동적 클래스 계층구조를 이용한 MAPbox상에서의 악성코드 탐지 기법
(Detecting Malicious Codes with MAPbox using Dynamic Class Hierarchies)

김 철 민 † 이 성 욱 †† 홍 만 표 †††
(Cholmin Kim) (Seong-uck Lee) (Manpyo Hong)

요 약 알려지지 않은 악성 코드의 수행에 의한 피해를 막는 방법으로 프로그램의 실행 환경을 제한하는 '샌드박스' 기법이 많이 쓰여져 왔다. 코드의 비정상 행위를 탐지하는 이 기법은 구현 방식에 따라 적용성(configurability)과 편리성(ease of use) 간의 양면성(trade-off)을 가진다. 기존의 MAPbox는 이 두 가지를 동시에 만족 시키기 위해 프로그램의 클래스별로 특정 샌드박스를 두는 클래스별 샌드박스 적용 기법을 사용한다[1]. 그러나, 이 방법은 정적으로 클래스의 수와 특성이 결정되므로 적용성에 한계가 있다. 본 논문에서는 MAPbox의 개념에 동적 클래스 생성 기능을 추가함으로써 적용성을 높이는 기법을 소개하고 이를 구현한다. 새로이 생성된 클래스에는 적절한 접근 제어를 가한다. MAPbox에 비해 적용성이 높아진 예로 MAPbox에서는 정상행위이지만 비정상행위로 판단되는 경우가 제안된 기법을 통해 올바르게 판단됨을 보인다. 또한 이 기법을 분석하고 실제로 구현 하기 위해 어떠한 문제를 해결 하였는지 보인다.

키워드 : 샌드박스, 적용성, 편리성, MAPbox

Abstract A Sandbox has been widely used to prevent damages caused by running of unknown malicious codes. It prevents damages by containing running environment of a program. There is a trade-off in using sandbox, between configurability and ease-of-use. MAPbox, an instance system of sandbox, had employed sandbox classification technique to satisfy both configurability and ease-of-use [1]. However, the configurability of MAPbox can be improved further. In this paper, we introduce a technique to attach dynamic class facility to MAPbox and implement MAPbox-advanced one. Newly generated class in our system has an access control with proper privileges. We show an example for improvements which denote our system have increased the configurability of MAPbox. It was determined as abnormal by MAPbox although is not. Our system could determine it as normal. We also show our techniques to overcome obstacles to implement the system.

Key words : Sandbox, Configurability, Ease-of-use, MAPbox

1. 서 론

일반적으로 컴퓨터에서 사용되는 프로그램이 항상 정상적인 행위만을 수행하는 것은 아니다. 바이러스에 의해 감염 되거나, 트로이 목마와 같이 프로그램의 제작자가 처음부터 나쁜 의도로 만든 프로그램, 또는 제작시의 실수나 이후의 테스트를 위해 남겨진 트랩도어 등에 의해 원래의 프로그램이 해야하는 기능이 아닌 악성 행위를 할 수 있다. 이러한 악성 프로그램을 검출하는 방법으로 실행 전에 정적인 테스트를 수행하는 방법이 있지만 프로그램 크기가 커지고, plug-in과 같은 바이너리 상태의 코드를 다운로드 받는 경우가 많아진 요즘의 컴퓨터 환경에서 정적인 테스트는 한계가 있다. 대신 프로그램이 수행할 정상행위(expected functionality)를 미리 정의하고 프로그램이 정상행위만을 수행하는지 관찰한다면 프로그램의 악성 행위를 막을 수 있을 것이다. 본 연구에서는 실시간으로 프로그램의 잘못된 행위를 탐지

· 본 연구는 한국과학재단 목적기초연구(과제번호 R05-2003-000-11235-0) 지원으로 수행되었음

† 정 회 원 : 아주대학교 정보통신전문대학원
cmkim@sysonchip.co.kr
†† 정 회 원 : 신구대학 인터넷정보과 교수
suleeip@yahoo.co.kr
††† 종신회원 : 아주대학교 정보및정보통신공학부 교수
mphong@ajou.ac.kr

논문접수 : 2003년 11월 19일
심사완료 : 2004년 9월 2일

하는 방식에 더욱 높은 정확성을 주는 방안을 제시하고자 한다.

2. 관련 연구

2.1 샌드박스

샌드박스는 개발자에 의해 기술된 프로그램의 정상행위에 의거하여 환경을 제한함으로써 주어진 환경의 제한을 위반하여 자원을 요구하는 프로그램을 검출하는 방법이다[2,3]. 즉 정의된 정상 행위를 가지고 프로그램을 실행하다 권한 이상의 것을 수행하려 할 때, 그 프로그램을 악성이라고 판단하는 것이다. 여기서 환경은 프로그램이 수행되는데 필요한 모든 자원을 의미한다. 샌드박스가 적용된 시스템에서는 아직 행위가 밝혀지지 않은 프로그램이 실행되어서 악성 행위를 보인다 하더라도 한정된 자원 이외의 부분에는 직접적인 영향을 주지 않게 된다. 따라서 악성 코드에 의한 피해를 최소화 할 수 있다.

샌드박스는 어떻게 구현하느냐에 따라 적용성(configurability)과 편리성(ease of use)의 양면성(trade-off)을 가진다. 적용성은 얼마나 더 정확하게 다양한 프로그램이 악성인지 아닌지를 판별 할 수 있는가를 나타내는 정도이고, 편리성은 환경을 제한하는데 사용자의 개입이 얼마나 적은가를 나타내는 정도이다. 샌드박스의 구현은 이 양면성에 근거해 크게 네 가지 부류로 나눌 수 있다.

