The Story of Jian – How APT31 Stole and Used an Unknown Equation Group 0-Day

By Eyal Itkin

Published: 2021-02-22 · Archived: 2026-04-11 02:00:15 UTC

Research by: Eyal Itkin and Itay Cohen

There is a theory which states that if anyone will ever manage to steal and use nation-grade cyber tools, any network would become untrusted, and the world would become a very dangerous place to live in.

There is another theory which states that this has already happened.

What would you say if we told you that a foreign group managed to steal an American nuclear submarine? That would definitely be a bad thing, and would quickly reach every headline.

However, for cyber weapons – although their impact could be just as devastating – it`s usually a different story.

Cyber weapons are digital and volatile by nature. Stealing them and transferring from one continent to another, can be as simple as sending an email. They are also very obscure, and their mere existence is a closely guarded secret. That is exactly why, as opposed to a nuclear submarine, stealing a cyber-weapon can easily go under the radar and become a fact known only to a selected few.

The implications of such a scenario can be devastating, as the world have already experienced with the case of the Shadow Brokers leak, in which a mysterious group have decided to publicly publish a wide range of cyber weapons allegedly developed by the Tailored Access Operations (TAO) unit of the NSA – also referred to as the 'Equation Group'.
The Shadow Brokers leak lead to some of the biggest cyber outbreaks in history – the most famous of which was the WannaCry attack causing hundreds of millions of dollars in damages to organizations across the globe – and which its implications are still relevant even 3 years after it happened.

The Shadow brokers leak however, just gave us a taste of some of the possible implications such a cyber-theft can cause. Many important questions still remain – could this have also happened before? And if so, who is behind it and what did they use it for?

Our recent research aims to shed more light on this topic, and reveal conclusive evidence that such a leak did actually take place years before the Shadow Brokers leak, resulting in US developed cyber tools reaching the hands of a Chinese group which repurposed them in order to attack US targets.

Key Findings

- The caught-in-the-wild exploit of CVE-2017-0005, a 0-Day attributed by Microsoft to the Chinese APT31 (Zirconium), is in fact a **replica of an Equation Group exploit** code-named "EpMe."

- APT31 had access to EpMe's files, both their 32-bits and 64-bits versions, more than 2 years **before** the Shadow Brokers leak.
- The exploit was replicated by the APT during 2014 to form "Jian", and used since at least 2015, until finally caught and patched in March 2017.
- The APT31 exploit was reported to Microsoft by Lockheed Martin's Computer Incident Response Team, hinting at a possible attack against an American target.
- The framework containing the EpMe exploit is dated to 2013, and contains 4 Windows Privilege Escalation exploits overall, two of which were 0-Days at the time of the framework's development.
- One of the 0-Days in the framework, code-named "EpMo", was never publicly discussed, and was patched by Microsoft with no apparent CVE-ID in May 2017. This was seemingly in response to the Shadow Brokers leak.

Figure 1: Timeline of the events detailing the story of EpMe / Jian / CVE-2017-0005.

Introduction

In the last few months, our malware and vulnerability researchers focused on recent Windows Privilege Escalation exploits attributed to Chinese actors. During this investigation, we managed to unravel the hidden story behind "Jian", a 0-Day exploit that was previously attributed to APT31 (Zirconium), and show its true origins.

In this blog we show that CVE-2017-0005, a Windows Local-Privilege-Escalation (LPE) vulnerability that was attributed to a **Chinese APT**, was replicated based on an **Equation Group** exploit for the same vulnerability that the APT was able to access. "EpMe", the Equation Group exploit for CVE-2017-0005, is one of 4 different LPE exploits included in the DanderSpritz attack framework. EpMe dates back to at least 2013 – four years before APT31 was caught exploiting this vulnerability in the wild.

This isn't the first documented case of a Chinese APT repurposing an Equation Group exploit. In the Bemstour case, discussed by both Symantec and our own research team, the main assumption was that APT3 (Buckeye) sniffed the EternalRomance exploit from network traffic, and later upgraded it to the equivalent of EternalSynergy using an additional APT3 vulnerability. In the present case, however, we have strong evidence that **APT31 had access to the actual exploit files of Equation Group**, in both their 32-bits and 64-bits versions.

In the following sections we introduce the 4 different Windows LPE exploits included in the DanderSpritz framework, and reveal an additional exploit code named "EpMo." This exploit was patched in May 2017, probably as part of the follow-up fixes for the Shadow Brokers "Lost in Translation" leak of Equation Group tools. While the vulnerability was fixed, we failed to identify the associated CVE-ID. To our knowledge, this is the first public mention of the existence of this additional Equation Group vulnerability.

Background

As part of our ongoing research on Windows LPE exploits, and tracking exploit authors, we started analyzing exploits attributed to Chinese APTs. As CVE-2019-0803 was recently mentioned in the NSA list of top 25 vulnerabilities used by Chinese actors, we decided this was a good place to start. After we finished documenting all the information we gathered on this unique exploit, originally a 0-Day attributed to Chinese actors, we went on to the next Chinese-attributed exploit in our list: CVE-2017-0005.

In our review of Microsoft's report on the vulnerability that was caught exploited in the wild and was attributed to Zirconium (APT31), we found a few interesting details:

- The exploit was caught and reported to Microsoft by Lockheed Martin's Computer Incident Response Team.
- The exploit uses a multi-staged packer, which appears identical to the one we saw used by CVE-2019-0803.

Figure 2: Comparison between the packer of CVE-2017-0005 (left) and that of CVE-2019-0803 (right).

Armed with these two leads, and already familiar with the packer used by these exploits, we set out to find the described exploit of CVE-2017-0005.

After we obtained a 64-bit sample of the CVE-2017-0005 exploit, we verified it against the information described by Microsoft in their blog. Not only did it match, when ignoring the random page allocation, both samples use the same addresses (same lower 3 nibbles).

Figure 3: Comparison between Microsoft's sample (left) and ours (right).

Comparing CVE-2017-0005 and CVE-2019-0803

In the following section, we describe in detail some of the characteristics of the packer and the loader used in CVE-2017-0005 and CVE-2019-0803, and highlight their commonalities and differences.

Jian, the exploit of CVE-2017-0005, was shipped in a DLL named `Add.dll`. It contained an interesting PDB path suggesting that it was written in 2015 under a project named "rundll32_getadmin".

F:\\code**2015**\\rundll32_**getadmin****Add**\\x64\\Release\\Add.pdb

When we checked the Time Date Stamp of the binary in the file header, we saw that the DLL was compiled on `Wed May 06 02:08:24 2015`, which fits nicely with the folder name from the PDB. An additional timestamp on the export directory points to the exact same date. Speaking of the export directory, the DLL has a single exported function named "AddByGod", which, as we will learn soon, is the entry function of the packer.

