The New and Improved macOS Backdoor from OceanLotus

By Erye Hernandez and Danny Tsechansky
June 22, 2017 at 10:00 AM
Category: Unit 42 Tags: backdoor, macOS, OceanLotus, threat intelligence

Introduction

Recently, we discovered a new version of the OceanLotus backdoor in our WildFire cloud analysis platform which may be one of the more advanced backdoors we have seen on macOS to date. This iteration is targeted towards victims in Vietnam and still maintains extremely low AV detection almost a year after it was first discovered. Despite having been in the wild for an extended period of time, the operation appears to still be active. During our analysis, we were able communicate directly with the command and control server as recently as early June 2017.

While there seem to be similarities to an OceanLotus sample discovered in May 2015, a variety of improvements have been made since then. Some of the improvements include the use of a decoy document, elimination of the use of command line utilities, a robust string encoding mechanism, custom binary protocol traffic with encryption, and a modularized backdoor.

Infection Vector

The new OceanLotus backdoor is distributed in a zip file. While we don't have direct evidence for the initial infection vector we presume it's most likely via an email attachment. Once the user has extracted the zip file, they see a directory containing a file with a Microsoft Word document icon. The file is actually an application bundle, which contains executable code. (see Figure 1). Once the user double clicks on the purported Word document, the Trojan executes and then launches Word to display a decoy document.

The malware uses the decoy document to help mask the execution of the malware. This technique is a common one for Windows-based malware, but rare on macOS. In order to achieve this layer of obfuscation, the malware author had to trick the operating system into believing the folder is an application bundle despite the .docx extension. Traditionally, macOS malware have emulated legitimate application installers such as Adobe Flash, which was how the previous version of OceanLotus was packaged.

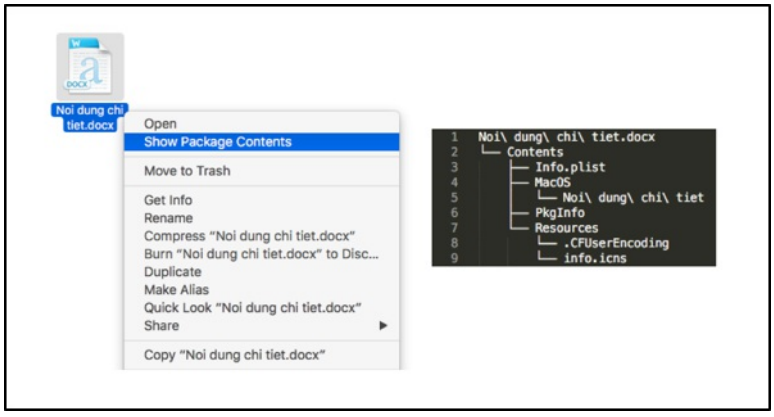

Figure 1. Context menu and file listing

Once the application bundle is launched, it opens a hidden file in the bundle's Resources folder named .CFUserEncoding which is a password-protected Word document (see Figure 2). It also copies this file to the executable path and essentially replaces the application bundle after persistence has been set up. This would lead the victim to believe that nothing was amiss, as they thought they were opening a Word document and a Word document opened. In this case, the Word file has the name "Noi dung chi tiet.docx", which is Vietnamese for "Details."

Figure 2. Decoy document prompts for a password to open the file

Persistence

Compared to the previous version of this backdoor, the persistence mechanism for this remained largely the same. This version creates a Launch Agent that runs when the victim host starts up, where as in the previous version execution was upon when a user logs in. It also copies itself to a different location and filename based on the UID of the user who ran the application.

For a user other than root, it takes the MD5 hash of the structure returned by *getpwuid()* and breaks the hash down into segments *<first 8 chars of*

hash>-<next 16 chars of hash>-<last 8 chars of hash>. This segmented MD5 hash is prepended with *"0000-"* then used as a directory in *~/Library/OpenSSL/* to store the executable file (see Figure 3). If the user is root, the executable is stored in the system wide library directory at /Library/TimeMachine/bin/mtmfs.