첫번째는 프로그램마다 독립적인 접근제어(per program access control)를 적용하는 것이다[8-10]. 이 방법은 일반적으로 가장 적용성이 높다. 그러나 사용자(또는 관리자)는 모든 프로그램에 대한 접근제어 정보를 정의 해 주어야 한다. 따라서 사용자는 시스템 자원과 이에 대한 프로그램의 요구를 정확히 이해 해야 하며 결과적으로 편리성은 떨어진다. 이 방법은 사용하는 프로그램의 수가 적고, 구성 프로그램의 변화도 적은 환경에는 적합하다. 그러나 현재의 컴퓨터 사용 환경은 프로그램을 다운로드 받는 등의 변화가 심하고 사용되는 프로그램의 수도 많으므로 일반적인 사용자들에게 이 방법은 적절치 않다.

두 번째 방법은 프로그램의 행위를 나타내는 상태 전이 머신(finite state machine)을 정의하는 것이다[11-13]. 이 방법은 같은 프로그램이라 할 지라도 실행 중의 엑세스 시퀀스가 다른 경우가 구별되므로 경우에 따라서는 첫번째 방법보다 더 높은 적용성을 줄 수 있다. 그러나 상태 전이 머신을 구현하기 위해서는 프로그램의 행위를 깊이 이해 해야 하므로 사용 환경이나 프로그램의 수에 따라 구현하기에 너무 어려울 수 있다.

세 번째 방법은 프로그램의 공급자(제작자, 제작 회사, 웹 사이트)에 따라 서로 다른 접근제어 리스트를 가지는 것이다[14,15]. 이 방법은 사용자가 각 프로그램마다 정의해 주어야하는 일이 적으므로 편리성 면에서는 큰 이득을 준다고 볼 수 있다. 그러나 공급자가 같은 프로그램이 반드시 같은 행위를 가진다고 보장 할 수 없으므로 적용성이 떨어진다. 또한 프로그램 공급자의 수가 증가하는 것도 문제이다. 다시 말해 새로운 공급자에 대해 새로운 샌드박스를 만들어야 한다는 것이 사용자에게 큰 부담이 될 수 있다.

네 번째 방법은 프로그램의 종류별로 특정 샌드박스를 만드는 것이다[3]. 여기서 프로그램의 종류는 다큐먼트 뷰어, 애플릿, CGI 스크립트, 루트 권한으로 실행되는 프로그램 등으로 나눌 수 있다. 이러한 샌드박스들은 명시적으로 제한되는 환경의 범위를 가지므로 사용자가 해야 할 일이 줄어든다. 예를 들어 샌드박스의 종류를 에디터와 게임으로 나눈다면 사용자가 해야 할 일은 에디터와 게임이 일반적으로 필요로 하는 자원을 조사하여 각각의 샌드박스를 만들고 그 후에는 어떤 샌드박스를 통해 프로그램이 실행될지 만 결정하는 것이다. 따라서 적용성과 편리성을 동시에 만족시켜 준다고 볼 수 있다. 이 샌드박스들은 기능별로 나누어져 있으므로 한번의 설정으로 계속해서 사용할 수 있다. 앞에서의 방법들과 비교해 볼 때 기능별 분류가 한가지 종류의 여러가지 프로그램에 대해 가장 잘 적용될 수 있다.

2.2 MAPbox[3]

MAPbox(Multi-purpose Application Profile type sandbox)는 앞 절에서 말한 환경을 제한하는 네 가지 방법중 마지막 방법에 의해 구현된 것이다. 즉 각 프로그램의 종류별로 만들어진 샌드박스들이 해당 프로그램 클래스의 실행을 전담하는 것이다. MAPbox도 프로그램 클래스를 어떻게 나누느냐에 따라 적용성과 편리성 간에 트레이드 오프를 가진다.

사용자가 실행 해야 할 프로그램으로 에디터와 게임이 있다고 가정하자. 이 두 프로그램은 서로 다른 정상 행위를 가질 것이다. 만약 적용성을 높이려 한다면 에디터를 위한 샌드박스와 게임을 위한 샌드박스를 개별적으로 만들면 된다. 그러나 이 경우 실행되는 프로그램이 에디터에 해당되는지, 게임에 해당되는지는 사용자가 결정해야 한다. 따라서 샌드박스를 세분화 할수록 편리성은 떨어지고 적용성은 높아진다. MAPbox는 다른 샌드박스 시스템과 비슷한 적용성을 유지 하면서도 높은 편리성을 제공한다[3]. 클래스의 분류로 인해 MAPbox의 편리성이 떨어지는 것으로 보일 수 있지만 여기서는 클래스들을 컴파일러, 에디터, 뷰어, 메일러, 브라우저 등 비교적 사용자가 직관적으로 알 수 있도록 구분하였으므로 그리 큰 문제가 되지 않는다. 즉 MAPbox는 적용성과 편리성을 동시에 만족시킨다.

MAPbox는 이외에도 적용성을 높이기 위해 파라미터를 받을 수 있도록 되어 있다. 여기서 파라미터란 프로그램이 실행되는 데 필요한 자원의 구체적인 제한을 의미한다. 같은 컴파일러 클래스에 있더라도 자바 컴파일러와 C 컴파일러는 서로 다른 라이브러리를 참조할 것이다. 따라서 컴파일러 클래스가 참조하는 라이브러리의 경로를 컴파일러 샌드박스 자체에 제한하는 것이 아니라 각각의 컴파일러 별로 실행 할 때의 파라미터로 제시하도록 되어 있다. 이 때 컴파일러 샌드박스는 라이브러리 경로를 파라미터로 받게 되어 있다는 것만 명시되어 있으면 된다. MAPbox에서 정의하고 있는 프로그램 클래스들과 그들의 파라미터, 그리고 그 파라미터로 가능한 정상행위는 부록 1로 첨부 하였다. 이 클래스 들의 분류는 대부분의 프로그램 종류들을 포함 할 수 있도록 디자인 된 것이다. 한편 하나의 프로그램이 다른 종류의 프로그램을 실행(exec) 할 수 있는 경우를 처리하기 위해 몇 가지 클래스는 해당 프로그램이 실행 되는 도중 실행 가능한 다른 클래스들을 정의 하고 있다.

