는지를 알아 볼것이다. 우리는 그것이 단지 가능하다는 것 뿐만아니라, 이문서에서 개발된 방법을 사용해서 라이브러들을 로드하고 심볼들을 해석하기위한 상대적인 간편성을 설명할 것이다. 이 방법은 ELF 포멧 과 /proc 파일 시스템을 지원하는 현대 유닉스 시스템 상에서 수행하기가 간편하고 이용되어 질 수 있다. 이 방법에 대한 수행은 3개의 대중적인 유닉스 변종 리눅스, 프리비에스디, 솔라리스 각각에 대해서 포함하고 있다.

Introduction:

최근에 공식적 부분과 개인적인 컴퓨터 연구자들 내부에서 웜(1)의 연구와 활동적인 개발이 다양하게 진행되어져 왔다. 바이러스는 웜에 매우 근접한 관계를 가지고 있는것으로 일박적이로 간주된다. 웜에대한 이러한 근접한 관련성과 관심의 증대에도 불구하고, 유닉스 바이러스들과 기생충들은 빈약한 연구 과제로 남아 있다. 유닉스 플렛폼 상에서 바이러스 기술에 대해 취급한 공식적인 출판물 또한 없다.

공식적인 연구와 문서가 부족함에도 불구하고, 유닉스 바이러스들은 전세계적으로 광범위하게 자주 출몰하고 있다: 최근에 Qualys 주식회사는 감염된 컴퓨터에 리모트 쉘 접근을 제공하는 바이러스를 발견했다(2). 확실히 바이러스에 관심을 갖는 언더그라운들이 늘어 나고 있다.

(1) 가끔 “자동 공격 대행자”이라고 불리어진다.
(2) https://www.qualys.com/form_remoteshell.html

바이러스는 기생충이라고 언급되어지는 일반적인 어플리케이션들의 한 분야이다. 이 문서의 목적에서, 기생충은 주 실행가능 파일안에 삽입되어지는 코드로서 정의되어진다. 기생충의 가장 일반적인 형태는 바이러스이다: 그러나 기생코드를위한 다양하고 잠재적인 사용들이 있다: 이름을 붙이자면, 바이너리 암호화, 압축풀기 그리고 저작권보호.

기생충은 유닉스 보안에서 개발되어지지 않은 영역이다. 개발자들에 있어서 여러가지 제한은 효과적이고 복잡한 기생충 개발을 막는다는 것이다. 라이브러리들을 로딩하고 심볼들을 해석하는 믿을만한 방법이 없다면, 기생충은 그들의 기능성에 심각한 제한을 받을 것이다. 파괴적 동적 링킹은 유닉스 기생충의 능력을 확실히 증대 시키고 그들의 이전 제한들로부터 많은 것들을 자유롭게하는 메커니즘을 제공한다.

현대 유닉스 플렛폼 상에서 동적 링킹은 동적 링커와 실행가능 파일사이에 상호 협조를 요구한다. 동적링커와 실행가능 파일은 라이브러들을 로딩하고 심볼들을 해석하는것과 관련된 작업을 위해서 중요한 공유성을 가지고 있다.

실행가능 파일은 동적 링커가 실행시에 어떤 라이브러리들을 로드할지, 어떤 심볼들을 해석할지들을 결정하기 위해서 복잡한 구조체들의 시리즈들을 제공한다. 이러한 구조체들에 새로운 요소들을 추가하는 것은 컴파일시에 링크 에디터가 실행가능 파일을 만든다음에는 지극히 어렵다. 이러한 어려움 때문에, 바이러스 작가들은 라이브러리 함수들을 사용하기위해서 조잡하고 신뢰성이 없는 방법으로 재정렬을 한다.

가장 자주 사용되어지는 기술은 개발 머신상에서 심볼을 해석하고 바이러스 안에 메모리 주소를 저장하는 것을 포함한다. 감염된 머신상에서 실행동안에, 동일한 심볼이 동일한 위치에 위치해 있기를 희망한다. 확실히 이것은 실행 환경시에 단지 조그만한 차이점에의해서도 쉽게 깨질수 있는 취약한 방법이다. 이방법은 주 실행파일에의해서 요구되어지지 않는 라이브러들을 사용 할 수 없다. 그래서, 프로세스 이미지의 한 부분으로서 로드되어질수 없다. 예를들면, 이러한 제한은 실행파일의 프로세스 이미지가 쓰레딩 라이브러리들을 포함하지않는 비 멀티 쓰레드 실행가능파일을 감염시키는 것으로부터 멀티 쓰레드 기생충을 방지한다.

외부 함수들에 접근하기위한 유일한 다른 기술은 주 실행가능 파일에 존재하는 동적 링킹 구조를 사용하는 것이다. 이 방법은 기생충이 주 파일에서 사용되어진 함수들과 라이브러들만을 사용하도록 제한한다. 이 메커니즘은 라이브러 함수들의 앞부분에 기생충을 끼워넣기위해서 가장 자주 사용되어진다.

하지만 기능성을 확보하기위해서는 거의 사용되어지지 않는다. 다른 실행가능 파일들의 라이브러리와 함수 요구에서 거대한 다양성때문에 주 실행가능 파일들의 거대한 집단을 뛰어넘는 믿을만한 방법은 아니다.

이제 유일하게 남아 있는 방법은 기생충 그자체에 요구되어지는 라이브러리의 사본을 제공하는 것이다. 이것은 크기가 주요 관심사인 기생충에는 극히 비최적화 해결책이다. 큰 기생충은 비밀성을 크게 감소시키고 그래서 발견의 기회가 증대되게 된다.

파괴적 동적 링킹 방법론은 기생 코드내부에 공유 라이브러 기능성을 사용하는 선택적 방법을 제공한다. 이 방법론은 기생충이 신뢰할만한 방법으로서 주 파일에서 외부 함수들을 접근하도록 할 수 있다. 개발자들은 기능성을 증대하기 위해서 라이브러들의 이점들을 취함으로서 복잡한 기생충을 만들기위한 충분한 자유로움을 마침내 갖게 된 것이다.

Background
다음의 각 섹션들은 디스크와 메모리상에서의 ELF를 소개할것이다. 뿐만이나라 ELF의 동적 링킹 메카니증의 개괄에 대해서 소개한다.

