METHODS of MALWARE PERSISTENCE

...on OS X Mavericks

@patrickwardle
PATRICK WARDLE

Synack: director of R&D

NASA: autonomous software dev.

VRL: chief science officer

NSA: something dark side
THE SHORT; HOW ‘BAD CODE’ ON OS X PERSISTS

→ MORE COMPREHENSIBLY: A TECHNICAL EXPLORATION OF OS X, MALWARE, & PERSISTENCE

THE MAC BOOT PROCESS
a technical overview of how macs startup

OS X MALWARE
persistent malware designed for computers running OS X

METHODS OF PERSISTENCE
getting code to run every time a mac reboots/user logs in

AUTORUNS, FOR OS X
what programs start automatically on reboot/log in
BACKGROUND

...and why you should care
THE RISE OF MACS

SO MACS ARE PRETTY MUCH EVERYWHERE

-> SURGING DEMAND, IN BOTH THE HOME AND ENTERPRISE

Apple is now the #3 vendor in US PC shipments!

Macs as % of total US PC sales

doesn’t include iDevices
MALWARE ON OS X?

BUT MACS DON'T GET MALWARE...RIGHT?

“IT DOESN'T GET PC VIRUSES, A MAC ISN'T SUSCEPTIBLE TO THE THOUSANDS OF VIRUSES PLAGUING WINDOWS-BASED COMPUTERS.” (APPLE.COM)

1982 Elk Cloner: First virus, infected Apple II's

Last year: 33 new OS X malware families, some infecting Apple, Facebook, and Twitter's corporate systems.
APPLE'S "RESPONSE"

APPLE HAS CLEARLY SHORED UP THEIR OS

⇒ AN ACKNOWLEDGMENT (OR CONFIRMATION?) OF AN OS X MALWARE PROBLEM

- **XProtect**
  - OS X's built-in anti-virus product

- **Gatekeeper**
  - Verifies software, disallowing unauthorized binaries

- **Sandboxing**
  - Prevents apps from accessing core/OS level components

- **Signed Code**
  - Signatures are verified and only signed Kexts can be loaded

...so we are all secure now, right?! 😊
XPROTECT IS APPLE’S ANTI-MALWARE SYSTEM

-> THE ‘90S CALLED, THEY WANT THEIR STATIC SIGNATURE-BASED A.V. PRODUCT BACK

XProtect's Signature File

Detect based on a hash/filename?!

DOESN'T DETECT NEW/MODIFIED MALWARE 😞

<table>
<thead>
<tr>
<th>Key</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root</td>
<td>Array</td>
<td>(38 items)</td>
</tr>
<tr>
<td>Item 0</td>
<td>Dictionary</td>
<td>(3 items)</td>
</tr>
<tr>
<td>Item 1</td>
<td>Dictionary</td>
<td>(3 items)</td>
</tr>
<tr>
<td>Item 2</td>
<td>Dictionary</td>
<td>(3 items)</td>
</tr>
<tr>
<td>Item 3</td>
<td>Dictionary</td>
<td>(3 items)</td>
</tr>
<tr>
<td>Description</td>
<td>String</td>
<td>OSX_LaoShu.A</td>
</tr>
<tr>
<td>LaunchServices</td>
<td>Dictionary</td>
<td>(1 item)</td>
</tr>
<tr>
<td>LSItemContentType</td>
<td>String</td>
<td>com.apple.application-bundle</td>
</tr>
<tr>
<td>Matches</td>
<td>Array</td>
<td>(1 item)</td>
</tr>
<tr>
<td>Item 0</td>
<td>Dictionary</td>
<td>(3 items)</td>
</tr>
<tr>
<td>Identity</td>
<td>Data</td>
<td>&lt;2e243393 a4e997d5 3d3d8051 6571a64f 10313116&gt;</td>
</tr>
<tr>
<td>MatchFile</td>
<td>Dictionary</td>
<td>(1 item)</td>
</tr>
<tr>
<td>NSURLNameKey</td>
<td>String</td>
<td>worty</td>
</tr>
<tr>
<td>MatchType</td>
<td>String</td>
<td>Match</td>
</tr>
</tbody>
</table>
GATEKEEPER

GATEKEEPER VERIFIES DOWNLOADED SOFTWARE

-> UNVERIFIED BINARIES THAT HAVE BEEN DOWNLOADED MAY ALERT OR BE DENIED EXECUTION

//attributes
$ xattr -l ~/Downloads/googlechrome.dmg
com.apple.quarantine:0001;534e3038;
Google Chrome; B8E3DA59-32F6-4580-8AB3...

