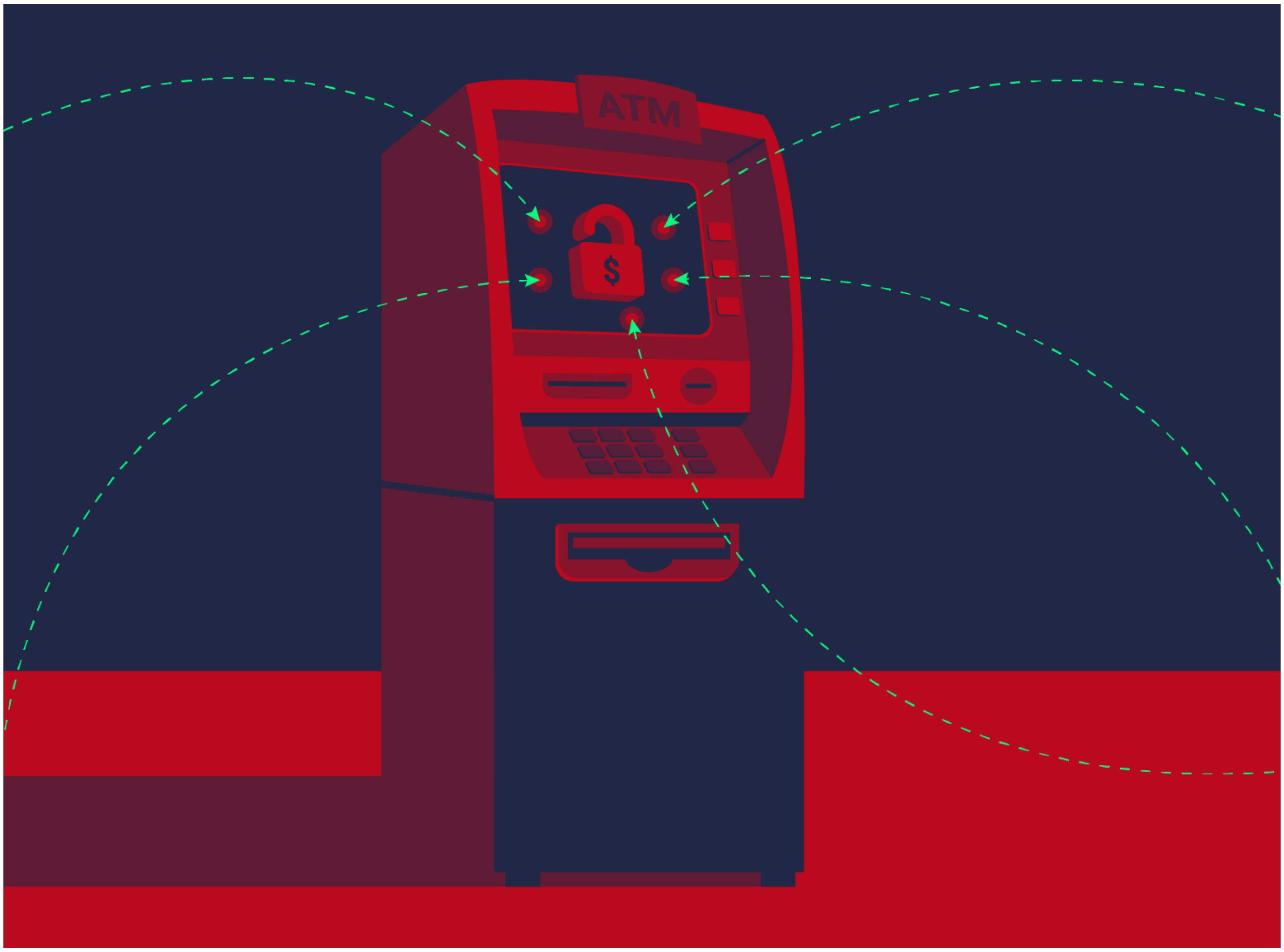
Ploutus is back, targeting Itautec ATMs in Latin America

metabaseq.com/recursos/ploutus-is-back-targeting-itautec-atms-in-latin-america



Ploutus, one of the most sophisticated ATM malware families worldwide, is back with a new variant focused on Latin America. Discovered for the first time in 2013, Ploutus enables criminals to empty ATMs by taking advantage of ATM XFS middleware vulnerabilities via an externally connected device. Since its first discovery, Ploutus has evolved to target various XFS middleware types, focusing on banks across Mexico and Latin America. Previously, researchers have discovered the following variants and associated target middleware:

Year	Variant name	Attacked Middleware
2013	Ploutus	NCR APTRA
2014	Ploutus SMS	NCR APTRA
2017	Ploutus-D	KAL multivendor
2018	Ploutus-D USA (Piolin)	Diebold Agilis

Table 1. Affected XFS Middleware

Ocelot, the Offensive Security research team of Metabase Q, identified a new variant of Ploutus in Latin America. This variant, dubbed Ploutus-I, controls ATMs from the Brazilian vendor Itautec. Itautec has been connected to other major ATM players over the years. In 2013, the Japanese manufacturer, OKI, partnered with Itautec to enter the Brazilian market; subsequently, NCR acquired OKI's IT services and selected software in Brazil in 2019.

Throughout this blog, we will describe the details of this new variant. We will cover the infection methodology, AV bypass technique, obfuscation layers, malware interaction with the crooks, and the XFS control to dispense the money on demand.

Ploutus-I heist operation overview

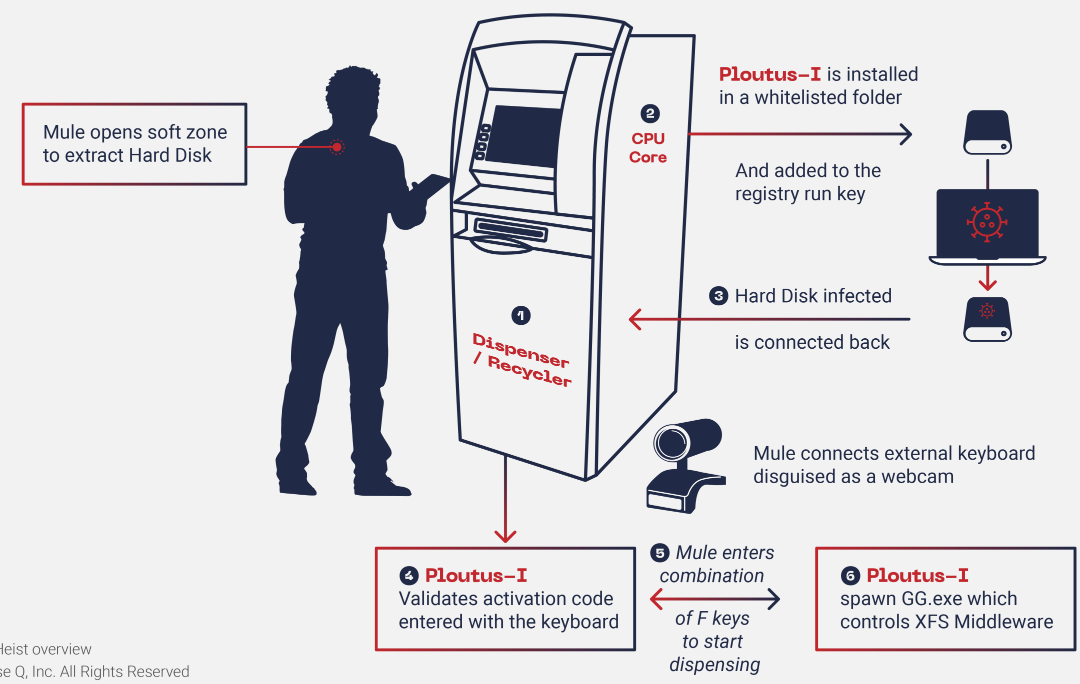


Figure 1. Heist overview

Ploutus-I Installation

At the beginning of the heist, the mule extracts the hard disk from the ATM. The binaries and artifacts (seen below) are copied to the path C:\itautec. Because this path is whitelisted by the Antivirus, the binaries and artifacts can bypass detection.