Introducing the ELF
The Executable and Linkable Format 이름에 제시하고 있는 것처럼 바이너리들에 대한 두가지 인터페이스를 제공한다: 실행가능 인터페이스, 링크가능 인터페이스. ELF 헤더는 인터페이스를 설명한다. 뿐만 아니라 바이너리에 관한 기본적인 정보를 설명하고 있다. 링커블 인터페이스는 실행을위해서 요구되어지지는 않는다: 그래서 이문서에서는 언급되어지지 않은 것이다. 실행가능 인터페이스는 프로그램 헤더 테이블에 저장되어진 프로그램 헤더에의해서 설명되어진다.

프로그램 헤더들은 프로세스 이미지를 만들기 위한 중요한 정보를 포함하고 있다. 동적 링킹 정보의 위치, 파일을 어떻게 메모리로 로드할것인지에 관한 것이다. 프로그램 헤더 테이블은 바이너리 파일안에 ELF 헤더 바로 다음에 위치해 있다. 각 프로그램 헤더는 세그먼트을 설명한다. 그것은 또한 파일의 시작으로부터의 옵셋과 크기를 가지는 불연속적인 집합이다. 프로그램 헤더안에 있는 타입 필드는 세그먼트가 어떻게 취급되어야 될지, 어떻게 메모리안으로 로드되어질지, 동적 디스크립터 테이블을 어떻게 해석할지등에 대한 묘사를 가지고 있다. 세그먼트들은 프로그램 text, data 그리고 프로그램의 실행시 요구들을 정의하는 정보를 가지고 있다.

세그먼트들은 로드되어 질수 있다. 각경우에 있어서 그것들은 그것들이 요구하는 메모리의 총양과 기대되어지는 퍼미션에 관한 사항도 묘사한다. 또한 파일에 관한 정보들도 포함할 수 있다. 이들 정보는 실행가능 파일이 어떤 프로그램 인터프리터를 사용할지, 세그먼트는 프로그램 헤더 그자체인지, 그리고 가장 중요한 동적 링킹 디스크립터 테이블의 위치들을 포함하고 있다. 동적 디스크립터 테이블은 ELF의 실행시 환경의 상세를 제공하는 간단한 구조체이다.

동적 구조체 각각은 그들 내용이 어떻게 해석되어질에관한 묘사를 하고있는 태그값을 가지고 있다. 내용은 포인터나 정수 값으로서 해석되어질수 있다. 동적 링킹에 대한 가장 중요한 태그들은 포인터들을 포함한다. 포인터들은 동적 심볼 테이블, 요구되어지는 다양한 다른 오브젝트들에대한 참조이다. 외부 함수들로 제어권을 변경하고 외부 변수들에 접근을 가능게 하는 이들 다른 오브젝트들은 “ELF 동적 링킹” 섹션에서 설명되어 질 것이다.

Process creation and runtime ELF layout

이번에는 디스크 상에 있는 ELF 바이너리가 메모리안의 실행 프로세스로 어떻게 변형되어지는지에 대해서 알아볼 것이다. ELF 실행가능 파일은 프로그램 로더에 의해서 프로세스 이미지로 변형되어진다. 프로세스 이미지를 만들기 위해서 프로그램 로더는 모든 로더블 세그먼트들을 mmap() 시스템 콜을 사용하여 요구되어지는 라이브러들의 로더블 세그먼트들과 함께 메모리안으로 맵한다. 프로세스 이미지를 만든 다음에 프로그램 로더는 실행의 제어를 주 ELF 오브젝트의 진입점으로 변경한다.

실행 가능 파일은 컴파일동안에 링크 에디터에의해서 선택되어진 고정된 주소(3)에 로드되어지기를 희망한다. ELF 실행가능 파일들은 캄파일시 링커가 로컬 text 와 data 오브젝트들을 재배치하도록 허락한 알려진 메모리 위치들에 맵되어진다. 실행가능 파일들은 올바르게 기능들을 수행하기 위해서 그들의 선택된 위치에 로드되어질 필요가 있다. 라이브러들은 프로세스 이미지안에있는 어떤 위치로 맵되어질수 있다: 그래서 공유 오브젝트들은 동적 링커가 마지막에 수정을 행하는 것을 허락하기 위해서 재배치 테이블들을 포함하고 있다. 추가로, 공유 오브젝트들은 위치 독립적인 코드(PIC, position independent code)를 포함하고 있다. PIC는 실행시 까지 알려지지 않은 외부 text 와 data 오브젝트들의 위치를 참조하기 위해서 ELF 이미지에 있는 로컬 구조체를 사용한다.

(3) 이 고정된 주소는 i386 바이너리를 위해서는 보통 0x08048000이다. 32비트 스파크 v8 바이너리에서는 0x010000, 64비트 스파크 v9 바이너리에서는 0x0100000000이다.

파일의 로더블 세그먼트들은 그들의 파일내에서의 크기값을 가지고 있을 뿐만 아니라 세그먼트가 차지하게될 메모리상의 크기 값을 가지고 있다. 실행시 크기는 가장 가까운 메모리 페이지로 반올림되어져야만한다. 대부분의 로더블 세그먼트들은 페이지 사이즈의 정확한 배수가 아니다. 그것들은 메모리상에 추가값이 추가되어질것이다. 추가되어지는 내용은 파일에서 주변에 있는 내용들이다. 이들 추가값들은 첫번째 메모리 세그먼트의 처음에 ELF 와 프로그램 헤더들을 보존한다. 이러한 보존은 메모리 이미지가 ELF 오브젝트로서 해석되어지도록 하는 것을 허락한다. ELF 오브젝트들의 집합으로서 프로세스 이미지를 조사하기위한 가능성은 전통적, 파괴적 동적 링킹을 가능하게 하는 것이다.

An introduction to ELF dynamic linking

프로세스 이미지를 성공적인 생성하고 실행하는 것은 간단한 메모리 개괄에대한 정보이상의 것을 요구한다. 컴파일시에 링크 에디터에 알려지지 않은 오브젝트의 절대주소들에 관한 참조 규정들이 요구되어진다. 이러한 규정들은 코드 오브젝트들(함수들) 과 데이터 오브젝트들(외부 변수들)이 프로세스 이미지 내부에서 ELF 메모리 맵들 사이에 참조되이질수 있도록 한다. 물론, 이러한 참조는 동적 링킹이다. 실행시 링크 에디터(rtld)는 동적 링킹 기능, 공유 오브젝트를 로딩, 심볼 참조를 해석하는 것을 제공한다. 인스톨되어 있는 ld.so 같은 동적 링커는 공유 오브젝트 그자체이거나 실행가능 파일 일수 있다.