It is interesting to note that the executable and plist locations look like legitimate applications.

UID	plist Location	Executable Location
0	/Library/LaunchDaemons/com.apple.mtmfsd.plist	/Library/TimeMachine/bin/mtmfs
> 0	~/Library/LaunchAgents/com.apple.openssl.plist	~/Library/OpenSSL/0000-<segmented MD5 hash>/servicessl

Figure 3. plist and executable names and locations based on UID

Once the malware has set up persistence, it deletes the application bundle from the executable path leaving the decoy document in its place and launches itself as a service from the new location.

No Command Line Utilities

One of the first things we noticed about this backdoor is the lack of suspicious strings which often times provides context as to what the malware might do on a victim host. In most macOS malware, calls to the *system()* or *exec()* functions to run additional scripts are in place. In this case, these were not present nor were there command line utility strings that may easily convey the malicious intention of the application. This shows a deep level of understanding of the macOS platform by the author of this backdoor compared to other threat actors that will commonly copy and paste scripts from the Internet.

The lack of these strings may also double as an anti-analysis technique to make the malware seem less suspicious, especially to basic static analysis.

String Decoding

Since there appear to be no obvious suspicious strings in plaintext, we move onto the possibility of use of encoded, or obfuscated strings.

The string decode routine for this backdoor is an upgrade from previous versions in which strings were XOR encoded with the word *"Variable"* as a key. The string decode routine now consists of a combination of bit shifting and XOR operations with a variable key that depends on the length of the string that was encoded. If the computation for the variable XOR key turns out to be 0, the default XOR key of 0x1B is used. Figure 4 shows a Python implementation of the decode function.

```
1   def revbits(c):
2     binary = bin(c)[2:].rjust(8,'0')
3     return int(binary[::-1], 2)
4
5   def rotbits(c, amt, lr):
6     amt = amt&7
7     if not lr:
8       return ((c << (amt)) | (c >> (8-amt)))&0xff
9     else:
10      return ((c >> (amt)) | (c << (8-amt)))&0xff
11
12  def decode(s):
13    s = bytearray(s)
14    #reverse bits
15    s = bytearray(revbits(c) for c in s)
16
17    #add shift
18    shift_amt = 0xfc
19    shift_amt = (-(len(s)%8))&0xff or shift_amt
20    s = bytearray((c+shift_amt)&0xff for c in s)
21
22    #xor
23    xor_amt = 0x1b
24    var_key = 0x21 * (  ((0x3E0F83E1*len(s))>>63) + ((0x3E0F83E1*len(s))>>35)  )
25    if len(s) != var_key:
26      xor_amt = len(s) - var_key
27    s = bytearray((c^xor_amt) for c in s)
28
29    #rotate bits in str
30    s = bytearray(rotbits(c, len(s)%8 or 4, 0) for c in s)
31
32    return s
```

Figure 4. Python implementation of the malware's string decode function

After decoding the strings (see Figure 5), we can glean that the malware sets up persistence, surveys the victim's computer, and sends this information back to a server. At this point, it is still not obvious that this malware contains backdoor functionality.

```
x86_64
call.raidstore.org
technology.macosevents.com
press.infomapress.com
24h.centralstatus.net
/Library/LaunchAgents/com.apple.openssl.plist
/Library/LaunchDaemons/com.apple.mtmfsd.plist
/bin/launchctl load
servicessl
mtmfs
/Library/OpenSSL/
/Library/TimeMachine/
0000-
bin
Label
com.apple.openssl
com.apple.mtmfsd
ProgramArguments
KeepAlive
RunAtLoad
OnDemand
IOPlatformExpertDevice
IOService:/
IOPlatformUUID
/Library/Preferences/.files
pth
/System/Library/CoreServices/SystemVersion.plist
ProductName
ProductVersion
%02X:%02X:%02X:%02X:%02X:%02X
Model ID:
CPU:
machdep.cpu.brand_string
Memory:
Serial No:
Contents/MacOS
.CFUserEncoding
```

Figure 5. List of decoded strings

Custom Binary Protocol and Encrypted Traffic

The threat actors responsible for this malware appear to have spent some amount of effort to develop their own custom communication protocol. They did not simply use an off-the-shelf web server for their command and control server, as is commonly done. Instead, they created their own command and control mechanism.