The decryption routine is very straightforward. The packer starts by allocating memory for the encrypted code and copies it to the newly allocated buffer. It then allocates a buffer with `PAGE_EXECUTE_READWRITE` protection to store the decrypted code. After the buffers are allocated, the packer checks if a string argument, which will be used as a decryption key, was passed to the `AddByGod` function. Next, the packer uses the AES256 algorithm with a SHA1 derived key of the passed argument to decrypt the encrypted code. If the decryption is successful, the decrypted code is executed and a second stage payload runs. Luckily, we managed to obtain the password that was needed to execute the binary and decrypt the encrypted payload.

rundll32.exe Add.dll AddByGod [password]

The second stage begins with a typical shellcode technique, searching the module's header for the address of `kernel32.dll` and dynamically retrieving a pointer to the `GetProcAddress` export function. Next, the program

decompresses another Portable Executable (PE) and jumps to its entry point. The decompressed PE, which is the 3rd stage in the loading sequence, has intentionally corrupted headers. It does basic loading operations and then begins with a reflective loading of an embedded executable (yes, another one). The loaded PE is the last stage in the loading sequence and is responsible for executing the exploit.

It's interesting to mention that the compilation time of the embedded binary — the exploit itself — goes back to October 2014. This suggests that the attackers used this 0-day in 2014, almost three years before it became publicly available in the Shadow Brokers leak and was fixed by Microsoft.

Figure 4: The execution flow of the loaders used for CVE-2017-0005 and CVE-2019-0803.

As can be seen in the figure above, the packer used for CVE-2019-0803 is very similar to the one used in CVE-2017-0005. In fact, the flow is almost identical. The file was compiled on September 18, 2018, and is also internally named "Add.dll". Like the previously packed exploit, CVE-2019-0803 also has an export function named "AddByGod" and contains debug information:

C:\Users\sms2056\Desktop**Add**（未修改dll‘）\x64\Release\Add.pdb

Unlike the previous sample, this one uses a different decryption password and needs an additional argument when running (used in later stages). The execution flow then continues exactly as we observed in the previous sample with one exception: after the program decompresses a PE payload and jumps to its entry point, it does not have a 4th stage of another embedded PE, but simply begins the exploitation stage.

We looked for more samples that use this or a similar variant of the packer we described, and found multiple samples and malware families that have used it for many years. All of the malware are clearly and exclusively attributed to Chinese-affiliated attack groups. Adding this conclusion to the contextual information we have, Microsoft's independent attribution of CVE-2017-0005 to a Chinese APT, and NSA's attribution of CVE-2019-

0803 to “Chinese State-Sponsored Actors”, make us believe that the exploit of CVE-2017-0005 was indeed used by attackers that are part of a Chinese group.

Jian – CVE-2017-0005

When analyzing Jian, we noticed the following interesting characteristics.

Operating System (OS) Version Context

The exploit creates a rich version context that includes multiple fields, each representing a different characteristic of the target’s operating system. This extensive context isn’t typical of the Chinese-attributed exploits we previously analyzed and looks like some sort of a utility/framework. This is even more suspicious, as some fields in the context aren’t even initialized (marked in red), and overall only three of them are used by the exploit itself (marked in blue).

 **Figure 5:** Rich version information, as collected by Jian.

Just for comparison, the exploit of CVE-2019-0803 supported only a single Windows version and used the hardcoded version-dependent constants for Windows Server 2008 R2. Alibaba even reported that the tool’s file name was `2008.dll`, leaving no doubt about the tool’s intended target.

Global Configuration Table

The OS version enum is used as an index for a global configuration table. This is a classic example of using such an enum when one needs a version-dependent configuration. The configuration table itself looks like a promising artifact that might appear in additional exploits by the same authors.

We created detection signatures and looked for samples that contain this configuration table. Our search query resulted in the following samples:

- `Mcl_NtElevation_EpMe_GrSa.dll` (x86) – `292fe1fc7d350cc7b970da0f308d22089cd36ab552e5659e3cfb0d9690166628`
- `Mcl_NtElevation_EpMo_GrSa.dll` (x64) – `1537cad1d2c5154e142af775ed555d6168d528bbe40b31f451efa92c9e4f02de`

The naming convention of the files and their context immediately caught us by surprise. We recognized them as part of the Shadow Brokers' "Lost in Translation" leak of Equation Group tools. Equation Group is the name given to an APT group which is believed to be the Tailored Access Operations (TAO) unit of the NSA. How did our search for extremely-unique artifacts extracted from a **Chinese 0-Day exploit** that was patched in **March 2017** show results of leaked **Equation Group tools** from **2013**? To answer this question, we started to dive deep and analyze the information we found.

Lost In Translation

Before we describe our analysis, we want to give a brief history of The Shadow Brokers group and their leaks of Equation Group tools. We believe that understanding the nature of the leak, and especially the timeline, is crucial for understanding what happened next.

The Shadow Brokers is a mystery group of hackers that first appeared on August 12, 2016, when they invited the public to participate in an auction of Equation Group's "Cyber Weapons." Since then, the group started to leak more and more files over a period of several months. One of these leaks, called "Lost in Translation", emerged in April 2017, and is well known for releasing Equation Group's notorious exploits such as Eternal Blue.

One of the main components in this leak is DanderSpritz, Equation Group's post-exploitation framework that contains a wide variety of tools for persistence, reconnaissance, lateral movement, bypassing Antivirus engines, and more. The framework is very modular and provides the operator many capabilities to access victims' computers. During the recent months, we revisited the DanderSpritz framework, reverse-engineered some of its modules and implants, and **plan to publish a detailed publication dedicated to the framework and our findings**.

Our project of profiling exploit authors focuses on Windows LPE exploits, like CVE-2017-0005 whose artifacts we searched. As common in post-exploitation frameworks, DanderSpritz and the Lost In Translation leak also contain LPE exploits, and two of them matched our query.

We now present a brief overview of some of the LPE exploits that were attributed to the Equation Group and their connection to the leaked DanderSpritz framework.

History of Equation Group LPE Exploits

PrivLib and the Houston Disk

The term “PrivLib” is often used when referring to a Privilege Escalation module embedded inside a given Equation Group implant. PrivLib contains a selected set of Windows LPE exploits chosen from the Equation Group arsenal, and all of them are wrapped using a thin wrapper known by its “prkMtx” unique mutex.

In 2015, Kaspersky reported a set of Windows LPEs embedded inside a booby-trapped disk given away at a Houston scientific convention. The exploits, attributed to Equation Group, were all **0-Day** at the time of development, and some even dated as far back as 2008. All in all, the Houston Disk contained a PrivLib version that executes a set of 3 exploits one after the other, until the desired privileges are acquired.

Here is a list of the exploits included in the disk, according to their execution order:

1. MS09-025 (Fanny / Stuxnet)
2. CVE-2013-3128
3. CVE-2011-3402

Note: The CVEs listed here don’t match those mentioned originally by Kaspersky in their report. First, Kaspersky’s researchers weren’t sure about the CVE-IDs to begin with, marking them as “possibly.” Second, we found additional information regarding the latter two exploits, which helped shed light on more probable CVE-IDs for each. More details about the CVE-ID identification can be found later on under the respective sections describing each exploit.

DanderSpritz NtElevation

DanderSpritz is a modular post-exploitation framework that contains dozens of different interdependent modules. For example, some modules do not run unless specific modules are not executed first, and others require special privileges or artifacts to run. Some of the modules require privileges of a SYSTEM account to run. For this to happen, DanderSpritz executes a set of modules named ‘NtElevation’ that are responsible for elevating the privileges of the implant running on the victim’s computer.