실행 동안에 심볼 해석은 실행가능 파일, 라이브러리들, 동적 링커사이에 중요한 상호 작용을 동반하는 복잡하고 정교한 처리이다. 사용되어진 메커니즘은 우리의 목표 구조(아키텍쳐)들 각각에 독립적인 것이다. i386 과 스파크는 몇몇 일반적인 구조체들과 방법들을 공유한다. 이러한 공유 구조체에대한 설명이 아래에 있다.

다른 ELF 에 유용하게 만들어진 각 오브젝트는 심볼 테이블 안에 심볼 엔트리에의해서 묘사되어져 있다. 심볼 엔트리는 사실 심볼의 이름을 취급하고 심볼을 위한 값을 제공하는 심볼 구조체이다. 심볼 이름은 동적 문자열 테이블내부에 인덱스로서 저장되어져 있다. 심볼의 값은 ELF 오브젝트 내부에서 그 심볼의 주소이다. 이 주소는 보통 심볼의 절대 주소를 결정하기 위해서 오브젝트의 기본 로드 주소를 재배치하기 위해서 필요하다. 실행가능 파일들은 실행동안에 그것들의 로드 주소가 있는곳을 안다. 그래서 그것들 내부적인 심볼 참조는 컴파일시에 재배치되어진다.

전역 옵셋 테이블(GOT)은 오브젝트들에 대한 포인터들과 일반적인 데이터 오브젝트들을 포함하고 있는 ELF 이미지의 데이터 세그먼트 안에 위치해 있는 배열이다. 데이타 세그먼트들을 로딩하는 동안에 동적 링커는 자신이 가지고 있는 심볼 엔트리들을 위해서 GOT 엔트리들을 처리할 것이다. 컴파일 동안에 알려져 있지않는 변수의 위치를 접근하기 위해서, ELF는 로컬 GOT 내부에 포함되어져있는 포인터들을 역참조 할수 있다. GOT는 i386 동적 링킹에서 중요한 역할을 한다.

프로시져 링키지 테이블(PLT)은 제어권을 외부 프로시져들로 변경하는 코드 조각들을 포함하고 있는 구조체 엔트리들이다. PLT 와 그것의 코드 조각 엔트리들은 i386 구조상에서 다음 포멧을 가지고 있다.:

```
PLT0:
      push GOT[1] ;   word of identifying information
      jmp  GOT[2] ;   pointer to rtld function. 실행시 링크 에디터 함수에 대한 포인터
      nop
      ...
PLTn:
      jmp  GOT[x + n] ;   GOT offset of symbol address. 심볼 주소의 GOT 옵셋
      push n ;            relocation offset of symbol. 심볼의 재배치 옵셋
      jmp  PLT0 ;         call the rtld. 실행시 링크 에디터 호출

PLTn + 1
      jmp  GOT[x +n +1];  GOT offset of symbol address
      push n +1 ;         relocation offset of symbol
      jmp  PLT0 ;         call the rtld
```

실행가능 파일이 제어권을 외부 함수로 변경할 때, 컴파일시 링크 에디터에 의해서 그 심볼을 위해서 셋 업된 PLT 엔트리로 실행을 전달한다. 그 PLT 엔트리의 첫번째 명령은 GOT 내부에 저장되어진 포인터로 점프할것이다; 만일 심볼이 해석되어져 있지 않다면 GOT는 PLT 엔트리 안에있는 다음 명령의 주소를 포함하고 있을 것이다. 이 명령은 스택상에 재배치 테이블 안에 있는 옵셋을 푸쉬한다. 그리고 다음 명령은 실행을 PLT안에있는 제로 엔트리로 전달한다. 제로 엔트리는 실행시 링크 에디터인 심볼 해석 함수를 호출하는 코드를 포함하고 있다. 이것은 두번째 GOT 엔트리에 삽입되어진 동적 링크 에디터 함수의 주소를 사용해서 프로그램 로더에의해서 이루어진다.

동적 링커는 스택을 풀고 재배치 테이블 엔트리를 위치시키기에 필요한 정보들을 받을 것이다. 재배치 엔트리들은 PLT 엔트리가 어떤 심볼을 참조할지, 그 심볼의 주소가 프라이비트 메모리안 어디에 저장되어질지를 결정하기 위해서 사용되어진다. 만일 가능하다면, 심볼은 해석되어지고 해석되어진 주소는 PLT 엔트리에의해서 사용되어진 GOT 내부에 저장되어진다. 다음번에 심볼이 요청되어질 때, GOT 엔트리는 심볼의 주소를 포함할 것이다. 그래서 모든 다음번 요청들은 제어권을 GOT를 통해서 변경하게 될 것이다. 심볼이 바이너리에 의해서 첫번째 참조되어질때만 동적 링커는 심볼을 해석한다.; 이것은 게으른 로딩으로서 참조되어진다. 심볼 해석의 이 게으른 로딩 방법론은 모든 ELF 수행을위해서 디폴트이다.

심볼 테이블, 전역 옵셋 테이블, 프리시져 링키지 테이블, 문자열 테이블에 추가로, ELF 오브젝트들은 동적 링커가 쉽게 심볼들을 해석 할수 있도록하기

위한 해쉬 테이블과 체인을 포함하고 있다. 해쉬 테이블과 체인은 심볼 테이블 안에 있는 어떤 엔트리가 요구되어지는 심볼이름에 일치하는지를 빠르게 결정하기 위해서 사용되어진다. 해쉬 테이블은 동반하는 체인과 함께, 정수들의 배열로서 저장되어진다. 해쉬 테이블은 해쉬 테이블 내부에 버켓들의 개수, 체인 내부에서 요소들의 개수를 위해서 첫번째 두 위치를 예약한다. 해쉬 테이블 그자체는 요소들의 수자 와 요소들이 순서대로 위치해 있는 또다른 복사본이다.

동적 링킹 구조체들은 모든 동적으로 링크된 실행가능 파일들에게 동적 링커에대한 묵시적 접근을 제공한다: 그러나, 명시적 접근 역시 가능하다. 공유 오브젝트들을 로딩하고 심볼들을 해석하는 동적 링킹은 실행시에 링크 에디터가 함수들- dlopen(), dlsym() and dlclose()-를 사용하는 직접 접근을 통해서 수행되어질수 있다. 이들 함수들은 동적 링커 그 자체안에 포함되어져있다. 동적 링킹 라이브러리(libdl)은 이러한 함수들을 접근하기 위해서 실행가능 파일안에 링크되어질 필요가 있다. 이 라이브러는 컴파일시에 링크 에디터가 함수 참조들을 해석하는 것을 가능하게 하는 stub 함수를 포함하고있다.; 하지만 이 stub 함수는 간단하게 영을 리턴한다. 실제 기능은 동적 링커 안에 존재하기 때문에, 만일 정적으로 링크되어진 ELF 바이너리로부터 호출되어 진다면 공유 오브젝트 로딩은 실폐할 것이다.

