폴리몰픽 쉘코드 탐지 기술 동향

The Recent Trends of Polymorphic Shellcodes Detection Technologies

김대원 (D.W. Kim) 보안게이트웨이연구팀 연구원
김익균 (I.K. Kim) 보안게이트웨이연구팀 선임연구원
오진태 (J.T. Oh) 보안게이트웨이연구팀 팀장
장종수 (J.S. Jang) 보안응용그룹 그룹장

목 차

공격자의 주요 목적은 원격 호스트의 제어 권한을 얻는 것이다. 이것은 공격자가 원격 호스트의 컨트롤 플로를 변경시켜 악의적인 코드를 임의로 실행시킬 수 있는 취약한 서비스가 존재하기 때문에 가능하다. 공격자들이 원격 호스트의 제어 권한을 얻기 위한 일반적인 방법은 취약한 서비스를 대상으로 쉘코드(shellcode) 전송을 통해서이다. 네트워크 기반 최신의 공격 탐지 기술들이 점점 사용영역을 넓혀가면서 이를 회피하기 위해 쉘코드들도 진화를 계속하고 있으며, 최근 2~3년 전부터 폴리몰픽(polymorphism)과 메타몰픽(metamorphism) 기법의 사용이 활발해지고 있다. 본 고에서는 이 중, 쉽게 이용할 수 있는 엔진들이 많이 알려져 있어 그 예상 피해가 심각하리라 생각되는 폴리몰픽 형태의 쉘코드가 가지는 특징 및 이를 탐지하기 위한 최신 기술들을 소개한다.

I. 서론

인터넷의 급격한 성장에 따라 이를 대상으로 하는 공격들로 인한 피해 규모도 기하급수적으로 증가하고 있다[1]. 오늘날 인터넷에 연결된 호스트들은 매일 수십, 수백 건의 공격 시도를 받고 있으며 공격의 형태도 점점 복잡해지고 다양해지고 있다[2]. 지난 몇 년간은 최신 공격 탐지 기술들의 발달로 대규모 피해를 유발하는 공격들이 적절히 차단되고 있었지만, 이를 회피하기 위해 공격들은 다양한 형태로 진화하고 있다.

네트워크 기반 탐지 시스템을 회피하기 위한 대표적인 방법들로는 폴리몰픽 방법(polymorphism)과 메타몰픽 방법(metamorphism)이 있다. 폴리몰픽 방법은 대상 호스트(target host)의 취약점을 악용하여 제어권을 얻기 위해, 공격 호스트(attack host)가 전송하는 쉘코드를 특정 키 값을 이용하여 인크립션(encryption)하고, 그 쉘코드 내에 디크립션(decryption) 코드를 같이 삽입하게 된다. 간단한 폴리몰픽 엔진들은 탐지가 쉬운 디크립션 코드를 생성하지만, 최신 엔진[3]-[6]들은 메타몰픽 방법을 접목하여 디크립션 코드를 찾아내는 것을 어렵게 만들고 있다.

최근의 네트워크 보안 관련 주요 이슈 중의 하나는 복잡한 기법이 사용된 폴리몰픽 쉘코드를 탐지하는 것이다. 그러나 현재 사용되고 있는 탐지 방법의 수준은 Metasploit과 같은 사이트에서 쉽게 구할 수 있는 폴리몰픽 엔진들조차도 적절히 탐지를 할 수가 없다. 본 고는 폴리몰픽 쉘코드들이 가지는 고유한 특징에 대해 설명하고 이를 탐지하기 위한 방법에 대한 관련 연구와 최근의 탐지 방법 몇 가지를 소개할 것이다.

II. 폴리몰픽 쉘코드의 특징

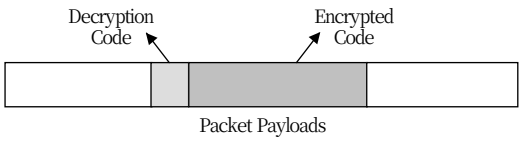
일반적으로 알려져 있는 폴리몰픽 쉘코드는 (그림 1)과 같은 형태로서, 인크립트(encrypt)된 원래의 쉘코드가 있고 이를 디크립트(decrypt)하는 코드로 구성된다. 따라서, 인크립트된 코드의 주소를 찾고, 이를 이용하여 인크립트된 값을 READ한 후, 디크립트 하여 다시 WRITE하는 과정으로 진행된다. 이 특징은 폴리몰픽 쉘코드 탐지에 관한 많은 연구들이 기본적으로 이용하는 것이다.