...DOES NOTHING TO PREVENT ADVANCED (EXPLOIT-BASED/‘DRIVE-BY’) ATTACKS ☹️
APPS EXECUTE WITHIN A SANDBOX

This actually does a solid job of preventing apps from accessing external resources.

Without app sandbox:
- User data
- System resources

With app sandbox:
- User data
- System resources
- Sandbox

The app without sandbox has direct access to user data and system resources. With the app sandbox, the app is isolated, preventing it from accessing external resources.
**Signed Apps**

The OS (Mach-O) loader verifies all signatures. This allows apps to be verified and prevents malicious modifications (infections).

- Process: Safari [1599]
- Path: /Applications/Safari.app/Contents/MacOS/Safari
- Crashed Thread: 0
- Exception Type: EXC_CRASH (Code Signature Invalid)
- Exception Codes: 0x0000000000000000, 0x0000000000000000

Invalid signature ‘crash’ killed by the loader!
(un)SIGNED APPS

OK, BUT CAN SIGNED APPS REALLY BE PROTECTED?

-> THE CRYPTO SEEMS SOLID...BUT WHAT IF IT WASN'T THERE ANYMORE?

```
# md5 Safari.app/Contents/MacOS/Safari -> 633d043cf9742d6f0787acdee742c10d
# Unsign.py Safari.app/Contents/MacOS/Safari
# Safari code signature removed

# md5 Safari.app/Contents/MacOS/Safari -> 825edd6a1e3ae936f9a60bac409

$ open Safari.app/Contents/MacOS/Safari

$ ps aux | grep Safari
patrick 15099 /Applications/Safari.app/Contents/MacOS/Safari
```
STARTING ON OS X MAVERICKS, KEXTS MUST BE SIGNED

Similar to Windows, this aims to prevent unauthorized code from being loaded into ring-0.

Kernel extension could not be loaded

The kernel extension at "/Library/Extensions/unsigned.kext" can't be loaded because it is from an unidentified developer. Extensions loaded from /Library/Extensions must be signed by identified developers.

Attempting to load an unsigned kext
(un)SIGNED KEXTS

BUT I REALLY WANT MY UNSIGNED KEXT TO LOAD!

=> LET'S ABUSE A DESIGN FLAW (SEE: HTTP://REVERSE.PUT.AS/ FOR MORE DETAILS).

```c
//check signature
sigResult = checkKextSignature(theKext);

//invalid signature?
if(sigResult != 0)
{
    //error msg
    OSKextLogCFString("ERROR: invalid signature, will not load");

    //bail
    goto finish;
}

//load kext
OSKextLoadWithOptions(theKext);
```

```bash
sh-3.2# kextload unsigned.kext
sh-3.2# kextstat | grep -i unsigned
0xffffff7f81bb0000 com.synack.unsigned
```

user mode signature verification

patch/unsigned kext loading
BUT I REALLY WANT MY UNSIGNED KEXT TO LOAD (AGAIN)!

-> IN-MEMORY PATCHING IS A PAIN, THERE'S GOTTA BE SIMPLER WAY

//unload
  kext
daemon
#
  launchctl
  unload
  /System/Library/LaunchDaemons/com.apple.kextd.plist

//try
to
load
(unsigned)
driver
#
  kextload
  -v
  unsigned.kext
  Can't contact kextd; attempting to load directly into kernel

//profit :)
#
  kextstat
    | grep
    -i
    unsigned
    378 0 0xffffffff7f82877000 0x2000 0x2000 com.synack.unsigned (1) <4>

直接加载 == 跳过检查
THE CURRENT SITUATION

Lots of Macs + Feeble Anti-Malware Protections + OS X Malware + Limited OS X Malware Analysis Tools

By gaining a solid understanding of the Mac boot process and how it may be targeted by persistent malware, we can (better) protect ourselves.

