YARA User's Manual Ver. 1.6

Víctor Manuel Álvarez
victor.alvarez@virustotal.com

YARA Korean User's Manual Ver. 1.6

Hakawati(Woo-seok Choi)
siansia007@gmail.com

ISAC & Research Engineer : http://www.tricubelab.com
Kisec 40th : http://www.kisec.com
Malware Project Member : http://www.boanproject.com
Blog : http://hidka.tistory.com

0. YARA install

YARA를 설치하고 사용하는 방법은 매우 간단합니다. Linux계열들은 소스코드를 다운로드 받고 컴파일 하여 사용할 수 있습니다. Windows계열은 컴파일을 통해 사용 할 수 없고, Python의 모듈을 이용하여 사용 할 수 있습니다.

YARA를 이용한 탐지를 효과적으로(자동화)하기 위해서는 Python 프로그래밍을 연습해야 합니다.

0.1 Yara 1.6 install on Linux

```
~# sudo apt-get install libpcre3-dev g++
~# wget http://yara-project.googlecode.com/files/yara-1.6.tar.gz
~# tar xfz yara-1.6.tar.gz
~# cd yara-1.6
~/yara-1.6# ./configure
~/yara-1.6# make
~/yara-1.6# make check
~/yara-1.6# make install
```

0.2 Yara-Python-1.6 install on Linux

```
~# wget https://yara-project.googlecode.com/files/yara-python-1.6.tar.gz
~# tar xfz yara-python-1.6.tar.gz
~# cd yara-python-1.6
~/yara-python-1.6# python setup.py build
~/yara-python-1.6# python setup.py install
```

Permission 에러가 발생 시 명령줄 앞에 sudo를 사용하시면 됩니다.

0.3 Yara install on Windows

Windows에서는 Python의 Yara모듈로 Yara를 사용 할 수 있습니다. 그래서 Yara가 요구하는 Python의 버전에 맞게 Python을 설치해야 합니다.

다운로드 : https://code.google.com/p/yara-project/downloads/list

```
>>> import yara
>>> rules = yara.compile(filepath="yara 룰 파일 주소")
>>> rules.match("대상 파일 주소")
```

1. YARA in a nutshell

YARA는 악성코드를 연구하는 연구원이 악성 코드의 종류들을 식별하고 분류하는 목적으로 사용하는 도구입니다. YARA를 사용하면 그 종류의 샘플에 포함 된 텍스트 또는 바이너리 정보의 기반으로 악성 코드를 분류할 수 있습니다. YARA는 설명, 이름 규칙 문자열의 집합 그리고 부울식으로 규칙 로직을 결정합니다. 규칙이 설명 된 악성 코드의 종류에 속해 있는지 결정하기 위해 파일 또는 실행 중인 프로세스에 적용 할 수 있습니다.

다음 예와 함께 설명합니다. 다음 두 변종의 악성코드 종류가 있다고 가정합니다. 그 중 하나는 http://bar.com/badfile2.exe 에서 다운로드하고, 다른 하나는 http://foo.com/badfile1.exe 에서 다운로드 합니다. 이 URL들의 악성코드는 하드코딩 되어 있습니다. win.exe 이름으로 다운로드 되어 떨어지는 두 변종의 파일 또한 하드코드의 샘플을 포함하고 있습니다. 이러한 설정 된 것들을 통해 우리는 다음과 같은 규칙을 만들 수 있습니다.

```
rule BadBoy
{
    strings:
        $a = "win.exe"
        $b = "http://foo.com/badfile1.exe"
        $c = "http://bar.com/badfile2.exe"

    condition:
        $a and ($b or $c)
}
```

위의 규칙은 파일 또는 프로세스가 "win.exe"문자열과 두 개의 URL 중 하나를 포함하고 있을 때, BadBoy로 보고하도록 YARA에게 지시합니다.

이는 단지 단순한 예제이며, 더 복잡하고 강력한 규칙은 wild-card, 대소문자를 구분하는 텍스트 문자열, 정규 표현식, 그리고 YARA에서 제공하는 다양한 기능들을 사용하여 만듭니다.

2. Writing rules

YARA의 규칙은 이해하고 쓰기 쉽게하기 위해 C언어의 구조체를 선언하는 방식과 유사한 구문으로 갖추고 있습니다. 다음은 YARA에 어떤 것도 적용하지 않은 아주 간단한 규칙입니다.

```
rule Dummy
{
    condition:
        false
}
```

YARA의 각 규칙은 식별자 규칙을 따르는 키워드 rules로 시작합니다. 식별자는 C 프로그래밍 언어와 동일한 어휘 규칙을 준수해야 하며, 영숫자와 밑줄 문자를 포함 할 수 있지만, 첫 번째 문자는 숫자를 사용해서는 안됩니다. 규칙 식별자는 대소문자를 구분하며 128자를 초과 할 수 없습니다. 다음 키워드들은 예약되어 있으며 식별자로 사용 할 수 없습니다.

all	in	private
and	include	rule
any	index	rva
ascii	indexes	section
at	int8	strings
condition	int32	true
entrypoint	matches	uint8
false	meta	uint16
filesize	nocase	uinit32
fullword	note	wide
for	or	
global	of	

규칙은 일반적으로 두 섹션으로 구분되어 있습니다. 어떤 문자열에도 의존하지 않는 규칙이면, 문자열 정의 및 조건, 문자열 정의 섹션을 생략할 수 있지만 상태 섹션은 항상 필요로 합니다. 문자열 정의 섹션은 정의되어 있는 규칙의 일부 문자열을 가집니다. 식별자는 $ 문자 다음에 영숫자와 밑줄로 구성되며, 해당 문자열을 참조하는 조건 섹션에서 사용 할 수 있습니다. 다음 예제와 같이 문자열은 텍스트나 진수 형태로 정의 할 수 있습니다.

```
rule ExampleRule
{
    strings:
```

```
        $my_text_string = "text here"
        $my_hex_string = { E2 34 A1 C8 23 FB }

    condition:
        $my_text_string or $my_hex_string
}
```

텍스트 문자열은 C언어처럼 따옴표로 묶여 있습니다. 16진수 문자열은 대괄호로 묶으며, 이는 16진수의 연속으로 구성하며 인접한 문자는 공백으로 표시하여 구분합니다. 10진수 숫자는 16진수 문자열로 인식하지 않습니다.

조건 세션의 규칙은 로직에 있습니다. 이 섹션에서 말하고자 하는 것은 파일 또는 프로세스에 조건 세션의 규칙을 충족하는 로직을 포함하며 이는 부울 변수로 표현합니다. 일반적으로 문자열 식별자를 사용하여 이전에 정의 된 문자열 찾습니다. 부울 변수의 역할은 문자열 식별자를 통해 파일이나 프로세스 메모리 등에서 찾은 문자열과 비교해 true 또는 false로 평가합니다. 이렇게 문자열 식별자의 문자열이 파일이나 프로세스, 메모리 또는 기타 등등에 의해서 부울 변수로 인한 true로 발견되었다면 설정한 규칙대로 역할을 수행합니다.

2.1 Comments

YARA 규칙에서 주석을 추가할 경우 C 소스파일에서 단일 라인과 여러 라인에 주석을 다는 것과 같이 주석을 지원합니다.

```
/*
    This is a multi-line comment ...
*/

rule CommentExample    // ... and this is single-line comment
{
    condition:
        false    // just an dummy rule, don't do this
}
```

3. Strings

YARA에는 3가지 타입의 문자열이 있습니다. 16진수 문자열, 텍스트 문자열 그리고 정규 표현식입니다. 16진수 문자열은 바이트의 시퀀스 그대로 정의하는데 사용하고, 텍스트 문자열과 정규 표현식은 텍스트문자 그대로 정의하여 읽는 부분에 사용합니다. 그러나 텍스트 문자열과 정규 표현식을 정의하는 규칙을 벗어나면 바이트 그대로의 시퀀스로 나타내어 아래에 표시됩니다.