동적 링킹을 수행하기위해서 요구되어지는 정보는 : 해쉬 테이블, 해쉬 테이블 요소의 수, 체인, 동적 스트링 테이블, 동적 심볼 테이블. 이 정보가 주어지면, 다음 알고리즘이 어떤 심볼의 주소를 제공한다:

```
1. hn = elf_hash(sym_name) % nbuckets;
2. for (ndx = hash[ hn ]; ndx; ndx = chain[ ndx ]) {
3.         symbol = sym_tab + ndx;
4.         if (strcmp(sym_name, str_tab + symbol->st_name) == 0)
5.                 return (load_addr + symbol->st_value);
   }
```

해쉬 수자는 ELF 규정(4)에 정의되어있는 elf_hash() 의 리턴값을 해쉬 테이블안에 있는 요소들의 수로 나눈 나머지 값 으로부터 계산되어진다.(라인 1). 이 수자는 해쉬 테이블 내부에대한 참조로서 사용되어진다. 또한 심볼의 이름들이 그 해쉬 값과 일치하는 체인의 첫번째 인덱스를 발견하기 위해서 사용되어진다(라인 2). 이 인덱스를 사용하여, 심볼은 심볼 테이블로부터 받아들여진다(라인 3). 요구되어지는 심볼 이름이 받아들여진 심볼의 이름과 비교되어진다(라인 4). 만일 일치한다면, 오브젝트의 로드 주소에 더해진

심볼의 위치는 요구된 심볼의 주소이다(라인 5). 하지만, 만일에 일치한 것이 없다면, 더 이상의 인덱스 값이 없을때까지 체인을 따라 갈것이다(라인 2). 심볼 타입(예를들면, 데이터 오브젝트, 코드 오브젝트) 뿐만 아니라 에러 점검등은 알아보기 쉽게 하기위해서 제거하였다. 이 알고리즘을 사용함으로서 임의의 심볼을 메모리상의 절대주소로 해석하는 것은 간단한 문제이다.

(4) 인텔 주식회사, "Tools Interface Standards: 이식가능한 포멧 규정 버전 1.1 권 1, ELF:Executable and Linkable Format" pg 2-19

Examining Processes via procfs

현대의 유닉스 시스템은 프로세스를 조사하기위해서 두가지 다른 방법을 제공한다(5). 프로세스 조사에 있어서 POSIX 표준을 따르는 방법은 세련되지 못하고, 매우 제한적이며, 프로세스의 메모리 이미지에 접근하는 ptrace() 시스템 호출이다. 보다 우수한 프로세스 조사 메커니즘은 일반적으로 procfs 또는 /proc로 불리어지는 proc 파일 시스템이다. 이 파일 시스템은 모든 현대적인 유닉스 시스템 상에서 가능하다.

proc 파일 시스템은 어떤 프로그램이 임의의 프로세스의 상태를 쉽게 조사할수 있도록 존재하는 파일 시스템 함수들(open(), read() 등등)을 통해서 프로세스에 대한 접근을 제공한다.

(5) 우리는 조사를 위해서 프로세스를 취급하는것으로부터 구별한다. 프로세스 조사, 프로세스 메모리 조사, 디버깅.

profs 디렉토리 /proc 아래에, 시스템상에 있는 각 프로세는 디렉토리를 가지고 있다. 각각의 디렉토리 이름은 프로세스의 프로세스 식별자(pid)이다. 추가로, 보통 거기에는 현제 프로세스의 디렉토리 엔트리에 대한 심볼릭 링크인 특별한 디렉토리, self,가 있다. 그래서 프로세스는 procfs 정보가 위치해 있는 /proc/self 를 사용해서 그 자신을 조사할 수 있다.

procfs 파일들을 위한 정확한 구조 및 포멧은 운영체제에 종속적이다; 그러나, 프로세스의 현재 상태(예, 실행중, 대기중, 좀비)을 설명 파일이다; 프로세스 이미지를 만들기위해서 사용되어진 바이너리와 일치하는 또다른 파일; 프로세스의 주소 공간에 접근을 부여받은 파일 그리고, 파괴적 링킹에서 가장 중요한, 프로세스 이미지의 메모리 맵을 설명하는 파일.

Subversive Dynamic Linking Theory

파괴적 동적 링킹은 라이브러리들을 로딩하는것에 기초하는 것이 아니라 이 함수를 수행하는 존재하는 프로시져를 위치시키는것에 기초한다. 동적 링크 에디터는 라이브러리 로딩을 제공하는 함수들을 포함해야만한다. 이들 함수들을 위치시키고 사용하는데 대한 간단한 문제이다. 방법론은 다음과 같다:

1) 라이브러리 로딩 함수들을 제공하는 ELF 오브젝트를 위치 시킨다.
2) 공유 오브젝트들을 로드하고 언로드하는 함수들을 위치시킨다.
    a) (옵션) 심볼들을 해석하는 함수를 위치시킨다.
3) 공유 오브젝트 로딩, 언로딩, 심볼 해석을 제공한다.

첫번째 단계는 가장 복잡하고 어렵다. 기생충은 그것의 주 파일의 프로세스 이미지를 조사해야만한다. 그리고 어떤 메모리 맵이 요구된 오브젝트와 일치하는지 발견해야만한다. 이것은 proc 파일 시스템의 도움과 ELF에 대한 상세한 이해로서 가능하다(6). prof “maps” 파일에 대한 분석은 단지 몇 개의 도움 함수들로서 쉽게 수행되어진다. 도전은 프로세스 이미지를 구성하는 많은 메모리 맵들의 어떤 것이 ELF 오브젝트에 올바른 것인지를 결정하는 것이다. 어떤 메모리 맵이 실행시에 동적 링커인가를 결정하는 메커니즘은 수행되어지고 있는 것의 내부에서 유일한 것이다. 그래서 그것을 결정하는 것은 적당한 섹션에서 설명되어질 것이다.

