코드 삽입 기법을 이용한 알려지지 않은 악성 스크립트 탐지

(Detection Of Unknown Malicious Scripts using Code Insertion Technique)

이 성 욱 † 방 효 찬 †† 홍 만 표 †††
(Seong-uck Lee) (Hyo-chan Bang) (Man Pyo Hong)

요 약 서버 수준의 안티바이러스는 특정 도메인 내에 진입하는 악성코드를 진입점에서 감지하므로 모든 클라이언트를 완벽하게 통제하기 어려운 실제 상황에서 전자우편 서버 등에 유용하게 사용된다. 그러나, 알려지지 않은 악성 코드에 감지에 유용한 행위 감시 기법은 서버에 적용이 어려우므로, 현재의 서버용 안티바이러스들은 이미 알려진 악성 코드에 대한 시그너쳐 기반의 감지, 단순한 필터링 그리고 화일명 변경과 같은 기능만을 수행한다. 본 논문에서는 서버에서의 실행만으로 별도의 안티바이러스가 탑재되지 않은 클라이언트에서도 지속적인 행위 감시가 가능하도록 하는 악성 스크립트 감지 기법을 제안하고 그 구현에 관해 기술한다.

키워드 : 컴퓨터 바이러스, 악성 코드, 스크립트, 코드 변환, 규칙

Abstract Server-side anti-viruses are useful to protect their domains, because they can detect malicious codes at the gateway of their domains. In prevailing local network, all clients cannot be perfectly controlled by domain administrators, so server-side inspection, for example in e-mail server, is used as an efficient technique of detecting mobile malicious codes. However, current server-side anti-virus systems perform only signature-based detection for known malicious codes, simple filtering, and file name modification. One of the main reasons that they don't have detection features, for unknown malicious codes, is that activity monitoring technique is unavailable for server machines. In this paper, we propose a detection technique that is executed at the server, but it can monitor activities at the clients without any anti-virus features. we describe its implementation.

Key words : computer virus, malicious code, script, code transformation, rule

1. 서 론

악성 코드(malicious code)는 비정상적인 동작 또는 시스템 위해(harm) 행위를 목적으로 작성된 코드를 말하며, 컴퓨터 바이러스(computer virus), 웜(worm), 그리고 트로이 목마(trojan)를 포함하는 개념이다[1]. 과거에는 이진(binary) 형태로 존재하는 악성 코드가 주류를 이루었으나, 1994년 12월의 최초 출현 이후 스크립트 형태의 악성 코드가 꾸준히 증가하여 보편적인 악성 코드의 한 형태로 간주되고 있다[2].

악성 스크립트는 스크립트 언어로 작성된 악성 프로그램들을 말한다. 현재까지 발견된 것들은 비주얼 베이직 스크립트(Visual Basic Script), mIRC 스크립트, 자바 스크립트가 수적으로 가장 많으며, 그 외에 PHP 스크립트, 코렐 드로우 스크립트 등으로 작성된 것들이 일부 존재한다[3]. 현재 가장 많이 만들어지고 큰 피해를 입히고 있는 악성 스크립트의 상당수는 비주얼 베이직 스크립트로 작성된 것들이며 이러한 악성 스크립트를 자동 생성하는 악성 스크립트 생성기까지 나와 있는 실

· 이 논문은 한국전자통신연구원 위탁연구에 의해 지원되었음
† 학생회원 : 아주대학교 컴퓨터공학과
suleeip@yahoo.co.kr
†† 비 회 원 : 한국전자통신연구원 능동보안기술연구팀 연구원
bangs@etri.re.kr
††† 종신회원 : 아주대학교 정보및컴퓨터공학부 교수
mphong@ajou.ac.kr
논문접수 : 2002년 4월 22일
심사완료 : 2002년 7월 10일

정이다.

이러한 악성 스크립트의 감지에는 이진 형태의 악성 코드와 마찬가지로 시그너쳐(signature) 기반의 스캐닝(scanning)을 통한 방법이 보편적으로 사용되고 있다. 이 기법은 사전에 면밀한 분석을 통해 시그너쳐를 추출한 악성 코드만을 감지할 수 있으므로, 알려지지 않은 새로운 악성 스크립트의 감지에는 휴리스틱(heuristic) 스캐닝, 정적 분석, 행위 감시 기법 등이 사용된다. 특히, 실행 전 분석을 통해 악성 코드 유무를 판별하는 휴리스틱 스캐닝 또는 정적 분석 기법에 비해, 실행 중 동적으로 변화하는 모든 자료를 얻어낼 수 있는 행위 감시 기법은 상대적으로 높은 정확도를 가질 수 있으므로 점차 많은 안티바이러스들이 이를 채용하고 있다.

한편, 상술된 일반적인 구분 외에도 전자우편과 같이 특정 도메인 내에 진입하는 모든 자료가 하나의 서버를 거치는 서비스에서는, 안티바이러스 시스템의 물리적인 설치 위치에 따라 서버측 대응 기법과 클라이언트 상에서의 대응 기법으로 바이러스 대응 기법을 분류할 수 있다. 서버 수준의 안티바이러스는 특정 도메인 내에 진입하는 악성코드를 진입점에서 차단하므로 모든 클라이언트를 완벽하게 통제하기 어려운 실제 상황에서 전자우편 서버 등에 유용하게 사용된다. 그러나, 서버에서의 악성 코드 감지를 위한 별도의 기법이 존재하지는 않으며, 이미 알려진 대응 기법을 서버에서의 동작에 적절하도록 약간의 수정을 가하여 이용하는 것이 보편적이다.

따라서, 현재 전자우편 서버 등에 탑재되는 서버용 안티바이러스는 시그너쳐 기반의 스캐닝을 기반으로 동작하며, 이에 필터링 또는 화일명 변환을 통해 그 확산을 늦추려는 소극적인 형태의 기능을 추가한 것이 대부분이다. 특히, 각각의 클라이언트에 설치된 감시 도구를 기반으로 동작하는 행위 감시 기법은 서버에 사용할 수 없으며, 에뮬레이션을 통한 가상 환경에서의 실행은 가능하나 서버에 많은 부담을 주므로 현실적으로 사용이 어렵다는 문제를 가지고 있다.

본 논문에서는 서버에서의 실행만으로 클라이언트 상에서도 지속적인 행위 감시를 수행할 수 있는 악성 스크립트 감지 기법을 제안하고 그 구현을 제시한다. 제안하는 기법은 억세스 제어(Access Control)를 위해 제안된 기존의 어플리케이션 변환(Application Transformation) 기법에 착안한 것으로, 스크립트 코드에 자체 진단 코드를 삽입함으로써 해당 스크립트가 실행될 때 외부의 도움 없이 자신의 악성 여부를 판단할 수 있도록 한 것이다. 이 기법은 악성 행위에 사용되는 메소드 호출 시퀀스를 감지한다는 점에서는 정적 분석과 유사하지만, 감지 루틴이 스크립트 실행 중에 동작하므로 동적으로 결정되는 리턴 값과 파라미터까지 검사할 수 있어 높은 감지 정확도를 가지게 된다. 또한, 외부로부터 유입된 스크립트에만 코드가 삽입되므로 다른 프로세스의 수행에까지 불필요한 오버헤드를 발생시키지 않으며, 일단 변형된 코드는 별도의 안티바이러스가 설치되지 않은 시스템에서도 자체 감지를 수행하므로 전파 억제 효과 또한 얻을 수 있게 된다.

