알려지지 않은 악성 암호화 스크립트에 대한 분석 기법

(An Analysis Technique for Encrypted Unknown Malicious Scripts)

이 성 욱 † 홍 만 표 ††

(Seong-uck Lee) (Man Pyo Hong)

요 약 악성 코드의 감지 및 분석에 있어 암호화된 악성코드의 해독은 필수적인 요소이다. 그러나, 기존의 엑스-레잉 또는 에뮬레이션에 의한 해독 기법들은 이진 형태의 악성 코드를 대상으로 개발되었으므로 스크립트 형태의 악성 코드에는 적합하지 않으며, 특정한 암호화 패턴을 기반으로 하는 접근 방식은 알려지지 않은 악성 스크립트가 암호화되어 있을 경우 대응하기 어렵다. 따라서 본 논문에서는 스크립트 암호화 기법에 대한 분석적인 접근을 통하여, 새로운 암호화 기법의 출현에 유연하게 대처하는 새로운 암호 해독 기법을 제시하고 그 구현에 관해 기술한다.

키워드 : 컴퓨터 바이러스, 악성 코드, 스크립트, 암호화

Abstract Decryption of encrypted malicious scripts is essential in order to analyze the scripts and to determine whether they are malicious. An effective decryption technique is one that is designed to consider the characteristics of the script languages rather than the specific encryption patterns. However, currently X-raying and emulation are not the proper techniques for the script because they were designed to decrypt binary malicious codes. In addition to that, heuristic techniques are unable to decrypt unknown script codes that use unknown encryption techniques. In this paper, we propose a new technique that will be able to decrypt malicious scripts based on analytical approach. we describe its implementation.

Key words : computer virus, malicious code, script, encryption

1. 서 론

일반적인 의미에서 암호화(encryption)는 그 의미가 드러나지 않도록 메시지를 인코딩(encoding)하는 절차 또는 기법을 의미한다[1]. 그러나, 컴퓨터 바이러스 또는 악성코드에 있어서의 암호화는 악성코드를 변조(scrambling)함에 의해 바이러스 탐색기(scanner)가 해당 악성코드의 시그니처(signature)를 찾지 못하도록 숨기는 기법을 지칭한다[2].

시그니처란 특정 악성코드에만 존재하며 다른 프로그램에는 존재하지 않는 짧은 문자열로서, 다른 일반적인 프로그램(legitimate program)들과 악성 코드를 구분하고 그것이 어떤 악성 코드인지를 식별하는데 이용된다[3]. 시그니처를 이용한 악성코드 탐지 방식은 다른 기법에 비해 상대적으로 빠른 속도를 보여주므로, 현존하는 대부분의 안티-바이러스(anti-virus) 제품들이 이러한 시그니처 기반의 감지 방식을 채택하고, 이에 약간의 휴리스틱(heuristic) 알고리즘을 결합한 형태가 주종을 이루고 있다.

이런 감지 방식에 대응하기 위하여, 악성코드 제작자들은 바이러스에 암호화 기능을 추가하였다. 암호화된 악성코드는 일반적으로 해독루틴과 키(key) 값, 그리고 암호화된 악성코드로 구성되고, 실행 시에 해독 루틴이 먼저 수행된다. 따라서, 해독 루틴은 암호화된 악성 코드를 해독하고 해독된 악성코드 쪽으로 제어를 넘김으로써 악성 코드가 실행되도록 한다[2]. 이로 인해, 악성코드가 다른 시스템 또는 다른 화일에 자기 복제(self-replication)를 시도할 때 새로운 키 값을 사용하여 인코딩 하는 것만으로 전혀 다른 형태를 보이므로 단순한 스캐닝만으로는 감지할 수 없게 된다.

이러한 암호화 바이러스에 대한 대응 기법으로는 엑

· 이 논문은 한국전자통신연구원 위탁연구에 의해 지원되었음

† 학생회원 : 아주대학교 컴퓨터공학과
suleeip@yahoo.co.kr
†† 종신회원 : 아주대학교 정보통신전문대학원 교수
mphong@ajou.ac.kr

논문접수 : 2001년 11월 7일
심사완료 : 2002년 6월 12일

스-레잉(X-raying)과 에뮬레이션(emulation) 기법이 알려져 있다[4]. 그러나, 이 같은 방법들은 해당 악성 코드의 특성과 행위가 알려져 있는 경우에만 적용 가능하거나 이진 화일의 형태를 가진 악성 코드의 해독에 적합한 형태이므로, 알려지지 않은 암호화 스크립트에는 적용하기 어려운 것이 사실이다. 따라서, 기존 스크립트 악성 코드에서 사용하는 암호화 기법의 패턴과 해독 방법을 정의하고 이를 이용하는 휴리스틱 기반의 방법론이 스크립트 악성코드를 위한 가장 현실적인 암호 해독 기법으로 간주되고 있다. 그러나, 이 같은 휴리스틱 기반의 접근은 새로운 암호화 패턴이 출현할 때마다 이를 처리할 수 있는 코드를 바이러스 탐색기에 추가하여야 하므로, 알려지지 않은 악성 스크립트에 원활하게 대응할 수 없다는 근본적인 단점을 가지게 된다. 실제로, 최근에는 향상된 암호화 기법들이 속속 등장하고 있고, 이 같은 기법이 스크립트 악성코드 생성기에 탑재됨으로써 누구나 암호화된 악성 스크립트를 손쉽게 만들어 낼 수 있는 것이 현실이다. 또한, 시그니처 기반의 스캐닝에서 탈피하여 알려지지 않은 악성코드에 대응하기 위해 많은 연구가 이루어지고 있는 최근의 경향에 비추어, 새로운 암호화 기법의 출현에 영향을 적게 받는 해독 기법이 제시되어야 함은 당연한 귀결이라 할 수 있다.

