HP Security Research Blog

The HP Security Research blog provides a platform for security experts from across HP to discuss innovative research, industry observations, and updates on the threat landscape to help organizations proactively identify and manage risk.

Playing with Adobe Flash Player Exploits and Byte Code

Matt_Oh | June 10, 2014 - last edited June 11, 2014

Post a Comment

Adobe Flash Player has been a major target for exploits and malware in recent years. I wrote about CVE-2014-1776 and CVE-2014-0515 exploits just a few weeks ago. CVE-2014-1776 is an IE vulnerability, but the exploit found in the wild was using an Adobe Flash Player file to achieve reliable exploitation against ASLR and DEP. CVE-2014-0515 was a vulnerability with the Adobe Flash Player Pixel Bender component.

Basically, SWF files are not something you can avoid analyzing if you are dealing with real-life exploits. A good methodology when analyzing SWF files is also very beneficial for current malware research. I talked about automating SWF exploits and malware analysis in a previous presentation, but here I want to share a more manual methodology you can use for daily research. All the tools are free and some of them are open source. For this example, I used a sample with a SHA1 value of *300a7e4d54eca8641d7a19ceb4ab68bb76696816*. This sample exploits the CVE-2014-0515 vulnerability.

Get ready...

Basically, there are many tools, commercial or otherwise, that you can use to analyze SWF files. There are many different decompilers available. One of the best tools I found was JPEXS Free Flash Decompiler aka *FFDec*. I use it to acquire source code for ActionScript byte code. Adobe Flex SDK is also a useful tool. It has a very basic SWF dumping tool called *swfdump.exe* and can dump ActionScript byte code, too. The *mxmlc.exe* is a compiler that can compile Flex applications, but you can also use it to compile ActionScript files to a SWF file. The RABCDAsm package is also used here. It is an ActionScript disassembler/assembler package. The following tool sets are provided with this package:

- abcexport.exe: Export ActionScript byte code from a SWF file
- abcreplace.exe: Embed ActionScript byte code to a SWF file
- swfbinexport.exe: Export DefineBinaryData tag from a SWF file
- swfbinreplace.exe: Embed DefineBinaryData tag from a SWF file
- rabcasm.exe: ActionScript byte code assembler
- rabcasm.exe: ActionScript byte code disassembler

The first step when analyzing a SWF exploit is to look at its various components. One SWF file is composed of multiple tags. Figure 1 shows the output from the *swfdump* tool. From this example, the *DefineBinaryData* and *DoABC2* tags look interesting. The *DefineBinaryData* tag contains binary data used in other places and the *DoABC2* tag contains ActionScript byte code.

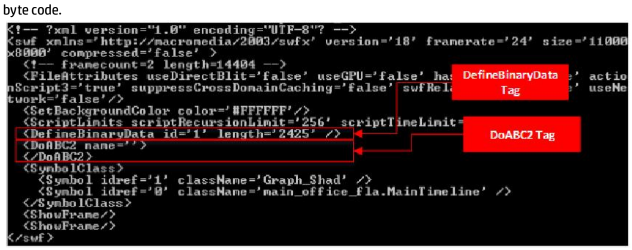

Figure 1 Swfdump result

You especially need to find a way to play with the ActionScript code, as it contains core parts of the exploit in many cases. I suggest two different methodologies here. One is the 'decompile and reconstruct' method and the other is the 'disassemble and reconstruct' method. Choosing the right method depends on how the decompiling technique works against the sample files. The 'decompile and reconstruct' method acquires source code through the decompiler and reconstructs the exploit after modification. The good thing about this method is that you can freely edit the source code and experiment with it as you wish. The bad thing is, in many cases, the decompiler won't work on malicious SWF files. When the decompiler fails, your only remaining option is to use the 'disassemble and reconstruct' approach. I'm going to explain both methodologies here.

Set...

When debugging SWF samples, it is beneficial to use a debug version of Flash player. You can download the developer versions of Flash Player here. With these developer binaries, you need to setup a Flash Player environment with *mm.cfg*. You can read more detailed instructions here. For Windows systems, you need to place an *mm.cfg* file under the user profile folder and add lines related to *trace* functionality. The *mm.cfg* file I used has the following lines to enable error reporting and *trace* output:

ErrorReportingEnable=1
TraceOutputFileEnable=1

When a SWF file is run in the debug version of Flash Player, it generates a debug log file in the following location on Windows systems:

%PROFILE%\AppData\Roaming\Macromedia\Flash Player\Logs\flashlog.txt

Go! The decompile & reconstruct method

Some Flash-based exploits and malware can be decompiled without any issues, as was the case with our example. Figure 2 shows the screen when I decompiled our sample file. You can export source code files to a folder by pressing the *Export selection* button.