그러나 모든 폴리몰픽 쉘코드들이 이와 같은 특징을 따르지는 않는다. (그림 2)는 폴리몰픽 쉘코드들의 특징에 대해 분류해 놓은 것이다.

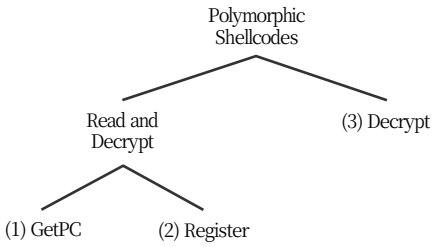
폴리몰픽 쉘코드의 전형적인 특징은 (1)의 과정을 따른다. 여기에서 GetPC는 현재 디크립션 코드의 주소 값을 얻는 코드로서 인크립트된 코드의 주소를 찾는 데 이용된다. GetPC를 이용하여 현재 주소 값을 알아내고, 이 값을 이용해서 인크립트된 코드를 READ하고, 그 READ한 값을 디크립트하여 다시 메모리 상에 WRITE를 하게 된다.

반면에, (2)의 과정을 따르는 폴리몰픽 쉘코드는 GetPC를 사용하지 않는다. 폴리몰픽 쉘코드 제작자(programmer)는 대상 호스트의 취약성을 더욱 정교하게 분석하여 폴리몰픽 쉘코드가 침입한 시점의 모든 상태 정보들을 이용하게 된다. 현재 폴리몰픽 쉘코드가 위치한 주소 값을 현재 레지스터 값 등을 이용하여 알아내고, 이 값을 이용하여 (1)과 같은 동작을 수행한다.

(3)의 경우는 폴리몰픽과 메타몰픽의 중간 형태를 띄는 것으로 폴리몰픽 쉘코드의 시작 주소로 인스트



(그림 1) 폴리몰픽 쉘코드의 일반적인 형태



(그림 2) 폴리몰픽 쉘코드들의 특징 분류

럭션 포인터(instruction pointer, program counter)가 이동된 시점부터는 메모리를 READ하는 과정 없이 바로 디크립션만 수행하게 된다.

III. 관련 연구

Snort[7]와 같은 시그니처 기반(signature-based) 네트워크 침입 탐지 시스템(NIDS)들은 기존에 알려진 쉘코드의 특정 부분, 연속된 NOP 슬레드(sled) 등과 같이 공통적인 실제 콘텐츠를 시그니처화 하여 공격을 탐지해 왔다. 그러나, 시그니처 기반 탐지 기법들을 회피하기 위해 쉘코드들은 간단한 폴리몰픽 기법으로 암호화 되었으며, 연속된 NOP 슬레드 부분들은 메타몰픽 기법에 의해 다양하게 변형되었다. 특히 윈도 시스템을 대상으로 하는 쉘코드의 경우는 레지스터 스프링 기법[8]을 통해 슬레드를 사용하지 않는 것들이 나타났다.

레지스터 스프링 기법을 이용한 공격들은 쉘코드에서 윈도의 취약한 리턴 주소 값을 탐지하는 방법[9]과 지속적인 윈도 보안 패치들에 의해 탐지 및 차단이 가능하였다. 그러나, 새로운 공격들은 변형된 슬레드와 변형된 암호화 키 값을 사용하기 때문에 시그니처 기반으로는 여전히 탐지가 힘들다.

새로운 쉘코드를 탐지하기 위해 Polygraph[10], PAYL[11], PADS[12] 등과 같이 통계적인 방법으로 공통적인 스트링(string)들을 분석하는 방법들이 연구되었지만, 간단히 변형된 쉘코드들은 탐지가 가능할지 몰라도 복잡한 형태로 변형된 쉘코드들에는 여전히 취약하였다. 또한, 네트워크 패킷들에서 공통적인 스트링들의 발생빈도를 측정하는 방법들이기 때문에 공격 대상이 정해져 있는 소규모의 공격들에는 효과적이지 못하다.