본 논문에서는 현재 가장 많이 유포되고 있는 비주얼 베이직 악성 스크립트에 대한 분석적인 접근을 통해 악성코드가 이용하는 암호화 방법론을 세 가지 유형으로 구분하고, 주류를 이루고 있는 한 유형에 대한 구체적인 해독 알고리즘을 제시 및 구현함으로써 그 가능성을 입증하였다. 따라서, 2장에서는 알려지지 않은 암호화된 악성 스크립트에 대응할 수 있는 분석적인 접근 방식을 제안하고, 3장에서 이러한 방법론을 구체화한 암호 해독 알고리즘을 소개한다. 4장에서는 제시한 알고리즘의 실험 결과를 제시하고, 5장에는 결론과 향후 연구 내용을 기술한다.

2. 악성코드의 암호화 및 해독

2.1 기존의 암호화 악성 코드 해독 기법

상술한 바와 같이, 암호화 악성 코드를 해독하기 위해 보편적으로 사용되는 방법으로는 엑스-레잉과 에뮬레이션 기법이 알려져 있다.

엑스-레잉 기법은 해당 악성 코드에서 찾아야 하는 시그니처와 악성 코드가 사용하는 해독 알고리즘에 대한 사전 지식을 이용하여 그 범위를 좁힌 후 모든 경우를 시도(brute-force decryption)하는 방법을 말한다. 즉, 해당 악성 코드의 암호화 기법과 시그니처에 대한 모든 정보가 있으며 단지 정확한 키 값만을 모를 경우, 가능한 모든 키 값을 이용하여 시그니처가 나타날 수 있는 위치의 문자열을 해독하고 이것이 시그니처와 동일한 값을 가지는 가를 검사하여 악성 코드 여부를 판단할 수 있게 된다[4]. 그러나, 이러한 방법은 찾고자 하는 악성코드의 암호화 기법과 특성이 면밀히 분석되어 충분한 사전 지식이 얻어진 후에 실행 가능하므로, 알려지지 않은 새로운 악성코드에는 적용하기 어렵다는 단점을 가지게 된다.

에뮬레이션 기법은 그 명칭이 의미하는 바와 같이, 악성 코드를 에뮬레이션 하여 해독된 코드를 얻는 방법을 말한다. 이진 형태의 악성 코드는 해독 루틴이 가장 먼저 실행되며, 그 크기가 매우 작으므로 가상 기계에서 해당 코드의 일부를 실행함으로써 해독된 악성코드를 얻을 수 있게 된다. 이 때, 모든 해독이 완료된 코드를 얻어내려 한다면 가상기계의 각 메모리 유닛을 감시하여 코드 부분의 메모리에 더 이상 변화가 일어나지 않는 시점까지 실행을 계속하여야 하나, 시그니처 기반의 감지 방법과 결합하여 사용하는 경우에는 시그니처가 위치한 메모리 값들의 해독이 완료되면 그 즉시 에뮬레이션을 중단하고 시그니처 비교를 실시하게 된다. 이 같은 에뮬레이션 기법은 이진 파일 형태의 악성 코드를 해독하는 데는 효과적이나, 스크립트를 위한 에뮬레이터 구축은 이진(binary) 실행 화일에 비해 어려운 것이 현실이다. 즉, 에뮬레이션을 위해서는 대상 코드가 실행될 수 있는 모든 환경을 가상적으로 만들어 주어야 하는데, 마이크로소프트 비주얼 베이직 스크립트(이하 비주얼 베이직 스크립트라 칭함)와 같은 스크립트 언어의 경우에는, 해당 프로그램에서 사용하는 각종 객체와 제반 환경들을 모두 에뮬레이션 하는 것이 현실적으로 매우 어려우며 부하 또한 크다고 알려져 있다[5]. 또한, 아무런 위해(harm) 행위를 하지 않는 일반적인 코드와는 달리, 악성코드는 단순하게 실행 시켜 보고 실행 내역을 프로화일링(profiling)하는 기법을 사용할 수도 없다.

이러한 문제들로 인해, 현재 악성 스크립트의 해독에는 알려진 암호화 패턴 각각에 대한 해독 방법을 적용하는 휴리스틱 기반의 방법론이 가장 보편적으로 사용되고 있다. 예를 들면, 기존의 많은 비주얼 베이직 스크립트 악성코드는 실제 악성코드를 하나의 문자열에 암호화하여 놓고 이를 스크립트 언어에서 정의하는 "execute" 문장을 통해 실행시키는 형태로 구성되어 있다. 이러한 경우, 주어진 스크립트에서 "execute" 문장을 찾고 이 문장에서 호출되는 함수를 해독 함수로 간주하여 이를 실행함으로써 해독된 악성코드를 얻을 수

있게 된다[6]. 그러나, 이 같은 접근은 새로운 암호화 패턴이 출현할 때마다 이를 처리할 수 있는 코드를 바이러스 탐색기에 추가하여야 하며, 스크립트 악성 코드에서 독특하게 나타나고 있는 문자열 단위의 부분 암호화에 원활하게 대응하기 어렵다는 단점을 가지게 된다.

2.2 암호화 기법

암호화된 악성 스크립트는 보편적으로 블랙 박스(black-box)와 같은 형태의 별도 해독 함수를 가지고 있다. 그러나, 기존의 안티-바이러스에 대응하기 위하여 소수이지만 이러한 형태에서 벗어난 암호화 기법들도 존재하고 있으며, 더욱 그 수가 증가할 것으로 예상되고 있다. 따라서, 악성 코드의 암호화에 근본적으로 대응하기 위해서는 현존 암호화 기법뿐만 아니라 향후 등장할 수 있는 기법들까지 포괄 할 수 있는 정확한 분류 체제의 수립이 선행되어야 한다.

본 논문에서는 암호화 기법을 해독 함수 존재 여부에 따라 크게 두 부류로 나누고, 해독 함수를 가지는 형태는 그 함수가 외부 코드와 의존성을 가지는 경우와 그렇지 않은 경우로 다시 구분하였다. 이를 도시하면 <그림 1>과 같다.

