Working mechanism of Eternalblue and its application in ransomworm

Zian Liu

Swinburne University of Technology, Victoria 3122, Australia

Abstract. After the leaking of exploit Eternalblue, some ransomworms utilizing this exploit have been developed to sweep over the world in recent years. Ransomworm is a global growing threat as it blocks users' access to their files unless a ransom is paid by victims. Wannacry and Notpetya are two of those ransomworms which are responsible for the loss of millions of dollar, from crippling U.K. national systems to shutting down a Honda Motor Company in Japan. Many dynamic analytic papers on Wannacry were published, however, static analytic papers about Wannacry were limited. Our aim is to present readers an systematic knowledge about exploit Eternalblue, from a high–leveled semantic view to the code details. Specifically, the working mechanism of Eternalblue, the reverse engineering analysis of Eternalblue in Wannacry, and the comparison with the Metasploit's Eternalblue exploit are presented. The key finding of our analysis is that the code remains almost the same when Eternalblue is transplanted into Wannacry, which indicates its potential for signatures and thus detection.

Keywords: Cyber Threat · Ransomworm · Static Analysis · Wannacry

1 Introduction

With computers and networks being applied more and more widely in daily life, enterprise and organizations tend to store large scale data digitally. However, those information management systems often contain vulnerabilities and are prone to be exploited by hackers. In May 2017, a ransomworm called *Wannacry* bursts out worldwide. It caused massive infection and enormous economic losses by infecting various industrial and government internal networks, such as UK's National Health Service and etc [2]. The average attack the other organizations suffer is 14,300 per day according to [6]. It is reported to derive from an NSA exploit tool [15]. Once Wannacry infects a system, it encrypts copies of various file types and deletes the originals. The encrypted files cannot be accessed without a decryption key.

A lot of work has been done to analyze Eternalblue and Wannacry. Most of the papers on Wannacry focused on the dynamical analysis. D.Y. KAO et al. analyzed Wannacry dynamically, from the aspects of process name, Registry, file system, and Network activity, respectively. They also applied the features to these aspects to create Yarra rules for pattern-matching detection [9]. Qian et al. also dynamically analyzed Wannacry for testing the performance of an automatic dynamical analysis tool [7]. As to the static analysis, D.Y. KAO et al. provided a detailed analysis based on different phases. Critical files and strings participating in those phases were highlighted [10]. Hirokazu statically analyzed Wannacry based on the Eternalblue and DoublePulsar modules. He also applied the discoveries into Snort rules to defend future network attacks based on those two modules [11].

Even though those works helped investigate the working mechanism of Wannacry and Eternalblue, there lacks the study on the comparison between Wannacry's Eternalblue module and the original Eternalblue module. To bridge this research gap, we proceed with our analysis by first studying the exploit's working mechanism that applied in *Wannacry*. Then we use code analytic tools and network capture tools to compare Wannacry's Eternalblue module and the original Eternalblue module. After a detailed study, we find that the exploit utilized in Wannacry shares a very similar pattern to the original exploit, which can be used as features for signature extraction.

The remainder of this paper is organized as follows. In section 2, the original Eternalblue module's working mechanism is introduced. Section 3 analyzes the Eternlablue module in *Wannacry* and Section 4 concludes this paper.

2 Eternalblue's working mechanism in Metasploit

Assume we have two computers: an attacking machine and a victim machine. At the very beginning, Eternalblue is a piece of code on the attacking machine. Once executed, it will send multiple SMB (Server Message Block) requests to the victim machine through the SMB protocol. As a result, the victim machine must respond to these requests. In this SMB communication phase, the attacking machine plays the role of client and the victim machine plays the role of server, which is the reason we refer them to attacking machine and victim machine respectively. Among these SMB requests, the Transaction SMB commands are essential because they are utilized to tamper the data on the server (victim) with a buffer overflow bug, which further leads to the execution of the ransomworm on the victim machine.

Hence, prior to introducing details of the working mechanism of Eternalblue, in this section we will firstly introduce the normal usage of Transaction SMB. According to MSDN [3], Transaction SMB commands enable the client to access advanced features on the server. Specifically, the three transaction messages are:

- SMB_COM_TRANSACTION (or *Trans*),
- SMB_COM_TRANSACTION2 (or *Trans2*),
- SMB_COM_NT_TRANSACT (or *NT Trans*).

It is also noted in [4] that, SMB_COM_NT_TRANSACT subcommands enable the transfer of very large data chunks. And SMB_COM_TRANSACTION2 subcommands provide richer file system services such as allowing clients to set and retrieve Extended Attribute key/value pairs, to make use of long file names, and to perform directory searches, etc.

The information above summarizes the legitimate usage of the Transaction SMB commands. However, in Eternalblue, they are not applied for the original legitimate purposes, but for a buffer overflow bug. Specifically, when responding to these crafted requests, the server will convert the payload contained in these requesting Transaction SMB packets, i.e., the original Os2Fea [1] list, to the currently used NtFea [1] format (result type), as Os2Fea (the original type) is outdated [12]. The NtFea list will be stored into a *result list buffer*. This process is also referred to the conversion process. Under some mild conditions, the server can be fooled by allocating a result list buffer smaller than the NtFea list to be stored. Thus the NtFea list can overwrite the next buffer. And the original list with a specific length will satisfy such mild conditions.

In more detail, the “next buffer” is a Srvnet.sys [1] buffer, which is allocated on the server for the attacker’s SMB request. Once allocated, this buffer will wait for another data package to be sent to the server. There are two parameters in the header of this Srvnet.sys buffer: one decides where to map the data package on this server upon receiving the data package and another decides what function to execute when the Srvnet connection is disconnected. So if these two parameters are modified to the same address, the payload will be mapped and be executed upon closing the connection.

To trigger the overflow of Srvnet.sys and thus inject the malicious codes to the victim, Eternalblue covers three essential steps: *crafting original list*, *buffer grooming*, and *sending the payload*. We proceed our discussion of Eternalblue by first showing a high-level description of the three steps, then dive into details of each step.

From a high-level point of view, in step *crafting original list*, an original list is crafted. In the second step, multiple grooming packages are sent in a deliberate order which changes the server’s buffer status to a point that is vulnerable to overflow. Then, sending the complete original list results in the overflow. In the final step, the payload is sent to the server’s Srvnet.sys buffer. Because of the overflow, the payload can be mapped to the desired location and executed upon closing the connection.

2.1 Crafting original list

To understand crafting of the original list, we firstly recall the normal conversion process on the server machine.

As shown in Algorithm 1, in the Os2Fea format, there is a parameter *ULONG SizeOfList* prior to the actual records describing the total bytes of the original list.

