Let's Learn: Dissecting Lazarus Windows x86 Loader Involved in Crypto Trading App Distribution: "snowman" & ADVObfuscator

vkremez.com/2019/10/lets-learn-dissecting-lazarus-windows.html

Goal: Document and dissect the latest Lazarus Windows 32-bit (x86) version involved in the crypto trading application distribution targeting Windows and macOS users. The malware and the campaign were originally discovered by MalwareHunterTeam.

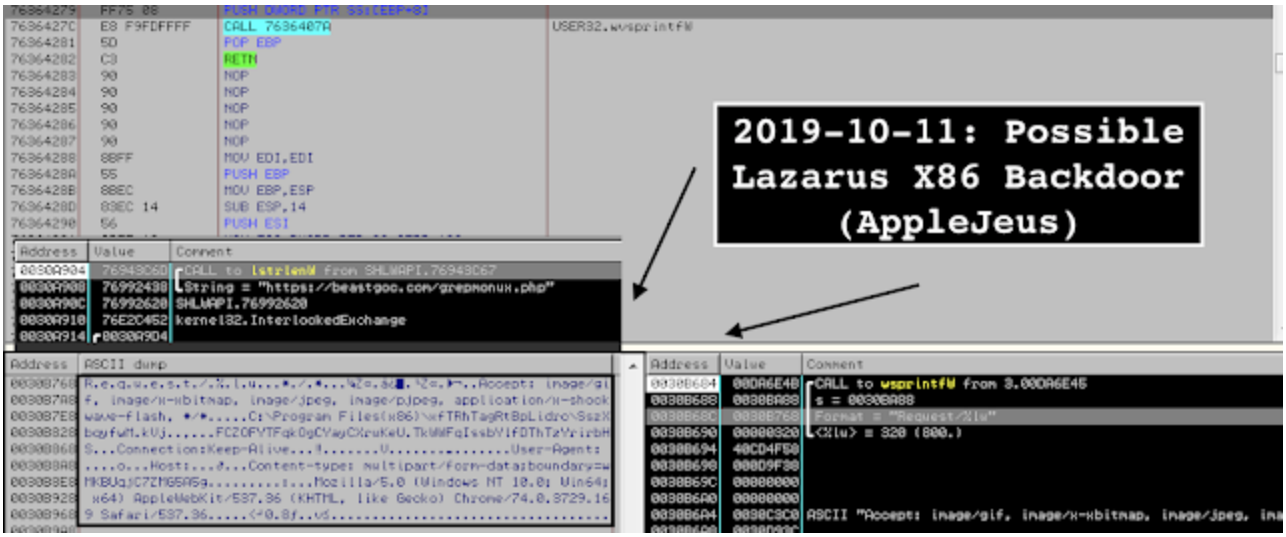
> Where we found it now?: https://www.jmttrading[.]org/ (Sectigo cert from July 11) -> https://github[.]com/jmttrading/JMTTrader/releases -> JMTTrader_Win.msi - signed installer (Sectigo given cert too) -> drops signed CrashReporter.exe to AppData. (The Mac .dmg has malware too...) pic.twitter.com/7r3SuWbItP
> — MalwareHunterTeam (@malwrhunterteam) October 11, 2019

Source:
Signed Windows .msi SHA-256:
07c38ca1e0370421f74c949507fc0d21f4cfcb5866a4f9c0751aefa0d6e97542
Signed Windows malware SHA-256:
9bf8e8ac82b8f7c3707eb12e77f94cd0e06a972658610d136993235cbfa53641
macOS .dmg SHA-256:
e352d6ea4da596abfdf51f617584611fc9321d5a6d1c22aff243aecdef8e7e55
macOS malware SHA-256:
4d6078fc1ea6d3cd65c3ceabf65961689c5bc2d81f18c55b859211a60c141806
Outline:

I. Background & Summary

The purported North Korean state-sponsored group known as “Lazarus” appears to continue targeting crypto users via elaborate and sophisticated malware distribution methodology by setting up the website, Twitter, and GitHub account as well as leveraging digital certificate for the Windows malware specifically.

Previously, Kaspersky researchers noted the very similar malware in 2018 in the report titled “Operation AppleJeus: Lazarus hits cryptocurrency exchange with fake installer and macOS malware.”