3. 동적 클래스 계층구조를 이용한 MAPbox 상에서의 악성코드 탐지 기법

3.1 MAPbox의 단점

MAPbox의 각 클래스는 프로그램의 정상행위를 정적으로 정의하고 있다. 즉 처음 샌드박스가 생성될 때 해당 프로그램 클래스의 특성이 결정되어 버린다. 파라미터로 쓰일 파일경로나 네트워크 포트를 바꿀 수는 있지만, 그전에 파라미터에 없던 접근을 추가하는 것이 불가능 한 것이다. 이것은 업데이트를 통해 새로운 기능을 가지는 프로그램이 등장하였을 경우 문제를 일으킨다. 새로운 프로그램은 정상적인 프로그램이지만 기존에 정의 되어 있는 파라미터에 없는 권한을 요구할 수 있다. 이때 MAPbox는 새로운 프로그램을 악성이라 판단할 것이다. 예를 들어 html파일을 작성할 수 있고 그것을 웹 서버에 업로드 시킬 수 있는 에디터를 가정하자, MAPbox가 정의하고 있는 에디터 클래스는 네트워크의 엑세스 권한을 갖지 않는다. 그러나 이 새로운 에디터는 네트워크의 접근을 필요로 할 것이고 MAPbox를 통해 실행 된다면 악성 코드로 인식 된다. 제안된 기법은 이와 같은 MAPbox의 한계를 보완하는 것을 목적으로 한다. 즉 실행중인 프로그램이 MAPbox가 정의하고 있는 해당 클래스의 정상행위 외의 행위를 보이더라도 바로 악성코드로 판단하지 않는다. 만약 프로그램이 요구하는 새로운 자원의 요구를 사용자가 허용할 경우, 이러한 권한을 정상행위로 가지는 클래스를 동적으로 추가한다. 이를 위해 일단 MAPbox에서 정의된 클래스들을 이용하고, 새롭게 생성되는 클래스들을 기존 클래스의 하위 클래스로 만든다. 이와 같이 기존의 클래스에 포함 되지 않는 새로운 프로그램의 종류를 계속해서 사용할 경우 제일 상위의 MAPbox에서 정의한 클래스들 중에서 새로운 프로그램에 가장 가까운 클래스의 하위 클래스가 생성된다. 이러한 과정이 반복되면 MAPbox의 클래스들을 루트 노드로 하는 클래스 트리의 집합이 생성된다. 제안된 기법을 이용하면 MAPbox를 이용할 때보다 정상인 프로그램을 악성이라 판단하는(false positive) 오류를 줄일 수 있다.

3.2 제안된 기법의 형식적 설명

제안된 기법을 형식적으로 표현하면 다음과 같다. 클래스의 집합을 C, 권한의 집합을 P라고 하자. 즉 일반적으로 n개의 클래스와 m개의 권한이 지정되어 있는 샌드박스 시스템이 가지는 초기 전체 클래스와 권한의 집합은 다음과 같이 나타난다.

$$\text{Class} : C = \{C_1, C_2, \cdots, C_n\}$$

$$\text{Privilege} : P = \{P_1, P_2, \cdots, P_m\}$$

클래스 집합에 대해 '초기'라는 표현을 쓰는 것은 이 집합이 유동적으로 변경 가능하기 때문이다. MAPbox는 이 집합이 정적인데 비해 제안된 기법은 이 집합을 동적으로 변경하는 것이 가능하다.

제안된 기법에 의해서 생성되는 새로운 클래스는 기존 클래스의 자식 노드 형태로 표현된다. 즉 제안된 기법에 의해 클래스들이 추가되면 전체 클래스의 구조가 초기 클래스들이 루트 노드가 되는 포레스트(forrest)의 형태를 띠게 된다. 이때 새롭게 생성되는 클래스가 C_a의 x번째 자식 노드로 생성이 된다면 새로운 클래스를 C_{ax}로 표기한다. 이 노드의 하위 노드가 또 생성되면 그 클래스는 C_{axy} 와 같은 형태로 표기한다.

각 클래스가 지니는 권한집합은 위 C 집합에서 P 집합으로의 일대 다 대응으로 표현 되며 각 클래스가 지니는 초기 권한대응은 MAPbox의 것을 이용한다. 특정 클래스에 대응되는 권한의 추가는 해당 클래스의 부모 노드 클래스가 지니는 권한대응에 새롭게 생성된 권한과의 대응을 추가하는 것이 된다.

예를 들어 클래스 C_a에 대응되는 권한의 집합을 AP라고 표기하자. AP는 P의 부분집합이 된다. 이 클래스의 b 번째 자식 노드가 새로운 권한 P_c를 받아서 생성될 때의 권한집합 ABP는 다음과 같다.

$$\text{Privilege } ABP = AP \cup \{P_c\}$$

$$|ABP| = |AP| + 1$$

$*|P| = P$ 집합의 원소 수

새롭게 추가된 권한은 C_{ab} 클래스의 $|AP| + 1$ 번째 권한이 되며 이 권한으로 C_{ab}클래스는 C_a클래스 보다 하나 더 많은 권한을 가지게 된다. 새롭게 생성되는 권

한이 할당되어도 되는지의 판단 여부는 사용자에게 맡겨진다. 시스템이 보여주는 할당여부에 대해 사용자가 승낙할 경우 시스템은 자동적으로 하위 클래스를 생성하고 새로운 권한을 하위 클래스에 할당한다.