The server’s legitimate conversion process is shown in Algorithm 2. In step *Compute S1*, the algorithm will go through the original list and discard the records that exceed the boundary set by SizeOfList, and the remaining original list (Os2Fea) size is S2, as shown in Fig. 1. S1 is the result list (NtFea) size corresponding to the remaining original list with size S2. Hence, in the third line of Algorithm 2, the server allocates the result list buffer with size S1.

Back into the second line of Algorithm 2, S2 needs to be assigned to the original list's parameter SizeOfList, as this parameter will be used later in the while loop. In the *while* statement of Algorithm 2, the server calls a subfunction repeatedly to convert the original list block by block and stores the result list into this result list buffer. The number of the loop is determined by the SizeOfList's value. The original_list_initial_address in this algorithm points to the beginning of the original list. It is later assigned to variable *Current_pointer*, which will increase after each iteration.

Above is the server's conversion process. The bug occurs when assigning S2 to the original list's parameter SizeOfList if the SizeOfList is no less than 2^{16} (0x10000 in hexadecimal) and the actual original list's record exceeds the boundary set by SizeOfList [13], like shown in Fig. 1.

In more detail, when parsing S2 to SizeOfList, only the LOWORD (low-order word) bytes of the DWORD (double word) variable SizeOfList is updated because of a wrong casting instruction. Hence the End_pointer in Algorithm 2 will be miscalculated, which leads to an unexpected conversion time. This corresponds to a different time to execute the *while* statement in Algorithm 2. For example, as occurred in this Eternalblue exploit, the SizeOfList is initiated with 0x10000. After discarding, the remaining original list size S2 is 0xff5d. However, when executing the second line of Algorithm 2 (i.e., assigning S2 to SizeOfList), only the LOWORD bytes of SizeOfList is updated, which turns SizeOfList from 0x10000 to 0x1ff5d rather than 0xff5d, hence enlarge this SizeOfList.

Algorithm 1 Os2Fea list structure

```
Struct Os2FeaList{
ULONG SizeOfList
UCHAR Os2FeaList[SizeOfList − 4]
}
```

Algorithm 2 The legitimate server's conversion process

```
Compute S1
S2 assigned to SizeOfList
Allocate buffer (result list buffer) with size S1
End_pointer = original_list_initial_address + SizeOfList
Current_pointer = original_list_initial_address
while Current_pointer<End_pointer do
  convert(Current_pointer)
  Current_pointer+ = each_record_size
end while
```

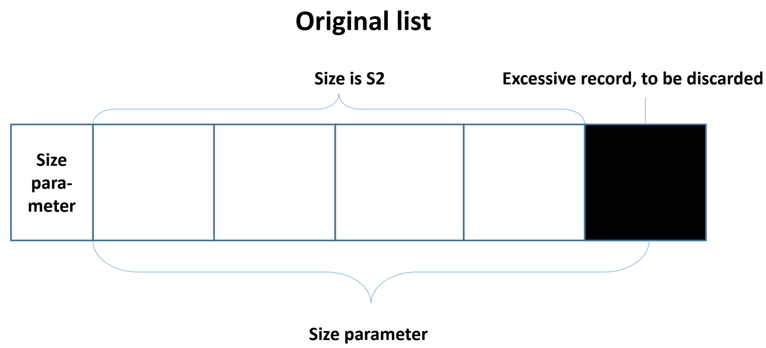

Fig. 1. Server discards the out-of-boundary records and calculates the result list size (S1) based on the remaining records

Based on the discussion above, the crafted list is as follows. The forged original list is of the Os2Fea type, its parameter SizeOfList is with value 2^{16} (0x10000 in hexadecimal), followed by a list of Os2Fea data, as demonstrated in Algorithm 1. There are 607 pieces of data included in this crafted list and garbage data at the end which confines the request packet to a particular size. The first 605 pieces of records are empty, the 606th record is not empty and can be filled with arbitrary data of a certain length. The 607th record contains the fake Srvnet.sys header and this 607th record exceeds the boundary set by SizeOfList [12]. As analog in Fig. 1, the 607th record is the black section, followed by some garbage data. After discarding, only the first 606 records should be converted.

When converting this crafted list on the server, as demonstrated in Algorithm 2, after discarding, S1 and S2 are calculated representing the first 606 records of the result list and the original list. Then, the SizeOfList should be assigned to S2 but in fact assigned to an enlarged value because of the wrong casting (assigned 0x1ff5d rather than 0xff5d). Hence the *End_pointer* is also enlarged. Afterwards, the result list buffer that can only store the first 606 result records will be allocated. Later, the conversion begins, and the loop will be executed for extra times as the *End_pointer* is enlarged. Hence the server will convert and store 607 records in the buffer for 606 records. This is the reason why this 607th record shall be crafted with a forged Srvnet.sys buffer header and the preceding records can be filled with arbitrary data of a certain size.

As mentioned earlier, once the Srvnet.sys buffer is allocated on the server, it waits for another data packet. There are two critical fields in the Srvnet.sys buffer header for processing the data packet: one is called *memory descriptor list* (MDL) which points to a virtual address that the data package shall be mapped to once received the other one is called *pSrvNetWskStruct*. It points to a function which shall be called when the Srvnet connection is closed. Therefore, overwriting these two fields with the same address can make the server map the shellcode to the desired location and execute the shellcode after closing the Srvnet connection.

Finally, it should be noted that the crafted data in the 607th record should be differentiated from the shellcode since the crafted 607th record is used to overwrite the Srvnet.sys buffer's header, which paves the way for sending the shellcode. The sending process of the shellcode is to be discussed later.

2.2 Buffer grooming

We have discussed how the crafted list will lead to the buffer overflow in Srvnet.sys. If the buffer Srvnet.sys is not allocated exactly after the result list buffer, the attack fails. This *buffer grooming* process aims to improve the success rate of overflowing the Srvnet.sys buffer. Table. 1 shows all the grooming packages sent by Eternalblue in timeline. We have validated the packets by analyzing the packets sent by the samples from 2 sources [14] and [8]. The ultimate goal of the grooming procedure is to make the server allocate a *Srvnet.sys buffer* immediately following the *result list buffer*. Only when this goal is achieved, the excessive data from the *result list buffer* can overwrite the *Srvnet.sys buffer*'s header later. The order of the packages sent in Table. 1 can increase the possibility of achieving our ultimate goal. However, the proof is complicated and out of the scope of this paper.

The following paragraphs introduce the packages sent in each step listed in Table. 1. To validate the buffer grooming process, we reproduce the spreading process in a virtual environment and check the captured network packages listed in Table. 1. The samples are created based on scripture on the exploit-db website [14] and Metasploit Eternalblue module [8]. The baseline of the virtual environment and experiment tools are shown below:

- Virtual Machine: VMWare Workstation
- Client (attacker) machine OS: Windows 7 x64 SP1
- Client (attacker) machine IP address: 10.10.10.151
- Server (victim) machine OS: Windows 7 x64 SP1
- Server (victim) machine IP address: 10.10.10.152
- Analysis tools: Wireshark

Firstly the exploit from the client (attacker) machine establishes a connection and determines the target operating system's version and architecture based on the SMB and DCE/RPC (Distributed Computing Environment / Remote Procedure Calls [5]) reply, respectively. Figure. 2 shows the server (victim) machine's buffer initial status before receiving any packages from the client (attacker) machine.

