From the Tunnels Below Gotham

An Uninvited Guest

(Who Won’t Go Home)

Black Hat DC 2010

Below Gotham Labs $-K_B \sum_i P_i \log_e(P_i)$

BBC NEWS

Woman lived hidden in Japan flat

A woman has been arrested in Japan for sneaking into a man's house and living in his wardrobe without him knowing.

Police found 58-year-old Tatsuko Horikawa living in a small storage space in the house in the southern city of Fukuoka.

The house belonged to a 57-year-old man, who had become suspicious after food disappeared from his fridge.

So he installed a surveillance system, which filmed the woman as she walked around in his absence.

On Wednesday afternoon police searched the house and found the woman in her cubby hole.

Police spokesman Hiroki Itakura Called the intruder "neat and clean"

Applying the Metaphor

With respect to anti-forensics,
one way to be "neat and clean:"

$$-K_B \sum_i P_i \log_e(P_i)$$

Applying the Metaphor

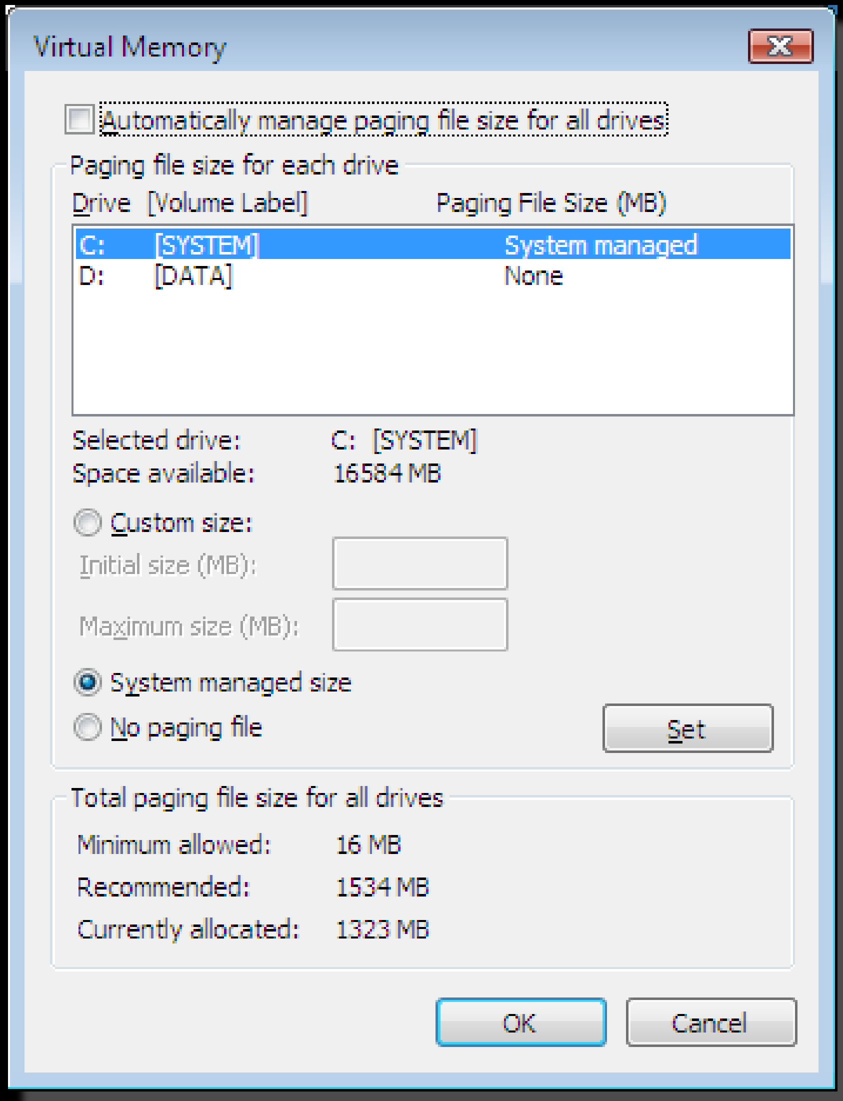
With respect to anti-forensics, one way to be "neat and clean:"

Avoid secondary storage

remain memory resident

$$-K_B \sum_i P_i \log_e(P_i)$$

If Properly engineered…
Not much outside of the page file
Can be captured post mortem



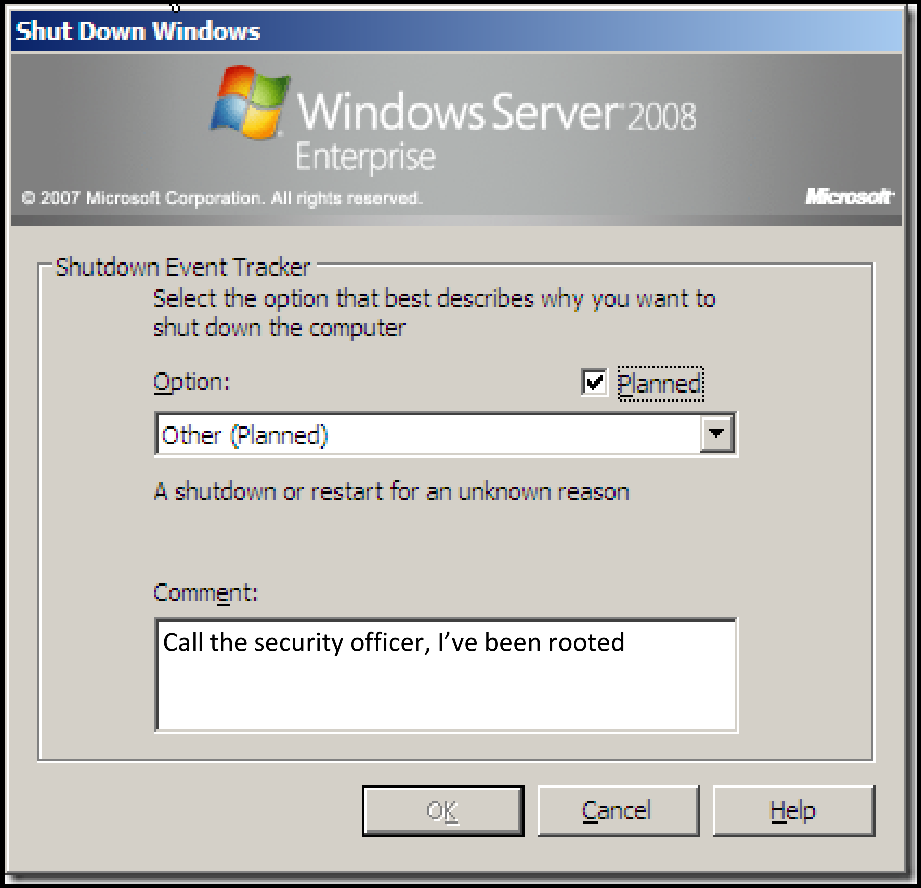
There are two challenges that this approach entails
These issues will define our primary design requirements

$$-K_B \sum_i P_i \log_e(P_i)$$

Evading Memory Analysis

```
Administrator: LiveKD - livekd
kd> !process 85113d90 2
PROCESS 85113d90  SessionId: 1  Cid: 0704    Peb: 7ffdf000  ParentCid: 0544
    DirBase: 13ffd000  ObjectTable: 955353b0  HandleCount: 271.
    Image: explorer.exe

        THREAD 84fa77f0  Cid 0704.0344  Teb: 7ffde000 Win32Thread: fcc5c348 WAIT: (W
            83733198  SynchronizationEvent

        THREAD 8361b7d8  Cid 0704.0f84  Teb: 7ffd9000 Win32Thread: ffa98490 WAIT: (U
            83392d18  SynchronizationEvent
            8476ef50  SynchronizationEvent

        THREAD 85040030  Cid 0704.06f4  Teb: 7ffd8000 Win32Thread: fe052858 WAIT: (U
            85147798  SynchronizationEvent
            83790918  SynchronizationEvent

        THREAD 836e9b38  Cid 0704.0a68  Teb: 7ffd3000 Win32Thread: fcd40e90 WAIT: (U
            84ee36e8  NotificationEvent
            83616a38  SynchronizationEvent

        THREAD 835cac88  Cid 0704.0dc4  Teb: 7ffdc000 Win32Thread: 00000000 WAIT: (W
            84fc58b0  QueueObject

kd>
```

Surviving System Restart



Design Goal #1

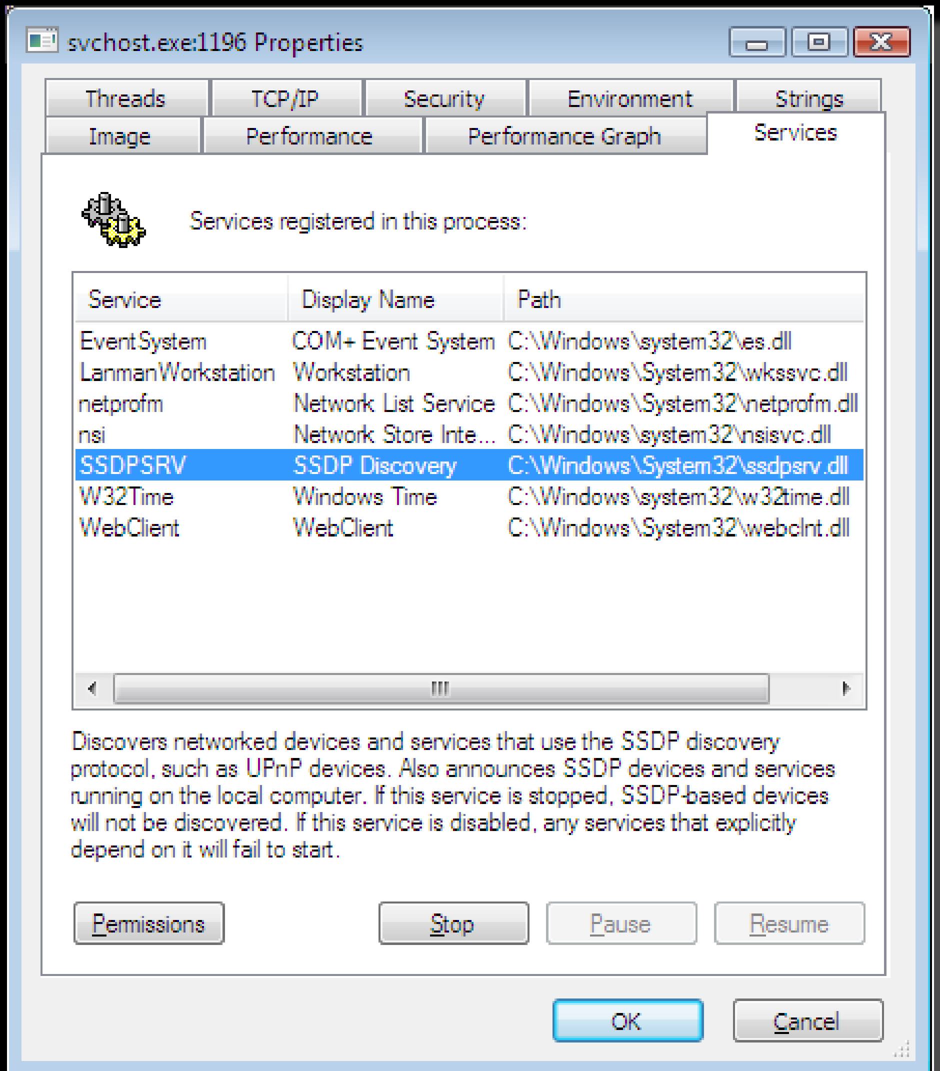
Achieve an Acceptable Level of Concealment

Different Approaches

- Hide in a Crowd
- Active Concealment
- Jump Out of Bounds

$$-K_B \sum_i P_i \log_e(P_i)$$

Hide in a Crowd



Basic Idea:

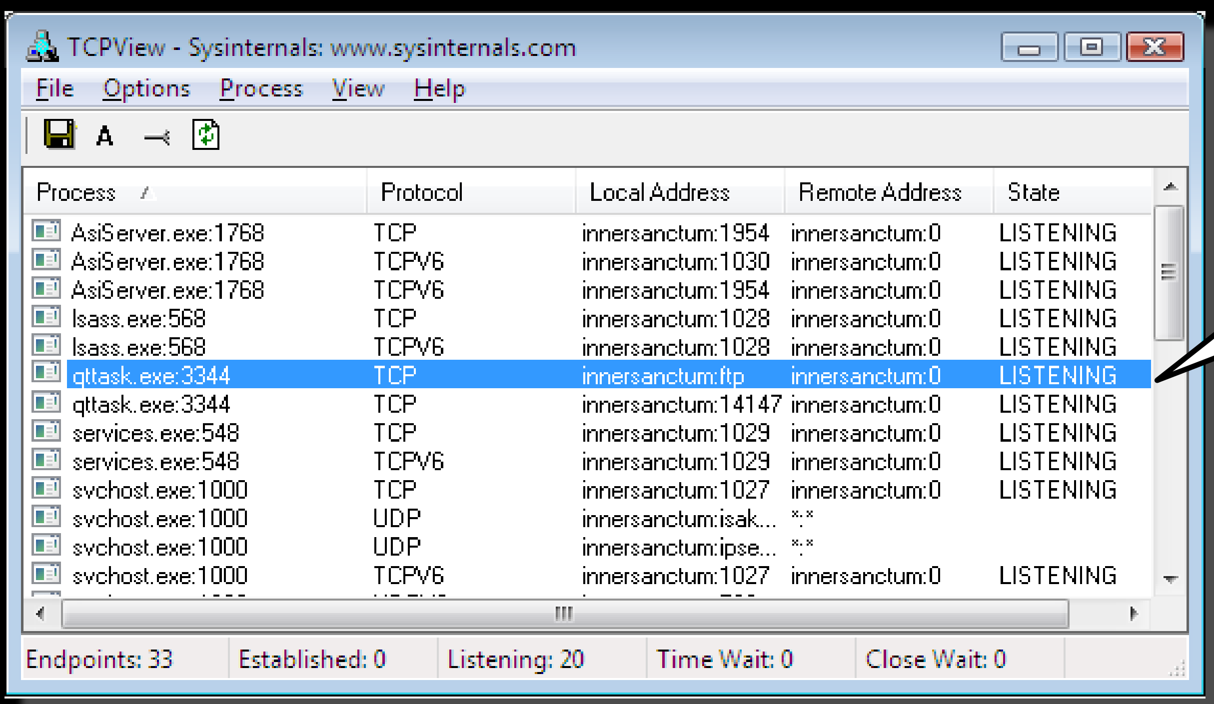
- This is the classic malware tactic
- Create a new process/thread
- Inject a module into an existing one
- Try to blend in with existing objects

$$-K_B \sum_i P_i \log_e(P_i)$$

Hide in a Crowd

Downsides:

- This tactic will **not** survive careful scrutiny
- Standard live response forensics will unearth this sort of rogue binary



Huh? QuickTime doesn't run an FTP service?

$$-K_B \sum_i P_i \log_e(P_i)$$

Active Concealment

Basic Idea:

- Install a module (e.g. a service, driver, injected library, etc.)
- Modify the system so that the module's presence isn't readily detectable

Strategy	Tactics	Objects Affected
Modify Static Elements	Hooking	IAT, SSDT, GDT, IDT, MSRs
	In-Place Patching	System Calls, Driver routines
	Detour Patching	System Calls, Driver routines
Modify Dynamic Elements	Alter Repositories	Registry Hives, Event Logs
	DKOM	EPROCESS, DRIVER_SECTION
	Patch Callback Tables	Module .data, .bss sections

Active Concealment

Downsides:

- You’re still creating bookkeeping data entries in OS data structures
- This is unavoidable (if you’re using native facilities to load the module)
- You may be able to hide from some tools, but not all of them simultaneously
- This is the basis for *cross-view detection*, which has proven effective

How RootkitRevealer Works

Since persistent rootkits work by changing API results so that a system view using APIs differs from the actual view in storage, RootkitRevealer compares the results of a system scan at the highest level with that at the lowest level. The highest level is the Windows API and the lowest level is the raw contents of a file system volume or Registry hive (a hive file is the Registry's on-disk storage format). Thus, rootkits, whether user mode or kernel mode, that manipulate the Windows API or native API to remove their presence from a directory listing, for example, will be seen by RootkitRevealer as a discrepancy between the information returned by the Windows API and that seen in the raw scan of a FAT or NTFS volume's file system structures.

Active Concealment

Current Trends in Memory Analysis:

- Sidestep the system-level APIs (which can be subverted by an intruder)
- Instead, forensic tools parse system data structures directly

MEMORYZE™

MANDIANT Memoryze can:

- image the full range of system memory (not reliant on API calls).
- image a process' entire address space to disk. This includes a process' loaded DLLs, EXEs, heaps, and stacks.
- image a specified driver or all drivers loaded in memory to disk.
- enumerate all running processes (including those hidden by rootkits). For each process,

Jump out of Bounds

Basic Idea:

- Eschew direct modification of the targeted operating system
- Migrate code outside of the OS proper and operate from this vantage point

Hiding Spot	Example
Host/Root Mode	Blue Pill Project http://bluepillproject.org/
SMM Mode	Embleton & Sparks Implementation http://www.blackhat.com/presentations/bh-usa-08/Embleton_Sparks/BH_US_08_Embleton_Sparks_SMM_Rootkits_Slides.pdf
AMT Environment	Ring -3 Rootkits http://www.blackhat.com/presentations/bh-usa-09/TERESHKIN/BHUSA09-Tereshkin-Ring3Rootkit-SLIDES.pdf

Jump out of Bounds

This Trend Highlights a Recurring Theme:

- Vendors try to counter malware by creating fortified regions of execution
- This seems like a great idea, until malware finds it way into these regions

Intel® Active Management Technology

Proactive alerting	**Isolate.** Proactively blocking incoming threats, Intel AMT System Defense contains infected clients before they impact the network while alerting IT when critical software agents are removed.

http://www.intel.com/technology/platform-technology/intel-amt

Jump out of Bounds

Downsides:

- These techniques tend to be **hardware dependent**
- You may not have any information on the target platform
- In some cases, all you'll have to start with is a bunch of open ports

Administrator: Command Prompt

```
C:\>nmap -sS 12.120.184.8

Starting Nmap 5.00 at 2009-10-26 13:35 Pacific Daylight Time

NSE: Loaded 0 scripts for scanning.
Initiating ARP Ping Scan at 13:35
Scanning 12.120.184.8 [1 port]
Completed ARP Ping Scan at 13:35, 0.18s elapsed (1 total hosts)
Initiating Parallel DNS resolution of 1 host. at 13:35
Completed Parallel DNS resolution of 1 host. at 13:35, 0.02s elapsed
Initiating SYN Stealth Scan at 13:35
Scanning 12.120.184.8 [1000 ports]
Discovered open port 80/tcp on 12.120.184.8
Discovered open port 8099/tcp on 12.120.184.8
Completed SYN Stealth Scan at 13:35, 0.26s elapsed (1000 total ports)
```

Engineering Concessions

Need to resolve conflicting directives

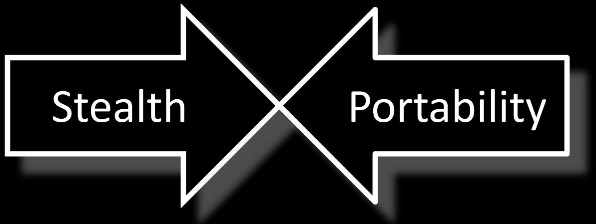
Stealth

On one hand, we wish to:

- Minimize the footprint we leave in system's data structures
- Establish a presence without creating a new process/thread
- Implement rootkit functionality without creating bookkeeping artifacts

Engineering Concessions

Need to resolve conflicting directives



On one hand, we wish to:

- Minimize the footprint we leave in system's data structures
- Establish a presence without creating a new process/thread
- Implement rootkit functionality without creating bookkeeping artifacts

At the same time, we'd like to:

- Remain as hardware agnostic as possible
- Use technology that's relatively transferable across the Intel platform
- Avoid writing custom driver code for a specific Intel/OEM chipset

Engineering Concessions

Professor G.H. Dorr:

“You, sir, are a Buddhist. Is there not a ‘middle’ way?”

The General:

“Mm. Must float like a leaf on the river of life...
and kill old lady.”

From *The Ladykillers*, Touchstone Pictures (2004)

One Potential Middle Path…

Shellcode

You Heard Me… Shellcode

The Benefits of Shellcode

x86 Shellcode offers a degree of **autonomy**

- It doesn't require address fix-ups to execute
- Therefore, it doesn't use the Windows loader
- Bookkeeping entries aren't generated in the kernel

```
find_kernel32:
    push esi
    xor eax, eax
    mov eax, fs:[eax+0x30]
    test eax, eax
    js find_kernel32_9x
find_kernel32_nt:
    mov eax, [eax + 0x0c]
    mov esi, [eax + 0x1c]
    lodsd
    mov eax, [eax + 0x8]
    jmp find_kernel32_finished
find_kernel32_9x:
    mov eax, [eax + 0x34]
    lea eax, [eax + 0x7c]
    mov eax, [eax + 0x3c]
find_kernel32_finished:
    pop esi
    ret
```

The Benefits of Shellcode

x86 Shellcode offers a degree of **autonomy**

- It doesn’t require address fix-ups to execute
- Therefore, it doesn’t use the Windows loader
- Bookkeeping entries aren’t generated in the kernel

x86 Shellcode also offers a modicum of **portability**

- It’s generally transferable across Intel motherboards

```
find_kernel32:
    push esi
    xor eax, eax
    mov eax, fs:[eax+0x30]
    test eax, eax
    js find_kernel32_9x
find_kernel32_nt:
    mov eax, [eax + 0x0c]
    mov esi, [eax + 0x1c]
    lodsd
    mov eax, [eax + 0x8]
    jmp find_kernel32_finished
find_kernel32_9x:
    mov eax, [eax + 0x34]
    lea eax, [eax + 0x7c]
    mov eax, [eax + 0x3c]
find_kernel32_finished:
    pop esi
    ret
```

The Benefits of Shellcode

x86 Shellcode offers a degree of **autonomy**

- It doesn't require address fix-ups to execute
- Therefore, it doesn't use the Windows loader
- Bookkeeping entries aren't generated in the kernel

x86 Shellcode also offers a modicum of **portability**

- It's generally transferable across Intel motherboards

Thus, we've reached a **middle ground**

- We want to rely as little as possible on native facilities
- Any facilities that we invoke can be used to detect us
- But we also want to avoid excessive hardware dependence

```
find_kernel32:
    push esi
    xor eax, eax
    mov eax, fs:[eax+0x30]
    test eax, eax
    js find_kernel32_9x
find_kernel32_nt:
    mov eax, [eax + 0x0c]
    mov esi, [eax + 0x1c]
    lodsd
    mov eax, [eax + 0x8]
    jmp find_kernel32_finished
find_kernel32_9x:
    mov eax, [eax + 0x34]
    lea eax, [eax + 0x7c]
    mov eax, [eax + 0x3c]
find_kernel32_finished:
    pop esi
    ret
```

The Drawbacks of Shellcode

- Raw assembly shellcode is **tedious** to write
- Logic can get lost in all those statements
- As a result, it can be prone to **subtle bugs**
- And also be generally difficult to maintain

Is there a way to sidestep all these issues?
Couldn't we just write shellcode in C?

Yes, we can!

Windows Shellcode Mastery

BlackHat Europe 2009

Benjamin CAILLAT

ESIEA - SI&S lab

caillat[at]esiea[dot]fr

bcaillat[at]security-labs[dot]org

Benjamin CAILLAT (ESIEA - SI&S lab) Windows Shellcode Mastery 1 / 172

http://www.blackhat.com/presentations/bh-europe-09/Caillat/BlackHat-Europe-09-Caillat-Wishmaster-slides.pdf

Types of Shellcode

Environment	Popular Example	Comments
User-Mode	Metasploit Shellcode Archive http://www.metasploit.com/shellcode/	Easier to implement Easier to detect, capture
Kernel-Mode	Deepdoor http://www.blackhat.com/presentations/bh-jp-06/BH-JP-06-Rutkowska.pdf	More powerful (Ring-0) More complicated

In the interest of stealth, I decided to employ **kernel-mode shellcode**

Design Goal #2

Persist (Without Persisting)

Related Concerns

- Is This Even Necessary?
- “Self-Healing” Software
- Persistence Modules

Why is persistence even an issue?

Enterprise Systems are often up for months

(Or, at least, that's how they're marketed)

But this isn't always the case…

Mission critical deployments managed by

- The Chicago Stock Exchange
- E*TRADE

Have been known to:

- Reboot their servers daily
- Implement rolling shutdowns periodically

http://staging.glg.com/tourwindowsntserver/CHX/technical.htm

One way to arrive at a potential solution
Is to examine the idea of “**self-healing**” software

A good example of a commercial implementation:
Absolute Software's **Computrace** product

Computrace is a loss prevention product
The client piece consists of two components

Application agent (`rpcnet.exe`)

- Runs as a nondescript service
- Phones home over an encrypted channel
- Manages “helper” applications
- Collects “inventory” data

Computrace is a loss prevention product
The client piece consists of two components

Application agent (`rpcnet.exe`)

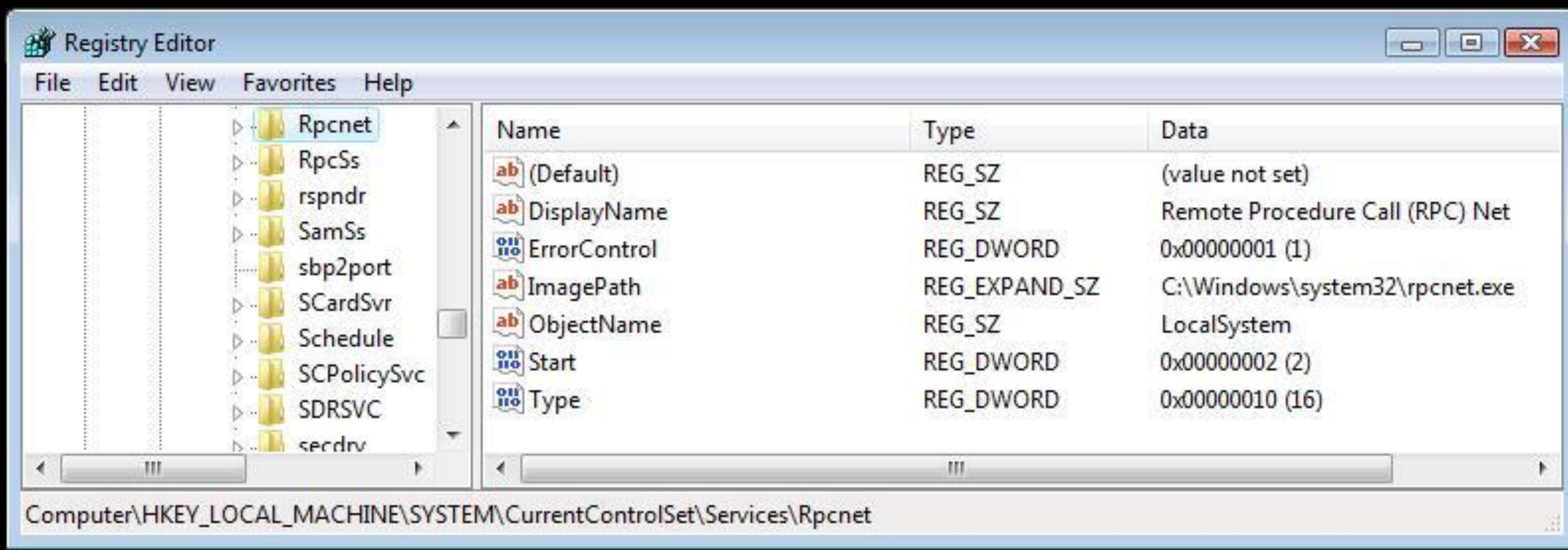
- Runs as a nondescript service
- Phones home over an encrypted channel
- Manages “helper” applications
- Collects “inventory” data

