It rather involved being on the other side of this airtight hatchway: Invalid parameters from one security level crashing code at the same security level (doesn't get old)

devblogs.microsoft.com/oldnewthing/20160421-00

April 21, 2016

Raymond Chen

We received a very professional security vulnerability report from a video driver manufacturer which reported a security vulnerability in DirectX. You can tell it's a professional operation because the vulnerability report took the form of a neatly-formatted PowerPoint presentation with corporate logos in every corner. Unfortunately, you'll have to make do with my distillation into boring text.

> An attacker program can pass intentionally invalid buffers to various DirectX functions, resulting in a number of identified vulnerabilities, of which the `!exploitable` debugger extension classifies several as high severity.
>
> The root cause of all of the vulnerabilities is that the functions in question are passed buffers and sizes which are not validated. The functions read from or write to these improperly-sized buffers, which can result in memory corruption and potential remote code execution, information disclosure, data tampering, denial of service, and the collapse of Western civilization.
>
> For example, the `IDirect3DResource9::SetPrivateData` method does not verify that the `SizeOfData` is correct. If `SizeOfData` is larger than the actual size of the data, then DirectX will perform an invalid pointer read beyond the end of the buffer. This can be used to extract information from memory managed by DirectX, resulting in information disclosure.
>
> Similarly, the `IDirect3DResource9::GetPrivateData` method does not verify that the `*pSizeOfData` is correct. If `*pSizeOfData` is larger than the actual size of the data, then DirectX will perform an invalid pointer write beyond the end of the buffer. This can be used to corrupt memory managed by DirectX, leading to further exploits.

There is no vulnerability here yet because there is no elevation. I mean, sure, the attacker can pass bogus buffers and trick DirectX but let's take a closer look at who the attacker is and who the victim is.

The attacker is a rogue program that is intentionally passing invalid buffers to DirectX. DirectX runs in-process and tries to read data from or write results to those buffers. The attacker can trick DirectX into reading or writing past the end of the buffer by lying about the buffer size.

```
void UseDirectXToReadPastEndOfBuffer()
{
  direct3dResource9->SetPrivateData(
    GUID_Attack,
    buffer,
    incorrectSize,
    0);
}

void UseDirectXToWritePastEndOfBuffer()
{
  DWORD size = incorrectSize;
  direct3dResource9->GetPrivateData(
    GUID_Attack,
    buffer,
    &size);
}
```

But so what? The attacker *can already do that* without needing to trick DirectX into doing anything.

```
void JustReadPastEndOfBuffer()
{
  memcmp(buffer, buffer, incorrectSize);
}

void JustWritePastEndOfBuffer()
{
  memset(buffer, 0, incorrectSize);
}
```

Using DirectX to read memory from within your process is just adding style points. You aren't doing you can't already do. You're just going through a middle man and then blaming the middle man.

Of course, one must verify that corruption or exposure of data within the process causes no damage beyond the process. If the corrupted data crosses a security boundary (e.g., gets passed to a kernel mode video driver), the recipient of the data must be resilient to intentionally invalid parameters.

The `!exploitable` debugger extension classifies these failures as high severity because it is looking at the world through data fuzzing-colored glasses. It's assuming that all the data upon which the program is operating is potentially untrusted, and it's assessing how bad it would be if an attacker could trigger this crash by giving your application an intentionally

corrupted file. In other words, the `!exploitable` debugger extension assumes that the program is trusted but it is operating on untrusted data. Any crashes are assumed to the result of the trusted program mishandling the untrusted data.

From that standpoint, these are indeed severe errors if a corrupted file can trick your application into passing incorrect buffer sizes to DirectX. And that's the bug you need to fix in the program: Make sure your program passes correct buffer sizes regardless of the contents of the file being parsed.

But if the program itself is the attacker, then we have another instance of MS07-052: Code execution results in code execution.

Previous discussion:

Raymond Chen

Follow