0patch Blog

Welcome to the era of vulnerability micropatching

Friday, June 17, 2016

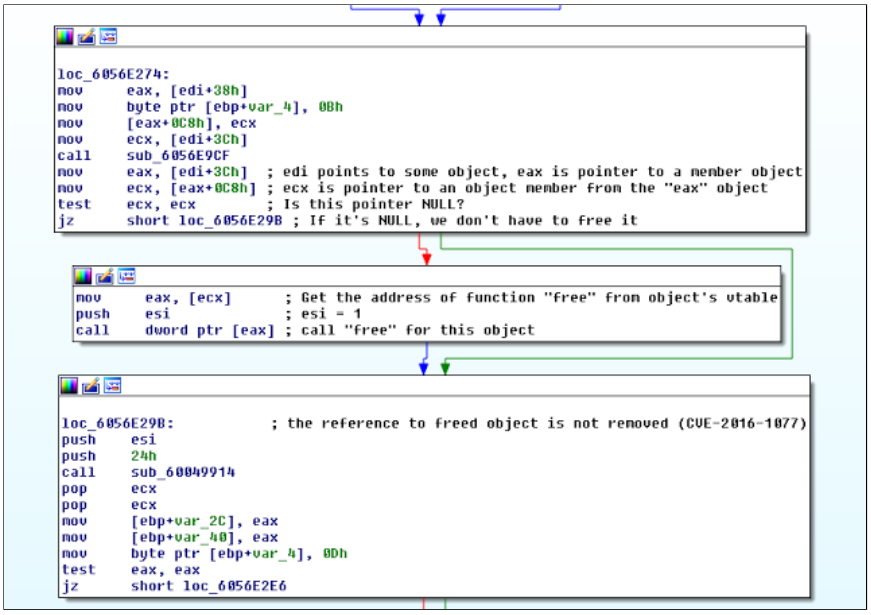
Writing a 0patch for Acrobat Reader's Use-After-Free Vulnerability CVE-2016-1077

This blog post will describe how we created a 0patch for CVE-2016-1077, a use-after-free vulnerability in Adobe's Acrobat Reader DC version 15.010.20060 for Windows.

According to Adobe Security Bulletin APSB16-14, this vulnerability was reported to Adobe by Pier-Luc Maltais of COSIG, and shortly after the release of this bulletin, a proof-of-concept exploit was published on Packet Storm. This *PoC* is in the form of a PDF document that crashes the Reader when opened. (By the way, it was apparently created with single-byte raw file fuzzing; we were able to compare it to the original PDF file we found on the Internet.)

This was enough for us to start analyzing the vulnerability. To keep this blog post reasonably short, we will omit the vulnerability analysis process, which is nicely described in this detailed post by Fortinet's Kai Lu and Kushal Arvind Shah for another identical vulnerability in the very same function as ours.

So let's fast-forward to the point where we know exactly what the flaw is. We have a use-after-free issue, as shown in the vulnerable code below.



Vulnerable code: after freeing an object, reference to this object is not removed

The above image shows three relevant code blocks somewhere in Acrobat Reader's implementation of *deflate* operation on an XObject (an element in a PDF document), where the code is obviously doing some clean-up by freeing an object it no longer needs. The pointer to this object is retrieved in register `ecx`, then the `test ecx, ecx` instruction effectively checks if the pointer is NULL, and this determines whether the execution will continue on the red branch (`ecx` not NULL) or the green branch (`ecx` is NULL). In the red branch, pointer to object's `free` function is copied from object's *vtable* (`free` is the first function in *vtable* in this case) to register `eax`, then a call to `free` is made.

About 0patch

0patch (pronounced 'zero patch') is a platform for instantly distributing, applying and removing microscopic binary patches to/from running processes without having to restart these processes (much less reboot the entire computer). Brought to you by ACROS Security.

Follow @0patch

Blog Archive

The relevant code can be represented by this simple pseudo-code:

```
ObjectA* a; // stored in edi register
ObjectB* b; // stored in eax register
ObjectC* c; // stored in ecx register
...
b = a->some_objectB;
c = b->some_objectC;
if (c) // the red branch
  delete c;
...
```

Note that after `c` is deleted, its reference (pointer to it) in `b->some_objectC` is not removed. And sure enough, at some point later in the execution, `b->some_objectC` is accessed again. But this time `c` points to an already deallocated memory address, and access violation occurs.

