UPSynergy: Chinese-American Spy vs. Spy Story

By Yaroslav Harakhavik

Published: 2019-09-05 · Archived: 2026-04-17 02:11:51 UTC

Research By: Mark Lechtik & Nadav Grossman

Introduction

Earlier this year, our colleagues at Symantec uncovered an interesting story about the use of Equation group exploitation tools by an alleged Chinese group named Buckeye (a.k.a APT3, or UPS team). One of the key findings in their publication was that variants of the Equation tools were used by the group prior to 'The Shadow Brokers' public leak in 2017. Moreover, it seems that APT3 developed its own in-house capabilities and equipped its attack tool with a 0-day that targeted the Windows operating system.

Following these revelations, we decided to expand on Symantec's findings and take a deeper look at Bemstour, the group's exploitation tool. In our analysis, we try to understand the background environment in which it was created, and provide our perspective of how it was developed. Our observations from the technical analysis allow us to provide evidence for a speculation that was formerly suggested by Symantec – APT3 recreated its own version of an Equation group exploit using captured network traffic. We believe that this artifact was collected during an attack conducted by the Equation group against a network monitored by APT3, allowing it to enhance its exploit arsenal with a fraction of the resources required to build the original tool.

APT3 is known to be a long-standing and sophisticated threat actor, having a record of using advanced TTPs, such as leveraging zero day exploits in its attacks. Such capabilities are consistent with former research by Intrusion Truth and Recorded Future, stating that the entity behind APT3 is the Chinese Ministry of State Security. That said, it wasn't very clear so far whether the group developed its exploits in-house or acquired them elsewhere. In this publication we deliver a glance into one possible modus operandi – the Chinese collect attack tools used against them, reverse engineer and reconstruct them to create equally strong digital weapons.

Following is a summary of our key findings:

- The group's exploitation tool named Bemstour makes use of a variant of a single Equation group exploit. Our research shows that the particular equivalent to this exploit is EternalRomance. APT3 developed their own implementation, possibly based on their analysis and understanding of EternalRomance's leveraged vulnerability.
- The group attempted to develop the exploit in a way that allowed it to target more Windows versions, similar to what was done in a parallel Equation group exploit named EternalSynergy. This required looking for an additional 0-day that provided them with a kernel information leak. All of this activity suggests that the group was not exposed to an actual NSA exploitation tool, as they would then not need to create another 0-day exploit. We decided to name APT3's bundle of exploits UPSynergy, since, much like in the case of Equation group, it combines 2 different exploits to expand the support to newer operating systems.

- The underlying SMB packets used throughout the tool execution were crafted manually by the developers, rather than generated using a third party library. As a lot of these packets were assigned with hardcoded and seemingly arbitrary data, as well as the existence of other unique hardcoded SMB artifacts, we can assume that the developers were trying to recreate the exploit based on previously recorded traffic.
- If network traffic was indeed used by the group as a reference, the traffic was likely collected from a machine controlled by APT3. This means either a Chinese machine that was targeted by the NSA and monitored by the group, or a machine compromised by the group beforehand on which foreign activity was noticed. We believe the former is more likely, and in that case could be made possible by capturing lateral movement within a victim network targeted by the Equation group.
- Finding a 0-day info leak, recreating the exploit based on the aforementioned vulnerability, and utilizing a lot of internal undocumented structures of SMB in the implants, implies that there was a similar expertise with and analysis performed on SMB drivers (with an eye to exploiting them) on the Chinese side, roughly at the same time it was widely used by the NSA. This, to some extent, suggests a narrative where China and the US are engaged in a cyber arms race to develop new exploits.

In the following sections, we provide the technical basis for our conclusions, by taking a tour through the tool's internals, its underlying exploit, and the implant's nuts and bolts. We also dive deeply into the root cause for the 0-day found by APT3. To the best of our knowledge, this hasn't been described anywhere else.

Overview of the Bemstour Tool

Besmtour is a tool developed by APT3 to gain remote code execution on a victim's machine using UPSynergy – a combination of an exploit based on EternalRomance and a 0-day found by the group itself. The goal is to deploy a payload on the victim's machine which is injected to a running process using an implant. This implant is highly similar to the Equation group's DoublePulsar.

The tool is meant to be run from a command line, and provides 2 modes of operation. In the first, the attacker sends a local file which will be executed on the victim machine with a given command line argument. In the 2nd mode, the attacker runs an arbitrary shell command without the need to send an actual file.
These functionalities are supported in both 32 and 64 bit versions. According to Symantec, the 64 bit versions were leveraged solely for executing shell commands, mostly to generate new user accounts in the victims' environments.

Figure 1: The 2 modes of operation provided by Bemstour.

One thing we noted about Bemstour's code is the way it generates and sends traffic to the victim's machine. In particular, we noticed that all packets are built manually, i.e. the developers created structs to represent the various SMB packets to send to the victim, and issued them over plain TCP sockets.

Figure 2: An example of a manually constructed SMB header.

As part of the manual crafting of SMBs, the developers assigned them with values hardcoded in the binary. Some of these reside within the data section in the form of custom structs, such as the one depicted in the figure below. When such a hardcoded assignment is required, an allocated SMB and the hardcoded structure are issued as arguments to a specific function, which in turn takes the custom struct's field values and assigns them to the corresponding SMB fields.

Figure 3: A custom structure containing fields to populate SMB headers with hardcoded values.