본 논문의 2장에서는 기존의 관련 연구들을 제시하고, 3장에서 제안하는 기법과 상세 알고리즘, 그리고 구현에 대해서 기술한다. 4장에서는 실제 악성 스크립트 샘플에 이를 적용한 실험 결과를 제시하고, 5장에는 결론과 향후 연구를 기술한다.

2. 기존 감지 기법 및 관련 연구

2.1 기존의 악성 스크립트 감지 기법

이론적으로 볼 때, 모든 바이러스를 완전하게 감지할 수 있는 악성 코드 감지 알고리즘은 존재하지 않음이 증명되어 있다[4-6]. 따라서, 근본적으로 모든 악성 코드 감지 기법은 휴리스틱을 이용하여 악성 코드 여부를 판단하며, 단지 이용하는 휴리스틱의 유형과 그것을 이용하는 방식에 따른 차이를 가질 뿐이다.

악성 스크립트들의 감지에 가장 보편적으로 사용되는 방법은 시그너쳐 기반의 스캐닝이다. 이 방식은 특정 악성 코드에만 존재하는 특별한 문자열을 데이타베이스화하고 이 문자열의 존재를 탐색함으로써 해당 스크립트의 악성 여부를 진단하므로, 진단 속도가 빠르고 그 대상이 어떠한 악성 코드인지를 명확하게 구분할 수 있다는 장점을 가지고 있다[7]. 그러나, 이러한 방식은 시그너쳐의 추출 작업이 사람에 의해 이루어져야 한다는 단점 또한 가지고 있다. 즉, 알려지지 않은 악성 스크립트에 대해서는 전혀 대응할 수 없으므로, 안티바이러스 업체에서 새로운 악성 코드 데이타베이스를 배포하기 전까지 많은 사용자들이 악성 스크립트에 그대로 노출될 수밖에 없다. 특히, 악성 스크립트들은 대부분 전자우편과 IRC, 네트웍 공유 등을 통해 주로 전파되므로 전파 속도가 빨라 그 피해가 큰 것이 현실이다[8].

이러한 단점을 보완하기 위해 흔히 사용되고 있는 방법이 휴리스틱 스캐닝과 행위 감시 기법이다. 이 기법들은 새로운 악성 코드의 출현은 매우 빈번하게 일어나나, 새로운 악성 행위 패턴의 출현은 상대적으로 긴 시간 간격을 두고 일어난다는 점에 착안한 것이다[9]. 이는 대부분의 악성 코드 작성자들이 이미 알려진 악성 행위 기법을 사용하여 새로운 악성 코드를 작성하며, 완전히

새로운 악성 행위 기법이 만들어지는 것은 극히 일부의 악성 코드 작성자들에 의해 이루어질 뿐 아니라 많은 시간을 필요로 한다는 데에 기인한다. 따라서, 이미 알려진 악성코드에서 사용하는 악성 행위 패턴을 추출하고 이를 탐색함으로써 알려진 악성 행위를 일으키는 새로운 악성 코드에 대응할 수 있다.

휴리스틱 스캐닝이란 악성 행위를 위해 자주 사용되는 메소드(method) 또는 내장 함수(intrinsic function) 호출들을 데이타베이스화 하여두고 대상 스크립트를 스캔하여 일정 수 이상의 위험한 호출이 나타나면 이것을 악성 스크립트로 간주하는 방식이다[10]. 이 방식은 속도가 비교적 빠르고 높은 감지율을 보이긴 하지만 악성이 아닌 선의의(legitimate) 스크립트를 악성으로 감지하는 긍정 오류(false positive)가 상당히 높다는 큰 단점을 가지고 있다. 따라서 최근에는 이런 단점을 극복하기 위해 악성 행위의 메소드 호출 시퀀스(sequence)를 정의하고 이러한 시퀀스 내에 속한 메소드 호출들의 리턴 값과 파라미터간의 관계까지 검사하여 정확한 악성 행위를 감지하는 방법이 제안되기도 하였다[2]. 이 방식은 감지 정확도는 높으나 실행 시간 이전의 정적 분석에 의존하므로 프로그램 실행 전에 결정되지 않는 값들이 존재하면 악성 여부를 사전에 정확히 판단할 수 없다는 단점을 가지고 있다.

행위 감시 기법은 프로그램 수행에 필요한 시스템 호출들을 가로채어 감시하다가 악성행위로 판단되는 시스템 콜의 시퀀스가 나타나면 해당 프로그램을 악성 코드로 간주하는 감지 방식이다[9]. 이 방식은 실행 시간 중에 감지를 행하므로 해당 코드의 정확한 수행 경로 추적이 가능하고 관련된 동적 데이타를 이용할 수 있다는 장점이 있다. 시스템 감시와 유사한 방법으로 에뮬레이션(emulation)이 사용되기도 하나, 스크립트 코드를 위한 에뮬레이터는 하드웨어, 운영체제뿐 아니라 관련된 시스템 객체(object) 및 제반환경을 모두 포함하여야 하므로 구현이 매우 어렵고, 부하 또한 큰 것으로 알려져 있다[11].

2.2 어플리케이션 변환 기법

어플리케이션 변환 기법은 본래 코드 안전(code safety)을 위해 제안된 것으로, 실행 시 안전을 확신할 수 없는 코드가 주어지면 사전에 정의된 정책(policy)을 강행할 수 있는 형태로 해당 코드를 변환하는 기법이다. 따라서, 변환이 완료된 코드를 실행하면 각각의 API가 호출될 때마다 해당 API 호출로 인해 접근하게 되는 시스템 자원이 허가되어 있는가를 검사한 후 원래의 작업을 수행하게 된다. 이를 위한 아키텍처는 <그림 1>과 같다[12].

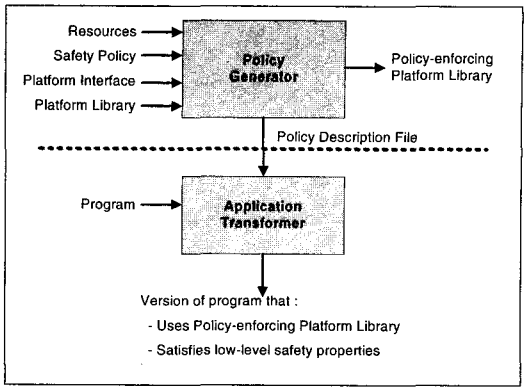

그림 1 어플리케이션 변환 시스템 아키텍처

이에 따르면 전체 시스템은 크게 정책 생성기(policy generator)와 어플리케이션 변환기(application transformer)로 구성된다. 정책 생성기는 최초 시스템 설치 시 또는 보안 정책 변경 시에 동작한다. 그리고, 이 때 입력으로는 시스템 리소스에 대한 추상적인 기술과 리소스 조작에 대한 제한 사항을 담고 있는 보안 정책(safety policy), 그리고 해당 플랫폼의 API 라이브러리와 이들의 리소스 사용 내역에 대한 정보가 주어진다. 따라서 이러한 입력을 바탕으로, 정책 강행에 필요한 코드를 삽입한 플랫폼 라이브러리(policy-enforcing platform library)와 실제적인 코드 수정 지침이 기술된 정책 기술 파일(policy description file)이 생성되면 어플리케이션 변환을 위한 준비 작업이 완료된다. 대상 코드가 주어지면 어플리케이션 변환기는 정책 기술 파일을 참조하여 해당 코드의 특정 API 호출을 변형된 플랫폼 라이브러리에 대한 호출로 교체함으로써 실행 시에 사전 정의된 정책이 적용되도록 한다.