....A new analysis tool will help us too!
MAC SYSTEM STARTUP

...from power on, to the desktop
THE PROCESS

LET’S DIVE IN!

> “THE BOOT [OR STARTUP] PROCESS IS A LONG AND ARDVOUS FLOW”

- Power On/Boot: pre-OS execution
- Kernel: load the OS’s core
- launchd: load daemon & agents
- loginwindow: auth the user

JONATHAN LEVIN
MAC OS X AND IOS INTERNALS: TO THE APPLE'S CORE
POWER ON/BOOT

EARLY STARTUP

→ BOOTING: FROM POWER ON UNTIL THE CPU BEGINS EXECUTING THE KERNEL (OS X)

Power On

BootROM

init hardware and select the OS

Boot.efi

load the kernel

OS X

KERNEL
init the OS's core
BOOTROM

BOOTROM; AKA ‘FIRMWARE’

> Composed of the P.O.S.T (Power-On Self Test) and a E.F.I. (Extensible Firmware Interface),

The BootROM verifies memory, initializes system hardware, and selects the OS partition.

Diagram:
- Verify memory
- Init hardware
- Select OS partition

BootROM version:
- MBP112.0138.002
- SMC Version (system): 2.19f3
BOOT.EFI

BOOT.EFI: LOAD UP THE KERNEL

Once the Bootrom is finished, control is passed to Boot.Efi on the selected partition.

- init device tree
- locate the kernel
- load boot kexts
- jump to kernel

.efi binaries are PE files

```
# hexdump -C /System/Library/CoreServices/boot.efi
00000000 ... MZ..............|
00000010 ... ........@.......|
00000020 ... ................|
00000030 ... ................|
00000040 ... |is program canno|
00000050 ... |t be run in DOS|
00000060 ... |mode...$......|
00000070 ... |PE..d......R....|
00000080 ...
```
BOOTING OS X

BOOTING OS X: FROM THE KERNEL TO THE DESKTOP

- Kernel, launchd, and finally logging in

1. The Kernel
   - init OS's core

2. launchd
   - load daemons/agents

3. LoginWindow
   - auth the user

User's Session

DESKTOP, etc
The kernel (XNU) must initialize its various subsystems before kicking off the user mode portion of the OS.

- Init Mach components
- Init IOKit (load kexts for devices)
- Init BSD subsystems
- Kick off launchd

//path to launchd
static char init_program_name[128] = "/sbin/launchd";

// Kick off launchd
void load_init_program(proc_t p)
{
    // Copies 'init_program_name' into 'init_exec_args'
    ...  
    // Launch it
    execve(p, &init_exec_args, retval);
}

Spawning launchd
LAUNCHD IS BASICALLY LINUX’S INIT

As the first program to launch, it boots the user component of the system, then maintains it.

- Launch all daemons
- Launch all agents (after login)
- Restart/maintain as needed

```
# ps -p 1
PID TTY  CMD
1   ?? /sbin/launchd
```
LOGINWINDOW: THE LOGIN GUI

- AUTHENTICATES THE USER, SETS UP THE ENVIRONMENT, THEN MANAGES THE SESSION

authenticate the user (via OpenDirectory)

sets up the user’s environment/preferences

manages user’s session (logout/restart/shutdown)
METHODS of PERSISTENCE

...where malware may live
The boot process affords several opportunities—often highly complex, though very insidious and difficult to detect.

- Re-flash the boot-ROM?
- Install malicious EFI components?
- Replace/patch the boot.efi?