When looking at a structure like this, it's noteworthy that some of its fields represent unique values that are generated per SMB connection. One such value is the UID, which can be declared by the client and therefore could be chosen arbitrarily by Bemstour. In this case, there are multiple instances where this field is given

hardcoded unique values in a particular range, which may hint that they were copied from a source like recorded network traffic.

Figure 4: Instances of the custom hardcoded SMB header structure.

We found other hardcoded structures that are actually not used in any place in the code, but whose values and order imply the field they represent. This suggests that these are header fragments that were left as residues in the binary from another source. An example of such structs is shown below, where a numeric proximity can be seen to those UIDs that are used in the code.

Figure 5: Unused fragments of SMB header that are left in the binary.

There are additional hardcoded artifacts that may provide some insight into the tool's nature. For example, the PDB path (seen in the figure below) points out that the tool's source name is "SMB Master", and it was part of a project called "SMB_FOR_ALL_Ultimate-signature." Based on this, we can speculate that the project was indeed about repurposing an SMB exploit to target "ALL" (or at least more) versions of Windows.

Figure 6: PDB path hardcoded in the tool's binary.

Finally, more unused strings show something that looks like a concatenation of a computer name, user name and perhaps domain name. It is unclear where they come from, but again, strengthens the idea that this network entity was part of a referenced traffic capture.

Figure 7: Unused strings that reveal a network entity.

Overview of the Eternal* Exploits

Before we take a further look at the details of APT3's exploit implementation, we need to understand the various Eternal exploits that were incorporated into the Lost in Translation leak by The Shadow Brokers. Back in 2017, when this leak was released, 4 Eternal exploits were uncovered: EternalBlue, EternalChampion, EternalRomance and EternalSynergy.

Both and EternalRomance targeted mostly Windows 7 systems (as well as lower version of Windows NT where SMBv1 is located). One of the problems in adapting EternalRomance to higher Windows versions was a patch introduced in Windows 8 which eliminated the possibility to use an information leak vulnerability leveraged by it.

To deal with this problem, the Equation group came up with an upgraded version where the problematic info leak was replaced with one that could be exploited on Windows 8. Essentially, there was nothing new there, as the info leak exploit was already used in EternalChampion and other parts of EternalRomance remained the same. This new hybrid exploit was named EternalSynergy, suggesting the way it was built – a synergy of 2 exploits.

When it comes to the exploit in the Bemstour tool, it is evident that there's an attempt to leverage the same vulnerability exploited by EternalRomance. At the same time, there is the use of a whole new information leak exploit, which was in fact a 0-day found by APT3. As we will see in the upcoming section, this particular information leak is quite robust and allowed the group to upgrade their version of EternalRomance to use in versions higher than Windows 7.

In this sense, APT3 crafted its own exploit from other exploits – a tactic very similar to one used by the Equation group. As this threat group also uses the name UPS team, we decided to name their version of the exploit bundle UPSynergy.

Root Cause Analysis of CVE 2019-0703

According to Microsoft, CVE-2019-0703 is "*an information disclosure vulnerability [that] exists in the way […] the Windows SMB Server handles certain requests. An authenticated attacker who successfully exploited this vulnerability could craft a special packet, which could lead to information disclosure from the server*.

To exploit the vulnerability, an attacker would have to be able to authenticate and send SMB messages to an impacted Windows SMB Server The security update addresses the vulnerability by correcting how Windows SMB Server handles authenticated requests."

Our analysis shows a slightly different picture. The vulnerability is in fact a logical bug related to querying information from the Windows Named Pipes mechanism, and **not** a vulnerabilityin the SMB protocol nor its

implementation. While it can be triggered using SMB, there are other ways to leverage it, e.g. using the NtQueryInformationFile Windows API call that is unrelated to SMB.

The bug resides within *npfs.sys* (Name Pipe File System driver) in a function named *NpQueryInternalInfo*. The latter is used to query named pipes and return a value called a file reference number, which according to Microsoft *"MUST be assigned by the file system and is unique to the volume on which the file or directory is located."*

However, our analysis shows that the returned value is not a file reference number, but rather a pointer to a kernel structure named CCB (Client Control Block). This is an undocumented struct defined in *npfs.sys*, which has a partial definition (named NP_CCB) provided by the ReactOS project. Clearly, this is not the intended value to be returned in this case, and the leak of this struct discloses useful information that can be leveraged by attackers.

Figure 8: The leaked object is in fact a CCB struct, as evident from WinDbg.

To trigger this information disclosure vulnerability, a call with the following arguments is made to the NtQueryInformationFile stub in ntdll.dll:

FileHandle – Handle to a named pipe (for example "\\.\pipe\browser").
FileInformationClass – FileInternalInformation (equals 0x6).

After this happens, we get the following call stack:

Figure 9: Kernel mode call stack corresponding to an *NtQueryInformationFile* call from user mode.

As already mentioned, it is also possible to trigger this vulnerability via SMB, as was used by APT3. The method was used to determine the bitness of the attacked operating system and overwrite (using a write primitive) a field in the leaked structure, which eventually provided the group with remote code execution.

To leverage the vulnerability, you must first establish an SMB connection to a named pipe on the victim's machine, as can be seen in the figure below.

Figure 10: Network capture of SMB packets that demonstrates an establishment of a connection to the *\pipe\browser* named pipe (FID 0x4000).

Next, it's possible to query information about the opened pipe using the 0x32 SMB command (SMB_COM_TRANSACTION2) and the 0x7 subcommand (TRANS2_QUERY_FILE_INFORMATION). The latter has a field named InformationLevel which describes the types of information that can be retrieved by the server.