2.3 정적 분석에 의한 악성 스크립트 감지

이 기법은 각각의 위험한 메소드 호출이 아니라 악성 행위를 구성하는 메소드 시퀀스들을 정의함으로써 악성 행위를 정확하게 감지하려는 의도에서 제안되었다[2]. <그림 2>는 메일을 통하여 자기 복제를 수행하는 비주얼 베이직 스크립트의 예이며, 다수의 메소드 호출이 하나의 악성 행위를 구성하기 위해서는 반드시 그것들의 파라미터와 리턴 값 사이에 특별한 관계가 존재해야 함을 확인할 수 있다. 예를 들어, 4행의 Copy 메소드는 현재 실행 중인 스크립트를 "LOVE-LETTER-FOR-YOU.TXT.VBS"라는 이름으로 복사하고, 7행의 Attachments.Add 메소드는 그 파일을 새로 만들어진 메일 객체에 첨부함으로써 메일을 통한 자기 복제를 달성한다.

그러나, 메소드 호출의 존재유무만을 검사하는 방식을 사용하게 되면, A라는 이름으로 스크립트 파일을 생성하고 B라는 이름의 파일을 첨부하는 관계없는 메소드 호출이 존재하여도 이를 악성 코드로 간주하므로 높은 긍정 오류를 보이게 되는 것이다. 이 기법은 메소드 호출의 존재뿐 아니라, 상술한 파일명, fso, c, out, male 등 모든 관계 있는 값들이 일치하는가를 검사함으로써, 기존 방식에서 나타나는 높은 오류율을 극복하려고 시도하였다.

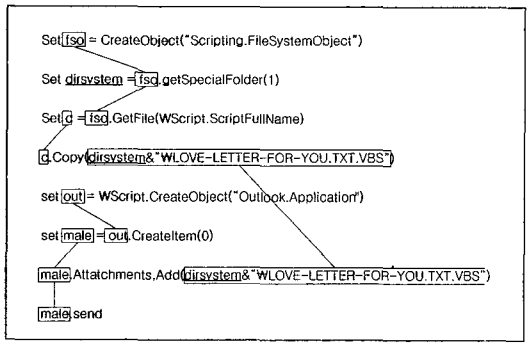

그림 2 메일을 통한 자기 복제를 수행하는 비주얼 베이직 스크립트의 일부

실제에 있어 이러한 악성 행위는 단순히 일련의 메소드 시퀀스로만 정의할 수 없으며, 다양한 메소드 또는 메소드 시퀀스들의 조합으로 이루어진다. 따라서, 이 기법에서는 악성 행위가 단위 행위들의 조합으로 이루어지며, 각각의 단위 행위는 더욱 작은 단위 행위 또는 하나 이상의 메소드 호출들로 이루어진다고 모델링하고, 각 단위 행위와 메소드 호출 문장을 하나의 룰(rule)로 표현하였다. 예를 들어, <그림 2>에 나타난 메소드들만을 고려하여 악성 행위의 패턴을 정의하면 <그림 3>과 같은 형태로 표현할 수 있다.

```
M1 : Set $1 = CreateObject(Scripting.FileSystemObject)
M2 : Set $1 = $2.GetFile(WScript.ScriptFullName)
M3 : $1.Copy($2)

M4 : set $1 = WScript.CreateObject(Outlook.Application)
M5 : set $1 = $2.CreateItem(0)
M6 : $1.Attatchments.Add($2)
M7 : $1.send

R1 : (M1.$1 == M2.$2) && (M2.$1 == M3.$1)
       => $1 = M3.$2          // code writing
R2 : (M4.$1 == M5.$2) && (M5.$1 == M6.$1) && (M5.$1 == M7.$1)
       => $1 = M6.$2          // send mail
R3 : R1.$1 == R2.$1
       => $1 = true           // A Malicious Behavior is Detected!
```

그림 3 메일을 통한 자기 복제 행위 정의의 예

그림과 같이 룰은 매칭 룰(matching rule)과 관계 룰(relation rule)의 두 가지 종류가 있으며, 각각의 이름 첫 자가 M, R 인 것으로 구분된다. 매칭 룰의 경우에는 우측에 기술한 것과 동일한 패턴을 가진 문장이 존재하면 조건이 만족되고, 관계 룰의 경우에는 우측의 조건식이 참(true)이면 만족된다. 실제로, 이 그림에 제시된 룰 정의는 본 논문에서 개선한 새로운 룰 기술 문법에 따른 것이므로 [2]에서 제시한 것과는 형태에 다소 차이가 있으나 그 의미는 동일하다고 볼 수 있다. 룰 기술 문법의 변경에 대한 상세한 내용은 3장에서 언급된다.

이러한 정적 분석을 통해 악성 행위 여부를 판단하는 것은 악성 행위에 사용될 수 있는 메소드 호출의 출현 빈도만을 고려하는 단순한 감지 기법에 비해, 극히 낮은 긍정 오류를 보장받을 수 있다는 장점을 가진다. 그러나, 궁극적으로 실행 전의 소스 코드 분석만으로는 실행 시 해당 파라미터 또는 리턴 값이 어떤 값을 가질지 예측할 수 없는 경우가 빈번하게 발생하므로, 실제로 악성 코드임에도 불구하고 이를 감지하지 못하는 부정 오류(false negative)가 높아질 가능성도 가지게 된다.

3. 코드 삽입에 의한 악성 코드 감지

3.1 코드 삽입 기법 개요

기존의 어플리케이션 변환 기법은 소스 또는 P-코드와 같이 소스 프로그램에 준하는 형태를 가진 이동 코드(mobile code)에 대한 접근 제어 강행에는 유용하게 사용될 수 있다. 그러나, 각각의 함수 호출간의 관계를 고려하는 것이 아니고 단지 특정 API의 실행 허가 여부만을 결정하므로 악성 행위의 패턴을 감지할 수는 없다. 또한, 정적 분석 기법은 악성 행위의 메소드 호출 시퀀스를 통해 정확한 감지를 시도하나 실행 시에만 결정할 수 있는 값이 하나라도 개입되면 다른 조건을 만족하여도 이를 악성행위로 간주할 수 없으므로 높은 부정 오류를 수반하게 된다. 따라서, 감지 방식에 있어서는 정적 분석 기법에서와 같은 룰 기반의 메소드 호출 시퀀스 탐지를 채용하되, 어플리케이션 변환 기법을 이용하여 탐지 루틴을 스크립트 소스에 삽입함으로써 실행 시에 악성 코드 감지를 수행하는 악성 코드 탐지 기법을 제안한다. 코드 삽입 동작을 간략히 도시하면 <그림 4>와 같다.

