다형성 엔진으로 생성된 웜의 탐지 기법

이기훈° 이승익 최홍준 김유나 홍성제 김종
포항공과대학교 컴퓨터공학과
{ lkh8291°, roke79, formetel, existion, sjhong, jkim}@postech.ac.kr

Detection of Worm Generated by Polymorphic Engine

Kihun Lee° , Seunglck Lee, Hongjun Choi, Yuna Kim, Sung Je Hong, Jong Kim
Department of Computer Science and Engineering
Pohang University of Science and Technology (POSTECH)

요 약

다형성 웜은 새로운 감염시도 때마다 그 형태가 계속 변형되기 때문에 시그너처 기반의 탐지 기법으로는 탐지될 수 없다. 또한 이러한 다형성 웜은 주로 공개된 다형성 엔진을 이용하여 쉽게 자동적으로 생성할 수 있다. 본 연구에서는 네트워크 트래픽에 포함된 실행코드의 명령어 분포를 분석하여, 다형성 웜을 포함한 트래픽을 탐지하는 기법을 제시한다. 또한 제안하는 시스템의 성능을 모의 실험을 통해 평가한 결과, 다형성 엔진으로 생성된 웜은 전부 탐지되고 약 0.0286%의 낮은 오탐지율을 보인다.

1. 서 론

인터넷에 연결된 호스트와 서버 소프트웨어의 지나친 획일화로 인해 인터넷 웜은 불과 수십분 만에 전 세계로 퍼져 나갈 수 있게 되었다. Morris 웜[1]이 처음으로 발생하여 6천여대의 서버에 피해를 입히고 인터넷 다운을 일으킨 1988년 이후로 웜의 전파에 대한 많은 연구가 진행 되었음에도 불구하고, 계속적으로 웜이 발생하고, 다른 보안 위협에 비해 매우 큰 피해를 입히고 있다. 예를 들어, 2001년 7월 Code-Red 웜이 전 세계적으로 전파가 되어 250,000대 이상의 컴퓨터가 이 웜에 공격을 당했다. 곧이어 Nimda가 발생하여 전 세계적으로 800만대 이상을 감염시키고, 60억 달러 이상의 피해를 입혔다. 2003년 1월 발생한 SQL-Slammer는 전세계의 90%이상의 Microsoft SQL 서버를 감염시켰다. 한국은 이 웜으로 인해 전 국가적으로 인터넷이 마비되는 사건이 일어나기도 했다. 이로인해 많은 전자상거래업체, 공공기관, 일반 기업등이 피해를 입었고, 피해액은 산출하기 어려울 정도이다.

인터넷 웜에 대한 방어책으로써 NIDS[2,3] (네트워크 기반 침입탐지 시스템)가 제안되었고, 현재까지 가장 많이 활용되고 있다. NIDS는 네트워크로 들어오는 트래픽을 분석하여 특정한 패턴을 찾아낸다. 이러한 특정 패턴을 *시그너처*라고 한다. 시그너처는 특정 악성코드에서 항상 또는 매우 빈번히 발생하는 패턴으로서 이를 이용해서 특정 악성코드의 유입을 막거나 감지할 수 있다.

일반적인 시그너처는 두 가지의 가정을 바탕에 두고 생성 된다. 첫째로, 웜의 감염 시도 시 항상 나타나고 변하지 않는 부분문자열이 트래픽의 페이로드(payload)에 존재한다는 것이다. 둘째는 이러한 부분문자열이 충분히 고유하여 특정한 웜에서만 나타난다는 것이다. 이는 부분문자열이 충분히 길다는 의미를 포함하고 있다.

AVS(안티바이러스 시스템) 역시 시그너처를 이용해서 바이러스를 탐지하는 시스템으로서 AVS와 NIDS는 매우 흡사한 방식으로 동작한다. 악성코드가 발생하면 전문가들이 이를 수집하고, 분석하여 시그너처를 찾아내고, 찾아낸 시그너처를 배포하면 패턴 매칭 방법을 이용하여 악성코드를 찾아내는 방식이다. 따라서 AVS를 회피하기 위해서 사용되었던 방법은 NIDS를 회피하기 위해서도 그대로 사용이 될 수 있다.