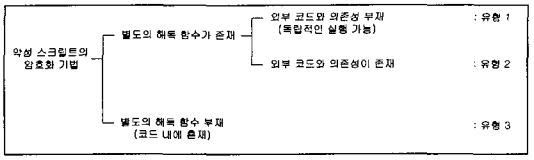


그림 1 악성 스크립트의 암호화 기법

결과적으로 이러한 분류는 암호 해독 루틴이 악성코드의 몸체(body)와 얼마나 깊게 연관되어 있는가에 의한 것이라 할 수 있다. 즉, 유형 1은 해독 루틴이 가장 독립적으로 존재하는 경우이며, 유형 3은 전체 코드 중 임의의 부분에 삽입되어 프로그램의 실행 상태와 밀접하게 연관되어 경우이다. 그러나, 이 같은 구분을 좀 더 명확하게 하기 위해서는 단일 루틴 내에서 정의되는 코드간의 의존성과는 달리, 함수와 외부 코드간의 의존성을 정의하여야 한다.

• 정의 1 : 프로그램 코드 내에 정의된 함수 f가 다음과 같은 세 가지 성질을 만족하면 "함수 f가 외부와 의존성이 없다" 또는 "함수 f가 독립적이다"라고 하고, 이 때 함수 f를 독립 함수라 한다.

i) 함수 내부의 모든 코드가 외부 코드와 의존성이 없어야 한다. 즉, 함수 f 내에서 정의 또는 사용된 전역 변수들의 집합을 $V(f)$라하고, 외부에서 정의 또는 사용된 전역 변수들의 집합을 $E(f)$라 할 때, $V(f) \cap E(f) = \emptyset$이어야 한다.

ii) 모든 프로그램 내에서, 함수 f 호출에 대한 실인자(actual parameter)는 반드시 상수로 주어져야 한다. 변수가 인자로 주어질 경우 이를 통하여 의존성이 발생하기 때문이다.

iii) 함수 내부에서 부작용(side effect)이 없는 함수만을 호출하여야 한다. 여기에서 부작용은 I/O 또는 이를 유발하는 모든 행위를 지칭하며, 외부와 의존성이 있는 다른 함수를 실행시켜 간접적으로 의존성을 유발하는 경우까지 포함한다.

따라서, 이러한 정의를 바탕으로 암호화 기법의 유형을 명확하게 정의하면 다음과 같다.

• 정의 2 : 악성 스크립트의 암호화 기법은 다음과 같은 세 가지 유형으로 나누어진다.

i) 유형 1 : 별도의 해독 함수가 존재하고, 이것이 독립함수인 경우

ii) 유형 2 : 별도의 해독 함수가 존재하고, 이것이 독립함수가 아닌 경우

iii) 유형 3 : 별도의 해독 함수가 존재하지 않는 경우

실용적인 측면에서 볼 때, 현재 보편적인 스크립트 악성코드의 암호화 패턴은 크게 두 가지로 나타난다. 첫째는 <그림 2>와 같이 악성코드 전체가 하나의 문자열로 암호화 되어있는 경우로, 그림에 제시된 것은 VBS/VBSWG.T로 명명된 악성 스크립트이다. 비주얼 베이직 스크립트의 "execute" 문장은 인자로 주어진 문자열을 프로그램 코드로 보고 실행하므로, 먼저 해독함수를 호출하게 된다. 따라서, 전체 코드가 완전히 해독된 후에 실행을 개시하게 된다.

```
Execute(snphhuatvsbkwuj("&Wcr/Wcrvf/H,Vnsl/@w ..... doe!gtobuhno"))

Function snphhuatvsbkwuj(zwbyjntbpmhqqgh)
    For vvpzxfszgnczrao = 1 To Len(zwbyjntbpmhqqgh)
        ccuhbhjhyzkheeq = Mid(zwbyjntbpmhqqgh, vvpzxfszgnczrao, 1)
        If Asc(ccuhbhjhyzkheeq) = 7 Then
            ccuhbhjhyzkheeq =Chr(34)
        End If
        If Asc(ccuhbhjhyzkheeq) <> 35 and Asc(ccuhbhjhyzkheeq) <> 34
          Then
            If Asc(ccuhbhjhyzkheeq) Mod 2 = 0 Then
                ccuhbhjhyzkheeq = Chr(Asc(ccuhbhjhyzkheeq) + 1)
            Else
                ccuhbhjhyzkheeq = Chr(Asc(ccuhbhjhyzkheeq) - 1)
            End If
        End If
        snphhuatvsbkwuj = snphhuatvsbkwuj & ccuhbhjhyzkheeq
    Next
End Function
```

그림 2 하나의 문자열로 암호화된 악성 스크립트의 예 (VBS/VBSWG.T)

두 번째 패턴은 <그림 3>과 같이 프로그램에서 사용하는 일부 문자열을 암호화한 경우이다. 이러한 유형의 악성 스크립트는 하나 또는 그 이상의 암호 해독 함수를 가지며, 함수의 인자 또는 대입문의 우변값(r-value)들에 사용되는 임의의 문자열이 암호화된 형태를 보인다. 따라서, 첫 번째 경우와 달리 실행이 진행되면서 필요한 시점에 필요한 문자열이 해독되는 방식으로 동작하게 된다.

```
...
Set Roy = Maggie.CreateTextFile(Maggie.BuildPath(Maggie.GetSpecialFolder
          (2),V("7594E44554D405E2458545")),True)
Roy.WriteLine(V("E402") & Maggie.BuildPath(Maggie.GetSpecialFolder(2),
          V("7594E44554D405E245D405")))
Roy.WriteLine("E 0100" & H("4D5AE701630001006002406FFFF5F0C"))
Roy.WriteLine("E 0110" & H("000200000001F0FF570000000132504B"))
...
...
Function V(Van)
    For Kirk = 1 To Len(Van) Step 2
        V = V & Chr("&h" & Mid(Van,Kirk + 1,1) & Mid(Van,Kirk,1))
    Next
End Function

Function H(Houten)
    For Luann = 1 To Len(Houten) Step 2
        H = H & " " & Mid(Houten,Luann,2)
    Next
End Function
```

그림 3 일부 문자열들이 암호화된 악성 스크립트의 예 (VBS/TripleSix)

첫 번째와 같은 단순한 형태는 특정한 코드 패턴을 찾는 단순한 접근으로도 해독 가능하나, 두 번째 경우에는 해당 악성 코드에 대한 사전지식 없이 자동적으로 해독 함수를 찾아내는 일이 난해하므로, 새로운 악성 스크립트에 대응하는데 어려움이 따르게 된다. 그러나, 앞서 정의한 바에 따르면 이 두 가지 패턴의 암호화 기법은 모두 첫 번째 유형에 속하는 것으로, 동일한 특성을 가지고 있으므로 외형상의 패턴에 관계없이 일관된 방법으로 해독 가능하다.