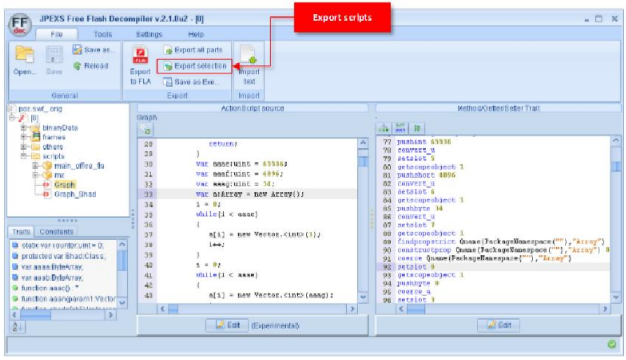

Figure 2 FFDec decompiling

Figure 3 shows the source code that is exported from *FFDec*. You just need to compile it back to a SWF file after modifying the code as you want.

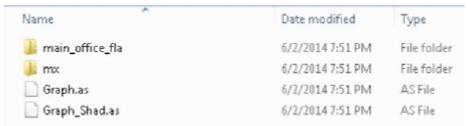

Figure 3 Exported source code

Figure 4 shows good examples when adding *trace* messages to the important parts of the exploit. The additional code dumps out bytes that are passed to a vulnerable object and later dumps out an index of an array element that is corrupted by this operation.

Figure 4 Adding trace messages

Adobe provides a free compiler (*mxmlc*) with Adobe Flex SDK. The main class of the source code is the *main_office_fla\MainTimeline.as* file. The following command line shows how you can compile the whole source code tree. The *-omit-trace-statements=false* option prevents *trace* statements from being removed by optimization.

mxmlc -omit-trace-statements=false -static-link-runtime-shared-libraries=true -compiler.source-path=. main_office_fla\MainTimeline.as

Figure 5 shows the *mxmlc* command result: You might see some warnings and errors. In many cases, you can ignore warnings on no type declarations, but if critical errors happen, you need to figure out how to fix them. Sometimes, the errors mean that the decompilation result itself might be erroneous. In that case, you'd be better off going with the 'disassemble and reconstruct' method.

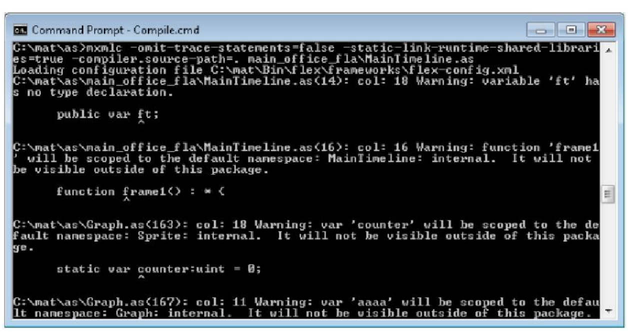

Figure 5 mxmlc result

Now we have built *main_office_fla\MainTimeline.swf*. The following command extracts the *DoABC2* tag from the SWF file and saves it to the *main_office_fla\MainTimeline-0.abc* byte code file:

abcexport main_office_fla\MainTimeline.swf

The following command embeds the *main_office_fla\MainTimeline-0.abc* byte code file back to original poc.swf_ file:

abcreplace.exe poc.swf_ 0 main_office_fla\MainTimeline-0.abc

Now we have an exploit file that has our own modified version of ActionScript byte code embedded in it. The benefit of this method is that it will not touch other parts of the SWF file. Sometimes malicious data presents in areas other than the DoABC2 byte code tag. When that happens, working on those tags with SWF-related tools can destroy the original format. In that case the 'decompile and reconstruct' method is useful for preserving the original malicious areas.

Disassemble & reconstruct

This method is more interesting than the 'decompile and reconstruct' method. Often when you're dealing with SWF files, they may use heavy obfuscation. Other exploits use the glitch in the ActionScript byte code interpreter for exploitation. In that case, the byte code itself is broken in such a way that decompiling is not possible. The decompiler technology is very helpful, but there are too many cases where it can fail. The worst problem is a silent fail. When the decompiler presents you with valid-looking ActionScript code, there is always a chance that the interpretation of the byte code is wrong. There is no good way to verify that the decompiled source code is valid. You should always be careful when you're dealing with malicious SWF files.