Then the exploit sends the original list to the target machine through connection No.1. However, the legitimate usage of the Transaction SMB request is to send Trans2 Secondary Request after Trans2 Request or to send the NT Trans Secondary Request after the NT Trans Request. Here in this exploit, the purpose of sending Trans2 Secondary Requests packets after the initial NT Trans Request is to utilize another data parsing bug, which permits the attacker to send the payload in a Trans2 request that is bigger than its limit, e.g. 0xffff [12].

Table 1. Eternalblue package sent in timeline

No.	Type	Description
1	Srv	Anonymous login and IPC$ tree connect, then send the crafted original list except the last segment to the server through an NT Trans Request and multiple Trans2 Secondary Requests. An Echo package is followed to ensure the list was sent successfully.
2	1st reserve	Send malformed Negotiate Protocol Request and Session Setup AndX Request to reserve buffer (0x10000 bytes) with size smaller than the result list buffer in NonPagedPool on the server.
3-15	Srvnet	Send multiple TCP packages to establish Srvnet connections which fill up the slot before the result list buffer.
16	2nd reserve	Send malformed Negotiate Protocol Request and Session Setup AndX Request to reserve buffer (0x11000) slightely bigger than the result list buffer. This reserved buffer serves as a place holder for the result list buffer.
2	1st reserve	Send a FIN TCP package to free the 1st reserved buffer.
17-22	Srvnet	Send TCP packages to establish extra Srvnet connections. One of them is expected to be allocated next to the 2nd reserved buffer.
16	2nd reserve	Send a FIN TCP package to free the 2nd reserved buffer.
1	Srv	Send the last segment of the original list through a Trans2 Secondary Request. So the Srv.sys will convert the list. To store the result list with size 0x10fe8 (S1), the server allocates 0x11000 bytes. Because of Windows memory's last-in-first-out working fasion, the 2nd reserved buffer just being freed should be allocated here.
3-15 and 17-22	Srvnet	Send the shellcode through Mutiple TCP packages. The overflow ensures the shellcode be mapped to a desired location. Then close the connections.

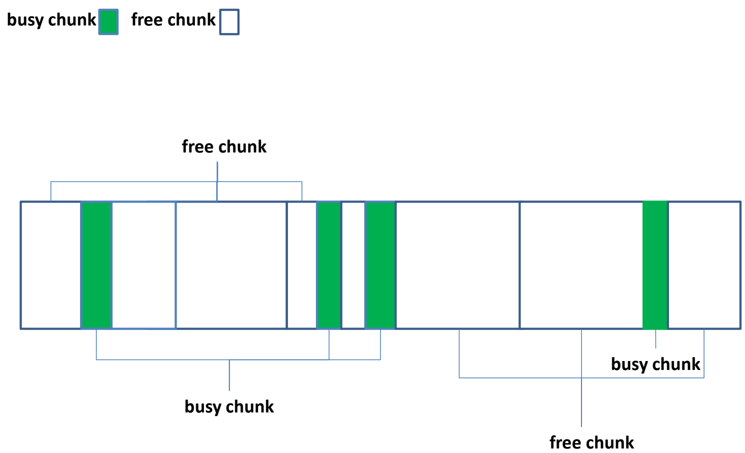

Fig. 2. Server machine's buffer initial status

To ensure the original buffer is received correctly by the target machine, the exploit on the client machine sends an echo package to the server machine. After receiving those packages, the server machine's buffer status changes to the one depicted in Fig. 3.

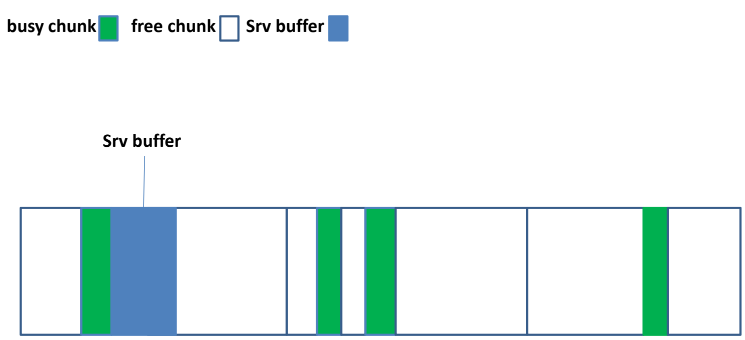

Fig. 3. Server machine's updated buffer state

Then in connection No.2, to reserve a buffer chunk which is used for grooming, another request is sent from the client to the server. After receiving the request, the server's buffer status is updated as shown in Fig. 4.

Next, in order to keep grooming the buffer, as in connection No.3-15 in Table. 1, multiple Srvnet requests are sent to allocate multiple Srvnet.sys buffer chunks on the server. This is the first series of the Srvnet request packages which fill up the slot before the second reserved buffer. Fig. 5 demonstrates the server's

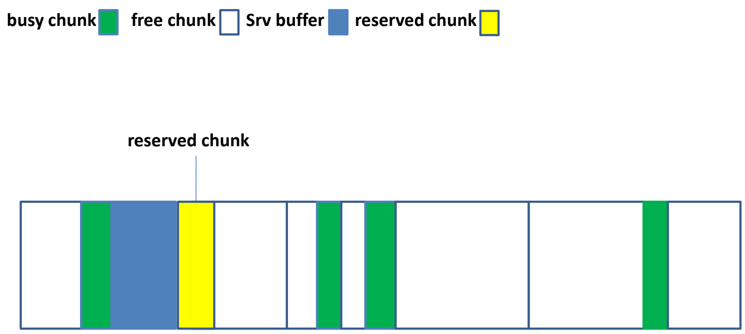

Fig. 4. Server machine's updated buffer state

buffer status after receiving these Srvnet requests. Srvnet connections in this step increase the probability that the Srvnet buffer allocated in connections No.17-22 be allocated immediately following the *result list buffer* because connections No.3-15 fill up the slot between the two reserved Srv buffer (connection 2 and connection 16).

