# Theories on Persistence

Neel Mehta

## The plan (aka an outline)

- Some basics (a framework to think about persistence)
- Persistence mechanisms for Windows
    - Commonplace persistence (why I hate malware)
    - Building on variations, introducing OS idiosynchracies and layers of the OS and hardware, going deeper.
    - Persistence possibilities off the CPU (AMT)
- Wrap up and deep thoughts.

## Some Basics

## Persistence in the malware sense is:

Bootstrapping future access to an already compromised system.

Can be broken down into:

- A persistence mechanism (restarts malware on reboot).
- A way to re-establish a useful presence on the system (initialization).

## Persistence Mechanism

Think Windows 'Run' keys, Linux init scripts, .plist files for OS X, etc...

Often, but not necessarily, tied to the OS boot process.

## Re-establish a useful presence

It might seem obvious, but...

The persistence mechanism should typically suit the initialization needs of an implant.

A persistence mechanism that lands you in a chroot jail as 'nobody' doesn't suit initialization well.

## Ever wonder why?

…

## Downsides of persistence

Leaves a semi-permanent, and often detectable/predictable footprint.

Most malware authors do not balance the benefits of persistence over time with the increased chances of detection.

Forensic analysis often focuses on persistence to find implants.

## Inevitable disappointment

Creative persistence is far from the norm, and most malware fall far short.

The most disappointing samples are dropped by great exploits, use stealthy comms, but drop to disk in clear and write by a Run key.

## 'Internet Hacking' is hard (really)

In its most effective forms, offense is far from a single discipline.

Can you:

- Find 0day, design/develop/maintain/QA exploits and implants, run infrastructure, ops, maintain perspective and oversight, and still find time for holiday shopping?

CNE programs beat the Lone Wolf, and the Lone Wolf beats...

## Flavors and Degrees of Persistence

Are you trying to persist through:
-context switches, application crashes, cold/warm boots, AV sig updates, OS reinstall or upgrade, hardware replacement, password resets, authorization revocation, network lockdown, full infrastructure rebuild/replacement?

Configuration and code are hard to bundle and update, how do you persist with both?

## Somehow, you found THE attacker!

Can you rewind to a clean state?

Finding unanticipated persistence must be balanced with the practical and paranoid.

## Did the attacker even need to persist?

How often is the target system rebooted?

How heavily monitored, and how homogenous an environment is it?

How valuable is access over multiple reboots, vs. initial access?

Is breaking in again even that hard for the attacker?

## Persistence Mechanisms on Windows

## Windows Persistence Basics

Create a new:

- Run registry key.
- Service.
- Item in a startup folder.
- INI file entry.
- Winlogon extension.
- COM control variant (shell extension, WMI provider, BHO, etc...)

Assumption: No one notices new software.

## File Replacement

For example:

- Replace an auto-started service dll in registry, then on startup actively proxy invocations of DllMain() and ServiceMain()

Or:

- Replace an existing COM control in HKCR\CLSID\{GUID}\InprocServer32, subclass the control's methods, and proxy.

Assumption: Replacing is less noticeable than adding.

## Displace instead of replace

Rather than replace, a file on disk, modify a registry key (ServiceDll, InprocServer32 default value, or equivalent), then proxy instantiation.

Assumes:

Modifying an existing reg key, and adding a new file, is less noticeable than replacing a system file (or simply adding new software).

## Displace via loader preference

Seen on Windows via DLL search order, Linux via LD_PRELOAD, and in any sufficiently complex loader.

DLL search order almost always favors the local directory over system32\. The shell (explorer.exe) is in C:\Windows, not system32.

Assumes: comctl32.dll in an odd path is less noticeable than an innocuous name and reg key.

## Other subystems

Print spooler drivers (popular)

Winlogon, LSA, Crypto providers, auth providers

.NET assemblies

Input method editors (IME's)

Sidebar gadgets

MIME types, protocol handlers

The catch all: various plugins!

## Subsystems with their own stack

Window messages
Image codecs
Directshow filters
WFP drivers
Filesystem filters
Any driver with IRP_MJ handlers

## Complicated loaders complicate *

WinSxS and PE manifests.

DLL redirection

- To what DLL does "api-ms-win-core-libraryloader-l1-1-0.SetDefaultDllDirectories" specifically refer to?

Lazy loading / symbol resolution

- Might used to selectively split dependencies between modules, or displace dependencies

## Compatility on x64

--WoW

- (take that everyone who sat on NTVDM 0day)

++WoW64

- Add address-space complexity in user-mode.
- 32 and 64-bit code segments co-exist in the same process at ring 3 - far JMP / CALL to swap segment (and default argument size)
- Loader implications for persistence ?

Also complicating loader behaviour:

- Internationalization / MUI (can be enabled on XP, but Vista+ by default)

## Persisting via loader intricacies

The basics don't change - you'll have to or introduce an anchor point, most likely a file on disk.

If you assume your loader will be found via persistence, you can obscure or delay detection of stage 2.

## INI File Redirection

You can redirect INI file settings, on a per-user basis, via a registry key (internals handled by CSRSS at runtime).

No need to modify the original INI, just the registry.

Some INI files deprecated in x64 (system.ini because WoW is gone), but there are others.

## Folder redirection

Introduced during the Vista / UAC transition.

Entire folders can be redirected wholesale, without modifying the originals, via registry keys (not visible at a FS level like NTFS junction points).

Do forensic tools account for this?

## Slide intentionally not mentioning ADS

## Use hardcoded paths for event-driven OS callbacks

Especially those absent by default:

Fxsst.dll is popular.