Furthermore, if the server declared a capability named Infolevel Passthru in its *Negotiate Response* field as a part of an earlier negotiation (a capability usually provided by default), more types of information can be retrieved, namely ones that provide native file information on the server. In this case, the former capability allows it to provide a code number named a Pass-thru Information Level by the client, which maps directly to another Windows NT numerical value called an Information Class on the server. This value corresponds to the *FileInformationClass* parameter of the *NtQueryInformationFile* API, which specifies what type of file information to query from a server destined file object.

To use one of the pass-thru Information Levels to request a corresponding Information Class for a file on the server, it is sufficient to add the value 0x3e8 (SMB_INFO_PASSTHROUGH) to the requested Information Class. As an example, if we take the FileInternalInformation Information Class (which has the value 6) and want to get the corresponding Information Level, we just need to add the previously mentioned value to it, resulting in the value 0x3ee.

In our case, using this very same Information Level by placing it as a parameter of the TRANS2_QUERY_FILE_INFORMATION subcommand, triggers the vulnerability by causing the invocation of the NtQueryInformationFile from the *srv.sys* driver (SMB driver). The latter in turn calls the vulnerable *NpQueryInternalInfo* from npfs.sys, as depicted in the stack trace below.

Figure 11: Kernel mode calls stack resulting from execution of the SMB transaction that triggers the bug.

Consequently, when we issue a Trans2 request to query for a file info using the previously mentioned Info Level, we get a CCB leaked pointer in the response.

Figure 12: Wireshark's view of triggering the vulnerability.

To examine the described root cause for the vulnerability from another angle, we can take a look at the diff between the patched and unpatched code in *npfs.sys*:

Figure 13: Patch diff – The vulnerable code can be seen in the upper part.

As can be seen in the vulnerable code, the *out_buffer* argument returned to the caller and then to the client contains a pointer to the *ClientControlBlock* (NP_CCB) argument instead of the file reference number. This is fixed in the patched code, where offsets 0xa0 and 0xa4 from *ClientControlBlock* are written to the *out_buffer* instead, thus returning the actual intended file reference number to the caller and client.

As mentioned previously, the information obtained from this info leak can give us the ability to execute code on the victim machine, using another write primitive. To understand how this is possible, we need to take a closer look at the CCB structure. One of its members points to yet another undocumented struct, which we will denote as 'struct x'. This struct contains a pointer to a function that is called when the connection to the named pipe is terminated, which we'll refer to as the 'pipe destructor function'.

In APT3's implementation of the exploit, the HAL heap is written with both shellcode and a rogue instance of 'struct x'. The latter simply contains a pointer to the shellcode in the position of the 'pipe destructor function'. Therefore, when we use a write primitive and know the whereabouts of the leaked CCB structure, we can overwrite its pointer to 'struct x' so that it points to the rogue instance. After the connection is closed, the shellcode is triggered and the attacker can run arbitrary code on the victim's machine.

Comparison of UPSynergy and Eternal Romance Implementations

One of the observations we made during our analysis of Bemstour was that its main exploit targets only a particular vulnerability that overlaps with one used by the Equation group. This vulnerability is rooted in a type confusion bug leveraged in a similar fashion in the EternalRomance exploit, which was then reused together with other exploits in EternalSynergy.

As a result of this type confusion between SMB messages, the server considers an unrelated SMB message as part of an SMB Transaction of a different type, and activates the wrong type of SMB handler. This handler in turn

shifts the Transaction struct's pointer to the incoming data buffer by the amount of data received in the SMB message. Because the pointer value was shifted by the wrong handler, data of further SMB messages (which are treated by the correct type of handler) can be potentially written outside the boundaries of the incoming data buffer. If there was successful grooming (i.e. the heap was correctly shaped beforehand), this out-of-bound write may allow us to overwrite an adjacent SMB Transaction structure.

Instead of going through every detail of APT3's exploit, the table below compares the underlying techniques used by EternalRomance vs. those used by UPSynergy. Detailed information about the bug (**CVE-2017-0143**) and how it was exploited in EternalRomance to gain a write-what-where and read-what-where primitives is explained very well by Microsoft in their analysis of EternalSynergy.

Exploitation Technique	**EternalRomance**	**APT3 Exploit (UPSynergy)**
Determine the OS Type	Determined from the server's session SetupAndX response (part of a session negotiation), where the underlying target OS is specified.	Same technique.
Determine the OS Bitness	Uses a leaked pool header structure that contains parameters from which the OS architecture can be inferred.	Uses the address of the leaked CCB structure to infer the range in which it resides and the underlying architecture.
Grooming Technique (Heap Shaping)	Uses 2 types of allocations with different sizes, named "bride" and "groom." Another technique is used for OS versions prior to Windows 7.	Uses "bride" allocations only, with a different allocation size.
Leaked Object	Leaks a kernel object named Transaction (corresponding to an SMB Transaction).	Leaks a kernel object named CCB (Client Control Block).
OOB Write Vulnerability	A result of a type confusion bug, as outlined above.	Same vulnerability.
Write-What-Where primitive	Can be achieved by overwriting the input buffer pointer of a target Transaction structure, as outlined above.	Same technique.
Read-What-Where primitive	Can be achieved by overwriting the output buffer pointer of a target Transaction structure, as outlined above.	APT3 doesn't use this primitive.
RWE Cave	Uses an RWE page in the *srv.sys* memory section.	Uses HAL's heap.
First Shellcode Execution	Overwrites an unimplemented SMB command pointer in the SMB command handler table, and	Overwrites a named pipe connection handler function

	sends an SMB transaction for this command to execute a handler (which is in fact shellcode).	which executes after the connection is closed.