3.1 Hexadecimal strings

16진수 문자열은 더욱 유연하게 wild-cards, jumps 그리고 alternatives의 세 가지 특별한 구조를 허용 합니다. Wild-cards는 어떤 것과도 매칭하지 않고, 어떤 바이트가 들어갈지도 모르는 자리를 표시하는 역할을 합니다. 자리를 표시하는 문자는 물음표(?) 입니다. 다음은 wild-cards와 함께 16진수 문자열을 사용한 예제입니다.

```
rule WildcardExample
{
    strings:
        $hex_string = { E2 34 ?? C8 A? FB }

    condition:
        $hex_string
}
```

위의 wild-cards 예제는 알 수 없는 값을 바이트 단위뿐만 아니라 nibble단위로 표현할 수 있는 의미를 가집니다.

Wild-cards는 문자열을 정의할 때 유용하지만 변수의 길이를 알고 있어야 합니다. 하지만 컨텐츠마다 그 길이는 다를 수 있습니다. 일부의 경우에서는 변수의 내용을 길이의 덩어리로 문자열 대신 정의해야 할 수 있습니다. 이러한 상황에서 wild-cards대신 jumps를 사용할 수 있습니다.

```
rule JumpExample
{
    strings:
        $hex_string = { F4 23 [4-6] 62 B4 }
```

```
    condition:
        $hex_string
}
```

위의 예제에서 대괄호로 묶여 있고 하이픈으로 구분 된 숫자 쌍이 있는데, 이는 jumps를 나타냅니다. 위 예제의 jumps는 4에서 6바이트의 임의의 시퀀스를 차지할 수 있다는 것을 나타내고 있습니다. 이는 다음 문자열의 패턴들과 일치합니다.

```
F4 23 01 02 03 04 62 B4
F4 23 00 00 00 00 00 62 B4
F4 23 15 82 A3 04 45 22 62 B4
```

가장 낮은 바운드 jumps는 (하이픈의 앞 숫자) 0보다 크거나 같아야 하고, 가장 높은 바운드 jumps는 (하이픈의 뒤의 숫자) 255이하여야 합니다. 다음은 유효한 jumps입니다.:

```
FE 39 45 [0-8] 89 00
FE 39 45 [23-45] 89 00
FE 39 45 [128-255] 89 00
```

다음은 유효하지 않은 jumps입니다:

```
FE 39 45 [10-7] 89 00
FE 39 45 [4-4] 89 00
FE 39 45 [200-300] 89 00
```

Jumps도 이와 같이 괄호로 묶어 하나의 번호로 지정 할 수 있습니다:

```
FE 39 45 [6] 89 00
```

위 jumps의 의미는 다음과 같이 정확히 6 바이트의 공간이 존재해야 한다는 것을 의미합니다.

```
FE 39 45 ?? ?? ?? ?? ?? ?? 89 00
```

물론 wild-cards와 jumps는 동일한 문자열에 혼합하여 사용 할 수 있습니다:

```
78 [4] 45 ?? 5F
```

Wild-cards와 jumps의 유일한 제한은 문자열의 시작 부분에 표시 할 수 없다는 것입니다. 다음 문자열은 올바르지 않습니다.

```
5? 00 40 23 [4] 12 35
```

16진수 문자열의 주어진 조각에 대한 다른 방법을 제공 해야만 하는 상황도 있습니다. 이러한 상황에서는 정규 표현식(regular expression)과 유사한 구문을 사용할 수 있습니다.

```
rule AlternativesExample1
{
    strings:
        $hex_string = { F4 23 ( 62 B4 | 56 ) 45 }

    condition:
        $hex_string
}
```

이 규칙은 F4 23 62 B4 45 또는 F4 23 56 45가 포함 된 파일과 일치합니다.

또한 두 개 이상의 대안도 표현 할 수 있습니다. 실제로 제공할 수 있는 대체 시퀀스의 양에는 제한이 없고 길이도 제한이 없습니다.

```
rule AlternativesExample2
{
    strings:
        $hex_string = { F4 23 ( 62 B4 | 56 | 45 ?? 67 ) 45 }

    condition:
        $hex_string
}
```

위의 예제에서도 볼 수 있듯이, wild-cards를 포함하는 문자열 대체 시퀀스의 일부로 사용할 수 있습니다. 그러나 jumps 포함하는 규칙은 만들 수 없습니다.

3.2 Text strings

이전 섹션과 같이 텍스트 문자열은 일반적으로 다음과 같이 정의되어 있습니다.

```
rule TextExample
{
    strings:
        $text_string = "foobar"

    condition:
        $text_string
}
```

ASCII-인코딩, 대소문자를 구분하는 문자열을 표현하는 가장 간단한 방법입니다. 그러나 텍스트 문자열은 문자열을 해석하는 방식에 유용한 수식을 동반 할 수 있습니다. 이러한 수식은 공백으로 구분 된 문자열 정의의 끝 부분에 추가되며, 아래와 같이 설명합니다.

텍스트 문자열은 C언어에서 사용 할 수 있는 escape sequences의 하위 집합을 포함 할 수 있습니다.

\" *더블 쿼터*
\\ *백슬래쉬*
\t *수평 탭*
\x*dd* *16진수 표기법의 모든 바이트*

3.2.1 Case-insensitive strings

YARA의 텍스트 문자열이 기본적으로 대소문자를 구분하지만 같은 줄의 문자열 정의 끝 부분에 보조 nocase를 추가하여 대소문자를 구분하지 않는 모드로 설정할 수 있습니다.

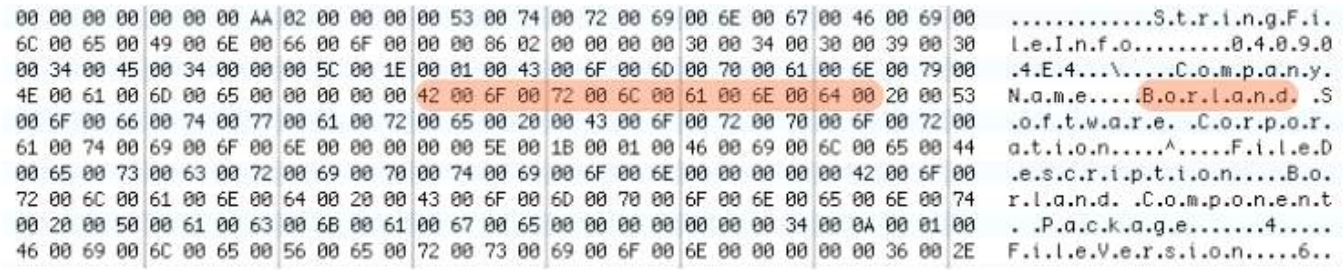
```
rule CaseInsensitveTextExample
{
    strings:
        $text_string = "foobar" nocase

    condition:
        $text_string
}
```

nocase 설정으로 문자열 "foobar"은 "Foobar", "FOOBAR"및 "fOoBaR"와 일치합니다.

이러한 설정은 다른 설정방식과 함께 사용할 수 있습니다.

3.2.2 Wide-character strings

Wide 설정은 실행에 필요한 많은 바이너리에서 문자 당 2바이트로 인코딩 된 문자열을 검색하는 데 사용할 수 있습니다.



위의 그림은 문자열 "Borland"에서 각 문자 당 1바이트로 인코딩되어 나타납니다. 그러므로 다음 규칙이 일치합니다.

```
rule WideCharTextExample
{
    strings:
        $wide_string = "Borland" wide

    condition:
        $wide_string
}
```

그러나 이 설정은 단지 0과 문자열에 있는 문자의 ASCII코드를 볼 수 있지만 영어가 아닌 문자가 포함 된 UTF-16 문자열은 지원하지 않습니다. 만약 wide와 ASCII 형태 둘 다 검색하길 원한다면, 순서 상관없이 wide와 함께 ASCII를 사용합니다.

```
rule WideCharTextExample
{
    strings:
        $wide_and_ascii_string = "Borland" wide ascii

    condition:
        $wide_and_ascii_string
}
```

Wide 설정 없이 ASCII 설정 같이 나타낼 수 있지만, wide에 사용되는 다양한 문자열은 기본적으로 ASCII로 간주되기 때문에 사용하지 않아도 됩니다.