Rout	Description	MD5
C:\itautec\exe\Itautec.exe	Variante de Ploutus-I	A0DEE20DD90B557BF411DF318740DDC2
C:\itautec\exe\log.dll	Utilidad para Logging	CAE007EF56306F7A8F07FF6678C15837
C:\itautec\exe\GG.exe	Controla el XFS Middleware	33E849EF4604B89BDD905CEAAC9C4E9E
C:\itautec\exe\XFSGG.dll	Controla el XFS Middleware	EAB939B1F5E310400A7DE60F62622B04
C:\itautec\exe\msxfs.dll	XFS APIs	3BDA1500AF49F91045D4BB93272F7352

Table 2. Ploutus-I Installation

Persistence is gained by adding the malware path to the Userinit registry key (see Figure 2), which lists the programs run by Winlogon when a user logs in to the system.

This path is found here:
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\Winlogon\

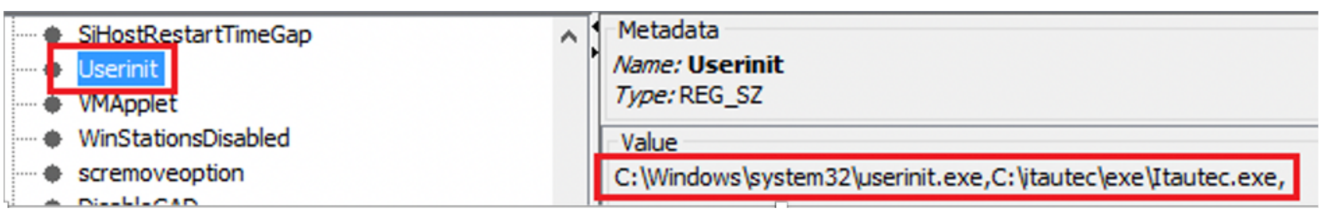


Figure 2. Registry key used for persistence

The Ploutus-I executable is shown as "Itautec Protection Agent," with a compilation time of April 17, 2020.

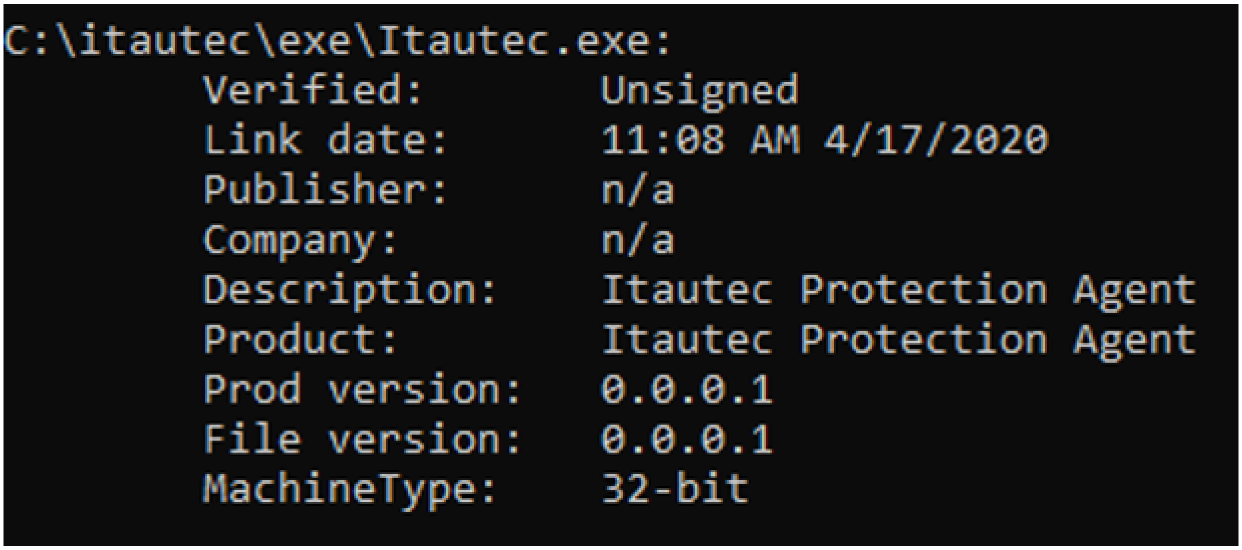


Figure 3. Fake description of Ploutus-I

Deobfuscating Ploutus-I

Every new variant of Ploutus is harder to deobfuscate, and this last version is not the exception. This section is highly technical but essential to share for researchers to improve awareness and ATM security in the future. If you are not interested in the technical details, please skip to the next section.

Ploutus-I has always been written in .NET Framework as a method of further obfuscation to avoid signature-based detection and to make the reverse engineering task very challenging.

Before getting into the deobfuscation details, it is imperative to understand how the execution of .NET managed code(C#, F#, etc.) occurs in memory. For a more detailed explanation, we recommend reading Phrack. For this blog, a minimalistic flow is shown in Figure 4.

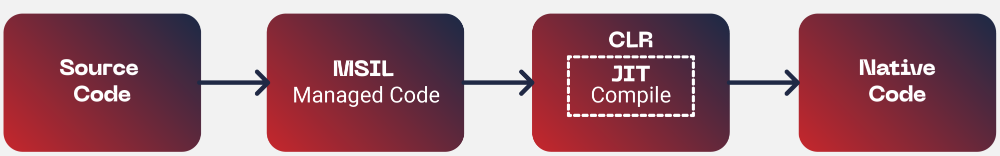


Figure 4. Overview of .NET process execution

In a glimpse, what Ploutus-I obfuscator (Reactor) does is obfuscate the MSIL Managed Code so that the source code cannot be displayed by DnSpy Debugger & Decompiler tool. At runtime, the malicious code is deobfuscated by the malware and then passed to the Just-In-Time (JIT) Compiler to create the native code that ends up been executed by the CPU. How can we recover the source code and understand the inner workings of the malware? Keep reading.

By opening the assembly file Itautec.exe (see Figure 5), we can immediately see the structure of the old variant Ploutus-D. Later in our discovery, we realized that the criminals behind Ploutus-D just added support to control Itautec/OKI XFS Middleware.

