루트킷 탐지 도구(Gibraltar) 성능 향상을 위한 자동화된 커널 메모리 자료 구조 추출에 관한 연구

최원하*, 이하윤*, 조영필*, 백윤흥*
*서울대학교 전기정보공학부 및 반도체공동연구소
e-mail : whchoi@sor.snu.ac.kr

A Design and study on automatic extraction of kernel data structure to improve performance of rootkit detection tool, Gibraltar.

Wonha Choi *, Hayoon Yi*, Yeongpil Cho*, Yunheung Paek*
*Dept. of Electrical and Computer Engineering and Inter-University Semiconductor Research Center (ISRC), Seoul National University

요 약

하이퍼바이저를 이용한 가상화 검사(Virtual Machine Introspection)의 하나인 Gibraltar[2]는 자동으로 무결성 명세서를 생성할 수 있고, 보안 위협이 높아지고 있는 데이터 영역에 대해서도 방어가 가능하다는 점에 존재하는 어떤 보안 도구보다 효과적인 시스템으로 여겨지고 있다. 본 연구에서는 루트킷 탐지 도구인 Gibraltar 를 Linux/ARM 3.14 버전에서 구현하고, 커널 메모리 자료 구조 추출 자동화 툴을 개발함으로써 기존 연구의 문제점을 해결하여 성능을 개선하였다. 이를 바탕으로 향후 Gibraltar 연구의 추가 개선 방향을 제시한다.

1. 서론

지난 몇 년 동안, 컴퓨터 보안 분야의 발전에도 불구하고, 악성 소프트웨어(루트킷)는 여전히 증가하였을 뿐만 아니라 탐지를 어렵게 하는 등 기술적으로도 지속적인 진화를 하고 있다.

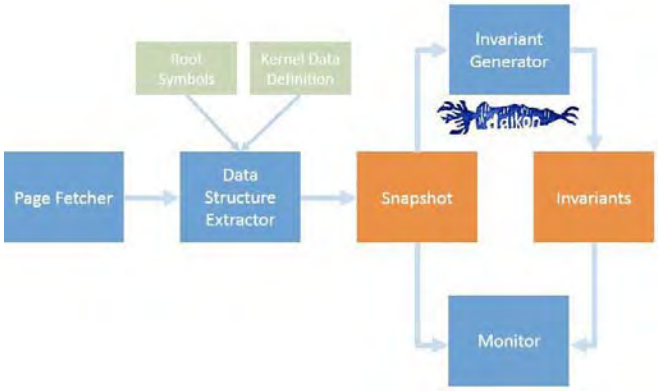
초기의 루트킷은 코드 등의 정적인 영역을 공격 대상으로 하였으나, 점차 데이터 영역으로 그 범위를 넓히고 있다. 일반적인 데이터는 그 종류와 형태가 다양할 뿐만 아니라, 데이터의 내용이 변경될 수 있기 때문에 정형화된 방법으로는 방어가 매우 어렵다는 특징을 가진다. 이러한 문제점을 처음으로 인식한 Petroni[1]는 수많은 데이터 영역의 자료구조 중에서 운영체제의 동작 중에 절대로 변경되어서는 안 되는 특성들을 모아 무결성 명세서(Integrity Specification)로 작성하였고, 이 규칙들이 위반 되는 지 여부를 통해 악성 행위 여부를 판별하는 새로운 방어 기법을 제안하였다. 이러한 접근 방식을 통해 까다로웠던 데이터 영역에 대한 방어가 부분적으로 가능해졌으나, 무결성 명세서가 반드시 운영체제와 시스템을 잘 이해하는 전문가들에 의해서 수동으로 작성 되어야 한다는 치명적인 약점이 있었다. 운영체제가 변경되면 명세서는 새로운 기준으로 다시 갱신되어야만 했고, 운영체제 내부에 있는 수많은 데이터 중 일부 만을 대상으로 명세서를 작성하였기 때문에 예측 가능하지 않은 공격을 막는 것은 불가능했다.