Go read: ‘Mac EFI Rootkits’ by Loukas K (Snare)
KERNEL EXTENSIONS

KERNEL EXTENSIONS ARE LOADED AUTOMATICALLY (RING-0)

AN IDEAL SPOT FOR ADVANCED OS X MALWARE TO PERSIST

create a KEXT (in XCODE)  
copy to KEXT directory  
set ownership to root  
rebuild kernel cache

also: /System/Library/Extensions

```
# cp -R persist.kext /Library/Extensions
# chown -R root:wheel /Library/Extensions/persist.kext
# kextcache -system-prelinked-kernel
# kextcache -system-caches
```

installing a kext
LAUNCH DAEMONS/AGENTS

LAUNCH DAEMONS & AGENTS SIMILAR TO WINDOWS SERVICES

-> MALWARE WILL OFTEN ABUSE THESE TO GAIN AUTOMATIC REBOOT/LOGIN PERSISTENCE

daemons and agents are all launched by launchd

- daemons
  - non interactive, launched pre-login
  - /System/Library/LaunchDaemons
  - /Library/LaunchDaemons

- agents
  - interactive, launched post-login
  - /System/Library/LaunchAgents
  - /Library/LaunchAgents
  - ~/Library/LaunchAgents
LAUNCH DAEMONS/AUTENTS

DAEMONS & AGENTS ARE REGISTERED VIA PROPERTY LISTS

→ THESE PLISTS INSTRUCT LAUNCHD HOW/WHEN TO LOAD THEM

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC ...>
<plist version="1.0">
  <dict>
    <key>Label</key>
    <string>com.example.persist</string>
    <key>ProgramArguments</key>
    <array>
      <string>/path/to/persist</string>
      <string>args?</string>
    </array>
    <key>RunAtLoad</key>
    <true/>
  </dict>
</plist>
```
Cron Jobs

Cron jobs can be used to automatically perform actions.

Malware writers coming from Linux-based backgrounds love this technique.

Creating/installing a cron job:

```bash
$ echo "* * * * * echo "I'm persisting\"" > /tmp/persistJob
$ crontab /tmp/persistJob
$ crontab -l
* * * * * echo "I'm persisting"
```

Can use @reboot, @daily, etc.
LOGIN/LOGOUT HOOKS ARE DEPRECATED, BUT STILL WORK

→ ALLOW A SCRIPT TO BE AUTOMATICALLY EXECUTED AT LOGIN AND/OR LOGOUT

```
# defaults write com.apple.loginwindow LoginHook /usr/bin/hook.sh
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist ...>
.plist version="1.0">
  <dict>
    <key>LoginHook</key>
    <string>/usr/bin/hook.sh</string>
  </dict>
</plist>
```
LOGIN ITEMS

LOGIN ITEMS ARE THE 'LEGITIMATE' WAY TO PERSIST

-> CAN BE VIEWED IN THE GUI: SYSTEM PREFERENCES -> USERS & GROUPS -> LOGIN ITEMS

~/Library/Preferences/com.apple.loginitems.plist

```xml
<dict>
  <key>com.apple.LSSharedFileList.Binding</key>
  <data>ZG5pYgAAAAACAAAAAAAAAAAAAAAAAAAAAAAAAAAA...</data>
  <key>com.apple.LSSharedFileList.ItemIsHidden</key>
  <true/>
  <key>com.apple.loginitem.HideOnLaunch</key>
  <true/>
  <key>Name</key>
  <string>iTunesHelper</string>
</dict>
```

base64 encoded path, etc
SANDBOX'D APPS CANNOT EDIT THE SYSTEM PREFERENCES LIST

- So Apple designed a new mechanism...that doesn't show up in any GUI

---

**STEP 1**

Copy the app to persist into:

<MAIN>.APP/CONTENTS/LIBRARYLOGINITEMS/

**STEP 2**

In the main app, invoke `SMLoginItemSetEnabled()` with the identifier of the app to persist

```swift
//enable auto launch
SMLoginItemSetEnabled((__bridge CFStringRef)"com.company.persistMe", YES);
```
RE-OPENED APPS

ON LOGIN, ANY OPENED WINDOWS OR APPS WILL BE RESTORED

-> AN ATTACKER COULD POSSIBLY ABUSE THIS FUNCTIONALITY TO PERSIST MALWARE