변형된 슬레드 및 암호화된 쉘코드에 대해 시그니처 기반 및 공통적인 스트링들의 발생 빈도를 이용한 탐지 방법들의 한계가 드러나면서, 최근의 연구들은 쉘코드를 찾기 위해 패킷의 페이로드에 대해 정적 바이너리 코드 분석 방법들[13]-[15]을 적용하기 시작하였다. 그러나 이런 방법들은 정상적인 데이터와 공격 데이터에 대한 학습에 따라 탐지 오차가 아주 크며, 자기 수정 코드(self-modifying code)와 같은 정적 분석 방법을 방지시키는 기법(static analysis resistant method)들에 의해 무력화 될 수 있다.

IV. 정적 분석 방지 기법

네트워크 단에서 알려지지 않은 폴리몰픽 쉘코드를 탐지하기 위해 정적 바이너리 코드 분석 기법을 기반으로 연구 노력들이 있어 왔다. 그러나, 프로그램 카운터(program counter)가 슬레드 부분을 거쳐 쉘코드에 도달한 시점부터 쉘코드 제작자는 쉘코드 구성에 대한 완벽한 권한을 갖게 되기 때문에 정적인 분석을 방지하기 위해 복잡한 형태의 쉘코드를 작성하게 된다.

이 장에서는 네트워크 단에서 수행되는 정적인 바이너리 코드 분석을 회피할 수 있는 최근의 폴리몰픽 쉘코드의 특징을 살펴보기로 한다. 본 고는 간단한 예제 코드를 제시할 것이며, 더욱 정교한 형태의 방법들도 존재한다.

1. 역어셈블 방지(Thwarting Disassembly)

바이너리 코드 분석에는 선형 역어셈블(linear disassembly)과 재귀 역어셈블(recursive disassembly) 이 두 가지가 주로 사용된다. 선형 역어셈블은 무효한(invalid) 인스트럭션이나 데이터의 끝에 도달할 때까지 차례대로 각각의 인스트럭션들을 해석(decode)하고, 재귀 역어셈블은 분기문(branch)의 목적지 주소를 따라 해석을 하게 된다.

(그림 3)은 Metasploit framework 프로그램의 countdown 암호화 엔진으로 생성된 쉘코드의 디크립션 코드 부분의 역어셈블된 모습을 보여 준다. 0x0003 주소의 call 인스트럭션은 실제 수행을 하게 되면 0x0007 주소로 분기되고, 그 주소부터 인스트럭션이 해석된다. 그러나 (a)의 선형 역어셈블

```
0000    6A7F            push  0x7F
0002    59              pop   ecx
0003    E8FFFFFFFF      call  0x7
0008    C15E304C        rcr   [esi+0x30], 0x4C
000C    0E              push  cs
000D    07              pop   es
000E    E2FA            loop  0xA
0010
...     <encrypted payload>
008F
```

(a) 선형 역어셈블

```
0000    6A7F            push  0x7F
0002    59              pop   ecx
0003    E8FFFFFFFF      call  0x7
0007    FFC1            inc   ecx
0009    5E              pop   esi
000A    304C0E07        xor   [esi+ecx+0x7], cl
000E    E2FA            loop  0xA
0010
...     <encrypted payload>
008F
```

(b) 재귀 역어셈블

(그림 3) Countdown 엔진으로 생성된 쉘코드

의 경우는 분기를 반영하지 않기 때문에 call을 해석한 후, 0x0008 주소에서 rcr 인스트럭션을 해석하는 오류를 범하게 된다. 반면 (b)의 재귀 역어셈블의 경우는 call에 의한 분기를 따라 0x0007 주소의 inc ecx를 정확히 해석한다.

그렇지만, 재귀 역어셈블의 경우에도 항상 정확한 지점에서 해석을 한다는 보장이 없다. 쉘코드는 역어셈블을 방지하기 위해 분기의 목적지 주소 부분에 절대 값이 아닌 상대적인 주소 값을 이용할 수 있기 때문이다. 이 경우는 상대 주소 값을 담고 있는 레지스터에 대한 연산이 먼저 이루어져야 하고, 분기문의 실행에 반영되어야 한다.

2. 자기 수정 코드

역어셈블의 이런 단점을 보완하기 위해 쉘코드의 대략적인 컨트롤 플로 그래프(CFG)를 추출하는 방법[14],[16]들이 연구되었다. 이 방법은 쉘코드의 구조적인 특징을 분석하는 것으로서 상대적인 주소를 이용하는 분기에 대해 대략적인 분석을 수행하여 인스트럭션들의 흐름에 대한 특징을 추출하게 된다.