이후 Baliga[2] 등은 무결성 명세서를 머신 러닝을 통해 자동적으로 추출할 수 있는 툴인 Gibraltar[2]를 제안하였다. 이 연구는 기존의 Petroni[1] 연구나 최근까지 이루어진 많은 연구와 달리, 무결성 명세서를 특정 조건에 구애 받지 않고 자동으로 생성한다. 뿐만 아니라, 상대적으로 방어가 쉬운 정적 영역뿐만 아니라 동적인 데이터 영역까지 모두 검사 대상으로 할 수 있다는 점에서, 현재까지 소개된 악성 코드 탐지 기법 중 가장 효과적이고 유연한 시스템이라고 평가 할 수 있다.

하지만, Gibraltar 의 훌륭한 접근 방식에도 불구하고 실제로 루트킷 탐지 툴로 사용하기에는 몇 가지 문제점을 가지고 있는데, 자동화 툴이라는 것이 가장큰 연구의 의의임에도 불구하고 피검사 대상 리눅스 커널 메모리로부터 자료구조를 추출하는 단계에서는 사람의 개입을 필요로 하는 모순점이 그 중 하나이다. 이런 문제점을 해결하기 위하여 본 연구에서는 기존 Linux 2.4.20 버전에서 구현되었던 Gibraltar 를 최신 Linux/ARM 3.14 버전에서 리눅스의 기본 하이퍼바이저인 KVM 을 이용하여 직접 구현하였고, 소스코드의 정적 분석을 통해 자료 구조 추출을 자동화 함으로써 성능을 개선하였다. 또한 Gibraltar 연구 이후에 진행된 데이터 자료 구조 추출에 관한 논문들을 분석하여 Gibraltar 의 개선 방향을 제안하고자 한다.

2. 기존 연구 및 문제점

리눅스의 모든 커널 모듈은 같은 권한 수준으로 실행되는 특성 때문에 커널과 동일한 권한을 가진 루트킷은 커널 내부의 보안 기능을 회피할 수 있다. 이를 해결하기 위해선 물리적으로 분리되어있는 보안 하드웨어를 사용하거나, 더 높은 권한을 가지고 있는 하이퍼바이저에서 보안 기능을 수행하여야 한다. 이 중 가상화 장치 상의 게스트 운영체제를 검사하여 악성 코드 등의 이상 유무를 판단하는 일련의 동작을 가상화 검사(Virtual Machine Introspection)라고 하며 T. Garfinkel[3] 등에 의해 처음 제안된 이후로 수많은 연구가 진행되어왔다. Gibraltar 역시 초기 모델에서는 외부의 PCI 카드 상에 구현되었으나, 이후 가상화 기술을 이용하도록 변경되었다.



(그림 1) Gibraltar 동작도

Gibraltar 는 (그림 1) 에 표시된 것과 같이 크게 4 부분으로 이루어 진다. Page Fetcher 에서는 게스트 운영체제의 물리 메모리 페이지를 읽고, Data Structure Extractor 에서는 페이지로부터 자료 구조의 값과 특성 등의 정보를 추출한다. 이 과정에서 악성 코드 등에 이미 감염 되었을 지도 모르는 게스트 운영체제의 도움 없이 하이퍼바이저에서 게스트 운영체제로부터 얻을 수 있는 정보는 하드웨어 수준의 물리 메모리 혹은 레지스터에 국한 된다. 이러한 문제를 시맨틱 갭(Semantic Gap)이라고 부르는데, 이를 해결하기 위해서는 하이퍼바이저에서 게스트 운영체제의 물리적 메모리에서 충분한 의미상의 정보를 얻어낼 수 있는 방법이 반드시 필요하다. 메모리 상에서 의미상 정보를 추출하는 방법은 크게 특정 메모리 시작 주소로부터 선형 검색을 통해 자료 구조를 판단하는 방법과 컴파일 시 생성되는 심볼 테이블(Symbol table) 혹은 디버깅 정보를 담고 있는 파일로부터 얻어진 주소 정보를 따라가면서 자료구조를 추적하는 방식으로 구분 할 수 있다. Gibraltar 는 게스트 운영체제의 의미상 정보를 얻기 위해서 리눅스 컴파일 시에 생성되는 심볼테이블(System.map)에서 얻어진 전역 변수의 주소 정보로부터 메모리 정보를 추출 하는 방식을 사용한다. 심볼 테이블에 명시된 주소에 존재하는 전역 변수들로부터 자료 구조가 가진 값 등의 정보를 얻기 위해서는 해당 하는 변수의 타입 정보를 알고 있어야 하는데 본 연구에서는 CIL(C Intermediate Language)[4]을 통해 이러한 정보를 얻는다. Data Structure Extractor 에서 추출된 자료 구조의 정보는 머신 러닝 툴인 Daikon[5]을 통해 정상적인 운영체제가 수행 되는 동안에는 반드시 그 정보가 유지되는 데이터 인배리언트(Data Invariant)를 형태로 표현 된다. 다시 말해, 기존 Petroni[1] 연구에서 전문가들에 의해 수동으로 생성되었던 무결성 명세서가 Gibraltar 에서는 Invariant 의 형태로 자동으로 생성되는 것이다. 최종 모니터(Monitor)단계에서 이렇게 얻어진 Invariant 들의 변경 여부를 검사함으로써 가상화 장치의 무결성을 검증할 수 있다.