다형성 바이러스(polymorphic virus)는 AVS를 회피하기 위해서 오래 전부터 사용된 방법이다. 이때 사용된 기술을 다형성 기법(polymorphic technique)이라 하고, 이는 자기 자신을 암호화(self-encryption)하고 복호화(self-decryption)하는 방법을 이용한다. 임의의 키를 이용하는 암호화 방법을 통해서 바이러스는 감염이 될 때마다 전혀 다른 모습을 띄게 된다. 다형성 바이러스와 마찬가지로 다형성 웜도 암호화를 통하여 새로운 감염 시도 시 보내는 트래픽마다 서로 다른 모습을 띄도록 할 수 있다. 이는 앞에서 언급했던 시그너처 탐지 기법의 가정이 성립하지 않게 만든다. 더 큰 문제는 웜에서 다형성 기법들을 더 쉽게 적용할 수 있게 하는 다형성 엔진들[4,5]이

이미 공개되어 있기 때문에 웜의 제작자들은 큰 노력 없이도 다형성 웜을 생성할 수 있다는 것이다.

본 논문에서는 웜이 다형성 기법을 적용을 하여 트래픽을 변형시키더라도 그에 영향을 받지 않고 탐지를 할 수 있는 새로운 방법을 제시한다. 우리의 기본전제는 다형성 웜은 내부에 실행가능한 코드 부분을 항상 포함하고 있다는 것이다. 이를 토대로 트래픽에서 실행코드를 찾아내고, 코드내의 명령어 분포를 분석하여 다형성 웜을 이루는 실행코드를 찾아내는 방법을 제안한다.

본 논문의 구성은 다음과 같다. 2장은 다형성 엔진의 분석을 통해 다형성 기법을 설명하고, 3장은 기존의 다형성 웜을 탐지하는 방법을 살펴보고, 문제점을 분석한다. 4장에서는 제안하는 탐지방법을 제시하고, 5장에서는 성능평가를 위한 모의 실험의 결과를 보인다. 마지막으로 6장에서 결론을 맺고 향후 연구방향을 제시한다.

2. 다형성 엔진 분석

다형성 기법은 자기 암호화 방식을 통해 스스로를 변형시키는 기술과, 명령어 치환, 쓰레기코드 삽입, 코드 블록의 위치변화와 같이 동일한 의미를 지닌 다른코드로 변환시키는 코드 모호화 기술을 포함한다. 일반적으로 다형성 엔진은 인터넷 웜의 코드 또는 쉘코드를 입력으로 받아들인다. 이 때 입력으로 주어지는 코드는 사용자에 따라서 다양하게 변화할 수 있다. 그러나 엔진이 사용하는 암호화 또는 인코딩 방식은 고정되어 있다.

따라서 다형성 엔진은 입력코드에는 코드 모호화를 적용하지 않고, 코드의 의미에 관계없이 기계적인 작업을 통해서 변화가 가능한 암호화 또는 인코딩을 통해서 변화를 시킨다. 더불어 그에 상응하는 복호화 루틴이나 디코딩 루틴은 코드의 의미를 다형성 엔진이 알고 있기 때문에 코드 모호화 기술을 적용하고 있다.

2.1. Alphanumeric 엔진

다형성 Alphanumeric 쉘코드 엔진[4]은 Matt Conover에 의해 2004년에 공개된 것으로 기존에 공개되었던 Alphanumeric 코드 기법과 쉘코드 다형성 기법을 함께 사용함으로써 향상된 성능을 제공한다.

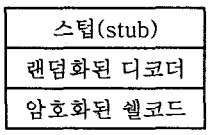

그림 1 변환후 쉘코드

인코딩된 쉘코드의 구조는 왼쪽 그림 1과 같다. 스텁은 디코더가 디코딩을 할 때 쉘코드의 주소를 가리키고 있는 레지스터를 참조하기 위해서 사용되고 스텁의 길이는 주소참조 방법에 따라 달라질수 있다. 디코더는 암호화된 쉘코드를 원래 모양으로 디코딩하는 역할을 수행하고 이때 사용되는 키는 디코더 내부에 포함되어 있다. 디코더 역시 출력가능한 alphanumeric으로만 만들어진 어셈블리 명령을 사용하여 코딩되어 있다. 이 엔진은 랜덤키와 함께 출력가능한 alphanumeric 코드로 쉘코드를 암호화 한다.

```
DecodedByte = EncodedByte1 * Key
DecodedByte ^= EncodedByte2
```