Figure 6: An example for the dependencies of the `PasswordDump` module, including `NtElevation`.

These privilege escalation modules are the ones we caught when we queried for Jian’s global configuration table. And they were not alone. We also found a couple of more Local Privilege Escalation exploits from the

NtElevation series.

While the `Eternal*` exploits received a lot of attention, and rightly so, mentions of the NtElevation exploits were somehow missing. We couldn't find any online reference that points to the existence of the NtElevation module as part of the Equation Group arsenal or even as part of the "Shadow Brokers Exploits", nor any reference to the following 4 exploit code names.

- **ElEi**
- **ErNi**
- **EpMo**
- **EpMe**

Note: Equation Group's exploits are known to have code names that are abbreviated using 4 letters. For example, Eternal Blue and Eternal Romance are internally referred to as `ETBL` and `ETRO`. Similarly, the Local Privilege Escalation exploits we discussed have their own code names, as listed above.

Despite our attempts, we couldn't manage to trace back the full code names for these exploits. However, the naming convention suggests that `EpMo` and `EpMe` are of the same type or that they exploit vulnerabilities in the same module, just like the `Eternal*` exploits (EternalBlue, EternalRomance, etc). This conclusion does make sense as our single search query found both of these exploits.

When we analyzed the DanderSpritz NtElevation API, we found the checks that each module deploys to declare that the exploit is indeed supported. When combined with the original patch dates Kaspersky estimated for the two font exploits from the Houston Disk, this new information helped us make a better estimate of the inner workings of each CVE-ID.

We thoroughly analyzed the found exploits and tried to match each exploit file to its respective CVE-ID. These are the results of our analysis and our conclusions.

ElEi – CVE-2011-3402

Houston Disk: 3rd in the execution order.
Supported DanderSpritz Windows Versions: Windows 2000 to Windows 7, inclusive.

The exploit also contains an additional check that `win32k.sys` is dated to **before** November 23, 2011. This is a clear indication of CVE-2011-3402, which is the only font vulnerability that was fixed in December 2011. The gap in the dates is explained by the fact that Microsoft compiled the patched driver on the mentioned date.

We are aware that CVE-2011-3402 was originally spotted as a 0-Day that was exploited in the wild, and was found in Duqu (1.0). Currently, we can only point out this interesting CVE-ID match, but we have not yet studied it further or compared the two exploits, as these font exploits are outside of the scope of our research, which focuses on the unknown DanderSpritz exploits and their connection to CVE-2017-0005.

We do recommend this lead for a future work publication and invite security researchers worldwide to examine this connection.

ErNi – CVE-2013-3128

Houston Disk: 2nd in the execution order.
Supported DanderSpritz Windows Versions: Only Windows 2000.

The exploit also contains an additional check that `ATMFD.dll` is of the exact version “5.0.2.227”. As the Houston Disk exploit supported additional versions, we aren’t fully sure why the version range was narrowed down in DanderSpritz. Compared to `ElEi`, there is no indicative patch check, which may be because the DanderSpritz files are dated to mid-2013, which is prior to the patch that was identified by Kaspersky and is dated to October 2013.

We chose CVE-2013-3128 instead of CVE-2013-3894 because this vulnerability is an OpenType Font vulnerability, which correlates with the exploit at hand. This identification should be taken with a grain of salt as none of these CVE-IDs were actually marked as “exploited in the wild.” The reason we chose this CVE-ID is merely because it is mentioned in the Patch Tuesday cited by Kaspersky. As with `ElEi`, further study of these font exploits is more than welcome.

User Mode Print Driver (UMPD) 101

Basic analysis of both `EpMo` and `EpMe` found them to be GDI User-Mode-Print-Driver (UMPD) related, which explains why we found them when searching for a GDI UMPD related artifact from Jian. Before diving into the exploits, we first provide some background on what exactly is a User-Mode-Print-Driver.

The Windows operating system supports the option of rendering most of the needed graphics for a given print job in user-mode, in contrast to the traditional implementation of such drivers inside the Windows kernel. The architecture of deploying such a User-Mode-Print-Driver (UMPD) is shown below.

Figure 7: UMPD architecture, based on figures from this Black-Hat Europe talk.

Supporting such a data flow dictates that the kernel is aware of the user's UMPD device and can forward it a set of requests, depending on the types that the driver declared to support. As is explained in more detail in this excellent Black Hat Europe 2020 talk focusing on UMPD, allowing for user-mode callbacks, invoked from the kernel, is a sure recipe for security vulnerabilities.

In the next few sections, we explain in detail how each Equation Group exploit uses the UMPD, and which vulnerabilities in this mechanism were exploited.

EpMo – Analysis

Houston Disk: N/A.
Supported DanderSpritz Windows Versions: Windows 2000 to Windows Server 2008 R2, inclusive.

Root Cause

After we finished reverse engineering the rich context and utilities exposed as part of the DanderSpritz framework and API, the vulnerability itself was quite simple. A short analysis revealed that this is probably a NULL-Deref vulnerability, as the NULL page is allocated, and the shellcode is immediately copied to it, as can be seen below:

Figure 8: Preparation of the NULL-page, as part of the EpMo exploit.

As this is a NULL-Deref vulnerability, we can immediately rule out CVE-2017-0005, as the stack trace shown in Microsoft's blog has nothing to do with the NULL page. This means that this is possibly another vulnerability found and exploited by Equation Group in 2013. With that out of the way, it is time to understand what triggers this NULL-Deref vulnerability.

Our first hint as to the identity of the affected module, which we expect to be the UMPD, can be found in this classic example of the use of the OS version enum field:

Figure 9: Using OS version enum to query for the `ppClientPrinterThunk` callback index.

After the version-dependent index of the callback is fetched, the callback itself is replaced with the attacker's fake `ClientPrinterThunk` callback.

Figure 10: Replacing the `KernelCallbackEntry` with the attacker's `ClientPrinterThunk`.

Bingo! The exploit indeed makes use of a fake `ClientPrinterThunk`. Let's dig in and analyze the exploit logic inside this fake callback.

The callback itself is a thin wrapper that forwards the `gdi_ctx` and the original argument to a function that is very similar to Windows's own `GdiPrinterThunk`. As a matter of fact, the code of the exploit is very modular, and each supported driver command is handled by its own virtual handler implemented in the `gdi_ctx` class. Aside from the chosen set of implemented handlers, there is no real logic in this function.

Figure 11: Driver command types that are handled by the exploit's user-mode driver.

While analyzing this function, we stumbled upon the GDI configuration array that originally pointed us to this exploit sample. Now, placed in the right context, we can easily deduce the role of this configuration array. It holds the print driver's `INDEX_LAST` value for each version of the target operating system.

Figure 12: Global `INDEX_LAST` configuration table, as used by the driver.

As can be seen above, this configuration value is crucial for the driver's logic as it represents the total number of dispatched functions that should be handled by the driver.

Now that we understood the overall flow of the exploit, and the structure of the User-Mode-Print-Driver (UMPD), tracing the root cause of the vulnerability proved to be a relatively simple task. The driver implemented special handlers only for the following basic command types:

Figure 13: List of driver-supported command types.