두번째 단계는, 공유 오브젝트 로딩 그리고 언로딩 함수들을 위치 시키는, ELF 오브젝트 내부에있는 절대주소안에 심볼을 해석하는 dlopen() 그리고 dlclose()을 요구하는 것이다. 동적 링킹 정보(소위, 해쉬 테이블, 심볼 테이블, 스트링 테이블)에대한 접근이 제공되어진 ELF 오브젝트 내부에있는 심볼을 해석하는 것은 매우 간단하다. 이 정보는 메모리 맵의 기본 주소에 위치해 있는 ELF 헤더를 사용해서 쉽게 발견되어질수 있는 동적 세그먼트로부터 쉽게 추출되어질 수 있다. 그래서, 필요한 심볼들에 해석은 타켓 오브젝트의 기본 로드 주소만을 요구한다.

(6) procfs 없이도 가능하지만, 이 프로세스 조사에대한 보조역할로서는 매우 신뢰할만하다.

또한 일반적으로 라이브러리 로딩을 취급하는 동일한 오브젝트가 그러한 라이브러들안에 있는 심볼들을 해석한다. 이것은 기생충이 다른 동적 링킹 함수들처럼 동일한 메모리 맵안에있는 심볼을 dlsym() 함수를 통해서 해석할수 있도록 허락하는 전형적인 경우이다. 라이브러리를 로드한 오브젝트가 심볼들을 해석할 수 없을 때, 그들에게는 두가지 선택이 존재한다: 또다른 ELF

오브젝트 내부에 dlsym() 함수를 위치시키거나 기생충 코드안에 심볼 해석 기능을 제공하는 것이다. 파괴적 링킹 수행은 두번째 옵션을 이용한다. 왜냐하면 심볼들을 해석하기위한 코드는 최초 라이브러리 로딩 함수들을 위치시키기 위해서 미리 존재해야만 하기 때문이다.

세번째, 그래고 마지막 단계는 첫번째와 두번째 단계에서 확득한 정보 및 실행시 추가적으로 확득한 정보를 처리하는 것을 수반한다. 이들 데이터 처리는 다양한 방법으로 수행되어질수 있다. 파괴적 링킹 방법론의 현재 수행을위한 메커니즘은 힙상에 저장되어진 링키드 리스트의 노드이다. 힙상에 데이터를 저장하는 것은 기생충의 실행주기동안에 존속성 및 동적 메모리 처리를 허락한다.

파괴적 링킹의 근본적인 목적은 라이브러리 로딩 함수들에대한 접근과 심볼 해석에대한 접근을 제공하는 것이다. 이것은 dlopen() 그리고 dlsym() 프로시져에 대한 함수 포인터들을 통해서 수행되어진다. 이들 포인터들 과 힙 처리함수들 malloc() 그리고 free()에 대한 포인터들은 구조체 안에 저장되어진다. 이 구조체에 있어서 분명하지 않는 포인터는 그것이 각 파괴적 동적 링킹 함수 호출을 전달하도록 기생충에의해서 취급되어질수 있다. dlopen() 함수에의해서 리턴되어지는 포인터들은 최초의 처리 구조체에 덧붙여짐으로서 링키드 리스트 안에 저장되어 질수 있다. 뿐만 아니라 dldym()에 대한 나중의 호출들을 위해서 기생충에 리턴되어질 수 있다. 쓰레기값 집합은 링키드 리스트를 여행하는 간단한 문제이다. 그리고 각 로드되어진 라이브러리를 dlclose()ing 한다. 그리고 리스트 노드를 free()ing 한다. 그래서, 주 파일의 프로세스 이미지는 쉽고 간한단 방법으로 그것의 본래의 최초의 상태로 되돌려 질수 있다.

Implementation details

설명되어진 이론을 가지고 이제 우리는 파괴적 동적 링킹을 연습해볼 순서이다. 다음 섹션들은 3가지 유닉스 플랫폼(리눅스, 프리비에스디, 솔라리스) 상에서 파괴적 동적 링킹 방법론을 수행하는 특성들을 조사할 것이다.

Linux

방법론의 첫번째 단계는 어떤 ELF 오브젝트가 공유 오브젝트 로딩 기능을 제공하는지를 알아보는 것을 요구한다. 리눅스하에서, 이 기능을 제공한는 오브젝트는 GNU C 라이브러리(glibc)이다. libdl 내부에 포함되어져있는 실질적인 함수 dlopen()은 glibc 함수 _dl_open()에대한 사실상의 랩퍼이다. 그래서 파괴적 동적 링킹의 첫번째 단계로서 위치되어질 필요가 있는 오브젝트는 glibc 이다.

glibc의 텍스트 세그먼트에 일치하는 메모리 맵을 위치시키는 것은 주 파일의 메모리 이미지를 조사하는 것을 수반한다. /proc/self/maps 는 프로세스의 메모리 맵에대한 접근을 제공한다. 이파일은 ASCII 문자열로 구성되어있다. 다음 포멧을 가지고 있다:

```
/* addr range      prot  offset    dev   inode    path name */
4001b000-400ff00  r-xp  00000000  03:01 390597  /lib/libc-2.1.3.so
```

첫번째 필드는 메모리 맵의 상위 제한값이 따라오는 오브젝트의 기본 로드 주소이다. 다음 그 맵에대한 보호(protection)를 설명하는 필드이다. read, write, execute 그리고 private(copy on write). 각각은 r, w, x, p 로 표시되어진다. 다음의 3, 4, 5번째 3 필드는 파괴적 동적 링킹에서는 의미를 가지 않는 옵셋, 디바이스 메이저 와 마이너 번호, 파일 시스템 아이노드 번호이다. 마지막 엔트리는 가장 관심있는 것인데, 메모리 상으로 맵되어진 오브젝트를 위한 소스 파일의 풀 패스 이름이다.

오브젝트의 패스 이름에 포함되어진 것은 glibc가 단지 strcmp()를 사용해서 위치되어진 것을 허락한다. 그래서, 라이브러리를 위치시키는것에 대한 메커니즘은 간단한 문자열 추출 과 문자열 탐색 알고리즘이다.

```
1. for (i = 0; i < nread; i++) {
2.         start = end = buffer + i;
3.         while ((*end++ != '\n') && (*end) && (i++ < nread))
                   ;
4.         *end = 0;
5.         for (ptr = end; (ptr > start) && (*ptr != ' '); ptr--)
6.                 if ((*ptr == *lib_name)) &&
7.                         (strncmp(ptr, lib_name, strlen(lib_name)) == 0))
8.                                 return ((void *)strtol(start, NULL, 16));
    }
```

buffer 는 /proc/self/maps 파일의 내용에서 read()에 의해서 채워져있는 문자 배열이다. 읽혀질 바이트들의 수는 nread 에 저장되어진다. buffer 는 우리가 읽고자하는 바이트를 벗어날때까지 반복되어진다(라인 1). 문자열의 시작에 대한 서있는 포인터 그리고 문자열의 끝을 위치시키위해서 사용되어진 걸가가는 포인터는 buffer 내부에있는 현재 위치로 초기화되어진다(라인 2).