In addition, we conducted a quantitative analysis of various actions performed during both exploits, as can be seen in the following table:

Parameter	**EternalRomance**	**APT3 Exploit (UPSynergy)**
Info leak exploit usage	2 times	1 time
Usage of a write-what-where primitive	24 times	3 times
Usage of a read-what-where primitive	4 times	Not used
Number of attempts to overwrite a Transaction structure in case of failure on the first try	2 attempts	0 attempts

From this table, we can infer that the UPSynergy information leak significantly eases the exploitation process, as the leaked CCB object described earlier contains almost a direct code execution primitive. In EternalRomance, we could see the usage of a read-what-where primitive, mainly used for dereferencing child structs of a leaked Transaction struct. In the case of UPSynergy, that would be redundant.

Having said that, there is a slight chance of instability in the grooming implementation of UPSynergy, where a write to an unallocated page might lead to an unintended BSOD. This will not happen in EternalRomance (point for the Equation group).

Comparison of APT3 and Equation Group Implants

The last action to take place following the exploitation is the set-up and invocation of an implant shellcode. The purpose is to serve as a basic backdoor, allowing the attacker to issue a further kernel mode payload and execute it on the target machine. In the case of both APT3 and Equation group, an implant named DoublePulsar is used. This implant was leaked by The Shadow Brokers in 2017.

In both cases, there is a very similar flow to the implant's operation – a hook is set up for a particular SMB handler function to handle invalid SMBs. This hook searches for one of 3 commands in a particular SMB field and executes a corresponding function for each one. One of the supported commands is responsible for accepting further shellcode and running it – the last stage payload. At this point, the attacker may issue an arbitrary piece of code for execution in the kernel space.

As far as APT3's implant is concerned, it seems likely f the DoublePulsar code was reused as is. The code is not executed directly, but has several layers of obfuscation. Essentially, the Equation group's DoublePulsar code is wrapped with an APT3 position independent crypter & loader.

In the following sections, we take a look at the differences and supplements provided by APT3. As we will see, the main logic flow was preserved in both cases. However, the differences show that APT3 did not want to fully

disclose the fact it was using an allegedly American implant.

1st Stage – DoublePulsar Loader

The very first stage of the implant's code is a custom loader written by APT3, which extracts an encrypted version of DoublePulsar code from incoming SMB packets, and decrypts and executes it. This is in fact a self-modifying piece of code, i.e. before it actually handles any of the aforementioned functionalities, it must decrypt subsequent parts of itself. The code is wrapped in 2 layers using simple crypters, so the first crypter decodes the second, and the latter decodes the actual loader code.

Figure 14: First and second phase of decoding the loader's payload.

After these phases are completed, the loader starts its operation which is broken down into the following steps:

1. Dynamic function resolution.
2. Determine the OS version.

3. Locate the *SrvTreeConnectList* in *sys*.
4. Extract the encoded shellcode from a *Transaction* object list.
5. Execute the shellcode.

The figure below summarizes this flow, showing the main code of this loader. We then present a detailed outline of each of these steps, and point out the major differences that set this code apart from that of the Equation group.

Figure 15: The main flow of the 1st stage loader.

- Step 1: Dynamic function resolution

As this is essentially position independent code, we need to resolve some API functions dynamically, which are then used during run time. First, we must locate the base address of the *ntoskrnl.exe* image. We do this by obtaining the KPCR structure from the FS register, and use offset 0x38 which points to *KIDTENTRY *IDT* (i.e. the interrupt dispatch table). As we know the latter resides within *ntoskrnl.exe* and is aligned to the beginning of a page, so it is sufficient to walk back in page multiples until the start of the page is equivalent to the magic number of a PE.

After that is done, it's possible to parse the export table of *ntoskrnl.exe* to achieve several basic API function addresses. A common technique is to parse the export tables of a relevant loaded image where these functions reside (e.g. *ntoskrnl.exe*), hash the names of their exports, and compare them to hardcoded ones. The latter represent the names of the functions that require address resolution. In this case, we see that the hashing function differs from that of the Equation group, resulting in different name hashes.

Function Name	**APT3 Hash**	**Equation Hash**
ZwQuerySystemInformaiton	0x8754A7F7	0x0D2515B2E
ExAllocatePoolWithTag	0x37F154D9	–

ExAllocatePool	–	0x0E3690194
ExFreePool	0x3F7747DE	0x0F0835485
RtlGetVersion	0x0DDE5CDD	–

Figure 16: Different name hashing implementations and their resulting string hashes.

We see that not only different hashing algorithms are used, but also different API functions. For instance, the Equation group uses a simple pool allocation via the *ExAllocatePool* API, while APT3 uses a tagged allocation and calls *ExAllocatePoolWithTag*. In the latter case, the used tag represents a work context structure.

Figure 17: APT3 tagged allocation.

- Step 2: Determining the OS version

Next, the loader invokes the *RtlGetVersion* function to obtain information about the underlying Windows version. It then assigns a numeric value to a field in a particular struct maintained by the loader, which corresponds to the OS version. The value is in fact an offset into an undocumented SMB struct called CONNECTION, which will result in a field that points to yet another undocumented struct called PAGED_CONNECTION. How this struct is used will be evident in subsequent steps.