2.3 알려지지 않은 암호화 스크립트의 해독

궁극적으로 볼 때, 암호화된 악성코드의 해독 과정은, 그것이 에뮬레이션에 의한 것이든, 실제 실행에 의한 것이든, 어느 정도의 프로그램 실행을 필요로 한다. 그러나, 서론에서 밝힌 바와 같이 완벽한 에뮬레이터의 이용이나 단순한 프로화일링 기법의 사용에는 많은 문제가 있으며, 이러한 문제를 회피하기 위해서는 해당 스크립트의 암호를 해독하는데 필요한 부분만을 발췌하여 실행하는 방법이 필요하게 된다. 특히, 알려지지 않은 새로운 스크립트를 대상으로 한다면 다음과 같은 두 가지 문제에 부딪히게 된다.

첫째, "대상 스크립트가 암호화되어 있는가"를 판단하는 문제이다. 이미 안티-바이러스 개발자들에 의해 분석된 악성 코드는 암호화 여부와 암호 해독 방법이 알려져 있다. 그러나, 새롭게 등장한 악성 코드의 경우에는 아무런 사전 지식 없이 대상 스크립트의 분석만으로 암호화 여부를 판단하여야 한다.

둘째, "대상 스크립트의 암호 해독 루틴이 어떤 것인가"를 탐색하는 문제이다. 암호화된 스크립트에는 해독 함수 외에도 많은 함수들이 정의되어 있을 수 있다. 따라서, 새로운 암호화 기법의 출현에 영향 받지 않도록 특정 코드 패턴에 의존하지 않고 해독 함수의 집합을 찾아내야 한다.

이러한 문제에 대하여 본 논문에서 제안하는 해결책은 대상 스크립트가 암호화되었는지, 또는 암호 해독 루틴이 어떤 것인지 분석하기보다는 해당 스크립트에서 상수화 할 수 있는 모든 값을 상수로 대치함으로써 자연적인 암호 해독을 유도하는 것이다. 자신이 악성 행위를 시도 중임을 사용자가 의식하지 못하도록 해야하는 악성 코드의 본질로 인해, 대부분의 악성코드는 특별한 사용자 입력을 받지 않으며 어느 시스템에서나 공통적으로 얻을 수 있는 자원만을 사용하게 된다. 따라서, 악성코드에 입력되는 데이타 집합은 일반 프로그램에 비해 상당히 고정적인 성향을 띄게 되고, 이로 인해 프로그램 내의 많은 변수들이 실제로 매 실행 시마다 동일한 값을 가지게 된다. 특히, 암호화에 관련된 부분은 어떤 상황에서도 원래 의도한 코드를 복원하여야 하므로 이 같은 경향이 더욱 두드러지게 나타난다. 이러한 기본 방법론은 상술한 각각의 암호화 유형에 다음과 같이 적용될 수 있다.

유형 1의 암호화는 해독 함수가 독립 함수로 존재하므로, 단순히 독립 함수의 실행 결과 값을 대치하는 것만으로 암호화된 내용의 해독이 가능하다. 즉, 독립 함수에 대한 호출식(call expression)과 함수 정의만이 기술된 임시 스크립트를 생성하고 이를 실행시킨 후, 각각의 결과 값을 원래의 호출식이 있던 자리에 대치하여 해독된 스크립트를 얻을 수 있게 된다. 이 과정을 정리하면 <그림 4>와 같다.

이 때, 실제로 암호 해독과는 관계없는 다른 함수도 독립 함수로 판정되어 실행될 수 있으나, 이것은 스크립트에 어떠한 문제도 발생시키지 않는다. 정의에 따라, 독립 함수의 리턴 값은 주어진 인자에만 영향을 받으므로 어떤 상태에서 실행하여도 동일한 인자에 대해서 동

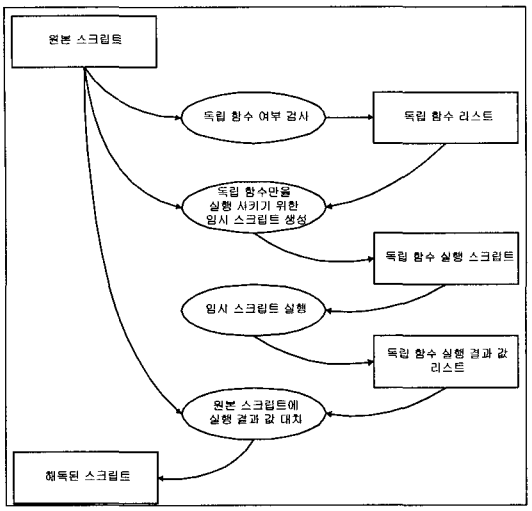


그림 4 암호 해독 절차

일한 값이 산출되기 때문이다.

유형 2는 별도의 해독 함수가 존재하나 독립 함수는 아닌 경우이다. 독립 함수가 되지 못한 이유가 변수 또는 인자에 있었다면 프로그램 분석을 통해 DU-체인(Def-Use chain)[7]을 형성하고, 이에 연관된 문장들도 임시 스크립트에 포함시킴으로서 첫 번째 유형과 동일한 결과를 얻을 수 있다. 그러나, 함수 내부 또는 연관된 문장이 I/O 등 부작용을 유발할 수 있는 문장이라면, 이것이 악성 행위로 연결될 소지가 있거나 I/O를 통한 의존성이 성립될 수 있으므로 별도의 처리를 해주어야 한다.