The group appears to employ both macOS and Windows malware variants. While the macOS version remained to be unobfuscated and simple, the Windows version of this malware is rather notable and included the renamed ADVObfuscation library as “snowman” to complicate malware reverse engineering.

The final payload is yet unknown; however, the group previously deployed this similar loader/backdoor to install the malware backdoor known as “Fallchill.”

II. Lazarus Windows 32-bit (x86) Loader/Backdoor Internals

The malware backdoor 32-bit (x86) is coded in Microsoft Visual C++ 8. It is signed and executed via "Maintain" parameter. The malware itself is heavily obfuscated executed as task "JMTCrashReporter".

```
$TaskName = "JMTCrashReporter"
$TaskDescription = "Crash Reporter for JMTTrader"
$TaskCommand = "$($env:APPDATA)\JMTTrader\CrashReporter.exe"
$TaskArg = "Maintain"
$TaskStartTime = [datetime]::Now.AddMinutes(1)
$service = new-object -ComObject("Schedule.Service")
$service.Connect()
$rootFolder = $service.GetFolder("\")
$TaskDefinition = $service.NewTask(0)
$TaskDefinition.RegistrationInfo.Description = "$TaskDescription"
$TaskDefinition.Settings.Enabled = $true
$TaskDefinition.Settings.AllowDemandStart = $true
$TaskDefinition.Principal.RunLevel = 1
$triggers = $TaskDefinition.Triggers
$trigger = $triggers.Create(9)
$trigger.StartBoundary = $TaskStartTime.ToString("yyyy-MM-dd'T'HH:mm:ss")
$trigger.Enabled = $true
$Action = $TaskDefinition.Actions.Create(0)
$Action.Path = "$TaskCommand"
$Action.Arguments = "$TaskArg"
$rootFolder.RegisterTaskDefinition("$TaskName", $TaskDefinition, 6, "System", $null, 5)
```

The compilation timestamp is Friday, October 04 02:22:31 2019 UTC with the Sectigo signer for "JMT TRADING GROUP INC" with the postal code "91748" and valid from 12/07/2019 00:00:00 to 11/07/2020 23:59:59. The zip code corresponds to the Los Angeles area, United States.

III. Command Line Check Function

The malware checks for the argument “Maintain” before final execution.

```
////////////////////////////////////////////////////////////////////////////////

////////////////////////////// Compare Cmd Function
//////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////

char __thiscall compare_command(void *this)
{
...
  v1 = this;
  v2 = GetCommandLineA();
  v3 = sub_445020(v2, 0x20u);
  **((_DWORD **)v1 + 1) = v3;
  if ( !v3 )
    goto LABEL_9;
  v4 = (_DWORD *)*((_DWORD *)v1 + 1);
  v5 = strcmp((const char *)++*v4, **((const char ***)v1 + 2));// 'Maintain' arg
check
  if ( v5 )
    v5 = -(v5 < 0) | 1;
  if ( v5 )
LABEL_9:
    result = 1;
  else
    result = 0;
  return result;
}
```

IV. Encoder Function

The binary encodes the victim information using the key “X,%`PMk--Jj8s+6=15:20:11” before submitting the information to the server. The pseudo-coded function is as follows:

```
//////////////////////////////////////////////////////////////////////////////

//////////////////////////////// Encoder Function
//////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////


int __thiscall encoder_func(int this)
{
...
  v1 = this;
  v2 = this;
  v3 = *(_DWORD **)(this + 8);
  *v3 ^= 0x721u;
  *v3 ^= 0x721u;
  v4 = **(_DWORD **)(this + 8);
  v5 = *(_DWORD **)(this + 8);
  *v5 ^= 0x721u;
  *v5 ^= 0x721u;
  v6 = **(_DWORD **)(v1 + 8);
  v7 = *(_DWORD **)(v1 + 12);
  *v7 ^= 0x721u;
  *v7 ^= 0x721u;
  v8 = **(_DWORD **)(v1 + 12);
  v9 = *(_DWORD **)(v2 + 8);
  *v9 -= 0xCBC;
  *v9 += 0xCBC;
  *(_BYTE *)(**(_DWORD **)(v2 + 8) + **(_DWORD **)(v2 + 4)) = *(_BYTE *)
     (**(_DWORD **)(v2 + 4) + v4) ^ key[v6 % v8]; // X,%`PMk--Jj8s+6=15:20:11
  return **(_DWORD **)(v2 + 16);
}
```