```xml
<dict>
    <key>TALAppsToRelaunchAtLogin</key>
    <array>
        <dict>
            <key>BundleID</key>
            <string>com.apple.terminal</string>
            <key>Hide</key>
            <false/>
            <key>Path</key>
            <string>/Applications/Utilities/Terminal.app</string>
        </dict>
    </array>
</dict>
```

~/Library/Preferences/ByHost/com.apple.loginwindow.<hardware UUID>.plist
STARTUP ITEMS

'STARTUP ITEMS' ARE DEPRECATED, BUT STILL WORK

→ CAN BE ABUSED TO AUTOMATICALLY EXECUTE A SCRIPT AT EACH REBOOT

```
#!/bin/sh
./etc/rc.common

StartService()
{
    anything here
}

RunService "$1"
```

```
{
    Description = "anything";
    Provides = ("<name>");
}
```

Persistent Script

in either:
/System/Library/StartupItems
/Library/StartupItems

StartUpParameters.plist

```
Persist

StartupParameters.plist
```
‘RC SCRIPTS’ CAN AUTOMATICALLY PERFORM ACTIONS

- Malware writers coming from Linux based backgrounds may abuse this technique.

```
# vim /etc/rc.common
...
add any commands (at end)
```

How easy is this?

Modifying `rc.common`
LAUNCHD.CONF

LAUNCHD.CONF IS THE CONFIGURATION FILE FOR LAUNCHD

-> CAN BE ABUSED FOR PERSISTENCE, BY INJECTING COMMANDS TO BE EXECUTED BY LAUNCHCTL

```bash
# echo bsexec 1 /bin/bash <anything.script> > /etc/launchd.conf
```

launchd.conf

'bsexec' is a launchctl command that executes other commands... perfect!

file does not exist by default
BINARY INFECTION IS ONE OF THE OLDEST PERSISTENCE TECHNIQUES

→ MACH-O BINARIES CAN BE INFECTED IN A MYRIAD OF WAYS
APPLICATION SPECIFIC

PERSISTENCE BY TARGETING APP SPECIFIC LOGIC/FRAMEWORKS
- For example, plugins or extensions are often abused for persistence

```c
#include <syslog.h>

//automatically invoked when loaded
__attribute__((constructor)) void myConstructor()
{
  //dbg msg
  syslog(LOG_ERR, "loaded in process %d", getpid());
  return;
}
```

'evil plugin' (fG!)
APPLE HAS FULLY DEPRECATED/ NOW PREVENTS SEVERAL METHODS

> MALWARE WRITERS: TIME TO UPDATE YOUR CODE! ;)

couldn’t get these working on Mavericks

~/.MacOSX/environment.plist

/Library/Preferences/com.apple.SystemLoginItems.plist

loginwindow

‘AutoLaunchedApplicationDictionary’

DYLD_INSERT_LIBRARIES (signed)
PERSISTENT MALWARE

...careful now ;}
CALL ME

CALL ME IS DISTRIBUTED VIA MALICIOUS WORD DOCUMENTS
¬ TARGETED, IT PROVIDES THE ABILITY TO UPLOAD/DOWNLOAD FILES AS WELL AS EXECUTE
ARBITRARY COMMANDS

```bash
# fs_usage -w -filesystem | grep OSX_CallMe

open /Library/LaunchDaemons/.dat035f.000
WrData[A] /Library/LaunchDaemons/.dat035f.000
rename /Library/LaunchDaemons/.dat035f.000
  -> /Library/LaunchDaemons/realPlayerUpdate.plist

$ ls /Library/LaunchDaemons/real*
realPlayerUpdate.plist

$ ps aux | grep -i real
root 0:00.06 /Library/Application Support/.realPlayerUpdate
```

fs_usage is like ‘fileMon’

launch daemon persistence
FLASHBACK EXPLOITS A JAVA BUG TO INFECT 1/2M+ MACS

-> INJECTS ADS INTO USERS' HTTP/HTTPS STREAMS

$ less ~/Library/LaunchAgents/com.java.update.plist