유형 3은 별도의 함수가 존재하지 않는 형태이므로, 프로그램 전체에 걸친 상수화를 시도하여야 한다. 즉, 프로그램 전체에 걸쳐 의존성 분석을 실시하고, 안전한 것으로 확신되는 함수들도 실행함으로써 가급적 많은 부분을 상수화 하여야 한다. 따라서, 프로화일링에 근접한 모습을 보이게 될 것이다.

결국, 제안하는 방법론은 컴화일러 최적화 단계에 존재하는 상수 전파(constant propagation)의 확장으로 볼 수 있으며, 이러한 상수화는 단지 암호화된 코드의 해독뿐 아니라 뒤에 이어지는 코드 분석 단계의 복잡도를 감소시키는 부수 효과도 가져올 수 있다. 이 같은 관점에서 기존의 해독 기법과 제안하는 기법의 특징을 정리하면 <표 1>과 같다.

그러나, 두 번째와 세 번째 유형의 암호화에 대응하기 위해서는 필요한 분석과 절차가 좀 더 명확하게 정의되어야 하고 I/O 등의 부작용을 처리하는 방법에 대한 추가적인 연구가 필요하다. 실제로 현재 유포되고 있는 대

표 1 기존 기법과 제안된 기법의 비교

해독 기법	엑스-레잉	에뮬레이션	휴리스틱 기법	제안하는 기법
필요한 해독 알고리즘의 수	해독을 시도하려는 악성 코드의 해독 루틴 수	1	2	1
해독에 필요한 추가 자료	없음	없음	부분 문자열 암호화의 경우, 각 악성 스크립트의 해독 함수	없음
구현 용이성	용이	어려움	용이	용이
실행시간	빠름	실제 실행시간보다 많은 시간 소요	빠름	빠름
알려지지 않은 악성 코드 해독	불가능	가능	부분 문자열 암호화의 경우, 불가능	가능

부분의 악성 스크립트는 암호화 패턴이 상이하더라도 첫 번째 유형에 속하는 것이 대부분이다. 따라서, 이하에서는 유형 1의 암호화에 대응하기 위한 해독 알고리즘과 구현, 그리고 실험 결과를 제시한다.

3. 해독 알고리즘

3.1 독립 함수의 탐색

특정 함수가 독립 함수인지를 판단하려면 이것이 사용하는 변수와 함수에 대한 분석이 이루어져야 한다. 독립 함수가 사용할 수 있는 함수는 부작용이 없는 내장 함수와 다른 독립 함수뿐이고, 내장 함수는 언어 사양(specification)에 명시적으로 정의되어 있으므로 그 중에서 I/O 등의 부작용이 없는 함수들의 리스트를 따로 정의하면 사용유무를 손쉽게 알아낼 수 있다. 그러나, 해당 함수가 지역 변수만을 사용하는 함수인가를 검사하는 작업은 이에 비해 복잡한 과정을 거쳐야 한다.

모든 변수가 명시적으로 선언된 후 사용되는 일반적인 함수 기반 언어에서는, 각 영역에서 선언된 변수의 집합을 추출하고 해당 함수 내에서 전역 변수를 사용하는지를 검사하는 작업이 비교적 단순하게 이루어진다. 그러나, 비주얼 베이직 스크립트와 같은 인터프리터 기반 스크립트 언어는 일반적인 함수 기반 언어와는 달리 변수의 선언을 반드시 필요로 하지 않는다. 또한, 함수 내부에서 정의되었다 하더라도 Dim 문장을 통해 그것이 지역 변수임을 명시하지 않은 것은 모두 전역 변수로 간주된다[8]. 따라서, 전역 변수의 완전한 리스트를 얻기 위해서는 각각의 함수 내부까지 모든 코드를 검사하여야 한다. 해독 알고리즘을 위한 독립 함수의 존재를

판단할 수 있는 조건은 다음과 같이 정의된다.

n = 스크립트에 정의된 함수의 갯수

F_i = 스크립트에 i번째로 정의된 함수 ($1 \le i \le n$)

A_i = 함수 F_i에서 정의 또는 사용된 모든 변수의 집합 ($1 \le i \le n$)

D_i = 함수 F_i에서 Dim으로 선언된 모든 변수의 집합 ($1 \le i \le n$)

V_0 = 어떤 함수에도 속하지 않는 글로벌(global) 영역에서 정의 또는 사용된 변수의 집합

이 때, 함수 F_i에서 정의 또는 사용된 전역 변수의 집합 V_i와, 함수 F_i의 외부 영역에서 정의 또는 사용된 변수들의 집합 E_i는 다음과 같이 구할 수 있다.

$$V_i = A_i - D_i$$

$$E_i = \bigcup_{i \neq j, 0 \le j \le n} Vj$$

따라서, 함수 F_i가 독립함수이려면 다음의 조건을 만족하여야 한다.

$$V_i \cap E_i = \varnothing$$

즉, 독립 함수 F_i는 자신을 제외한 외부 영역에서 정의 또는 사용된 어떠한 변수도 정의하거나 사용하지 않는 함수이다. 효율성을 고려하여 독립 함수를 찾아내기 위한 실제 과정은 두 개의 테이블을 이용하여 한번의 스크립트 스캔과 후처리 작업으로 이루어진다. 각 테이블의 필드와 그 의미는 <표 2>에 기술된 바와 같다.