The backdoor uses a custom binary protocol on TCP port 443, a well-known port that is unlikely to be blocked by traditional firewalls due to its use in HTTPS connections. The packet seen in Figure 6 is encoded with a combination of bit shifting (see Figure 7) and XOR with a key of 0x1B before it is sent. The bits are always rotated to the left 3 times before doing the XOR operation. This is an improvement from the previous version where the packet was only XOR encoded with a key of 0x1B.

```
00000000  0b 41 61 54 03 e0 c3 8a  c3 63 63 63 63 63 63 2d   .AaT.... .ccccccc-
00000010  23 81 23 8c 67 ef 67 43  3f 63 63 63 63 8c 63 63   #.#.g.gC ?cccc.cc
00000020  63 63 63 63 63 63 63 63  63 63 63 63 63 63 63 63   cccccccc cccccccc
00000030  63 63 63 63 63 63 63 63  63 63 63 63 63 63 63 63   cccccccc cccccccc
00000040  63 63 5e 63 63 63 63 63  73 43 6f 9c 63 42 63 63   cc^ccccc sCo.cBcc
00000050  77 43                                              wC
```

Figure 6. Initial packet sent by the client to the server

```
__int64 __fastcall rotbits(unsigned int a1, int amt, char lr)
{
  unsigned int v3; // ecx@3
  char v4; // di@3
  unsigned int v5; // eax@3

  if ( amt > 0 )
  {
    do
    {
      if ( lr )
      {
        v3 = (char)a1;
        v4 = (_BYTE)a1 << 7;
        v5 = v3 >> 1;
        if ( BYTE1(v3) & 1 )
          LOBYTE(v5) = v5 + -128;
        LOBYTE(a1) = v5 | v4;
      }
      else
      {
        a1 = ((unsigned int)(unsigned __int8)a1 >> 7) | (unsigned __int8)(2 * a1);
      }
      --amt;
    }
    while ( amt );
  }
  return (unsigned int)(char)a1;
}
```

Figure 7. Bit shifting function used in the encode/decode routine for network packets

After decoding the packet, we can see a breakdown of different fields. Figure 8 shows the initial packet sent by the client to the server. It is relatively empty aside from the “magic” bytes, length of data and type of communication.

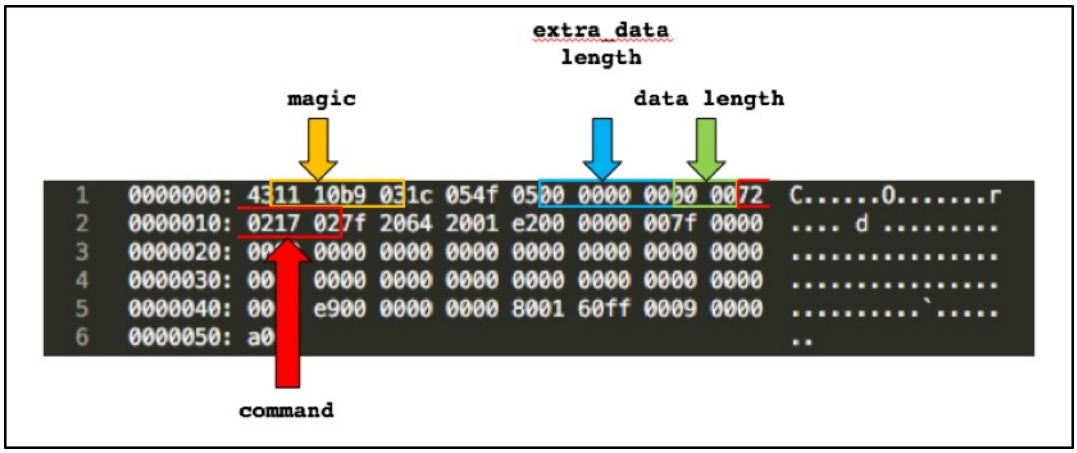

Figure 8. Initial packet sent by the client to the server (decoded)

Depending on the command response sent from the server, a packet may be bigger than 0x52 bytes. Data beyond 0x52 bytes is zlib compressed then encrypted with AES in CBC mode with a null initialization vector (IV) and a key sent from the server that is padded to 32 bytes.