Now let's see how we would fix the above pseudo-code. It's trivial, we only need to write NULL to `b->some_objectC` after deleting `c`:

```
...
b = a->some_objectB;
c = b->some_objectC;
if (c)
  delete c;
b->some_objectC = NULL; // destroy reference to deleted object
...
```

So we have a pseudo-code patch, now let's translate it to machine code. First look at the vulnerable code in a "text" view:

```
6056E274 8B 47 38           mov   eax, [edi+38h]
6056E277 C6 45 FC 0B        mov   byte ptr [ebp+var_4], 0Bh
6056E27B 89 88 C8 00 00 00  mov   [eax+0C8h], ecx
6056E281 8B 4F 3C           mov   ecx, [edi+3Ch]
6056E284 E8 46 07 00 00     call  sub_6056E9CF
6056E289 8B 47 3C           mov   eax, [edi+3Ch]  ; edi points to some object, eax is pointer to
                                                    a member object
6056E28C 8B 88 C8 00 00 00  mov   ecx, [eax+0C8h] ; ecx is pointer to an object member from the
                                                    "eax" object
6056E292 85 C9              test  ecx, ecx        ; Is this pointer NULL?
6056E294 74 05              jz    short loc_6056E29B ; If it's NULL, we don't have to free it


6056E296 8B 01              mov   eax, [ecx]      ; Get the address of function "free" from
                                                    object's vtable
6056E298 56                 push  esi             ; esi = 1
6056E299 FF 10              call  dword ptr [eax] ; call "free" for this object


6056E29B    loc_6056E29B:                         ; CODE XREF: sub_6056DC50+644 j
6056E29B 56                 push  esi             ; the reference to freed object is not removed
                                                    (CVE-2016-1077)
6056E29C 6A 24              push  24h
6056E29E E8 71 B6 AD FF     call  sub_60049914
6056E2A3 59                 pop   ecx
6056E2A4 59                 pop   ecx
```

The patch code must come after `call dword ptr [eax]`, which is a call to object's `free`. We need the address where the pointer to the object is stored, so we can put a NULL there.This address was `[edi+3Ch]+C8h` just a few instructions earlier, and we know `edi` is not corrupted by the call to `free` as it is still being used later in the code without being assigned a new value. (This is easy to check with IDA as it highlights all uses of a register when you click on one.)

When creating a micropatch, we need to be careful about a few things:

1. Injecting a micropatch works by overwriting one or more existing instructions with a 5-byte `JMP` instruction to some nearby-allocated memory block, where the overwritten instructions are added at the end of our patch code, followed by a `JMP` back to the place in the original code right after them. Of the overwritten instructions, the first one may be a destination point of some `CALL` or `JMP`, but others may not be, as that would certainly lead to an error in execution. Fortunately, IDA makes it really easy to verify this with its graphical representation of code.
2. We should not cause any unwanted side-effects, like corrupting values of registers or stack-based variables that subsequent code may still be using. When in doubt, we should `PUSH/POP` the registers we use or otherwise preserve them.
3. We should not merely *assume* that a register value will survive a function call (e.g., that `edi` will survive a call to `free` in our case) - we must *prove* it by either reviewing this function (and all functions it calls etc.) and showing that the value is reliably preserved, or showing that the original code after the function call relies on the fact that the register hasn't changed.
4. Our code should be as easy as possible to review. On one hand, this means using as few instructions as possible, and on the other hand it means making it easy for a reviewer to verify our claims upon which the patch code is constructed (e.g., that "patch code can safely change `eax` because...", or "at this point, `edx` will always point to..."). Sometimes, these goals are in contradiction: for example, in our patch code below, we could simply use `eax` instead of `edi` and avoid having to `PUSH/POP edi`, but it would make code review harder.

First, let's find a suitable location for our patch. There's an obvious candidate immediately after `call dword ptr [eax]` at location `6056E29B` (`AcroRd32.dll + 0x56E29B`). The instructions there are:

```
6056E29B 56                push  esi
6056E29C 6A 24             push  24h
6056E29E E8 71 B6 AD FF    call  sub_60049914
```

Remember, we need 5 bytes for our "`JMP` to patch", so we'll have to relocate all these three instructions to the space right after our patch code. Note that the third instruction is a relative `CALL`, which means that its offset has to be recalculated when it is moved. (Don't worry, 0patch agent does this automatically.)

