Wiper Malware – A Detection Deep Dive

This post was authored by Christopher Marczewski with contributions from Craig WIlliams

A new piece of wiper malware has received quite a bit of media attention. Despite all the recent press, Cisco's Talos team has historic examples of this type of malware going back to the 1990s. Data **is** the new target, this should not surprise anyone. Recent examples of malware effectively "destroying" data -- putting it out of victims' reach – also include Cryptowall, and Cryptolocker, common ransomware variants delivered by exploit kits and other means.

Wiping systems is also an effective way to cover up malicious activity and make incident response more difficult, such as in the case of the DarkSeoul malware in 2013.

Any company that introduced proper back-up plans in response to recent ransomware like Cryptolocker or Cryptowall should already be protected to a degree against these threats. Mitigation strategies like defense in depth will also help minimize the chance of this malware reaching end systems.

The Deep Dive

Initially we started investigating a sample reported to be associated with the incident to improve detection efficacy. Based off our analysis of e2ecec43da974db02f624ecadc94baf1d21fd1a5c4990c15863bb9929f781a0a we were able to link 0753f8a7ae38fdb830484d0d737f975884499b9335e70b7d22b7d4ab149c01b5 as a nearly identical sample. By the time we reached the network-related functions during our analysis, the relevant IP addresses belonging to the C2 servers were no longer responding back as expected. In order to capture the necessary traffic we had to modify both of the aforementioned disk wiper components. One modification replaced one of the hard-coded C2 server IP addresses with a local address belonging to a decoy VM while changing references to the other hard-coded addresses to point to this local address instead. The other modification simply changed the parameter being passed to an instance of the Sleep() function so debugging efforts wouldn't be put on hold for 45 minutes (the original sample used a 10 minutes sleep).

When we initially examined a rule that was being distributed in the public we were looking for areas where we could improve coverage to better protect our customers. The new Wiper variant is poorly written code and luckily includes very little obfuscation.The author(s) made the mistake of allocating a buffer for the send() function that surpasses the data they wished to include in the payload: a null-terminated opening parentheses byte, the infected host's local IP address, and the first 15 bytes of the host name. This incorrect buffer allocation results in the desired data, in addition to some miscellaneous data already present on the stack (including the 0xFFFFFFFF bytes we alerted on in the first revision of our rule).

Simply running the disk wiper component on different versions of Windows proves the miscellaneous data from the stack that we onced alerted on only applies to beacons being sent from Win XP hosts:

Beacon payload from infected WinXP x86 VM:

Follow TCP Stream

Stream Content

```
00000000  28 00 0a 0b fa b7 57 49  4e 58 50 2d 53 50 33 2d (.....WI NXP-SP3-
00000010  58 38 36 00 ac 71 80 6b  ab 71 ff ff ff ff 63 6b X86..q.k .q....ck
00000020  ab 71 d5 13 40 00 04 00  00 00                   .q..@... ..
```

Beacon payload from infected Win7 x64 VM:

Follow TCP Stream

Stream Content

```
00000000  28 00 0a 0b fa aa 4d 41  52 43 5a 5f 57 37 45 4e (.....MA RCZ_W7EN
00000010  54 5f 53 50 31 00 a9 de  7f 06 fe ff ff ff eb 3b T_SP1... .......;
00000020  60 76 d5 13 40 00 04 00  00 00                   `v..@... ..
```

We have tested part of this hypothesis by running the malware on the same VMs when they had maximum length host names. The resulting beacons continued to limit the hostname bytes in the payload to 15 bytes. To confirm the entire hypothesis, we had to debug and step carefully through the instructions responsible for the data in these beacon payloads. You start by running the disk wiper component alone with the -w flag (which will naturally occur at some point when the disk wiper component is executed and copies itself to host three times). When you hit the following instruction…

```
004012AF call    inet_addr
004012B4 mov     dword_411E3C, eax
004012B9 mov     eax, dword_415F44
004012BE mov     word_411E40, si
004012C5 mov     word_415D88, 7DEh
004012CE mov     word_415D8A, 0Ah
004012D7 mov     word_415D8E, 1Ah
004012E0 mov     word_415D90, 5
004012E9 mov     word_415D92, 1Eh
004012F2 mov     ecx, [eax+4]
004012F5 mov     al, [ecx+1]
004012F8 cmp     al, 6Bh
004012FA jnz     loc_401408
```