Persistence Module

- A secondary, independent, subsystem
- Embedded in disk partition gap (or firmware)
- Monitors for presence of Application Agent
- Re-installs agent if detects that it’s missing

The application agent hides in a crowd

It attempts to blend in with all of the other RPC services



It doesn’t take much to abstract these ideas
And then recast the two components as a rootkit

Application agent (`rpcnet.exe`)

- Runs as a nondescript service
- Phones home via encrypted channel
- Manages helper applications
- Collects inventory data

Persistence Module

- An independent subsystem
- Stashed on disk, or in firmware
- Monitors for presence of Agent
- Re-installs agent if missing

Original (White Hat) Package

Black Hat Incarnation

Rootkit (`kmd.sys`)

- Provides concealment services
- Implements Command & Control
- Performs Surveillance

Secondary Rootkit

- An independent subsystem
- Provides concealment services
- Monitors for presence of Rootkit
- Re-installs Rootkit if missing

$$-K_B \sum_i P_i \log_e(P_i)$$

Implementing the Backup Rootkit

- There are a number of ways that we could implement the secondary rootkit
- Each approach has its own set of tradeoffs

Possible Implementation	Comments
Backup Service/Driver	Robust, but conspicuous during a post-mortem
Bootkit (e.g. Stoned Again)	Less conspicuous, but still vulnerable to forensics
Firmware-Based Module	Very stealthy, but also fairly hardware dependent

More Engineering Concessions

Again, conflicting directives

Fault Tolerance

On one hand, we wish to:

- Survive a system restart

More Engineering Concessions

Again, conflicting directives

On one hand, we wish to:

- Survive a system restart

At the same time, we'd like to:

- Minimize the amount of forensic evidence on the target system
- Keep our runtime footprint as small as possible

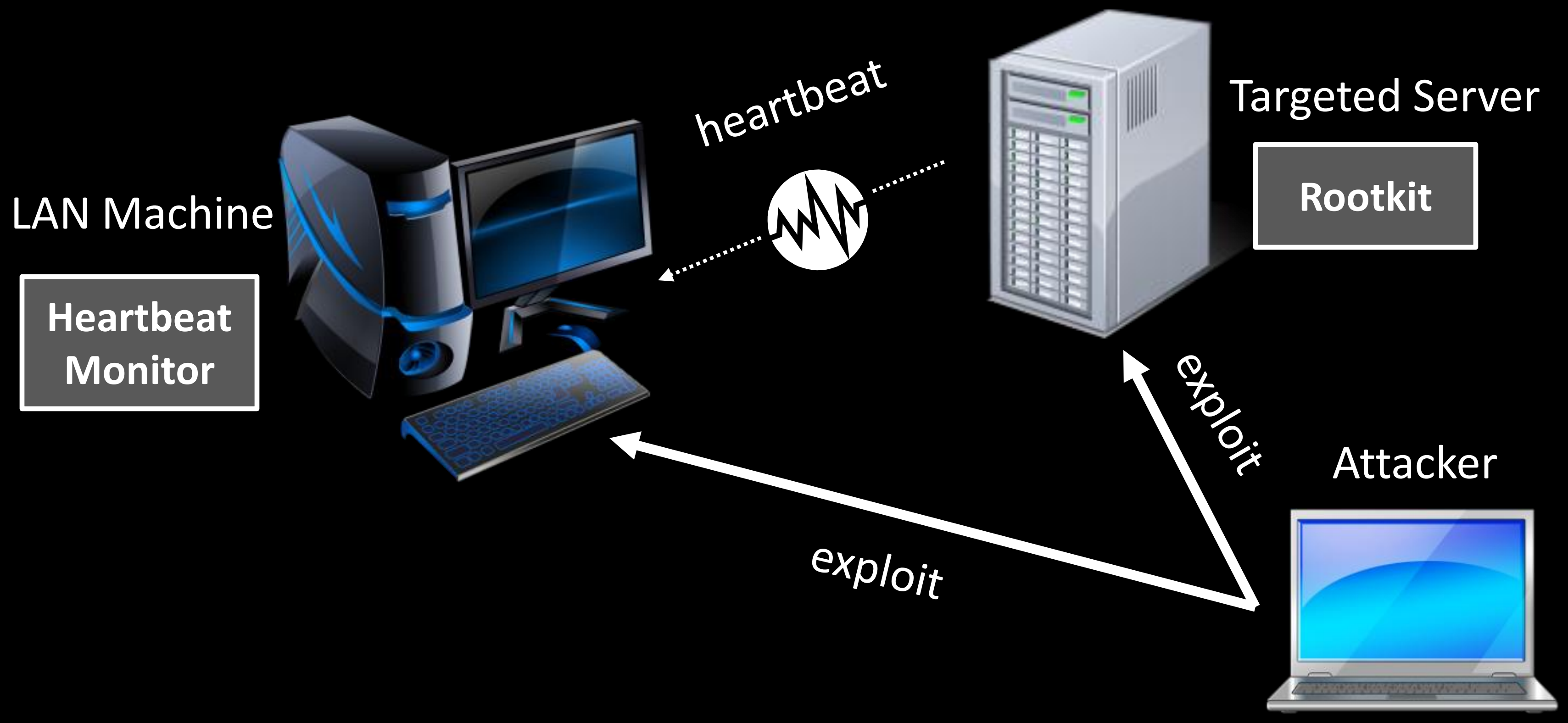
In other words…

We want a stealthy, fault-tolerant, and logistically tenable solution

$$-K_B \sum_i P_i \log_e(P_i)$$

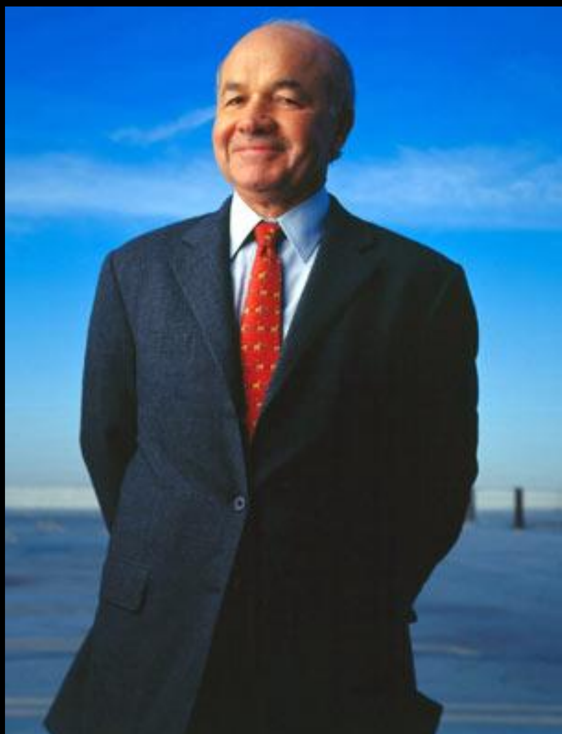
One Solution

Install the persistence module on another machine
Where it can monitor the target for a heartbeat signal



An Aside on Deployment

The Desktop Machines of High-Ranking Officials are *Soft* Targets



- Their status often provides them with admin rights
- But they’re not the most technically savvy people
- And they also install all sorts of 3rd party software
- So their machines are typically “noisy” to begin with
- In the mind of the admin, availability trumps security

Implementation

Kernel-Mode Shellcode in C

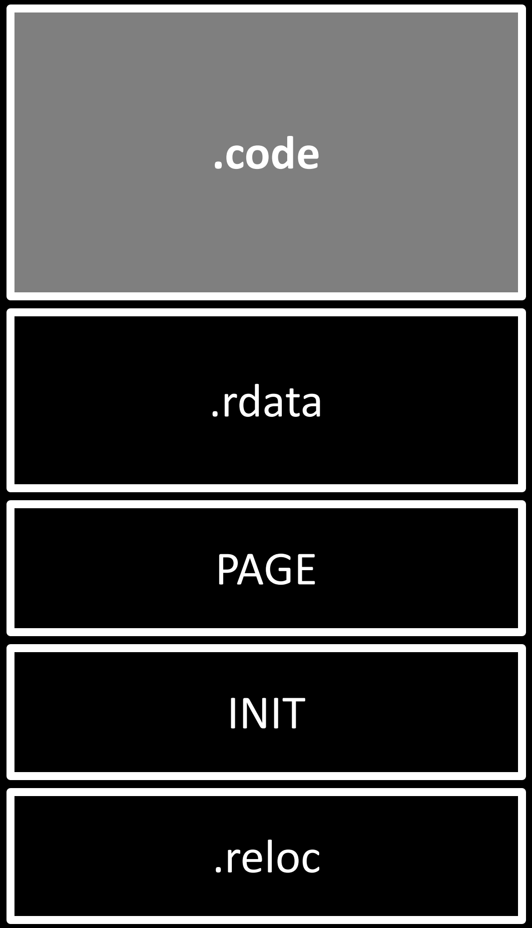
- Creating
- Extracting
- Deploying
- Executing

Creating Kernel-Mode Shellcode

Shellcode is merged into a single segment
Using Visual Studio preprocessor directives

```
#pragma section(".code",execute,read,write)
#pragma comment(linker,"/MERGE:.text=.code")
#pragma comment(linker,"/MERGE:.data=.code")
#pragma comment(linker,"/SECTION:.code,ERW")
#pragma code_seg(".code")
```

This section encapsulates both code and data



$$-K_B \sum_i P_i \log_e(P_i)$$

Creating Kernel-Mode Shellcode

Don't use conventional address resolution tables

- .idata
- .reloc

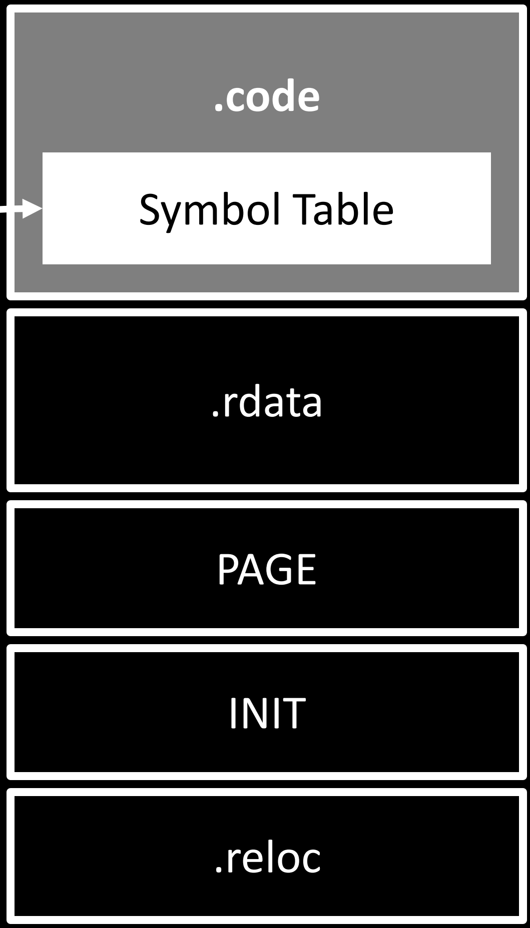
The shellcode has its own internal symbol table

This table is used to store the addresses of

- Imported Routines
- Local Routines (referenced in callbacks)

The internal symbol table is just a C structure

```
typedef struct GD_
{
    // "GD" as in Global Data
}GD;
```



Creating Kernel-Mode Shellcode

The composition of GD is imposed upon storage that's reserved for a routine

```
GD* gd = (GD*)GlobalDataRoutine();
```

The storage routine also returns the address of its data at runtime

```
unsigned int GlobalDataRoutine()
{
        unsigned int globalDataAddress;
        __asm
        {
                call endOfData
                //allocate shellcode data storage here
                endOfData:
                pop eax
                mov globalDataAddress,eax
        }
        return(globalDataAddress);
}
```

Creating Kernel-Mode Shellcode

An entry in this internal symbol table is referenced at runtime as follows:

address of entry = (Table's address) + (Offset into table)

```
; Call a routine whose address is stored in the symbol table
mov         eax, GobalDataRoutine
call        DWORD PTR [eax+24]
```

Notice how the table entry offset is predetermined at compile time

End Result:

A series of addresses is replaced by a single address and a bunch of offsets

Creating Kernel-Mode Shellcode

The internal symbol table is populated when the shellcode is loaded

In other words, the shell code takes over work traditionally done by the loader

- Most of the real work involves resolving external routines
- MSR Scandown is used to locate routines exported by `ntoskrnl.exe`
  http://www.uninformed.org/?v=3&a=4&t=sumry
- `AuxKlibQueryModuleInformation()` is also invoked when necessary

Note: using routines in `aux_klib.lib` will require makefile adjustments
This library is *not* mentioned in the WDK's default `makefile.new`

```
GETLIB=$(DDK_LIB_PATH)\ntoskrnl.lib $(DDK_LIB_PATH)\hal.lib $(DDK_LIB_PATH)\wmilib.lib
```