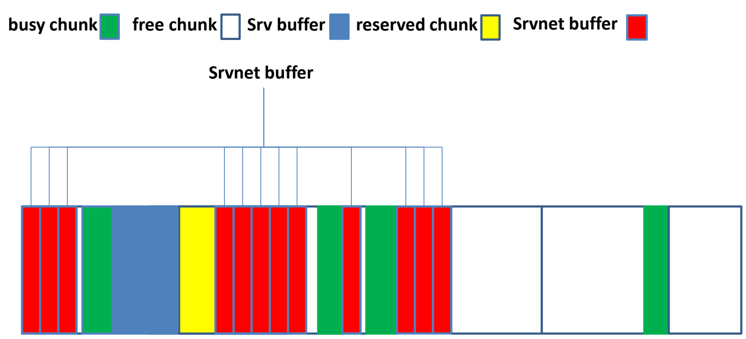

Fig. 5. Server machine's updated buffer state

As in connection No.16 in Table. 1, the second reserving buffer chunk is reserved as a placeholder (to be replaced with the *result list buffer* later). Afterwards, the first reserved buffer chunk through connection No.2 shall be freed. The first and second buffer reserving packets also utilize a bug by setting special parameters in the request to make the large NonPagedPool allocation [12], which is much greater than it is permitted to. After these two steps, the server's buffer status changes to the one depicted in Fig. 6.

Next, in connection No.17-22, extra Srvnet request packages are sent to the server. It is expected that one Srvnet.sys buffer allocated by these requests can be immediately after the *result list buffer*, hence the overflow in the *result list*

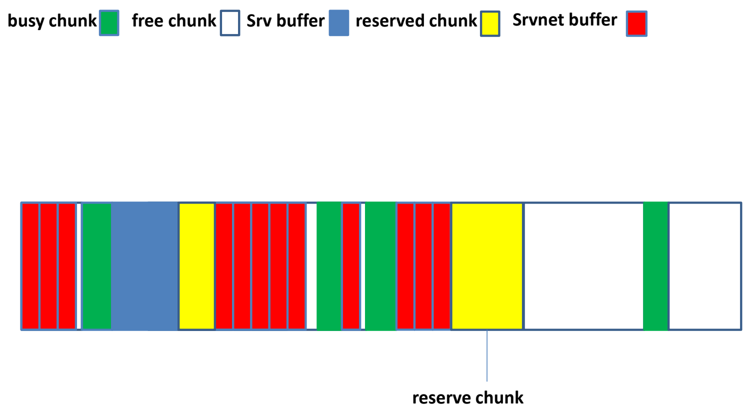

Fig. 6. Server machine's updated buffer state

buffer can overwrite the following Srvnet.sys buffer's header. Figure. 7 shows the server buffer status after receiving the extra Srvnet requests.

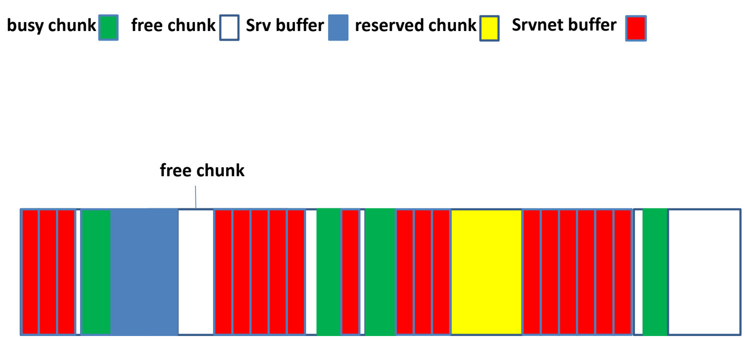

Fig. 7. Server machine's updated buffer state

In connection No.16, the second reserved buffer is freed. And in connection No.1, the last segment of the original list is sent to the server, making the system start the conversion.

To start the conversion process, the server tries to allocate a *result list buffer* with size S1. Since the second reserved buffer, which has the size slightly greater than the result list, is just freed, Windows memory's last-in-first-out working fashion guarantees this buffer be allocated as the *result list buffer*. During the conversion process, as introduced before, the data in the result buffer can overwrite the following Srvnet.sys buffer. The server buffer status changes to the one shown in Fig. 8.

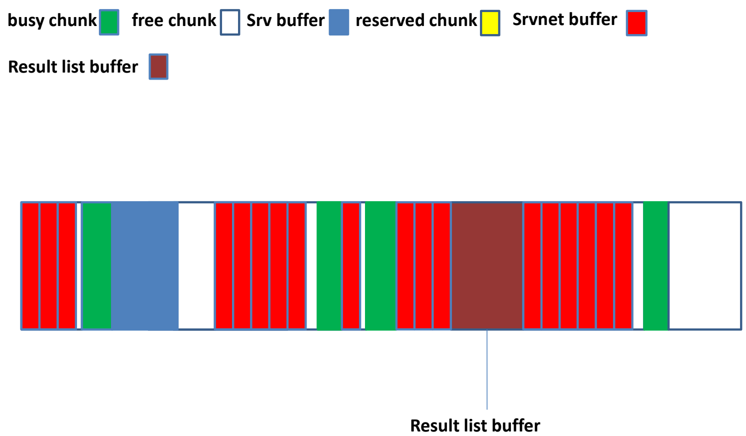

Fig. 8. Server machine's updated buffer state

Finally, the exploit sends the shellcode through each of the previously established Srvnet connections (i.e, connection No.3-15, and 17-22).

2.3 Sending the shellcode

The payload is a piece of executable code that is sent after the target machine is penetrated. Once the payload is executed, the attacker can leverage the vulnerability and do whatever he wants to do. In previous discussion, we have explained that by sending the grooming packages, multiple Srvnet connections are established and one of these corresponding Srvnet.sys buffes is expected to be overflown. According to the Check Point Reseach paper [1], after the Srvnet connections are established, these connections wait for another data packages and upon receiving the data packet, it will map the data according to the parameter *pMdl* contained in the Srvnet buffer's header. Also, upon closing the Srvnet connection, the function pointed by *HandlerFunction* in the *pSrvnetWskStruct* will be called. The Srvnet.sys buffer has a structure as shown in Algorithm 3.

In this section, we have introduced the *shellcode sending process*. However, the detailed code analysis on the shellcode is not discussed in this paper as it is beyond the scope of this paper.

3 Code analysis

As described in Section 1, *Wannacry* utilizes the famous exploit Eternalblue in its spreading process. In this section, we provide a static analysis on Wannacry's exploit module to investigate how this exploit is utilized. The baseline of the analysis tools are shown below:

- *Wannacry* SHA256 hash: 24d004a104d4d54034dbcffc2a4b19a11f39008a575aa614ea04703480b1022c

Algorithm 3 Srvnet.sys buffer structure

$Struct\ Srvnet\_header\{$
......
$MDL * pMdl1$
......
$Srvnet\_receive * pSrvnetWskStruct\{$
......
$PVOIDHandleFunction$
$\}$
......
$\}$

– Static analysis tool: IDA 6.8

3.1 Wannacry

Wannacry creates local network spreading threads and Internet spreading threads to propagate through the network. Both threads use the same exploit Eternalblue to infect other systems.