다른 가상화 검사와 마찬가지로 Gibraltar 를 통해 게스트 운영체제의 무결성을 충분히 검증하기 위해서는, Data Structure Extractor 단계에서 게스트 운영체제의 물리 메모리로부터 자료구조를 명확히 추출할 수 있어야 한다. 하지만 리눅스 내부에서 가장 많이 사용되는 리스트 연결(linked-list) 자료 구조는 이러한 주소 추적 기반 자료 구조 복원을 어렵게 만든다. 리눅스에서는 연결하고자 하는 자료 구조의 임의의 위치에 list_head 구조체를 포함하고 list_head 내부의 next/prev 포인터 변수를 서로 연결하는 방식으로 리스트 생성한다. 리스트 변수를 사용하고자 할때는 (그림 2)와 같이 리스트가 속해있는 컨테이너 구조체를 반드시 코드 상에 명시해야만 컴파일 시에 해당 리스트의 컨테이너 자료 구조의 시작 주소를 알 수 있게 된다.

```
struct list_head {
   struct list_head *next, *prev;
};

#define container_of(ptr, type, member) ({\
   const typeof(((type*)0)->member)*_ptr = (ptr);\
   (type *)((char*)_ptr - offsetof(type, member));})

#define list_entry(ptr, type, member) \
   container_of(ptr, type, member)
```

(그림 2) list_head 매크로

이러한 리눅스의 방식은 메모리의 주소와 타입 정보만으로 자료 구조를 따라 가는 것을 불가능 하게 만든다. 기존 Gibraltar 에서는 이 문제를 우회하기 위해서, 운영체제의 동작에서 중요하다고 판단되는 리스트 자료구조를 선정하여 (그림 3)과 같이 수동으로 코드에 컨테이너 정보를 기입하는 방식을 사용하였다.

```
struct task struct {
    struct list_head
    Container(struct task struct, run list) run_list;
};
```

(그림 3) Gibraltar 의 수동 컨테이너 정보 예제

이는 전문가의 개입이 문제가 되었던 기존 Petroni[1] 연구와 동일한 문제점을 야기하는 모순에 빠지게 된다. 또한 선정된 일부분의 리스트 구조체를 대상으로 하였기 때문에 신뢰도를 증명 할 수가 없으며, 코드 정보가 수정되는 경우 모두 직접 수정을 해야 한다.

3. 자동화된 linked-list 자료 구조 추출

본 연구는 소스 코드 정적 분석을 통하여 메모리로부터 자료구조를 추출하는 중 만나게 되는 리스트의 최종 컨테이너 자료 구조를 자동으로 찾아내어, 전문가가 수동으로 코드를 수정하여야만 했던 기존 연구의 문제점을 개선 하고자 하는 것을 목표로 하였다. 이를 위해 리눅스 소스 코드 상에 존재하는 다양한 리스트 관련 매크로를 아래와 크게 네 가지로 경우로 분류하고, 각각의 경우에 따라 맞는 구현 방식을 선택하였다.