Creating Kernel-Mode Shellcode

The **SOURCES** file deviates slightly from the KMD standard

```
TARGETNAME=HeartBeat
TARGETPATH=.
TARGETTYPE=DRIVER
SOURCES=HeartBeat.c
INCLUDES=.
MSC_WARNING_LEVEL=/W3
USER_C_FLAGS=/Od /Oy /GS- /J /GR-  /FAc /TC
TARGETLIBS=$(DDK_LIB_PATH)\netio.lib
```

Really important settings

Also, to prevent the linker from treating warnings as errors
Change the following line in the WDK's default **makefile.new**:

```
LINKER_WX_SWITCH=/WX
```

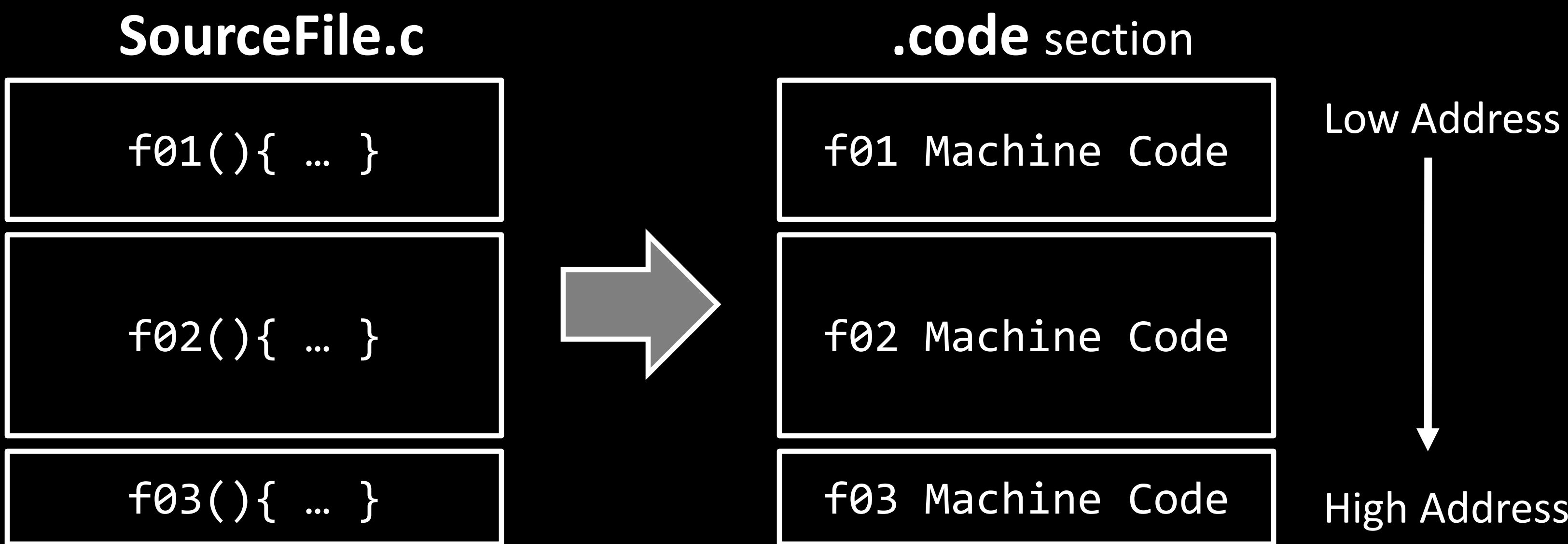
To

```
LINKER_WX_SWITCH=/WX:NO
```

Creating Kernel-Mode Shellcode

The **USER_C_FLAGS** build macro is crafted such that:

- Machine code for a routine is emitted when the compiler encounters it
- Thus, the first routine in the source will be located at the lowest address



Creating Kernel-Mode Shellcode

To see this in action…

Check out the **`shcode.h`** file, then compare it to **`HeartBeat.c`**

```
unsigned char ShCodeArray[]=
{
          // doDNSQueries()
          /* 00000000 */ 0x8B, 0xFF, 0x55, 0x8B, 0xEC, 0x83, 0xEC, 0x10, ...

          // getHashA()
          /* 00000270 */ 0xCC, 0xCC, 0xCC, 0xCC, 0x8B, 0xFF, 0x55, 0x8B, ...

          // walkExportList()
          /* 000002B0 */ 0xCC, 0xCC, 0xCC, 0xCC, 0x8B, 0xFF, 0x55, 0x8B, ...

          //...
```

Extracting Kernel-Mode Shellcode

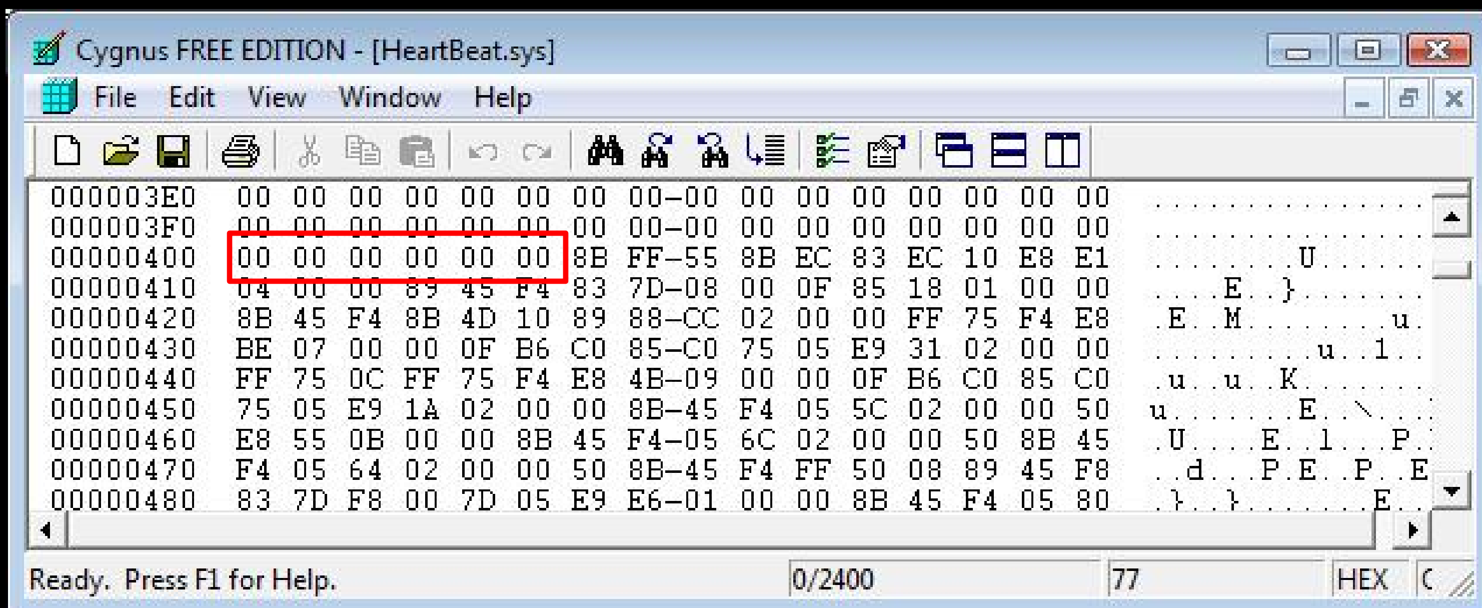
The shellcode's position in the driver can be found via **`dumpbin.exe`**

```
C:\>dumpbin.exe /headers kmd.sys

SECTION HEADER #1
         .code name
         3A4 virtual size
         1000 virtual address (00011000 to 000113A3)
         400 size of raw data
         400 file pointer to raw data (00000400 to 000007FF)
         0 file pointer to relocation table
         0 file pointer to line numbers
         0 number of relocations
         0 number of line numbers
```

Location of shellcode in **`.SYS`**

Extracting Kernel-Mode Shellcode

Once you've isolated the shellcode, you can extract it out with a hex editor



You can ignore the leading zero bytes (the code is position independent)

 www.belowgotham.com

$$-K_B \sum_i P_i \log_e(P_i)$$

Deploying Kernel-Mode Shellcode

Initially, I stayed within the confines of a Kernel-Mode Driver (KMD)
I defined a placeholder routine, consisting of junk instructions

```
void placeholder( )
{
        __asm _emit 0x90
        __asm _emit 0x90
        __asm _emit 0x90
        __asm _emit 0x90
        __asm _emit 0x90
        ...
        __asm _emit 0x90
        return;
}
```

Deploying Kernel-Mode Shellcode

At runtime the KMD would overwrite this dead space with shellcode

```
void placeholder( )
{
        __asm _emit 0x90
        __asm _emit 0x90
        __asm _emit 0x90
        __asm _emit 0x90
        __asm _emit 0x90
        ...
        __asm _emit 0x90
        return;
}
```

```
0x8B
0xFF
0x55
0x8B
0xEC
...
0xC2
0x04
```

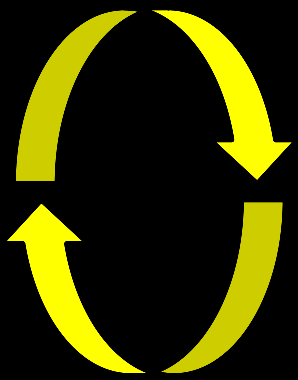
Deploying Kernel-Mode Shellcode

Then, the KMD launched the shellcode as a separate system thread

```
void placeholder( )
{
        __asm _emit 0x90
        __asm _emit 0x90
        __asm _emit 0x90
        __asm _emit 0x90
        __asm _emit 0x90
        ...
        __asm _emit 0x90
        return;
}
```

