Zero-day vulnerability in Desktop Window Manager (CVE-2021-28310) used in the wild

SL securelist.com/zero-day-vulnerability-in-desktop-window-manager-cve-2021-28310-used-in-the-wild/101898

While analyzing the CVE-2021-1732 exploit originally discovered by the DBAPPSecurity Threat Intelligence Center and used by the BITTER APT group, we discovered another zero-day exploit we believe is linked to the same actor. We reported this new exploit to Microsoft in February and after confirmation that it is indeed a zero-day, it received the designation CVE-2021-28310. Microsoft released a patch to this vulnerability as a part of its April security updates.

We believe this exploit is used in the wild, potentially by several threat actors. It is an escalation of privilege (EoP) exploit that is likely used together with other browser exploits to escape sandboxes or get system privileges for further access. Unfortunately, we weren't able to capture a full chain, so we don't know if the exploit is used with another browser zero-day, or coupled with known, patched vulnerabilities.

The exploit was initially identified by our advanced exploit prevention technology and related detection records. In fact, over the past few years, we have built a multitude of exploit protection technologies into our products that have detected several zero-days, proving their effectiveness time and again. We will continue to improve defenses for our users by enhancing technologies and working with third-party vendors to patch vulnerabilities, making the internet more secure for everyone. In this blog we provide a technical analysis of the vulnerability and how the bad guys exploited it. More information about BITTER APT and IOCs are available to customers of the Kaspersky Intelligence Reporting service. Contact: intelreports@kaspersky.com.

Technical details

CVE-2021-28310 is an out-of-bounds (OOB) write vulnerability in dwmcore.dll, which is part of Desktop Window Manager (dwm.exe). Due to the lack of bounds checking, attackers are able to create a situation that allows them to write controlled data at a controlled offset using DirectComposition API. DirectComposition is a Windows component that was introduced in Windows 8 to enable bitmap composition with transforms, effects and animations, with support for bitmaps of different sources (GDI, DirectX, etc.). We've already published a blogpost about in-the-wild zero-days abusing DirectComposition API. DirectComposition API is implemented by the win32kbase.sys driver and the names of all related syscalls start with the string "NtDComposition".

DirectComposition syscalls in the win32kbase.sys driver

For exploitation only three syscalls are required: NtDCompositionCreateChannel, NtDCompositionProcessChannelBatchBuffer and NtDCompositionCommitChannel. The NtDCompositionCreateChannel syscall initiates a channel that can be used together with the NtDCompositionProcessChannelBatchBuffer syscall to send multiple DirectComposition commands in one go for processing by the kernel in a batch mode. For this to work, commands need to be written sequentially in a special buffer mapped by NtDCompositionCreateChannel syscall. Each command has its own format with a variable length and list of parameters.

```
enum DCOMPOSITION_COMMAND_ID
{
ProcessCommandBufferIterator,
CreateResource,
OpenSharedResource,
ReleaseResource,
GetAnimationTime,
CapturePointer,
OpenSharedResourceHandle,
SetResourceCallbackId,
SetResourceIntegerProperty,
SetResourceFloatProperty,
SetResourceHandleProperty,
SetResourceHandleArrayProperty,
SetResourceBufferProperty,
SetResourceReferenceProperty,
SetResourceReferenceArrayProperty,
SetResourceAnimationProperty,
SetResourceDeletedNotificationTag,
AddVisualChild,
RedirectMouseToHwnd,
SetVisualInputSink,
RemoveVisualChild
};
```

List of command IDs supported by the function DirectComposition::CApplicationChannel::ProcessCommandBufferIterator

While these commands are processed by the kernel, they are also serialized into another format and passed by the Local Procedure Call (LPC) protocol to the Desktop Window Manager (dwm.exe) process for rendering to the screen. This procedure could be initiated by the third syscall – NtDCompositionCommitChannel.