In the local network spreading process, Wannacry creates a target IP address table and tries to attack the potential victims in the table exhaustively. In the Internet spreading process, Wannacry generates a random IP address and tries to attack the system sharing the same network segment. Like Eternalblue, the spreading process in Wannacry also consists of 3 essential steps: *crafting original list*, *buffer grooming* and *sending the payload*. Table. 2 depicts the summery of Wannacry's package capture after we analyzed the network traffic during the infection. This table describes almost the same process as shown in Table. 1, except for several differences. Even though the general process described in Table. 2 is similar to the process given in Table 1, some of the packets are not introduced in Table. 1, as they are unique in Wannacry.

Through the static analysis of Wannacry by using IDA 6.8, we discovered the function beginning at offset 0x00401D80 crafts the fake original list and prepares the grooming packages, which is discussed in the *crafting original list* and *buffer grooming* steps. These fake original list, grooming packages, and the shellcode mentioned above are embedded into the ransomworm by the ransomworm author. During the runtime, they are extracted and pasted into a buffer chunk in the same order as listed in Table 1. Then data in this buffer chunk will be sent later to the server, which spreads the ransomworm and executes the ransomworm on the server. All the data is in plain-text format and is barely different from the Metasploit's exploit. We will discuss the particulars in the following paragraphs.

Prior to preparing the grooming packages and the shellcode, Wanancry sends a PeekNamePipe package and a Trans2 Request package to detect the existence of MS17_010 and backdoor Doublepulsar respectively [9] as in Fig. 9 and Fig. 10. As step 1 listed in Table. 2, the PeekNamePipe package data is embedded

Table 2. Summery of package capture of Wannacry

Step	Attempt	Packages
1	Detect the existence of MS17_010 and DoublePulsar.	PeekNamedPipe Request and Trans2 Request
2	Send the original list except the last segment.	A NT Trans Request and multiple Trans2 Secondary Requests
3	Ensure the package in last step were sent successfully.	Echo Request
4	Reserve the first buffer.	Negotiate Protocol Request and Session Setup Andx Request
5	Reserve Srvnet.sys buffers.	Multiple TCP packages
6	Reserve the second buffer.	Negotiate Protocol Request and Session Setup Andx Request
7	Free the first reserved buffer.	A FIN TCP package
8	Reserve extra Srvnet.sys buffers.	Multiple TCP packages
9	Ensure the packages sent in last step were sent successfully.	Echo Request
10	Free the second reserved buffer.	A FIN TCP package
11	Send the last segment of the original list.	A Trans2 Secondary Request
12	Send the shellcode.	Mutiple TCP packages

into the ransomworm, as depicted in Fig. 9. It is used when the ransomworm needs to send it (by instruction *call send* at offset 00401AFE). After sending this package, the ransomworm waits for the response from the server by calling the *recv* function at offset 00401B15. If the data in the response package equal to *STATUS_INSUFF_SERVER_RESOURCES* (*0xC0000205* in hexadecimal), that denotes the MS17_010 vulnerability resides on the server. As shown in Fig. 10, the Trans2 package data is also embedded into the ransomworm. The ransomworm waits for the response from the server by the instruction *call recv*. If the *Multiplex ID* field in the response package equals to *0x51*, that denotes the server is infected with Doublepulsar, whereas if the field equals to *0x41*, that denotes the server is not infected.

To establish the connection to the server (victim) machine, the first Negotiate Protocol package is crafted as shown in Fig. 11. The Session Setup AndX and Tree Connect AndX Request packages are crafted in the similar way as depicted in Fig. 12 and Fig. 13.

Next, as in step 2 of Table. 2, an NT Trans Request package, and multiple Trans2 Secondary Request packages containing the crafted Os2Fea list without the last segment are prepared as in Fig. 14, Fig. 15, and Fig. 16.

In step 3 of Table. 2, to ensure the original list is received completely, an echo package is prepared as shown in Fig. 17.

In step 4 of Table. 2, a package which reserves the first buffer chunk on the target is prepared. The corresponding Negotiate and Session Setup Request packages are shown in Fig. 18.

```
.text:00401ADF push    4Eh    ; length
.text:00401AE1 push    offset PeekNamePipe ; PeekNamePipe \
.text:00401AE1                ; source buffer
.text:00401AE6 push    esi    ; socket
 ......        ......
.text:00401AFE call    send   ; smb PeekNamePipe package
.text:00401B03 cmp     eax, 0FFFFFFFFh
.text:00401B06 jz      short loc_401B50
.text:00401B08 push    0      ; flags
.text:00401B0A lea     ecx, [esp+420h+buf]
.text:00401B0E push    400h   ; len
.text:00401B13 push    ecx    ; buf
.text:00401B14 push    esi    ; s
.text:00401B15 call    recv   ; receive response from server
.text:00401B1A cmp     eax, 0FFFFFFFFh
.text:00401B1D jz      short loc_401B50
.text:00401B1F
.text:00401B1F compareWithError: ; STATUS_INSUFF_SERVER_RESOURCES,\
.text:00401B1F cmp     byte ptr [esp+25h], 5 ; C0000205 in hexadecimal
.text:00401B24 jnz     short loc_401B50
```

Fig. 9. PeekNamePipe package

As in step 5 of Table. 2, to continue the buffer grooming process, multiple Srvnet connection requests should be sent to reserve Srvnet.sys buffer chunks on the target system. Figure. 19 shows the crafting of each Srvnet package.

As in step 6 of Table. 2, the second reserving package shall be sent to the server. The crafting process is shown in Fig. 21, including Negotiate Protocol Request and Session Setup AndX Request. In step 7, the first reserving buffer allocated previously shall be freed.

In step 8, extra 5 Srvnet connection requests are crafted as shown in Fig. 22 and will be sent to the target machine to reserve Srvnet.sys buffers. It is expected that one Srvnet.sys buffer allocated in this step is immediately after the second reserved buffer, which will be replaced with the *result list buffer* later. In step 9, an Echo package is crafted as shown in Fig. 23.

In step 10, the second reserved buffer shall be freed and in step 11, the last segment of the crafted original list shall be sent to the target as shown in Fig. 24. Once this last segment is received, the target's conversion process (converting the original list to the result list) begins.