그림 2 암호화 인코딩

원본 쉘코드의 한 바이트는 암호화된 후 두개의 바이트로 인코딩이 된다. 그림 2와 같이 첫번째 바이트에 키를 곱하고 그 결과에 두번째 바이트를 XOR한다. 이때 나온 결과가 원래 쉘코드의 바이트와 같게 됐을 때, 그때 사용된 두 개의 바이트가 인코딩된 바이트가 된다. 원래 쉘코드의 한 바이트를 암호화하기 위한 바이트쌍은 여러 개 존재한다. 랜덤키를 이용해 인코딩 전에 모든 alphanumeric 문자에 대해서 가능한 인코딩수를 저장해 두고 그 수 범위에 대해서 랜덤값을 이용해서 암호화하기 위한 바이트쌍을 선택함으로써 매번 인코딩할 때마다 다른 결과를 얻을 수 있다. 이때 쉘코드의 길이는 두배로 증가하게 되고 디코더가 쉘코드의 끝을 알기 위해서 종결문자를 추가한다. 디코더는 모든 어셈블리 명령이 ASCII 문자범위(33h~ 7eh)에 포함되는 것들로만 작성이 되어 있다. 출력가능한 alphanumeric으로 표현된 디코더의 한가지 예는 " Zh!!!!X5!!!!H4C0B6RYkA7@A3A7A2B70B7Bh!!!!X5!!!!4_8A7ub" 와 같다.

특정 서버 소프트웨어는 입력값으로 출력가능한 문자만을 받아들이고, 만약 출력 불가능한 문자가 포함되어 있을 경우 이를 입력 버퍼에 삽입시키지 않음으로써 공격을 방지한다. 이런 방법을 회피하기 위해 위와 같이 모든 명령을 출력가능한 코드로 변형하는 방법을 사용한다. 하지만 이 방법만으로는 디코더의 고정된 코드를 기반으로 하는 시그너처 탐지방법을 회피할 수가 없다. 이를 위해서 이 엔진은 alphanumeric 디코더에 다형성 기법을 적용하였다.

alphanumeric으로 표현된 위의 디코더에서 보면 명령 사이에는 특수문자(!, @, _)들이 삽입되어 있어 있는데, 이것은 디코더가 수행되는 동안 결과에 영향을 주지않고 임의의 값으로 변경이 가능한 값들이다. 특히 ' @' 은 키로 변형이 되고, ' _' 은 쉘코드의 끝을 나타내는 종결자로 변형이 되고, 쉘코드 끝부분에 첨가된다. 이 값들은 인코딩할 때마다 임의의 값으로 변경이 가능하다. 따라서 디코더 전체에 대한 시그너처를 생성하기가 어렵게 된다.

특수문자를 이용한 다형성 방법을 이용하더라도 디코더에는 여전히 " H4C0B6RYkA7" , " A3A7A2B70B7Bh" 와 같은 비교적 긴 고정된 문자열이 포함되어 있다. 이를 해결하기 위해서 엔진은 추가적인 다형성 기법을 제공한다. 임의의 위치에 의미없는 즉, 디코더가 수행하는데 결과에 영향을 주지 않는 명령어 집합을 추가할수 있게 해준다. IA-32 alphanumeric NOP은 PUSH, POP, INC 그리고 DEC을 사용할수 있다. 위의 명령어 조합을 이용해서 의미없는 명령어 집합을 만들어서 삽입이 가능한다. 이 방법을 이용하면 위의 고정된 문자열의 길이를 줄일수 있고 고정된 길이가 너무 짧기 때문에 시그너처 생성이 어렵게 된다.

위에서 살펴본것과 같이 쉘코드에 엔진을 적용할 때 마다 다른 값들이 생성되고 디코더의 고정된 부분 또한 짧

길이로 줄일수 있기 때문에 시그너처 생성에 의한 탐지로는 변환된 쉘코드를 탐지할 수 없다.

2.2. ADMmutate

ADMmutate 엔진[5]은 다형성 쉘코드 엔진으로 현재 0.8.4 버전까지 공개되었다. 이 엔진은 다형성 기법을 사용하여 입력 쉘코드에 대한 출력을 키에 따라 랜덤하게 변화 시킴으로써 시그니처를 생성하기 어렵게 한다. 생성되는 출력은 그림 3 과 같은 구조이다.

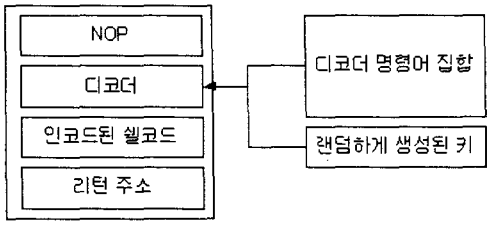

그림 3 출력의 구조

ADMmutate 엔진에서 정의된 NOP 명령어는 최소 1바이트부터 최대 3바이트의 명령어들로 구성되어 있지만기본적으로 1바이트 명령어를 사용하는 것을 권장하고 있다. 이는 리턴 주소가 ADMmutate 엔진에서 정의된 3바이트 명령어들의 중간으로 점프할 경우 발생할 수 있는 예외 상황 때문이다. 그리고 이 엔진은 그림 3 과 같이 키를 사용한 인코딩 방법을 사용하여 키에 따라 인코드된 쉘 코드가 랜덤하게 변하기 때문에 인코드된 쉘 코드를 기반으로 시그니처를 생성할 수 없다. 키를 사용한 인코딩 알고리즘은 그림 4 와 같다.

```
int i = start address of shellcode;
while ( shellcode != end ) {
shellcode [i] ^= (KEY >> 24) & 0xFF;
shellcode [i+1] ^= (KEY >> 16) & 0xFF;
shellcode [i+2] ^= (KEY >> 8) & 0xFF;
shellcode [i+3] ^= (KEY >> 0) & 0xFF;
}
```