On top of these commands, there is one additional supported command, with the os-dependent value of `INDEX_LAST + 4`:

Figure 14: Handler for the `DrvFN` command type.

In this command, we initialize an array that tells the operating system which function handlers we support. The attackers chose to mark all of the functions as "supported" except for 3 specific function handlers:

- `DrvStartDoc` (0x23)
- `DrvEnableSurface` **(0x03)**
- `DrvDisableSurface` (0x04)

Please pay close attention to `DrvEnableSurface`. The syscall that triggers the vulnerability is `NtGdiStartDoc`, which is responsible for starting the print job. However, to do so, the vulnerable function `win32k!PDEVOBJ::bMakeSurface()` is invoked and tries to create a surface, exactly the operation that isn't supported by our driver. Here is a debugging output from the vulnerable function:

Figure 15: Debugger output inside the vulnerable function showing the missing handler entry.

While the entry for `DrvDisablePDEV` (0x02) exists, and points to the correct Windows function, the adjacent entry for `DrvEnableSurface` (0x03) contains only zeros.

From this point, the vulnerability is clear. The vulnerable function assumes that our driver supports this handler, fetches the empty entry from the struct, and invokes NULL as the function responsible for enabling the surface. The figure below shows the full stack trace, right at the point in which the control flow is passed to the shellcode that the attacker stored at address 0x0:

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

00 8aeb8c6c 8fa42330 0x0

01 8aeb8c88 8fbae3f6 win32k!PDEVOBJ::bMakeSurface+0x43

02 8aeb8cb0 8fbae94c win32k!GreStartDocInternal+0x7e

03 8aeb8d1c 82a4f42a win32k!NtGdiStartDoc+0x2ff

04 8aeb8d1c 000f0813 (T) nt!KiFastCallEntry+0x12a

05 0022eed0 10008118 (T) 0xf0813

06 0022ef18 10006f53 Mcl_NtElevation_EpMo_GrSa+0x8118

07 0022ef28 10006fc3 Mcl_NtElevation_EpMo_GrSa+0x6f53

08 0022ef44 10007af4 Mcl_NtElevation_EpMo_GrSa+0x6fc3

09 0022ef74 10006ce5 Mcl_NtElevation_EpMo_GrSa+0x7af4

0a 0022efcc 10004a31 Mcl_NtElevation_EpMo_GrSa+0x6ce5

0b 0022f324 100038c0 Mcl_NtElevation_EpMo_GrSa+0x4a31

0c 0022f338 10003a7b Mcl_NtElevation_EpMo_GrSa+0x38c0

0d 0022f370 10002adf Mcl_NtElevation_EpMo_GrSa+0x3a7b

0e 0022f3d8 77d5af24 Mcl_NtElevation_EpMo_GrSa+0x2adf

00 8aeb8c6c 8fa42330 0x0 01 8aeb8c88 8fbae3f6 win32k!PDEVOBJ::bMakeSurface+0x43 02 8aeb8cb0 8fbae94c win32k!GreStartDocInternal+0x7e 03 8aeb8d1c 82a4f42a win32k!NtGdiStartDoc+0x2ff 04 8aeb8d1c 000f0813 (T) nt!KiFastCallEntry+0x12a 05 0022eed0 10008118 (T) 0xf0813 06 0022ef18 10006f53 Mcl_NtElevation_EpMo_GrSa+0x8118 07 0022ef28 10006fc3 Mcl_NtElevation_EpMo_GrSa+0x6f53 08 0022ef44 10007af4 Mcl_NtElevation_EpMo_GrSa+0x6fc3 09 0022ef74 10006ce5 Mcl_NtElevation_EpMo_GrSa+0x7af4 0a 0022efcc 10004a31 Mcl_NtElevation_EpMo_GrSa+0x6ce5 0b 0022f324 100038c0 Mcl_NtElevation_EpMo_GrSa+0x4a31 0c 0022f338 10003a7b Mcl_NtElevation_EpMo_GrSa+0x38c0 0d 0022f370 10002adf Mcl_NtElevation_EpMo_GrSa+0x3a7b 0e 0022f3d8 77d5af24 Mcl_NtElevation_EpMo_GrSa+0x2adf

```
00 8aeb8c6c 8fa42330 0x0
01 8aeb8c88 8fbae3f6 win32k!PDEVOBJ::bMakeSurface+0x43
02 8aeb8cb0 8fbae94c win32k!GreStartDocInternal+0x7e
03 8aeb8d1c 82a4f42a win32k!NtGdiStartDoc+0x2ff
04 8aeb8d1c 000f0813 (T) nt!KiFastCallEntry+0x12a
05 0022eed0 10008118 (T) 0xf0813
06 0022ef18 10006f53 Mcl_NtElevation_EpMo_GrSa+0x8118
07 0022ef28 10006fc3 Mcl_NtElevation_EpMo_GrSa+0x6f53
08 0022ef44 10007af4 Mcl_NtElevation_EpMo_GrSa+0x6fc3
09 0022ef74 10006ce5 Mcl_NtElevation_EpMo_GrSa+0x7af4
0a 0022efcc 10004a31 Mcl_NtElevation_EpMo_GrSa+0x6ce5
0b 0022f324 100038c0 Mcl_NtElevation_EpMo_GrSa+0x4a31
0c 0022f338 10003a7b Mcl_NtElevation_EpMo_GrSa+0x38c0
0d 0022f370 10002adf Mcl_NtElevation_EpMo_GrSa+0x3a7b
0e 0022f3d8 77d5af24 Mcl_NtElevation_EpMo_GrSa+0x2adf
```

The Patch

Microsoft fixed this vulnerability in May 2017, one month after the Shadow Brokers's Lost in Translation leak. However, we failed to find a CVE-ID that refers to this fix. The patch itself addresses the exact flaw in the

vulnerable function `win32k!PDEVOBJ::bMakeSurface()`. It adds a sanity check after the handler is fetched from the struct, and before it is invoked by the function. If the entry is NULL, the function aborts.

Figure 16: Patched version now checks the function handler before invoking it.

To conclude, the `EpMo` Equation Group exploit is a NULL-Deref in GDI's UMPD module, and is therefore **not** an exploit for CVE-2017-0005. It was patched by Microsoft in May 2017, and we couldn't find a clear CVE-ID associated with it.

Now that we better understand the framework's API, and have a better understanding of the UMPD module, it is time to focus on the next exploit – `EpMe`.

Just to recap, we found both of these exploits when searching for an artifact from Jian, APT31's exploit for CVE-2017-0005. As `EpMo` is a different vulnerability that originates from the same module, our hope was that `EpMe` does indeed exploit CVE-2017-0005. Time to analyze it and find out.

EpMe – Analysis

Houston Disk: N/A.
Supported DanderSpritz Windows Versions: Windows XP to Windows 8, inclusive.

Root Cause

Armed with our newly acquired knowledge about the UMPD module, we started analyzing the EpMe exploit. The exploit itself shares a lot of code with EpMo, thus allowing us to easily focus on the exploit-specific logic that is unique to EpMe. While the initialization phase of this exploit is longer and involves a lot of GDI-related bootstrapping (`DrawStream()`, finding the wanted Display, etc.), the actual flow that hijacks the control flow is relatively simple.