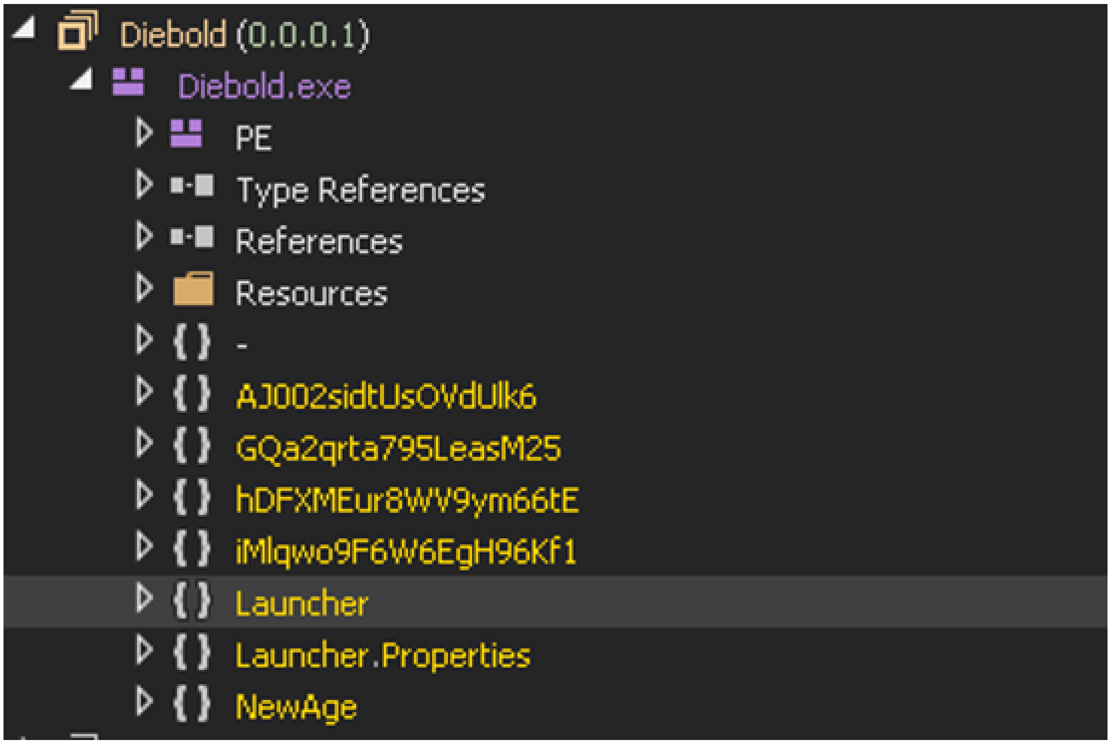


Figure 5. The same template of the previous variant used

Figure 6. All Functions and Class names generated with random names

```
using System;
using System.Runtime.CompilerServices;
using System.Windows.Forms;
using GQa2qrta795LeasM25;

namespace Launcher
{
    // Token: 0x0200000E RID: 14
    internal class Keyboard
    {
        // Token: 0x060000C2 RID: 194 RVA: 0x000035B0 File Offset: 0x000017B0
        [MethodImpl(MethodImplOptions.NoInlining)]
        public static void Read(int Data)
        {
        }

        // Token: 0x060000C3 RID: 195 RVA: 0x000035B8 File Offset: 0x000017B8
        [MethodImpl(MethodImplOptions.NoInlining)]
        public static void StartTheThread(int KeyData)
        {
        }

        // Token: 0x060000C4 RID: 196 RVA: 0x000035C8 File Offset: 0x000017C8
        [MethodImpl(MethodImplOptions.NoInlining)]
        private static void RealStart(int KeyData)
        {
        }

        // Token: 0x060000C5 RID: 197 RVA: 0x000035D8 File Offset: 0x000017D8
        [MethodImpl(MethodImplOptions.NoInlining)]
        private static void RealStart(object KeyData)
        {
        }

        // Token: 0x060000C6 RID: 198 RVA: 0x000035E8 File Offset: 0x000017E8
        [MethodImpl(MethodImplOptions.NoInlining)]
        public Keyboard()
        {
        }

        // Token: 0x060000C7 RID: 199 RVA: 0x000035F8 File Offset: 0x000017F8
        [MethodImpl(MethodImplOptions.NoInlining)]
        internal static void WsktXSgLs6tcY9GkiTo(object A_0)
        {
        }

        // Token: 0x060000C8 RID: 200 RVA: 0x00003600 File Offset: 0x00001800
        [MethodImpl(MethodImplOptions.NoInlining)]
        internal static bool ITEtgdgtXWFZf0O3uGd()
        {
            return true;
        }

        // Token: 0x060000C9 RID: 201 RVA: 0x00003608 File Offset: 0x00001808
        [MethodImpl(MethodImplOptions.NoInlining)]
        internal static Keyboard O1FmgAgg8eDwdpvEX2X()
        {
            return null;
        }
```

Figure 7. No code available inside the functions

Deobfuscation strategy

Before digging into the deobfuscation strategy, it is crucial to understand how Reactor obfuscator hides the malicious code in memory. We can explain this obfuscation by looking at Figure 8, where:

> When a specific MSIL Code Function (let's say the Ploutus-I Keyboard one) is called, JIT is going to call getJIT to get the address of the compileMethod to compile it into Native Code

> However, Ploutus-I already installed a hook in memory redirecting that address to its own malicious one. It then deobfuscates the function and finally calls the original compileMethod to proceed with normal execution.

It is worth mentioning that this process is performed at the memory and only for the function been called at that moment, explaining why there is no visibility of functions in DnSpy.

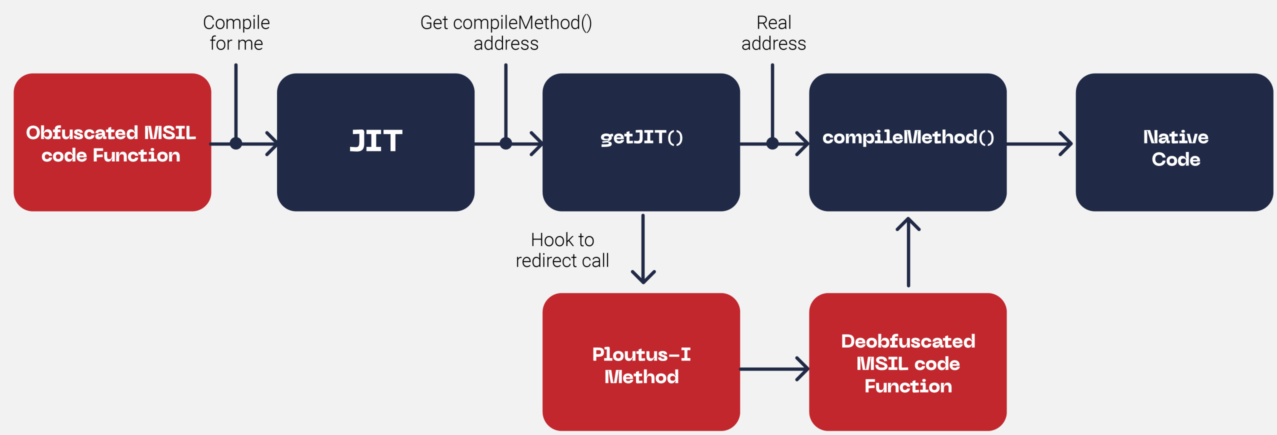


Figure 8. Process of hooking compileMethod function

With this context, our strategy is to set a breakpoint in the original compileMethod in memory, to pivot from there into identifying the function of Ploutus-I controlling the deobfuscation process.

For this, we need to switch to a more advanced tool, the Windbg debugger, with its SOS.dll extension to deal with .NET Managed code.

You can see in Figure 9 that we set a breakpoint at function getJit() (exported by clrjit.dll) because it returns a VTable (array of pointers) where the first value is the address of the compileMethod!

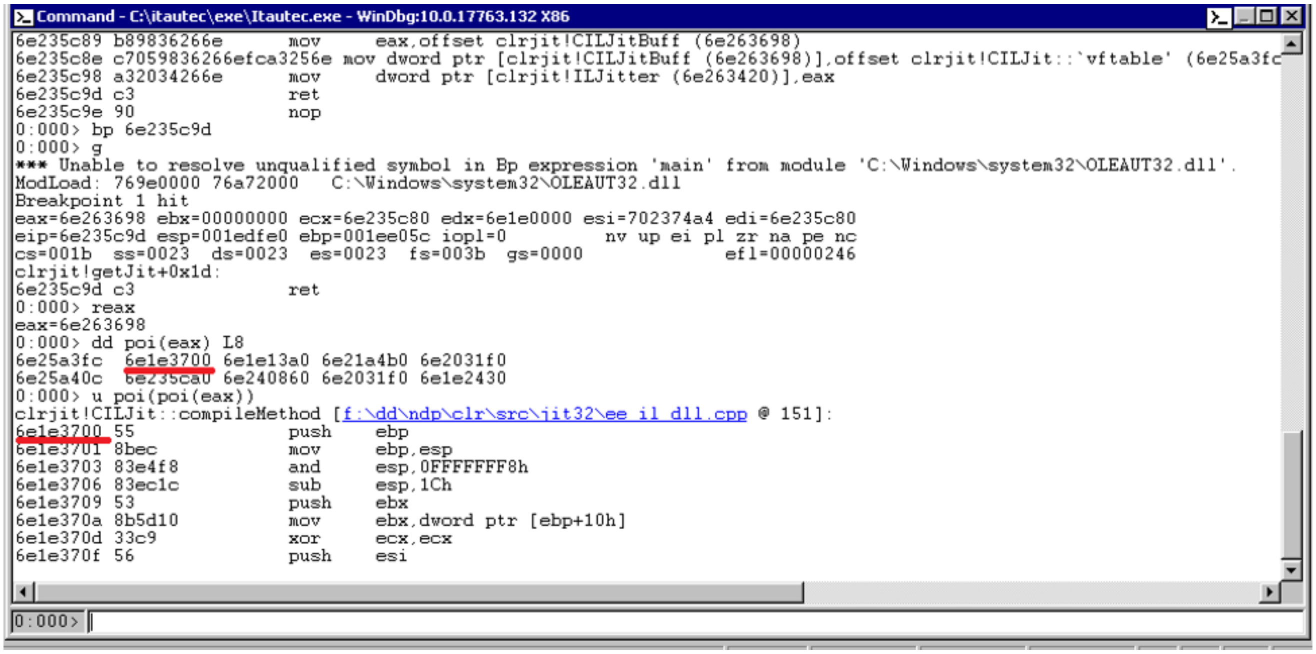


Figure 9. Identifying compileMethod address

Once we set a breakpoint at compileMethod(at 0x6e1e3700) and let the malware run, we can see in Figure 10 when the breakpoint hits. We then use the !CLRStack command to see the stack trace of managed code, and voila! We found the malicious method that redirects the execution when compileMethod is called:

GQa2qrta795LeasM25.vlIg50mEXlJEDAGw36.GyQV7V7HyQ()

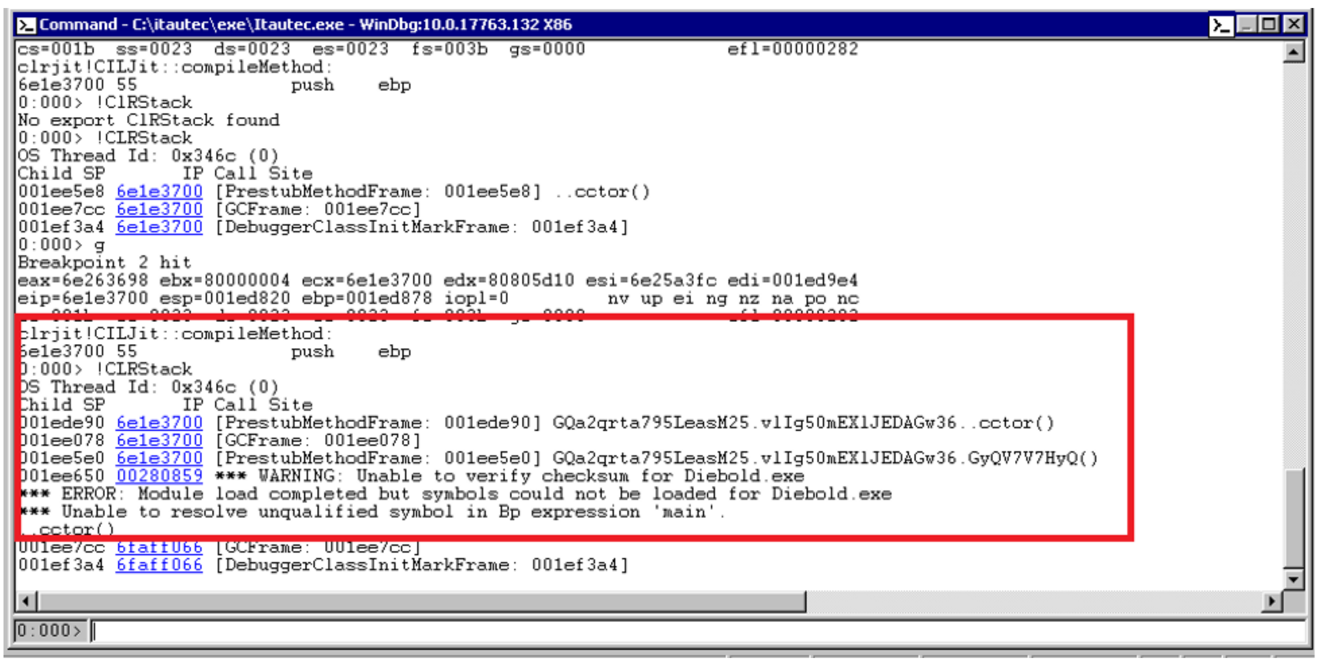


Figure 10. Identifying the malicious method that hooks compileMethod

It is essential to mention that each class's static constructor (.cctor) in the malware code uses this function. This function usage makes sense because, as previously mentioned, every method is going to be deobfuscated in memory before getting compile into native code for execution.

Unfortunately, we are not there yet. In previous versions of Ploutus, the above function would contain the deobfuscation code for us to dump. However, when looking at the function (see Figure 11) in DnSpy, we realized we entered a vast obfuscated function with hundreds of switch cases, spaghetti code, death code, and other tricks, which make it impossible to debug.

```
// GQa2qrta795LeasM25.vlIg50mEXlJEDAGw36
// Token: 0x06000230 RID: 560 RVA: 0x0000A900 File Offset: 0x00008B00
[MethodImpl(MethodImplOptions.NoInlining)]
internal unsafe static void GyQV7V7HyQ()
{
    int num = 534;
    for (;;)
    {
        int num2 = num;
        byte[] array;
        int num3;
        byte[] array2;
        vlIg50mEXlJEDAGw36.KJi0CHPCaJiALAqUfW kji0CHPCaJiALAqUfW;
        int num5;
        int num6;
        int num7;
        byte[] array3;
        int num9;
        byte[] array4;
        long num14;
        IntPtr intPtr;
        byte[] array8;
        byte[] array9;
        byte[] array11;
        int num22;
        byte[] array12;
        uint num25;
        byte[] array13;
        long value;
        int num39;
        string u4;
        IntPtr u3;
        int num41;
        byte[] array14;
        long num42;
        uint num43;
        int num44;
        uint num45;
        vlIg50mEXlJEDAGw36.Vd0ZfZTVwtkACAZdOo vd0ZfZTVwtkACAZdOo;
        IntPtr intPtr3;
        bool j23c7vI0mE;
        vlIg50mEXlJEDAGw36.Vd0ZfZTVwtkACAZdOo vd0ZfZTVwtkACAZdOo2;
        byte[] array17;
        int num49;
        int u6;
        vlIg50mEXlJEDAGw36.ecAMUBgSdNopQW3v1o u7;
        IntPtr u8;
        IntPtr u9;
        vlIg50mEXlJEDAGw36.gBJCUDQY36cbqAjPfV gBJCUDQY36cbqAjPfV;
        int num67;
        int num68;
```

Figure 11. Obfuscated function GyQV7V7HyQ()

We keep digging into the new function. Based on previous versions of Ploutus, we expected some keylogging to be running in the background. Based on prior discoveries, we pressed some F keys and eventually got a new hit at compileMethod (see Figure 12). When we looked at the stack trace, we identified the function that contained the deobfuscated MSIL Code that was about to call compileMethod to get executed!

GQa2qrta795LeasM25.vlIg50mEXlJEDAGw36.NvQ34uZt895nxEhi2FIr()