3.2.3 Searching for full words

텍스트 문자열에 적용 할 수 있는 또 다른 설정에는 fullword가 있습니다. 이 설정은 숫자가 아닌 문자로 일치되는 문자열이 있는 경우에만 구분하여 나타내도록 약속하는 것입니다. 예를 들어 문자열 "domain"이 있고, fullword로 정의했다면 "www.mydomain.com"은 일치하지 않지만, "www.my-domain.com"이나 "www.domain.com"은 일치합니다.

3.3 Regular expressions

정규 표현식은 YARA의 가장 강력한 기능중 하나입니다. 이것은 텍스트 문자열과 같은 방식으로 정의하지만, 펄 프로그래밍 언어와 같이 더블쿼터 대신 백슬러시로 둘러쌉니다. 정규 표현식 문법 또한 펄과 호환됩니다.

```
rule RegExpExample1
{
    strings:
        $re1 = /md5: [0-9a-zA-Z]{32}/
        $re2 = /state: (on|off)/

    condition:
        $re1 and $re2
}
```

정규 표현식은 텍스트 문자열과 같이 뒤에 nocase, ascii, wide, 그리고 fullword와 같은 수식어가 붙을 수 있습니다. 이 수식어들의 의미는 이전에 텍스트 문자열에서 설명했던 것과 같습니다. 펄 정규 표현에 대한 좀 더 자세한 내용은 아래에 방문하시기 바랍니다.

http://www.pcre.org/pcre.txt
http://gypark.pe.kr/wiki/Perl/%EC%A0%95%EA%B7%9C%ED%91%9C%ED%98%84%EC%8B%9D

4. Conditions

Conditions는 모든 프로그래밍 언어에서 찾을 수 있는(예를 들어 "if" 구문에서 사용하는..) 부울 연산식에 불과합니다. 일반적인 부울 연산자 and, or 그리고 not 및 관계 연산자 >=, <=, <, >, == 그리고 !=를 포함할 수 있습니다. 또한 숫자 표현은 산술 연산자 (+,-,*, \ ) 및 비트 연산자 (&, |, <<, >>, ~)를 사용할 수 있습니다.

문자열 식별자 또한 Condition 내에서 부울 변수로 동작 할 수 있으며, 그 값은 파일 내부에 관련된 문자열의 존재 여부에 따라 달라집니다.

아래 예제를 보면, condition에 사용된 $a는 파일(또는 프로세스) 내에 "text1"이라는 문자열이 존재하면 True가 되고, 그렇지 않으면 False가 됩니다.

```
rule Example
{
    strings:
        $a = "text1"
        $b = "text2"
        $c = "text3"
        $d = "text4"

    condition:
        ($a or $b) and ($c or $d)
}
```

4.1 Counting strings

때때로 우리는 특정 문자열의 존재여부에 대해 알 필요가 있지만, 얼마나 많은 문자열이 파일 또는 프로세스 메모리에 나타날지는 모릅니다. 각 일치하는 문자열 항목의 수는 변수로 표현한 문자열 식별자 이름이지만 $ 문자 대신에 #문자로 사용합니다.
예제:

```
rule CountExample
{
    strings:
```

```
        $a = "dummy1"
        $b = "dummy2"

    condition:
        #a == 6 and #b > 10
}
```

이 규칙은 정확히 문자열 $a를 6번 포함하고 문자열 $b를 10번 보다 많이 포함하는 파일이나 프로세스를 검사합니다.

4.2 String offsets or virtual addresses

대부분의 경우에, 문자열 식별자를 조건에서 사용하는 경우는 관련 문자열이 어떤 파일이나 프로세스 메모리 내에 있어 그것을 알고자 하는 경우지만, 때때로 문자열이 파일이나 일부 가상 주소에 포함되어 있는 경우 프로세스 주소 공간의 특정 오프셋을 알 필요가 있습니다. 이러한 상황에서 연산자 at이 필요합니다. 이 연산자는 다음 예제와 같이 사용됩니다.

```
rule AtExample
{
    strings:
        $a = "dummy1"
        $b = "dummy2"

    condition:
        $a at 100 and $b at 200
}
```

위 예에서 $a at 100 표현은 만약 문자열 $a가 파일 안의 오프셋 100에서(또는 실행중인 프로세스의 가상 주소 100에서) 발견되는 경우에만 해당합니다. 문자열 $b는 오프셋 200에서 나타납니다. 두 오프셋이 10진수임을 참고합니다. 16진수 숫자는 C언어에서처럼 숫자 앞에 접두사로 0x를 추가하여 사용하며, 이는 가상 주소를 작성할 때 매우 유용하게 사용할 수 있습니다. 추가로 참고할 것은 at이 and보다 연산자 우선순위가 높습니다.

at 명령어는 프로세스 메모리 영역을 포함하는 가상 주소 또는 파일의 고정된 오프셋에서 문자열을 탐색할 수 있고, in 명령은 오프셋 또는 주소의 범위 안에서 문자열을 탐색하는데 사용할 수 있습니다.

```
rule InExample
```

```
{
    strings:
        $a = "dummy1"
        $b = "dummy2"

    condition:
        $a in (0..100) and $b in (100..filesize)
}
```

위의 예제에서는 문자열 $a를 오프셋 0과 100사이에서 찾을 것이고, 문자열 $b는 오프셋 100과 파일의 끝 사이에서 찾을 수 있습니다. 다시 말하지만, 숫자는 기본적으로 10진수입니다.

또한 오프셋 또는 가상 주소의 i번째에 발생하는 문자열 $a은 @a[i]로 사용하여 가져올 수 있습니다. 이는 첫 번째 나타내는 것은 @a[1] 두 번째는 @a[2] 등등 색인기반입니다. 만약 문자열이 나타낸 수보다 더 큰 색인 값을 정의하는 경우, 그 결과 값은 NaN (Not A Number) 값이 될 것입니다.

4.3 File size

문자열 식별자는 변수상태 뿐만 아니라 사용할 수 있는 다른 특별한 변수들이 있습니다. (실제로 밑에서 볼 수 있는 것처럼 어떤 문자열을 정의하지 않고 규칙을 정의 할 수 있습니다.) 이러한 특별한 변수 중 하나로 filesize가 있는데 이는 이름 그대로 파일크기로 분석합니다. 크기는 바이트로 표시됩니다.

```
rule FileSizeExample
{
    condition:
        filesize > 200KB
}
```

앞의 예는 KB 접미사의 사용을 보여줍니다. 이 접미사는 숫자 상수에 이어 사용하면 자동으로 1024의 상수 값을 곱합니다. MB 접미사를 사용하면 2^{20}값을 곱하게 됩니다. 두 접미사는 십진수 상수에서만 사용 할 수 있습니다.

filesize를 사용한 규칙을 파일에 적용할 수 있지만, 만일 실행중인 프로세스에 적용할 경우 filesize는 이해할 수 없는 맥락이 되기 때문에 절대 검사하지 않습니다.