The following command exports the ActionScript byte code to a separate file named *poc-0.abc*.

*abcexport poc.swf_*

Now you can use a disassembler to disassemble the *poc-0.abc* file. The following command disassembles the *poc-0.abc* file and creates a folder named *poc-0* with ActionScript disassembly files. (Figure 6)

rabcdasm.exe poc-0.abc

Name	Date modified	Type
main_office_fla	6/3/2014 3:01 PM	File folder
mx	6/3/2014 3:01 PM	File folder
Graph.class.asasm	6/4/2014 10:25 AM	ASASM File
Graph.script.asasm	6/3/2014 3:01 PM	ASASM File
Graph_Shad.class.asasm	6/3/2014 3:01 PM	ASASM File
Graph_Shad.script.asasm	6/3/2014 3:01 PM	ASASM File
poc-0.main.asasm	6/3/2014 3:05 PM	ASASM File

Figure 6 ActionScript disassembly files

You can edit these ASASM files and re-assemble them to a SWF file. To work on these files, you need to understand ActionScript byte code instructions. A good reference for the ActionScript virtual machine and byte code can be found here. Figure 7 shows a good example of using the *trace* call to dump out useful information. It resolves the *trace* function first (*findpropstrict*), after pushing a string object (*pushstring*) and calling a *trace* function (*callpropvoid*). This dumps out a message to a log file when the routine is called.

```
class
 refid "Graph"
 instance QName(PackageNamespace(""), "Graph")
  extends QName(PackageNamespace("flash.display"), "Sprite")
  flag SEALED
  flag PROTECTEDNS
  protectedns ProtectedNamespace("Graph")
  iinit
   refid "Graph/instance/init"
   flag NEED_ACTIVATION
   body
    maxstack 6
    localcount 4
    initscopedepth 10
    maxscopedepth 15
    code
     getlocal0
     pushscope

     newactivation
     dup
     setlocal1

     pushscope

     getscopeobject      1
     pushundefined
     coerce_a
     setslot             1

|>   findpropstrict      QName(PackageNamespace(""), "trace")
|>   pushstring          "Graph init called"
|>   callpropvoid        QName(PackageNamespace(""), "trace"), 1
|>
     getscopeobject      1
     pushundefined
     coerce_a
```

Trace debug statement

Figure 7 Adding a trace statement

Calling a simple *trace* function may not be enough; in many cases, you might want a more complicated operation. However, writing very complicated logic with byte code instructions is challenging. One method we can use is to write a utility function that can do complex operations in a separate ActionScript file and combine it with existing code. You can just call the utility method and provide data to it for further processing. For example, if we want to dump byte array data in a hex form, we create an *Util.as* file with *DumpByteArray* method. (Figure 8)

```
package
{
        import flash.text.*;
        import flash.utils.*;
        import flash.net.*;

        public class Util
        {
                public function Util() {
                }

                public static function DumpByteArray(buffer:ByteArray):void
                {
                        var out:String = fillUp("Offset", 8, " ") + "  00 01 02 03 04 05 06
0F\n";
                        var offset:int = 0;
                        var l:int = buffer.length;
                        var row:String = "";
                        buffer.position = 0;
                        for (var i:int = 0; i < l; i += 15)
                        {
                                row += fillUp(offset.toString(16).toUpperCase(), 8, "0") +
                                var n:int = Math.min(16, buffer.length - buffer.position);
                                var string:String = "";
                                for (var j:int = 0; j < 16; ++j)
                                {
                                        if (j < n)
                                        {
                                                        var value:int = buffer.readUnsigned
                                                        string += value >= 32 ? String.from
                                                        row += fillUp(value.toString(16).to
";
                                                        offset++;
                                        }
                                        else
                                        {
                                                        row += "        ";
                                                        string += " ";
                                        }
                                }
                                row += " " + string + "\n";
                        }
                        out += row;

                        trace(out);
                }
```

Figure 8 Util class with DumpByteArray method

Next, we compile the *Util.as* file, export the *DoABC2* tag, and disassemble it to ASASM files using the following command sequences. The folder *Util-0* has multiple ASASM files in it after this. You can just copy them to the original *poc-0* folder. (Figure 9)

mxmlc -omit-trace-statements=false -static-link-runtime-shared-libraries=true -compiler.source-path=. Util.as
abcexport Util.swf
rabcdasm Util-0.abc