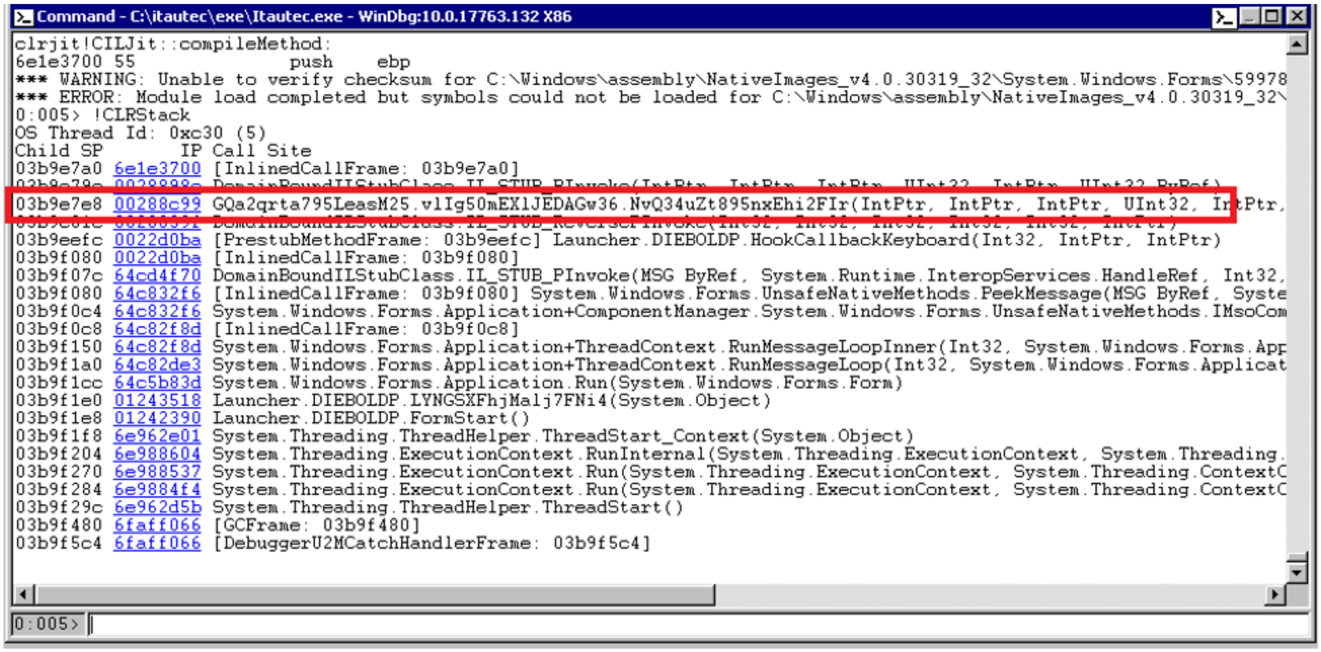


Figure 12. Identifying the function that contains the deobfuscated MSIL Code

By accessing that function, as shown in Figure 13, the deobfuscated MSIL code is passed to the original compileMethod function (line 35). This process is described further in the Phrack article (referenced above). As a result, we receive the second parameter, the CORINFO_METHOD_INFO structure, where we can get the address where the MSIL Code is located and its size (highlighted in yellow):

```
struct CORINFO_METHOD_INFO
{
    CORINFO_METHOD_HANDLE       ftn;
    CORINFO_MODULE_HANDLE       scope;
    BYTE *                      ILCode;
    unsigned                    ILCodeSize;
    unsigned                    maxStack;
    unsigned                    EHcount;
    CorInfoOptions              options;
    CorInfoRegionKind           regionKind;
    CORINFO_SIG_INFO            args;
    CORINFO_SIG_INFO            locals;
};
```

With this information, we can either dump the MSIL Code from memory via DnSpy or directly in Windbg, and we are all set! An excellent tool written by @s4tan deobfuscating a previous variant of Ploutus.

```
internal static uint NvQ34uZt895nxEhi2FIr(IntPtr \u0020, IntPtr \u0020, IntPtr \u0020, [MarshalAs(UnmanagedType.U4)] uint \u
{
    long num;
    if (IntPtr.Size == 4)
    {
        num = (long)Marshal.ReadInt32(\u0020, IntPtr.Size * 2);
    }
    else
    {
        num = Marshal.ReadInt64(\u0020, IntPtr.Size * 2);
    }
    object obj = vlIg50mEXlJEDAGw36.FSpc1m0kWw[num];
    if (obj == null)
    {
        return vlIg50mEXlJEDAGw36.hKjckP8nUU(\u0020, \u0020, \u0020, \u0020, \u0020, ref \u0020);
    }
    vlIg50mEXlJEDAGw36.Vd0ZfZTVwtkACAZdOo vd0ZfZTVwtkACAZdOo = (vlIg50mEXlJEDAGw36.Vd0ZfZTVwtkACAZdOo)obj;
    IntPtr intPtr = Marshal.AllocCoTaskMem(vd0ZfZTVwtkACAZdOo.BLUcm91WRu.Length);
    Marshal.Copy(vd0ZfZTVwtkACAZdOo.BLUcm91WRu, 0, intPtr, vd0ZfZTVwtkACAZdOo.BLUcm91WRu.Length);
    if (vd0ZfZTVwtkACAZdOo.j23c7vI0mE)
    {
        \u0020 = intPtr;
        \u0020 = (uint)vd0ZfZTVwtkACAZdOo.BLUcm91WRu.Length;
        vlIg50mEXlJEDAGw36.X6BVURWu80(\u0020, vd0ZfZTVwtkACAZdOo.BLUcm91WRu.Length, 64, ref vlIg50mEXlJEDAGw36.UPFcimQMXu);
        return 0U;
    }
    Marshal.WriteIntPtr(\u0020, IntPtr.Size * 2, intPtr);
    Marshal.WriteInt32(\u0020, IntPtr.Size * 3, vd0ZfZTVwtkACAZdOo.BLUcm91WRu.Length);
    uint result = 0U;
    if (\u0020 != 216669565U || vlIg50mEXlJEDAGw36.firstrundone)
    {
        result = vlIg50mEXlJEDAGw36.hKjckP8nUU(\u0020, \u0020, \u0020, \u0020, \u0020, ref \u0020);
    }
    else
    {
        vlIg50mEXlJEDAGw36.firstrundone = true;
    }
    return result;
}
```

Figure 13. Call to the original compileMethod function

Now, let's compare the results by looking at the function Launcher.KeyBoard::RealStart() before deobfuscation. We can see it is empty in Figure 14.

```
[MethodImpl(MethodImplOptions.NoInlining)]
private static void RealStart(object KeyData)
{
}
```

Figure 14. Functions **before** been deobfuscated

And then, after the magic happens, we can see in Figure 15, the deobfuscated MSIL Code ready to be analyzed!

```
.method private hidebysig static void  RealStart(object KeyData) cil managed noinlining
{
  // Code size       2174 (0x87e)
  .maxstack  50
  IL_0000:  br.s       IL_0007
  IL_0002:  call        [ERROR: INVALID TOKEN 0x40191C03]
  IL_0007:  ldc.i4     0x43
  IL_000c:  stloc      V_0
  IL_0010:  br         IL_0015
  IL_0015:  ldloc      V_0
  IL_0019:  switch     (
                        IL_06d1,
                        IL_0520,
                        IL_0758,
                        IL_078a,
                        IL_0432,
                        IL_01c7,
                        IL_0248,
                        IL_07eb,
                        IL_03a2,
  IL_012e:  br         IL_06d1
  IL_0133:  call       void Launcher.Launch::LaunchClientTest()
  IL_0138:  ldc.i4     0x30
  IL_013d:  br         IL_0019
  IL_0142:  ldarg.0
  IL_0143:  call       valuetype [System.Windows.Forms]System.Windows.Forms.Keys Launcher.Keyboard::KTsHkpgSWYncEH3cfVa(object)
  IL_0148:  ldc.i4.s   49
  IL_014a:  beq        IL_068c
  IL_014f:  ldc.i4     0x3e
  IL_0154:  br         IL_0019
  IL_0159:  call       void Launcher.Launch::LaunchDieboldDiagnostic()
  IL_015e:  ldc.i4     0x27
  IL_0163:  br         IL_0019
  IL_0168:  call       void Launcher.Keyboard::gDWZpCgUBsggGOhWR0Q()
  IL_016d:  ldc.i4     0x2a
  IL_0172:  call       bool Launcher.Keyboard::ITEtgdgtXWFZf0O3uGd()
  IL_0177:  brtrue     IL_0019
  IL_017c:  pop
  IL_017d:  br         IL_0015
  IL_0182:  ldarg.0
  IL_0183:  call       valuetype [System.Windows.Forms]System.Windows.Forms.Keys Launcher.Keyboard::KTsHkpgSWYncEH3cfVa(object)
  IL_0188:  ldc.i4.s   120
  IL_018a:  beq        IL_068c
  IL_018f:  ldc.i4     0x28
  IL_0194:  br         IL_0019
  IL_0199:  ldarg.0
  IL_019a:  call       valuetype [System.Windows.Forms]System.Windows.Forms.Keys Launcher.Keyboard::KTsHkpgSWYncEH3cfVa(object)
```

Figure 15. Function **after** been deobfuscated

Understanding Ploutus-I Inner workings

With the MSIL Code in our hands, we can understand what is going on with this new variant. The primary function we focused on is Launcher.KeyBoard::RealStart() since it triggers all the actions executed by the malware. It implements a keylogger (already seen before) to intercept all keys and numbers entered by the mule via an external keyboard. It is essential to mention that this variant was successfully executed in the Windows 7 and Windows 10 versions.