…we have to force execution of the alternate jump condition using the debugger to get to the next interesting chunk of assembly:

```
00401300 mov     esi, ds:Sleep
00401306 push    edi
00401307 push    9 -> Was 2700000    ; dwMilliseconds
0040130C call    esi ; Sleep
0040130E mov     ecx, 81h
00401313 xor     eax, eax
00401315 lea     edi, [esp+3A0h+var_396]
00401319 mov     [esp+3A0h+var_398], 0
00401320 rep stosd
00401322 lea     edx, [esp+3A0h+var_398]
00401326 push    offset aA           ; "-a"
0040132B push    edx                 ; wchar_t *
0040132C stosw
0040132E call    _wcscpy
00401333 lea     eax, [esp+3A8h+var_398]
00401337 push    eax
00401338 call    sub_4033A0
0040133D add     esp, 0Ch
00401340 push    1388h               ; dwMilliseconds
00401345 call    esi ; Sleep
```

We eventually arrive to our function call in the code block following the ZF toggle. It's responsible for setting up the necessary socket and sending the beacon payload once a connection has been established:

Later on, we reach a call within the current function (sub_402D10) that is purely responsible for sending the constructed payload:

```
00402D84 cmp     esi, 0FFFFFFFFh
00402D87 jz      short loc_402DA6

00402D89 push    28h              ; int
00402D8B push    offset unk_415D60 ; int
00402D90 push    esi              ; s
00402D91 call    sub_402C80
00402D96 add     esp, 0Ch
00402D99 test    eax, eax
00402D9B jnz     short loc_402DB4
```

When we arrive at the following instruction…

```
00402C9A push    edi
00402C9B lea     edi, [esp+40Ch+var_3FE]
00402C9F shr     ecx, 2
00402CA2 mov     word ptr [esp+40Ch+buf], bx
00402CA7 add     ebx, 2
00402CAA rep movsd
00402CAC mov     ecx, eax
00402CAE and     ecx, 3
00402CB1 rep movsb
```

The code is just about to move 10 double words (ECX is currently 0x0A) from ESI (currently assigned to 0x415D60, which was on the stack prior to calling sub_402C80) to the stack itself (starting at EDI, currently assigned stack pointer 0x12F4CE).

Finally, we reach the call to the Windows function send():

```
00402CAA rep movsd
00402CAC mov     ecx, eax
00402CAE and     ecx, 3
00402CB1 rep movsb
00402CB3 mov     edi, [esp+40Ch+s]
00402CBA xor     esi, esi
00402CBC push    esi             ; flags
00402CBD lea     ecx, [esp+410h+buf]
00402CC1 push    ebx             ; len
00402CC2 push    ecx             ; buf
00402CC3 push    edi             ; s
00402CC4 call    send
00402CC9 test    eax, eax
00402CCB jz      short loc_402CEE
```

Now, at this point you're probably thinking, "Cool. You explained how the payload is ultimately sent out, but how does this explain the random bytes in the payload?". Glad you asked…

Shortly after the instruction where you had to manually toggle the ZF but prior to sub_402D10, there's a call to a function that fetches the name of the infected host:

```
004013D0 call    WSAStartup
004013D5 mov     ecx, 0Ah
004013DA xor     eax, eax
004013DC mov     edi, offset unk_415D60
004013E1 push    offset unk_415D60
004013E6 rep stosd
004013E8 call    sub_402DD0
004013ED add     esp, 4
004013F0 mov     dword_415D84, 4
```

The first block of instructions belonging to this function is shown below:

```
00402DD0 sub     esp, 24h
00402DD3 push    ebx
00402DD4 push    esi
00402DD5 lea     eax, [esp+2Ch+nSize]
00402DD9 push    edi
00402DDA lea     ecx, [esp+30h+Buffer]
00402DDE push    eax             ; nSize
00402DDF push    ecx             ; lpBuffer
00402DE0 mov     [esp+38h+nSize], 20h
00402DE8 call    ds:GetComputerNameA
00402DEE mov     ebx, [esp+30h+arg_0]
00402DF2 mov     ecx, 8
00402DF7 lea     esi, [esp+30h+Buffer]
00402DFB lea     edx, [esp+30h+Buffer]
00402DFF lea     edi, [ebx+4]
00402E02 push    edx             ; name
00402E03 rep movsd
00402E05 call    gethostbyname
00402E0A test    eax, eax
00402E0C jz      short loc_402E17
```

When you get to the following instruction in that block…

```
00402DD0 sub     esp, 24h
00402DD3 push    ebx
00402DD4 push    esi
00402DD5 lea     eax, [esp+2Ch+nSize]
00402DD9 push    edi
00402DDA lea     ecx, [esp+30h+Buffer]
00402DDE push    eax             ; nSize
00402DDF push    ecx             ; lpBuffer
00402DE0 mov     [esp+38h+nSize], 20h
00402DE8 call    ds:GetComputerNameA
00402DEE mov     ebx, [esp+30h+arg_0]
00402DF2 mov     ecx, 8
00402DF7 lea     esi, [esp+30h+Buffer]
00402DFB lea     edx, [esp+30h+Buffer]
00402DFF lea     edi, [ebx+4]
00402E02 push    edx             ; name
00402E03 rep movsd
00402E05 call    gethostbyname
00402E0A test    eax, eax
00402E0C jz      short loc_402E17
```