3.3 제안된 기법에 의한 문제 해결

그림 1은 제안된 기법의 pseudo-code로서 대략적인 구현방법을 제시하고 있다. 대상 프로그램의 실행 도중 클래스에서 정의한 자원의 범위를 벗어나는 권한을 프로그램이 요구할 경우 일단 이 권한을 가지는 하위 클래스가 있는지를 확인한다[그림 1, 줄 번호 7]. 만약 현재의 하위 클래스 중 해당 권한을 가지는 것이 없다면 제안된 기법에서는 일단 이 새로운 권한을 사용자에게 알린다. 사용자가 이 권한을 현재 프로그램에게 허용한다면 이것은 앞으로 추가될 권한의 후보로 기록되고 새로운 클래스를 만들 준비를 위해 Make_New_Class() 서브루틴을 호출한다[그림 1, 줄 번호 9-11]. 한편 이 권한을 사용자가 허용하지 않는다면 프로그램은 악성으로 인식되고 새로운 클래스는 생성되지 않는다[그림 1, 줄 번호 12, 13]. 이후에 프로그램은 곧바로 종료되어 다른 피해를 막는다. 그러나 Make_New_Class() 서브루틴으로 프로그램 실행이 넘어간 후, 프로그램 종료 시까지의 새로운 권한들을 사용자가 모두 허용한다면 이 권한들을 정상행위로 가지는 새로운 클래스가 생성되고 이것은 처음 프로그램이 실행되었던 클래스의 하위 클래스로 지정된다[그림 1, 줄 번호 21, 22, 25].

새롭게 생성된 클래스에 해당되는 다른 프로그램이 실행될 경우 이 프로그램은 일단 생성된 클래스가 속하는 트리의 루트 노드 클래스에 의해 실행 되기 시작한다. 새로운 권한을 요구할 때마다 현재 노드의 하위 노드에 속하는 클래스중 해당 권한을 가지는 클래스로 프로그램이 이동되어 실행되고[그림 1, 줄번호 8] 결국 앞서 생성된 클래스에 와서 종료된다면 정상적인 프로그램으로 판단될 것이다. 만약 트리의 어떤 노드 상에서 이 프로그램이 요구하는 권한을 지원하는 하위 노드도

```
1  알고리즘 : Main ( current_class, lst )
2  Input        -  current_class : 현재 프로그램이 실행되고 있는 샌드 박스상의 클래스
3               -  lst : 현재 프로그램이 소유하고 있는 권한의 리스트
4
5  while ( 프로그램 실행 )
6          if ( lst가 가지는 권한을 벗어남 )
7              if ( current_class의 자식 클래스 중 해당 권한을 가진 클래스가 있음 )
8                  call Main ( current_class -> child , current_class -> child.lst )
9              else if ( 사용자가 새로운 권한을 허용 )
10                 lst = lst +  새로운 권한
11                 call Make_new_class( current_class, lst )
12             else if ( 사용자가 새로운 권한을 허용하지 않음 )
13                 악성 프로그램으로 판정, 프로그램 강제 종료
14
15 알고리즘(서브루틴) : Make_new_class ( current_class, lst )
16 Input        -  current_class : 현재 프로그램이 실행되고 있는 샌드 박스상의 클래스
17              -  lst : 현재 프로그램이 소유하고 있는 권한의 리스트
18
19 while ( 프로그램 실행 )
20      if ( lst가 가지는 권한을 벗어남 )
21             if ( 사용자가 새로운 권한을 허용 )
22                 lst = lst +  새로운 권한
23             else if ( 사용자가 새로운 권한을 허용하지 않음 )
24                 악성 프로그램으로 판정, 프로그램 강제 종료
25 current_class를 Make_new_class()서브루틴를 호출한 클래스의 자식 클래스로 만듦
```

그림 1 제안된 기법의 알고리즘

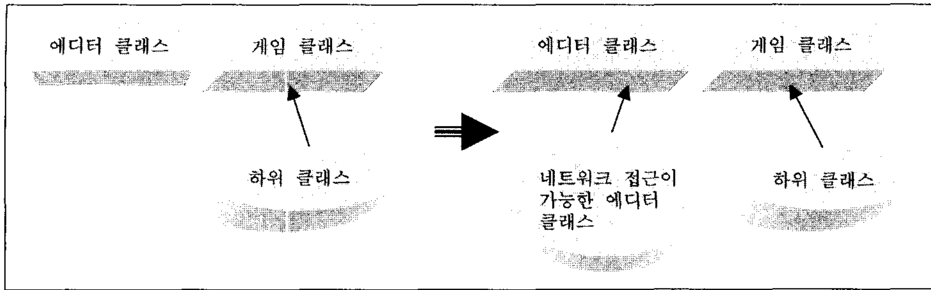

그림 2 제안된 기법에 의한 클래스의 생성

없고, 사용자가 이 권한을 허용하지도 않는다면 그 프로그램은 악성으로 판정된다.

3.4 제안된 기법에 의한 개선

앞서 MAPbox를 통해 실행 될 경우 정상적인 프로그램이지만 악성프로그램인 것으로 판단되는 경우의 예로 네트워크 접근을 요구하는 에디터를 들었었다. 만일 제안된 기법을 통해서 이 프로그램이 실행된다면, 우선 사용자는 이 프로그램을 루트 클래스중 하나인 에디터 클래스로 실행하려 할 것이다. MAPbox와 동일하게 이 클래스에 의한 실행도중 프로그램이 네트워크 엑세스를 요구하지만, 에디터 클래스는 이것을 허용하지 않는다. 에디터 클래스의 하위 클래스가 아직 생성되지 않았다면 제안된 기법은 사용자에게 네트워크 엑세스를 허용할지를 물어본다. 단순히 시스템 콜을 보여주는 것 만으로도 사용자는 이 프로그램이 html의 업로드를 위해 네트워크 엑세스를 한다는 것을 알 수 있고, 네트워크 엑세스 권한을 허용할 것이다. 이 프로그램의 실행이 끝나면 에디터 클래스의 하위 클래스로 네트워크 엑세스 기능을 가지는 에디터를 위한 클래스가 새로 생성되며 앞으로 이 클래스를 이용한 보다 정확한 악성과 정상 프로그램간의 판별이 가능할 것이다[그림 2].

사용자에게 시스템 콜을 보여 주기 위해서는 애플리케이션이 운영체제에 요구하는 시스템 콜을 가로챌 필요가 있다. 시스템 콜을 가로채는 방안에 대해서는 4장 구현에서 자세히 다룬다. 파일 엑세스에 대한 시스템 콜의 경우 파라미터는 엑세스 대상인 파일에 대한 정보이다.