PsCreateSystemThread()

```
0x8B
0xFF
0x55
0x8B
0xEC
...
0xC2
0x04
```



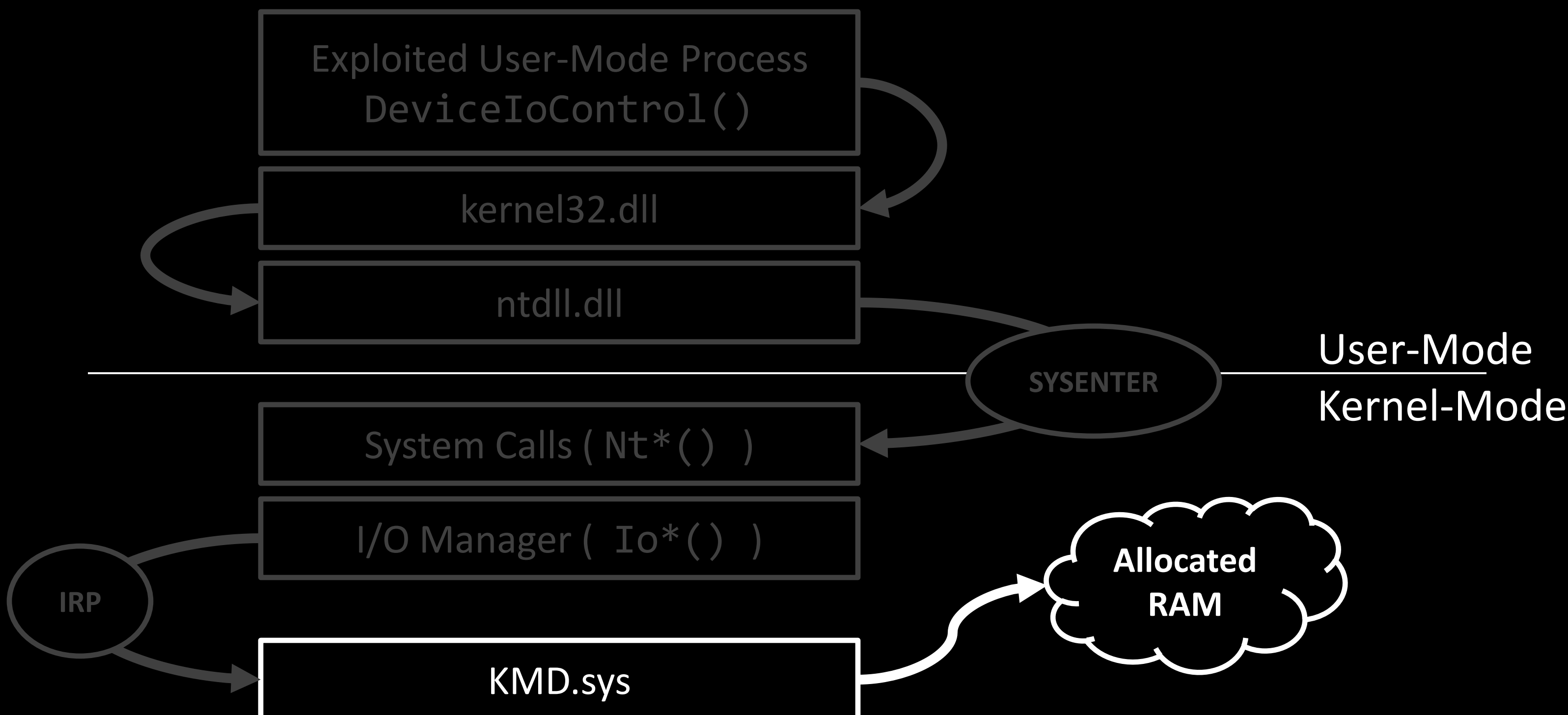
Deploying Kernel-Mode Shellcode

This approach is far too **conspicuous** for a production rootkit

But it's useful as a **testing area**, before you wade into deep water

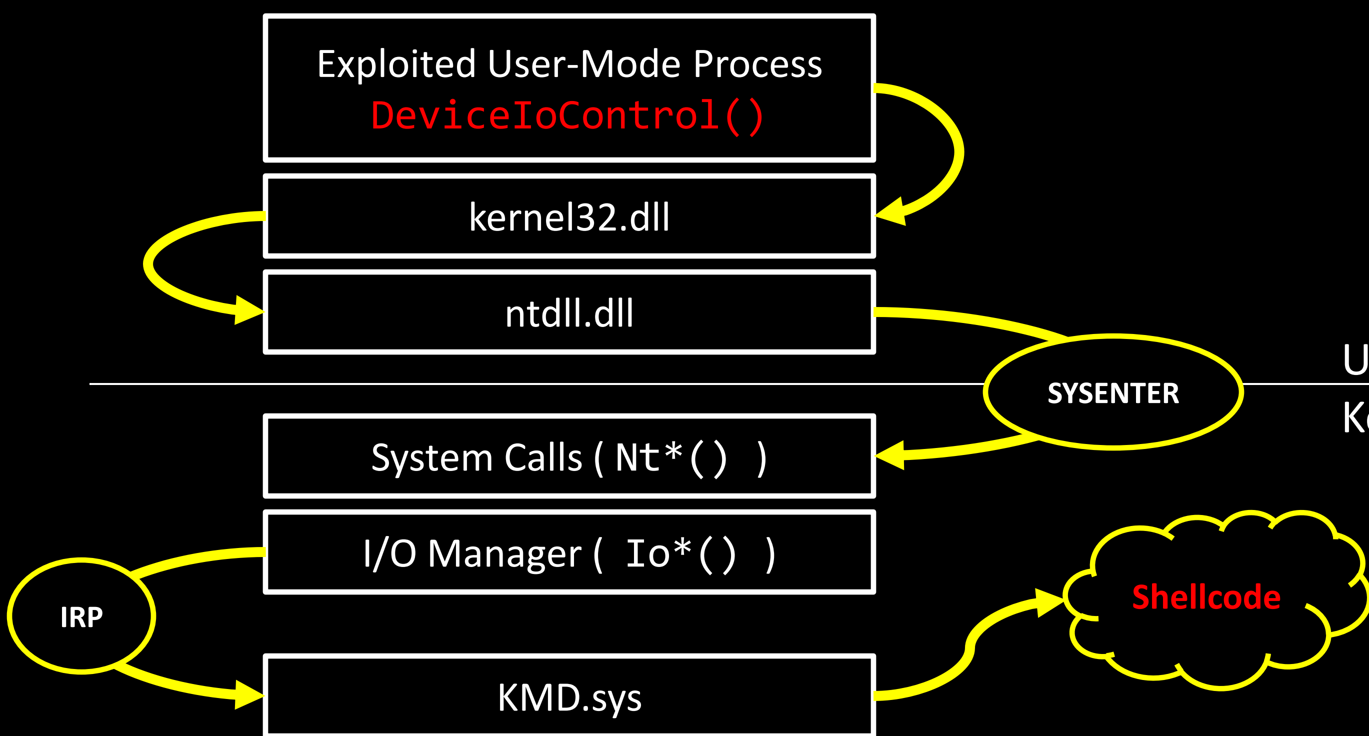
Deploying Kernel-Mode Shellcode

One alternative is to simply to load the shellcode into memory somewhere
Specifically, a KMD could allocate storage from the non-paged pool



Deploying Kernel-Mode Shellcode

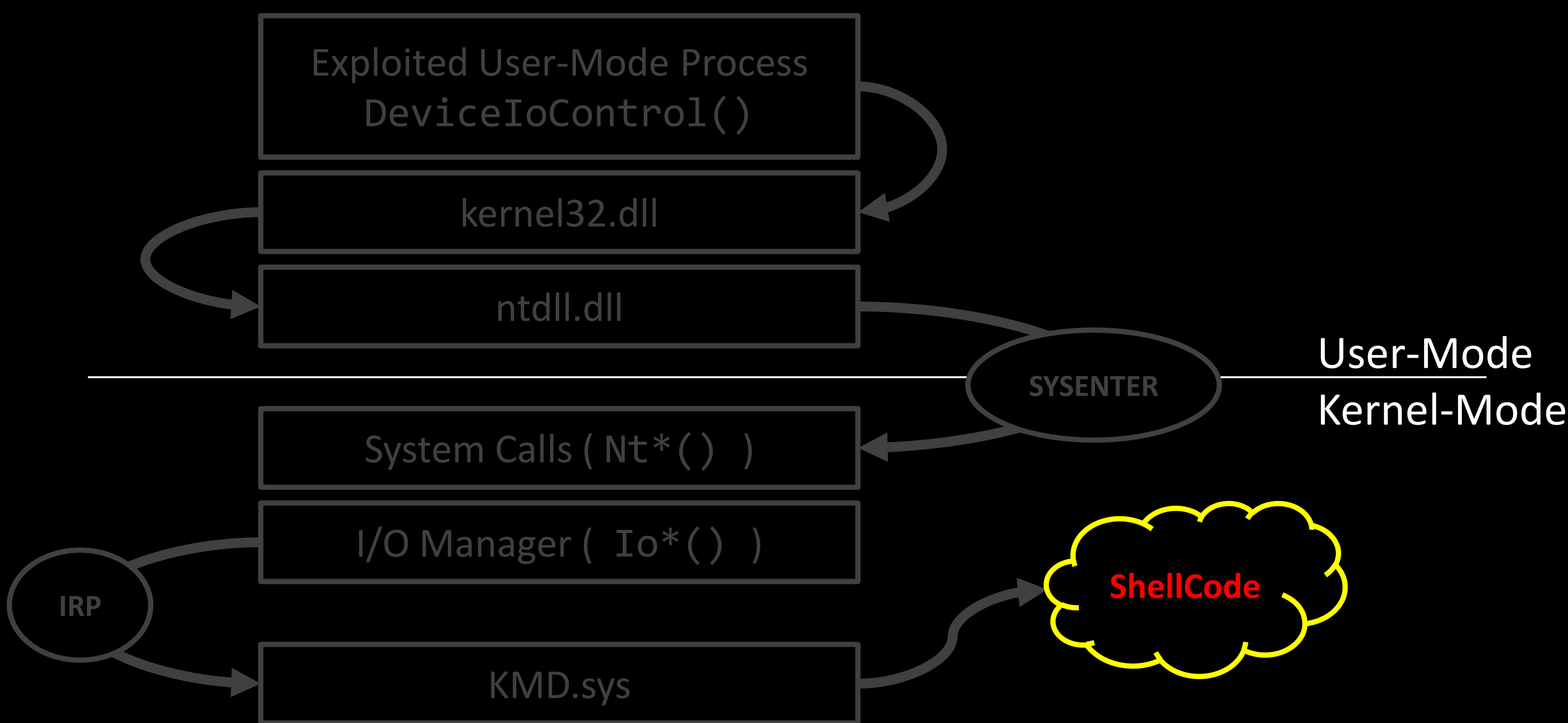
Then, it receives a shellcode payload via a call to **`DeviceIoControl()`**



$$-K_B \sum_i P_i \log_e(P_i)$$

Deploying Kernel-Mode Shellcode

Finally, the KMD unloads, leaving the shellcode alone in memory



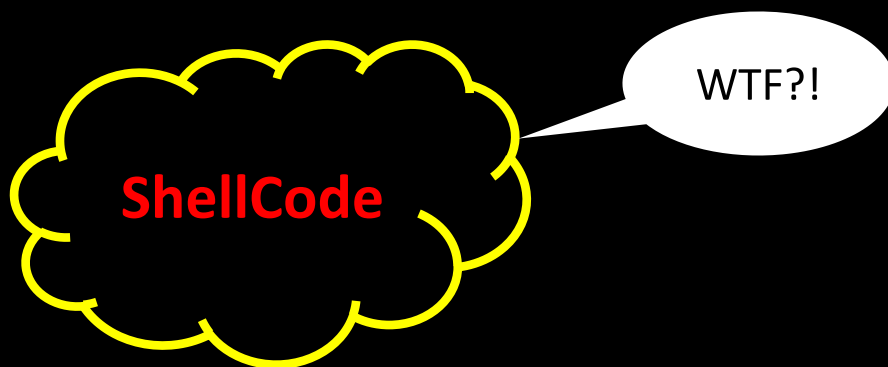
Executing Kernel-Mode Shellcode

So, we have this inert blob of shellcode in memory

Executing Kernel-Mode Shellcode

By itself, it really can't do that much

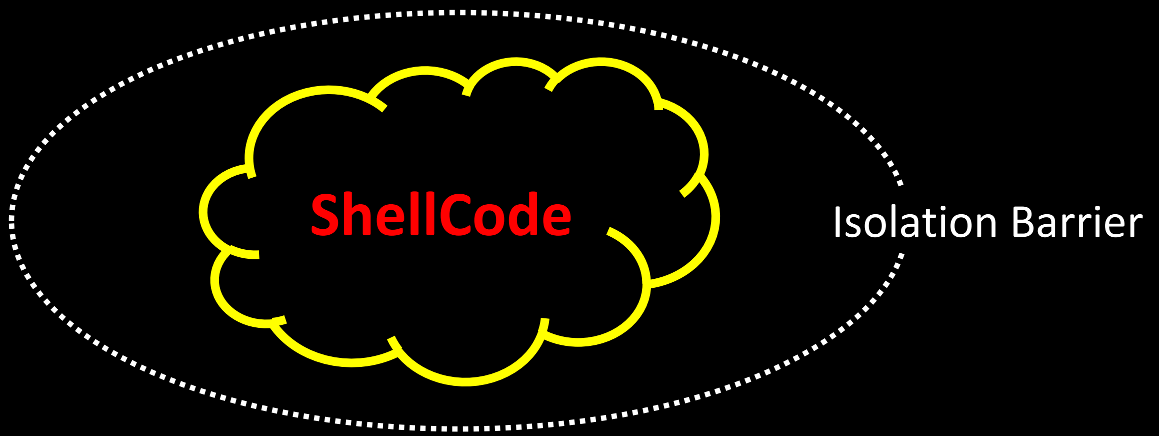
- It’s not a registered driver (e.g. no interface to the I/O Manager)
- It’s not a legitimate thread (e.g. not scheduled by the Windows kernel)



Executing Kernel-Mode Shellcode

It's swimming alone in memory,

With no explicit connection to anything else

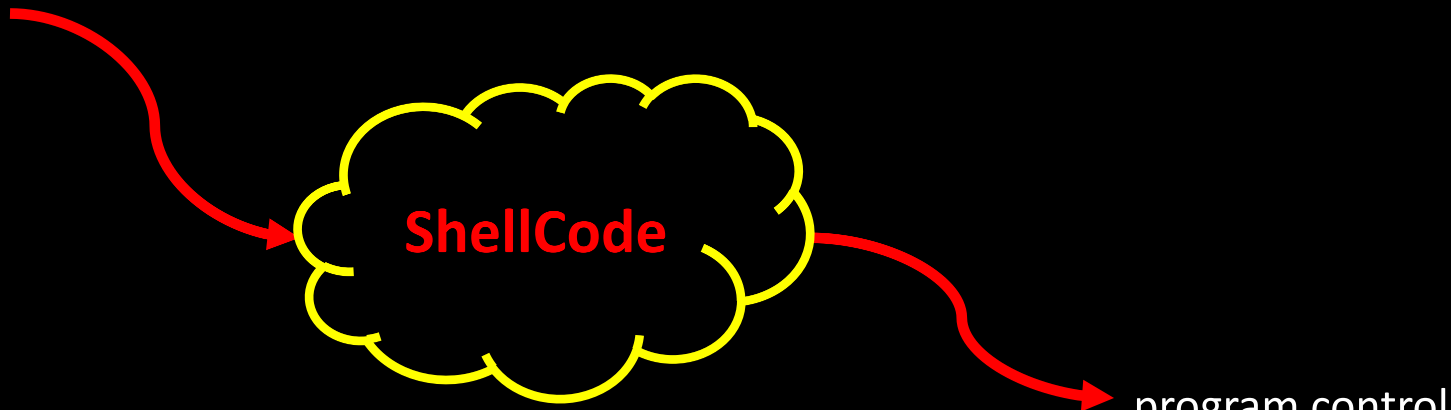


Executing Kernel-Mode Shellcode

Question: How do we get our shellcode to execute?