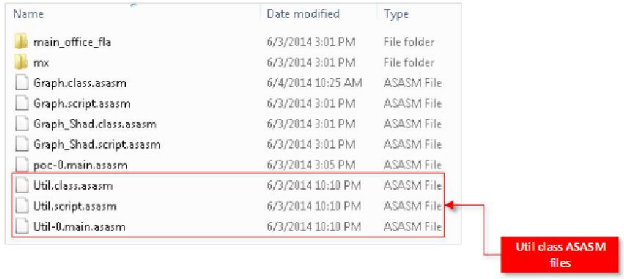

Figure 9 Add Util ASASM files to the existing package

Now you need to include the *Util* class in the main package. You can achieve that by adding a *#include* line to the *poc-0.main.asasm* file. (Figure 10)

Figure 10 Adding Util.script.asasm to poc-0.main.asasm

After this, you are ready to use the *Util.DumpByteArray* method. Figure 11 shows a good example of how you can call this method from an ASASM file. First, it resolves the *Util* class (*getlex*) and passes a *Graph.Shad* object to the *DumpByteArray* method (*callpropvoid*). The part where it creates the *Graph.Shad* object is copied over from the existing instructions that follow:

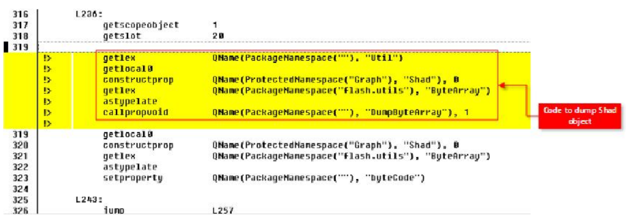

Figure 11 Adding a call to DumpByteArray method

The following command re-assembles the whole ASASM file tree and generates a *poc-0.main.abc* file with new byte code in it:

rabcasm.exe poc-0\poc-0.main.asasm

Now replace the ActionScript byte code of poc.swf_ with the contents of *poc-0.main.abc*:

abcreplace.exe poc.swf_ 0 poc-0\poc-0.main.abc

If you run the output SWF file on a debug version of Flash Player, a debug log is written to the flashlog.txt file. (Figure 12)

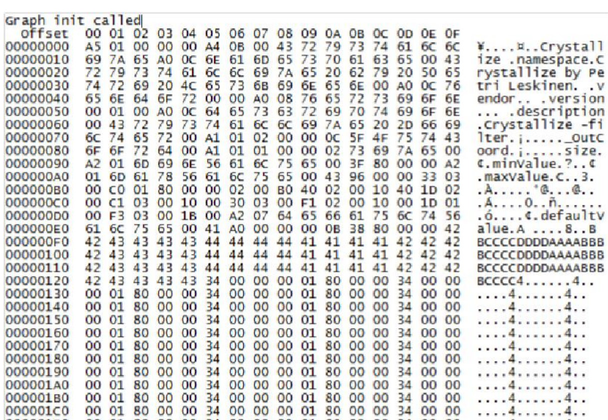

Figure 12 flashlog.txt content

Summary

Adobe Flash Player is often used for exploitation and critical vulnerabilities are still found in the wild. It can even be used for defeating ASLR and DEP with other vulnerabilities. Even though many malicious SWF samples can be decompiled without problems, there are some heavily obfuscated or manipulated files that defy decompilation. Disassembling them and re-assembling is a good method to use in these cases. Even though this method requires a good understanding of ActionScript byte code instructions, if it is used in the right way, it is very powerful.

Leave a Comment

We encourage you to share your comments on this post. Comments are moderated and will be reviewed and posted as promptly as possible during regular business hours

To ensure your comment is published, be sure to follow the Community Guidelines.

*Name

*Email

Website (optional)

http://

Rich Text | Preview

Quote

*Word verification by reCAPTCHA

Get a sound challenge Help with word verification

Cancel Post Your Comment

United States

About HP
Contact us
Newsroom
Investor relations
Living Progress
Accessibility
Events
HP Labs
Jobs

Social Media
Consumer support forum
Enterprise business community
Developer community
Corporate blogs

HP Partners
Become a partner
Find a reseller
PartnerOne

Customer Support
Power cord replacement
Download drivers
Register your product
HP replacement parts
Authorized service providers
Training & certification
Product recycling