표 2 독립 함수 탐색을 위한 테이블 구성

테이블	필드	의미 또는 내용
Var Table	VarName	해당 변수의 이름을 나타내는 문자열
	FuncId	해당 변수의 정의/사용이 발견된 함수의 ID. 함수마다 고유한 정수값이 부여되며, 글로벌 영역은 0 값이 주어짐
	isLocal	해당 변수가 특정 함수내에서 dim으로 선언된 지역변수이면 truc, 아니면 falsc. (글로벌 영역에서 dim으로 선언된 것도 falsc임)
Func Table	FuncName	해당 함수의 이름을 나타내는 문자열
	FuncId	해당 함수의 ID
	isIndep	해당 함수가 독립함수이면 true, 아니면 falsc

이들 테이블을 이용한 실제 독립 함수 탐색과정은 다음과 같다.

<단계 1> 스크립트를 스캔하여 변수 및 함수의 사용/정의를 찾는다.

이 과정에서 스크립트 화일의 끝에 도달했으면

<단계 4>로 간다.

성공적으로 찾았으면

name = 찾아낸 변수/함수명

curFunc = 현재 스캔 중인 함수의 ID

<단계 2> 찾아낸 토큰의 유형에 따라 다음의 동작을 수행한다.

찾아낸 것이 함수 정의이면

이것을 FuncTable에 추가한다. (isIndep = true로 놓는다.)

찾아낸 것이 함수 사용이면

함수 테이블에 있는 함수이고 인자가 모두 상수가 아니면

해당 함수를 함수 테이블에서 찾아 isIndep = false로 한다.

허용된 내장 함수 리스트에 없는 것이면

현재 스캔 중인 함수를 함수 테이블에서 찾아 isIndep = false로 한다.

찾아낸 것이 변수이면

변수 테이블에 name = VarName 이고 curFunc = FuncId 인 레코드가 없으면

이것을 변수 테이블에 기록한다.

<단계 3> 스캔을 계속하기 위하여 <단계 1>로 간다.

<단계 4> 변수 테이블의 모든 레코드를 대상으로 다음과 같은 후처리를 수행한다.

isLocal = false 이고 VarName이 같은 두 개의 레코드가 존재하면

해당 레코드의 FuncId를 ID로 하는 함수(VarTable.FuncId = FuncTable.FuncId)를 함수 테이블에서 찾아 isIndep = false로 기록한다.

따라서, 이러한 과정을 거친 후 함수 테이블의 isIndep 값이 true로 남아 있는 함수가 독립함수로 간주된다.

3.2 독립 함수 실행 스크립트 생성

독립 함수가 모두 밝혀지면, 이에 대한 호출식을 모두 추출하여 임시 스크립트를 생성한다. 임시 스크립트는 다음과 같은 역할을 수행하는 문장들로 구성되어 있다.

- 결과 값 출력을 위한 화일 개방 및 종결
- 독립 함수 호출 후 리턴 값을 화일에 기록
- 독립 함수 정의
- 타입 핸들링(type handling) 함수 정의

<그림 5>는 <그림 3>에 제시된 악성 스크립트에서

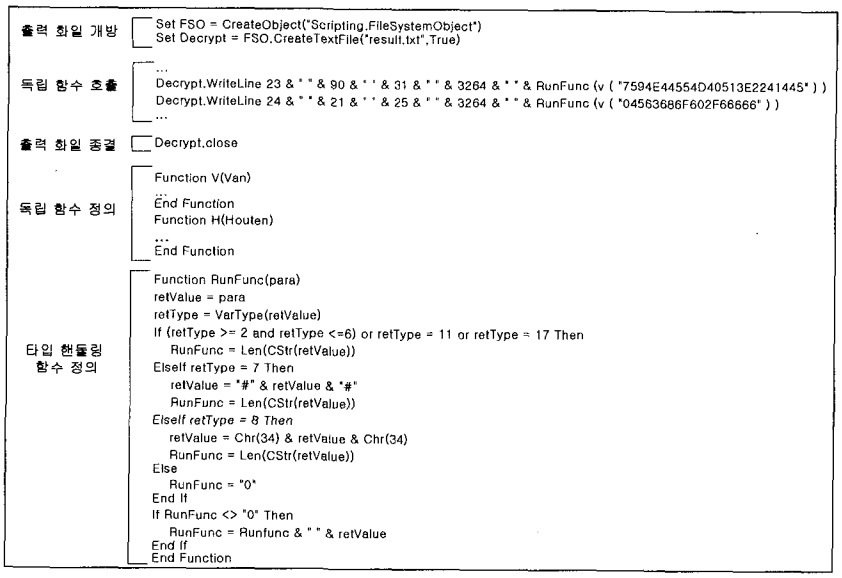


그림 5 독립 함수 실행 결과 값의 추출을 위해 생성된 임시 스크립트의 예

표 3 비주얼 베이직 스크립트의 부타입(subtype)

D Subtype	Description	convertible
vbEmpty	Empty (uninitialized)	×
vbNull	Null (no valid data)	×
vbInteger	Integer	○
vbLong	Long integer	○
vbSingle	Single-precision floating-point number	○
vbDouble	Double-precision floating-point number	○
vbCurrency	Currency	○
vbDate	Date	○
vbString	String	○
vbObject	Automation object	×
vbError	Error	×
vbBoolean	Boolean	○
vbVariant	Variant (used only with arrays of Variants)	×
vbDataObject	A data-access object	×
vbByte	Byte	○
vbArray	Array	×

얻어진 임시 스크립트의 예로서, 함수 V와 H가 독립함수로 판정되어 이들에 대한 호출 결과를 화일에 기록하고 있음을 보여준다. 이 때, 함수 호출식의 결과 값 뿐 아니라 해당 호출식에 관련된 정보가 함께 기록되는데, 제시된 예에서 함수 호출식의 앞에 기술된 숫자들이 이에 해당한다. 이 정보들은 다음 단계에서의 처리에 이용되며, 그 의미는 다음과 같다.