이 알고리즘은 문자열을 탐색한다; 그래서, 문자열은 바이트들의 임의의 계속으로서 추출되어져야한다. 알고리즘의 이러한 목적을 위해서, 문자열은 새로운 라인 문자('\n') 또는 ASCII NUL('\0')에 의해서 종료되어지는 바이트들의 연속으로 정의되어진다. 바이트들로 구성되어진 임의의 스트림으로부터 문자열을 추출하는 것은 문자열 문자의 끝을 탐색함으로서 가능하도록 만들어졌다. 뿐만아니라 buffer의 끝을 위한 에러 검사도 수행한다(라인 3). 문자열은 아래에서 사용되어지는 strcmp() 의 속도를 증가 시키기 위해서 널로 종료되어진다(라인 4)

추출되어진 다음에, 문자열은 요구된 오브젝트의 이름을 위해 탐색되어진다. 탐색은 추출된 문자열의 끝부분에대한 포인터 와 뒤쪽부터 걸어오는 포인터에 의해서 수행되어진다. (' ')로 구분되어진 첫번째 단어가 발견되어질때까지 또는 문자열의 시작에 도달할때까지 계속되어진다(라인 5). 함수 호출에 대한 부하를 초래하기전에, 최적화 방법으로서 각 작업에서 첫번째 문자가 먼저 비교되어진다. 만일 이것들이 일치한다면 비로서 strncmp() 쓰여질 것이다(라인 6). 현재 포인터는 요구되어진 이름과 비교되어진다. 만일 문자열이 일치한다면 문자열 탐색은 끝난다(라인 7). 만일 문자열이 요구되어진 라이브러리 이름과 일치한다면, 문자열의 시작은 나중에 조작과 해석을 위한 포인터로 변경되어질 필요가 있는 로드 주소의 ASCII 16진수 표현이다. 이 변환은 포인터에대한 함수 strtol()의 리턴값을 취함으로서 실행되어진다(7)(라인 8).

일반적으로 심볼릭 참조를 절대 메모리 위치로 변환하기위해서 사용되어지고, 리눅스에게 유일한 어떤것인, dlsym() 함수는 libdl 안에 실질적으로 포함되어져있다. GNU C 라이브러를 위치시키고 가로채기위해서 사용되어진 같은 함수는 프로세스 이미지 안으로 로드된 어떤 공유 오브젝트를 위치시키기 위해서 재사용되어 질 수 있다.

(7) C 언어에서 longs 과 pointers 는 32비트(ILP32) 와 64비트(LP64) 구조상에서 동일한 크기이다.

FreeBSD
동적 링크 에디터의 프리비에스디 수행은 자신이 소유한 ELF 오브젝트 로딩 과 자신이 소유한 심볼 해석 기능을 제공한다. 프리비에스디 상에서 방법론의 첫번째 단계를 만족시키는 것은 ELF 오브젝트가 함수들 dlopen(), dlclose(), dlsym()을 포함하고 있는 것 처럼 동적 링커를 위치시키는 것을 요구한다. 런타임 링커의 로드 주소를 위치시키는 것은 이러한 심볼들 해석을 향하는 첫번째 단계이다.

프로세스의 메모리 맵을 철저하게 조사하는 것은 profs 으로부터 검색된 정보로부터 쉽게 수행되어질수 있다. 프리비에스디 procfs 은 디록토리 /proc/curproc 를 통해서 그 자신에관한 정보를 접근하도록 프로세스에게 제공한다. 프로세스 내부에 있는 메모리 맵에 대한 정보를 제공한는 파일은 map 이다. 그래서 기생충은 /proc/cour/proc/map 을 통해서 주 파일의 메모리 맵 정보를 접근할 수 있다. 정보는 다음 포멧을 가지는 아스키 문자열로 저장되여져있다.

```
/* start    end          real       prot       priv  type */
0x804800 0x805500 13 15 0xc6e18960 r-x 21 0x0 COW NC vnode
```

이 문자열에서 첫벌째와 두번째 필드는 메모리 범위의 시작과 끝 주소이다. 나머지 엔트리들에서 기생충이 흥미를 가질만한 추가적은 정보는 세그먼트의 보호에 관한 표현이다. 하지만, 이것은 이문서에서 제안되어진 파괴적 동적 링킹 방법론에 관심사항이 아니다.

동적 링커에 어떤 메모리 맵이 일치하는가를 결정하기위해서 사용되어진 메커니즘은 전역 옵셋 테이블(GOT)에 대한 접근을 요구한다. GOT 의 위치는 동적 링킹 구조체안에 포함되어져 있다. DT_PLTGOT 태그를 가지는 구조체는 동적 링커가 전통적인 ELF 동적 링크을 수행하는 것을 허락하는 GOT 의 주소를 부여한다.

GOT 배열안에 있는 제로 엔트리는 동적 링킹 구조체의 주소를 가지도록 예약되어져 있다; 첫번째와 두번째 엔트리는 i386 상에서 예약되어져 있다. i386 ELF 규정 부록은 첫번째 엔트리 GOT[1]을 “정보를 식별에대한 워드”로서 정의한다. 이것이 정보를 식별하는 것은 ELF 오브젝트의 메모리 맵을 설명하는 동적 링커의 개별적인 구조체에대한 포인터이다. 이정보는 많은 방법들로서 기생충에게 잠재적인 유용성을 가지고 있다. 하지만 이 파괴적 링킹 수행에대해서는 아니다. 두번째 엔트리 GOR[2] 는 동적 링크 에디터 내부에있는 함수에대한 포인터이다. 이 함수는 실행시 링커의 심볼 해석

프로시져안으로서 진입점을 제공한다. 이 포인터는 프로세스 이미지의 많은 메모리 맵 사이에서 동적 링커가 위치하기위한 열쇄이다. 포인터가 어떤 메모리 맵 범위를 참조하여 위치하는가는 동적 링커의 로드 주소를 결정하는 것을 가능하게 한다.

```
1. rtld_func = (char *)((int *)got)[2];
2. for (i = 0, ptr = buffer; (i < nread) && (*ptr); i++, ptr++) {
3.         start = buffer + i;
4.         load_base = (char *)strtol(start, &end, 16);
5.         if (rtld_func < load_base) {
6.                 while ((*ptr++ != ' \ n') && (*ptr) && (i++ < nread))
                           ;
                   continue;
           }
7. load_high = (char *)strtol(++end, NULL, 16);
8. if (rtld_func < load_high)
9.         return (load_base);
   }
```