4.4 Executable entry point

규칙에서 사용 할 수 있는 다른 특별한 변수는 entrypoint입니다. 만약 파일이 Portable Executable (PE) 또는 Executable and Linkable Format (ELF), 실행 가능한 진입점의 오프셋을 가지고 있는 경우 이 변수로 인해 파일을 스캐닝 할 수 있습니다. 만약 실행중인 프로세스의 entrypoint를 스캐닝 할 경우 메인 가상 주소의 실행 가능한 진입점을 가져옵니다. 이 변수의 일반적인 사용은 간단한 파일 감염 또는 packers의 진입점을 감지 할 수 있는 일부 패턴을 찾는 것입니다.

```
rule EntryPointExample1
{
    strings:
        $a = { E8 00 00 00 00 }

    condition:
        $a at entrypoint
}

rule EntryPointExample2
{
    strings:
        $a = { 9C 50 66 A1 ?? ?? ?? 00 66 A9 ?? ?? 58 0F 85 }

    condition:
        $a in (entrypoint..entrypoint + 10)
}
```

규칙에 entrypoint 변수의 존재는 오직 PE 또는 ELF 파일만이 규칙을 충족할 수 있다는 것을 의미합니다. 만약 PE 또는 ELF가 아닌 파일에 이 규칙을 사용한다면 이 변수는 false로 평가됩니다.

4.5 Accessing data at a given position

파일 또는 실행중인 프로세스를 검사하는 경우에 따라 특정 파일의 오프셋이나 메모리의 가상 주소에 저장된 데이터의 조건을 쓸 수 있는 경우가 있습니다. 이러한 상황에서는 파일로

부터 읽은 오프셋을 다음 기능들 중 하나를 사용할 수 있습니다.

int8(*<offset or virtual address>*)
int16(*<offset or virtual address>*)
int32(*<offset or virtual address>*)

uint8(*<offset or virtual address>*)
uint16(*<offset or virtual address>*)
uint32(*<offset or virtual address>*)

intXX 기능은 *<offset or virtual address>*에 쓰인 8, 16, 32 비트의 부호가 있는 정수를 읽는 것이고, uintXX 기능은 부호가 없는 정수를 읽습니다. 16 과 32비트의 정수는 little-endian으로 간주됩니다. 기본적으로 *<offset or virtual address>* 파라미터는 uintXX 기능의 리턴 값을 포함하여 부호가 없는 정수를 반환하는 표현이 될 수 있습니다. 다음 예제에서는 PE 파일을 구별 할 수 있는 규칙을 볼 수 있습니다.

```
rule IsPE
{
    condition:
        // MZ signature at offset 0 and ...
        uint16(0) == 0x5A4D and
        // ... PE signature at offset stored in MZ header at 0x3C
        uint32(uint32(0x3C)) == 0x00004550
}
```

4.6 Sets of strings

파일은 지정된 설정에서 숫자를 사용하여 특정 문자열을 포함해야 한다는 것을 표현 할 필요가 있습니다. 설정하는 문자열은 모두 존재할 필요는 없지만, 적어도 일부는 있어야 한다고 가정하면, 이러한 상황에서 연산자 of가 필요합니다.

```
rule OfExample1
{
    strings:
        $a = "dummy1"
        $b = "dummy2"
```

```
        $c = "dummy3"

    condition:
        2 of ($a,$b,$c)
}
```

위 규칙은 설정한 ($a,$b,$c)의 문자열에서 적어도 두 문자열이 어떤 문제도 없이 파일에 존재해야 합니다. 물론, 이 명령을 사용하려면 of 키워드 앞의 숫자가 설정한 문자열의 개수보다 작거나 같아야 합니다.

설정의 요소는 명시적으로 앞의 예와 같이 열거하여 사용하거나 wild-cards를 사용할 수 있습니다. 예를 들면 다음과 같습니다.

```
rule OfExample2
{
    strings:
        $foo1 = "foo1"
        $foo2 = "foo2"
        $foo3 = "foo3"

    condition:

        /* ($foo*) is equivalent to ($foo1,$foo2,$foo3) */

        2 of ($foo*)
}

rule OfExample3
{
    strings:
        $foo1 = "foo1"
        $foo2 = "foo2"

        $bar1 = "bar1"
        $bar2 = "bar2"

    condition:
        3 of ($foo*,$bar1,$bar2)
}
```

($*)대신에 them를 사용하여 규칙에 있는 모든 문자열을 참조하거나 더 가독성을 높일 수 있습니다.

```
rule OfExample4
{
    strings:
        $a = "dummy1"
        $b = "dummy2"
        $c = "dummy3"

    condition:
        1 of them /* equivalent to 1 of ($*) */
}
```

위의 모든 예제에서는 문자열의 번호는 숫자 상수로 지정되었지만, 숫자 값을 반환하는 식으로 사용할 수도 있습니다. 키워드 any 와 all 가 함께 사용할 수 있습니다.

```
all of them          /* all strings in the rule */
any of them          /* any string in the rule */
all of ($a*)         /* all strings whose identifier starts by $a */
any of ($a,$b,$c)    /* any of $a, $b or $c */
1 of ($*)            /* same that "any of them" */
```

4.7 Applying the same condition to many strings

of 명령어와 매우 유사하지만 좀 더 강력한 for..of 명령어가 있습니다. 구문은 다음과 같습니다.

for *expression* of *string_set* : (*boolean_expression*)

그 의미는 다음과 같습니다. *string_set*의 문자열로부터 적어도 *expression*는 *boolean_expression*을 만족해야합니다.

즉 *boolean_expression*는 *string_set*의 모든 문자열을 평가하여 적어도 반환 값이 true가 되는 *expression*이어야 합니다.

물론, *boolean_expression*의 중요한 세부 사항을 제외하고, 규칙의 조건 섹션에서 부울 식을 표현하여 사용할 수 있습니다. 여기에는 평가되는 문자열에 대한 달러 기호 ($)를 사용 할 수 있습니다. 다음 식을 살펴보십시오.

```
for any of ($a,$b,$c) : ( $ at entrypoint )
```

부울 식의 $ 기호는 특정 문자열에 연결하지 않고, $a다음 $b다음 $c의 연속하는 세 가지 표현을 평가합니다.

이미 알겠지만 of 명령어는 for..of 의 특별한 형태입니다. 다음 표현식은 동일합니다.

```
any of ($a,$b,$c)
for any of ($a,$b,$c) : ( $ )
```

또한 #과 @를 사용한 사건의 횟수에 대한 참조를 만들어 각각의 첫 번째 문자열의 오프셋을 구합니다.

```
for all of them : ( # > 3 )
for all of ($a*) : ( @ > @b )
```

4.8 Using anonymous strings with "of" and "for..of"

of 와 for..of 연산자 다음에 them을 사용할 경우, 각 문자열에 할당된 식별자는 일반적으로 불필요한 것입니다. 개별적으로 모든 문자열을 참조하지 않음으로써 각각의 고유한 식별자를 제공할 필요가 없습니다. 이러한 상황에서는 다음 예에서와 같이 $ 문자로만 구성된 익명의 식별자를 이용하여 문자열을 선언할 수 있습니다.

```
rule AnonymousStrings
{
    strings:
        $ = "dummy1"
        $ = "dummy2"

    condition:
        1 of them
}
```

4.9 Iterating over string occurrences

4.2 절에서와 같이 특정 문자열이 있는 파일 또는 프로세스 주소 공간 내에서 나타나는 오프셋 또는 가상 주소는 구문을 사용하여 액세스 할 수 있습니다. @a[i]에서 i 가 나타내는 지수는 $a 에서 발생하는 문자열로 말할 수 있습니다. (@a[1], @a[2],...).

때때로 이러한 반복구문과 오프셋의 일부를 이용하여 주어진 조건을 만족시켜야 합니다. 예를 들면 다음과 같습니다.

```
rule Ocurrences
{
    strings:
        $a = "dummy1"
        $b = "dummy2"

    condition:
        for all i in (1,2,3) : (@a[i] + 10 == @b[i])
}
```

위 규칙을 말하자면 $b의 첫 세 가지 사건은 $a의 첫 세 가지 사건에서 10바이트 차이로 있어야 알려줍니다.