4. 구현

제안된 기법을 실제로 구현하기 위해 본 논문에서는 각 클래스별 샌드박스들을 객체 지향 언어의 class로 정의 하였다. 한편 프로그램의 모니터링을 위해 MAPbox는 실제 프로그램의 실행 중에 그 프로그램의 시스템 콜을 가로채는 방식을 이용해 구현 되었지만 본 논문에서는 시스템 콜에 대한 프로그램의 실행시간 로그 데이타를 만든 다음 그 로그 데이타를 멤버변수로 갖는 클래스를 정의 하였다. 실시간의 모니터링을 이용하지 않고 로그를 이용해서 시스템 콜을 분석하는 이유는 단지 구현상의 문제이며 실시간성이 요구될 경우 추가적인 구현이 필요하게 된다. 실행 환경은 리눅스이고 시스템 콜을 가로채어 로그를 남기는 프로그램은 C를 이용하여 리눅스 의존적으로 작성하였다[5,6]. 한편 이러한 로그를 분석하여 실제 이 논문에서 제안하고 있는 기법을 실험하는 프로그램은 C++를 이용하되 ANSI C++의 라이브러리들만을 이용하여 작성하였다[7].

먼저 시스템 콜을 가로채어 로그를 남기도록 하기위해 시스템 콜의 엔트리 포인트를 바꾸는 모듈형 프로그램을 작성하여 이것을 커널에 삽입한다[그림 3]. 그림 3에서 제안된 기법의 필요에 의해 구현된 부분은 가운데의 직사각형 부분이며 이 부분이 모듈형태로 리눅스 커널에 삽입된다. 모듈이 삽입되기 전에는 점선 형태로 표현한 원래 시스템 콜의 엔트리 포인트 대로 운영체제가 처리하나, 삽입된 모듈에 의해 운영 체제의 시스템 콜 처리 순서가 변경된다. 구현된 모듈은 변경된 순서에 의해 시스템 콜 호출 시 반드시 같이 호출되며 본 연구의 대상이 되는 모든 시스템 콜을 가로챌 수 있다.

모듈이 삽입된 후에는 프로그램이 시스템 콜을 호출하였을 경우 삽입된 모듈이 로그 데이타를 남기는 작업을 수행한다. 즉 엑세스한 파일의 경로와 그 파일이 갖

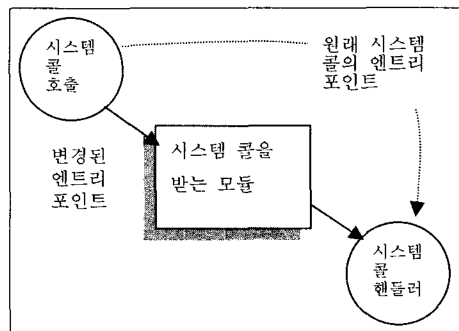

그림 3 시스템 콜 가로채기

```
/root/file0    r
/root/file0    x
/root/file0    w
/dev/fd0       r
/dev/fd0       w
/dev/tty0      w
/dev/tty0      r
/etc/file1     x
```

그림 4 로그 데이타의 형식

```
class Application
{
private :
        int NumOfParams;
        int Runned_System_Call;
        char Type[];
        char AppName[];
        char Parameters[][];
        bool Permissions[][3];

public :
        Application(char *, char *Name);
        char *Get_App_Type();
        void Add_Parameters(
            char AppName[]);
        void Show_Params();
        int   Get_Num_Of_Params();
        bool Program_Execution();
};
```

그림 5 애플리케이션 클래스

는 권한을 텍스트 형식으로 남기게 된다[그림 4].

이러한 형식으로 로그 파일이 만들어 진 후 이 로그 파일을 멤버변수로 갖는 에플리케이션 클래스를 생성하였다[그림 5]. 즉 로그 파일이 가지고 있는 다른 파일에 대한 권한들을 MAPbox에서의 각 에플리케이션이 요구하는 파라미터와 같은 것으로 해석한다. MAPbox에서는 파라미터마다 특정 파일의 경로나 포트 번호등을 받게 되어 있지만 실험에서는 구현의 편의를 위하여 모든 파라미터를 하나인 것으로 가정한다. 즉 파라미터로 들어올 개개의 파일이름, 포트 번호, 호스트 이름 등등에 대해 일일이 권한을 적용하는 것이다. Unix시스템 들은

```
class ClassifiedSandbox
{
private :
        int   NumOfPolicies;
        int   NumOfChild;
        int   NodeNumber;
        char TypeOfRoot[];
        char PolicyList[][];
        bool PermissionList[][3];
        class ClassifiedSandbox *Child[];

public :
        ClassifiedSandbox(char *RootType);
        char *Get_Root_Type();
        void Add_Policy(void);
        void Add_Policy(char *, bool *);
        void Show_Policies();
        void Run_App(Application App);
        void Make_New_Class(Application App
            ,char SelectedParam[], bool *);
        bool Check_Policy(char Policy[]
            ,bool Permission[]);
};
```

그림 6 샌드박스 클래스

모든 디바이스 및 소켓 접근을 파일과 같은 개념으로 처리하므로 이것은 해볼만한 일이다. 현재의 에플리케이션이 특정 파일에 대해 어떤 권한을 가지는지 혹은 가지지 않는지를 나타내기 위해 각 파일의 경로와 권한은 스트링과 불린형 변수로 저장된다(멤버 변수 Parameters, Permissions). 로그 데이타로부터 인자를 읽어 들이는 데는 Add_Parameters() 멤버 함수를 이용한다. 한편 이러한 에플리케이션 클래스의 인스턴스는 실제 샌드박스 클래스의 인스턴스에 의해 실행된다. 샌드박스 클래스는 루트 노드에 해당되는 것만이 최초에 인스턴스화 된다. 루트 노드가 아닌 샌드박스들은 실행도중에 동적으로 생성된다. 샌드박스 객체들은 그림 1 알고리즘의 Main에 해당하는 멤버함수인 Run_App()를 가진다. 각각의 샌드박스들은 하위의 샌드박스를 가르키는 Child 포인터를 가지고 있다. 오버로드된 두개의 Add_Policy() 멤버함수는 처음에 샌드박스가 생성될 때 전체 권한을 주기위한 것과 프로그램 실행도중 자식 클래스가 생성되어 하나의 권한을 더하는데 사용되는 것으로 구별되어 있다. Make_New_Class() 멤버함수는 위의 알고리

즘에서 제시한 것과 같은 기능을 가진다[그림 6].