그림 4 ADMmutate 엔진의 인코딩 알고리즘

ADMmutate 엔진은 디코더가 여러 개의 패턴을 가질 수 있도록 하는 코드 모호화 기술을 지원하고 있다. 코드 모호화 기술은 ADMmutate 엔진에서 사용하고 있는 디코더가 여러 개의 패턴을 가질 수 있도록 하는 과정이다. ADMmuate 엔진 코드 내에 정의된 바에 따르면 인텔 아키텍처 기반의 디코더가 가질 수 있는 이론적인 패턴의 수는 약 1만 5천개이고 새로운 인스턴스를 생성할 때마다 사용할 디코더를 임의로 결정하기 때문에 디코더에서도 시그니처를 생성할 수 없다.

ADMmutate 엔진을 사용하는 웜은 쉘코드와 디코더가 하나의 패턴이 아닌 다수의 패턴을 가질 수 있기 때문에 시그너처를 기반으로 하여 웜을 검출하는 방법을 사용할 수 없게 된다.

3. 관련 연구

다형성 웜을 탐지하는 방법으로 두가지의 기존연구가 존재한다.

3.1 Polygraph

시그너처 기반의 NIDS에서 다형성 웜의 문제 해결을 처음으로 제안한 연구가 Polygraph[6]이다. Polygraph는 다형성 엔진이 웜의 코드를 변화 시킬 때 웜의 특성상 변해서는 안되는 부분이 존재한다는 사실을 이용한다. 저자들은 이 변하지 않는 부분들이 너무 짧아서 기존의 단일한 부분문자열을 이용하는 방법으로는 시그너처로 사용할 수 없음을 지적하고, 새로운 형태의 시그너처를 제안한다. 즉, 전통적인 시그너처와는 달리 단일한 부분문자열 대신에 여러 개의 부분문자열을 조합하여 시그너처를 생성한다. 기본 아이디어는 시그너처를 이루는 각각의 부분문자열들은 너무 짧고 일반적이지만 이들을 모두 합치면 충분히 고유한 문자열을 얻을 수 있다는 것이다. 저자들은 실제로 다형성 웜에서 변하지 않는 부분들이 여러 부분 존재함을 보이고, 이를 조합한 시그너처 생성 방법을 제안한다.

하지만 실제로 생성이 되는 시그너처를 분석해보면 여러 부분문자열 중에서 실제로 탐지에 영향을 주는 부분문자열은 단 하나뿐이다. 일반적으로 버퍼오버플로우 공격의 리턴주소가 이에 해당이 된다. 따라서 리턴주소를 포함하지 않거나 일반적인 문자열과 대조가 되지 않는 리턴주소를 사용한다면 시그너처를 제대로 사용할 수가 없다. 또한 시그너처 매칭을 하기 위해서는 변하지 않는 부분문자열들이 모두 나타날 때까지 기다려야한다. 만약 공격자가 이런 부분을 의도적으로 트래픽의 뒷편에 위치 시키게 되면 탐지를 하는 시점에서 이미 내부의 호스트가 감염이 될 수도 있다.

3.2 제어 흐름 그래프 탐지방법

다형성 웜을 탐지하기 위한 다른 방법으로는 Kruegel *et al.* 이 제안한 제어 흐름 그래프(control flow graph)를 이용하는 방법이 있다[7]. 이 시스템은 모든 웜이 실행코드 부분을 포함하고 있다는 가정에서 시작한다. 다형성 웜의 경우 복호화 코드가 이에 해당이 된다. 이 시스템은 우선 네트워크 트래픽에서 제어 흐름 그래프를 추출해낸다. 그리고 비정상적일 정도로 많이 생성이 되는 그래프를 웜의 트래픽을 나타내는 그래프로 판단하고 시그너처로 사용을 하게 된다. 이는 복호화 코드의 제어 흐름 그래프를 변화 시키기 힘들 것이라는 가정을 두고 있다.