Case 1. list_entry(ptr, type, member)
list_entry 는 리스트의 컨테이너 구조체의 시작 주소를 알 수 있도록 해주는 기본적인 매크로이다. 코드 상에 명시된 타입 정보를 통해 메모리 자료 구조 추적 중 만나게 되는 list_head 변수(member)의 컨테이너 구조체의 타입 정보(type)을 간단하게 알 수 있다.

```
X FUNC(...)
{
   list_entry(p, struct A , a)
}
```

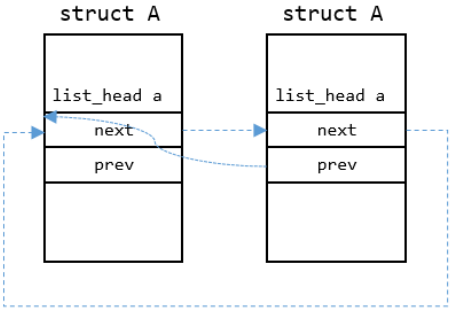
(그림 4) list_entry 예제

Case 2. list_for_each_entry(pos, head, member)
전체 리스트를 순회할때 사용되는 매크로에는 각각의 목적에 따라 list_for_each_entry, list_for_each_entry_safe, list_for_each_entry_rcu, list_for_each_entry_reverse 등이 있다. 이들 매크로를 분석해보면 list_entry 와 마찬가지로 pos 구조체의 멤버인 member 리스트끼리 연결된 구조를 가진다. Pos 구조체의 타입을 알기 위해서 list_for_each_entry 함수로부터 코드 역 추적을 통해 함수 내부에 선언된 타입 정보를 얻어야한다.

```
X FUNC(...)
{
   struct A *p;
   …
   …
   list_for_each_entry(p, &n, a)
}
```

(그림 5) list_for_each_entry 예제

Case 1,2 의 각 리스트 간의 연결 관계를 도식화 하면 아래 (그림 6)과 같이 동일한 구조체의 리스트간의 연결로 표현 된다는 것을 알 수 있다.



(그림 6) Case 1,2 리스트 연결

Case 3. list_for_each_entry(pos, head, member)
Case 2 와 코드는 같은 형태이지만 pos 을 통해 리스트를 순회하는 것이 아니라, liked_list 의 헤더를 나타내는 head 가 pos 구조체의 member 를 가리키고 있는 형태이다. 리눅스에서 헤더가 포함되어있는 구조체 타입과 실제로 리스트로 연결되어있는 구조체의 타입이 다를 경우가 많은데 이런 경우를 검출할 수 있다. Case 2 와 마찬가지로 list_for_each_entry 로부터 코드 역추적을 통해 head 와 pos 의 타입 정보를 각각 얻어낸다.

```
X FUNC(struct B *b)
{
   struct A *p;
   …
   …
   list_for_each_entry(p, &b->list, a)
}
```

(그림 7) 함수 내부의 리스트 헤더 정보 추적 예제

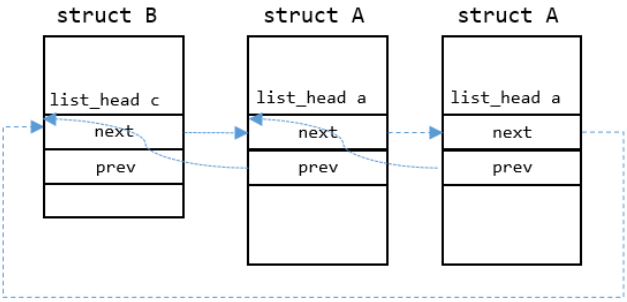
Case 4. list_for_each_entry(pos, head, member)
case 3 과 같은 경우이지만 함수 내부에 head 정보가 선언 된 것이 아니라 전역변수로 선언된 경우이다. 이 경우 함수 내부의 역추적을 통해서는 헤더 정보를 얻을 수 없기 때문에 별도의 코드 정적 분석을 통해 전역 변수 정보를 얻어야한다.

```
struct B *b;

X FUNC(...)
{
   struct A *p;
   …
   …
   list_for_each_entry(p, &b->list, a)
}
```

(그림 8) 전역 선언된 헤더 정보 추적 예제

Case 3, 4 와 같이 리스트의 헤더를 포함하고 있는 자료구조와 실제 리스트로 연결된 자료 구조가 구분되어있는 경우는 아래 (그림 9)과 같이 도식화 할 수 있다.