다음은 위와 같은 형태로 사용한 것입니다.

```
for all i in (1..3) : (@a[i] + 10 == @b[i])
```

인덱스 값 범위를 (1,2,3)대신 (1..3)으로 사용한 것을 확인 할 수 있습니다. 물론, 범위를 지정하는데 있어 상수를 사용하도록 강요하지만, 다음 예에서 보는 것과 같이 표현식을 사용할 수 있습니다.

```
for all i in (1..#a) : (@a[i] < 100)
```

이 경우 $a의 모든 사건을 통한 반복을 하고 있습니다. (#a는 $a의 일치하는 발생의 수를 나타낸 다는 것을 기억하면 됩니다.) 이 규칙은 $a의 모든 사건이 파일의 시작에서부터 100 바이트 내에 존재한다는 것을 말하고 있습니다.

문자열의 일부 발생 조건이 만족해야하는 것을 표현하려는 경우, 동일한 논리로 for..of 연산

자를 사용하여 적용할 수 있습니다.

```
for any i in (1..#a): ( @a[i] < 100 )
for 2 i in (1..#a): ( @a[i] < 100 )
```

이 연산자는 반복 구문입니다.

for *expression identifier* in *indexes* : (*boolean_expression*)

4.10 Referencing other rules

규칙에 대한 조건을 작성할 때 기존 프로그래밍 언어의 함수 호출과 유사한 방식으로 이전에 정의 된 규칙에 대한 참조를 만들 수 있습니다. 이 방법을 통해 다른 함수를 참조하는 규칙을 만들 수 있습니다. 다음 예에서 볼 수 있습니다.

```
rule Rule1
{
    strings:
        $a = "dummy1"

    condition:
        $a
}

rule Rule2
{
    strings:
        $a = "dummy2"

    condition:
        $a and Rule1
}
```

위의 예에서 볼 수 있듯이, 오직 파일의 문자열 "dummy2"와 Rule1을 만족하는 경우에만 Rule2를 만족 할 수 있습니다. 참고로 호출을 하기 전에 호출되는 규칙을 정의하는 것을 엄격하게 관리 할 필요성이 있습니다.

5. More about rules

이것들은 지금까지 적용하지 않은 YARA 규칙의 일부 측면들이지만, 중요합니다. 이는 global rules, private rules, tags 그리고 metadata입니다.

5.1 Global rules

Global rules는 한 번에 모든 규칙에 제한을 부여할 수 있습니다. 예를 들어, 특정 크기에 제한을 두어 초과하면 해당 파일을 무시한다고 가정합니다. 그러면 규칙을 통해 자신의 조건에 필요한 규칙으로 수정하거나, 다음과 같이 global rules로 작성 할 수 있습니다.

```
global rule SizeLimit
{
    condition:
        filesize < 2MB
}
```

많은 global rules를 정의하려면, 다른 규칙을 적용하기 전에 원하는 만큼 사용하여 global rules를 적용 할 수 있습니다.

5.2 Private rules

Private rules는 매우 단순한 개념입니다. 이것은 단순히 규칙이 특정 파일과 일치 할 때 YARA에 이해 보고되지 않는 규칙입니다. 전혀 보고되지 않는 규칙은 처음에는 아무것도 아닌 것처럼 보일 수 있으나, YARA에서 제공하는 다른 규칙과 혼합하여 (4.5 섹션을 참조) 매우 유용하게 사용 될 수 있습니다. Private rules는 다른 규칙에 대한 블록을 구축하는 역할을 하고, 동시에 관련이 없는 정보를 YARA를 통해 복잡하지 않게 출력하도록 할 수 있습니다. Private 규칙은 private키워드를 추가하여 선언할 수 있습니다.

```
private rule PrivateRuleExample
{
    ...
}
```

private와 global 둘 다 사용하여 만든 규칙은 YARA에 이해 보고되지는 않지만 글로벌 규칙을 모두 만족 해야만 합니다.

5.3 Rule tags

YARA의 또 다른 유용한 기능으로는 규칙에 태그를 추가 할 수 있습니다. 이러한 태그는 나중에 YARA의 출력을 필터링하고 오직 흥미로운 룰만을 보여주는데 사용 할 수 있습니다. 룰에 많은 태그들을 추가하길 원하는 경우, 아래에서 보여주는 것과 같이 규칙 식별자 이후에 선언하여 사용 할 수 있습니다.

```
rule TagsExample1 : Foo Bar Baz
{
    ...
}

rule TagsExample2 : Bar
{
    ...
}
```

태그는 규칙 식별자의 동일한 어휘 규칙을 준수해야하기 때문에 영문자와 숫자 및 밑줄을 사용할 수 있으며, 숫자로 시작 할 수 없습니다. 또한 대소문자를 구분합니다.

YARA를 사용 할 때 사용자가 설정한 태그나 태그 된 태그를 사용한 규칙을 출력 할 수 있습니다.

5.4 Metadata

규칙은 문자열 정의 및 조건 섹션 외에도, metadata 섹션을 이용하여 사용자의 규칙에 대한 추가 정보를 넣을 수 있습니다. metadata 섹션은 다음 예외 같이 meta 키워드와 식별자/값을 쌍으로 포함하여 정의 할 수 있습니다.

```
rule MetadataExample
{
    meta:
        my_identifier_1 = "Some string data"
```

```
        my_identifier_2 = 24
        my_identifier_3 = true

    strings:
        $my_text_string = "text here"
        $my_hex_string = { E2 34 A1 C8 23 FB }

    condition:
        $my_text_string or $my_hex_string
}
```

예제에서 볼 수 있듯이 metadata 식별자는 등호 뒤에 항상 다음과 같은 값이 할당됩니다. 할당된 값은 문자열, 정수 또는 부울 값 중에 하나인 true 또는 false가 될 수 있습니다. 참고할 것은 metadata 섹션에 정의 된 식별자/값 쌍은 상태 섹션에서는 사용 할 수 없고, 규칙에 대한 추가 정보를 저장하는 목적으로만 사용 할 수 있습니다.

5.5 External variables

External variables는 외부에서 제공되는 값에 따라 규칙을 정의 할 수 있습니다. 예를 들어 다음과 같은 규칙으로 작성할 수 있습니다.

```
rule ExternalVariableExample1
{
    condition:
        ext_var == 10
}
```

이 경우에는 ext_var의 값은 실행시간에 할당 된 외부 변수입니다. (command-line 툴의 옵션 –d와 yara-python의 메소드인 match와 compile의 externals 파라매터에서 볼 수 있습니다.) 외부 변수의 유형은 할당 된 값에 따라 integer, string 또는 부울이 될 수 있습니다. 정수 변수는 상태와 논리 변수, 부울 표현의 자리를 차지할 수를 상수로 대체 할 수 있습니다. 다음은 이 예제입니다.

```
rule ExternalVariableExample2
{
    condition:
        bool_ext_var or filesize < int_ext_var
```

```
}
```

형식 문자열의 외부 변수는 연산자 contains와 matches와 함께 사용할 수 있습니다. 문자열은 지정한 하위 문자열을 포함하는 경우 contains 연산자는 true를 반환합니다. 만약 문자열이 주어진 정규 표현식과 일치하는 경우 matches 연산자는 true를 반환합니다.

```
rule ExternalVariableExample3
{
    condition:
        string_ext_var contains "text"
}

rule ExternalVariableExample4
{
    condition:
        string_ext_var matches /[a-z]+/
}
```

규칙에 사용되는 모든 외부 변수는 command-line 도구의 –d 옵션을 사용하여 실행 시간에 정의되거나, 또는 yara-python에서 적절한 방법으로 외부 파라미터를 제공하여 사용합니다.