그러나, 이런 분석 방법도 자기 수정 코드 기법을 사용하는 쉘코드들을 탐지하는 데 한계가 있다. 이

```
0000  6A7F            push 0x7F
0002  59              pop ecx
0003  E8FFFFFFFF      call 0x7
0007  FFC1            inc ecx
0009  5E              pop esi
000a  80460AE0        add [esi+0xA], 0xE0
000e  304C0E0B        xor [esi+ecx+0xB], cl
0012  02FA            add bh, dl
0014
...   <encrypted payload>
0093
```

(a) 폴리몰픽 쉘코드를 생성한 모습

```
0000  6A7F            push 0x7F
0002  59              pop ecx                 ;ecx = 0x7F
0003  E8FFFFFFFF      call 0x7                ;PUSH 0x8
0007  FFC1            inc ecx                 ;ecx = 0x80
0009  5E              pop esi                 ;esi = 0x8
000a  80460AE0        add [esi+0xA], 0xE0     ;ADD [0012] 0xE0
000e  304C0E0B        xor [esi+ecx+0xB], cl   ;XOR [0093] 0x80
0012  E2FA            loop 0xE                ;ecx = 0x7F
000e  304C0E0B        xor [esi+ecx+0xB], cl   ;XOR [0092] 0x7F
0012  E2FA            loop 0xE                ;ecx = 0x7E
...
```

(b) 런타임 때의 실제 코드 모습

(그림 4) 수정된 Countdown 엔진을 이용

기법은 스스로의 코드를 런타임(run-time)에 동적으로 변경하게 된다. 따라서, CFG에 의해 추출된 흐름과 런타임 때의 흐름과는 차이가 있게 된다. (그림 4)는 수정된 countdown 엔진으로 작성된 폴리몰픽 쉘코드에 대해 생성된 폴리몰픽 쉘코드 모습과 런타임 때의 실제 코드 모습에 대한 간단한 예제를 보여준다.

(그림 4)의 (a)를 보면 폴리몰픽 쉘코드의 특징이 쉽게 발견되지 않는다. 그러나 0x000a 주소의 add [esi+0xA], 0xE0는 0x0012 주소의 add bh, dl을 수정하여 loop 0xE를 생성한다. 이 loop에 의해 폴리몰픽 쉘코드 내의 인크립트된 실행 코드가 디크립트 되는 것이다. (b)는 (a)가 실행되어 자기 수정을 한 후의 실제 쉘코드 모습을 나타낸다.

Ⅴ. 동적 분석 탐지 방법

자기 수정 코드 기법이 사용된 폴리몰픽 쉘코드의 경우 기존의 정적 분석 방법으로는 탐지의 한계가 있다. 따라서, 최근의 연구 추세는 동적 분석 방법을 사용하거나 정적 분석과 동적 분석을 병합한

형태로 진행되고 있다. 본 고는 이와 관련된 최근의 연구들에 대해 설명할 것이다.

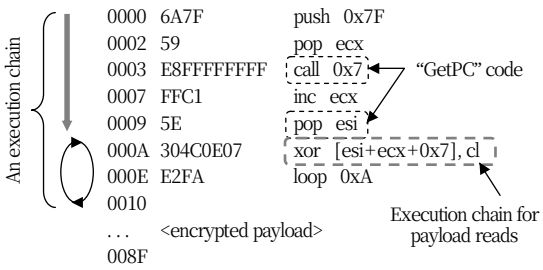
1. Polychronakis's First Method

본 연구[17]는 자기 수정 코드 기법으로 작성된 폴리몰픽 쉘코드도 탐지할 수 있도록 모든 인스트럭션 시퀀스(instruction sequence)를 실행해 보기 위해 CPU 에뮬레이터(emulator)를 이용한다. 그 에뮬레이터는 네트워크에 연결된 호스트들의 CPU 중 가장 많이 사용되고 있는 IA-32의 많은 부분을 에뮬레이션 한다.

CPU 에뮬레이터를 이용하여 매 바이트마다 인스트럭션들을 에뮬레이션 한다. 수행중 입력된 데이터를 벗어나는 무효한 메모리 지역을 액세스하거나 무효한 인스트럭션이 나타나면 다음 바이트부터 에뮬레이션을 수행한다.

만약, 수행되는 인스트럭션 시퀀스가 체인(chain) 형태라면 그 체인 내에서 유효한 메모리를 READ하는 횟수를 측정한다. (그림 5)는 인스트럭션 체인의 의미를 설명하고 반복적으로 메모리 READ를 하는 폴리몰픽 쉘코드의 예이다.