하지만 실제로 제어 흐름 그래프를 변화 시키는 것은 그다지 어려운 기술이 아니다. 또한 제어 흐름 그래프를 추출 하는 작업은 너무 오버헤드가 커서 탐지 시스템이 실시간으로 동작할 수 없다는 단점이 있다.

4. 제안하는 시스템

다형성 웜은 항상 실행가능한 코드 부분을 포함한다. 복호화 루틴 또는 디코더 루틴 부분이 이에 해당하며, 이 부분은 일반적인 실행코드와 구분되는 특징을 갖는다. 본 연구는 두 단계로 이루어진다. 먼저 데이터와 실행가능한 코드를 구분짓는 요소와 다형성 웜에 포함된 실행가능한 코드와 일반적인 실행코드를 구분짓는 요소를 찾아낸다. 이를 기반하여, 네트워크 트래픽에서 실행가능한 코드를 찾아내고, 이중에서 다형성 웜의 실행코드를 분류해낸다.

4.1. 구분 요소 추출

데이터와 실행코드 간의 구분, 일반 실행코드와 다형성 웜의 실행코드 간의 구분을 할 수 있는 특성을 찾는다.

4.1.1 데이터와 실행코드의 구분

데이터와 실행코드의 특성을 파악하기 위해서 각각을 디스어셈블했을 때의 결과를 비교하는 실험을 하였다. 우리는 실행파일과 데이터파일(그림, 음악, 동영상, 텍스트 등)을 동일한 길이로 나눈후, 각 부분을 이용하여 디스어셈블 하였다. 디스어셈블 한 결과의 명령어(instruction) 시퀀스에서 같은 종류의 명령어끼리 분류를 한 뒤, 각각 발생 빈도를 계수하였다. 그리고 발생 빈도에 따라서 순위를 부여하였다. 그림 5는 실험한 결과를 나타낸 그래프이다. 그래프의 x축은 순위를 나타내고, y 축은 빈도를 나타낸다. 그래프는 순위의 하락에 따른 명령어 빈도수의 변화를 나타낸다. 그래프에서 선 하나는 실험 데이터 하나에 해당한다.

이 그래프에서 실행코드의 특성을 알 수 있다. 우선 실행 코드는 발생하는 명령어의 종류가 데이터에 비해 적다. 이는 실행코드에 대한 그래프들이 데이터에 대한 그래프들보다 왼쪽으로 치우쳐져 있음을 보면 알 수 있다. 또 다른 특징은 실행코드의 경우 순위 하락에 따른 명령어 빈도수의 변화가 더 크다는 것이다. 이는 그래프에서 나타나는 실행코드의 기울기가 더 가파른 점을 통해 알 수 있다.

그래프에는 나타나지 않지만 실험을 통해서 추가적으로 알 수 있었던 사실이 있다. 인텔 아키텍쳐의 경우 특수한 바이트인 *prefix*가 존재한다. *Prefix* 바이트가 명령어의 맨 처음에 나오게 되면 그 명령어에 한해서만 세그먼트 레지스터를 변화시키거나, 피연산자(operand)의 길이를 다르게 해석을 하게 된다. 이는 하향 호환성을 위해서 존재하는 기능으로서 실제 실행코드에서는 잘 사용이 되지 않는다.

Prefix 바이트는 각각 16진수 값 *26h, 2Eh, 36h, 3Eh, 64h, 65h, 66h, 67h* 로 정의 되어있다. 이들 중 36h, 64h, 65h, 66h, 67h는 각각 ASCII 코드 ' 6' , ' d' , ' e' , ' f' , ' g' 에 해당한다. 이는 텍스트 파일을 디스어셈블 하게 되면 비정상적으로 많은 양의 *prefix* 바이트가 발생하는것을 의미한다.

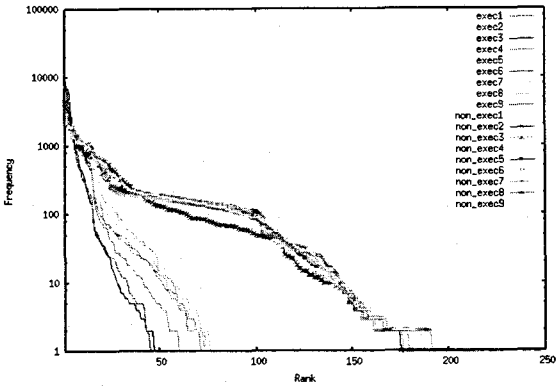

그림 5 데이터와 실행코드의 디스어셈블 결과 비교

4.1.2 일반 실행코드와 다형성 웜의 실행 코드의 구분

디스어셈블한 결과를 이용해 일반 실행코드와 다형성 웜의 실행 코드를 구분하기 위해서는 일반 실행코드에서는 자주 사용되지만 다형성 웜에서는 사용될 수 없는 명령어를 이용하면 된다. 우리는 이런 조건을 만족 시키는 명령어를 *검증명령어*라고 정의를 내리고, 이를 만족시키는 명령어를 찾았다.