As for the patch code, let's try this:

```
push edi                    ; store edi as subsequent code still needs it
mov edi, dword[edi+03ch]    ; edi = pointer to object b in pseudo code
mov dword[edi+0c8h], 0h     ; set pointer to object c to NULL
pop edi                     ; restore edi
```

This code just uses a single register (`edi`), making it trivial for a reviewer to see that it has no side effects while writing a NULL where the original code forgot to do it. As already noted, we could make it even shorter if we used `eax` instead of `edi`, as we wouldn't have to preserve `eax`'s value. However, *proving* that we don't have to preserve `eax`'s value would require reviewing quite some additional code after our patch location, including a function that is called there. This would consume patch developer's as well as patch reviewer's time, and increase the risk of error for both of them.

One last check: Can we be absolutely sure that `edi` will point to the same object `a` at address `6056E29B` as it did a couple of instructions back at address `6056E289`? There are two execution paths between these two addresses (as IDA nicely shows). One goes through the aforementioned green branch, and the other goes through the red branch. The "green branch" path is easy to prove: there is simply no instruction there that would change `edi`, so it must still have the same value at the end. The "red branch" path, however, includes a call to `free` from object's vtable. Proving that this function does not alter `edi` could be a daunting task, especially as IDA can't show you where the code behind this dynamically-set `free` is. (In fact, this function could be different for various types of objects being processed here, and only the original developers of this code know what objects that could possibly be.) So we'll use another approach: we look at the code after our patch location to see whether it relies on `edi` being the same as before. And in fact, we can see that it does (but that's beyond the scope of this article).

We have all we need, but how do we turn this into an actual 0patch? We use *0patch Factory*. 0patch Factory is a simple tool we wrote, which takes a *0patch source file* as input, and turns it into a 0patch that can be immediately applied on any computer with our 0patch Agent. Let's take a look at a 0patch source file for this micropatch:

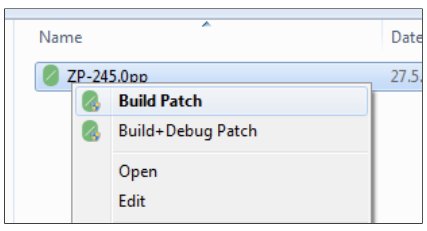
```
MODULE_PATH "C:\Program Files (x86)\Adobe\Acrobat Reader
DC\Reader\AcroRd32.dll"
PATCH_ID 245
```

```
PATCH_FORMAT_VER 2
VULN_ID 1272
PLATFORM win32

patchlet_start
PATCHLET_ID 1
PATCHLET_TYPE 2

PATCHLET_OFFSET 0x56e29b
N_ORIGINALBYTES 5

code_start

    push edi
    mov edi, dword[edi+03ch]
    mov dword[edi+0c8h], 0h
    pop edi

code_end
patchlet_end
```

Some explanation is probably in order:

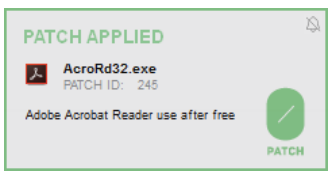
- Each 0patch consists of one or more *patchlets*; a patchlet is a set of machine code instructions that are to be injected at a specific offset from the module's base.
- `MODULE_PATH` is full path to the module (binary file) we're patching. 0patch Factory needs this for two reasons: (1) calculating the module's hash, as the patch should only be applied to this *exact* binary, and (2) extracting the original bytes from all locations where we're injecting our patchlets, as any patchlet should only be applied when the bytes found in the "live" module on its injection location exactly match these original bytes.
- `PATCH_ID` is a unique ID of the patch in the 0patch database. It can be arbitrary during patch development, but a patch will get a suitable unique ID before it gets signed and uploaded to the server.
- `PATCH_FORMAT_VER` is the version of the format in which the patch is written. 2 is the only supported value at this time.
- `VULN_ID` is the ID of the vulnerability (in the 0patch database) fixed by this patch.
- `PLATFORM` is either `Win32` or `Win64` at this time, specifying the format of machine code in this patch's patchlets. Acrobat Reader DC we're patching here is a 32-bit application.
- `patchlet_start` and `patchlet_end` mark beginning and end of an individual patchlet.
- `PATCHLET_ID` is a unique ID of a patchlet inside a patch, usually iterative from 1 upwards.
- `PATCHLET_TYPE` is the type of the patchlet, currently the only supported type is type 2 (which means "code injection patchlet"). In the future, we'll support other types of patchlets.
- `PATCHLET_OFFSET` is the offset from the module's base where the patchlet code should be injected. In our case, this offset is `0x56e29b`.
- `N_ORIGINALBYTES` is the number of original bytes that we should check when applying the patchlet. The default is 5, which means all 5 bytes that we will overwrite with the "`JMP` to patch" instruction.
- `code_start` and `code_end` mark beginning and end of patchlet's code, which will be compiled to binary code by 0patch Factory.