6. Includes

좀 더 유연한 규칙을 만들 수 있도록 하기 위해, YARA에서는 include 지시어를 제공합니다. 이 지시자는 C 프로그램에서 *#include* 사전지시자와 비슷하게 컴파일 하는 동안 현재 파일에서 지정한 소스 파일의 내용을 삽입하는 방식으로 작동합니다. 다음 예제는 현재 파일이 "other.yar"를 포함합니다.

```
include "other.yar"
```

include 지시문에서 파일을 검색하는 기본 경로는 현재 파일이 상주하는 디렉터리 입니다. 이러한 이유에서, 앞의 예제의 "other.yar"파일은 현재 파일과 같은 디렉터리에 위치해야 합니다. 하지만 상대 경로를 지정하여 이러한 문제점을 해결할 수 있습니다.

```
include "./includes/other.yar"
include "../includes/other.yar"
```

또한 절대 경로를 사용할 수 있습니다.

```
include "/home/plusvic/yara/includes/other.yar"
```

Windows에서는 드라이브 문자를 작성하는 것을 잊으면 안되며, 슬래시 및 백 슬래시 모두 사용할 수 있습니다.

```
include "c:/yara/includes/other.yar"
include "c:₩₩yara₩₩includes₩₩other.yar"
```

7. Using YARA from command-line

YARA를 호출하기 위해서는 다음 두 가지가 필요합니다: 적용 할 규칙 집합, 파일에 대한 정보와 경로, 폴더 또는 스캔 할 프로세스의 PID. 규칙은 규칙을 포함하는 하나 이상의 일반 텍스트 파일을 통해 YARA에게 제공하거나 또는 표준 입력을 통해 어떤 규칙 파일을 지정하지 않고 제공 할 수 있습니다.

```
usage: yara [OPTION]... [RULEFILE]... FILE | PID
options:
-t  <tag>                 print rules tagged as <tag> and ignore the rest.
-i  <identifier>          print rules named <identifier> and ignore the rest.
-n                        print only not satisfied rules (negate).
-g                        print tags.
-m                        print metadata.
-s                        print matching strings.
-l  <number>              abort scanning after a <number> of rules matched.
-d  <identifier>=<value>  define external variable.
-r                        recursively search directories.
-f                        fast matching mode.
-v                        show version information.
```

규칙은 YARA의 마지막 인자로 지정된 객체에 적용되는데, 이것은 현재의 디렉터리의 경로에 있다면, 그 안에 포함 된 모든 파일을 검사 할 수 있습니다. 기본적으로 YARA는 재귀적으로 디렉터리를 검색하도록 시도하지 않지만 –r 옵션을 사용하여 이를 해결 할 수 있습니다.

-t 옵션은 YARA의 출력 필터 역할을 하지만 하나 이상의 태그를 지정 할 수 없습니다. 이 옵션을 사용하는 경우, 지정된 태그만 규칙에 표시됩니다. -i 옵션은 비슷한 동작을 하는데, 주어진 식별자를 갖는 하나를 제외한 모든 규칙을 필터링 합니다. 또한 –n을 이용하여 만족스럽지 못한 규칙을 출력하는데 변형하여 사용할 수 있습니다.

-l 옵션은 주어진 번호와 일치 하면 규칙을 중지 할 수 있습니다.

-d 옵션은 외부 변수를 정의하는데 사용합니다. 다음과 같습니다.

```
-d flag=true
-d beast=666
-d name="James Bond"
```

8. Using YARA from Python

YARA는 파이썬 스크립트에서 호출 할 수 있습니다. **yara-python** 확장은 파이썬 사용자에게 YARA 기능을 사용 할 수 있도록 하기 위해 제공됩니다. 한번 **yara-python**을 설치하고 나면 아래와 같이 사용 할 수 있습니다.

```
import yara
```

import yara를 한 다음 데이터를 적용하기 전에 YARA 규칙을 컴파일 하고, 컴파일은 규칙의 파일 경로를 설정하여 할 수 있습니다.

```
rules = yara.compile(filepath='/foo/bar/myrules')
```

기본 argument는 *filepath* 이므로, 따로 이름을 지정 할 필요가 없습니다.

```
rules = yara.compile('/foo/bar/myrules')
```

또한 규칙의 파일 개체화를 통해 컴파일 할 수 있습니다.

```
fh = open('/foo/bar/myrules')
rules = yara.compile(file=fh)
fh.close()
```

또는 파이썬 문자열에서 직접 컴파일 할 수 있습니다.

```
rules = yara.compile(source='rule dummy { condition: true }')
```

동일한 시간에 파일이나 문자열의 그룹을 컴파일하려 한다면 *filepaths* 또는 *sources* 인수 이름을 사용 할 수 있습니다.

```
rules = yara.compile(filepaths={
    'namespace1':'/my/path/rules1',
    'namespace2':'/my/path/rules2'
```

```
})

rules = yara.compile(sources={
    'namespace1':'rule dummy { condition: true }',
    'namespace2':'rule dummy { condition: false }'
})
```

참고로 *filepaths* 와 *sources*는 문자열 유형의 키가 존재하는 dictionary형태가 있는 것을 확인 할 수 있습니다. dictionary키는 네임스페이스의 식별자로 사용됩니다. 이는 두 번째 예에서 "dummy"라는 이름으로 발생하는 것과 같이, 다른 소스에서 같은 이름으로 사용되는 규칙을 구별할 수 있도록 해줍니다.

또한 컴파일 방법은 *includes*라는 이름의 선택적 부울 파라미터를 갖고 있습니다. 그 변수는 소스 파일에 접근하는 지시자 include directive를 제어 할 수 있는지 없는지를 선택해주는 파라미터입니다. 해당 예는 다음과 같습니다.

```
rules = yara.compile('/foo/bar/myrules', includes=False)
```

만약 소스 파일에 include 지시어가 포함되어 있는 경우 이전 라인은 예외를 일으킬 것입니다.

만약 개인적인 규칙속의 외부 변수를 사용한다면 해당 외부변수를 컴파일 하거나 혹은 다른 파일에 그 규칙을 적용하는 동안, 해당 외부 규칙을 정의해야 할 것입니다. 그리고 컴파일 하는 순간에 변수를 정의하기 위해서는 컴파일 메서드에 외부 파라미터를 보내야 합니다. 이것은 다음 예와 같습니다.

```
rules = yara.compile( '/foo/rules',
                      externals= {
                          'var1': 'some string',
                          'var2': 4,
                          'var3': True
                      })
```

외부 파라미터는 키와 같은 변수 이름의 사전이어야 하고, 정수 또는 부울 유형의 두 문자열의 연관 값이어야 합니다.

컴파일의 모든 경우는 결과적으로 일치하는 방식을 가지고 있는 클래스 Rules의 인스턴스를 반환합니다.

```
matches = rules.match('/foo/bar/myfile')
```

그러나 또한 파이썬 문자열로 규칙을 적용 할 수 있습니다.

```
f = fopen('/foo/bar/myfile', 'rb')

matches = rules.match(data=f.read())
```

아니면 프로세스를 실행합니다.

```
matches = rules.match(pid=1234)
```

컴파일의 경우와 마찬가지로, 탐색 방법은 외부 파라미터 안에 외부 변수에 대한 정의를 받을 수 있습니다.

```
matches = rules.match( '/foo/bar/myfile',
                       externals= {
                            'var1': 'some other string',
                            'var4': 100,
                       })
```

컴파일 하는 시간동안 정의된 외부변수는 match함수에서 재차 정의될 필요는 없습니다. 그러나 필요에 따라 어떤 변수를 재정의 하거나, 컴파일하는 동안 제공되지 않은 정의를 추가 제공 할 수 있습니다.

일치하는 방법을 호출 할 때 콜백 함수를 지정할 수 있습니다. 제공되는 기능은 매칭 여부와 상관없이 모든 규칙에 대한 호출이 이루어집니다. 콜백 기능은 하나의 사전 타입 파라미터를 결정하며, 차기 규칙이나 혹은 데이터에 규칙을 적용하는 것을 중지하는 CALLBACK_ABORT를 수행하여 CALLBACK_CONTINUE을 리턴합니다.