5. 제안된 기법의 분석

제안된 기법은 이후에 실행 하게 될 모든 프로그램을 MAPbox에서 정의한 클래스들 중 하나로 분류할 수 있다고 가정한다. 즉 루트 노드에 해당되는 클래스는 더 이상 생성시키지 않고, 루트 노드의 하위 노드만 생성시킨다. 또한 이 기법을 이용하는 사용자는 프로그램이 요구하는 권한의 허용 여부를 판단할 수 있을 정도의 지식을 갖춘 사용자라고 가정한다. 최종적으로 새로운 클래스를 생성하는 것은 결국 사용자의 결정이기 때문이다. 제안된 기법은 유닉스 관련 시스템을 대상으로 하였고, 실제 구현도 리눅스 상에서 이루어졌다. 그런데 이러한 시스템 상에서의 응용프로그램 추가는 주로 관리자에 의해서 이루어진다. 따라서 일반 사용자에게는 권한 추가가 어려워지는 문제가 발생하는 것이 사실이지만 제안된 연구의 사용 대상을 시스템 관리자 라고 볼 때 큰 문제가 되지 않는다. 생성된 클래스는 상위 클래스가 가지는 모든 권한을 상속 받는다. 실행 중인 프로그램이 생성된 클래스에 전달되기까지 상위 클래스가 가지는 모든 권한의 범위 내에서 이미 실행 되었기 때문에 상속되는 권한이 악성 코드에게 가져다 주는 이득은 없다. 제안된 기법은 MAPbox에 비해 악성과 정상 프로그램을 구별할 때 적용성이 더 높다는 장점을 가진다. 반면 사용자가 프로그램이 새롭게 요구하는 권한이 정상인지 비정상인지를 판단해야 하므로 MAPbox에 비해 편리성이 떨어진다. 그러나, 만일 사용자가 MAPbox와 동일한 편리성을 가지기 위해 모든 판단을 일괄적으로 부정적으로 해버린다 해도 MAPbox와 동일한 적용성과 정확성을 보장 받는다. 그리고, 일반적인 샌드박스 구현방식들이 사용자의 권한 설정에 크게 의존하는 것에 비하면 이것은 그리 큰 문제가 아니다.

제안된 기법은 학습의 기능을 가지고 정상행위를 통해 악성 행위를 탐지하므로 기존의 학습형 비정상행위(anomaly) 탐지 기법과 유사점을 가진다[4]. 그러나 이러한 방법은 학습을 하더라도 대상 프로그램이 악성프로그램이다, 아니다의 판정이 더 정확해졌을 뿐이지 새롭게 추가된 권한에 대한 분석을 얻을 수는 없다. 이에 반해 제안된 기법은 미확인 프로그램이 요구하는 권한들이 새롭게 생성되는 클래스에 의해 밝혀진다는 장점이 있다.

6. 향후 연구 방향

제안된 기법의 성능을 향상 시키기 위해 앞으로 연구 가능한 과제를 몇 가지 생각 할 수 있다.

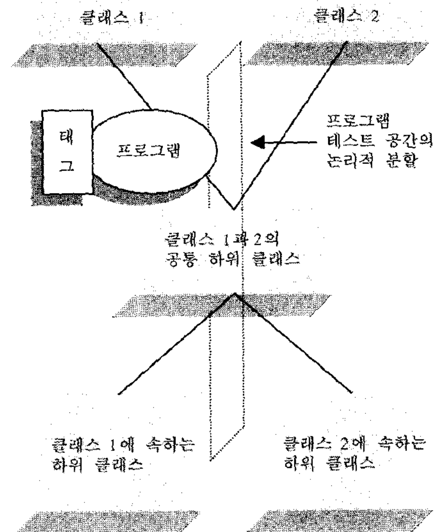

그림 7 같은 목적의 노드 통합

첫째로는 프로그램이 요구하는 권한에 대한 사용자의 판단을 돕는 방안에 대한 것이다. 프로그램이 요구하는 새로운 권한을 허용하는지의 여부는 전적으로 사용자의 의무이다. 따라서 사용자가 시스템에 대한 지식이 부족함에도 불구하고 프로그램이 요구하는 권한을 허용할 경우 오히려 피해가 더욱 심해질 수 있다. 따라서 확신할 수 없는 권한을 프로그램이 요구할 경우에는 사용자가 그 권한을 주지 않도록 해야 한다. 이런 경우에도 적어도 MAPbox와 같은 적용성을 보장한다. 한편 프로그램이 새로운 권한을 요구하기 전에 이미 그 동안의 시스템 콜의 로그 데이타를 저장하고 있으므로 이 데이타에 의한 일반적인 예측 결과를 사용자에게 보여줄 수 있을 것이다. 예를 들어 패스워드 파일이나 설정(configuration)파일에 대한 접근이 요청될 경우 해당 파일의 변경이 가져올 수 있는 시스템의 취약성을 사용자에게 보여준다면 사용자의 권한 허용 여부에 도움이 될 것이다.