Figure 18: Determining the version of Windows, and choosing a corresponding offset value.

- Step 3: Locating *sys* and *SrvTreeConnectList*

At this point, the loader tries to find *Srv.sys* (the SMB driver's image) and parse it. This is done to locate a global undocumented list named *SrvTreeConnectList*. *Srv.sys* is located using *ZwQuerySystemInformation* to obtain a list of loaded module information (where a base address of the loaded images is specified), while the struct is found by going through *Srv.sys'.data* section and looking for several identifying numeric parameters.

- Step 4: Extracting encoded shellcode from a *Transaction* object list.

After the list is found, it is used to go through several linked SMB structures to finally obtain a list of *Transaction* structs. The latter allows us to access the data obtained from relevant SMB Trans packets which contains the subsequent shellcode.

The chain of these structures can be seen in the figure below. The main takeaway is that all of these structures are undocumented – i.e. the developers of APT3 must have done quite a bit of reverse engineering on *Srv.sys* to infer them (on more than one Windows version, as evident from the offset to PAGED_CONNECTION). This effort is very similar to the one invested by the NSA to find the various Eternal exploits around the same time.

Figure 19: APT3's code to 'walk' through various undocumented SMB strucures, suggesting that a considerable analysis was performed and the group's members have a good understanding of the SMB internals.

- Step 5: Executing the shellcode

After the shellcode is obtained and decoded, it is finally executed. This leads to the next stage, which is yet another piece of self-modifying PIC. However, in this case, most of the code that is unravelled after 2 layers of decoding is a variant of the original DoublePulsar, as used by the Equation group.

2nd Stage – DoublePulsar Installation & Hook

In this stage of the implant's operation, yet another shellcode runs. As previously mentioned, this code is obfuscated with 2 layers of crypters, the same ones used to wrap the loader in the 1st stage. The code that is unpacked was mostly not written by APT3.

The first part of the resulting PIC seems to be custom-made, and invokes a system thread that works periodically to form paged allocations of various sizes. It can run in rounds indefinitely, creating 256 allocations for each round and counting the number that get an address within the range of 64 bytes from the point in which the first shellcode was written. Only if there are more than 64 'faulty' allocations can this loop terminate. The purpose is not fully clear, but could be an attempt to avoid paging out the shellcode buffers from the paged pool.

The other part of this internal payload installs DoublePulsar. This is done by replacing a function pointer to point at a hook function instead of the original function named *SrvTransactionNotImplemented*. The replacement of this pointer happens in a hard-coded table in the SMB driver (*srv.sys*) named *SrvTransaction2DispatchTable*.

In essence, both APT3 and the Equation group take similar steps to achieve this goal. These are outlined in the figure below, and are more thoroughly explained here and here.

Figure 20: The general steps taken to install DoublePulsar's hook, in both implementations of the shellcode.

You can see the similarity in the call flow graph comparison of these hook functions:

Figure 21: CFG comparison of the DoublePulsar hook functions.

This particular hook function anticipates an initial command named "ping" where a XOR key is obtained from the attacker. This key can then be used to decode the payload of subsequent SMBs carrying additional shellcode. The latter is executed as part of another command called "exec".

Figure 22: Commands supported by the 2nd stage hook backdoor.

There is an addition to the code that was not observed in other variants of DoublePulsar. This addition is a common snippet used to disable the WP bit flag of the CR0 register, which allows the kernel to write into read only pages. It is not clear if this serves any purpose in the implant's operation, but it is reasonable to assume that it was bundled to a version of DoublePulsar that was captured by APT3 and was simply left as a code residue.

Figure 23: Code snippet from DoublePulsar, used to clear the WP flag in CR0.

3rd Stage – APC Injector

The last stage of the implant is a piece of code that performs APC injection of a hardcoded routine to the "services.exe" process in the user space. In turn, this routine can write a given payload to a new file and execute it, or run a shell command. In both cases, the API used for the execution is *WinExec*.

It's worth noting that while an arbitrary command can be issued by the user, there are several hardcoded commands that the shellcode runs through the invoked APC in the user space. One of these commands adds a new user as local admin with a hardcoded name and password. In the sample analyzed for this publication, this username is *cessupport* and the password is *1qaz#EDC*.

Figure 24: Hardcoded shell command to add a new admin user to the system.

The implementation of this part doesn't resemble that of the Equation group (compared to their equivalent APC injector). It's also different from the previous stages of APT3's implant. For example, function resolution does not

use string hashes anymore, but rather makes comparisons to strings stored in the stack. The allocations are no longer tagged and the overall choice of API functions for similar actions looks different. This may mean that there was another entity within the group that was involved in the development of this part, but not of previous ones.

Conclusion

In our research, we analyzed and compared the exploit development efforts done by 2 major actors in the APT landscape – the Equation group and APT3. While the former is known for its advanced and almost unparalleled capabilities in the field of vulnerability research, it is interesting to observe how other groups focus on similar research objectives, with a considerable degree of success.

It's not always clear how threat actors achieve their exploitation tools, and it's commonly assumed that actors can conduct their own research and development or get it from a third party. In this case we have evidence to show that a third (but less common) scenario took place – one where attack artifacts of a rival (i.e. Equation group) were used as the basis and inspiration for establishing in-house offensive capabilities by APT3.