여기 예제를 보십시오.

```
import yara

def mycallback(data):
    print data
    yara.CALLBACK_CONTINUE

matches = rules.match('/foo/bar/myfile', callback=mycallback)
```

전달 된 dictionary형태는 다음과 같이 될 것입니다.

```
{
    'tags': ['foo', 'bar'],
    'matches': True,
    'namespace': 'default',
    'rule': 'my_rule',
    'meta': {},
    'strings': [(81, '$a', 'abc'), (141, '$b', 'def')]
}
```

규칙 데이터 여부와 일치하는 경우 일치하는 필드를 나타냅니다.

문자열 필드 형식의 벡터로, 검색 문자열의 목록입니다.

(<offset>, <string identifier>, <string data>)

탐색 방법은 Match 클래스의 인스턴스의 목록을 반환합니다. 이 클래스의 인스턴스는 일치하는 규칙의 이름을 포함하여 텍스트 문자열로 간주될 수 있습니다. 다음 예는 이 목록을 프린트합니다.

```
for m in matches:
    print "%s" % m
```

일부 환경에서는 명시 적으로 Match 문자열의 인스턴스를 변환해야 할 수 있습니다. 다른 문자열과 비교했을 때의 예를 들면 다음과 같습니다.

```
if str(matches[0]) == 'SomeRuleName':
    ...
```

Match 클래스는 콜백 함수에 전달된 dictionary와 같은 속성이 있습니다.

- rule
- namespace
- meta
- tags
- strings

9. PerformanceGuidelines

YARA의 규칙을 만들 때 최상의 성능을 위해 다음 가이드라인을 유지하길 바랍니다.

첫 번째 두 바이트의 문자열에는 wild-cards 또는 jumps의 사용을 피해야 합니다. YARA는 가장 효과적으로 수행하기 위해서는 빠른 속도로 파일을 검색해야 하는데, 이러한 목적을 위해서 해시 테이블에 선언 된 문자열을 저장합니다. 이 해시 테이블에 저장되는 해시 키는 첫 번째 두 바이트의 문자열입니다. 엄격하게 정의 된 첫 두 바이트가 없는 문자열은 해시 테이블에 저장 될 수 없어 검색 과정에서 속도가 느려집니다.

```
$a = { EB ?? 00 34 12 0F }  /* bad, wild-card on second byte */
$a = { EB 00 ?? 34 12 0F }  /* ok */

$a = /.*ab/   /* bad, regular expression starting with wild-card */
$a = /ab.*/   /* ok, regular expression have the first two chars well defined */
```

jumps를 사용 할 때 가장 짧은 길이의 범위를 유지하여 사용합니다. YARA가 0-255의 길이로 jumps를 한다는 것은 좋지 않고, 더 나은 jumps를 위해 범위를 짧게 유지하는 것이 좋습니다.

```
$a = { FF 25 00 [0-200] FF 25 }  /* bad, range from 0 to 200 too wide */
$a = { FF 25 00 [0-8] FF 25 }        /* ok */
$a = { FF 25 00 [200-210] FF 25 }  /* ok */
```

하지만 너무 짧은 문자열을 정의하지는 마십시오. 5~6 바이트들 보다 짧은 문자열은 너무 많은 파일들을 탐색 할 것입니다.

또한 필요한 경우에만 정규표현식을 사용합니다. 정규표현식을 이용한 평가는 본질적으로 느리지만, 사용해야 한다면 hex 문자열과 함께 jumps와 wild-cards를 사용하지 않는 것으로 이 문제를 완화 시킬 수 있습니다.

10. YARA Editor!

YARA는 악성 코드 샘플을 식별하고 분류하여 악성코드 연구자들을 돕는 목적으로 만들어진 도구입니다. 그리고 Yara-editor는 이러한 YARA툴에 사용되는 규칙을 쉽게 빌드하도록 도와줍니다. 즉, 규칙을 만들고 바이너리에 적용하기 위한 편집기입니다.

10.1 YARA-Editor-1.0.5 Install on Linux

```
~# apt-get install python-sip
~# apt-get install python-gt4
~# wget http://yara-editor.googlecode.com/files/yara-editor-0.1.5.tar.gz
~# tar xfz yara-editor-0.1.5.tar.gz
~# cd yara-editor-0.1.5
~yara-editor-0.1.5# python setup.py install
```

python-sip : C와 C++의 라이브러리와 인터페이스를 파이썬에서 작성하기 위해 사용하는 모듈입니다. 아래의 python-qt4를 사용하기 위해 설치합니다.

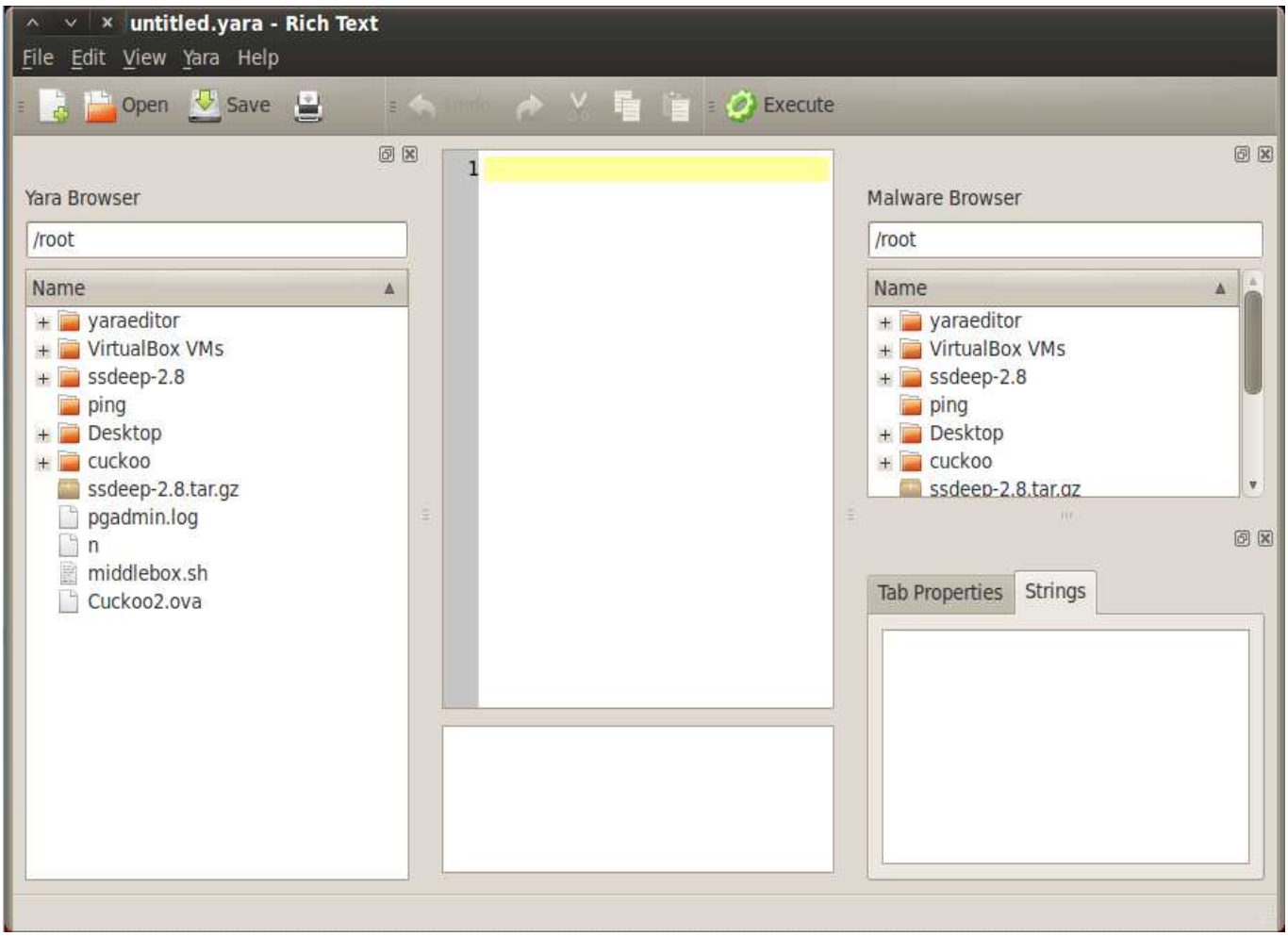
python-qt4 : qt4는 데스크탑, 임베디드 및 모바일 플랫폼에 대한 응용 프로그램 및 사용자 인터페이스의 생성을 간소화하여 사용 할 수 있도록 만들어진 프레임워크입니다. 이는 C++, CSS, Javascript와 같은 API를 제공합니다. python-qt4는 이러한 qt4를 python에서 사용 할 수 있도록 합니다.