V. Malware Capabilities

The similar binary capabilities were documented as part of the analogous unobfuscated MacOS version in the report titled “Pass the AppleJeus.” In this case, the Windows malware includes the separator “--wMKBUqjC7ZMG5A5g”

The malware capabilities include the following shortened functionality:

```
Read/write itself to various directories
Query registry and save in the registry
Connect to the server
Find files
Extract and decode resource
Collect processes delete and terminate them
```

The malware formats the request and processes the command from the server as follows:

```
Request/%lu
%sd.e%sc "%s > %s 2>&1"
```

The malware connection form-data is as this for example:

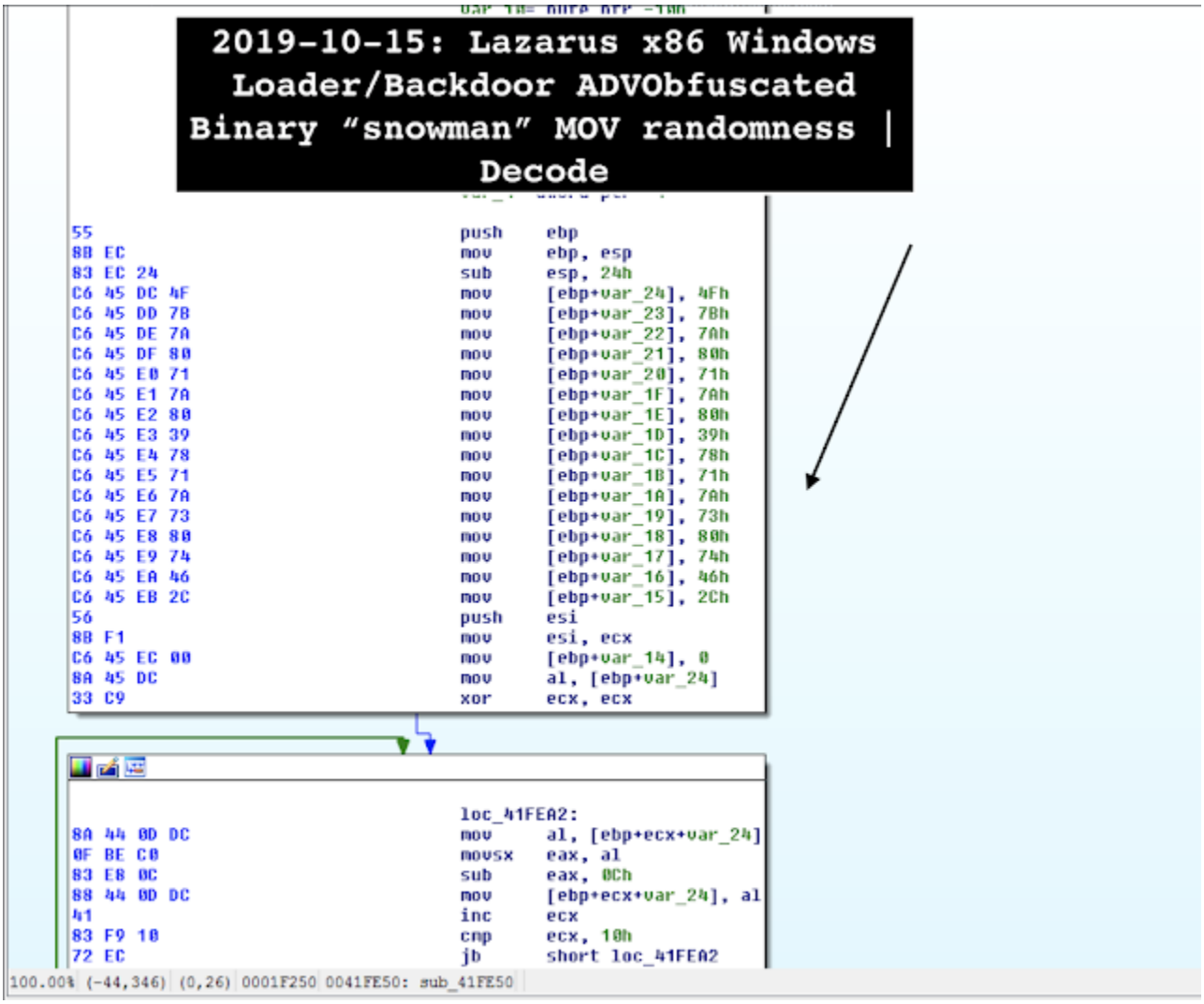
```
xX7ZXX5A5g
--wMKBUqjC7ZMG5A5g
Content-Disposition: form-data; name="token";

11056
--wMKBUqjC7ZMG5A5g
Content-Disposition: form-data; name="query";

conn
--wMKBUqjC7ZMG5A5g
Content-Disposition: form-data; name="content"; filename="mont.jpg"
Content-Type: application/octet-stream
```