Although we can't prove this beyond any doubt, we brought many facts and analysis findings to back up our speculations. We will continue our efforts to find the answers to these as well as any future questions that arise.

Check Point protects against the exploits issued by the Bemstour tool with the IPS protection 'Microsoft SMB Client Transaction Memory Corruption (MS10-020)'.

We would like to thank Eyal Itkin for assisting in parts of the analysis during this research.

IOCs

MD5:

```
F595228976CC89FFAC02D831E774CFA6
```

SHA1:

```
80143E32F887B2583B777DAEC5982FB5C2886FB3
```

SHA256:

```
0B28433A2B7993DA65E95A45C2ADF7BC37EDBD2A8DB717B85666D6C88140698A
```

Yara Rules:

```
rule apt3_bemstour_strings
{
meta:

description = "Detects strings used by the Bemstour exploitation tool"
```

```
author = "Mark Lechtik"
company = "Check Point Software Technologies LTD."
date = "2019-06-25"
sha256 = "0b28433a2b7993da65e95a45c2adf7bc37edbd2a8db717b85666d6c88140698a"
strings:

$dbg_print_1 = "leaked address is 0x%llx" ascii wide
$dbg_print_2 = "========== %s ==========" ascii wide
$dbg_print_3 = "detailVersion:%d" ascii wide
$dbg_print_4 = "create pipe twice failed" ascii wide
$dbg_print_5 = "WSAStartup function failed with error: %d" ascii wide
$dbg_print_6 = "can't open input file." ascii wide
$dbg_print_7 = "Allocate Buffer Failed." ascii wide
$dbg_print_8 = "Connect to target failed." ascii wide
$dbg_print_9 = "connect successful." ascii wide
$dbg_print_10 = "not supported Platform" ascii wide
$dbg_print_11 = "Wait several seconds." ascii wide
$dbg_print_12 = "not set where to write ListEntry ." ascii wide
$dbg_print_13 = "backdoor not installed." ascii wide
$dbg_print_14 = "REConnect to target failed." ascii wide
$dbg_print_15 = "Construct TreeConnectAndX Request Failed." ascii wide
$dbg_print_16 = "Construct NTCreateAndXRequest  Failed." ascii wide
$dbg_print_17 = "Construct Trans2  Failed." ascii wide
$dbg_print_18 = "Construct ConsWXR  Failed." ascii wide
$dbg_print_19 = "Construct ConsTransSecondary  Failed." ascii wide
$dbg_print_20 = "if you don't want to input password , use server2003 version.." ascii wide

$cmdline_1 = "Command format  %s TargetIp domainname username password 2" ascii wide
$cmdline_2 = "Command format  %s TargetIp domainname username password 1" ascii wide
$cmdline_3 = "cmd.exe /c net user test test /add && cmd.exe /c net localgroup administrators test /a
$cmdline_4 = "hello.exe  \"C:\\WINDOWS\\DEBUG\\test.exe\"" ascii wide
$cmdline_5 = "parameter not right" ascii wide

$smb_param_1 = "browser" ascii wide
$smb_param_2 = "spoolss" ascii wide
$smb_param_3 = "srvsvc" ascii wide
$smb_param_4 = "\\PIPE\\LANMAN" ascii wide
$smb_param_5 = "Werttys for Workgroups 3.1a" ascii wide
$smb_param_6 = "PC NETWORK PROGRAM 1.0" ascii wide
$smb_param_7 = "LANMAN1.0" ascii wide
$smb_param_8 = "LM1.2X002" ascii wide
$smb_param_9 = "LANMAN2.1" ascii wide
$smb_param_10 = "NT LM 0.12" ascii wide
$smb_param_12 = "WORKGROUP" ascii wide
$smb_param_13 = "Windows Server 2003 3790 Service Pack 2" ascii wide
$smb_param_14 = "Windows Server 2003 5.2" ascii wide
$smb_param_15 = "Windows 2002 Service Pack 2 2600" ascii wide
```

```
$smb_param_16 = "Windows 2002 5.1" ascii wide
$smb_param_17 = "PC NETWORK PROGRAM 1.0" ascii wide
$smb_param_18 = "Windows 2002 5.1" ascii wide
$smb_param_19 = "Windows for Workgroups 3.1a" ascii wide

$unique_str_1 = "WIN-NGJ7GKNROVS"
$unique_str_2 = "XD-A31C2E0087B2"

condition:
    uint16(0) == 0x5a4d and (5 of ($dbg_print*) or 2 of ($cmdline*) or 1 of ($unique_str*)) and 3 of
}
```

```
rule apt3_bemstour_implant_byte_patch
{
meta:

description = "Detects an implant used by Bemstour exploitation tool (APT3)"