We captured live traffic from the server, and observed that the encryption keys sent from the server are ephemeral. This means that each new session with the server is given a different key used to encrypt data sent back and forth within that session. This is a marked improvement compared to the previous version, where only XOR encoding with a one-byte key was used for encryption.

After decoding the packet it receives from the server, the backdoor validates certain fields like the “magic” bytes and makes sure the length of the data being received is not over a certain amount. Throughout the program execution, it also checks and handles any errors that may have been generated.

Command and Control Communications

The command and control server communication sequence is as follows:

1. The client initiates a session with the server by sending a packet with 0x2170272 in the command field.
2. The server then responds with an ephemeral encryption key and a command.
3. The client checks if the received packet from the server is valid.
4. The client executes the command sent by the server and responds with a zlib compressed and AES encrypted blob of the result then sends this back to the server.

Unlike the previous versions of OceanLotus where the commands can be easily gathered from its strings, the author has obfuscated the functions with constant values. We decoded the following available commands as seen in Figure 9.

Command	Command Description
0x2170272	Initialize
0x5CCA727	???
0x2E25992	receive file from server
0x2CD9070	get info on a file / directory
0x12B3629	delete file / directory
0x138E3E6	???
0x25D5082	execute function from a dynamic library
0x25360EA	send file to server
0x17B1CC4	???
0x18320E0	send victim and computer information together with the backdoor’s watermark
0x1B25503	execute a function from a dynamic library
0x1532E65	execute a function from a dynamic library

Figure 9. List of commands available

Command 0x2170272

When the backdoor is launched, a file is created in */Library/Preferences/.files* or *~/Library/Preferences/.files* depending on the victim’s user ID. This file (see Figure 10) contains a timestamp and the victim’s name concatenated with the machine’s serial number which is then hashed twice with MD5. This is then copied to a buffer that is 0x110 bytes long and AES encrypted in CBC mode with a null IV and a key of *“pth”*. It is then saved into the file.

Timestamp + MD5(MD5(<victim’s name + machine serial number>))

After this file is created, the client sends its first packet to the server with 0x2170272 in the command field. The server acknowledges and responds with the same command and the client verifies that the file has been created.

\xa7\xf1\xd9*\x82\xc8\xd8\xfe**4137674062B3226FE630C24F7DE1021E**\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0

Figure 10. Decrypted contents of ~/Library/Preferences/.files

Command 0x18320E0

The server then sends this command with an ephemeral key shortly after it sends the 0x2170272 command. The client gathers all the data seen in Figure 11, encrypts it with the key provided by the server and sends it back. One thing to note is the Base64 string that is sent in this packet. This string is static in the binary and does not change, which may be indicative of a marker for campaign or version identification.

\x00\x00\x00**4137674062B3226FE630C24F7DE1021E**\xe9\x0f\x00\x00\x00**Mac OS X 10.X.X**\xb6\x03\x00\x00

\x00**username**\t\x00\x00\x00**localhost**\x18\x00\x00\x00**Ze0pXcpfbqbS4wD0eS/LVQ==**\xb6\xbc\x1cY\x00\x00\x00\x00M\x00\x00\x00**/Users/usernam**
ABCDEF01-23456789ABCDEF01-23456789/

servicessl\x8b\xbc\x1cY\x00\x00\x00\x00\x17\x00\x00\x00**en0 : AA:BB:CC:DD:EE:FF**[\x00\x00\x00**lo0 : fe80::1**\n**lo0 : 127.0.0.1**\n**lo0 : ::1**\n**en0 : fe80::a**
: 192.168.1.254