런타임시 링크 에디터에대한 포인터는 전역 옵솟 테이블안에 있는 두번째 엔트리로부터 추출되어진다(라인1). 문자 배열 buffer는 /proc/curproc/maps 의 내용을 가지고 채워져있다. 이들 내용은 문자열로서 취급되어진다: 뉴라인이나 널로 종료되어지는 임의 길이를 가지는 바이트 배열이다. 이런한 문자열의 각각은 메모리 위치로 변환되어지기 위해서 그리고 이 메모리 범위에 비교되어지는 rtld 포인터를 위해서 필요하다. 문자열은 조건을 만족하는 동안에 반복되어진다(라인2). 포인터 start는 buffer 안에 있는 현자 문자열 위치로 초기화 되어진다. 문자열의 출발은 strtol() 함수에 의해서 포인터로 변환되어진다(라인4). 만일 동적 링킹 함수의 위치가 현재 맵의 로드 주소보다 낮다면(라인5), 다음 문자열은 축출되어지고 루프를 continue 한다(라인6). 그렇지 않다면, 메모리 맵의 최상위 위치가검색되어진다(라인7). 만일 포인터가 맵의 최대 범위보다 낮다면(라인8), 포인터는 메모리 맵 안에서 실폐한다. 그리고 기본 로드 주소가 리턴되어진다(라인9)

동적 링커의 로드 주소를 위치 시킨후에, 필수 함수들을 해석하고 실행시 데이터를 위에서 제안한 방법으로 취급하는 것은 미약한 문제일 뿐이다.

Solaris

솔라리스에서 실행시 링크 에디터는 자신이 소유하고 있는 dlopen(), dlsym(), dlclose() 를 제공한다. 그래서 동적 링커는 위치되어질 필요가 있고 파괴적 링킹 방법론의 첫번째 단계를 만족시키기위해서 분석되어질 필요가 있는 오브젝트이다.

솔라리스 profs 파일 /proc/self/map 은 prmap_t 구조체에대한 임의의 수자로 구성되어있다. 이 구조체는, procfs.h 에서 정의되어졌고, 각각은 메모리 맵을 설명하고, 다를 것들 사이에 있는 메모리 맵의 사직 주소와 크기를 포함한다. 프리비에스디 하에서 채용되어진 비슷한 방법이 동적 링커에 어떤 메모리 맵이 일치되어지는지를 결정하기위해서 사용되어야만한다. 실행시 링크 에디터 함수에대한 포인터가 배치되어진다. 그리고 메모리 맵은 올바른 주소 범위를 탐색한다.

스파크 상에서 프로시져 링키지 테이블(PLT)는 개별적인 메모리상에 저장되어진다. 링크 에디터는 직접적으로 그것을 수정한다. GOT 는 심볼 해석에 관여하지 않는다. 처음 4 PLT 엔트리들은 동적 링커 에디터를 위해서 예약되어져 있다. 제로 엔트리는 실행시 링커의 심볼 해석 루틴을 호출하기위해서 작은 stub 함수를 포함하고 있다. 첫번째 엔트리는 “정보 식별에대한 워디”를 포함하고 있다. 두번째와 세번째 엔트리는 64비트 프로그램들을 위해서 사용되어진다. 하지만 채용되어진 논리는 동일한 의미로서 남아있다. Stub 함수는 레지스터 윈도우를 생성하고 동적 링커 프로시져를 호출한다. 이 프로시져는 링크 에디터의 실행시 심볼 참조 함수안오르의 진입점이다.

PLT 안에 있는 각각의 엔트리는 세 스파크 연산자를 허용하도록 12 바이트 길이를 가지고 있다. 스파크 연산자의 포멧은 약간의 설명을 필요한다. 그래서 동적 링커의 배치에 집중한다. 모든 스프크 연산자들은 4바이트 길이를 가지고있고 4가 가능한 타입중에 하나이다; 불리어지는 포멧들은, 바꾸어말하면, format1, format 2 등이다. 이러한 포멧들은 연산자의 첫번째 두 비트들로 구별되어진다. 제어권을 메모리안의 다른 곳으로 변경하는 콜 연산자는 단지 format 1 명령이다.

스파크가 제어 명령을 변경하는 것은, 예를들면 가지치기들 과 호출들, 프로그램 카운터의 현재위치 또는 명령 포인터의 현재위치에 관련이 있다. 이것은 코드가 재배치 수정없이 메모리안에서 쉽게 재배치되어 질수 있도록한다. 호출 연산자는 부호 32비트 변위를 프로그램 카운터에 더한다. 그래서 제어를 변경한다. 모든 스파크상에서 제어를 변경하는 작동은 4바이트

경계로 경계 맞추어진다(얼라인된다). 모든 연산자가 4바이트 길이를 갖기 때문에 변위의 하위 2비트는 영일 것이다. 그래서, 32비트 변위는 30비트 수자로서 저장되어진다, 최상위 2비트가 연산자 타일을 지시하기위해서 복구되어질수 있도록 허락한다.

포인터를 복구하는 것은 실제로 제어권을 동적 링커로 변경하는 호출 연산자로부터 정보를 추출함으로서 가능하다. 프로지져 링키지 테이블의 영번째 엔트리(PLT0)안에 있는 두번째 명령(1 인덱스, 예, 4바이트 번째)은 상대적 호출 명령이다. 이 포인터를 축출하는 것은 왼쪽으로 두번을 쉬프트 하는것으로 수행되어질수 있는 32비트 옵셋을 재선언하는 것을 요구한다; 이것은 최상위 2비트에 저장되어있는 연산자 타입 정보를 줄이도록한다. 옵셋은 일반적인 상황하에서 프로그램 카운터의 값을 가지는 호출명령의 주소(PLT0 + 4)에 더해질수 있다. 또한 동적 링커의 심볼 해석 함수에대한 절대 메모리 위치를 배치하기위해서 더해질수 있다. 이 절대 메모리 위치는 어떤 메모리 맵 주소 범위에서 실폐하는지 검사되어질수 있다. 그래서 동적링커의 기본 로드 주소는 결정되어질 수 있다.

```
1. rtld_func = (char *) &plt[1];
2. rtld_func += (char *) (plt[1] << 2);
3. for (prmap = buf; prmap < buf + sizeof buf; prmap++)
4.   if ((rtld_func > (char *)prmap->pr_vaddr) &&
5.             (rtld_func < (char *)(prmap->pr_vaddr + prmap->pr_size)))
6.                       return (prmap->pr_vaddr);
```