(그림 5)에서 0x0000 주소의 push 0x7F를 시작으로 정상적으로 에뮬레이션 되고 있는 인스트럭션 시퀀스는 0x000E 주소의 loop 0xA에 의해 0x000A 주소와 0x000E 주소가 반복적으로 수행되는 체인 형태를 나타낸다. 그 체인 속의 xor에 의해 메모리가 반복적으로 READ되고, 이 횟수가 설정된 탐지 값을 넘어서면 폴리몰픽 쉘코드로 판단한다.



(그림 5) 인스트럭션 체인의 의미를 설명하고 반복적으로 메모리 READ를 하는 폴리몰픽 쉘코드

오탐(false positive)을 줄이기 위하여 GetPC 코드가 시퀀스 내에 존재하는지를 함께 탐지한다. GetPC 코드는 현재 실행되는 인스트럭션의 주소를 스택(stack)에 저장하고, 특정 레지스터로 그 값을 읽어 들여 인크립트된 코드의 주소를 찾는 데 사용한다. (그림 5)의 예에서는 call 0x7과 pop esi가 GetPC 동작을 수행한다.

본 연구는 자기 코드 수정 기법을 이용할 폴리몰픽 쉘코드를 탐지할 수 있지만, 반면에 매 바이트마다 수행되는 에뮬레이션에 의해 오버헤드가 너무 크다는 단점이 있다.

2. Zhang's Method

본 연구[18]는 Polychronakis 방법의 에뮬레이션 오버헤드 문제를 해결하려 한다. 정적인 역어셈블을 통해 우선 GetPC 코드를 찾은 후, 그 코드의 주변 인스트럭션에 대해서만 에뮬레이션을 수행하는 하이브리드(hybrid) 형태의 탐지 방법을 제안하고 있다.

정적 분석을 통해, GetPC 코드를 구성하는 인스트럭션들 중 현재 수행되는 주소를 스택에 저장하는 call, fnstenv 등의 seed 인스트럭션을 찾는다. (그림 6)은 ShikataGaNai 엔진에 의해 생성된 폴리몰픽 쉘코드와 실행 후의 코드 모습을 보여준다.

(그림 6)의 (a)에서 정적 분석에 의해 0x0006 주소에서 fnstenv가 탐지된다. 그 주소부터 재귀 역어셈블을 이용하여 메모리를 READ하는 xor [ebx+15], edi 인스트럭션을 탐지한다. 차후에 수행할 에뮬레이션이 올바로 이루어지기 위해서는 xor 인스트럭션에 사용된 ebx, edi 레지스터에 대한 값이 정의되어 있는지를 탐색해야 한다. xor 인스트럭션이 탐지된 역방향으로 해당 인스트럭션이 정의되어 있는지를 판단한다. ebx는 0x000F, edi는 0x000A 주소의 인스트럭션에서 값이 할당되어 있다.

만약, seed 인스트럭션 전까지 레지스터가 모두 정의되어 있지 않다면, seed 인스트럭션을 기준으로 다시 역방향 탐지를 수행한다. 이때, seed 인스

```
0000    31 c9                   xor ecx, ecx
0002    da c7                   fcmovb st(0), st(7)
0004    b1 23                   mov cl, 23
0006    d9 74 24 f4             fnstenv 14/28byte[esp-0c]
000A    bf 78 0f 5e f3          mov edi, f35e0f78
000F    5b                      pop ebx
0010    31 7b 15                xor [ebx+15], edi
0013    03 7b 15                add edi, [ebx+15]
0016    83 bb 0b bc 06 c7 fc    cmp [ebx+c706bc0b], -4

...     <encrypted payload>
```

(a) 폴리몰픽 쉘코드

```
0000    31 c9                   xor ecx, ecx
0002    da c7                   fcmovb st(0), st(7)
0004    b1 23                   mov cl, 23
0006    d9 74 24 f4             fnstenv 14/28byte[esp-0c]
000A    bf 78 0f 5e f3          mov edi, f35e0f78
000F    5b                      pop ebx
0010    31 7b 15                xor [ebx+15], edi
0013    03 7b 15                add edi, [ebx+15]
0016    83 c3 04                add ebx, 4
0019    e2 f5                   loop 0010
...     <encrypted payload>
```