외부로부터 유입되거나 악성 여부가 의심되는 스크립트는 실행 전 임의의 시점에 스크립트 변환기(script transformer)를 거쳐 실행 중 지속적으로 자체 진단을 수행할 수 있는 형태로 변환된다. 그러나, 변환기는 본래부터 스크립트에 기술되어 있던 문장을 변경하지는

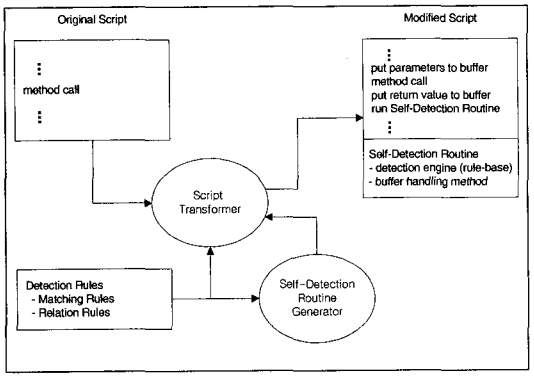

그림 4 코드 삽입 기법 개념도

않으며, 추가적인 코드의 삽입만을 수행한다. 이 때 삽입되는 코드는 크게 두 종류로 나누어 질 수 있다.

첫째는 메소드 호출 전후에 파라미터와 리턴 값을 취하고 자체 진단 루틴을 호출하는 문장들이다. 이것은 모든 메소드 호출 문장에 삽입되는 것이 아니며, 매칭 룰에 기술된 것과 정확히 일치하는 형태를 보이는 메소드 호출 문장 전후에만 삽입된다. 또한, 메소드에 따라 취해야 할 파라미터의 수가 다르고, 리턴 값이 없는 메소드 호출도 존재하므로 스크립트 변환기는 매칭 룰을 적절히 분석하여 필요한 값들만을 취하고 자체 진단 루틴을 호출하는 코드를 삽입한다.

둘째로 자체 진단을 수행하는 루틴이다. 이것은 스크립트의 내용과 무관하며, 오직 악성 행위를 정의한 룰에 따라서만 달라진다. 즉, 이 루틴은 악성 행위 정의가 갱신되지 않는 한 변경되지 않으므로, 자체 진단 루틴 생성기(self-detection routine generator)에 의해 사전에 생성되며, 룰 기반의 진단 엔진(detection engine)과 버퍼 조작 메소드들로 구성된다. 진단 엔진은 매칭 룰과 일치하는 형태의 메소드 호출 시에만 수행되므로, 해당 매칭 룰에 관련된 모든 관계 룰들을 실행하여 현재까지의 메소드 호출 시퀀스가 악성 행위를 구성하였는가를 검사하는 역할을 담당한다. 버퍼 조작 메소드는 매칭 룰을 만족하는 메소드 호출 문장의 파라미터와 리턴 값을 진단 엔진이 사용할 수 있는 버퍼에 저장하는 메소드를 의미한다.

이러한 형태로 스크립트가 변형되면, 추후 실행 시 위험 메소드가 실행 될 때마다 당시까지의 메소드 호출 과정을 고려하여 악성 코드 여부를 검사하게 된다. 따라서, 안티-바이러스가 설치되지 않은 시스템에서도 자기 진단을 지속적으로 수행하는, 일종의 예방 접종(vaccination) **효과를** 가지게 된다. 또한, COM(Component Object Model) 계층 전체를 가로채는(hooking) 시스템 수준에서의 감시에서 발생하는 오버헤드, 복잡한 구현, 운영체제의 기반 라이브러리 변형 등의 문제를 회피하면서도 동일한 효과를 얻을 수 있다는 장점을 가진다.

또한, 이러한 감지 기법은 특정 스크립트 언어에 국한되지 않으며 단지 사용하는 룰 집합만을 다르게 정의함으로써 다수의 스크립트 언어에 동일한 알고리즘을 적용할 수 있다. 특히, 마이크로소프트 윈도우즈에서는 비주얼 베이직 스크립트, 자바스크립트와 같은 다수의 언어가 동일한 윈도우즈 스크립팅 호스트를 통해 실행되며 동일한 런-타임(run-time) 객체 및 환경을 사용한다. 따라서, 대부분의 경우 각각의 스크립트 문법에 맞도록 매칭 룰만을 수정하는 것만으로 서로 다른 언어를 위한 룰 집합을 정의할 수 있게 된다. 아래에서는 상술한 코드 삽입 기법을 구체화할 때 고려해야 할 점들과 구현 내용을 기술한다.

3.2 룰 기술(rule description)

악성 행위의 정의에 사용되는 룰은 기존의 코드 분석 기법의 것과 유사하나, 좀 더 일관성 있고 단순한 형태로 변경되었다. 수정된 룰 기술 문법은 다음과 같다.

```
rule_description ::= { rule }1.
rule ::= rule_identifier : rule_body
rule_body ::= matching_rule_body | { relation_rule_body }1.
matching_rule_body ::= script statement with variable_string
variable_string ::= variable | *
variable ::= $ { digit }1.
relation_rule_body ::= condition_phrase => action_phrase
condition_phrase ::= condition_expr { logical_operator
condition_expr }0.
condition_expr ::= rule_identifier | local_variable
string_compare_operator local_variable
logical_operator ::= && | ||
string_compare_operator ::= < | ==
action_phrase ::= action_stmt { , action_stmt }0.
action_stmt ::= variable = rvalue
rvalue ::= local_variable | variable | true | false
local_variable ::= rule_identifier . variable
```

그림 5 룰 기술 문법

이에 따르면 하나의 룰 기술 파일은 다수의 룰 정의로 구성되며, 각각의 룰은 룰 ID(rule_identifier)와 룰 바디(rule_body)로 구성된다. 룰 바디의 형식은 매칭 룰과 관계 룰에 따라 달라지는데, 매칭 룰의 경우에는 룰 변수(variable_string)를 포함하는 것 외에는 일반적인 스크립트 문장의 형태를 가지게 된다. 관계 룰의 경우에는 조건절(condition_phrase)과 동작절(action_phrase)로 구성되어, 조건절의 조건이 만족될 경우 동작절의 내용을 수행하게 된다. 조건절은 하나 이상의 조건식으로 구성되며, 각각의 조건식은 특정 룰이 이미 만족되었는

표 1 악성 스크립트가 수행하는 자기 복제 행위

구 분	내 용
로컬 시스템 내의 자기 복제	대상 시스템에 자신의 복사본을 만든다. 엄밀하게 구분하면 존재하지 않는 새로운 복사본을 생성하는 경우와, 대상 시스템에 이미 존재하던 스크립트 파일의 내용을 자신과 같은 내용으로 바꾸어 넣는 행위로 나눌 수 있다.
전자우편을 통한 자기 복제	주소록에 기재된 계정으로 자신을 첨부한 전자우편을 전송한다.
IRC를 통한 자기 복제	mIRC와 같은 IRC 클라이언트의 초기화 스크립트를 변경하여 대화상대 접속시, 자신을 전송하도록 한다.
공유 폴더를 통한 자기 복제	네트웍 공유 폴더를 검색하여 자신을 복사한다.

지 또는 두 룰의 특정 변수 값들이 같거나 다른 한편에 포함되는지를 검사하도록 기술된다.

기존 연구에서 룰 기술을 복잡하게 만들었던 가장 큰 요인은 AND, OR와 같은 논리 연산자(logical operator)를 지원하지 않는다는 점이었다. 이로 인해 A, B, C가 각각 하나의 조건식(condition_expr)이라 하였을 때, "(A AND B) OR C"와 같은 조건은 다음과 같이 기술되어야만 했다.