After the bootstrap is finished, the exploit triggers a call to the `NtGdiBitBlt` syscall. This initiates a chain of events in the Windows kernel and eventually passes the flow back to our user-mode callback (`DrvBitBlt()`) registered by our UMPD. Here lies the heart of the exploit.

Our function allocates a new `Rbrush` using `NtGdiBRUSHOBJ_pvAllocRbrush()`, whose sole purpose is to allow UMPD implementations to allocate themselves an `Rbrush` and couple it with a `BRUSHOBJ`. As a direct result, it also means that the `Rbrush` is allocated in **user-mode**, using `EngAllocUserMemEx()`. Storing it in user-mode means that we can access it and craft the struct's content. And so, the attackers corrupted the `Rbrush` to point at a set of fake GDI objects that were forged on a local stack buffer inside the callback.

To hijack the control flow, the attackers chose to use a Palette and crafted it so that `PALETTE.pfnGetNearestFromPalentry` points to their shellcode, exactly as Microsoft pointed out in their blog on the caught-in-the-wild exploit. After everything is built, the callback invokes the `NtGdiEngBitBlt` syscall with a `rop4` parameter of `0xCCAA`. This specific syscall was chosen because of two key features:

- The user passes to it a `BRUSHOBJ`.
- A `rop4` value of `0xCCAA` means the kernel directly accesses the user-controlled `Rbrush` from within the supplied `BRUSHOBJ`.

More specifically, a Stream is extracted from the fully-controlled `Rbrush` and is forwarded on to `EngDrawStream()`, causing the unsuspecting kernel function to use our fully crafted Stream.

Figure 17: Control flow of the EpMe exploit of CVE-2017-0005.

This chain of functions gradually uses all of the GDI objects that we've crafted in our user-mode callback, eventually leading to `XLATEOBJ_iXlate()`. This last function invokes our crafted `PALETTE.pfnGetNearestFromPalentry` function pointer, thus hijacking the control flow and triggering the execution of our shellcode.

To summarize, the root cause for this vulnerability is based on the complex design involved in supporting UMPD, and the need to allocate objects for it in user-mode. The vulnerability itself lies inside `EngBitBlt()`, which blindly trusts and directly uses our crafted `Rbrush` and the set of fake GDI objects it points to. Not only does this vulnerability give an attacker a powerful exploit primitive, but it also points at a design issue in the Windows kernel. As long as there is a function somewhere in the kernel that directly accesses a user-supplied `Rbrush`, it also blindly trusts all the values that it points to and that are fully controlled by the user.

The Patch – CVE-2017-0005

Another important conclusion we drew from analyzing the exploited vulnerability is that we now know for sure that **EpMe exploits CVE-2017-0005**. On top of our analysis of both the Equation Group and APT31 exploits, the EpMe exploit aligns perfectly with the details reported in Microsoft's blog on CVE-2017-0005. And if that wasn't enough, the exploit indeed stopped working after Microsoft's March 2017 patch, the patch that addressed the said vulnerability.

The patch itself is rather straightforward: `EngBitBlt()` with a `rop4` value of `0xCCAA` no longer supports the option to draw a Stream, an action that demands extracting a Stream from the user-supplied `Rbrush`. By removing this feature altogether, Microsoft completely eliminated the vulnerable code flow.

Before:

Figure 18: Vulnerable `win32k.sys`, supports both `EngDrawStream()` and `EngTransparentBlt()`.

After:

Figure 19: Patched `win32k.sys`, no longer supports `EngDrawStream()`.

It is important to remember that two APTs exploiting the same vulnerability (CVE-2017-0005) could just be a coincidence. When this happens to security researchers, such a case is often referred to as a "bug-collision." It's possible that researchers on both sides found this vulnerability independently, and it doesn't necessarily mean that there is a real connection between the tools.

We now compare the two exploit samples, Jian and EpMe, and see if we can spot any connection between them aside from them exploiting the same vulnerability.

EpMe vs Jian

Similar Version Context

At the beginning of our research, we saw that Jian uses a context that holds multiple fields about the version of the target's operating system. Used fields are marked in blue, and uninitialized fields are marked in red.

 **Figure 20:** Rich version information, as collected by Jian.

Now, when we review the version context used by all exploits shared by the DanderSpritz framework, we can see the following, very similar, structure:

Figure 21: Version context used by DanderSpritz and shared by all Equation Group exploits.

The fields that are marked in red in Jian were marked again in the sample from the Equation Group exploits. As can be seen, one field is still unused in all 4 DanderSpritz exploits, but the other field is heavily used and holds the handle for the mapped version of NTOS kernel. It is hard to miss the wide similarity between the two structures, up to the **order and size of the first 9 fields**, even including the size of the unused field in between.

The changes between the two configuration structures are that Equation Group's configuration contains more fields, mostly used for security mitigation policies, that are relevant for Windows 8 systems and higher. The last difference between the structures is in the field specifying the architecture of the target's kernel, which for some reason was negated in Jian. Anyway, this field was never used by the exploit.

Overall, having a Chinese-attributed exploit use a version context is uncommon. Having one that is nearly identical to the version context used by the entire DanderSpritz NtElevation module can't be a coincidence.

Same Memory Layout

At the heart of the exploits lies a single function that populates a buffer with the various fake GDI objects, which is pointed to by our user-mode `Rbrush` object. Not only do both exploits use a single function for the

construction of all of these fake objects, but the memory layout of the objects in the argument buffer is also **identical**.

 **Figure 22:**

Construction of the fake GDI objects, as done in Jian.

When we analyzed the code of the Equation Group exploit, we used it to recreate a source code Proof-Of-Concept (POC). The result is the following beautified and labeled code:

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

```
void populate_buffer_and_brush(char * pBuffer, HBRUSH hbrush)

{

memset(pBuffer, 0, 0x200);
```

```
memset(&pBuffer[0x200], 0, 0xA8);

memset(&pBuffer[0x2A8], 0, 0x30);

memset(&pBuffer[0x2D8], 0, 0x10);

// 0x00: PALETTE

*(DWORD *)( pBuffer + 0x18) = 2; // 0x14 - 0x1C: flPal

*(size_t *)(pBuffer + 0x80) = pBuffer + 0x2A8; // 0x80 - 0x88: apalColor

// Brush (DRAWSTREAMINFO)

size_t * pBrush = (size_t*)hbrush;

pBrush[3] = pBuffer + 0x29C; // pptlDstOffset - Pointer to 0

pBrush[4] = pBuffer + 0x208; // pxloSrcToBGRA

pBrush[5] = pBuffer + 0x208; // pxloDstToBGRA

pBrush[6] = pBuffer + 0x208; // pxloBGRAToDst <-- We use this one (offset 0x30)

pBrush[7] = 60; // ulStreamLength - 60

pBrush[8] = pBuffer + 0x260; // pvStream - Pointer to our built "Stream"

// Second Struct

*(size_t *)(pBuffer + 0x200) = hbrush;

// 0x208: _XLATE

*(size_t *)(pBuffer + 0x230) = pBuffer; // ppalSrc

*(size_t *)(pBuffer + 0x238) = pBuffer; // ppalDst <-- We use this one (offset 0x30)

*(size_t *)(pBuffer + 0x240) = pBuffer; // ppalDstDC

// 0x260 - 0x2A8: Our "Stream"

*(DWORD *)(pBuffer + 0x260) = 9; // DS_NINEGRIDID (ulCmdID)

*(DWORD *)(pBuffer + 0x26C) = 1; // rclDst.right = 0x01

*(DWORD *)(pBuffer + 0x270) = 1; // rclDst.bottom = 0x01

*(DWORD *)(pBuffer + 0x27C) = 80; // rclSrc.right = 0x50

*(DWORD *)(pBuffer + 0x280) = 80; // rclSrc.bottom = 0x50
```