…ECX = 0x08, ESI = 0x14F8D4, & EDI = 0x415D64. This means that eight double words will be extracted starting at the pointer in ESI and moved to the pointer in EDI. Guess what's on the stack right now?:

```
Stack view
0012F8C0  0012F8D4  Stack[00000444]:0012F8D4
0012F8C4  00415D88  .data:word_415D88
0012F8C8  7C802446  kernel32.dll:kernel32_Sleep
0012F8CC  00000000
0012F8D0  0000000D
0012F8D4  584E4957
0012F8D8  50532D50
0012F8DC  38582D33
0012F8E0  71AC0036  ws2_32.dll:ws2_32_WSAGetServiceClassNameByClassIdW+B5
0012F8E4  71AB6B80  ws2_32.dll:ws2_32_WSAStartup+12B
0012F8E8  FFFFFFFF
0012F8EC  71AB6B63  ws2_32.dll:ws2_32_WSAStartup+10E
0012F8F0  004013D5  sub_401270+165
0012F8F4  004013ED  sub_401270+17D
0012F8F8  00415D60  .data:unk_415D60
0012F8FC  7E419E36  user32.dll:user32_LoadStringW
0012F900  00410144  .data:00410144
```

The data from these eight stack frames will get moved to the .data section, starting at 0x415D64. You'll get the four "prefix bytes" added on once the local IP address is acquired from that same code block via:

```
00402DD0 sub     esp, 24h
00402DD3 push    ebx
00402DD4 push    esi
00402DD5 lea     eax, [esp+2Ch+nSize]
00402DD9 push    edi
00402DDA lea     ecx, [esp+30h+Buffer]
00402DDE push    eax             ; nSize
00402DDF push    ecx             ; lpBuffer
00402DE0 mov     [esp+38h+nSize], 20h
00402DE8 call    ds:GetComputerNameA
00402DEE mov     ebx, [esp+30h+arg_0]
00402DF2 mov     ecx, 8
00402DF7 lea     esi, [esp+30h+Buffer]
00402DFB lea     edx, [esp+30h+Buffer]
00402DFF lea     edi, [ebx+4]
00402E02 push    edx             ; name
00402E03 rep movsd
00402E05 call    gethostbyname
00402E0A test    eax, eax
00402E0C jz      short loc_402E17
```

And, as we've already detailed earlier, 0x2800 will be added as final prefix bytes to the resulting payload. But, we now have another hard-coded element we can alert on in the beacon payload:

```
004013E8 call    sub_402DD0
004013ED add     esp, 4
004013F0 mov     dword_415D84, 4
004013FA call    sub_402D10
```

The third instruction shown above will store 0x04 as a doubleword to 0x415D84, which just happens to be at the very end of the payload currently stored in the .data section.

With this information, we were able to revise accordingly and design the following rule:

```
alert tcp $HOME_NET any -> $EXTERNAL_NET [8000,8080] ( \
        msg:"MALWARE-CNC Win.Trojan.Wiper variant outbound connection"; \
        flow:to_server,established; \
        dsize:42; \
        content:"(|00|"; depth:2; \
        content:"|04 00 00 00|"; within:4; distance:36; \
        metadata:impact_flag red, policy security-ips drop; \
        reference:url,virustotal.com/en/file/e2ecec43da974db02f624ecadc94
        classtype:trojan-activity; \
        sid:32674; rev:2; \
)
```

Click for a text version. It is important to note that sid 32674 will continue to be improved in the future as the malware evolves. This blog applies to the variants we are aware of as of revision 2 of the signature.

This rule will alert on the samples we've analyzed thus far that send these beacons back to their respective C2 servers. What's more, the rule alerts on all of the hard-coded portions of the payload, providing more complete coverage regardless of the major Windows version running on these infected hosts.

Conclusion

We always want to deliver up-to-date detection for the latest threats in the quickest most efficient manner possible. However, the quality of the detection should never be dismissed. The suggested rule we initially landed upon did cover these wiper components when run under select Windows environments, but our team wanted to fully understand the reasoning and justification behind every option of that rule. This helps us ensure we cover the threat to the best extent possible and do so in the most efficient way possible. Once we did we were able to analyze further and release coverage that was more robust for our customers to help prevent further compromises of this magnitude that may just utilize the Wiper malware family.

Coverage

Advanced Malware Protection (AMP) is well suited to detect and block this type of attack.

CWS or WSA web scanning will prevent access to malicious websites and detect the malware used in this attack.

The Network Security protection of IPS and NGFW have up-to-date signatures and will block this threat.

ESA is not applicable for this attack because this threat is not using email.

Tags: APT, malware, security, Talos

Product	Protection
AMP	✓
CWS	✓
ESA	N/A
Network Security	✓
WSA	✓