- 해당 함수 호출식이 있었던 원본 스크립트의 행, 열, 호출식의 문자열 길이
- 해당 호출을 포함하고 있는 함수의 ID
- 실행 결과 값의 문자열 길이
- 실행 결과 값

이 정보들 중 실행 결과 값의 문자열 길이와 실행 결과 값은 타입 핸들링 함수를 통해 얻어진다. 타입 핸들링 함수의 존재는 비주얼 베이직 스크립트의 특성에 기인한 것으로, 비주얼 베이직 스크립트에는 Variant라는 단 한 가지의 타입(type) 만이 존재하며 특정 값이 주어지면 <표 3>과 같은 부타입(subtype)이 결정된다[8]. 따라서, 비주얼 베이직 스크립트의 함수는 마치 매크로 함수와 같이 매번 다른 타입의 인자를 받을 수도, 그에 따라 다른 타입의 결과 값을 돌려 줄 수 있다.

이러한 비주얼 베이직 스크립트의 특성으로 인해 리

턴 값의 타입을 실행 전에 확신할 수 없게 되고, 실행 시간에 이를 위한 별도의 처리가 필요하게 된다. 즉, 임시 스크립트를 생성하고 실행하는 것은 그 결과 값을 미리 얻어 원본 스크립트의 함수 호출식에 대치하기 위한 것인데, 부타입에 따라 문자열로의 변환이 불가능하여 스크립트에 직접 써넣을 수 없는 무형의 값도 존재하게 된다. 이에 덧붙여, 문자열로 표현 가능한 부타입이라도 원본 스크립트에 삽입하기 위해서는, 문자열의 양끝에는 따옴표를, Date 타입의 양끝에는 #을 붙여주고, 기타 숫자형인 경우에는 그대로 문자열로 변환하는 작업을 해주어야 한다.

이러한 문제의 해결을 위해, 각각의 함수 호출식의 실행 결과는 <그림 5>에 RunFunc로 나타난 타입 핸들링 함수를 거쳐 적절한 형태의 문자열로 먼저 변환된다. 이 때 문자열로 변환할 수 없는 부타입을 가지는 결과 값이 나타나면, 결과 값의 길이를 0으로 기록함으로써 함수 호출식 대치 과정에서 이를 인지하도록 한다.

3.3 실행 결과의 획득

이 같은 과정을 통해 임시 스크립트가 얻어지면, 윈도우즈 스크립팅 호스트(Windows Scripting Host)[9]의 호출을 통해 이를 실행시켜 해당 함수의 실행 결과 값을 얻는다. <그림 5>에서 제시한 임시 스크립트에서 얻어진 실행 결과는 <그림 6>과 같다.

```
  original_expr          return_value
row  col  len funcID  len      string
 22   28   59  3264    28 "Scripting.FileSystemObject"
 23   90   31  3264    14 "WINTEMP1.BAT"
 24   21   25  3264    11 "@echo off"
 25   21   29  3264    13 "debug.exe <"
 25  105   29  3264    13 "WINTEMP.TXT"
 25  140   17  3264     7 " >nul"
...
```

그림 6 임시 스크립트 실행 결과의 예

이렇게 함수 호출식의 결과 값이 얻어지면, 이 값들을 원본 스크립트에 대치하여 해독된 스크립트를 얻는다.

```
...
set maggie = createobject ("Scripting.FileSystemObject")
set marjorie = maggie.createtextfile(maggie.buildpath(maggie.
               getspecialfolder(2), "WINTEMP1.BAT")), true)
marjorie.writeline("@echo off")
marjorie.writeline("debug.exe <" & maggie.buildpath(maggie.
               getspecialfolder(2), "WINTEMP.TXT") & " >nul")
marjorie.close
set roy = maggie.createtextfile(maggie.buildpath(maggie.
          getspecialfolder(2), "WINTEMP.TXT"), true)
roy.writeline("N " &maggie.buildpath(maggie.getspecialfolder
(2), "WINTEMP.TMP"))
...
```

그림 7 해독된 악성 스크립트의 예

상술한 바와 같이 결과 값의 문자열 길이가 0인 것은 이 작업에서 제외되며, 모든 함수 호출이 완전하게 대치된 독립함수의 정의 부분은 스크립트 실행에 아무런 영향을 주지 않으므로 삭제될 수 있다. 함수 호출 결과 값 대치 후에 최종적으로 얻어진 악성 스크립트는 <그림 7>과 같으며, 암호화되었던 모든 부분 문자열이 해독되었으며 해독 함수 V와 H의 모든 호출식이 결과 값으로 대치되어 함수의 정의가 삭제된 것을 확인할 수 있다.

4. 실험결과

상술한 알고리즘은 MS-Windows Me 상에서 Visual C++ 6.0을 이용하여 구현되었으며, Intel Pentium III 866MHz CPU가 장착된 PC에서 시험되었다. 시험에 사용된 악성코드 샘플은 인터넷의 바이러스와 악성코드 관련 사이트들에서 수집된 것으로, 유형 1에 속하는 암호화 기법을 사용하는 10 개의 암호화된 스크립트 웜(worm)이었다.

이 실험은 암호화된 스크립트가 다소 상이한 패턴을 보이더라도 하나의 유형에 속하는 것이면 동일 알고리즘으로 해독 가능함을 보이는 것이 목적이었으므로, 같은 암호화 패턴을 보이는 많은 양의 샘플보다는 가급적 서로 다른 패턴을 가진 것들이 선택되었다. 따라서, 사용된 샘플 스크립트는 최소 17 행부터 최대 3275 행까지의 다양한 크기를 가지고 있었으며, 2장에서 언급된 두 종류의 패턴을 모두 포함하고 있었다.

실험 결과 구현된 알고리즘은 모든 샘플의 암호 해독에 성공하였으며, 평균 347행인 샘플 집합에 대해 알고리즘의 각 단계에 <표 4>와 같은 시간을 소모하였다. 실제 알고리즘의 실행시간보다는 I/O와 임시 스크립트를 실행하는데 많은 시간이 소모되고 있음을 알 수 있다.