In step 12, multiple packages that contain the same shellcode are crafted as Figs. 25. Later they will be sent through the Srvnet connections established earlier.

```
.text:00401CA7 push    0      ; flags
.text:00401CA9 push    52h    ; length
.text:00401CAB push    offset trans2 ; Trans2 \
.text:00401CAB                ; source buffer
.text:00401CB0 push    esi    ; socket
 ......         ......
.text:00401CC8 call    send   ; trans2 request
.text:00401CCD cmp     eax, 0FFFFFFFFh
.text:00401CD0 jz      loc_401D62
.text:00401CD6 push    0      ; flags
.text:00401CD8 lea     eax, [esp+424h+buf]
.text:00401CDC push    400h   ; len
.text:00401CE1 push    eax    ; buf
.text:00401CE2 push    esi    ; s
.text:00401CE3 call    recv   ; receive response from server
.text:00401CE8 cmp     eax, 0FFFFFFFFh
.text:00401CEB jz      short loc_401D62
.text:00401CED
.text:00401CED compareWithError: ; Multiplex ID, 0x51--infected,\
.text:00401CED cmp     [esp+420h+var_3DE], 51h ;  0x41--not infected
```

Fig. 10. Trans2 package

```
mov     ecx, 22h        ; copy time
mov     esi, offset first_negotiate ; neogotiate \
                        ; package data source address
mov     edi, offset unk_433BB4 ; destination address

......  ......

rep movsd               ; copy from source address \
                        ; to destination address
movsb
```

Fig. 11. Negotiate Protocol package

4 conclusion

The wide application of exploit Eternalblue is a meaningful security incident. The massive infection based on Eternalblue spurs everyone to raise the awareness of patching computers to current status. This paper introduced the underlying mechanism of exploit Eternalblue, as well as the reverse engineering result of

```
mov     ecx, 23h          ; copy time
mov     esi, offset session_setup ; sessionsetup \
                          ; package source address
mov     edi, offset unk_438A04 ; destination address

......  ......

rep movsd                 ; copy from source to \
                          ; destination
```

Fig. 12. Session Setup AndX package

```
mov     ecx, 1Ch          ; copy time
mov     esi, offset treeConnectAndX ; tree connect \
                          ; source address
mov     edi, offset unk_43D854 ; destination address

......  ......

rep movsd                 ; copy from source address \
                          ; to destination address
```

Fig. 13. Tree Connect AndX package

Eternalblue. The code of Eternalblue applied in Wannacry is compared with the original exploit based on the reverse engineering results. The analysis reveals that the exploit Eternalblue is slightly modified when applied in Wannacry. Our work gathered much-known knowledge of Eternalblue to provide readers with a clear picture of this exploit. We have concluded the similarity and the difference of Eternalblue's code in Wannacry. We have also analyzed Notpetya and found the exploit in Notpetya is encrypted and only decrypted while the shellocode is executed. After decrypting it, the Notepetya exploit is almost identical to the original Eternalblue exploit. Due to the length constrains of the paper, the analysis details are not included here. It is possible to extend our work to the code analysis for ransomworm detection.

References

1. EternalBlue — Everything There Is To Know (2017), https://research.checkpoint.com/eternalblue-everything-know/, [Online]; accessed 15 April 2018
2. Investigation: Wannacry Cyber Attack and the NHS (2018), https://www.nao.org.uk/report/investigation-wannacry-cyber-attack-and-the-nhs/, [Online]; accessed 18 March 2018
3. (2019), https://msdn.microsoft.com/en-us/library/ee441928.aspx, [Online]; accessed 16 May 2018
4. (2019), https://msdn.microsoft.com/en-us/library/ee441720.aspx, [Online]; accessed 16 May 2018
5. (2019), https://www.dcerpc.org/documentation/rpc-porting.pdf, [Online]; accessed 18 March 2018

```
.text:00401F60 mov     ecx, 26h ; copy time
.text:00401F65 mov     esi, offset Nt_Trans_S ; NT Trans Request\
.text:00401F65                 ;  data source address
.text:00401F6A mov     edi, offset Nt_Trans_D ; NT Trans Request\
.text:00401F6A                 ; data destination address
.text:00401F6F rep movsd       ; copy from source to destination
.text:00401F71 movsb           ; copy from source to destination
```

Fig. 14. Nt Trans Request

```
.text:0040200D mov     ecx, 134h ; copy time
.text:00402012 mov     esi, offset NT_Trans2_S ; NT Trans 2 Request\
.text:00402012                 ;  data (part1) source address
.text:00402017 mov     edi, offset NT_Trans2_D ; NT Trans 2 Request\
.text:00402017                 ;  data (part1) destination address
.text:0040201C mov     dword_44C330, 0B0000000h
.text:00402026 mov     dword_44C334, 3F43A905h
.text:00402030 mov     word_44C338, ax
.text:00402036 mov     dword_44C33C, edx
.text:0040203C mov     dword_44C340, 4D1h
.text:00402046 rep movsd       ; copy from source to destination
.text:00402048 movsb           ; copy from source to destination
```

Fig. 15. Part of Nt Trans2 Request

6. Alterson, G.: Confronting One of Healthcare's Biggest Challenges: Cyber Risk (2019), https://www.forbes.com/sites/insights-intelai/2019/02/11/confronting-one-of-healthcares-biggest-challenges-cyber-risk/amp/, [Online]; accessed 11 February 2019
7. Chen, Q., Bridges, R.A.: Automated Behavioral Analysis of Malware A Case Study of Wannacry Ransomware (2017), https://arxiv.org/abs/1709.08753, [Online]; accessed 03 April 2018
8. Dillon, S., Jennings, L.: ms17_010_eternalblue (2017), https://github.com/rapid7/metasploit-framework/blob/master/modules/exploits/windows/smb/ms17_010_eternalblue.rb., [Online]; accessed 24 April 2018
9. KAO, D.Y., HSIAO, S.C.: The Dynamic Analysis of WannaCry Ransomware. In: International Conference on Advanced Communications Technology (ICACT) (2018), https://www.dcerpc.org/documentation/rpc-porting.pdf, [Online]; accessed 18 April 2018
10. KAO, D.Y., HSIAO, S.C.: The Static Analysis of Wannacry Ransomware. In: International Conference on Advanced Communications Technology (ICACT) (2018), https://ieeexplore.ieee.org/document/8323679/, [Online]; accessed 18 April 2018
11. Murakami, H.: Reverse Engineering of Wannacry Worm and Anti Exploit Snort Rules (2018), https://www.sans.org/reading-room/whitepapers/malicious/paper/38445, [Online]; accessed 07 July 2018
12. Pradeep Kulkarni, Sameer Patil, P.K., Dolas, A.: Eternalblue: A prominent threat actor of 2017-2018 (2018), https://www.virusbulletin.com/uploads/pdf/magazine/2018/201806-EternalBlue.pdf, [Online]; accessed 23 May 2018
13. Sanchez, W.G.: MS17-010: EternalBlue's Large Non-Paged Pool Overflow in SRV Driver (2017), https://blog.trendmicro.com/trendlabs-security-intelligence/ms17-010-eternalblue, [Online]; accessed 16 March 2018

```
.text:00402086 mov     ecx, 16Dh ; copy time
.text:0040208B mov     esi, offset NT_Trans2_1_S ; NT Trans 2 \
.text:0040208B                 ; Request data (part2) source address
.text:00402090 mov     edi, offset NT_Trans2_1_D ; NT Trans 2 \
.text:00402090                 ; Request data (part2) source address
.text:00402095 mov     dword_451180, 0D5800000h
.text:0040209F mov     dword_451184, 3F8ECD73h
.text:004020A9 mov     word_451188, ax
.text:004020AF mov     dword_45118C, edx
.text:004020B5 mov     dword_451190, ebp
.text:004020BB rep movsd       ; copy from source to destination
```