Executing Kernel-Mode Shellcode

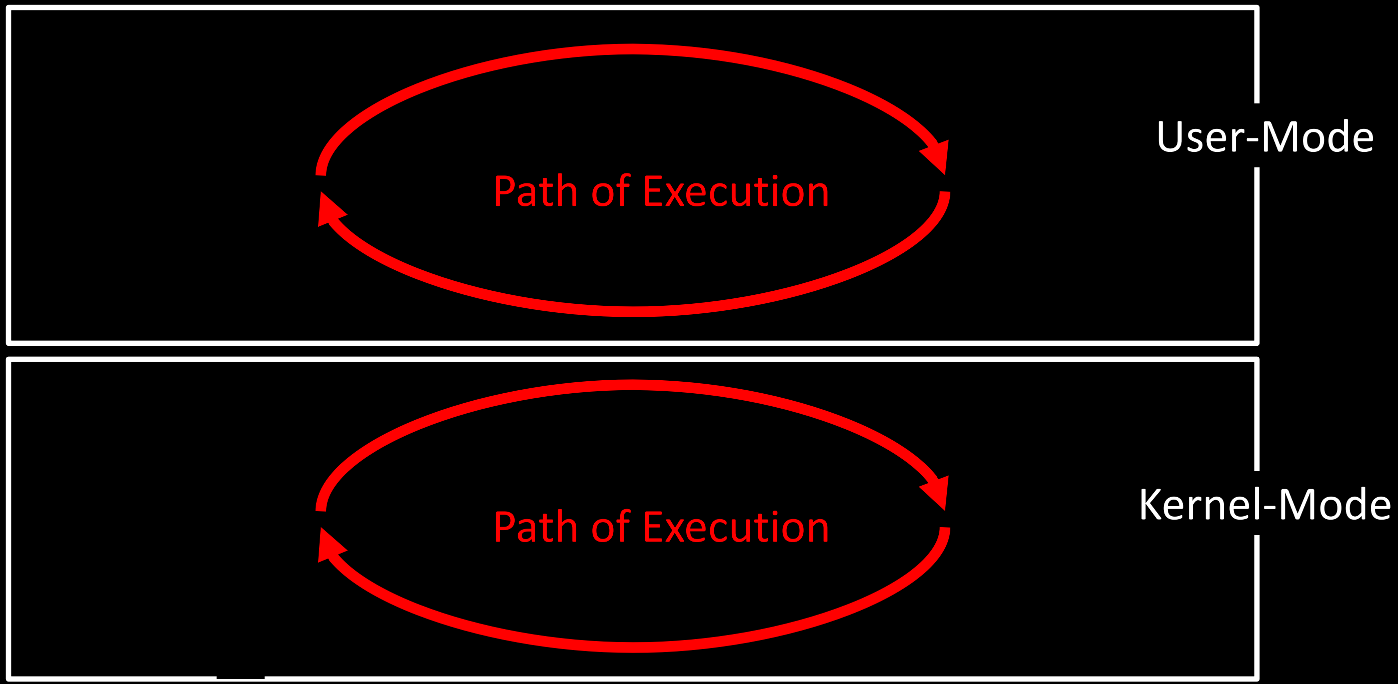
Answer: We need to intercept an existing path of execution



Executing Kernel-Mode Shellcode

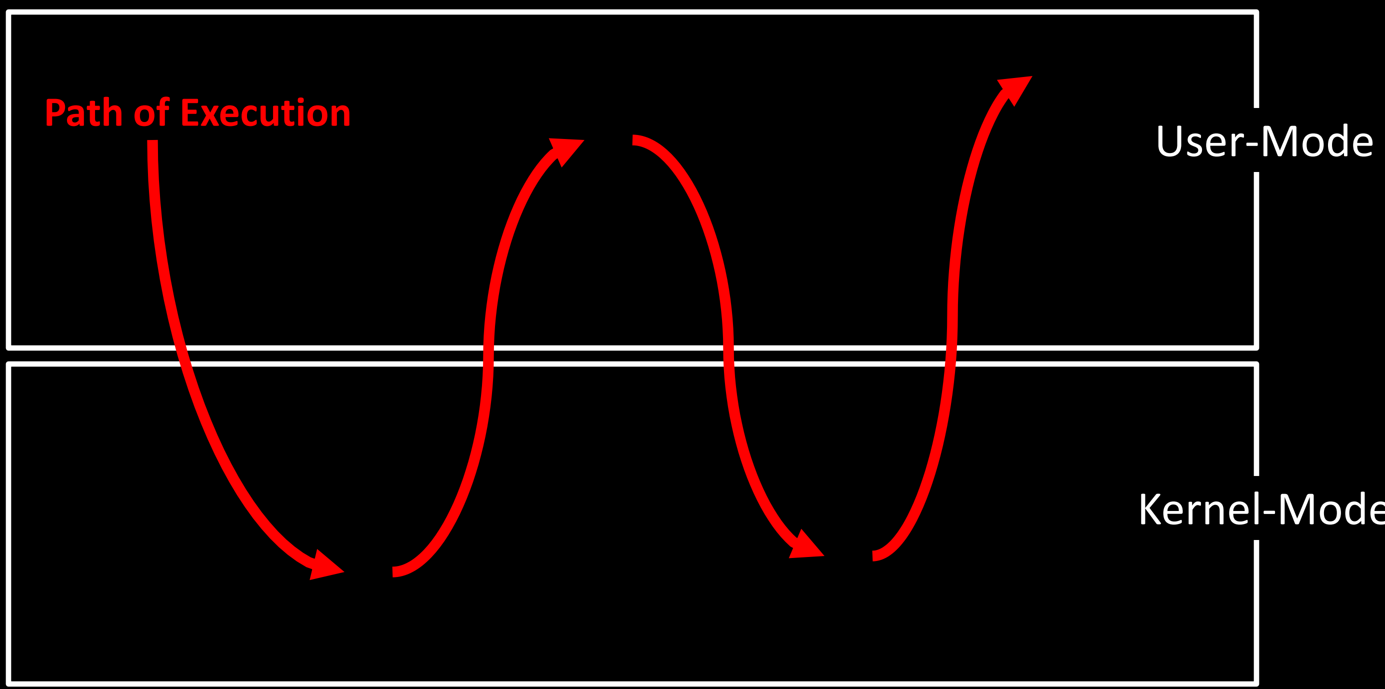
Common misconception:

Application and driver code are confined to their relative address spaces



Executing Kernel-Mode Shellcode

Execution paths are actually able to transition between the two modes



Executing Kernel-Mode Shellcode

There are a variety of different ways to sidetrack the EIP register:

Method of Interception	Level of Stealth
Call Table Hooking	Low: call tables are the epitome of static objects
Detour Patching	Moderate: depending on where and what you patch
Callback Object Modification	High: you're changing naturally dynamic objects

A first cut could implement call table hooking, just to get things to work

As you become more confident, you can adopt more advanced tactics

$$-K_B \sum_i P_i \log_e(P_i)$$

Implementation

Heartbeat Generation

- Alternatives
- Compromises

Heartbeat Generation - Alternatives

We can tunnel data from the targeted machine using different approaches

Tactic	Stealth	Comments
Use the Existing TCP/IP Stack	Low	Connection will be locally visible
Roll Your Own TCP/IP Stack	Moderate	More work, but less conspicuous
Talk Directly to the NIC	High	Hardware dependent

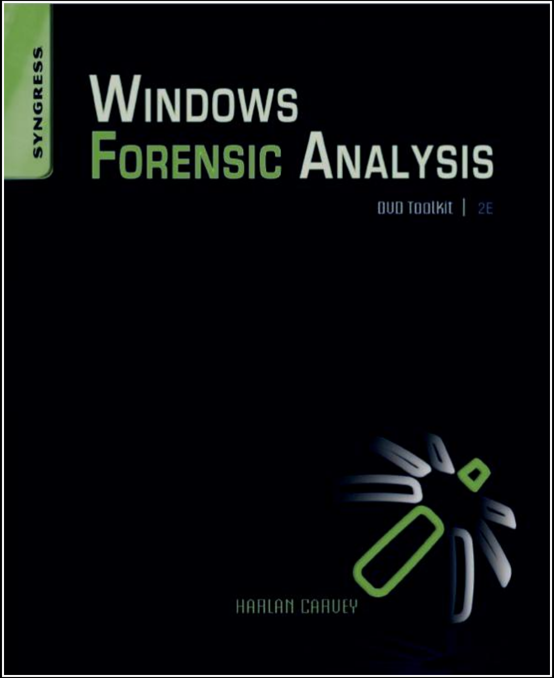
Heartbeat Generation - Alternatives

Sidestepping the native TCP/IP stack offers better (local) concealment

Tactic	Stealth	Comments
Use the Existing TCP/IP Stack	Low	Connection will be locally visible
Roll Your Own TCP/IP Stack	**Moderate**	**More work, but less conspicuous**
Talk Directly to the NIC	**High**	**Hardware dependent**

It also allows an intruder to bypass existing firewall rules

Heartbeat Generation - Alternatives



But, there are problems with this approach:

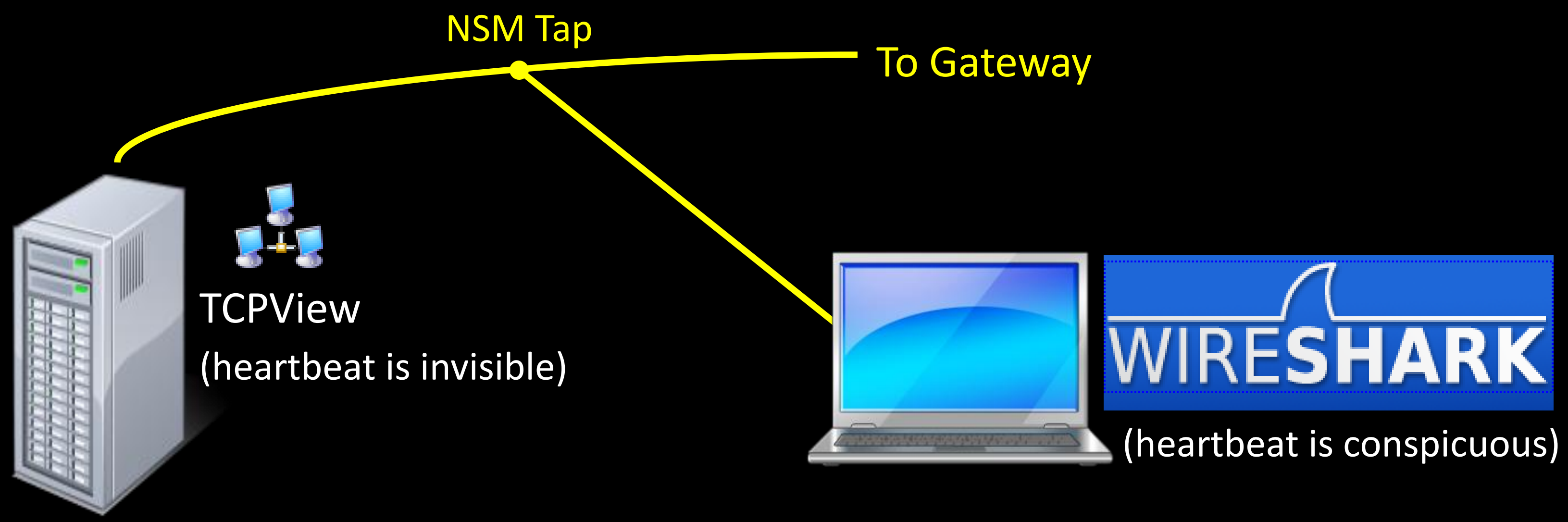
"The absence of an artifact
is in itself an artifact"

–Harlan Carvey, *Windows Forensic Analysis*, p. 372

$$-K_B \sum_i P_i \log_e(P_i)$$

Heartbeat Generation - Alternatives

- NSM may be deployed, and will capture heartbeat traffic
- The absence of a corresponding local connection is a telltale sign…
- Hence, overtly hiding network connections may **not** be a **good idea**



Yet More Engineering Concessions

Again, must find a middle path

On one hand, we wish to:

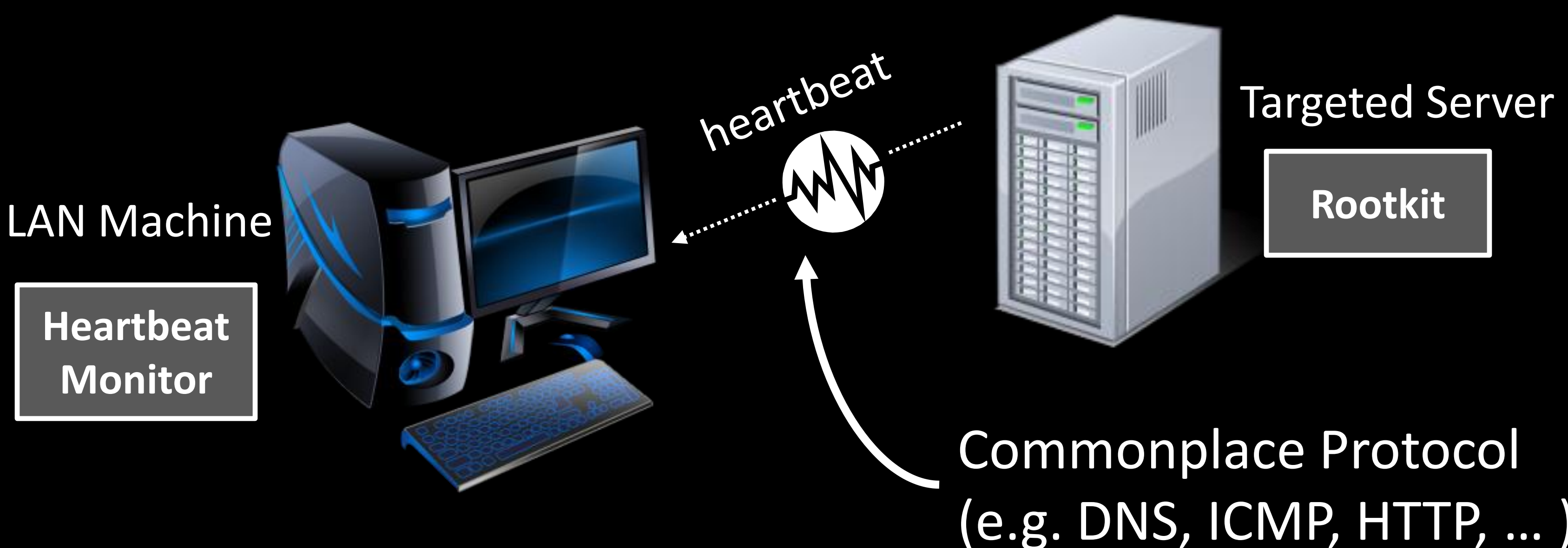
- Be stealthy enough to evade a cursory inspection

At the same time, we'd like to:

- Not be so stealthy that we alert a forensic investigator

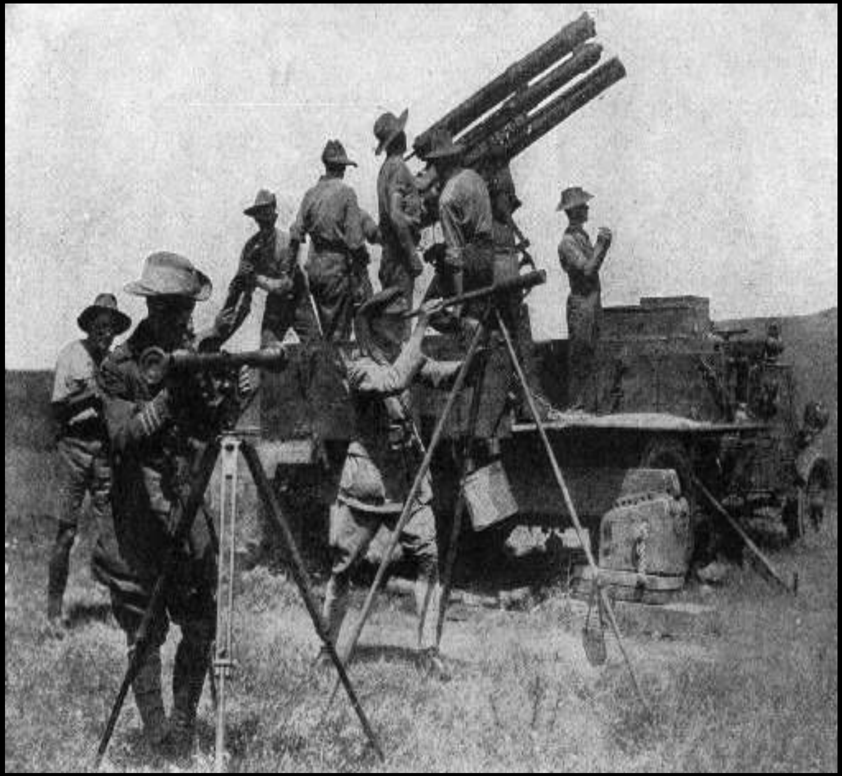
One Solution:

Hide in as large a crowd as possible
Tunnel the heartbeat over a ubiquitous protocol
This isn't perfect, as we'll see, but can be "good enough"
(Joanna Rutkowska jokingly told me this was 1990s tech, and rightfully so*)



*http://www.phrack.org/issues.html?issue=49&id=6

$$-K_B \sum_i P_i \log_e(P_i)$$



Countermeasures

- The Rootkit Paradox
- Detecting Local Modifications
- NSM: The Final Frontier
- Reality Sinks In

The Rootkit Paradox

"All rootkits obey two basic principles:

- They want to remain hidden
- They need to run

…If a deterministic process like the operating system can find the rootkit, then an examiner can find it as well"

–Jesse Kornblum, *International Journal of Digital Evidence*
Fall 2006, Volume 5, Issue 1
http://www.utica.edu/academic/institutes/ecii/publications/articles/EFE2FC4D-0B11-BC08-AD2958256F5E68F1.pdf

The Rootkit Paradox

Corollary:

In addition to acquiring the attention of a processor

Most rootkits **communicate** with the outside

(Otherwise implementing C2 could be problematic…)

$$-K_B \sum_i P_i \log_e(P_i)$$

Nevertheless…

Just because rootkit code executes and communicates
Doesn't necessarily mean it will be *easy* to identify
(It just indicates that detection is *possible*)

It's *possible* to make a lot of money in the stock market

(You just buy low and sell high)

This doesn't mean that it's *easy* in practice

Detecting Local Modification

Recent Solution: HookSafe

- Employs a hypervisor to act as a watchdog
- Monitors some 5,900 kernel hooks in a *Linux* guest OS
- Relocates kernel hooks to a reserved region of memory
- Control access to these kernel hooks using hardware features

http://discovery.csc.ncsu.edu/pubs/ccs09-HookSafe.pdf

LINUX MAGAZINE

HookSafe Protects Kernel from Rootkits

Nov 13, 2009 **A research group in the computer sciences faculty at North Carolina State University has written a prototype to prevent rootkits from manipulating kernel object hooks to do their damage.**

$$-K_B \sum_i P_i \log_e(P_i)$$

Detecting Local Modification

Not all kernel “hooks” are equal

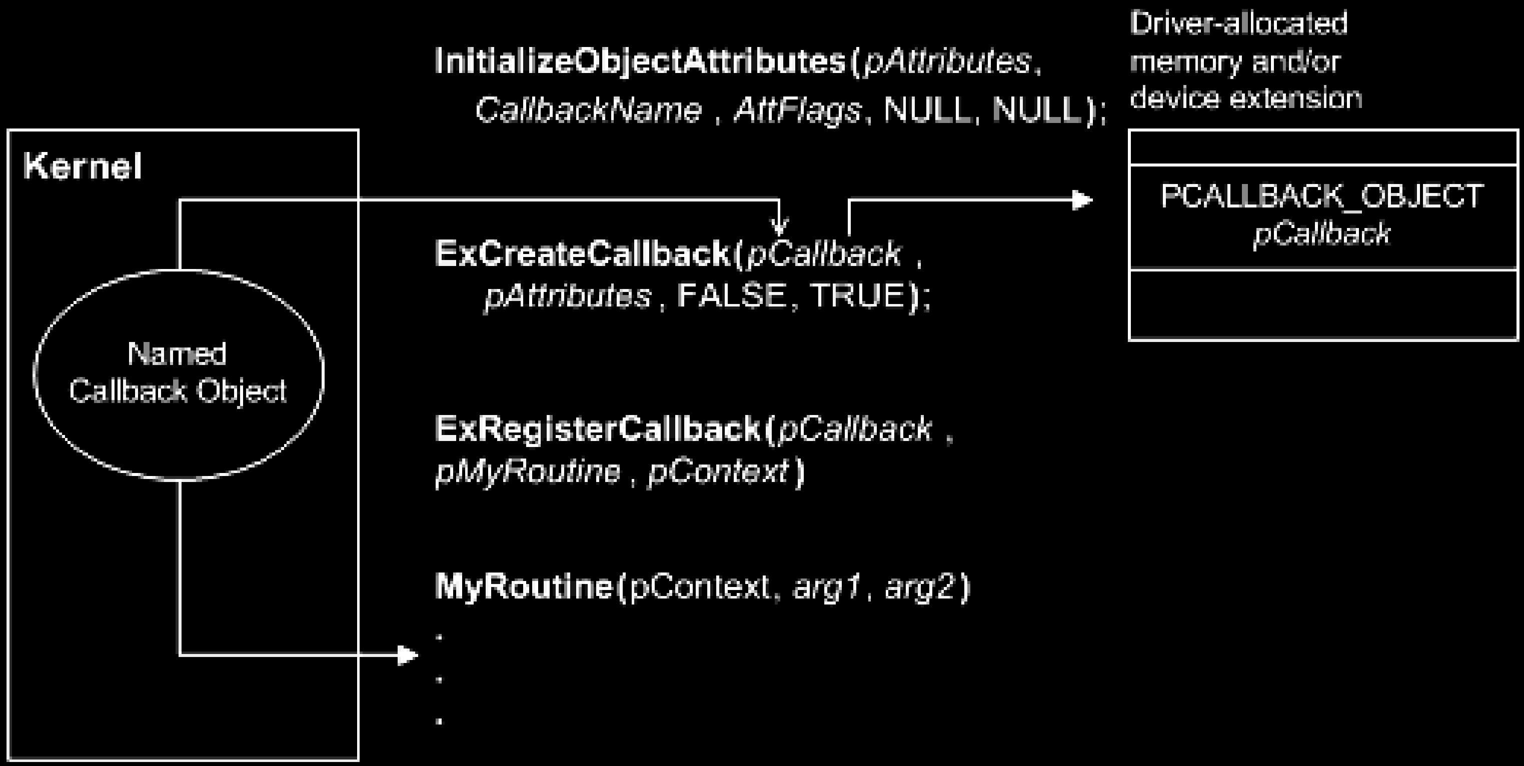
Method of EIP Interception
Call Table Hooks
Detour Patches
Callback Object Modification

Call Tables/Code ≈ Static
(very rarely “write”-accessed)

Callbacks are fluid
(inherently dynamic)

Detecting Local Modification

Callbacks, in particular, are a *nightmare*



$$-K_B \sum_i P_i \log_e(P_i)$$

Detecting Local Modification

```
PVOID ExRegisterCallback
(
    IN PCALLBACK_OBJECT     CallbackObject,
    IN PCALLBACK_FUNCTION   CallbackFunction,
    IN PVOID                CallbackContext
);

VOID ExUnregisterCallback
(
    IN PVOID                CbRegistration
);
```

- There can be an arbitrary number of routines registered with a callback object
- Routines can be registered and unregistered dynamically
- Callbacks are spread over the far reaches of kernel space
- It's not always obvious what constitutes a malicious function pointer

Detecting Local Modification

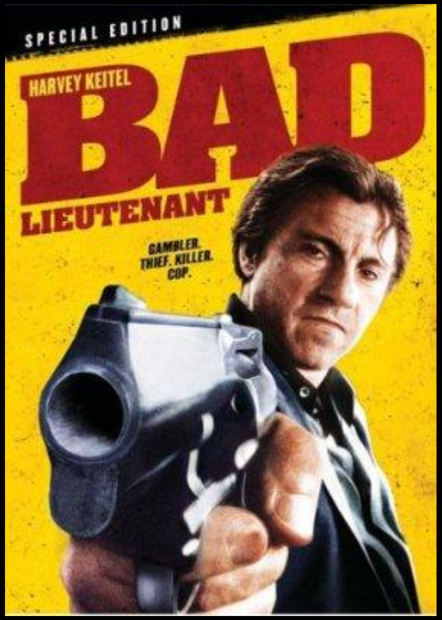
General Lesson:

- Modify system components that are inherently dynamic

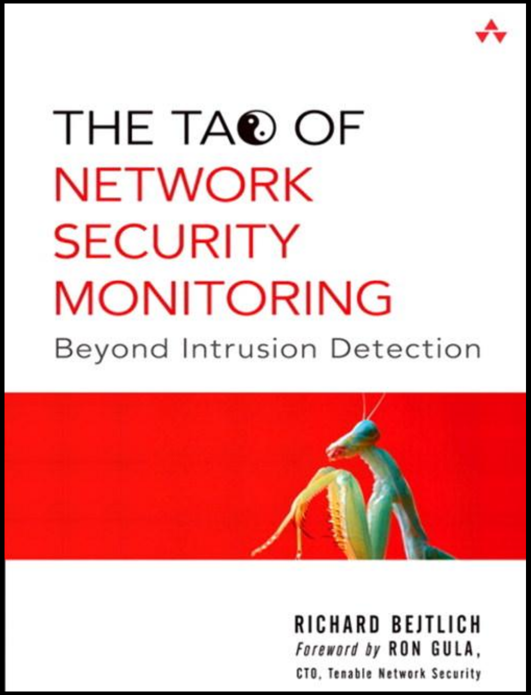
Addendum:

- Watchdog code can be targeted
- Exhibit-A: the arms race to subvert PatchGuard

  http://www.uninformed.org/?v=all&a=38&t=sumry
- Recall what I said about dedicated protected regions…
- This is akin to a police department that goes bad



NSM: The Final Frontier



Rootkits can “interfere” with local data collection

- It’s difficult to obtain an objective POV
- A rootkit can obfuscate or eliminate evidence

But it’s a whole new ballgame on the network

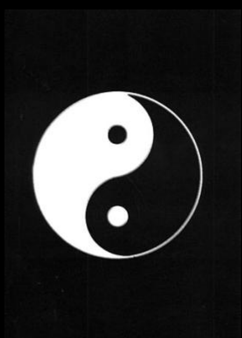
- It’s much harder to conceal data
- Responders can capture and analyze everything
- Sometimes just seeing a connection is enough

$$-K_B \sum_i P_i \log_e(P_i)$$

Reality Sinks In

Fact: IT Divisions operate on a budget

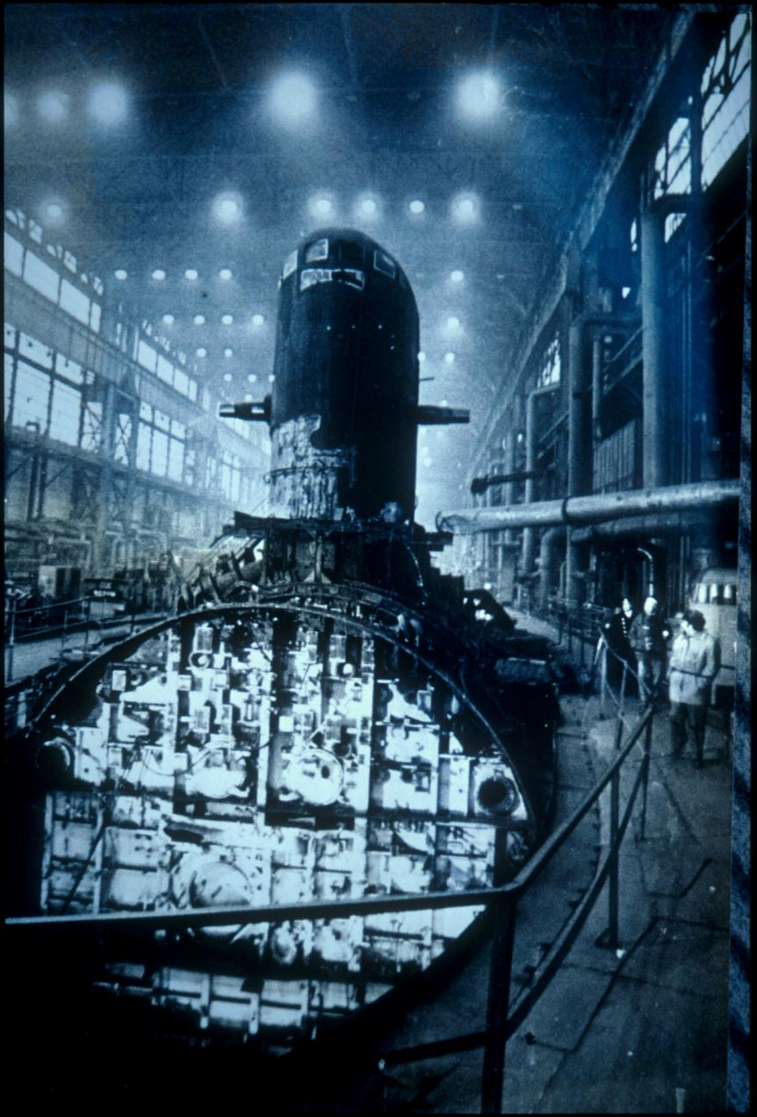
- Overworked responders often don’t have the time to unearth a rootkit
- As a result, imperfect concealment is often sufficient