R1 : cond A
R2 : precond A
cond B
action $global = true
R3 : cond C
action $global = true

즉, AND로 연결된 조건은 다수의 룰로 분리하여 사전 만족 조건절(pre-condition phrase)에 기술하고, OR로 처리해야 할 부분은 각각의 조건식이 만족될 경우 false로 초기화된 전역변수를 true로 변경하는 우회적인 방법으로 기술되었다. 이것은 룰 기술과 추후 해독을 어렵게 하는 요소이므로, 조건식에 논리 연산자를 직접 사용할 수 있도록 하고 전역 변수의 사용을 금지함으로써, 전체적으로 룰 기술이 단순하고 악성 행위의 논리와 일치하도록 수정하였다.

한편, 실제적인 룰 정의는 기존 악성 코드들에 대한 경험적인 분석을 통해 얻어질 수 있다. 이론적으로 증명된 바와 같이 가능한 모든 악성 행위를 정확하게 감지할 수 있는 휴리스틱 룰의 집합은 존재하지 않으며, 새로운 악성 코드 또는 행위 패턴이 등장할 때마다 지속적인 갱신이 이루어져야 한다. 악성 코드가 일반적인 프로그램과 확연하게 구분될 수 있는 특성이 자기 복제(self-duplication)의 수행이라는 점이라는데 주목하여, 현재까지 알려진 악성 스크립트의 자기 복제 행위를 정리하면 <표 1>과 같다.

이상과 같은 각각의 악성 행위는 하나 이상의 단위 행위 또는 메소드 호출 패턴으로 정의되고, 각각의 단위 행위는 다시 하나 이상의 단위 행위 또는 메소드 호출 패턴으로 정의되므로, 특정 악성 행위는 하나의 룰이 하나의 노드로 나타나는 트리 형태로 표현될 수 있다. 따라서, 알려진 많은 악성 스크립트를 분석하여 자기 복제 동작을 수행하는 코드 패턴들을 트리 형태로 정리하고, 이것을 정의된 문법에 따라 기술함으로써 완성된 룰을 얻을 수 있게 된다.

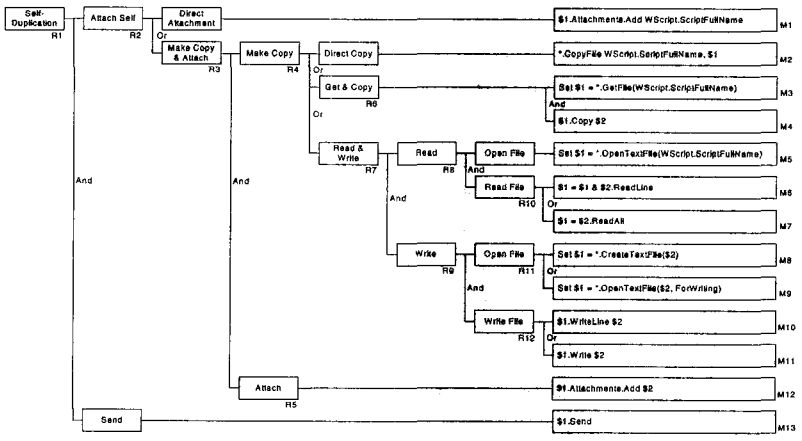

그림 6 비주얼 베이직에서 전자우편을 통한 자기 복제 패턴

예를 들어, 이미 알려진 비주얼 베이직 악성 스크립트들로부터 전자우편을 이용한 다양한 자기 복제 패턴들을 추출하여 하나의 트리로 정리하면 <그림 6>과 같은 형태를 보이게 된다. 그림에 제시된 트리의 단말 노드는 변형 없이 그대로 매칭 룰로 이용될 수 있으므로, 중간 노드들의 의미를 고려하여 정의된 문법에 따라 표현하면 <그림 7>과 같은 형태의 관계 룰을 얻을 수 있다.

```
R1  : R2.$1 == M13.$1                R7  : R8.$1 == R9.$2
      -> $1 = true                         -> $1 = R9.$1
R2  : M1 || R3                       R8  : M5.$1 == R10.$2
      -> $1 = $1                           -> $1 = R10.$1
R3  : R4.$1 == R5.$2                 R9  : R11.$1 == R12.$1
      -> $1 = R5.$1                        -> $1 = R11.$2, $2 = R12.$2
R4  : M2 || R6 || R7                 R10 : M6 || M7
      -> $1 = $1                           -> $1 = $1, $2 = $2
R5  : M12                            R11 : M8 || M9
      -> $1 = $1, $2 = $2                  -> $1 = $1, $2 = $2
R6  : M3.$1 == M4.$1                 R12 : M10 || M11
      -> $1 = M4.$2                        -> $1 = $1, $2 = $2
```

그림 7 전자우편을 통한 자기 복제 패턴 감지를 위한 관계 룰 정의

그림에 나타난 룰 기술 중, 매칭 룰에서 사용된 "*"는 어떠한 토큰에도 부합될 수 있는 와일드카드(wildcard)를 의미한다. 또한, 조건절에서 룰의 존재 여부만을 검사하는 관계 룰에서는 동작절의 우변에 별도의 룰 ID를 기술하지 않아도 조건을 만족하는 룰의 룰 변수로 인식하여 동작하게 된다. 예를 들어 R4의 경우, M2의 룰이 만족됨으로 인해 조건식이 만족되었다면 동작절은 "$1 = M2.$1"으로 해석되고, R6의 룰이 만족됨으로 인한 것이라면 "$1 = R6.$1"으로 해석되므로 간략한 룰 기술이 가능하게 된다.

이와 같은 룰 기술을 통해 비주얼 베이직 스크립트 뿐 아니라, 다른 스크립트 언어로 작성된 행위 패턴도 동일한 방법으로 감지할 수 있다. 실제로 <그림 6>에 제시된 매칭 룰의 R3, R5, R8, R9의 첫 번째 토큰인 "Set"만을 제거하면 자바스크립트에서 메일을 통한 자기복제 행위를 감지할 수 있다. 객체의 대입에는 반드시 "Set" 문장을 사용하여야 하는 비주얼 베이직 스크립트와는 달리, 자바 스크립트는 일반적인 형태의 대입문으로 이것이 가능하므로, 이러한 문법 상의 차이만을 고려해 주면 관계 룰의 수정 없이도 동일한 감지 동작을 수행할 수 있다.