Ploutus-I encrypts all its strings. When needing one of them, the malware will call the instruction **ldc.14.s** passing an offset as an argument that will be the pointer into a Unicode byte array decrypted from the resources section at runtime pointing to the plaintext value. For example, in Figure 16, the instruction "ldc.14.s 0x9f0", goes to the offset 0x9f0 and returns the string "F8F1F1". You can see all the strings extracted in the Appendix A section at the end.

```
IL_0741:  ldarg.0
IL_0742:  callvirt   instance void class [mscorlib]System.Collections.Generic.Dictionary`2<int32,int32>::Add(!0,!1)
IL_0747:  ldc.i4.s   119 //F8 key
IL_0749:  bne.un     IL_03a2 //Branch if not equal--> IL_03a2
IL_074e:  ldc.i4     0x1b //case 27 --> IL_0473
IL_0753:  br         IL_0019_swtich

IL_0473:  ldsfld     object NewAge.MemoryData::Command //Load the pressed keys
IL_0478:  ldc.i4     0x9f0 // this is an index for string--> "F8F1F1"
IL_047d:  call       object Launcher.Keyboard::OTBe2ygBotUiJFJAIeE(int32)
IL_0482:  call       void [mscorlib]System.Threading.Monitor::Exit(object) //Releases an exclusive lock on the specified object.
IL_0487:  brfalse    IL_07c8
IL_048c:  ldc.i4     0x1 //case 1 -->IL_0520
IL_0491:  call       class Launcher.Keyboard Launcher.Keyboard::O1FmgAgg8eDwdpvEX2X() // ret keyobject
IL_0496:  brfalse    IL_0019_swtich
IL_049b:  pop
IL_049c:  br         IL_0015
```

Figure 16. Malware validating F keys entered

Following this process, we were able to identify the combinations to trigger specific actions to Ploutus-I, as shown in Figure 17.

```
0x000009F0 -> "F8F1F1"       --> PrintScreen.Windows();
0x00000A00 -> "F8F2F2"       --> Launch.LaunchAgilis();
0x00000A10 -> "F8F3F3"       --> Launch.LaunchXFS();
0x00000A20 -> "F8F1F2F3F4"   --> Launch.LaunchClient();
0x00000A38 -> "F8F4F5"       --> Launch.LaunchClientTest();
0x00000A48 -> "F8F4F4"       --> Launch.Reboot();
0x00000A58 -> "F8F5F5"       --> Launch.LaunchCMD();
0x00000A68 -> "F8F6F6"       --> Launch.LaunchDriver();
0x00000A78 -> "F8F7F7"       --> Launch.LaunchSysAPP();
0x00000A88 -> "F8F9F9"         --> Launch.LaunchKill();
0x00000A98 -> "F8F11F11"     --> Launch.LaunchDelete();
0x00000AAC -> "F8F12F12"     --> Launch.LaunchPE();
0x00000AC0 -> "F8F5F6F7"     --> Launch.LaunchDelete();
0x00000AD4 -> "D"            --> Enter digits
```

Figure 17: Sequence of keys to execute specific actions

Some functions are from the previous version of Ploutus, but still work in this variant. As an example, PrintScreen.Windows() that once the correct combination is received, the screen at Figure 18 is displayed.

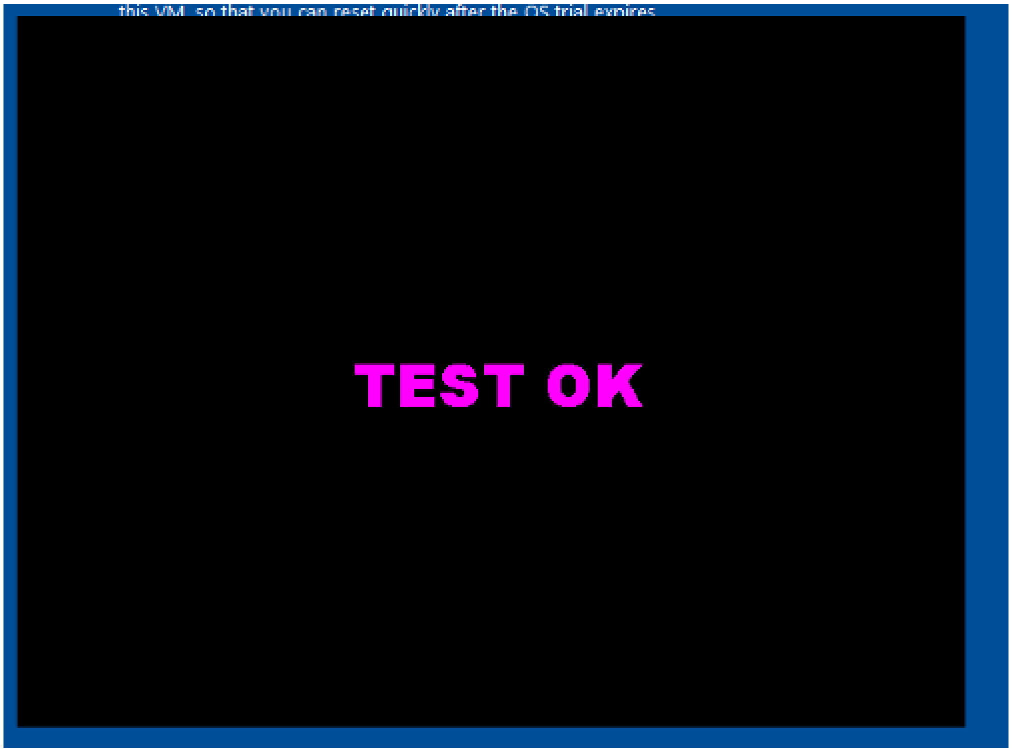


Figure 18. Window displayed by Ploutus-I

Once the combination "F8F1F2F3F4" is entered by the criminals, the Launcher.Launch::LaunchClient() is called as seen at Figure 19.

```
IL_0547:  ldsfld     object NewAge.MemoryData::Command
IL_054c:  ldc.i4     0xa20 // offset "F8F1F2F3F4"
IL_0551:  call       object Launcher.Keyboard::0IBe2ygBotUiJFJAIeE(int32) //returns string
IL_0556:  call       bool Launcher.Keyboard::bWosoPgdBj7WxSPKuKq_monitor_exit(object,object)
IL_055b:  brfalse    IL_02d5
IL_0560:  ldc.i4     0x4 // case 4 --> IL_0432
IL_0565:  call       bool Launcher.Keyboard::ITEtgdgtXWFZf0O3uGd_cmp_null()
IL_056a:  brtrue     IL_0019_swtich
IL_056f:  pop
IL_0570:  br         IL_0015

IL_0432:  call       void Launcher.Launch::LaunchClient() //launch client
IL_0437:  ldc.i4     0x1d //case 29 --> IL_02d5
IL_043c:  br         IL_0019_swtich
```

Figure 19. Call to LaunchClient function

Then inside Launch.LaunchClient() function, we can see the offset 0x218 is used to decrypt a string which ended up being “GG.exe” that eventually is able to control the XFS middleware in the ATM (see Figure 20).

```
IL_094a:  ldc.i4     0x218 //GG.exe index string
IL_094f:  call       object Launcher.Launch::YLA9RBrUOTIUhUhlBn(int32) //get GG.exe
IL_0954:  call       object Launcher.Launch::dGauGqtdTNsp0GZITZY(object)
IL_0959:  stloc.s    V_15
IL_095b:  ldc.i4     0x0
IL_0960:  call       class Launcher.Launch Launcher.Launch::SlPkTdR4fIM7XdI96Y()
IL_0965:  brfalse    IL_0974
IL_096a:  pop
IL_096b:  br         IL_0970
```

Figure 20. “GG.exe” string is decrypted
Finally the binary gets executed but fails in our system since no DLLs are present. See Figure 21.