둘째는 클래스의 생성에 의한 계층 구조의 효율성에 관한 것이다. 제안된 방식으로 계속해서 새로운 프로그램을 실행하면 새로운 프로그램이 많아 질수록 트리의 모양이 복잡해진다. 따라서 너무 트리의 깊이가 깊어지지 않도록 한계를 줄 수 있다. 또한 노드의 생성이 계속되면서 목적이 같은 노드들이 생겨날 수 있다. 예를 들어 네트워크 접근용의 프로그램이 에디터 기능을 가지게 된다면 이 프로그램에 의해 생성된 노드는 에디터이지만 네트워크 접근 기능이 추가된 프로그램을 테스트 할 수도 있을 것이다. 따라서 노드를 따로 만드는 것

표 1 제안된 기법과 기존 기법의 비교

	기존 기법(MAPbox)	제안된 기법
적용성(configurability)	중간 (새로운 클래스를 생성할 수 없음)	높다 (새로운 클래스의 동적 생성이 가능)
편리성(ease of use)	높다	높다 (시스템 구성을 이해하지 못하는 사용자의 경우 Default로 권한 허용을 하지 않을 수 있음)
확장성(scalbility)	낮다(새로운 권한으로의 접근을 무조건 차단, False Positive)	중간(새로운 권한을 사용자에게 알리며 새로운 클래스의 생성이 가능)
장점	처음의 권한 설정 (configuration) 이후 재설정이 필요 없음	알려지지 않은 새로운 프로그램의 추가에 대응할 수 있음
단점	알려지지 않은 새로운 프로그램을 악성으로 인식함	클래스의 지속적 확장에 의해 클래스 기술 정보의 크기가 지나치게 늘어날 수 있음

보다는 하나의 노드로 두개의 프로그램을 테스트하는 것이 더 효율적인 것이다. 그러나 이 노드의 하위 노드들의 성격이 반드시 같다고는 말할 수 없으므로 테스트 되는 프로그램이 하위 노드로 이동할 필요가 있을 경우 어떤 노드로 이동되는 지에는 제약이 가해져야 한다. 이를 위해 테스트 되는 프로그램마다 원래의 클래스가 무엇이었는지를 알리는 태그를 이용할 수 있을 것이다[그림 7].

7. 결 론

본 논문을 통하여 기존의 MAPbox가 알려지지 않은 악성 프로그램의 진단이 가능한 반면 고정적인 프로그램 클래스를 가짐으로 해서 나타나던 한계점, 즉, 정상적인 프로그램이라도 기존의 클래스에 적합한 형태를 가지지 않는 경우 무조건 악성으로 판단하던 문제점을 극복하기 위해, MAPbox를 확장하여 동적으로 클래스를 생성할 수 있도록 하는 방안을 제시하고 이를 구현해 보았다. 이것은 기존 클래스로부터 새로운 자원에 대한 접근을 명시함으로써 좀더 구체적이면서 확장성을 가지는 방법이며 그 결과 MAPbox의 false positive오류를 줄일 수 있었다.

본 논문에서 제안한 기법과 기존 기법을 비교하여 표 1에 정리하였다.

제안된 기법은 클래스의 지속적인 확장에 의해 클래스를 기술하는 정보가 지나치게 커질 수 있고, 클래스를 동적으로 생성하는 일 자체가 시스템에 어느 정도의 지식을 가진 사람에 한해서 새로운 형태의 프로그램에게 정책에 맞는 권한을 허용할 수 있다라는 단점을 가지고 있다. 때문에, 확장된 클래스 트리를 주기적으로 혹은 실시간으로 최적화 시키는 알고리즘을 추가하는 연구가 지속된다면 클래스 정보의 계속적인 확장을 어느 정도 줄일 수 있을 것이며, 100% 사용자의 선택에 의해 클래스를 확장 상속하는 것에서 자동화 기능을 첨가하는 것을 통해 사용자의 부담을 덜 수 있을 것이다. 마지막으로 이 기법을 이용하면 미확인 프로그램의 권한 특성이 밝혀지므로 이를 이용한 차후 연구가 가능할 것이다.

참 고 문 헌

[1] Anurag Acharya and Mandar Raje, "MAPbox : Using Parametererized Behavior Classes to Confine Aplications," *Computer Science Technical Report TRCS99-15*, 1999.

[2] Scott Oaks, JAVA Security, pp.120-135, O'REILLY, 1999.

[3] Ian Goldberg, David Wagner, Randi Thomas and Eric A. Brewer, "A Secure Environment for Untrusted Helper Applications," *Proceedings of the 1996 USENIX Security Symposium*, p.207, 1996.

[4] Roland Biischkes, Mark Borning and Dogan Kesdogan, "Transaction-based Anomaly Detection," *Proceedings of the Workshop on Intrusion Detection and Network Monitoring*, Vol.1, p.146, 1999.

[5] W. Richard Stevens, Advanced Programming in the UNIX Environment, p.287, Addison Wesley 1998.

[6] M. Beck, H Bohme, M Dziadzka, U Kunitz, R Magnus, D Verworner, LINUX KERNEL INTERNALS, p.324, Addison Wesley, 1999.

[7] Stephen Prata, C++ Primer Plus, pp.140-210, the White Group, 1995.

[8] G. Edjlali, A. Acharya, and V. Chaudhary, "History-based access control for mobile code," *Proceedings of the Fifth ACM Conference on Computer and Communication Security*, Vol.1, p.68, 1998.

[9] T. Gamble, "Implementing execution controls in Unix," *Proceedings of the 7th System Administration Conference*, Vol.1, pp.237-242, 1993.

[10] P. Karger, "Limiting the damage potential of the discretionary Trojan horse," *Proceedings of the 1987 IEEE Symposium on Reserch in Security and Privacy*, Vol.1, p.182, 1987.

[11] C. Ko, G. Fink, and K. Levitt, "Automated detection of vulnerabilities in privileged programs by execution monitoring," *Proceedings. 10th Annual computer Security Applications Conference*, Vol.1, pp.134-144, 1994.