author = "Mark Lechtik"
company = "Check Point Software Technologies LTD."
date = "2019-06-25"
sha256 = "0b28433a2b7993da65e95a45c2adf7bc37edbd2a8db717b85666d6c88140698a"

/*

0x41b7e1L C745B8558BEC83                mov dword ptr [ebp - 0x48], 0x83ec8b55
0x41b7e8L C745BCEC745356                mov dword ptr [ebp - 0x44], 0x565374ec
0x41b7efL C745C08B750833                mov dword ptr [ebp - 0x40], 0x3308758b
0x41b7f6L C745C4C957C745                mov dword ptr [ebp - 0x3c], 0x45c757c9
0x41b7fdL C745C88C4C6F61                mov dword ptr [ebp - 0x38], 0x616f4c8c

*/

strings:

$chunk_1 = {

C7 45 ?? 55 8B EC 83
C7 45 ?? EC 74 53 56
C7 45 ?? 8B 75 08 33
C7 45 ?? C9 57 C7 45
C7 45 ?? 8C 4C 6F 61

}

condition:
```

```
    any of them
}
```

```
rule apt3_bemstour_implant_command_stack_variable
{
meta:

description = "Detecs an implant used by Bemstour exploitation tool (APT3)"
author = "Mark Lechtik"
company = "Check Point Software Technologies LTD."
date = "2019-06-25"
sha256 = "0b28433a2b7993da65e95a45c2adf7bc37edbd2a8db717b85666d6c88140698a"


strings:


/*

0x41ba18L C78534FFFFFF636D642E          mov dword ptr [ebp - 0xcc], 0x2e646d63
0x41ba22L C78538FFFFFF65786520          mov dword ptr [ebp - 0xc8], 0x20657865
0x41ba2cL C7853CFFFFFF2F632063          mov dword ptr [ebp - 0xc4], 0x6320632f
0x41ba36L C78540FFFFFF6F707920          mov dword ptr [ebp - 0xc0], 0x2079706f
0x41ba40L C78544FFFFFF2577696E          mov dword ptr [ebp - 0xbc], 0x6e697725
0x41ba4aL C78548FFFFFF64697225          mov dword ptr [ebp - 0xb8], 0x25726964
0x41ba54L C7854CFFFFFF5C737973          mov dword ptr [ebp - 0xb4], 0x7379735c
0x41ba5eL C78550FFFFFF74656D33          mov dword ptr [ebp - 0xb0], 0x336d6574
0x41ba68L C78554FFFFFF325C636D          mov dword ptr [ebp - 0xac], 0x6d635c32
0x41ba72L C78558FFFFFF642E6578          mov dword ptr [ebp - 0xa8], 0x78652e64
0x41ba7cL C7855CFFFFFF65202577          mov dword ptr [ebp - 0xa4], 0x77252065
0x41ba86L C78560FFFFFF696E6469          mov dword ptr [ebp - 0xa0], 0x69646e69
0x41ba90L C78564FFFFFF72255C73          mov dword ptr [ebp - 0x9c], 0x735c2572
0x41ba9aL C78568FFFFFF79737465          mov dword ptr [ebp - 0x98], 0x65747379
0x41baa4L C7856CFFFFFF6D33325C          mov dword ptr [ebp - 0x94], 0x5c32336d
0x41baaeL C78570FFFFFF73657468          mov dword ptr [ebp - 0x90], 0x68746573
0x41bab8L C78574FFFFFF632E6578          mov dword ptr [ebp - 0x8c], 0x78652e63
0x41bac2L C78578FFFFFF65202F79          mov dword ptr [ebp - 0x88], 0x792f2065
0x41baccL 83A57CFFFFFF00                and dword ptr [ebp - 0x84], 0

*/

$chunk_1 = {

C7 85 ?? ?? ?? ?? 63 6D 64 2E
C7 85 ?? ?? ?? ?? 65 78 65 20
C7 85 ?? ?? ?? ?? 2F 63 20 63
```

```
C7 85 ?? ?? ?? ?? 6F 70 79 20
C7 85 ?? ?? ?? ?? 25 77 69 6E
C7 85 ?? ?? ?? ?? 64 69 72 25
C7 85 ?? ?? ?? ?? 5C 73 79 73
C7 85 ?? ?? ?? ?? 74 65 6D 33
C7 85 ?? ?? ?? ?? 32 5C 63 6D
C7 85 ?? ?? ?? ?? 64 2E 65 78
C7 85 ?? ?? ?? ?? 65 20 25 77
C7 85 ?? ?? ?? ?? 69 6E 64 69
C7 85 ?? ?? ?? ?? 72 25 5C 73
C7 85 ?? ?? ?? ?? 79 73 74 65
C7 85 ?? ?? ?? ?? 6D 33 32 5C
C7 85 ?? ?? ?? ?? 73 65 74 68
C7 85 ?? ?? ?? ?? 63 2E 65 78
C7 85 ?? ?? ?? ?? 65 20 2F 79
83 A5 ?? ?? ?? ?? 00
}



/*

0x41baeeL C785D8FEFFFF636D6420          mov dword ptr [ebp - 0x128], 0x20646d63
0x41baf8L C785DCFEFFFF2F632022          mov dword ptr [ebp - 0x124], 0x2220632f
0x41bb02L C785E0FEFFFF6E657420          mov dword ptr [ebp - 0x120], 0x2074656e
0x41bb0cL C785E4FEFFFF75736572          mov dword ptr [ebp - 0x11c], 0x72657375
0x41bb16L C785E8FEFFFF20636573          mov dword ptr [ebp - 0x118], 0x73656320
0x41bb20L C785ECFEFFFF73757070          mov dword ptr [ebp - 0x114], 0x70707573
0x41bb2aL C785F0FEFFFF6F727420          mov dword ptr [ebp - 0x110], 0x2074726f