Fig. 16. Part of Nt Trans2 Request

```
.text:004029D6 mov     ecx, 17h ; copy time
.text:004029DB mov     esi, offset echo_S ; echo data source\
.text:004029DB                 ;  address
.text:004029E0 mov     edi, offset echo_D ; echo data destination\
.text:004029E0                 ;  address
.text:004029E5 mov     dword_4B56E8, 0E0000000h
.text:004029EF mov     dword_4B56EC, 3F11B578h
.text:004029F9 mov     word_4B56F0, ax
.text:004029FF mov     dword_4B56F4, edx
.text:00402A05 mov     dword_4B56F8, 5Fh
.text:00402A0F rep movsd       ; copy from source to destination
.text:00402A11 movsw           ; copy from source to destination
```

Fig. 17. Echo package to check original list well received

14. sleepya: Microsoft Windows Windows 7/2008 R2 (x64) - 'Eternalblue' SMB Remote Code Execution (MS17-010) (2017), https://www.exploit-db.com/exploits/42031/, [Online]; accessed 12 April 2018
15. Thomson, I.: Leaked NSA Point-and-pwn Hack Tools Menace Win2k to Windows 8 (2017), https://www.theregister.co.uk/2017/04/14/latest_shadow_brokers_data_dump/, [Online]; accessed 5 March 2018

```
.text:00402A2C mov     ecx, 22h ; copy time
.text:00402A31 mov     esi, offset negotiate_S ; Negotiate\
.text:00402A31                 ;  Request data source address\
.text:00402A31                 ;  (first reserve)
.text:00402A36 mov     edi, offset negotiate_D ; Negotiate\
.text:00402A36                 ;  Request data destination address\
.text:00402A36                 ;  (first reserve)
 ......        ......
.text:00402A90 rep movsd       ; copy from source to destination
 ......        ......
.text:00402AA9 mov     esi, offset Session_S ; Session Setup \
.text:00402AA9                 ; Request data source address (first reserve)
.text:00402AAE mov     edi, offset Session_D ; Session Setup\
.text:00402AAE                 ;  Request data destination \
.text:00402AAE                 ; address (first reserve)
 ......         ......
.text:00402AF2 rep movsd       ; copy
```

Fig. 18. Grooming package reserves a buffer chunk

```
.text:00402B0A mov     byte ptr [esp+5A8h+var_594+1], al ; SMB2 Rquest\
.text:00402B0A                 ;  data (reserve Srvnet.sys buffer)
.text:00402B0E mov     byte ptr [esp+5A8h+var_594+2], 0FFh ; SMB2 Rquest\
.text:00402B0E                 ;  data (reserve Srvnet.sys buffer)
.text:00402B13 mov     byte ptr [esp+5A8h+var_594+3], 0F7h ; SMB2 Rquest\
.text:00402B13                 ;  data (reserve Srvnet.sys buffer)
.text:00402B18 mov     ecx, [esp+5A8h+var_594] ; SMB2 Request source\
.text:00402B18                 ;  address (part2)
.text:00402B1C mov     byte ptr [esp+5A8h+var_590], 0FEh ; SMB2 Rquest\
.text:00402B1C                 ;  data (reserve Srvnet.sys buffer)
.text:00402B21 mov     byte ptr [esp+5A8h+var_590+1], 53h ; SMB2 Rquest\
.text:00402B21                 ;  data (reserve Srvnet.sys buffer)
.text:00402B26 mov     byte ptr [esp+5A8h+var_590+2], 4Dh ; SMB2 Rquest\
.text:00402B26                 ;  data (reserve Srvnet.sys buffer)
.text:00402B2B mov     byte ptr [esp+5A8h+var_590+3], 42h ; SMB2 Rquest\
.text:00402B2B                 ;  data (reserve Srvnet.sys buffer)
.text:00402B30 mov     esi, [esp+5A8h+var_590] ; SMB2 Request source\
.text:00402B30                 ;  address (part1)
```

Fig. 19. Crafting grooming package which reserves a buffer chunk

```
.text:00402BBE mov     dword_4CB764, ecx ; copy SMB2 Request\
.text:00402BBE                 ;  (part1) to destination address
.text:00402BC4 mov     dword_4CB768, esi ; copy SMB2 Request\
.text:00402BC4                 ;  (part2) to destination address
......         ......
.text:00402C1D mov     dword_4D05B4, ecx ; copy SMB2 Request\
.text:00402C1D                 ;  (part1) to destination address
.text:00402C23 mov     dword_4D05B8, esi ; copy SMB2 Request\
.text:00402C23                 ;  (part2) to destination address
......         ......
.text:00402C57 mov     dword_4D2CDC, ecx ; copy SMB2 Request\
.text:00402C57                 ;  (part1) to destination address
.text:00402C5D mov     dword_4D2CE0, esi ; copy SMB2 Request\
.text:00402C5D                 ;  (part2) to destination address
```

Fig. 20. Copying crafted grooming package to destination address

```
.text:0040303B mov     esi, offset negotiate2_S ; Negotiate \
.text:0040303B                ; Request data source address \
.text:0040303B                ; (second reserve)
.text:00403040 mov     edi, offset negotiate2_D ; Negotiate \
.text:00403040                ; Request data destination address \
.text:00403040                ; (second reserve)

......         ......
.text:00403088 mov     esi, offset Session_S_0 ; Session Setup\
.text:00403088                ;  Request data source address \
.text:00403088                ; (second reserve)
.text:0040308D mov     edi, offset Session_D_0 ; Session Setup \
.text:0040308D                ; Request data destination address \
.text:0040308D                ; (second reserve)
......         ......
.text:004030D1 rep movsd       ; copy from source to destination
.text:004030D3 movsb           ; copy from source to destination
```

Fig. 21. Second package to reserve a buffer chunk

```
.text:004030D4 mov     esi, [esp+5A8h+var_590] ; SMB2 Request \
.text:004030D4                ; source address (part1)
.text:004030D8 mov     ecx, 11h
.text:004030DD mov     edi, 2
.text:004030E2 mov     dword_514E0C, ecx
.text:004030E8 mov     dword_517534, ecx ; index=11
.text:004030EE mov     ecx, [esp+5A8h+var_594] ; SMB2 Request \
.text:004030EE                ; source address (part2)
......         ......
.text:00403195 mov     dword_51753C, ecx ; copy SMB2 Request \
.text:00403195                ; (part1) to destination address
.text:0040319B mov     dword_517540, esi ; copy SMB2 Request \
.text:0040319B                ; (part2) to destination address
......         ......
.text:0040320B mov     dword_51EAB4, ecx ; copy SMB2 Request \
.text:0040320B                ; (part1) to destination address
.text:00403211 mov     dword_51EAB8, esi ; copy SMB2 Request \
.text:00403211                ; (part2) to destination address
......         ......
.text:00403263 mov     dword_523904, ecx ; copy SMB2 Request \
.text:00403263                ; (part1) to destination address
.text:00403269 mov     dword_523908, esi ; copy SMB2 Request \
.text:00403269                ; (part2) to destination address
```