To trigger the vulnerability the discovered exploit uses three types of commands: CreateResource, ReleaseResource and SetResourceBufferProperty.

```
1  void CreateResourceCmd(int resourceId)
2  {
3  DWORD *buf = (DWORD *)((PUCHAR)pMappedAddress + BatchLength);
4  *buf = CreateResource;
5  buf[1] = resourceId;
6  buf[2] = PropertySet; // MIL_RESOURCE_TYPE
7  buf[3] = FALSE;
8  BatchLength += 16;
9  }
10
11 void ReleaseResourceCmd(int resourceId)
12 {
13 DWORD *buf = (DWORD *)((PUCHAR)pMappedAddress + BatchLength);
14 *buf = ReleaseResource;
15 buf[1] = resourceId;
16 BatchLength += 8;
17 }
18
19 void SetPropertyCmd(int resourceId, bool update, int propertyId, int storageOffset, int hidword, int lodword)
20 {
21 DWORD *buf = (DWORD *)((PUCHAR)pMappedAddress + BatchLength);
22 *buf = SetResourceBufferProperty;
23 buf[1] = resourceId;
24 buf[2] = update;
25 buf[3] = 20;
26 buf[4] = propertyId;
27 buf[5] = storageOffset;
28 buf[6] = _D2DVector2; // DCOMPOSITION_EXPRESSION_TYPE
29 buf[7] = hidword;
30 buf[8] = lodword;
31 BatchLength += 36;
32 }
```

Format of commands used in exploitation

Let's take a look at the function CPropertySet::ProcessSetPropertyValue in dwmcore.dll. This function is responsible for processing the SetResourceBufferProperty command. We are most interested in the code responsible for handling DCOMPOSITION_EXPRESSION_TYPE = D2DVector2.

```
1  int CPropertySet::ProcessSetPropertyValue(CPropertySet *this, ...)
2  {
3    ...
4
5    if (expression_type == _D2DVector2)
6    {
7      if (!update)
8      {
9        CPropertySet::AddProperty<D2DVector2>(this, propertyId, storageOffset, _D2DVector2, value);
10     }
11     else
12     {
13       if ( storageOffset != this->properties[propertyId]->offset & 0x1FFFFFFF )
14       {
15         goto fail;
16       }
17
18       CPropertySet::UpdateProperty<D2DVector2>(this, propertyId, _D2DVector2, value);
19     }
20   }
21
22   ...
23 }
24
25 int CPropertySet::AddProperty<D2DVector2>(CResource *this, unsigned int propertyId, int storageOffset, int
   type, _QWORD *value)
26 {
27   int propertyIdAdded;
28
29   int result =
30 PropertySetStorage<DynArrayNoZero,PropertySetUserModeAllocator>::AddProperty<D2DVector2>(
31     this->propertiesData,
32     type,
33     value,
```

```
34      &propertyIdAdded);
35    if ( result < 0 )
36    {
37      return result;
38    }
39
40    if ( propertyId != propertyIdAdded || storageOffset != this->properties[propertyId]->offset & 0x1FFFFFFF )
41    {
42      return 0x88980403;
43    }
44
45    result = CPropertySet::PropertyUpdated<D2DMatrix>(this, propertyId);
46    if ( result < 0 )
47    {
48      return result;
49    }
50
51    return 0;
52  }
53
54  int CPropertySet::UpdateProperty<D2DVector2>(CResource *this, unsigned int propertyId, int type,
    _QWORD *value)
55  {
56    if ( this->properties[propertyId]->type == type )
57    {
58      *(_QWORD *)(this->propertiesData + (this->properties[propertyId]->offset & 0x1FFFFFFF)) = *value;
59
60      int result = CPropertySet::PropertyUpdated<D2DMatrix>(this, propertyId);
61      if ( result < 0 )
62      {
63        return result;
64      }
65
66      return 0;
67    }
68    else
69
```

```
70    {
71      return 0x80070057;
      }
    }
```

Processing of the SetResourceBufferProperty (D2DVector2) command in dwmcore.dll

For the SetResourceBufferProperty command with the expression type set to D2DVector2, the function CPropertySet::ProcessSetPropertyValue(...) would either call CPropertySet::AddProperty<D2DVector2>(...) or CPropertySet::UpdateProperty<D2DVector2>(...) depending on whether the update flag is set in the command. The first thing that catches the eye is the way the new property is added in the CPropertySet::AddProperty<D2DVector2>(...) function. You can see that it adds a new property to the resource, but it only checks if the propertyId and storageOffset of a new property are equal to the provided values after the new property is added, and returns an error if that's not the case. Checking something after a job is done is bad coding practice and can result in vulnerabilities. However, a real issue can be found in the CPropertySet::UpdateProperty<D2DVector2>(...) function. No check takes place that will ensure if the provided propertyId is less than the count of properties added to the resource. As a result, an attacker can use this function to perform an OOB write past the propertiesData buffer if it manages to bypass two additional checks for data inside the properties array.

```
1  (1) storageOffset == this->properties[propertyId]->offset & 0x1FFFFFFF
2  (2) this->properties[propertyId]->type == type
```

Conditions which need to be met for exploitation in dwmcore.dll

These checks could be bypassed if an attacker is able to allocate and release objects in the dwm.exe process to groom heap into the desired state and spray memory at specific locations with fake properties. The discovered exploit manages to do this using the CreateResource, ReleaseResource and SetResourceBufferProperty commands.