CPL files in system32 (post-Stuxnet).

Startup folders.

Autorun.inf / desktop.ini on fixed drives.

## Task scheduling

Scheduler / atsvc - pick a future time to run

## Persistence on boot / shutdown

When started, delete persistence and register a power event handler to catch shutdown.

On shutdown event, re-register persistence.

Works best for drivers, but far from perfect

- Reliable enough?
- No shield from inert forensics (unless you can detect that last shutdown).

## Moving Beyond the Really Boring

## Lower layers of persistence

Modify NTOSKRNL or its dependencies (like HAL), and cloak signs of this before the OS starts.

## Use hibernation to persist

Hiberfil.sys stores all physical memory on hibernation. Space is pre-allocated in perpetuity.

When to resume then?

- Magic bytes (hibr) indicate the OS should awake from hibernation, not just boot (see Ruff/Suiche analysis).

Peter Kleissner suggested modifying the MBR to tweak hiberfil.sys on boot.

## Use hibernation to persist

Modifying the MBR is overly noisy.

Instead, hook the OS after it marks 'wake' in hiberfil.sys, and re-mark it 'hibr', and add persistence to hiberfil.sys (see Ruff / Suiche analysis).

Not ideal (why does my system always resume from hibernation???).

## Go lower maybe?

NTLDR, ntdetect.com, MBR.

All are relatively noisy, but low enough to avoid most active detection. Payload must be small, the dropper must be file-system aware.

Overly used in recent years to bypass code-signing requirements for drivers on Win x64 (TDL4 for example).

## Even lower

Partition table or GPT - boot another OS, start a hypervisor/VMM (insert random Matrix reference).

Still obvious from a forensics perspective.

## Even lower

BIOS, option and extension ROM's for peripherals.

Almost always signed, but implementations are flawed - Ryan Smith's (Hustle Labs) work bypassing RSA signatures on Lenovo BIOS updates - Baythreat 2010.

## Beyond Main Memory

## Other peripherals

Many are flashable, like keyboards. Some are even permanently attached!

A trend toward IO virtualization will devalue this over time.

- Most peripherals use a structured protocol like USB.
- IOMMU's or VTd should be ubiquitious for peripherals that do DMA, but we're not there yet.

## Even lower?

Microcode - generally not feasible.

Intel microcode format is a black box (to me and my search engine anyways).

Some public work on AMD microcode:

- Uncovered hardware debugging features
- Identified and modifed instruction encoding maps.

Still tricky, + microcode won't survive cold boots.

# AMT

## If not lower, then laterally?

Intel AMT (part of vPro) - an enterprise management feature off on most consumer rdevices.

Runs off the main CPU, on the North Bridge, and is treated as ring -1 (more trusted).

Often bundled with other features of vPro that enterprises really need (VTx, TXT).

## AMT - terrifying or not?

OEM's turn it off on most consumer hardware, but when on:

- Loads before the OS, runs off CPU, and is flashable.
- Has its own dedicated persistent storage off disk.
- Remotely provisionable, even from factory settings, or via light touch (USB)

## AMT - terrifying or not?

AMT survives some severe HW failures, like the main disk dying (zombie-like).

Modern versions have no prescribed ON/OFF switch, the intention being that someone at the keyboard is unable to disable it (not in the BIOS or MEBX settings).

The OS has no visibility.

## AMT

- Provides:
  - Hardware KVM or VNC
  - Disk access
  - Network filtering / monitoring
  - Wake on Lan (even 802.11 I think).
  - Voice over WLaN.

- AMT is an embedded OS, including a full TCP/IP stack, HTTP, SSL, and much more.

## Zero-touch Configuration (ZTC)

Intel term for remote AMT provisioning; how are authentication and authorization done?

Buy an SSL cert provisioned for AMT/vPro from a CA in their root of trust to authenticate yourself.

Using this cert, you connect over a couple hardcoded ports (16992/3) to authenticate.

## ZTC Authn / Authz

For a feature this risky, the choice for authn sounds reasonable (to me).

Authorization is done through two checks:

1. Match the CN from the provided cert to the machine
2. Check a CNAME for the cert domain, and match it to the IP of the provisioning machine.

But, how does AMT match a machine to a domain?

## The catch...

A machine is matched to a domain name via DHCP option 15 (domain name). This must match the CN (domain) from the cert.

Also, a CNAME must be set up for the provisioning server, matching the cert as well.

Hrm.

## Rogue DHCP Servers

Typically a nuisance (when not exploiting DHCP clients).

However, some router vendors have features to block rogue DHCP by port.

- If you're willing to accept the added config/maintainence complexity.

## Vulns in AMT?

AMT firmware 3 years ago was ARC assembly (I found one toolchain with objdump).

Today, it's ARM running Linux - accessible and easyto decompose and reverse for anyone with embedded systems experience (1 hour, a hex editor, and IDA)

## Deep Thoughts :)

## Something to consider

Do your IR procedures have any chance of catching sophisticed persistence mechanisms?

Is a re-image enough?

## Final un(in)formed thoughts

Why is jailbreaking not only legal but good over time?

Smartphones run at least 2 OS's, the phone OS (Android, iOS, BBOS, Symbian) being the subordinate.

Persistence can have nothing to do with the OS or machine - like credentials, signing keys, poisoned software or hardware.

## Lastly

Are malware authors myopic or compulsive in their need for persistence?

Is the need for persistence part of the human condition? <-- deep

And a final plea: Please give us something interesting to look at (and get caught :).

Thanks!