*(DWORD *)(pBuffer + 0x284) = 4; // ngi.flFlags := DSDNG_PERPIXELALPHA (4)

// 0x2A8: Palette Color Table

*(DWORD *)(pBuffer + 0x2CC) = 100;

// Fourth Struct

*(size_t *)(pBuffer + 0x2D8) = AllocMemoryPage(0x10000);

*(size_t *)(pBuffer + 0x2E0) = g_pRtlCopyUnicodeString;

}

void populate_buffer_and_brush(char * pBuffer, HBRUSH hbrush) { memset(pBuffer, 0, 0x200); memset(&pBuffer[0x200], 0, 0xA8); memset(&pBuffer[0x2A8], 0, 0x30); memset(&pBuffer[0x2D8], 0, 0x10); // 0x00: PALETTE *(DWORD *)(pBuffer + 0x18) = 2; // 0x14 - 0x1C: flPal *(size_t *)(pBuffer + 0x80) = pBuffer + 0x2A8; // 0x80 - 0x88: apalColor // Brush (DRAWSTREAMINFO) size_t * pBrush = (size_t*)hbrush; pBrush[3] = pBuffer + 0x29C; // pptlDstOffset - Pointer to 0 pBrush[4] = pBuffer + 0x208; // pxloSrcToBGRA pBrush[5] = pBuffer + 0x208; // pxloDstToBGRA pBrush[6] = pBuffer + 0x208; // pxloBGRAToDst <-- We use this one (offset 0x30) pBrush[7] = 60; // ulStreamLength - 60 pBrush[8] = pBuffer + 0x260; // pvStream - Pointer to our built "Stream" // Second Struct *(size_t *)(pBuffer + 0x200) = hbrush; // 0x208: _XLATE *(size_t *)(pBuffer + 0x230) = pBuffer; // ppalSrc *(size_t *)(pBuffer + 0x238) = pBuffer; // ppalDst <-- We use this one (offset 0x30) *(size_t *)(pBuffer + 0x240) = pBuffer; // ppalDstDC // 0x260 - 0x2A8: Our "Stream" *(DWORD *)(pBuffer + 0x260) = 9; // DS_NINEGRIDID (ulCmdID) *(DWORD *)(pBuffer + 0x26C) = 1; // rclDst.right = 0x01 *(DWORD *)(pBuffer + 0x270) = 1; // rclDst.bottom = 0x01 *(DWORD *)(pBuffer + 0x27C) = 80; // rclSrc.right = 0x50 *(DWORD *)(pBuffer + 0x280) = 80; // rclSrc.bottom = 0x50 *(DWORD *)(pBuffer + 0x284) = 4; // ngi.flFlags := DSDNG_PERPIXELALPHA (4) // 0x2A8: Palette Color Table *(DWORD *)(pBuffer + 0x2CC) = 100; // Fourth Struct *(size_t *)(pBuffer + 0x2D8) = AllocMemoryPage(0x10000); *(size_t *)(pBuffer + 0x2E0) = g_pRtlCopyUnicodeString; }

```
void populate_buffer_and_brush(char * pBuffer, HBRUSH hbrush)
{
    memset(pBuffer, 0, 0x200);
    memset(&pBuffer[0x200], 0, 0xA8);
    memset(&pBuffer[0x2A8], 0, 0x30);
    memset(&pBuffer[0x2D8], 0, 0x10);
    // 0x00: PALETTE
    *(DWORD *)( pBuffer + 0x18) = 2;               // 0x14 - 0x1C: flPal
    *(size_t *)(pBuffer + 0x80) = pBuffer + 0x2A8; // 0x80 - 0x88: apalColor
    // Brush (DRAWSTREAMINFO)
    size_t * pBrush = (size_t*)hbrush;
    pBrush[3] = pBuffer + 0x29C; // pptlDstOffset  - Pointer to 0
    pBrush[4] = pBuffer + 0x208; // pxloSrcToBGRA
    pBrush[5] = pBuffer + 0x208; // pxloDstToBGRA
    pBrush[6] = pBuffer + 0x208; // pxloBGRAToDst  <-- We use this one (offset 0x30)
```

```
    pBrush[7] = 60;              // ulStreamLength - 60
    pBrush[8] = pBuffer + 0x260; // pvStream       - Pointer to our built "Stream"
    // Second Struct
    *(size_t *)(pBuffer + 0x200) = hbrush;
    // 0x208: _XLATE
    *(size_t *)(pBuffer + 0x230) = pBuffer; // ppalSrc
    *(size_t *)(pBuffer + 0x238) = pBuffer; // ppalDst <-- We use this one (offset 0x30)
    *(size_t *)(pBuffer + 0x240) = pBuffer; // ppalDstDC
    // 0x260 - 0x2A8: Our "Stream"
    *(DWORD *)(pBuffer + 0x260) = 9;  // DS_NINEGRIDID (ulCmdID)
    *(DWORD *)(pBuffer + 0x26C) = 1;  // rclDst.right  = 0x01
    *(DWORD *)(pBuffer + 0x270) = 1;  // rclDst.bottom = 0x01
    *(DWORD *)(pBuffer + 0x27C) = 80; // rclSrc.right  = 0x50
    *(DWORD *)(pBuffer + 0x280) = 80; // rclSrc.bottom = 0x50
    *(DWORD *)(pBuffer + 0x284) = 4;  // ngi.flFlags  := DSDNG_PERPIXELALPHA (4)
    // 0x2A8: Palette Color Table
    *(DWORD *)(pBuffer + 0x2CC) = 100;
    // Fourth Struct
    *(size_t *)(pBuffer + 0x2D8) = AllocMemoryPage(0x10000);
    *(size_t *)(pBuffer + 0x2E0) = g_pRtlCopyUnicodeString;
}
```

Aside from the added struct at the end of the buffer, which uses `RtlCopyUnicodeString`, the memory layout of the objects inside the argument buffer was **completely identical**.

As we also labeled the different objects, we can see that the important part is the references from one object to another, and not the location in which they are stored in the buffer itself. And yet, as if by magic, both exploits share this memory layout.

Shared Constants

One more advantage of our recreated code POC for EpMe is that it enabled us to play around with various constants used throughout the exploit, such as:

- GUI Window Name – Originally “h”.
- Point locations – One of which was originally (100, 100).
- Print Job ID – Originally 5.
- Driver Name / Document Name – A weird Unicode string is shown below.

Figure 23: Weird Unicode string, later used as both the driver and document names.

Needless to say, none of the above were related to the vulnerability itself, and changing them didn't affect the exploit at all. They are simply hardcoded constants chosen by the original developers of the exploit.