다형성 웜의 실행코드는 복호화 코드가 대표적이다. 복호화 코드는 암호화 된 부분을 복호화 하고, 복호화된 코드에 통제권(control)을 넘겨주는 역할을 한다. 따라서 복호화 코드는 자신이 넘겨줘야할 통제권을 복호화가 끝날 때까지 소유하고 있어야 한다. 만약 통제권을 외부로 넘기게 되면 다시 돌려받을 수 있다는 보장이 없기 때문이다. 왜냐하면 복호화 코드는 외부의 메모리 환경을 알 수 없기 때문에, 통제권을 잘못 넘기게 되면 세그먼테이션 오류를 발생 시킬 수 있기 때문이다.

그러나 일반 실행코드는 많은 루틴들을 포함하고 공유 라이브러리를 이용하기 때문에, 통제권을 옮기는 역할을 하는 명령어가 자주 사용된다. 그 중에서 *call* 명령어가 실행코드 전체에 걸쳐서 고르게 분산이 되어 있다. 이는 실행코드의 어느 부분이 패킷에 포함되더라도 *call*이 비슷하게 포함이 됨을 의미한다. 실험의 결과 평균 10~15개의 명령어당 하나의 *call* 명령어가 포함 되어 있었다.

Call 이외에도 통제권을 이동 시키는 명령어가 다수 있으나 *call*은 스택의 상태를 변화 시키고, 피연산자의 길이가 길어서 가까운 곳으로 이동하기 위해서는 null 바이트가 포함되는등의 추가 제약 사항이 존재하기 때문에 이번 연구에서는 call을 *검증명령어*로 사용하였다.

4.2. 다형성 웜의 실행코드 탐지

앞 절에서 추출한 구분 요소를 이용하여, 네트워크 트래픽에서 실행코드가 포함된 트래픽을 분류하고, 이들 중에서 다형성 웜의 실행코드 포함 여부를 검사한다.

4.2.1 실행코드 포함 여부 검사

4.1.1절에서 밝혀낸 특성을 이용하여 실행코드를 포함한 패킷을 찾아낼 수 있다. 우선 네트워크로 들어오는 패킷에 대하여 디스어셈블을 수행한다. 디스어셈블한 결과에서 발생한 명령어의 총 개수를 T, 그 중에서 서로 다른 명령어의 종류의 수를 D, 발생한 prefix의 개수를 P_0라 하자. 디스어셈블한 결과에 대하여 다음과 같은 식을 계산한다.

$$S_1 = T / (D * P) \quad (1)$$

if $P_0 > 0$

$P =$ 1 if number of white space $< Th_w$

P_0 otherwise

if $P_0 = 0$

$P =$ 1

식 (1)에서 T/D는 그림 5 그래프의 기울기의 절대값에 정비례한다. 따라서 데이터 패킷의 경우 T/D가 실행코드에 비해서 작은 값을 가진다. 또한 텍스트 파일의 경우 P가 크게 나타나기 때문에 더욱 작은 값을 갖는다. 하지만 alphanumeric 코드의 경우 텍스트와 마찬가지로 prefix가 많이 나타난다. 따라서 일반 텍스트와 alphanumeric 코드를 구분하기 위해서 공백문자를 계수하여 공백문자가 나타나지 않으면 alphanumeric코드로 간주하여 prefix의 영향을 제거한다. 이는 alphanumeric 엔진이 공백문자를 사용하지 않는다는 특징에 기인한다.

결과적으로 데이터 패킷은 실행코드를 포함한 패킷에 비해서 훨씬 낮은 S_1 값을 갖게 되고, 둘 사이를 구분할 수 있는 한계값 Th_1를 설정할 수 있다.

4.2.2 다형성 웜의 실행코드 포함 여부 검사

4.2.1절의 과정을 통해서 실행코드가 포함된 패킷을 찾아낼 수 있다. 포함되어있는 실행코드가 일반적인 실행코드인지 다형성 웜의 코드인지를 밝혀내면 된다. 이 단계에서는 *검증명령어*의 포함에 따라서 둘 사이를 구분 한다.