(그림 9) Case 3,4 리스트 연결

4. 실험결과

실험을 통해 Linux/ARM 3.14 버전의 심볼 테이블로부터 총 31,569 개의 변수의 메모리 주소를 얻었으며, CIL 정적 분석을 통해 총 4,829 개의 타입 정보를 추출하였다. 이러한 타입 정보 중 리눅스에서 리스트를

나타내는 list_head 는 총 1,289 개가 존재하는 것으로 나타났다. 기존의 Gibraltar 의 경우 소스 코드 상에 수동으로 163 개의 연결된 컨테이너 구조체 정보를 명시하였고, 이를 통해 236,444 개의 Invariant 정보를 얻었다. 본 연구에서는 Python 으로 개발된 자동화된 정적 분석 툴을 이용하여 기존 연구 대비 약 38% 가 증가한 224 개의 컨테이너 구조체 정보를 얻어내었다. 이를 바탕으로 커널 메모리 영역으로부터 자료구조를 찾아내었고, 기존 연구 대비 46% 증가한 345,573 개의 invariant 를 추출해내었다. 이 실험 결과에는 별도의 분석 툴을 통해 리스트 헤더의 선언이 함수 범위 밖에 있거나 외부의 소스코드를 분석 할 필요로 하는 Case 4 를 포함되지 않았다. 따라서 향후 연구에서 이 부분에 대한 추가적인 성능 향상을 기대할 수 있다.

5. 결론 및 향후 연구

본 연구의 의의는 실험 결과에도 잘 나타나 있듯이 자동화된 메모리 분석 기법을 도입함으로써 기존 루트킷 탐지 도구인 Gibraltar 의 성능을 획기적으로 개선하였다는 점이다. 이는 다시 말하자면, 이후 추가적인 Gibraltar 의 성능 향상을 기대 하기 위해서는 반드시 하이퍼바이저와 게스트 운영체제간의 시맨틱 갭을 줄이는 연구가 필요하다고 할 수 있다.

따라서, 향후 연구에서는 Gibraltar 가 발표된 이후 진행되어온 커널 메모리 자료구조 추출에 관한 Kernel Object Pinpointer (KOP)[6], MAS[7] 등의 연구와의 접목을 고려해 볼 수 있다. 뿐만 아니라 현재 Gibraltar 에서 사용하고 있는 것처럼 게스트 운영체제의 메모리를 심볼 테이블로부터 찾아가는 방식 대신, OSck[8] 와 같은 연구가 제시하고 있는 메모리 선형 검사 방식을 통해서도 성능을 개선할 수 있을 것으로 기대한다.

Acknowledgement

본 연구는 미래창조과학부 및 한국산업기술평가관리원의 산업융합원천기술개발사업(정보통신) [No.10047212, 1kB 이하 암호문 간의 연산을 지원하는 동형 암호 원천기술 개발 및 응용 연구]과 2015 년도 두뇌한국 21 플러스사업, 중소기업청에서 지원하는 2014 년도 산학연협력기술개발사업(No. C0218072), IDEC 의 지원 및 2014 년도 정부(미래창조과학부)의 재원으로 한국연구재단의 지원(No. 2014R1A2A1A10051792)을 받아 수행된 연구임을 밝힙니다.

참고문헌

[1] Petroni Jr, Nick L., et al. "An Architecture for Specification-Based Detection of Semantic Integrity Violations in Kernel Dynamic Data." *Usenix Security*. 2006.

[2] Baliga, Arati, Vinod Ganapathy, and Liviu Iftode. Automatic inference and enforcement of kernel data structure invariants." *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*. IEEE, 2008.

[3] Garfinkel, Tal, and Mendel Rosenblum. "A Virtual Machine Introspection Based Architecture for Intrusion Detection." *NDSS*. Vol. 3. 2003.

[4] Necula, George C., et al. "CIL: Intermediate language and tools for analysis and transformation of C programs." *Compiler Construction*. Springer Berlin Heidelberg, 2002.

[5] Ernst, Michael D., et al. "The Daikon system for dynamic detection of likely invariants." *Science of Computer Programming* 69.1 (2007): 35-45.

[6] Carbone, Martim, et al. "Mapping kernel objects to enable systematic integrity checking." *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009.

[7] Cui, Weidong, et al. "Tracking Rootkit Footprints with a Practical Memory Analysis System." *USENIX Security Symposium*. 2012.

[8] Hofmann, Owen S., et al. "Ensuring operating system kernel integrity with OSck." *ACM SIGPLAN Notices*. Vol. 46. No. 3. ACM, 2011.