3.3 스크립트 코드의 삽입

대상 스크립트가 주어지면 악성 행위를 구성하는 메소드 시퀀스에 속한 메소드 호출 전후에, 파라미터와 리턴 값을 버퍼에 저장하고 악성 행위를 검사하는 함수를 호출하는 코드가 삽입되어야 한다. 이 작업은 <그림 4>의 스크립트 변환기에 의해 이루어지며, 코드 삽입이 이루어진 후 메소드 호출 코드는 <그림 8>과 같이 변경된다.

```
RuleBase.SetVal 0, "M2" : RuleBase.SetVal 2, FSO
Set c = FSO.GetFile(WScript.ScriptFullName)
RuleBase.SetVal 1, c : RuleBase.Check
```

그림 8 코드 삽입 후 메소드 호출 코드의 예

그림에서 2행의 FSO.GetFile이 검사 대상인 메소드이고, 1행과 3행이 이 메소드의 동작을 검사하기 위해 삽입된 코드이다. RuleBase 객체는 룰 정의와 이에 의한 악성 행위 감지 루틴을 제공하는 객체이며, 그 구현은 코드 삽입 단계가 끝나면 수정된 코드의 뒷부분에 덧붙여진다. 이 객체의 코드는 자체 진단 루틴 생성기에 의해 생성되며, 상세한 내용은 다음절에 기술된다. 메소드 호출을 검사하기 위해 사용되는 메소드는 그림에서와 같이 SetVal과 Check 뿐이며, SetVal은 배열로 구성된 버퍼의 주어진 위치에 해당 값을 대입하고, Check는 버퍼의 내용을 참조하여 룰에 따라 검사를 수행하는 진단 엔진의 역할을 담당하게 된다. 이 때 사용하는 버퍼는 다수의 값들을 저장할 수 있도록 배열로 구성되어 있으므로 해당하는 매칭 룰의 이름 외에도, 모든 파라미터와 리턴 값들이 하나의 배열에 저장된다. 비주얼 베이직이나 자바스크립트 등과 같은 많은 스크립트 언어에서는 일반적인 프로그래밍 언어에서의 어떠한 형(type)도 제약 없이 하나의 변수에 담길 수 있어서, 별도의 구조체를 이용하지 않고 배열만으로 동일할 작업을 수행할 수 있게 된다[13]. 버퍼를 구성하는 배열의 각 위치에 저장되는 자료의 의미는 <표 2>와 같다.

표 2 메소드 호출 검사를 위한 버퍼의 구조

배열 위치	의미	비고
0	일치한 매칭 룰의 이름	문자열
1	리턴 값	이름이 아닌 실행 시의 값이 저장됨
2	호출된 메소드를 제공하는 객체	
3 이상	파라미터	

이상과 같은 스크립트 변환기의 동작을 정리하면 다음과 같다.

<단계 1> 룰 기술 파일로부터 매칭 룰을 로드한다.

<단계 2> 악성 행위 감지 루틴을 초기화하는 문장을

출력한다(RuleBase 객체 초기화 메소드 호출).

<단계 3> 주어진 스크립트의 모든 문장에 대해 다음의 작업을 수행한다.

한 문장을 읽는다.

읽은 문장이 매칭 룰과 일치하면

읽은 문장의 전후에 파라미터와 리턴 값을 저장하고 자기진단 루틴을 호출하는 문장을 덧붙여 출력한다.

그렇지 않으면 (읽은 문장이 매칭 룰과 일치하지 않으면)

읽은 문장을 그대로 출력한다.

<단계 4> 자체 진단 루틴 생성기로부터 얻어진 악성 행위 감지 루틴 코드(RuleBase 클래스 코드)를 추가한다.

이러한 코드 삽입은 기존의 어플리케이션 변환 기법과 비교할 때 다음과 같은 점이 다르다.

첫째, 가장 큰 차이점은 기존의 어플리케이션 변환 기법이 각각의 API가 접근할 수 있는 자원을 제한함으로써 샌드박스(sandbox)와 유사한 효과를 얻기 위한 것이나, 제안하는 기법은 각각의 메소드 호출이 아니라 이들이 구성하는 행위 패턴의 감지를 통해 악성 코드를 구별한다는 점이다.

둘째, 기존의 어플리케이션 변환 기법에서는 접근 정책이 반영된 별도의 라이브러리가 해당 도메인의 사용자들에게 배포되어야 한다. 그러나, 제안하는 코드 삽입 기법은 악성 행위의 감지에 필요한 모든 코드가 대상 스크립트 내부에 삽입되므로, 코드 삽입이 이루어진 스크립트는 여타 환경에 구애받지 않고 지속적인 자기 감시를 수행할 수 있게 된다.

셋째, 어플리케이션 변환 기법이 단지 지정된 함수 호출을 새로운 함수의 호출로 변환하며, 이 함수가 주어진 파라미터만을 평가하여 접근 권한을 부여하는 것과는 달리, 제안하는 기법에서는 리턴 값까지 검사의 대상에 포함한다. 또한, 단지 메소드 이름만을 참조하지 않고 매칭 룰에 기술된 모든 토큰이 동일한 형태로 나타났을 때만 코드 삽입을 실시한다. 예를 들어 <그림 2>의 M2 룰은 첫 단어가 Set으로 시작하고 하나의 단어 뒤에 '=' 이 나오며, 그 뒤에 객체이름, '.', 그리고 메소드 이름이 나온 뒤, Wscript.ScriptFullName을 파라미터로 가지는 GetFile 호출만을 유효한 매칭으로 간주한다. 즉, 스크립트에 Set a = fso.GetFile("test.vbs")와 같은 문장이 나타났다면, 이 문장은 파라미터 값이 다르므로 M2를 만족하지 못한다. 이러한 정확한 매칭은 삽입되는 코드 자체에도 영향을 줄 수 있는데, 룰 M2의 예를 보면 리턴 값과 객체 이름만이 관심 대상이며 파라미터인 Wscript.ScriptFullName는 코드 삽입 단계에서 이미 일치함을 확인하였으므로 <그림 7>에서와 같이 그 값을 버퍼에 넣지 않게 된다. 이것은 일반적인 어플리케이션 변환 기법이 단지 메소드 이름만을 참조하여 모든 메소드 호출의 코드를 변환하는 것과는 달리, 매칭 단계에서 악성 행위에 참여할 가능성이 좀 더 높은 호출과 이의 감지에 필요한 값들만을 고려하도록 함으로써 실행 시의 효율을 높여준다.

3.4 악성 행위 감지 루틴의 추가

메소드 호출 정보를 버퍼에 삽입하고 감지를 수행하는 악성 행위 감지 관련 루틴들은 하나의 클래스로 묶여질 수 있으며, 이것이 앞 절에서 언급한 RuleBase 클래스이다. 이 클래스가 제공하는 공개(public) 메소드는 <표 3>과 같다.

표 3 악성 행위 감지 클래스의 공개 메소드

메 소 드	내 용
init	클래스 초기화
SetVal(pos, value)	value 값을 버퍼의 pos 번째에 대입
Check	악성 행위 여부 검사

코드 삽입 단계를 거치게되면 매칭 룰에 기술된 형태를 가진 메소드 호출 문장 전후에 SetVal 메소드를 사용하여 검사에 필요한 값을 버퍼에 저장하고, Check 메소드를 호출하는 코드가 삽입된다. 실제에 있어 Check 메소드는 단지 버퍼의 내용을 참조하여 어떠한 매칭 룰로 인해 자신이 호출되었는가를 알아낸 뒤, 해당 매칭 룰을 구현한 메소드를 호출하는 진입점(entry point)의 역할만을 수행한다. 즉, 각각의 룰은 악성 행위 탐지 클래스에 속한 하나의 내부 메소드(private method)로 나타나며, <그림 4>의 자체 진단 루틴 생성기가 룰 정의를 참조하여 각각의 메소드를 자동 생성하게 된다.