\x05\x01\x00\x00f\x00\x00\x00**Model ID:iMac8,1**\n**CPU:Intel(R) Core(TM)2 Duo CPU T7700 @ 2.40GHz**\n**Memory:4.00**\n**Serial No:XXXXXXXXXXX**\x

Figure 11. Decrypted contents of a packet sent by the client to the server

Not highlighted in Figure 11 but also included in this packet is the kernel boot time which may be used by the C2 server to help determine if the backdoor is being run in a sandbox environment.

Commands 0x25D5082, 0x1B25503, 0x1532E65

These commands load a dynamic library using *dlopen()* and obtains a function pointer to execute within that shared library using *dlsym()*. Unfortunately, we do not know which dynamic libraries or functions are used for each command since these are server supplied and we were not able to capture any communication that used these commands.

However, we can postulate that since the parameters to the functions have the same number of arguments with the first being a fairly large constant similar to the command constants, (see Figure 12) and the backdoor has a function for receiving files, it is possible that these functions correspond to a shared library that the server uploads to the victim host. This means that additional functionality can be added to this backdoor by loading modules directly from the C2 server.

```
v12 = ((__int64 (__fastcall *)(signed __int64, _QWORD, _QWORD, void **, int *, int *))func_ptr)(
        0x116DE9CLL,
        0LL,
        0LL,
        v20,
        &v23,
        &v22);

func_ptr(0x179896CLL, v6, v24, v18, &v21, &v20);


if ( func_ptr(0x5E57538LL, v8, v21, v17, &v20, &v19) )
```

Figure 12. Snippets showing loaded function pointers and their parameters

Conclusion

Most macOS malware in the wild today are not very complex, but threat actors have been quickly improving their tradecraft. The increased level of sophistication and complexity may be indicative of increased targeting of macOS hosts looking to the future. With this OceanLotus attack in combination with recent macOS versions of the Sofacy group's toolset, we have now observed multiple espionage motivated threat actors targeting macOS. It is imperative that the same types of strong security practices and policies organizations use to defend Windows devices are applied universally to include macOS devices as well.

Apple has already updated the macOS protection systems to address this variant of OceanLotus.

Palo Alto Networks customers are protected and may learn more via the following:

- Samples are classified as malicious by WildFire
- Domains and IPs have been classified as malicious and IPS signatures generated
- AutoFocus users may learn more via the OceanLotus tag

Indicators of Compromise

Hashes

b3337016785333070494568 4c50ce0af6eb27838e1e3f88ea457d2c88a223d8b Noi dung chi tiet.zip

b3cf3e3b52b4b899cd0814fc75698ea24f08ce18642665adcd3555a068b5c16d Info.plist

07154b7a45937f2f5a2cda5b701504b179d0304fc653edb2d0672f54796c35f7 Noi dung chi tiet

82502191c9484b04d685374f9879a0066069c49b8acae7a04b01d38d07e8eca0 PkgInfo

f0c1b360c0b24b5450a79138650e6ee254afae6ce8f6c68da7d1f32f91582680 .CFUserEncoding

e84b5c5152d8edf1e814cc4b4975bfe4dc0063ef90294cc96b383f523042f783 info.icns

C2 Server

call[.]raidstore[.]org

technology[.]macosevents[.]com

press[.]infomapress[.]com

24h[.]centralstatus[.]net

93.115.38.178

Dropped Files

UID == 0	UID > 0
/Library/LaunchDaemons/com.apple.mtmfsd.plist	~/Library/LaunchAgents/com.apple.openssl.plist
/Library/TimeMachine/bin/mtmfs	~/Library/OpenSSL/0000-<segmented MD5 hash>/servicessl
/Library/Preferences/.files	~/Library/Preferences/.files

SALES > 866.320.4788 »

SEE A DEMO »

TAKE A TEST DRIVE