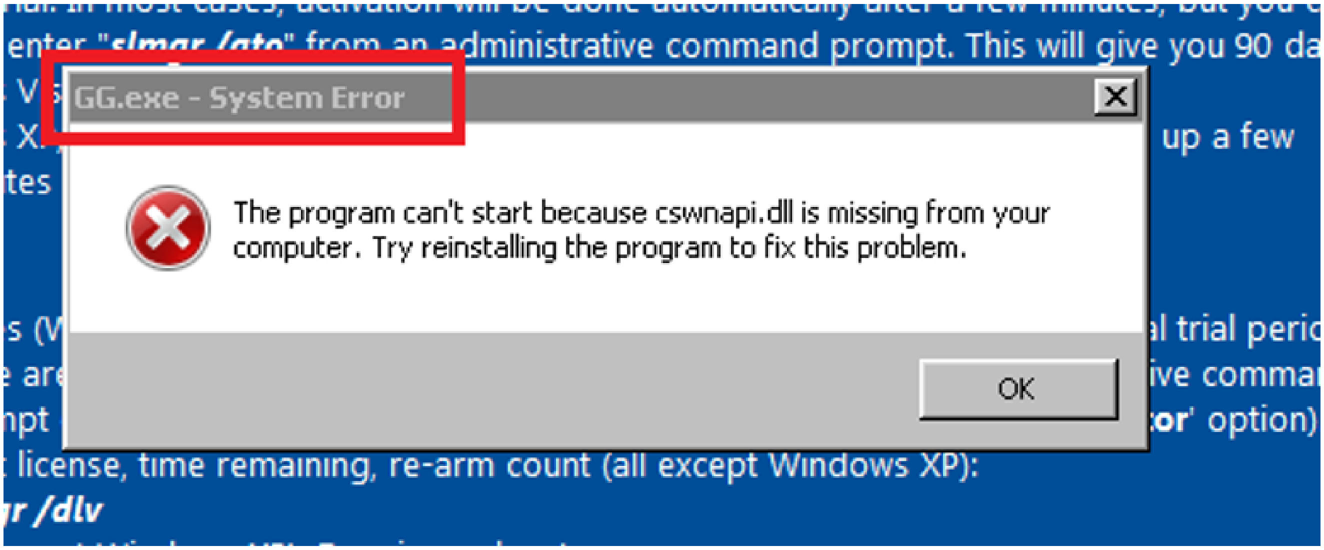


Figure 21. GG.exe gets executed

Controlling XFS to dispense the money

The binary GG.exe and XFSGG.dll are used to interface with Itautec/OKI XFS Middleware. When examining the properties of GG.exe, it is described as "JIG NMD" as seen in Figure 22. This resembles a legitimate Itautec tool used to test the functionality of the Dispenser. While it is not novel that criminals utilize ATM maintenance tools for malicious purposes, it is interesting that the criminals behind Ploutus did not follow the same methodology to control the XFS middleware directly. This suggests that the group behind Ploutus-I may not be the same group that created prior variants.

Figure 22. Itautec Maintenance tool
Additionally of note, the tool is written in Portuguese. In Figure 23, some extracts of the strings in the binary are visible.

K7 rej. falha shutter indo para empilhamento
K7 rej. falha fechando o K7
K7 rej. falha shutter indo p/ rej. simples
K7 rej. falha shutter indo p/ rej. pacote
K7 rej. erro de check sum nos dados do K7
Nota entre sensor rejeicao e NoteQualifier
Nota no sensor de rejeicao
Nota enviada a rej. simples no empilhador
Falha de comunicacao com K7 de rejeicao
Nao implementado
Sem K7
Intervencao necessaria no K7
Nivel baixo no K7
K7 vazio (K7 baixo nao detectado)
K7 vazio
K7 vazio-Alimentacao continua em outro canal
K7 marcado como vazio
O feeder nao consegue alimentar as notas
Aliment. interrompida-Nota entre feeder e NQ
Falha de sensor (NF ou NFC)
Aliment. interrompida-Rejeicao simples cheia
Rej. pacote,abortado nova alimentacao notas
Impossivel abrir ou fechar o K7
Falha de alimentacao com canal
Impossivel comunicacao interna com canal
Tarefa note feeder nao pode ser iniciada
Nao implementado
Falha nos pulsos de clock
Velocidade motor principal nao alcancada
Velocidade motor principal abaixo tolerancia
Velocidade motor principal acima tolerancia
Impossivel acesso a tarefa de transporte
Tarefa transporte nao pode ser iniciada
Nao implementado

Figure 23. Tool written in Portuguese

GG.exe opens a session with the Dispenser by using its logical name as "NDC_CASH_DISPENSER" in order to request information via code number 310 and action "WFS_INF_CDM_CONF" as shown in Figure 24.

Figure 24. GG.exe asking for Dispenser Status

Once the session opens, GG.exe reads data from the Dispenser via "WFS_CMD_CDM_READ_DATA" action, typically to get the total number of notes (bills) available and denomination. See Figure 25.

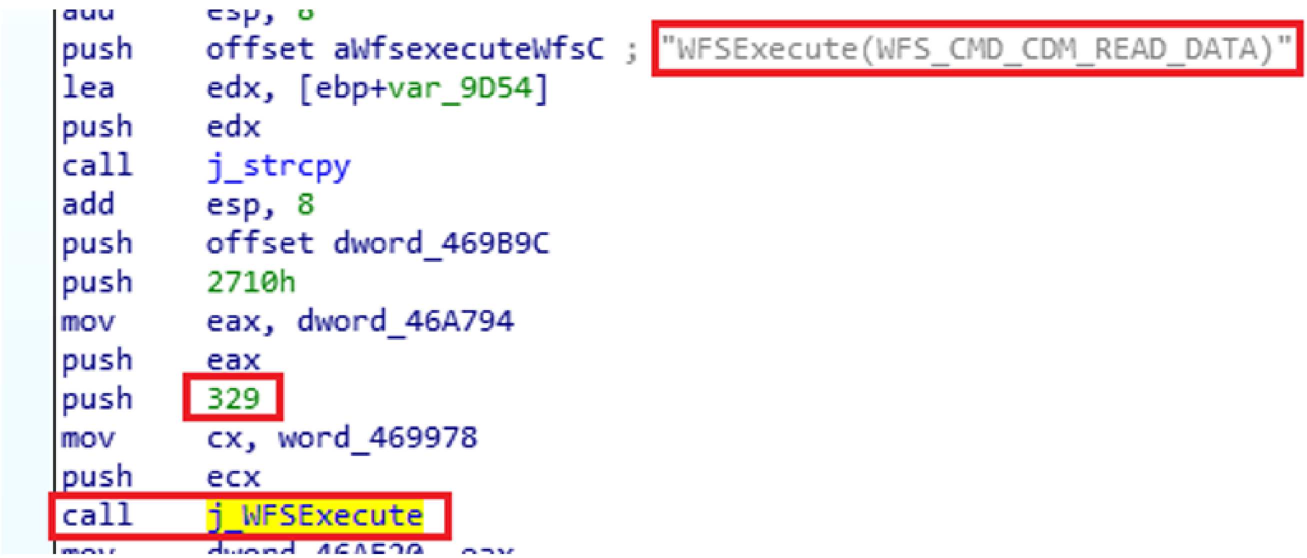


Figure 25: Gathering information from the Dispenser

In the next step, Ploutus-I requests an activation code, similar to a software license. This code enables criminals to limit the number of times the mules can use Ploutus-I to once a day. If the code is correct, it's "show me the money" time! In this stage, the XFS command "WFS_CMD_CDM_PRESENT" instructs the Dispenser to present the requested bills to the mule (see Figure 26).

```
mov     dword_46A794, 0
push    offset aWfsexecuteWfsC_8 ; "WFSExecute(WFS_CMD_CDM_PRESENT)"
push    offset unk_469AAC
call    j_strcpy
add     esp, 8
push    offset dword_469B9C
push    3E8h
mov     eax, dword_46A794
push    eax
push    303
mov     cx, word_469978
push    ecx
call    j_WFSExecute
```

Figure 26. “Show me the money”time!