실행하면 다음과 같은 에러가 발생합니다.

```
~yara-editor-0.1.5# yara-editor
Traceback (most recent call last):
      File "/usr/local/bin/yara-editor", line 112, in <module>
          main(sys.argv)
      File "/usr/local/bin/yara-editor", line 109, in main
          mapp = app(args)
      File "/usr/local/bin/yara-editor", line 101, in __init__
          config=config)
      File "/usr/local/lib/python2.6/dist-packages/yaraeditor/core/controlleur.py", line 78,
in __init__
          self.path_yara=config.get(CONF_PREFERENCE, CONF_PATH_YARA)
      File "/usr/lib/python2.6/ConfigParser.py", line 531, in get
          raise NoSectionError(section)
 ConfigParser.NoSectionError: No section: 'preference'
~yara-editor-0.1.5#
```

하지만 다시 실행하게 되면 정상적으로 동작하게 되지만, 프로그램을 종료하면 다음과 같은 메시지를 보여줍니다.

```
~yara-editor-0.1.5# yara-editor
QWidget::setWindowModified: The window title does not contain a '[*]' placeholder
~yara-editor-0.1.5#
```



10.2 UI – Yara Browser!

왼쪽 측면은 "YARA Browser"입니다. 이 두 부분은 두 가지 요소들로 구성되어 있습니다.

- 경로
- 경로로 지정된 디렉토리의 내용

파일의 경우 사용자가 더블클릭을 할 때, 파일이 중앙 부분에서 열립니다.

10.3 UI – Malware Browser!

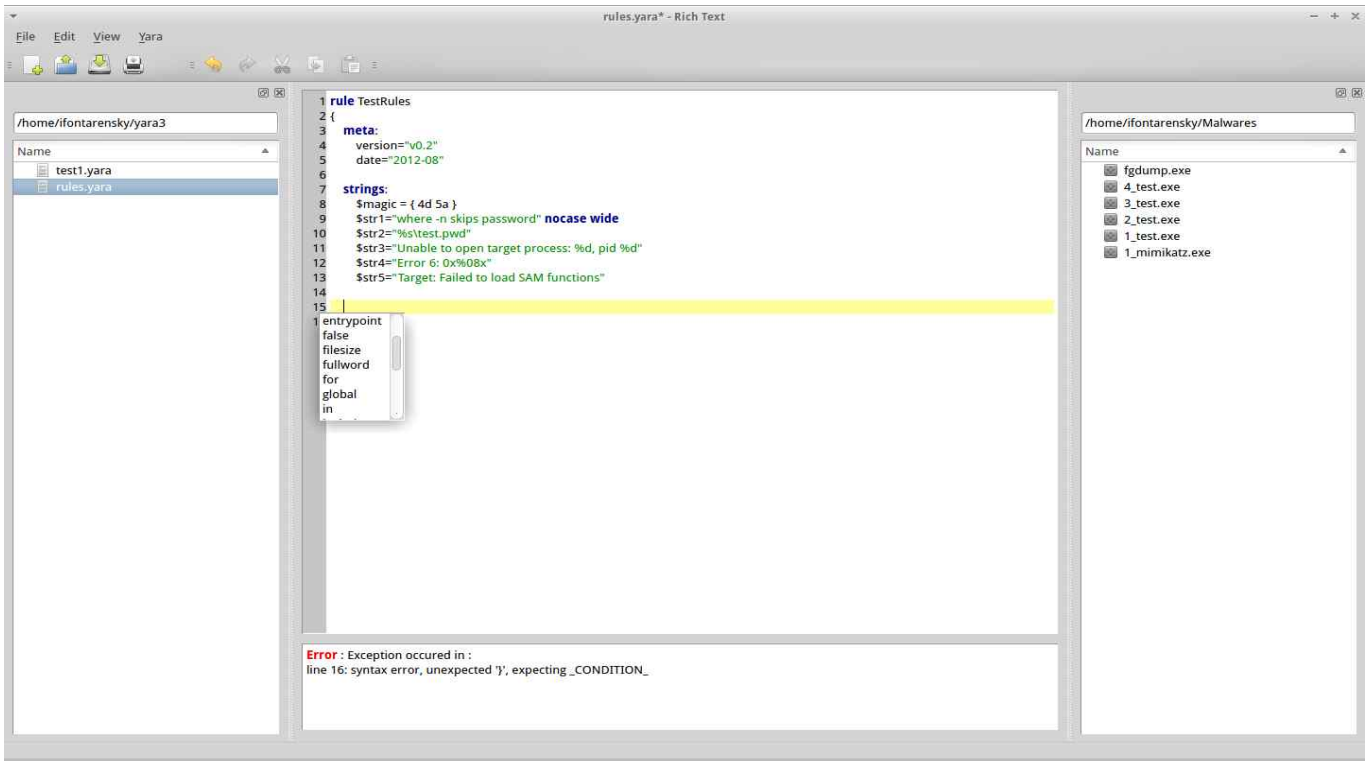
오른쪽 측면은 "Malware Browser"입니다. 이 두 부분은 두 가지 요소들로 구성되어 있습니다.

- 경로
- 경로로 지정된 디렉토리의 내용

여기에 폴더와 파일을 선택할 수 있습니다. 선택한 파일은 강조되어 표시됩니다. 사용자가 규칙을 테스트하려 할 경우, 각각의 악성코드를 선택하거나 각각의 폴더를 선택하면 됩니다.(폴더는 재귀적으로 검사합니다.)

10.4 Editor

중앙 부분은 규칙을 작성 할 수 있습니다. 각각의 키워드는 강조되어 표시됩니다.



10.5 Config File

구성 파일이 없는 경우, 응용프로그램이 파일을 만듭니다.

~/.yara/yara-editor/conf

이 파일에는 많은 요소가 들어있지 않습니다.(임시)

[preference]
lang = /home/user/.yara/yara-editor/i18n/i18n_en.qm
path_yara_rules = /home/user/yara
path_malwares = /home/user/

lang : 언어는 다음 버전에 대한 것입니다.
path_yara_rules : 응용 프로그램 탐색을 위한 YARA 규칙의 저장소의 경로
path_malware : 악성코드 탐색을 위한 편집기의 경로

10.6 Key Bindings

이 에디터에서 모든 키 바인딩은 구성하지 않습니다.

10.7 File Menu

Ctrl+N	새 문서
Ctrl+O	YARA 룰 열기
Ctrl+S	저장
Ctrl+Shift+S	다른 이름으로 저장
Ctrl+P	출력
Ctrl+Shift+P	미리보기
Ctrl+D	PDF로 내보내기
Ctrl+Q	종료

10.8 Edit Menu

Ctrl+Z	실행 취소
Ctrl+Shift+Z	다시 실행
Ctrl+X	잘라내기
Ctrl+C	복사하기
Ctrl+V	붙여넣기

10.9 Yara Menu

F5	선택한 악성코드에 룰 적용하기

10.10 Editor

Ctrl+E	가능한 모든 완료들을 표시