```
<?xml version="1.0" encoding="UTF-8"?>
...
<dict>
  <key>Label</key>
  <string>com.java.update.plist</string>
  <key>ProgramArguments</key>
  <array>
    <string>/Users/user/.jupdate</string>
  </array>
  <key>RunAtLoad</key>
  <true/>
...
```

(user) launch agent persistence

malware's binary

auto run at login
CRIS

CRISIS IS A “ROOTKIT USED BY GOVERNMENTS”

-> DELIVERED THRU TARGETED EXPLOITS, IT COLLECTS AUDIO, IMAGES, SCREENSHOTS, AND KEYSTROKES

IDAPseudo disassembly

```objc
-[RCSMUtils createLaunchAgentPlist:forBinary:]
call  NSHomeDirectory
mov  [esp+0Ch], eax
lea  edx, @"Library/LaunchAgents/com.apple.mdworker.plist"
mov  [esp+10h], edx
lea  edx, "%@/%@"
mov  [esp+8], edx
mov  [esp+4], stringWithFormat_message_refs
mov  [esp], NSString_clsRef
callobjc_msgSend
```

```
[NSString stringWithFormat:"%@/%@",
 NSHomeDirectory(), @"Library/LaunchAgents/com.apple.mdworker.plist"]; (user) launch agent persistence
```
JANICAB IS SIGNED AND WRITTEN IN (COMPILED) PYTHON

ABUSING A RIGHT-TO-LEFT OVERRIDE (RLO) TRICK, IT COLLECTS AUDIO AND SCREENSHOTS

```python
""
  add	
  to	
  crontab	
  ""

#add the script to crontab
subprocess.call("echo \"* * * *
    python ~/.t/runner.pyc \" >>/tmp/dump", shell=True)

#import the new crontab
subprocess.call("crontab /tmp/dump", shell=True)
subprocess.call("rm -f /tmp/dump", shell=True)
```

# added cron job
```
crontab -l
* * * * python ~/.t/runner.pyc
```

RLO trick

added cron job

cron job persistence

janicab's installer.py

**name & extension:**

<table>
<thead>
<tr>
<th>RecentNews.ppa.pdf</th>
</tr>
</thead>
<tbody>
<tr>
<td>hide extension</td>
</tr>
</tbody>
</table>

```
$ crontab -l
* * * * python ~/.t/runner.pyc
```
KITMOS was discovered on an ‘African Activist’s Mac’

-> It’s signed, and takes screenshots and uploads them to a remote C&C server

; build path for malware’s launch agent plist

- [FileBackupAppDelegate checkAutorun]
  mov  dword ptr [esp+18h], 0
  mov  dword ptr [esp+14h], 0
  mov  [esp+10h], ebx
  mov  dword ptr [esp+0Ch], 0
  mov  dword ptr [esp+8], 0
  mov  [esp+4], eax ; _kLSSharedFileListItemLast_ptr
  mov  [esp], edi ; _LSSharedFileListCreate
  call LSSharedFileListInsertItemURL
YONT00

YONT00 tricks users by masquerading as an ‘HD video codec’. It installs plugins (extensions) for Safari, Chrome, and Firefox to inject ads.

```
;create paths for malicious plugins
lea   edi, cfstr_InstallingExtensions ; "Installing extensions"
lea   ebx, cfstr_Ok                  ; "Ok"
...
+[ExtensionsInstaller installSafariExtension:]
"~/Library/Safari/Extensions/Extensions.plist"
+[ExtensionsInstaller installFirefoxExtension:]
"~/Library/Application Support/Mozilla/Extensions"
+[ExtensionsInstaller installChromeExtension:]
"~/Library/Application Support/Google/Chrome/External Extensions"
```

Persistence via browser plugins

Extracted from IDA disassembly
RENEPO (OPENER)

RENEPO IS AN OLDER SAMPLE, THAT’S WRITTEN AS A SCRIPT

-> DISABILITIES SECURITY MECHANISMS, AND CAN DOWNLOAD/INSTALL OTHER
"HACKER TOOLS" (E.G. PASSWORD CRACKERS)