(b) 실행 후의 코드 모습

(그림 6) ShikataGaNai 엔진에 의해 생성된 모습

트럭션의 주소에서 한 바이트씩 감소시키면서 역어셈블을 수행하여 seed 인스트럭션을 덮어쓰지 않는 인스트럭션들을 찾아낸다. 이 인스트럭션들 중에서 나머지 레지스터들이 정의된 시퀀스를 찾게 된다. 만약, 레지스터들이 아직 모두 정의되지 않는다면, 넓이 우선 탐색(breadth first search)을 계속 수행하여 레지스터들이 모두 정의될 때까지 트리(tree)를 구성해 나간다. 이 중, 마지막에 탐지된 주소가 가장 낮은 트리가 에뮬레이션을 수행할 트리가 된다.

(a)에서는 seed 인스트럭션 전에, 모든 레지스터가 정의되었으므로 fnstenv의 시맨틱(semantic)을 어떻게 해석하느냐에 따라 seed 인스트럭션 역방향으로 분석을 더 하든가, 아니면 0x0006 주소에서 에뮬레이션을 시작할 수 있다. 에뮬레이션이 시작되면 (b)에서 fnstenv에 의해 스택 상에 현재 주소가 저장되고, 그 값은 pop에 의해 ebx에 저장된다. xor [ebx+15], edi에 의해 자신의 코드가 수정이 되며, 인크립트된 코드를 디크립트 하게 된다. 디크립트 과정은 0x0010과 0x0019 주소 사이에서 반복된다.

에뮬레이션을 통해 탐지된 체인 시퀀스 내에 상대적인 주소를 이용하는 메모리 WRITE 인스트럭션이 존재하고, 해당 레지스터들이 업데이트 된다면 이는 폴리몰픽 쉘코드로 탐지하게 된다.

본 연구는 seed 인스트럭션을 먼저 탐지하기 때문에 순수한 에뮬레이션에 비해 오버헤드가 크게 줄어들지만, seed 인스트럭션이 없는 경우는 탐지할 수 없으며, loop 형태가 아닌 폴리몰픽 쉘코드도 탐지할 수 없다. 또한, 일반적인 데이터에도 call과 같은 seed 인스트럭션은 아주 많기 때문에 분석 오버헤드를 줄이기 위해 더욱 정교한 방법으로 개선되어야 한다.

3. Polychronakis's Second Method

본 연구[19]는 seed 인스트럭션이 없는 경우와 loop가 없는 폴리몰픽 쉘코드를 탐지하기 위한 방법을 제시한다. Polycronakis의 첫번째 방법과 같이 매 바이트마다 에뮬레이션을 수행하되, 인스트럭션 시퀀스 상에서 고유한 주소로의 메모리 WRITE 횟수와 그 주소 상에서 실행된 인스트럭션의 개수가 설정된 값 이상이면 폴리몰픽 쉘코드로 판단한다.

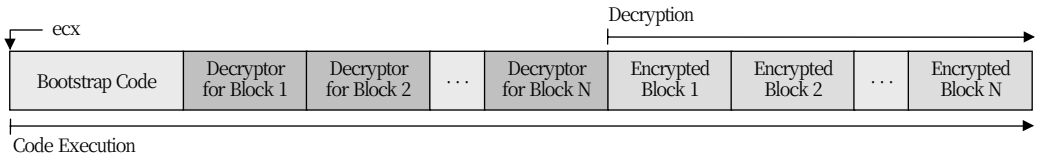
(그림 7)은 Avoid UTF8/tolower 엔진으로 생성된 seed 인스트럭션이 없는 폴리몰픽 쉘코드와 스택에서의 배치 상태를 보여준다.

(그림 7)의 폴리몰픽 쉘코드는 인크립트된 코드의 주소를 찾기 위해 seed 인스트럭션을 사용하지 않는다. 대상 호스트 내에 위 코드가 전송이 되어 스택 상에 놓이게 되면, 디크립트 코드의 시작 부분의 주소는 이미 ecx 레지스터에 저장되어 있는 상태다. 쉘코드 제작자는 디크립트 코드가 실행될 시점에 ecx 레지스터에 현재 주소 값이 들어 있다는 정보를 취약점 분석을 통해 알고 있었기 때문에 seed 인스트럭션을 사용할 필요가 없는 것이다.