각각의 룰이 구현된 메소드의 내용은 매칭 룰과 관계 룰의 경우가 다르다. 매칭 룰은 이미 해당 문장이 주어진 형식과 일치할 경우에만 수행되므로, 아무런 조건 없이 해당 룰에 대한 매칭이 일어났음을 기록하기 위해 하나의 룰 인스턴스(instance)를 생성하고 상위 룰을 검사한다. 룰의 인스턴스는 해당 룰에 관련된 정보를 담고 있는 데이타 구조이며 주어진 조건이 만족될 때 생성된다. 매칭 룰의 인스턴스에 저장되는 정보는 $1, $2와 같은 매칭 룰 변수의 값들이므로, 생성되는 인스턴스의 적

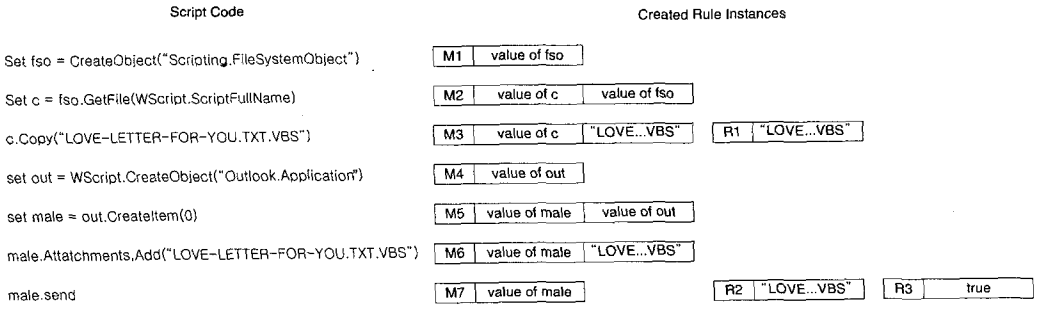

그림 9 스크립트 실행 중의 룰 인스턴스 생성

절한 위치에 버퍼의 값들을 대입하는 것만으로 인스턴스의 생성이 가능하다.

관계 룰의 동작은 기본적으로 매칭 룰과 동일하나 항상 조건식을 먼저 검사하고 이것이 만족되었을 경우에만 해당 룰의 인스턴스를 생성한 뒤 상위 룰을 검사한다는 점이 다르다. 여기에서 조건식의 만족이란 실제로는 해당 조건식을 만족시켜주는 룰 인스턴스들이 존재함을 의미한다. 또한, 상위 룰이란 해당 룰을 자신의 조건식에 담고 있는 룰을 말하며, <그림 6>과 같은 트리 형태로 룰을 나타낼 때 해당 룰의 부모 노드에 위치한 룰을 지칭하므로, 룰 정의의 분석을 통해 실행 전에 모두 결정이 가능하다. 자체 진단 루틴 생성기의 동작을 정리하면 다음과 같다.

<단계 1> 매칭 룰과 관계 룰을 로드한다.

<단계 2> 로드된 룰들을 분석하여 각 룰들의 상위 룰을 기록한다. 그 내용은 다음과 같다.

모든 관계 룰에 대하여 다음의 작업을 수행한다.

하나의 룰 R_c를 선택한다.

선택된 룰의 조건식에 나타나는 룰들의 집합 S를 구한다.

S에 속한 모든 룰들에 대해, 상위 룰을 R_c로 기록한다.

<단계 3> 모든 매칭 룰에 대해, 이에 대응하는 메소드를 생성한다. 메소드 내용은 다음과 같다.

버퍼의 내용을 참조하여 새로운 룰 인스턴스를 생성한다.

상위 룰에 대응하는 메소드를 호출한다.

<단계 4> 모든 관계 룰에 대해, 이에 대응하는 메소드를 생성한다. 메소드 내용은 다음과 같다.

룰 바디 부분 부분을 파싱(parsing)하여 스크립트 문법에 맞게 변환한다.

상위 룰에 대응하는 메소드를 호출한다.

이렇게 생성된 악성 행위 감지 루틴에 의해 스크립트의 실행 중에 이루어지는 룰 인스턴스의 생성을 예로 들면 <그림 9>와 같다. 이 그림은 좌측에 제시한 스크립트의 각 행이 수행될 때마다 생성되는 룰 인스턴스를 우측에 도시하고 있으며, 각 룰 인스턴스의 이름 뒤에 붙여진 필드는 해당 룰 정의에서 $1, $2와 같이 기술된 룰 변수의 값을 의미한다.

4. 실험결과

코드 삽입에 의한 악성 코드 감지 기법은 악성 행위 감시를 위한 코드가 해당 어플리케이션의 외부에 있는 것이 아니라 해당 코드 내부에 삽입된다는 점 외에는 근본적으로 기존의 행위 감시 기법과 유사한 방법론을 채택하고 있다. 따라서, <그림 10>에 제시된 기존의 행위 감지(behavior detection) 기법과 동일한 감지 범위 및 정확도를 보이게된다.

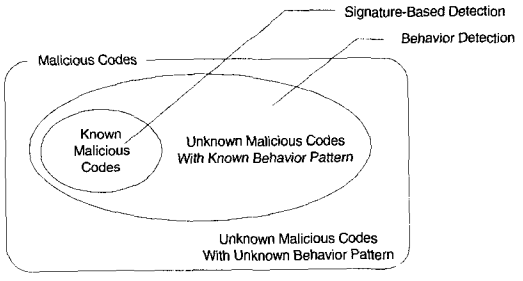

그림 10 악성 코드 감지 기법과 적용 가능 범위

행위 감시를 통한 악성 행위 감지 기법의 정확도는 감지 알고리즘 자체에는 관련이 적으며, 사용되는 악성 행위 정의의 정확성에 크게 의존하게 된다. 특히 제안하

는 기법은 단순하게 메소드 호출 자체만을 검사하는 것이 아니라 기존의 정적 분석 기법을 활용하여 호출에 수반되는 인자까지 정확하게 검사하므로, 악성 행위 패턴의 정의가 정확하다면 긍정 오류 없이 알려진 악성 행위의 발생을 감지할 수 있게 된다.

따라서, 실험은 코드 삽입으로 인해 발생되는 오버헤드의 측정에 중점을 두고 이루어졌다. 실험에 사용한 샘플은 각각 10개씩의 악성 코드와 일반(legitimate) 코드로 구성되었으며, 사용하는 메소드 호출의 패턴이 유사한 것들은 가급적 배제하고 서로 다른 행위 패턴을 보이는 것들이 선택되었다. 샘플 집합에 대한 실험 결과 긍정 또는 부정 오류는 발생하지 않았으며, 펜티엄 III 500MHz CPU를 장착하고 마이크로소프트 윈도우즈 2000 운영체제를 탑재한 PC에서 다음과 같은 수행 결과를 보였다.

표 4 코드 삽입으로 인한 오버헤드 (평균값)

구분	코드 크기 (byte)			실행시간 (초)		
	원본	변환 후	비율*	원본	변환 후	비율*
일반코드	3012	6177	2.386	2.319	2.394	1.051
악성코드	8925	23291	2.871	34.034	42.124	1.216

* "비율"은 변환된 코드에서 얻어진 값을 원본코드에서 얻어진 값으로 나눈 것임

이 표에서 알 수 있듯, 일반적인 용도로 사용되는 스크립트는 악성코드에 비해 작은 크기를 가지고 있으며, 코드 크기 또는 실행 시간의 증가율이 악성코드에 비해 낮게 나타나고 있다. 특히 변환 후에는 어떠한 코드에나 동일한 자기 진단 루틴이 삽입되는 제안된 기법의 특성을 감안해볼 때, 그 이상의 코드 증가가 거의 발생하지 않았고, 이로 인해 수행시간에도 거의 영향을 미치지 않음을 확인할 수 있다. 그러나, 악성 코드의 경우에는 상대적으로 큰 오버헤드를 보이게 된다.