C를 패킷에 포함된 *검증명령어*의 개수라고 하자. 실행코드를 포함한 패킷에 대하여 다음 식을 계산한다.

$$S_2 = S_1 / C \quad (2)$$

일반 실행코드는 *검증명령어*를 많이 포함하고 있기 때문에 S_2의 값이 다형성 웜의 코드에 비해서 낮게 나타난다. 둘 사이를 구분할 수 있는 한계값 Th_2를 설정하면, 결과적으로 식(2) 에서 S_2가 Th_2보다 크면 다형성 웜으로 탐지된다.

마지막으로 다형성 웜으로 의심되는 패킷에 대하여 확인절차를 거치게 된다. 이 과정은 false positive를 줄이기 위해서 고안이 되었다. 이 과정에서는 다형성 웜으로 판단한 웜에 대해서 실제로 공격코드로서 동작할 수 있는 코드인지를 간단히 검사하게 된다.

공격코드가 제대로 동작하기 위해서는 내부에 null 바이트가 존재하지 않아야 함은 잘 알려진 사실이다[8]. 따라서 null 바이트의 분포를 검사함으로써 false positive를 줄일 수가 있다.

5. 모의실험

제안하는 시스템의 성능을 모의 실험을 통하여 평가하였다. 모의실험은 일반데이터 패킷의 집합과 일반 실행코드를 포함한 패킷의 집합, 다형성 웜을 포함한 패킷의 집합을 이용하여 각각이 얼마나 잘 구분이 되는지 평가한다. 또한 Th_1 와 Th_2 의 값 각각 1.0과 1.4로 설정하여 false alarm의 비율을 측정하였다. 각각의 실험 데이터는 다음과 같다.

a. 일반데이터 집합 : 일상적인 인터넷 사용으로 인해 발생한 tcp 트래픽. Web page, mail message, ftpdata, ssh trffic, DNS Query등을 포함한다. 20,000개의 패킷으로 이루어져있다.

b. 일반 실행코드 집합 : ftp서비스를 이용하여 Linux의 /bin, /sbin 디렉토리의 유틸리티들을 전송하고, capture한 패킷들. 13,331개의 패킷으로 이루어져있다.

c. 다형성 웜의 집합 : ADMmutate와 Alphanumeric 엔진을 이용해 생성한 imapd의 취약점을 이용하는 웜코드. 203개의 패킷으로 이루어져있다.

각각의 실험 데이터 집합에 대하여 개별적으로 실험을 진행하였다. 각 데이터 집합은 실험시 리플레이가 되고, 이 패킷은 디스어셈블 되기 전에 페이로드의 길이에 의해서 필터링 과정을 거친다.

5.1 모의실험 결과

그림 6는 일반데이터 집합에 대한 실험 결과이다. 일반데이터는 실행코드를 포함하지 않기 때문에 대부분 Th_1을 넘지 못한다. 또한 다형성 웜도 포함하지 않기 때문에 Th_2도 넘지 못한다. 따라서 그래프에서 Th_2를 넘은 패킷은 false positive로 볼 수 있다.

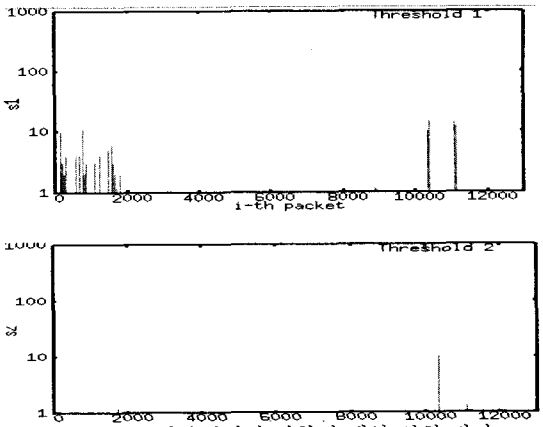

그림 6 일반데이터 집합에 대한 실험 결과

그림 7은 일반 실행코드에 대한 실험 결과이다. 이는 실행코드가 대부분이기 때문에 거의 모든 패킷들이 Th_1을 넘는다. 하지만 다형성 웜은 포함하고 있지 않기 때문에 대부분이 Th_2를 넘지 못한다. 마찬가지로 Th_2를 넘은 패킷은 모두 false positive이다.

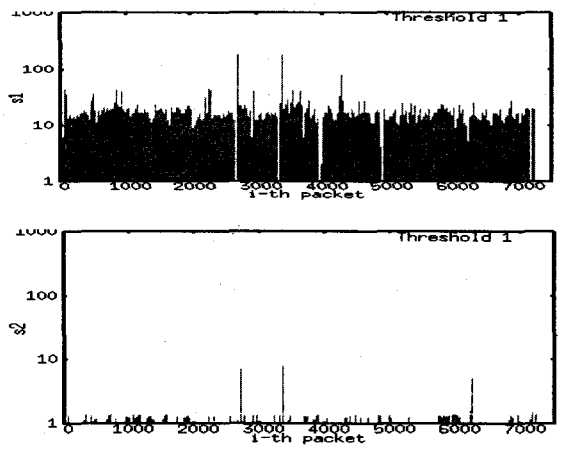

그림 7 일반실행파일 집합에 대한 실험 결과

그림 8는 다형성 웜에 대한 실험 결과이다. 이 집합은 모두 Th_1과 Th_2을 넘게 된다.