영 엔트리안에 있는 호출명령의 주소는 의사 프로그램 카운터처럼 행동하는 rtld_func 안에 저장되어져 있다(라인1). 호출 명령의 32비트 변위는 축출되어지고 동적 링크 함수의 절대 메모리 주소를 배치하기 위해서 의ㅣ사 프로그램 카운터에 더한다.(라인2) 바이트 배열 buf 는 prmap_t 구조체의 배열인 /proc/self/map 의 내용으로 채워져있다. 이 구조체 배열은 조건을 만족하는 동안 반복되어진다(라인3). 각 구조체의 주소 범위는 만일 그것이 rtld 함수의 주소를 포함하고 있는지를 알아보기위해서 검사되어진다. 첫번째, 만일 rtld 함수가 기본 주소부다 아래에 있다면, 메모리 맵의 기본 주소가 조사되어진다. 그리고 나서 다음 구조체가 조사되어진다(라인4). 만일 함수의 위치가 기본 주소보다더 높고 가장높은 주소보다도 낮다면(라인5), 맵의 기본 주소는 동적 링커의 로드 주소이다. 이 로드 주소가 리턴되어진다(라인6).

동적 링커의 로드 주소를 위치시킨후에, 필수 함수들 해석하고 실행시 데이터를 위에서 제안한 방법으로 취급하는 것은 심히 미약한 문제일 따름이다.

Conclusion

우리는 기생충 코드를 위한 동적 링킹의 중요성을 설명했다. 기생충의 기능성을 엄청나게 강화할수 있는 라이브러를 사용하는 믿을만한 어떠한 메커니즘이 있는지도 설명했다. 제공된 방법론은 이미 로드된 동적 링커내부에 있는 dlopen() 함수의 메모리 주소를 결정하기 위해서 플랫폼에 특성을 가지는 알고리즘을 사용하는 것을 수반한다. 리눅스, 프리비에스디, 솔라리스에서 예제의 수행을 살펴봄으로서 이 방법이 합벅적이고, 이식가능하다는것을 알아보았다. 그래서, 특정된 시스템상에서 기생충을 취급하는 것은 영원히 끝났다.

Appendices

Appendix A: ELF Headers

The format of an ELF header is as follows:

```
#define EI_NIDENT (16)
typedef struct {
    unsigned char e_ident[EI_NIDENT]; /* Magic number and other info */
    Elf32_Half    e_type;         /* Object file type */
    Elf32_Half    e_machine;      /* Architecture */
    Elf32_Word    e_version;      /* Object file version */
    Elf32_Addr    e_entry;        /* Entry point virtual address */
    Elf32_Off     e_phoff;        /* Program header table file offset */
    Elf32_Off     e_shoff;        /* Section header table file offset */

    Elf32_Word    e_flags;        /* Processor-specific flags */
    Elf32_Half    e_ehsize;       /* ELF header size in bytes */
    Elf32_Half    e_phentsize;    /* Program header table entry size */
    Elf32_Half    e_phnum;        /* Program header table entry count */
    Elf32_Half    e_shentsize;    /* Section header table entry size */
    Elf32_Half    e_shnum;        /* Section header table entry count */
    Elf32_Half    e_shstrndx;     /* Section header string table index */
} Elf32_Ehdr;
```

The format of an ELF program header is as follows:

```
typedef struct {
    Elf32_Word    p_type;         /* Segment type */
    Elf32_Off     p_offset;       /* Segment file offset */
    Elf32_Addr    p_vaddr;        /* Segment virtual address */
    Elf32_Addr    p_paddr;        /* Segment physical address */
    Elf32_Word    p_filesz;       /* Segment size in file */
    Elf32_Word    p_memsz;        /* Segment size in memory */
    Elf32_Word    p_flags;        /* Segment flags */
    Elf32_Word    p_align;        /* Segment alignment */
} Elf32_Phdr;
```

The format of an ELF dynamic linking structure is as follows:

```
typedef struct {
    Elf32_Sword   d_tag;                /* Dynamic entry type */

    union {
        Elf32_Word        d_val;  /* Integer value */
        Elf32_Addr        d_ptr;  /* Address value */
    } d_un;
} Elf32_Dyn;
```

Appendix B: Generic ELF image parser

```
ehdr = load_addr;
phdr = load_addr + ehdr->e_phoff

for (i = 0; i < ehdr->e_phnum; i++, phdr++)
    if (phdr->p_type == PT_DYNAMIC)
            break;

dyn = load_addr + phdr->p_vaddr;

for (; dyn->d_tag; dyn++) {
    switch(dyn->d_tag) {
    case DT_STRTAB:
            str_tab = load_addr + dyn->d_ptr;
            break;
    case DT_SYMTAB:
            sym_tab = load_addr + dyn->d_ptr;
            break;
    case DT_HASH:
            {
                    Elf32_Word *p;

                    p = load_addr + dyn->d_ptr;

                    nbuckets = *p++;
                    nchains = *p++;
                    hash = p;
                    chain = p + nbuckets;
            }
            break;
```

```
        default:
                break;
        }
    }
```

Appendix C: Generic dynamic linker locator

```
    ehdr = BASE_ADDR;
    phdr = ehdr + ehdr->e_phoff;
    for (i = 0; i < ehdr->e_phnum; i++, phdr++)
        if (phdr->p_type == PT_DYNAMIC)
                break;
    dyn = phdr->p_vaddr;

    for (; dyn->d_tag; dyn++)
        if (dyn->d_tag == DT_PLTGOT)
                pltgot = dyn->d_ptr;
    load_addr = locate_rtld(pltgot);
```

Bibliography

1. TIS Committee. “Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2”, TIS Committee, 1995.
2. Paul, Richard P "SPARC Architecure, Assembly Language Programming, and C". Prentice Hall, Upper Saddle River, NJ, 2000

Acknowledgements

To the following people I owe a great debt:
That awesome bastard mammon_ for wasting years of his life at university so I wouldn’t have to; Doc Marvel, for encouraging me to waste years of my life at university (but not giving me the money for it); Silvio Cesare, for ELF conversations that would bore normal men to death; Grendel, PhD, for spending his life at university; _dose, for no real reason; and last, but not least, Knotty Dread for living the university life without me (bastard).