VI. Lazarus Loader/Backdoor: ADVObfuscator as "snowman" Library

One of the more interesting discoveries was that the Lazarus malware utilizes the ADVObfuscator open-source library simply renamed as “snowman” based on the template C++ definition left in the binary. The malware developer appears to have simply manually inserted the obfuscator library with the definitions.

“ADVobfuscator demonstrates how to use C++11 language to generate, at compile-time, obfuscated code without using any external tool and without modifying the compiler. The technics presented rely only on C++11, as standardized by ISO. It shows also how to introduce some form of randomness to generate polymorphic code and it gives some concrete examples like the encryption of strings literals and the obfuscation of calls using finite state machines.”

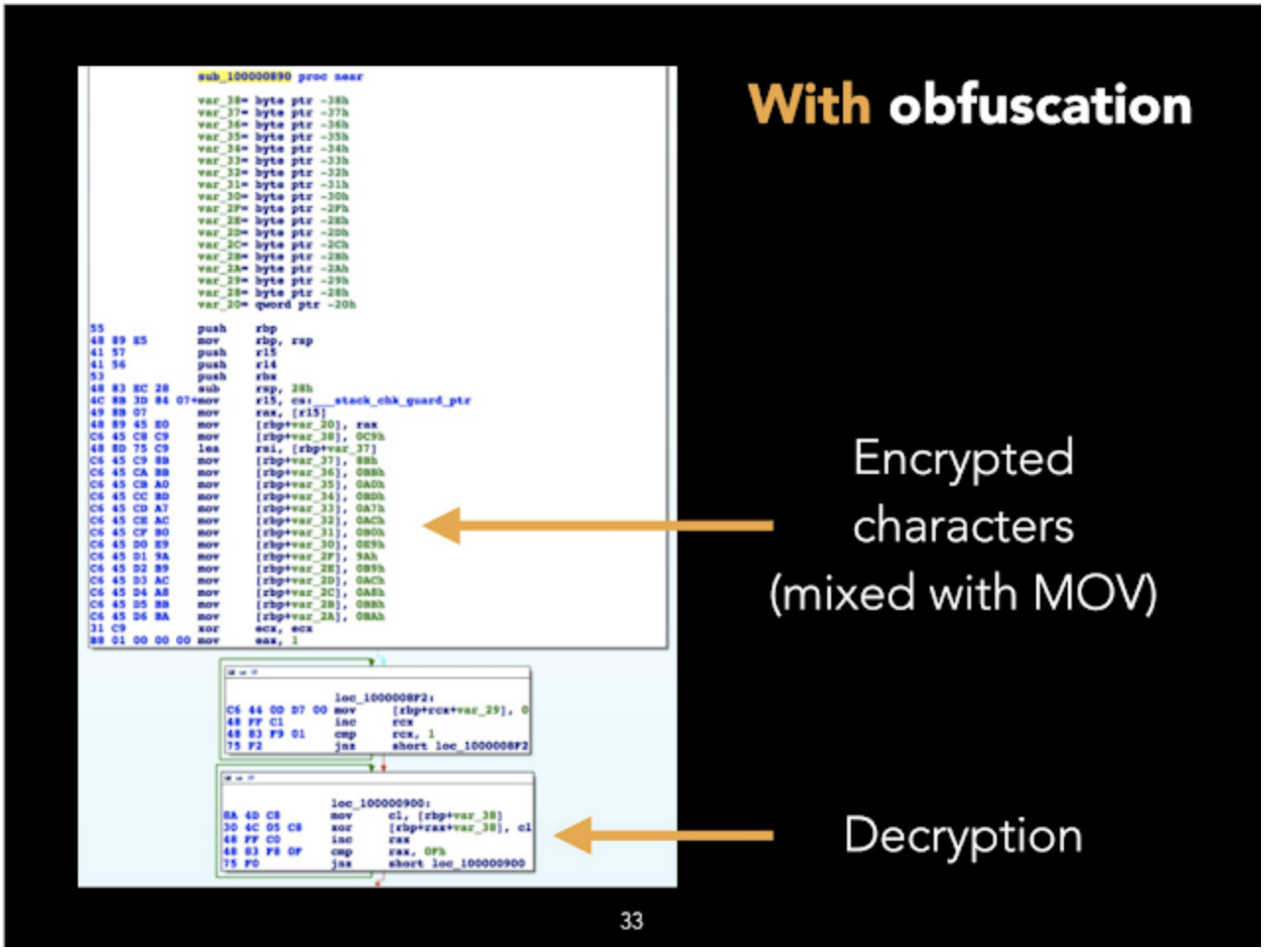
The Lazarus sample introduces randomness leveraging mov instruction and offsets to function to complicate static and dynamic analysis as well as reverse engineering efforts.