```
# less OSX_Renepo

scriptpath=`pwd`
scriptfolder=`basename $scriptpath`
scriptname=`basename $0`

mkdir /System/Library/StartupItems/"${scriptname}"
cp "${scriptpath}"/"${scriptname}"
    /System/Library/StartupItems/"${scriptname}"/"${scriptname}"

# The lines below echo out the StartupParameters.plist file.
echo "<plist version="0.9">" >>
    /System/Library/StartupItems/${scriptname}/StartupParameters.plist
...
```

startup items persistence
MAC PROTECTOR

MAC PROTECTOR IS FAKE ("ROGUE") AV PRODUCT

→ LEVERAGES SAFARI'S "OPEN 'SAFE' FILES AFTER DOWNLOADING" TO DOWNLOAD AND BEGIN EXECUTION

```
<dict>
  <key>Alias</key>
  <data>
    ZG5pYgAAAAACAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...
  </data>
  <key>Name</key>
  <string>MacProtector</string>
</dict>
```

Login item persistence

base64 encoded path, etc

`~/Library/Preferences/com.apple.loginitems.plist`
CLAPZOK

CLAPZOK IS A MULTI-PLATFORM VIRUS
-> INFECTS EXECUTABLE BINARIES BY INSERTING VIRAL CODE, THEN HIJACKING THE ENTRY POINT

from http://reverse.put.as/ (fG!)

entry point LC

binary infection persistence
KNOCK KNOCK (who's there?)

...‘autoruns’ for OS X
KNOCK KNOCK'S GOAL & DESIGN

KNOCK KNOCK FINDS STUFF THAT WILL AUTOMATICALLY EXECUTE

⇒ ITS OPEN-SOURCE, PLUGIN-ORIENTED, DESIGN WILL ENCOURAGE COLLABORATION AND EVOLUTION
KNOCK KNOCK’S PLUGINS

PLUGINS ARE DESIGNED TO SCAN A PARTICULAR TECHNIQUE

⇒ FOR EXAMPLE, LAUNCH DAEMONS/AGENTS, KERNEL EXTENSIONS, LOGIN ITEMS, ETC

```python
#launch daemon directories
LAUNCH_DAEMON_DIRS = ['/System/Library/LaunchDaemons/', '/Library/LaunchDaemons/']

#init results
results['launchAgents'] = []

#get all files in each launch daemon directory
for directory in LAUNCH_DAEMON_DIRS:
    launchItems.extend(glob.glob(directory + '*'))

#iterate over these files (plists)
# ⇒ save the program of those that have ‘RunAtLoad’ set
for item in launchItems:
    plistData = utils.loadPlist(item)
    if plistData['RunAtLoad']:
        results.append(file.File(plistData['ProgramArguments'][0]))
```

launch daemon plugin
KNOCK KNOCK IS CURRENTLY COMMAND LINE ONLY

EXECUTES ALL PLUGINS, (OR JUST ONE), AND BY DEFAULT WILL IGNORE APPLE-SIGNED BINARIES

# python knockknock.py -p launchDandA

who's there?

# python knockknock.py -p cronjob

who's there?

[cronJobs]
* * * * *
python ~/.t/runner.pyc

hash: 544539ea546e88ff462814ba96a+f1a

cron jobs (janicab)

[launchAgents]
Little Snitch Agent
path: /Library/Little Snitch/Little Snitch Agent.app/Contents...
hash: 1a4a575ba98acd455bbb0712abee8df7

launch daemons & agents (callme)
SOME CONCLUSIONS

MACS ARE NOT THAT SECURE

+ MAC MALWARE IS A REALITY

= ...BUT KNOCK KNOCK CAN HELP!
(me) chillin in the 808

www.synack.com

patrick@synack.com

@patrickwardle

github.com/synack/knockknock