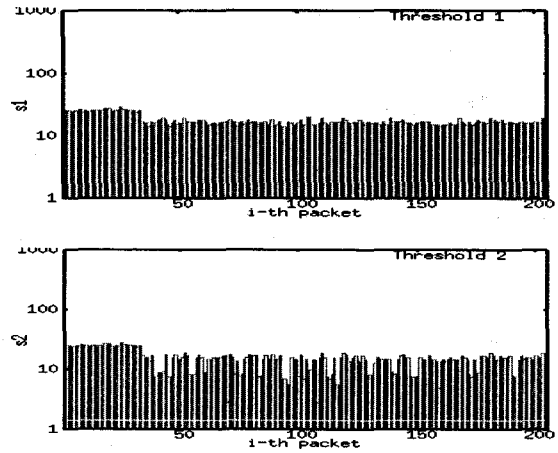

그림 8 다형성 웜 집합에 대한 실험 결과

5.2 모의실험 결과 분석

세종류의 데이터 집합으로 실험하여 총 212개의 패킷이 다형성 웜으로 탐지 되었다. 일반데이터 집합에서 1개, 일반실행파일 집합에서 8개, 다형성 웜 집합에서 203개가 발생하였다. 이로서 false negative rate는 0이라는 사실은 쉽게 알 수 있다. 이 때 false positive rate를 구해보면 다음과 같다.

False positive rate = (1+8) / (20,000 + 13,331 + 203)
≈ 0.000268 = 0.0286%

위처럼 약 0.0286%로 낮은 false positive rate 를 보인다.

6. 결론

본 논문에서는 다형성 웜이 포함하는 실행코드를 탐지하기 위해 일반 트래픽과의 차이를 나타내는 구분 요소를 실험을 통해서 추출했고, 이 구분요소를 이용하여 다형성 웜을 찾아낼 수 있는 방법을 제안하였다. 제안하는 시스템은 패킷의 명령어 분포 특징을 수식으로 나타내어 각 계산값이 한계값보다 크면, 다형성 웜코드의 패킷임을 나타낸다.

성능 평가의 결과는 제안한 방법이 효과적으로 다형성 웜을 탐지해낼 수 있다는 것을 보였다. 본 연구는 패턴 매칭에 기인한 방법이 아니므로 다형성 엔진에 의한 변화에 강하다. 또한 공격코드의 실행가능 부분 즉, 복호화 코드 부분을 탐지해냄으로써 그 이전에 네트워크로 흘러간 패킷들을 무력화 시킨다. 결국 공격이 실패하도록 만든다.

향후 연구는 성능분석 시 찾아낸 false positive를 제거하는 방법 연구와 alphanumeric 엔진에서 공백문자를 사용할 경우에 대한 대처방안을 중심으로 이루어질 것이다

참고 문헌

[1] J. Rochlis and M. Eichin, " With microscope and tweezers: The worm from MIT' s perspective" , Communication of the ACM, vol. 32, no. 6, 689-698, June 1989.

[2] T. S. Project, " Snort, the open-source network intrusion detection system." Available at http://www.snort.org

[3] V. Paxon, " Bro: a system for detecting network intruders in real-time" , Elsevier Computer Networks, vol. 31, no. 23-24, 2435-2463, 1998.

[4] M. Conover, " Polymorphic alphanumeric IA-32/IA-32e shellcode" available at http://www.cybertech.net/~sh0ksh0k/projects/enc2alnum/, July 2004

[5] S. Macaulay, " ADMmutate: Polymorphic shellcode generator" , Available at http://www.ktwo.ca/security.html, 2003.

[6] J. Newsome, B. Karp and D. Song, " Polygraph: automatically generating signatures for polymorphic worms" , IEEE Symposium on Security and Privacy, May, 2005.

[7] C. Kruegel, E. Kirda, D. Mutz, W. Robertson and G.Vigna, " Polymorphic worm detection using structural information of executables" , 8th International Symposium on RAID, 2005.

[8] A. One, " Smashing the stack for fun and profit" , Available at http://www.phrack.org/show.php?p=49&a=14, 1996.