As expected, the criminals know the exact ATM version they are targeting and its physical capabilities. As a result, in every round of attacks, the malware requested the maximum amount of bills to retrieve. In this case, the maximum number is 70, starting from the cassette with the highest denomination, to equal $35,000 MXN (~$1677 USD) per round. All the dispensing activity is stored in the log in: C:\itautec\exe\LibraryLog.txt. See Figure 27.

```
Activacion Correcta Codigo:
WFSExecute Result[-351] Data Leng [1] Cassete 1: 0 Cassete 2: 0 Cassete 3: 0 Cassete 4: 70 Cassete 5: 0
WFS_CMD_CDM_PRESENT Result[0]
WFSExecute Result[-351] Data Leng [1] Cassete 1: 0 Cassete 2: 0 Cassete 3: 0 Cassete 4: 70 Cassete 5: 0
WFS_CMD_CDM_PRESENT Result[0]
```

Figure 27. Malware cashing out

Also, Ploutus creates a SQLite Database at c:\Users\%USERNAME%\AppData\Roaming\NewLog, showing the dispensing related activities. See Figure 28.

Id	Data
1	C:\itautec\exe\GG.exe
2	WFSStartUp
3	WFSOpen
4	WFSOpen LogicalName: JIG_PTR : C:\itautec\exe\GG.exe
5	WFSOpen
6	WFSOpen LogicalName: NDC_CASH_DISPENSER : C:\itautec\exe\GG.exe
7	WFSGetInfo: 310
8	WFSFreeResult
9	WFSExecute Start
10	WFSExecute: 329 C:\itautec\exe\GG.exe
11	WFSExecute END [0]
12	WFSFreeResult
13	WFSExecute Start
14	WFSExecute: WFS_CMD_CDM_COUNT C:\itautec\exe\GG.exe
15	C:\itautec\exe\GG.exe
16	WFSStartUp

Figure 28. Dispensing Activity Logging

Recommendations

> Periodic check of AV whitelist folders to make sure they are up to date and do not have malicious paths added

> Automatic updates for all the software running in the ATM if possible

> Up-to-date AV signatures

> A proper implementation of hard disk encryption, but it is critical to do it correctly. An incomplete implementation can allow an attack to sniff the Volume keys from TPM to CPU over SPI/I2C bus, among other flaws.

> Next-generation centrally managed end-to-end encrypted cameras with tampering detection, motion alerts and facial detection

> Periodic ATM Penetration Testing to identify vulnerabilities and countermeasures at Hardware, Middleware, Firmware and Software level

> Make sure your provider generates of Indicators of Attack(IOA) and Indicators of Compromise (IOCs) during this exercise to improve the detection and monitoring of these attacks

> Set alerts on specific events inside the Journal, AV, EventLog or XFS log to detect and respond to these attacks in a timely manner

> Make sure your provider understands the format of the Journal of your ATM and can recommend what type of events to monitor.

Who we are

Ocelot, by Metabase Q, is the leading Offensive Security team in Latin America. This elite team of researchers represents the best of the best, partnered together to transform cybersecurity in the region. Ocelot threat intelligence, research, and offensive skills power Metabase Q's cybersecurity managed solutions.

Our Advanced ATM Penetration testing covers logic and physical attacks. We test ATMs with customized malware like Ploutus and others, as well as perform multiple physical attacks in the Dispenser, including Endoscope, TPM sniffing, DMA Attacks, TRF, CMOS Shock, etc., to provide a real assessment experience.

Do you know how your systems would perform with ransomware or other advanced attacks? Due to our reverse engineering capabilities, we track and dissect APTs to replicate their TTPs in our customers' environment. As a result, we are able to simulate advanced attacks and measure your security controls' effectiveness and investment

Do you have devices? IoT/ICS? We can assess them as well, from Hardware, Boot Loader, Middleware, Firmware all the way to Application level

We wrote the first secure code guideline for BASE24 to find vulnerabilities at the Switch or Bank BASE24/CONNEX to identify payment authorization bypass and PCI violations.

Please reach out at contact@metabaseq.com

Indicators of Compromise:

Paths:

C:\itautec\exe*

C:\itautec\exe\LibraryLog.txt

c:\Users\<user>\AppData\Roaming\NewLog

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\Winlogon\UserInit

Appendix A

Decrypted strings

IEBOLDP6

C:\Diebold\EDC\edclocal.dat2

[Launcher Client] Request

[LauncherSysApp] Request

CMD.exe /C wmic os where Primary='TRUE'

reboot [Launcher]

TaskKill.exe /F /IM

GG.exe /F /IM

NDCPlus.exe /F /IM

winvnc.exe /F /IM

MSXFSEXE.exe /F /IM

CajaExpress.exe

GG.exe

C:\NDC+\Lib\MsXfsExe

C:\NDC+\Bin$

[Launcher Client] Admin /C

TaskKill /F /IM

XFSConsole.exe /C

START XFSConsole.exe /C

TaskKill /F /IM

NewAge.exe /C

START NewAge.exe P /C

"C:\Program Files\Diebold\AgilisStartup\AgilisShellStart.exe"

[Launcher] Start

AgilisT:\Program Files\NCR APTRA\SSS Runtime

Coren:\Program Files\NCR APTRA\SSS Runtime Core\ulSysApp.exe

[LauncherSysApp]

"C:\Probase\ProDevice\BIN\ProDeviceStart.bat"

C:\Probase\ProDevice\BIN8 /C

START Delete.bat & pause /C

CMD.exe

[Launcher] Start

CMD procexp.exe

C:\ProgramFiles\Diebold\AMI\Diagnostics\bin\Diebold.Ami.Diagnostics.Diagnostics.exe

C:\Program Files\Diebold\AMI\Diagnostics\bin$ /C

START Main.exe /F /IM

CMD.exe

[Launcher] Start

END /F /IM

Wscript.exe /F /IM script.exe /F /IM vpncli.exe

DIEBOLDJ[Launcher Client]

Inicio Directo BootH

[Launcher Client]

Inicio Directo EPP

LauncherStart

Loading Wait

Press[Esc] to Continue

Software\Microsoft\Windows NT\CurrentVersion\winlogon

/C net localgroup administrators /add

[Launcher]

UserPermision Done

Done

[LauncherConfig:]

Service: >[LauncherConfig:]

Launch Menu: <[LauncherConfig:]

Launch App: <[LauncherConfig:]

LaunchDate: 6[LauncherConfig:]

TimeOut: 8[LauncherConfig:]

ReadFile: B[LauncherConfig:]

ExternalDrive: 2[LauncherConfig:]

Patch:

Reset.txt

[Launcher] Windows 7 Detected

install /c

C:\Windows\Microsoft.NET\Framework\v2.0.50727\InstallUtil.exe: & net start DIEBOLDP & pause

installonly

& pauseuninstall /c

C:\Windows\Microsoft.NET\Framework\v2.0.50727\InstallUtil.exe/u

test

[Launcher] Starting App Mode Detect Windows 7.B[Launcher]

Starting Service Mode.:[Launcher]

Starting App Mode.Launcher

43246*******4354

5204167231340092

CopyData:

$Config

Read

Start

File Exist.

File Open.

Read End.

Error.

Config New File.

Agilis.log

Config New File

Close.

ConfigCopy:

N.bin

Ploutos

Log.txt

Diebold Event

LogTSYSTEM\CurrentControlSet\Services\DIEBOLDP

Typej

SOFTWARE\Microsoft\WindowsNT\CurrentVersion\Winlogon

Userinit /C REG ADD"HKEY_LOCAL_MACHINE\Software\Microsoft\WindowsNT\CurrentVersion\Winlogon" /v Userinit /t REG_SZ /d "" /f

cmd.exe

Abrir

Arial

Black

Cerrar

Reiniciar

\\.\DISPLAY1

TEST OK

DISPLAY2

END OK

Could not impersonate the elevated user.

LogonUser returned error code {0}.

Load

Ver archivo adjuntoButton Text

Sigue leyendo

Para descargar el archivo y seguir leyendo, por favor danos la siguiente información.

Al menos Nombre de pila y un apellido

Por favor utilice su e-mail de trabajo

Gracias, haga click abajo para ver el archivo.

Descargar
Algo salió mal, por favor intente de nuevo.

Keep reading

To download this file and keep reading, please fill out the following form.

At least First and Last Name

Please use your work e-mail

Thank you, click below to view the file.

DOwnload

Oops! Something went wrong while submitting the form. Please try again.

Related

Relacionado