The interesting thing is, both **EpMe and the Jian use the exact same hardcoded constants.** The fact that all of these constants are shared between the two samples, even the weird looking Unicode string above, just shows that one of the exploits was most probably copied from the other.

Figure 24: Jian containing the same constants as the Equation Group exploit.

It is also possible that both parties were inspired by some unknown 3rd-party implementation that used all of these constants. Alas, we failed to find any evidence for the existence of such a module. We must say that the odds of this scenario are rather slim, especially when taking into account the weird-looking Unicode string.

Comparison Conclusion

The meaning of all of this is pretty simple:

- One APT found the vulnerability and developed an exploit for it.
- Another APT caught it and replicated it for their own use.

While both of them deserve full credit for these remarkable achievements, we still want to find out who copied from whom. Time to attribute the exploit.

Exploit Attribution – Who was the original developer?

Weird Looking Unicode String

To the eyes of a Western researcher, the Unicode string used for both the print driver name and the name of the printed document looks foreign. And indeed, we can surely say that the string “屁焚“ doesn’t look like an obvious choice for native English speakers.

We therefore consulted with colleagues around the world who are fluent in Chinese, Korean and Japanese, and asked for their opinion about these two symbols. The unanimous answer we received declared that there is no language in which these symbols make sense. In each language, only one symbol has a meaning, and in any case **doesn’t make sense** as part of a two-symbol phrase. We also checked for a meaning for the symbols created from inverting the order of the original bytes, and the result was the same.

So this is probably not a Chinese phrase used by the original developers of the exploit, but what is it?

Our main hypothesis is based on the op-sec of the developers. Aside from names of DLL files and imported functions, it is very rare to find any string inside Equation Group samples. Even the name of the created window is only “h”, not exactly a long string that could be used by YARA rules so as to “sign” the binary. Faced with the need to use a Unicode string that is surely longer than a single ASCII char string, we believe that the developers chose to use a pattern that matches their needs:

- Unicode-enough for the Windows API.
- Not a real string that might be traced by security researchers / solutions.

If we look closely at the chosen Unicode string, we can see that it actually makes more sense as an x64 assembly snippet:

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

41 5C pop r12

5F pop rdi

C3 retn

41 5C pop r12 5F pop rdi C3 retn

```
41 5C               pop     r12
5F                  pop     rdi
C3                  retn
```

As a matter of fact, this byte sequence is actually a very popular assembly snippet, found almost 150 times just in `ntdll.dll`. Choosing such a popular assembly sequence for a “Unicode string” achieves all of the stated goals.

Finally, the string can also be randomly generated and lacks any meaning whatsoever.

Window Name – “h”

As we just mentioned in the previous section, the GUI Window name used in both of the exploits is “h”. What may seem like a randomly selected short string actually has quite a history behind it. Since the earliest PrivLib version we managed to find, dated to 2008, all of the Equation Group exploits that we’ve analyzed used the exact same string when a window name was needed. And this string was always “h”.

As a matter of fact, all of the 3 exploits included in the Houston Disk used the exact same global string when creating their window:

Figure 25: Global string used as the window name in all Houston Disk exploits.

This is one small indication that the original authors of the exploits could indeed have been Equation Group. However, as this artifact could also be just a coincidence, we now review additional aspects of the exploits.

Quirks in Jian

Windows 2000 support

Close examination of the vulnerable function shows us that Windows 2000 was never vulnerable, as can be seen below:

Figure 26: `win32k.sys` from Windows 2000, just before the version reached End-of-Life.

Despite Windows 2000 not being vulnerable, the UMPD code in Jian has special cases for Windows 2000, and Windows 2000 is part of the OS Version enum.

Figure 27: Jian's logic for supporting Windows 2000, inside the UMPD module.

The interesting issue is that according to the exploit configurations of Equation Group, **EpMe doesn't support Windows 2000**. The minimal supported version is Windows XP, which aligns perfectly with the vulnerable Windows versions.

The fact that Equation Group built proper frameworks means that the UMPD module was shared between EpMe and EpMo. The variation between the exploits is based on exploit-specific logic that was implemented inside virtual handlers that are invoked by the generic UMPD module. Because EpMo supports Windows 2000, so does the UMPD module, which explains why EpMe might **seem to support** this version of Windows.

Figure 28: Support for Windows 2000 in the shared UMPD module of EpMo and EpMe.

If we assume for a moment that APT31 was the original developer of the exploit for CVE-2017-0005, why would they even attempt to add support for Windows 2000? Windows 2000 was never vulnerable in the first place. To be clear, we have no indication that the actors had their own version of the EpMo exploit or anything similar, meaning we have no indication they ever needed such Windows 2000 support for any other tool / exploit.

A far more probable scenario is that APT31 copied the exploit from Equation Group. It is likely that the threat group's developers probably didn't fully understand the limitations of the exploit, and so left the Windows 2000 specific code untouched. An old relic inherited from the EpMo exploit of which APT31 wasn't even aware of or care about.

Enum value rotation

For some unknown reason, Jian contains the syscall definitions for a 32-bit exploit, on top of those needed for the 64-bit exploit. While they aren't used in practice, as the sample is 64-bit, they still give us a glimpse of how their 32-bit exploit would have looked.

We can see that Jian's 32-bit logic for each syscall ID once again matches that of the Equation Group sample, up to the level of adjusting some IDs based on the refined Service Pack Major Number.

Figure 29: Jian's logic for configuring 32-bit syscalls, using a refined Service Pack value.

The problem is, if we check the actual syscall numbers for Windows XP Service Pack 0 and Service Pack 1, we can see that the condition for setting the value of `SP_delta` is flawed. It was correct for the Equation Group exploit, but is not correct here as APT31 modified the `wSpMajor_refined` value during the exploit's initialization.

Figure 30: Jian's rotation of almost all `wSpMajor` values.

This value update is even stranger, as the Equation Group exploits have no mention of it whatsoever:

Figure 31: Equation Group exploit setting the value of `wSpMajor_refined`.

You might ask, "Why would someone perform the above value rotation?" The answer is actually rather simple. From our past analyses of several exploits attributed to Chinese-affiliated actors, we saw that the developers have a habit of using the value 0 as a marker for "illegal value." This can be clearly seen as all Service Pack values above 6 are mapped back to the value 0, which marks them as "illegal." This was also the case for the OS Version Enum, which was fully incremented by 1, making Windows NT use a value of 1 instead of 0, and reserving the sacred value of 0 to mark an error state.

And yet, this time the developers used only a **partial** rotation for the Service Pack value, causing a collision with the legitimate value for Service Pack 0 which for some reason they didn't remap to "1". This means that 0 is simultaneously an illegal value that shouldn't be supported, and a legitimate value that is crucial for configuring

the correct syscall numbers. The correct adjustment should probably have been to increment all values by 1, map the illegal values to 0, and adjust the syscall check to `wSpMajor_refined != 1`.

Once again, we see a weird pattern. Even under the premise that Chinese exploits should reserve the value 0 for illegal values, the code still looks odd. A developer writing this exploit from scratch would probably have just incremented the `wSpMajor_refined` value by 1, while remembering that in future checks Service Pack 0 is marked with the value 1. Instead, as if not to break an existing piece of code, the syscall initialization still checks for 0, and this value is simultaneously both valid and illegal at the same time.