We store this 0patch source file in `ZP-245.0pp` and build the patch using 0patch Factory on a computer with 0patch Agent installed:



0patch Factory makes it easy to build a micropatch: just right-click a *0pp* file and select "Build Patch"

This both compiles and installs the new 0patch, making it ready for immediate application to a running vulnerable Acrobat Reader. Without further ado, let's open `POC_ZP-245.pdf` to test our patch.

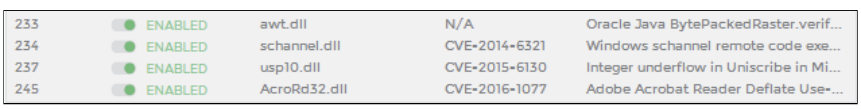


0patch Agent notifies you that patch 245 just got applied to the running Acrobat Reader

Acrobat Reader gets launched, the "Patch Applied" pop-up is shown notifying us that patch 245 was applied to `AcroRd32.exe`, and... no crash. The document seems to be displayed correctly. We scroll down and get the "A drawing error occurred." error message from the Reader, but this is expected, as there is still a corrupt graphical element in the PDF document, and Reader correctly notifies us about that. (Latest version of Reader also shows this message.) We can continue to use Acrobat Reader as if nothing has happened.

There you go, we've just created a simple micropatch for a remotely executable memory corruption vulnerability. It consists of just four machine code instructions and can be instantaneously applied to a running Acrobat Reader while a user is reading some document with it.

Should you like to experiment with this micropatch on your own, create a free 0patch account and download 0patch Agent for Windows if you haven't already. (If you have, than patch ZP-245 has probably already been downloaded by your agent and is waiting for you to run the vulnerable Acrobat Reader.)



Patch 245 is now in the "Patches" list in the 0patch Console, where you can enable or disable it

Once you have 0patch Agent installed and registered, install Acrobat Reader DC 15.010.20060 and use it to open `POC_ZP-245.pdf`, first with patch 245 enabled, and then disabled. Notice the difference? ;)

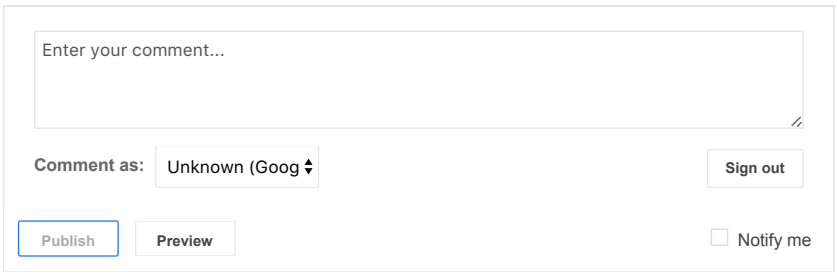
If you'd like to write your own 0patches, and we sure hope many of you would, give us some time to polish our *0patch Factory* for you. It's nothing fancy, but you deserve a decently tested tool with useful instructions. In the mean time, please send us an email at crowdpatching@0patch.com to express your interest. We look forward to our collaboration in changing the way vulnerabilities are getting fixed.

@mkolsek
@0patch

Posted by Mitja Kolsek on June 17, 2016

No comments:

Post a Comment



Newer Post Home Older Post

Subscribe to: Post Comments (Atom)