At the time of writing, we still hadn't analyzed the updated binaries that are fixing this vulnerability, but to exclude the possibility of other variants for this vulnerability Microsoft would need to check the count of properties for other expression types as well.

Even with the above issues in dwmcore.dll, if the desired memory state is achieved to bypass the previously mentioned checks and a batch of commands are issued to trigger the vulnerability, it still won't be triggered because there is one more thing preventing it from happening.

As mentioned above, commands are first processed by the kernel and only after that are they sent to Desktop Window Manager (dwm.exe). This means that if you try to send a command with an invalid propertyId, NtDCompositionProcessChannelBatchBuffer syscall will return an error and the command will not be passed to the dwm.exe process. SetResourceBufferProperty commands with expression type set to D2DVector2 are processed in the win32kbase.sys driver with the functions DirectComposition::CPropertySetMarshaler::AddProperty<D2DVector2>(...) and DirectComposition::CPropertySetMarshaler::UpdateProperty<D2DVector2>(...), which are very similar to those present in dwmcore.dll (it's quite likely they were copy-pasted). However, the kernel version of the UpdateProperty<D2DVector2> function has one notable difference – it actually checks the count of properties added to the resource.

```
int DirectComposition::CPropertySetMarshaler::UpdateProperty<D2DVector2>
(DirectComposition::CPropertySetMarshaler *this, unsigned int *commandParams, _QWORD *value)
{
  unsigned int propertyId = commandParams[0];
  unsigned int storageOffset = commandParams[1];
  unsigned int type = commandParams[2];

  if ( propertyId >= this->propertiesCount
    || storageOffset != this->properties[propertyId]->offset & 0x1FFFFFFF)
    || type != this->properties[propertyId]->type )
  {
    return 0xC000000D;
  }
  else
  {
    *(_QWORD *)(this->propertiesData + (this->properties[propertyId]->offset & 0x1FFFFFFF)) = *value;
    ...
  }
  return 0;
}
```

DirectComposition::CPropertySetMarshaler::UpdateProperty<D2DVector2>(...) in win32kbase.sys

The check for propertiesCount in the kernel mode version of the UpdateProperty<D2DVector2> function prevents further processing of a malicious command by its user mode twin and mitigates the vulnerability, but this is where DirectComposition::CPropertySetMarshaler::AddProperty<D2DVector2>(...) comes in to play. The kernel version of the AddProperty<D2DVector2> function works exactly like its user mode variant and it also applies the same behavior of checking property after it has already been added and returns an error if propertyId and storageOffset of the created property do not match the provided values. Because of this, it's possible to use the AddProperty<D2DVector2> function to add a new property and force the function to return an error and cause inconsistency between the number of properties assigned to the same resource in kernel mode/user mode. The propertiesCount check in the kernel could be bypassed this way and malicious commands would be passed to Desktop Window Manager (dwm.exe).

Inconsistency between the number of properties assigned to the same resource in kernel mode/user mode could be a source of other vulnerabilities, so we recommend Microsoft to change the behavior of the AddProperty function and check properties before they are added.

The whole exploitation process for the discovered exploit is as follows:

1. Create a large number of resources with properties of specific size to get heap into predictable state.
2. Create additional resources with properties of specific size and content to spray memory at specific locations with fake properties.
3. Release resources created at stage 2.

4. Create additional resources with properties. These resources will be used to perform OOB writes.
5. Make holes among resources created at stage 1.
6. Create additional properties for resources created at stage 4. Their buffers are expected to be allocated at specific locations.
7. Create “special” properties to cause inconsistency between the number of properties assigned to the same resource in kernel mode/user mode for resources created at stage 4.
8. Use OOB write vulnerability to write shellcode, create an object and get code execution.
9. Inject additional shellcode into another system process.

Kaspersky products detect this exploit with the verdicts:

- HEUR:Exploit.Win32.Generic
- HEUR:Trojan.Win32.Generic
- PDM:Exploit.Win32.Generic

Zero-day vulnerability in Desktop Window Manager (CVE-2021-28310) used in the wild

Your email address will not be published. Required fields are marked *