A more probable explanation is that the original code was the Equation Group version, and that the developers affiliated with Chinese attack groups were afraid to break it, thus going only half-way in remapping the values. This fear of breaking the code also reflects on their poor understanding of the overall exploit.

Debug string in the trigger function

Jian contains a debug string "*int the overflow!!!*" that can be found inside UMPD's `DrvBitBlt()`, the callback responsible for triggering the vulnerability.

Figure 32: APT31 debug string inside the trigger function.

As we already established, the exploited vulnerability is not an "overflow" vulnerability. While the string may hint at either a "buffer overflow" or an "integer overflow", none of them have any connection to the user-mode callback design issue that was actually exploited.

While it may just be a language barrier issue, this is yet another possible clue that the attackers behind Jian didn't properly understand the true nature of the exploited vulnerability.

Attribution Conclusion

Together with additional artifacts that match Equation Group artifacts and habits shared between all exploits even as far back as 2008, we can safely conclude the following:

- Equation Group's EpMe exploit, existing since at least 2013, is the **original exploit** for the vulnerability later labeled CVE-2017-0005.
- Somewhere around 2014, APT31 managed to capture both the 32-bit and 64-bit samples of the EpMe Equation Group exploit.
- They replicated them to construct "Jian", and used this new version of the exploit alongside their unique multi-staged packer.

- Jian was caught by Lockheed Martin's IRT and reported to Microsoft, which patched the vulnerability in March 2017 and labeled it CVE-2017-0005.

Timeline

Below is a timeline of the events surrounding both exploit versions of what began as EpMe (Equation Group) and was eventually patched by Microsoft as CVE-2017-0005 (APT31).

Figure 33: Timeline of the events detailing the story of EpMe / Jian / CVE-2017-0005.

In greater detail:

- 2008/2009 – Early Equation Group exploit tools: PrivLib / Houston Disk.
- 2013 – DanderSpritz NtElevation exploits: ElEi, ErNi, EpMo, EpMe.
- October 27, 2014 – Early timestamp from the embedded PE – cloned EpMe.
- May 6, 2015 – Multiple indicators that the complete APT31 tool was compiled in 2015.
- August 13, 2016 – Initial Shadow Brokers publication.
- January 8, 2017 – Shadow Brokers leak a directory structure of their files, clearly indicating the possession of DanderSpritz and Eternal* exploits.
- February 14, 2017 – Patch Tuesday is **cancelled**, merged with March's fixes.
- March 14, 2017 – Patch Tuesday – Fixed CVE-2017-0005 and 1st round of critical Equation Group exploits included in the to be published "Lost in Translation" SB leak.
- March 27, 2017 – Microsoft publishes the blog on CVE-2017-0005 that was reported by Lockheed Martin, and attributed it to a Chinese APT (Zirconium / APT31).
- April 14, 2017 – Lost in Translation leak is published.
- May 9, 2017 – Patch Tuesday: Second round of Equation Group patches includes a **silent fix** for the **EpMo** Equation Group exploit.

Summary

We began with analyzing “Jian”, the Chinese (APT31 / Zirconium) exploit for CVE-2017-0005, which was reported by Lockheed Martin’s Computer Incident Response Team. To our surprise, we found out that this **APT31 exploit** is in fact a **reconstructed version of an Equation Group exploit** called “EpMe”. This means that an Equation Group exploit was eventually used by a Chinese-affiliated group, probably against American targets.

This isn’t the first documented case of a Chinese APT using an Equation Group 0-Day. The first was when APT3 used their own version of EternalSynergy (called UPSynergy), after acquiring the Equation Group EternalRomance exploit. However, in the UPSynergy case, the consensus among our group of security researchers as well as in Symantec was that the Chinese exploit was reconstructed from captured network traffic.

The case of EpMe / Jian is different, as we clearly showed that Jian was constructed from the actual 32-bits and 64-bits versions of the Equation Group exploit. This means that in this scenario, the **Chinese APT acquired the exploit samples** themselves, in all of their supported versions. Having dated APT31’s samples to 3 years prior to the Shadow Broker’s “Lost in Translation” leak, our estimate is that these Equation Group exploit samples could have been acquired by the Chinese APT in one of these ways:

- Captured during an Equation Group network operation on a Chinese target.
- Captured during an Equation Group operation on a 3rd-party network which was also monitored by the Chinese APT.
- Captured by the Chinese APT during an attack on Equation Group infrastructure.

While reviewing the NtElevation exploits used in Equation Group’s DanderSpritz post-exploitation framework, we found 4 Windows LPE exploits. The first two NtElevation exploits were font vulnerabilities that were previously discussed as part of the Houston disk (an earlier sample attributed to Equation Group). In addition, EpMe (CVE-2017-0005) was mentioned and patched when Jian was caught, even if at that point in time the true origins of it weren’t yet known.

Finally, although EpMo was indeed patched by Microsoft in May 2017, we couldn’t trace the CVE-ID that was assigned to the patched vulnerability. Not only that, to our knowledge, our publication is the first to even mention the existence of this Equation Group exploit, even though it was publicly accessible on GitHub for the last 4 years.

These are our new additions to the attribution map:

- **EpMe (CVE-2017-0005)** – An Equation Group exploit that was cloned by APT31, thus causing CVE-2017-0005 to be attributed to the latter, instead of to Equation Group.
- **EpMo** – An additional Equation Group exploit that was never discussed before.
- **Jian** – APT31’s cloned version of EpMe, which was caught-in-the-wild by Lockheed Martin’s IRT.
- **CVE-2019-0803** – Attributed by multiple sources to a “Chinese State-Sponsored Actor.” We showed that it shares the same exploit loader (and tool API) as APT31’s Jian.

Appendix – IOC Table

Jian – CVE-2017-0005:

Jian: `AE512F13136774B4AAB79EBCC378927143BE77181E3B256E6F9940CE73696DE4`

CVE-2019-0803:

tools.dll: `68A3710765DA1886F00E40F2D5E02776D224C77AEA114CD22C3A6204A7FAD363`
2008.dll: `279320EE5C3B2DA4364AFBACBE5286EC4EED9AB5E887D4E0B9AAB2EB618BC539`

Equation Group Exploits:

Note: There are several variants of each exploit in the leak. The following are single examples of each exploit.

Houston Disk: `868EB363F32BEACD8BCDC7A114E020D4CFE67913A15275F4E7493D87DB643FF2`
DanderSpritz – ElEi: `C99FFACBA6D6689F7934E6E912E36EFCC4BD6A09C8A4D1E43BB27C3AFD131882`
DanderSpritz – ErNi: `E4FBF75ABF928CD1C9073656A61755FD3F0C25DC2E7922FB5073E1F64E5E9161`
DanderSpritz – EpMo: `1537CAD1D2C5154E142AF775ED555D6168D528BBE40B31F451EFA92C9E4F02DE`
DanderSpritz – EpMe (CVE-2017-0005):
`634A80E37E4B32706AD1EA4A2FF414473618A8C42A369880DB7CC127C0EB705E`

Source: https://research.checkpoint.com/2021/the-story-of-jian