Fig. 22. Crafting extra SMB2 requests which reserve Srvnet.sys buffer

```
.text:004032AC mov     ecx, 17h
.text:004032B1 mov     esi, offset echo1 ; echo package source \
.text:004032B1                ; buffer
.text:004032B6 mov     edi, offset unk_52FCCC ; echo package \
.text:004032B6                ; destination buffer
......         ......
.text:0040335E rep movsd
.text:00403360 movsw
```

Fig. 23. Crafting Echo request

```
.text:00403363 mov     ecx, 177h ; copy time
.text:00403368 mov     esi, offset lastNT_partA_S ; fake original \
.text:00403368                ; list last segment partA source address
.text:0040336D mov     edi, offset lastNT_partA_D ; fake original \
.text:0040336D                ; list last segment partA destination address
......         ......
.text:004033C1 mov     ecx, 16Dh ; copy time
.text:004033C6 mov     esi, offset lastNT_partB_S ; fake original \
.text:004033C6                ; list last segment partB source address
.text:004033CB mov     edi, offset lastNT_partB_D ; fake original \
.text:004033CB                ; list last segment partB destination address

......         ......
.text:004033F8 mov     ecx, 134h ; copy time
.text:004033FD mov     esi, 42B76Ch ; fake original list last segment\
.text:004033FD                ;  partC source address
.text:00403402 mov     edi, offset lastNT_partC_D ; fake original list \
.text:00403402                ; last segment partC destination address
```

Fig. 24. Last segment of original list

```
.text:00403434 mov     ecx, 16Dh ; copy time
.text:00403439 mov     esi, offset shellcode_partA_S ; 1st \
.text:00403439                ; package data (partA) source address
.text:0040343E mov     edi, offset unk_53E7BC ; 1st package data \
.text:0040343E                ; (partA) destination address
......         ......
.text:0040348E rep movsd      ; copy
......         ......
.text:004034BA mov     ecx, 16Dh ; copy time
.text:004034BF mov     esi, offset shellcode_partB_S ; 1st \
.text:004034BF                ; package data (partB) source address
.text:004034C4 mov     edi, offset unk_540EE4 ; 1st package \
.text:004034C4                ; data (partB) destination address
.text:004034C9 rep movsd      ; copy from source to destination
......         ......
.text:004066C1 mov     ecx, 120h ; copy time
.text:004066C6 mov     esi, offset shellcode_partC_S ; 1st \
.text:004066C6                ; package (part C) source address
.text:004066CB mov     edi, offset unk_59695C ; 1st package \
.text:004066CB                ; (part C) destination address

......         ......
.text:004066F6 rep movsd      ; copy from source to destination
```

Fig. 25. Copying part A, B and C of the first shellcode package

```
.text:00403541 mov     ecx, 177h ; copy time
.text:00403546 mov     esi, offset shellcodeV2_partA_S ; 3rd \
.text:00403546                 ; package data (partA) source address
.text:0040354B mov     edi, offset unk_54845C ; 3rd package \
.text:0040354B                 ; data (partA) destination address
......         ......
.text:0040357E rep movsd       ; copy from source to destination
.text:00403580 mov     ecx, 163h ; copy time
.text:00403585 mov     esi, offset shellcodeV2_partB_S ; 3rd \
.text:00403585                 ; package data (partB) source address
.text:0040358A mov     edi, offset unk_54AB84 ; 3rd package data \
.text:0040358A                 ; (partB) destination address
......         ......
.text:004035BD rep movsd       ; copy from source to destination
.text:004035BF mov     ecx, 177h ; copy time

......         .......
.text:00406733 mov     ecx, 120h ; copy time
.text:00406738 mov     esi, offset shellcode_partC_S ; 3rd \
.text:00406738                 ; package (part C) source address
.text:0040673D mov     edi, offset unk_59B7AC ; 3rd package \
.text:0040673D                 ; (part C) destination address
......         ......
.text:0040676C rep movsd       ; copy from source to destination
```

Fig. 26. Preparing part A, B and C of the third shellcode package

```
.text:004036E1 mov     ecx, 177h ; copy time
.text:004036E6 mov     esi, offset shellcodeV2_partA_S ; 6th package data \
.text:004036E6                 ; (partA) source address
.text:004036EB mov     edi, offset unk_556F4C ; 6th package data (partA) \
.text:004036EB                 ; destination address
.text:004036F0 mov     [esp+5A8h+var_58C], 0C1h
.text:004036F5 rep movsd       ; copy from source to destination
......         ......
.text:0040371D mov     [esp+5A8h+var_58B], 0E7h ; package data
.text:00403722 mov     [esp+5A8h+var_58A], 7    ; package data
.text:00403727 mov     [esp+5A8h+var_589], 29h  ; package data
.text:0040372C mov     [esp+5A8h+var_588], 0C7h ; package data
.text:00403731 mov     [esp+5A8h+var_587], bl   ; package data
.text:00403735 mov     [esp+5A8h+var_586], 0F8h ; package data
.text:0040373A mov     [esp+5A8h+var_585], 31h  ; package data
.text:0040373F mov     [esp+5A8h+var_584], 0C9h ; package data
.text:00403744 mov     [esp+5A8h+var_583], 8Ah  ; package data
.text:00403749 mov     [esp+5A8h+var_582], 0Eh  ; package data
.text:0040374E mov     [esp+5A8h+var_581], 80h  ; package data
.text:00403753 mov     [esp+5A8h+var_580], 0F9h ; package data
.text:00403758 mov     [esp+5A8h+var_57F], al   ; package data
.text:0040375C mov     [esp+5A8h+var_57E], 74h  ; package data
.text:00403761 mov     [esp+5A8h+var_57D], 5    ; package data
......         ......
.text:004067E4 mov     ecx, 120h ; copy time
.text:004067E9 mov     esi, offset shellcode_partC_S ; 6th \
.text:004067E9                 ; package (part C) source address
.text:004067EE mov     edi, offset unk_5A2D24 ; 6th package \
.text:004067EE                 ; (part C) destination address
......         ......
.text:0040681D rep movsd       ; copy from source to destination
```

Fig. 27. Part A and B of the sixth shellcode package is prepared