ecx에 저장된 현재 주소 값은 인크립트된 코드들을 찾기 위한 레퍼런스(reference)로 사용이 된다. (그림 7)의 예는 디크립션 loop를 탐지하는 것을 회피하기 위해 loop를 사용하지 않고 선형적인 시퀀스로 디크립트를 한다. (그림 7b)를 보면, 각각의 디크립션 블록들이 각각의 인크립트된 코드들을 해석하는 것을 알 수 있다. 마지막 디크립션 블록의 수행이

```
0   60000000   6A20          push 0x20              ;  ecx points here
1   60000002   6B3C240B      imul edi, [esp], 0xb   ;  edi = 0x160
2   60000006   60            pusha                  ;  push all registers
3   60000007   030C24        add ecx, [esp]         ;  ecx = 0x60000160
4   6000000a   6A11          push 0x11
5   6000000c   030C24        add ecx, [esp]         ;  ecx = 0x60000171
6   6000000f   6A04          push 0x4               ;  encrypted block size
7   60000011   6826191413    push 0x13141926
8   60000016   5F            pop edi                ;  edi = 0x13141926
9   60000017   0139          add [ecx], edi         ;  [600000171] = "ABCD"
10  60000019   030C24        add ecx, [esp]         ;  ecx = 0x60000175
11  6000001c   6817313F1E    push 0xle3f3117
12  60000021   5F            pop edi                ;  edi = 0x1E3F3117
13  60000022   0139          add [ecx], edi         ;  [60000175] = "EFGH"
14  60000024   030C24        add ecx, [esp]         ;  ecx = 0x60000179
```

(a) Seed 인스트럭션이 없는 폴리몰픽 쉘코드



(b) 스택에서의 배치 상태

(그림 7) Avoid UTF8/Tolower 엔진으로 생성

끝나면 해석된 코드들로 실행이 바로 연결된다.

(그림 7)과 같이 대상 호스트에서 수행중인 레지스터 값을 이용하는 폴리몰픽 쉘코드는 seed 인스트럭션이 없기 때문에 기존의 방법으로 탐지가 불가능하다. 본 방법은 이와 같은 폴리몰픽 쉘코드들을 찾기 위해 매 바이트마다 에뮬레이션을 수행할 때, 8개의 32비트 레지스터 중의 하나를 현재 주소로 할당하고 나머지들을 랜덤(random)한 값으로 설정하여 에뮬레이션을 반복하게 된다. 즉, 매 바이트 에뮬레이션을 수행할 때 8번을 수행하게 된다.

현재의 인스트럭션 시퀀스에서 메모리 WRITE가 중복되지 않은 주소 상에 특정 횟수 이상 수행되고, 이렇게 WRITE된 값들이 특정 횟수 이상 인스트럭션으로 해석되어 실행된다면 폴리몰픽 쉘코드로 판단하게 된다.

본 연구는 현재까지 알려져 있는 대부분의 폴리몰픽 쉘코드를 탐지할 수 있는 장점이 있지만, 성능상의 오버헤드가 가장 크다는 단점이 있다. 또한 실제 (그림 7)과 같이 seed 인스트럭션이 필요 없는 취약성의 경우는 굉장히 드물기 때문에 모든 종류의 폴리몰픽 쉘코드의 탐지 목적으로는 성능 문제로 인해 효용성 측면이 떨어지게 된다.

VI. 결론

본 고에서는 폴리몰픽 쉘코드의 기본적인 특징을 소개하고 이를 분류하였으며, 이 특징을 이용한 탐지 방법들과 이를 회피하기 위한 폴리몰픽 쉘코드의 발전 양상에 대해 설명하였다.

앞서 소개된 방법들로 현재까지 알려진 대부분의 폴리몰픽 쉘코드는 탐지가 가능하지만 이를 위한 성능 상의 문제는 아직 해결되지 않은 상태이다. 또한 분석을 위해 입력 받은 패킷 외에 대상 호스트의 메모리에 존재하는 내용들을 이용하여 디크립션을 수행하는 공격들도 발생할 가능성을 배제할 수 없다. 현재는 호스트 정보를 이용하는 폴리몰픽 쉘코드를 네트워크 레벨에서 탐지할 수 있는 방법은 없는 상태이다. 다만, 이런 폴리몰픽 쉘코드는 제작하는 입

● 용 어 해 설 ●

GetPC: Get Program Counter의 약자. 공격자는 쉘코드가 놓일 절대주소를 미리 알 수 없기 때문에 그 주소를 알기 위한 인스트럭션 패턴을 사용한다.