0x41bb34L C785F4FEFFFF3171617A          mov dword ptr [ebp - 0x10c], 0x7a617131
0x41bb3eL C785F8FEFFFF23454443          mov dword ptr [ebp - 0x108], 0x43444523
0x41bb48L C785FCFEFFFF202F6164          mov dword ptr [ebp - 0x104], 0x64612f20
0x41bb52L C78500FFFFFF64202626          mov dword ptr [ebp - 0x100], 0x26262064
0x41bb5cL C78504FFFFFF206E6574          mov dword ptr [ebp - 0xfc], 0x74656e20
0x41bb66L C78508FFFFFF206C6F63          mov dword ptr [ebp - 0xf8], 0x636f6c20
0x41bb70L C7850CFFFFFF616C6772          mov dword ptr [ebp - 0xf4], 0x72676c61
0x41bb7aL C78510FFFFFF6F757020          mov dword ptr [ebp - 0xf0], 0x2070756f
0x41bb84L C78514FFFFFF61646D69          mov dword ptr [ebp - 0xec], 0x696d6461
0x41bb8eL C78518FFFFFF6E697374          mov dword ptr [ebp - 0xe8], 0x7473696e
0x41bb98L C7851CFFFFFF7261746F          mov dword ptr [ebp - 0xe4], 0x6f746172
0x41bba2L C78520FFFFFF72732063          mov dword ptr [ebp - 0xe0], 0x63207372
0x41bbacL C78524FFFFFF65737375          mov dword ptr [ebp - 0xdc], 0x75737365
0x41bbb6L C78528FFFFFF70706F72          mov dword ptr [ebp - 0xd8], 0x726f7070
0x41bbc0L C7852CFFFFFF74202F61          mov dword ptr [ebp - 0xd4], 0x612f2074
0x41bbcaL C78530FFFFFF64642200          mov dword ptr [ebp - 0xd0], 0x226464
0x41bbd4L 6A5C                          push 0x5c
```

```
*/

$chunk_2 = {

C7 85 ?? ?? ?? ?? 63 6D 64 20
C7 85 ?? ?? ?? ?? 2F 63 20 22
C7 85 ?? ?? ?? ?? 6E 65 74 20
C7 85 ?? ?? ?? ?? 75 73 65 72
C7 85 ?? ?? ?? ?? 20 63 65 73
C7 85 ?? ?? ?? ?? 73 75 70 70
C7 85 ?? ?? ?? ?? 6F 72 74 20
C7 85 ?? ?? ?? ?? 31 71 61 7A
C7 85 ?? ?? ?? ?? 23 45 44 43
C7 85 ?? ?? ?? ?? 20 2F 61 64
C7 85 ?? ?? ?? ?? 64 20 26 26
C7 85 ?? ?? ?? ?? 20 6E 65 74
C7 85 ?? ?? ?? ?? 20 6C 6F 63
C7 85 ?? ?? ?? ?? 61 6C 67 72
C7 85 ?? ?? ?? ?? 6F 75 70 20
C7 85 ?? ?? ?? ?? 61 64 6D 69
C7 85 ?? ?? ?? ?? 6E 69 73 74
C7 85 ?? ?? ?? ?? 72 61 74 6F
C7 85 ?? ?? ?? ?? 72 73 20 63
C7 85 ?? ?? ?? ?? 65 73 73 75
C7 85 ?? ?? ?? ?? 70 70 6F 72
C7 85 ?? ?? ?? ?? 74 20 2F 61
C7 85 ?? ?? ?? ?? 64 64 22 00
6A 5C

}

/*

0x41be22L C745D057696E45              mov dword ptr [ebp - 0x30], 0x456e6957
0x41be29L C745D478656300              mov dword ptr [ebp - 0x2c], 0x636578
0x41be30L C7459C47657450              mov dword ptr [ebp - 0x64], 0x50746547
0x41be37L C745A0726F6341              mov dword ptr [ebp - 0x60], 0x41636f72
0x41be3eL C745A464647265              mov dword ptr [ebp - 0x5c], 0x65726464
0x41be45L C745A873730000              mov dword ptr [ebp - 0x58], 0x7373
0x41be4cL C745C443726561              mov dword ptr [ebp - 0x3c], 0x61657243
0x41be53L C745C874654669              mov dword ptr [ebp - 0x38], 0x69466574
0x41be5aL C745CC6C654100              mov dword ptr [ebp - 0x34], 0x41656c
0x41be61L C745B857726974              mov dword ptr [ebp - 0x48], 0x74697257
0x41be68L C745BC6546696C              mov dword ptr [ebp - 0x44], 0x6c694665
0x41be6fL C745C065000000              mov dword ptr [ebp - 0x40], 0x65
0x41be76L C745AC436C6F73              mov dword ptr [ebp - 0x54], 0x736f6c43
```

```
0x41be7dL C745B06548616E                mov dword ptr [ebp - 0x50], 0x6e614865
0x41be84L C745B4646C6500                mov dword ptr [ebp - 0x4c], 0x656c64
0x41be8bL 894DE8                        mov dword ptr [ebp - 0x18], ecx

*/

$chunk_3 = {

C7 45 ?? 57 69 6E 45
C7 45 ?? 78 65 63 00
C7 45 ?? 47 65 74 50
C7 45 ?? 72 6F 63 41
C7 45 ?? 64 64 72 65
C7 45 ?? 73 73 00 00
C7 45 ?? 43 72 65 61
C7 45 ?? 74 65 46 69
C7 45 ?? 6C 65 41 00
C7 45 ?? 57 72 69 74
C7 45 ?? 65 46 69 6C
C7 45 ?? 65 00 00 00
C7 45 ?? 43 6C 6F 73
C7 45 ?? 65 48 61 6E
C7 45 ?? 64 6C 65 00
89 4D ??

}


condition:
    any of them
}
```

Source: https://research.checkpoint.com/upsynergy/