[12] N. Mehta and K. Sollins. Extending and expanding the security features of Java. In *Proceedings of the 1998 USENIX Security Symposium*, 1998.
[13] F. Schneider. Enforceable security policies. Technical report, Dept of Computer Science, Cornell university, 1998.
[14] L. Gong. New security architectural directions for Java. In *Proceedings of IEEE COMPCON97*, 1997.
[15] T. Jaeger, A. Rubin, and A. Prakash, "Building systems that flexibly control downloaded executable content," *Proceedings of the Sixth USENIX Security Symposium*, Vol.1, p.139, 1996.
[16] 김철민, 홍만표, 예홍진, 조은선, 이철원, "샌드박스의 동적 클래스 계층구조를 통한 악성코드 탐지 기법", 정보보호연구회 발표 논문집, Vol.1, pp. 2001. 2.
[17] 김철민, 임영환, 홍만표, 예홍진, 조은선, "샌드박스의 동적 클래스 계층구조를 통한 악성코드 탐지 기법의 설계", *한국정보처리학회 춘계 학술발표논문집 제8권 제1호*, 2001. 4.
[18] Cholmin Kim, Younghwan Lim, Manpyo Hong, Sunho Hong and Eunsun Cho, "Design Mechanism for Malicious Code Detection with Sandboxes in Dynamic Class Hierarchies," *Proceeding of the 1st ACIS Annual International Conference on Computer and Information Science ICIS'01*, 2001. 10.
[19] IBM, "The Sandbox," Developer Works, http://www-10.lotus.com/ldd/sandbox.nsf, 2004.
[20] Security Company, Digital Sandbox, "Sandbox Solutions," http://www.dsbox.com/company/index.html, 2003.

부 록

표 2 각 프로그램 클래스들과 파라미터

프로그램 클래스	파라미터	설 명
filter		유닉스 필터; 파일 오픈, 네트워크 접근할 수 없음, 디스플레이 접근할 수 없음, 프로세스 실행할 수 없음. (e.g., sed, grep, comm., detex, deroff)
transformer	infile, outfile	infile을 읽어서 outfile을 write 할 수 있음, 네트워크 접근할 수 없음, 디스플레이 접근할 수 없음, 프로세스 실행할 수 없음. (e.g., compress, gzip, image format converter)
compiler	dir/filelist, libpath, outfile	dir/filelist의 파일과 libpath내에 있는 디렉토리를 읽을 수 있음, outfile에 쓸 수 있음, 네트워크 접근할 수 없음, 디스플레이 접근 가능, compiler 실행 가능. (e.g., gcc, make, tar, dvips, latex, nroff, bibtex, ld)
editor	dir/filelist	dir/filelist의 파일을 읽을 수 있음, 네트워크 접근할 수 없음, 디스플레이 접근 가능, filter/transformer 실행 가능. (e.g., gnu-emacs, vi, pico, xfig, idraw)
viewer	dir/filelist	dir/filelist의 파일들을 읽을 수 있음, 네트워크 접근할 수 없음, 디스플레이 접근 가능, viewer 실행 가능(e.g., ghostview, pageview, imagetool,xdvi)
download	host, port, dir	port 번호로 host에 접속할 수 있음, dir에 있는 파일에 쓸 수 있음, 프로세스 실행할 수 없음, 디스플레이 접근할 수 없음. (e.g., finger, whois, rwho, ftp get commands)
upload	host, port, dir	port 번호로 host에 접속할 수 있음, dir에 있는 파일들을 읽을 수 있음, 프로세스 실행할 수 없음, 디스플레이 접근할 수 없음. (e.g., ftp put commands, HTTP file upload)
mailer	mailbox, gateway	mailbox를 읽고 쓸 수 있음, gateway에 port 25로 접속할 수 있음, 디스플레이 접근 가능, viewer 실행 가능. (e.g., pine, elm, mailtool)
browser	hostlist, port	hostlist의 호스트에 port 번호로 접속할 수 있음, 디스플레이 접근할 수 있음, viewer 실행할 수 있음. (e.g., lynx, hotjava, trn)
info-provider	hostlist, port, dir	hostlist로부터의 port 번호에 의한 접속을 받아들임, dir에 있는 파일을 읽을 수 있음, 디스플레이 접근할 수 없음, 프로세스 실행할 수 없음. (e.g., fingerd, rwhod, ftpd)
server	hostlist, port, dir	hostlist로부터의 port 번호에 의한 접속을 받아들임, dir에 있는 파일을 읽을 수 있음, 디스플레이 접근할 수 없음, transformer 실행할 수 있음. (e.g., httpd)
shell	mapfile, classlist	mapfile을 읽을 수 있음, mapfile에 지정되어 있는 바이너리 코드를 실행할 수 있음, 네트워크 접근할 수 없음, 디스플레이 접근할 수 없음. (e.g., ksh, csh, tcsh)
game		디스플레이 접근할 수 있음, 네트워크 접근할 수 없음, 프로세스 실행할 수 없음.
applet	host, port	디스플레이 접근할 수 있음, port 번호로 host에 접속할 수 있음, 파일을 읽거나 쓸 수 없음, 프로세스 실행할 수 없음.

김 철 민

1997년 3월~2000년 8월 아주대학교 정보 및 컴퓨터 공학부(학사). 2000년 9월~2002년 8월 아주대학교 정보통신 전문대학원(석사). 2002년 9월~ 아주대학교 정보통신 전문대학원(박사과정)

이 성 욱

1994년 아주대학교 컴퓨터공학과 학사 1996년 아주대학교 교통공학과 공학석사 1996년~1997년 기아정보시스템 지능형 교통시스템팀. 2003년 아주대학교 컴퓨터공학과 공학박사. 2003년~현재 신구대학 인터넷정보과 전임강사. 관심분야는 컴퓨터 보안, 병렬처리

홍 만 표

1981년 서울대학교 계산통계학과 이학사 1983년 서울대학교 계산통계학과 이학석사. 1991년 서울대학교 계산통계학과 이학박사. 1983년~1985년 울산공과대학 전자계산학과 전임강사. 1985년~현재 아주대학교 정보 및 컴퓨터공학부 교수 1993년~1994년 미네소타대학 전자공학과 교환교수. 관심분야는 병렬처리