슬레드(Sled): 쉘코드의 메인코드로 컨트롤 플로를 이동시키기 위한 일련의 인스트럭션들

장에서도 요즘은 대부분의 OS들이 프로세스의 주소 공간의 위치를 랜덤화하기 때문에 쉽게 개발할 수는 없는 상태이다.

약 어 정 리

CFG Control Flow Graph
NIDS Network Intrusion Detection System

참 고 문 헌

[1] Computer Economics, http://www.computereconomics.com

[2] V. Yegneswaran, P. Barford, and J. Ullrich, "Internet Intrusions: Global Characteristics and Prevalence," *In Proc. of the 2003 ACM SIGMETRICS Int'l Conf. on Measurement and Modeling of Computer Systems*, 2003, pp.138–147.

[3] Metasploit, http://www.metasploit.com/

[4] TAPiON, http://pb.specialised.info/all/tapion/

[5] ADMmutate, http://www.ktwo.ca/ADMmutate-0.8.4.tar.gz.

[6] "Alpha 2," http://www.edup.tudelft.nl/~bjwever/src/alpha2.c

[7] Snort, http://www.snort.org.

[8] J.R. Crandall, S.F. Wu, and F.T. Chong, "Experiences Using Minos as a Tool for Capturing and Analyzing Novel Worms for Unknown Vulnerabilities," *In Proc. of the Conf. on Detection of Intrusions and Malware & Vulnerability Assessment(DIMVA)*, July 2005.

[9] A. Pasupulati, J. Coit, K. Levitt, S. Wu, S. Li, J. Kuo, and K. Fan, "Buttercup: On Network based Detection of Polymorphic Buffer Overflow Vulnerabilities," *In Proc. of the Network Operations and Management Symp.(NOMS)*, Apr. 2004, pp.235–248.

[10] J. Newsome, B. Karp, and D. Song, "Polygraph: Automatically Generating Signatures for Polymorphic Worms," *In Proc. of the IEEE Security & Privacy Symp.*, May 2005, pp.226–241.

[11] K. Wang and S.J. Stolfo, "Anomalous Payload-based Network Intrusion Detection," *In Proc. of the 7th Int'l Symp. on Recent Advanced in Intrusion Detection(RAID)*, Sep. 2004, pp.201–222.

[12] Y. Tang and S. Chen, "Defending Against Internet Worms: a Signature-based Approach," *in Proc. of the 24th Annual Joint Conf. of IEEE Computer and Commun. Societies(INFOCOMM)*, 2005.

[13] P. Akritidis, E. Markatos, M. Polychronakis, and K. Anagnostakis, "STRIDE: Polymorphic Sled Detection through Instruction Sequence Analysis," *In Proc. of the 20th IFIP Int'l Information Security Conf.(SEC'05)*, June 2005, pp.375–392.

[14] R. Chinchani and E. Berg, "A Fast Static Analysis Approach to Detect Exploit Code Inside Network Flows," *In Proc. of the 8th Int'l Symp. on Recent Advances in Intrusion Detection(RAID'05)*, Sep. 2005, pp.284–308.

[15] X. Wang, C. Pan, P. Liu, and S. Zhu. SigFree, "A Signature-free Buffer Overflow Attack Blocker," *In Proc. of the 15th USENIX Security Symp.*, July 2006, pp.225–240.

[16] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Polymorphic Worm Detection Using Structural Information of Executables," *In Proc. of the Int'l Symp. on Recent Advances in Intrusion Detection(RAID)*, Sep. 2005.

[17] M. Polychronakis, K. Anagnostakis, and E. Makatos, "Network-Level Polymorphic Shellcode Detection Using Emulation," *In Proc. of the Conf. on Detection of Intrusions and Malware & Vulnerability Assessment(DIMVA'06)*, July 2006.

[18] Q. Zhang, D.S. Reeves, P. Ning, and S.P. Lyer, "Analyzing Network Traffic to Detect Self-decrypting Exploit Code," *In Proc. of the ACM Symp. on Information, Computer and Commun. Security (ASIACCS)*, 2007.

[19] M. Polychronakis, K. Anagnostakis, and E. Makatos, "Emulation-based Detection of Non-self-contained Polymorphic Shellcode," *In Proc. of the Int'l Symp. on Recent Advances in Intrusion Detection(RAID)*, 2007.