The fragment of the typical generated deobfuscation code is as follows:

```
////////////////////////////////////////////////
//// FRAGMENT OF OBFUSCATION ADDRESS RETRIEVE /////
////////////////////////////////////////////////
0FBEC0         MOVSX EAX,AL
83E8 07        SUB EAX,7
88440C 78      MOV BYTE PTR SS:[ESP+ECX+78],AL
41             INC ECX
83F9 1C        CMP ECX,1C
```

During AppSec2014, Sebastien Andrivet demonstrated this exact similar technique used by the Lazarus sample via the renamed ADVObfuscation library.



The relevant template and definition of the original relevant source code are as follows:

```
#define OBFUSCATED_CALL0(f)
andrivet::ADVobfuscator::ObfuscatedCall<andrivet::ADVobfuscator::Machine1::Machine>
(MakeObfuscatedAddress(f, andrivet::ADVobfuscator::MetaRandom<__COUNTER__,
400>::value + 278))
#define OBFUSCATED_CALL_RET0(R, f)
andrivet::ADVobfuscator::ObfuscatedCallRet<andrivet::ADVobfuscator::Machine1::Machine,
 R>(MakeObfuscatedAddress(f, andrivet::ADVobfuscator::MetaRandom<__COUNTER__,
400>::value + 278))

#define OBFUSCATED_CALL(f, ...)
andrivet::ADVobfuscator::ObfuscatedCall<andrivet::ADVobfuscator::Machine1::Machine>
(MakeObfuscatedAddress(f, andrivet::ADVobfuscator::MetaRandom<__COUNTER__,
400>::value + 278), __VA_ARGS__)
#define OBFUSCATED_CALL_RET(R, f, ...)
andrivet::ADVobfuscator::ObfuscatedCallRet<andrivet::ADVobfuscator::Machine1::Machine,
 R>(MakeObfuscatedAddress(f, andrivet::ADVobfuscator::MetaRandom<__COUNTER__,
400>::value + 278), __VA_ARGS__)

    // Obfuscate the address of the target.
    // Very simple implementation but enough to annoy IDA and Co.
    template<typename F>
    struct ObfuscatedAddress
    {
        // Pointer to a function
        using func_ptr_t = void(*)();
        // Integral type big enough (and not too big) to store a
     function pointer
        using func_ptr_integral = std::conditional<sizeof(func_ptr_t) <=
        sizeof(long), long, long long>::type;

        func_ptr_integral f_;
        int offset_;

        constexpr ObfuscatedAddress(F f, int offset):
       f_{reinterpret_cast<func_ptr_integral>(f) + offset}, offset_{offset} {}
        constexpr F original() const { return reinterpret_cast<F>(f_ - offset_); }
    };

    // Create a instance of ObfuscatedFunc and deduce types
    template<typename F>
    constexpr ObfuscatedAddress<F> MakeObfuscatedAddress(F f, int offset) {
     return ObfuscatedAddress<F>(f, offset); }

}}
```