이상의 실험에서 다음과 같은 결론을 얻을 수 있다.

첫째, 클라이언트에 아무런 안티바이러스 모듈의 설치 없이, 서버에서의 처리만으로도 실질적인 행위 감지를 수행할 수 있다. 따라서, 이러한 기법은 전자우편 서버, 프록시 서버 등과 같이 해당 도메인에 유입되는 모든 프로그램이 특정 서버를 통과하는 서비스에서 유용하게 사용될 수 있다. 특히, 코드 분석이나 에뮬레이션 등과 같은 여타 서버 기반 방법론에 비해 서버와 클라이언트 모두에게 적은 부하를 주면서도 정확한 감지를 수행할 수 있다.

둘째, 실행 시간의 관점에서 볼 때, 일반 스크립트는 악성 스크립트보다 적은 오버헤드를 보인다. 변환된 코드는 악성 행위에 사용되는 메소드의 출현 시마다 자기 진단 루틴이 호출되는데, 일반 스크립트는 이러한 메소드가 적게 나타나므로 전체적인 수행 시간의 증가가 적게 나타남은 당연한 결과로 해석될 수 있다. 따라서, 실제 상황에서는 시스템에 보유한 전체 스크립트 중 극히 일부만이 악성이라는 사실을 고려해보면 사용자가 체감할 수 있는 시스템의 오버헤드는 더욱 작을 것으로 예상할 수 있다.

셋째, 코드 삽입으로 인한 코드 크기의 증가는 실행시간의 증가에 비해 상대적으로 크게 나타남을 관찰할 수 있다. 이는 모든 룰을 처리할 수 있는 고정된 자기 진단 루틴이 항상 삽입되어야 한다는 것과, 매칭 룰에는 나타나지만 해당 코드 내에서는 실질적으로 악성 행위에 가담하지 않는 메소드 호출들의 검사를 위한 코드가 삽입된다는 점에 기인한다. 예를 들어, <그림 6>에서와 같이 "write" 메소드는 자기 복제에 사용될 수 있다. 그러나, 실제 특정 스크립트에 존재하는 "write" 메소드 호출이 모두 자기 복제에 사용되는 것은 아니며, 극히 일부의 "write" 메소드만이 이 같은 악성 행위에 가담하고 나머지는 일반적인 출력 작업에 사용된다. 따라서, 코드 삽입 단계에서 정적분석을 먼저 수행하고, 정적 분석을 통해 판단할 수 없는 메소드 호출만을 실행 시간에 감시하도록 코드를 변환하게 되면 코드의 크기나 실행 시간이 현저하게 감소할 것으로 예상된다. 이 같은 정적분석과 코드 삽입 기법의 결합은 메소드 검사를 위한 코드 삽입의 양을 줄일 뿐 아니라 실행 시에 가동되지 않을 룰들도 진단 루틴에서 삭제할 수 있도록 하므로, 악성 행위를 수행하지 않는 일반적인 스크립트는 변환 후에도 원본과 유사한 크기와 내용을 가지게 될 것이다.

5. 결론 및 향후연구

서버 수준의 안티바이러스는 기업 환경이나 조직의 전자우편 서버, 프록시 서버 등과 같이 해당 도메인에 유입되는 모든 프로그램이 특정 서버를 통과하는 서비스에서 유용하게 사용될 수 있다. 그러나, 알려지지 않은 악성 스크립트 감지에 있어, 코드 분석에 기반한 방법은 동적으로 생성되는 정보를 얻을 수 없으며, 에뮬레이션에 의한 방법은 서버에서 수행하기에 부하가 크고 모든 프로그램 흐름을 검사하기 어렵다는 단점을 가지고 있다. 본 논문에서는 서버에서의 실행만으로 지속적인 행위 감시가 가능하도록 하는 악성 스크립트 감지 기법을 제안하고 그 구현 및 실험 결과를 제시하였다. 실험 결과, 클라이언트에 별도의 안티바이러스 구성 요소가 없이 행위 감시에 의한 악성 코드 판별이 가능함

을 확인하였으나 변환된 코드에 오버헤드가 존재함이 확인되었다. 따라서, 현재 정적분석 기법의 적극적인 활용을 통하여 오버헤드를 감소시키는 방법에 대한 연구가 진행 중이다. 향후에는 악성 코드의 감지 이전에 수행된 악성 코드의 영향으로부터 시스템을 복구하는 대응 기법에 대한 연구가 이루어질 것이며, 궁극적으로는 이진 코드에도 이와 동일한 기법을 적용함으로써 기업 환경에서의 서버용 안티바이러스 프레임웍에 이용 가능하도록 할 것이다.

참 고 문 헌

[1] Eugene H. Spafford, *"Computer Viruses as Artificial Life,"* Journal of Artificial Life, MIT Press, 1994.

[2] 배병우, 이성욱, 조은선, 홍만표, "정적 분석 기법을 이용한 악성 스크립트 탐지", 2001년 한국정보보호학회 종합학술발표회 논문집, Vol. 11, No. 1, pp.91-95, 2001. 11.

[3] 차민석, "악성 스크립트의 종류와 역사", 안철수연구소, 2002.

[4] Frederick B. Cohen, "Computer Viruses: Theory and Experiments," Computers and Security 6, 1987, pp.22-35.

[5] Frederick B. Cohen, "A Short Course on Computer Viruses," John Wiley & Sons, Inc, 1994.

[6] David M. Chess, Steve R. White, "Undetectable Computer Viruses," Virus Bulletin Conference, 2000. 9.

[7] Vesselin Bontchev, *"Macro Virus Identification Problems,"* 7th International Virus Bulletin Conference, 1997.

[8] Mark Kennedy, *"Script-Based Mobile Threats,"* Symantec AntiVirus Research Center, 2000. 6.

[9] Baudouin Le Charlier, Morton Swimmer, Abdelaziz Mounji, *"Dynamic detection and classification of computer viruses using general behaviour patterns,"* Fifth International Virus Bulletin Conference, Boston, September 20-22, 1995.

[10] Francisco Fernandez, *"Heuristic Engines,"* 11th International Virus Bulletin Conference, 2001. 9.

[11] Gabor Szappanos, *"VBA Emulator Engine Design,"* Virus Bulletin Conference, 2001. 9.

[12] David Evans, Andrew Twyman, *"Flexible Policy-Directed Code Safety,"* IEEE Security and Privacy, Oakland, CA, May 9-12, 1999.

[13] Microsoft, *VBScript User's Guide*, http://msdn.microsoft.com, Microsoft, 2001.

이 성 욱

정보과학회논문지 : 정보통신
제 29 권 제 5 호 참조

방 효 찬

1995년 3월 북해도공업대학 경영공학과 졸업. 1997년 3월 북해도공업대학 기계시스템공학과 석사 졸업. 1997년 6월 ~ 1999년 10월 한국통신 운용연구단 전임연구원. 2000년 8월 ~ 현재 한국전자통신연구원 능동보안기술연구팀 연구원. 관심분야는 네트워크보안, 액티브네트워크, 네트워크관리

홍 만 표

정보과학회논문지 : 정보통신
제 29 권 제 5 호 참조