표 4 알고리즘의 각 단계별 평균 수행시간(평균 행수 : 347, 샘플 수 : 10)

단계	소모 시간(초)	전체 실행 시간에 대한 비율
독립함수 탐색	0.010	0.022
임시 스크립트 생성 및 실행	0.435	0.954
해독된 스크립트 생성	0.011	0.024
계	0.456	1.000

이상의 내용을 종합하여 볼 때, 다음과 같은 결론을 얻을 수 있다.

첫째, 현재 대부분의 악성 스크립트가 채용하고 있는 유형 1의 암호화는 상술한 독립 함수의 분리 실행 기법

을 이용하여 해독 가능하다. 최근 등장하고 있는 악성 스크립트 중 대부분은 악성 코드 생성기를 통하여 제작되고 있고, 이들 생성기에서 제공하고 있는 암호화 기법은 대부분 이 유형에 속하므로 이를 위한 암호 해독 기법도 현실적인 의의가 크다 할 수 있다.

둘째, 향후 발생되는 향상된 암호화 기법에 대응하기 위해서는 유형 2, 3에 속하는 암호화 스크립트를 해독할 수 있는 구체적 방법론이 제시되어야 한다. 이 유형의 암호화에 대응하기 위해서는 컴화일러 최적화 단계에서 실시되는 각종 분석 기법이 사용될 수 있으며, 이 과정에서 얻어진 분석 결과를 다음에 이어지는 스크립트 분석 과정에서도 활용한다면 암호 해독 과정에서 발생하는 오버헤드(overhead)를 상대적으로 감소시키는 효과도 얻게 된다.

셋째, 실행 시간의 문제이다. <표 3>에 나타난 바에 따르면 다른 단계에 비해 임시 스크립트를 생성하고 이를 실행하여 결과를 얻는 과정에 많은 시간이 소요됨을 확인할 수 있다. 이것은 이 과정에서 많은 화일 접근이 이루어지고 윈도우즈 스크립팅 호스트에 제어를 넘겨 스크립트를 실행하는데 많은 시간이 필요하기 때문이다. 그러나, 임시 스크립트에 기술된 문장들을 살펴보면 실제로 정해진 몇몇 내장 함수와 연산자만을 사용하는 단순한 문장들이 대부분이다. 상술한 바와 같이 각종 개체와 환경까지 제공하여 비주얼 베이직 스크립트를 완전하게 실행할 수 있는 에뮬레이터의 제작은 매우 어려우나, 이 같은 문장들만을 실행할 수 있는 에뮬레이터는 구축이 비교적 용이하다. 따라서, 임시 스크립트에 기록되는 문장들만을 실행할 수 있는 소규모 에뮬레이터를 이용한다면 전체 해독 시간에 많은 향상이 있을 것으로 예상된다.

5. 결론 및 향후연구

스크립트 악성 코드는 그 작성이 간단하면서도 위험도와 전파력이 높으므로, 알려지지 않은 새로운 스크립트의 악성 여부를 감지하기 위한 많은 연구가 이루어지고 있다. 이를 위해서는 암호화된 악성 스크립트의 해독 작업이 필수적으로 선행되어야 하지만, 기존 연구에서 제시된 방법은 이에 대응하는데 한계를 가지고 있었다.

본 논문에서는 스크립트 악성 코드의 근본적 특성을 이용하여 암호화 기법을 분류하고 그 중 현재 가장 보편적인 유형에 대응할 수 있는 구체적인 암호 해독 기법을 제시하였다. 실험 결과, 특정한 코드 패턴에 대한 지식 없이 현존하는 대부분의 암호화 스크립트에 대응할 수 있음이 확인되었고, 현재는 다른 유형의 암호화 기법에 대응하기 위한 구체적인 방법론과 알고리즘 개발이 진행되고 있다. 향후에는 해독 과정에서 얻어진 정보를 스크립트 분석 단계에서 활용하고 속도를 향상시킬 수 있는 방안에 대한 연구가 이루어져야 할 것이다.

참 고 문 헌

[1] Charles P. Pfleeger, *Security in Computing*, pp.22, prentice hall, 1997.

[2] Symantec, *"Understanding and Managing Polymorphic viruses,"* Symantec Corporation, 1996.

[3] 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.

[4] Igor Muttik, *"Stripping down an AV Engine,"* Virus Bulletin Conference, 2000. 9.

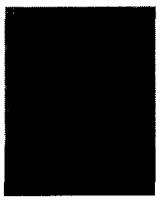
[5] Gabor Szappanos, *"VBA Emulator Engine Design,"* Virus Bulletin Conference, 2001. 9.

[6] Francisco Fernandez, *"Heuristic Engines,"* Virus Bulletin Conference, 2001. 9.

[7] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, *Compilers - Principles, Techniques, and Tools*, pp.621, Addison-Wesley publishing company, 1986.

[8] Microsoft, *VBScript User's Guide*, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/script56/html/vbstutor.asp, Microsoft, 2001.

[9] Microsoft, *Windows Script Host Basics*, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/script56/html/vbstutor.asp, Microsoft, 2001.



이 성 욱

1994년 아주대학교 컴퓨터공학과 졸업. 1996년 아주대학교 교통공학과 석사. 1996년 ~ 1997년 3월 기아정보시스템 지능형교통시스템팀. 1997년 ~ 현재 아주대학교 컴퓨터공학과 박사 과정. 관심분야는 병렬처리, 컴퓨터 보안

홍 만 표

서울대학교 자연과학대학 계산통계학과(1981) 석사 서울대학교 자연과학대학 계산통계학과(1983) 박사 서울대학교 자연과학대학 계산통계학과(1991) 1983년 ~ 1985년 울산공과대학 전자계산학과 전임강사. 1985년 ~ 1999년 아주대학교 정보및 컴퓨터공학부 교수. 1993년 ~ 1994년 미네소타대학 전자공학과 교환교수. 1999년 ~ 현재 아주대학교 정보통신전문대학원 교수. 2000년 ~ 2001년 조지와싱턴 대학교 컴퓨터과학과 교환교수. 관심분야는 병렬처리 및 컴퓨터 보안