“I have encountered plenty of roles where I am motivated and technically equipped, but without resources and power.
I think that is the standard situation for incident responders”
–Richard Bejtlich
http://taosecurity.blogspot.com/2008/08/getting-job-done.html

Future Directions

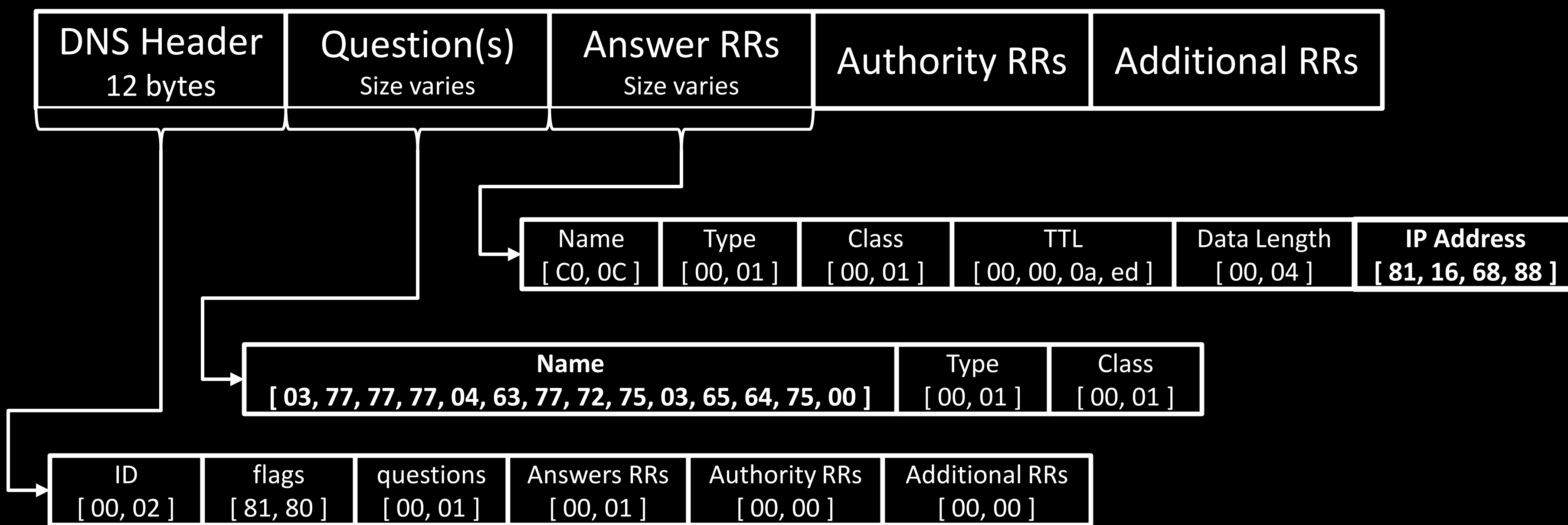
- Heartbeat Mechanism
- Command & Control
- Runtime Deployment



Heartbeat Mechanism

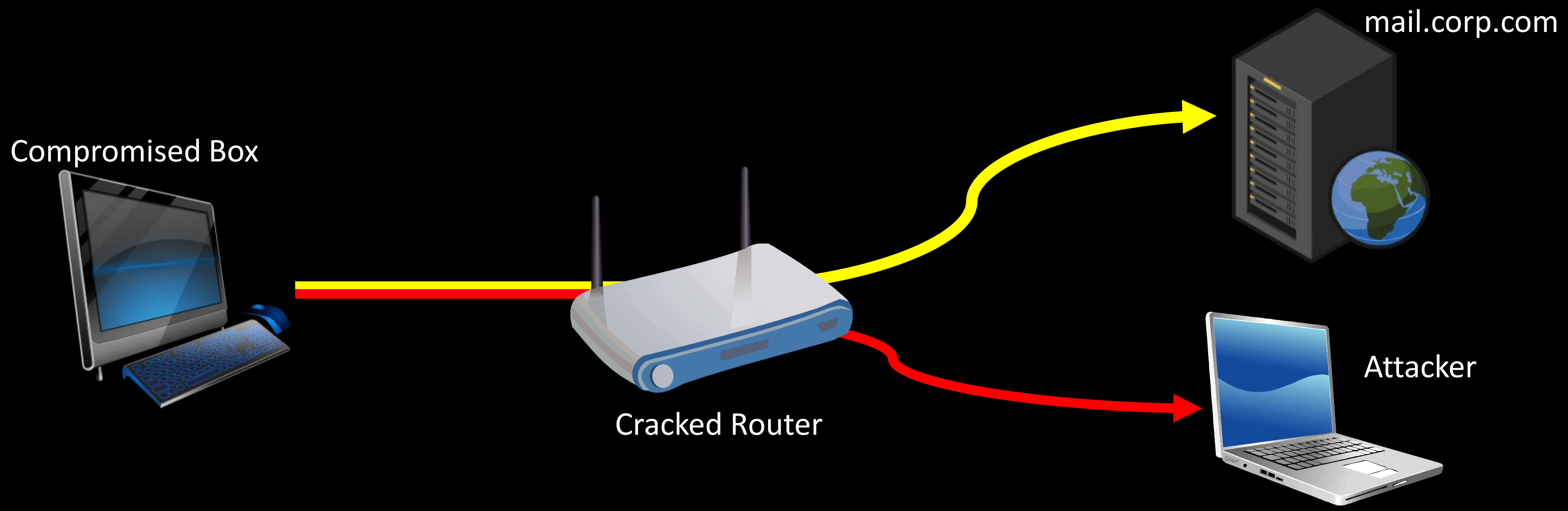
My heartbeat code introduces **new packets** into the network stream

Under careful scrutiny, this could indicate that something is amiss



Heartbeat Mechanism

One alternative is simply to embed data in existing network traffic
In other words, establish a ***Passive Covert Channel*** (PCC)



There's been some publicly available research done in this area

- NUSHU http://www.invisiblethings.org/papers/passive-covert-channels-linux.pdf
- Lathra http://www.cl.cam.ac.uk/~sjm217/papers/ih05coverttcp.pdf

Heartbeat Mechanism

There are a couple of challenges that accompany the PCC strategy

- The necessity to intercept all traffic emitted by the compromised host
  - Could entail cracking a hardened gateway device
  - Involves extra time and resources

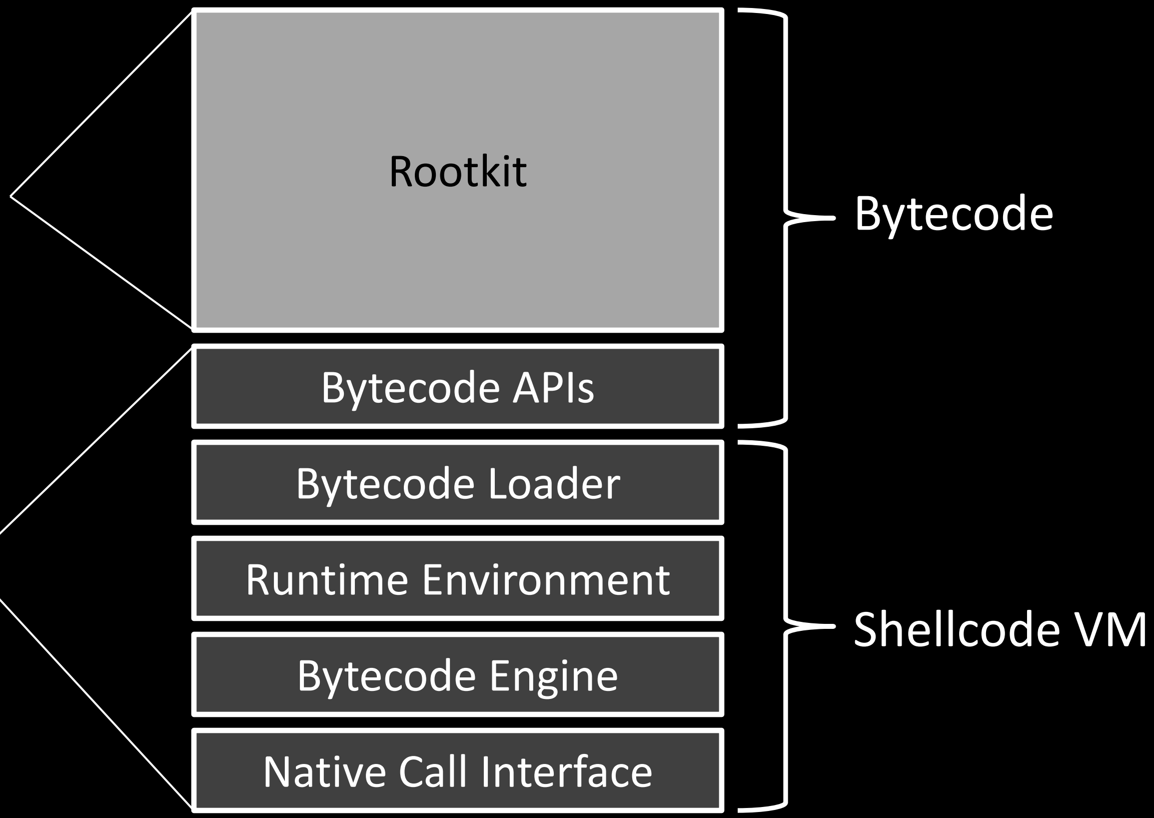
- Data exfiltration can a slow and tedious process
  - Not a good scheme for looting a data warehouse
  - The longer you operate, the greater your risk
  - But, for smuggling out a list of password hashes…

Command & Control (C2)

For a full-featured rootkit deployments, we wish to optimize ROI

Rootkit logic implemented
Using arbitrary bytecode

Virtual machine isolates
The foibles of a given OS



$$-K_B \sum_i P_i \log_e(P_i)$$

Command & Control (C2)

This approach lends itself to **loading bytecode dynamically**

Bytecode Rootkit

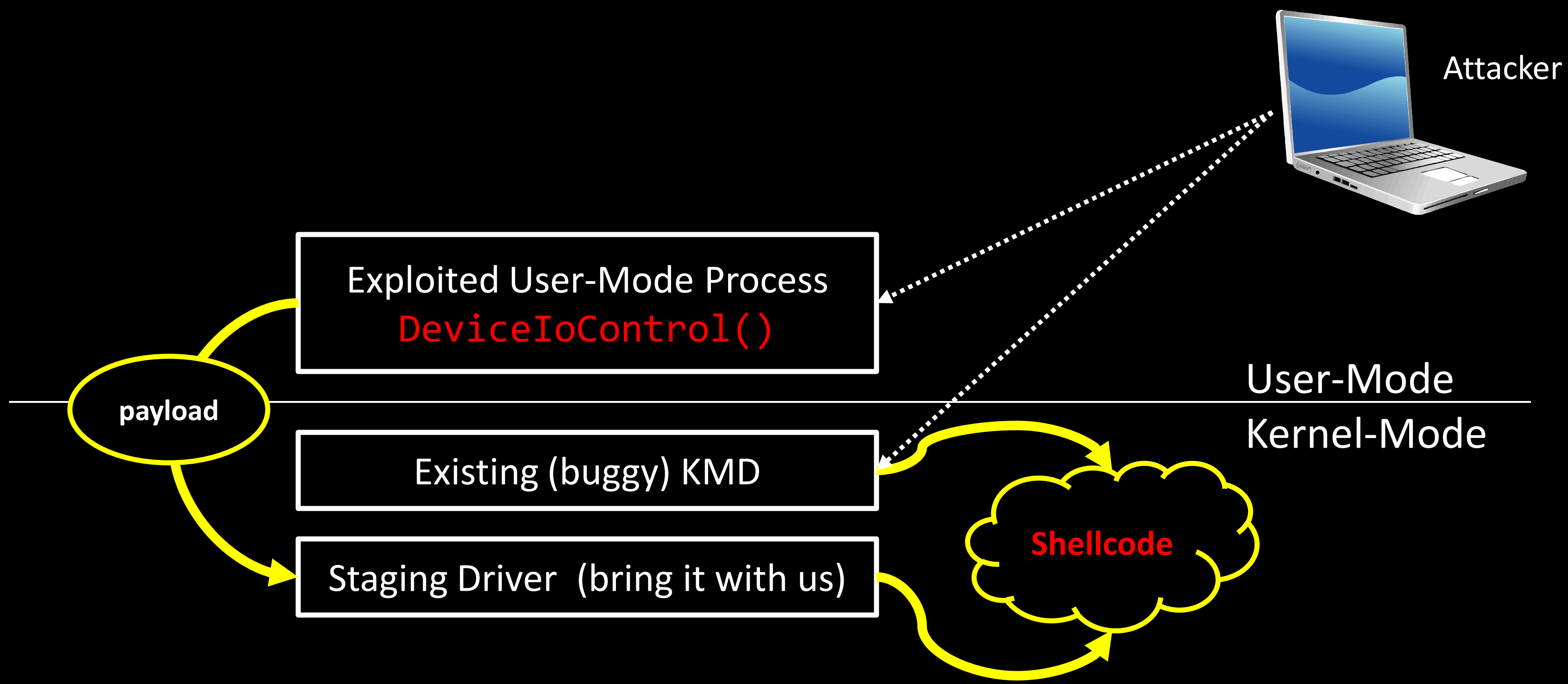
- Instance-Specific Attack
- Packet Sniffer
- Key Logger
- Hash Extractor
- Call Gate Monitor
- Command Interpreter

Virtual Machine

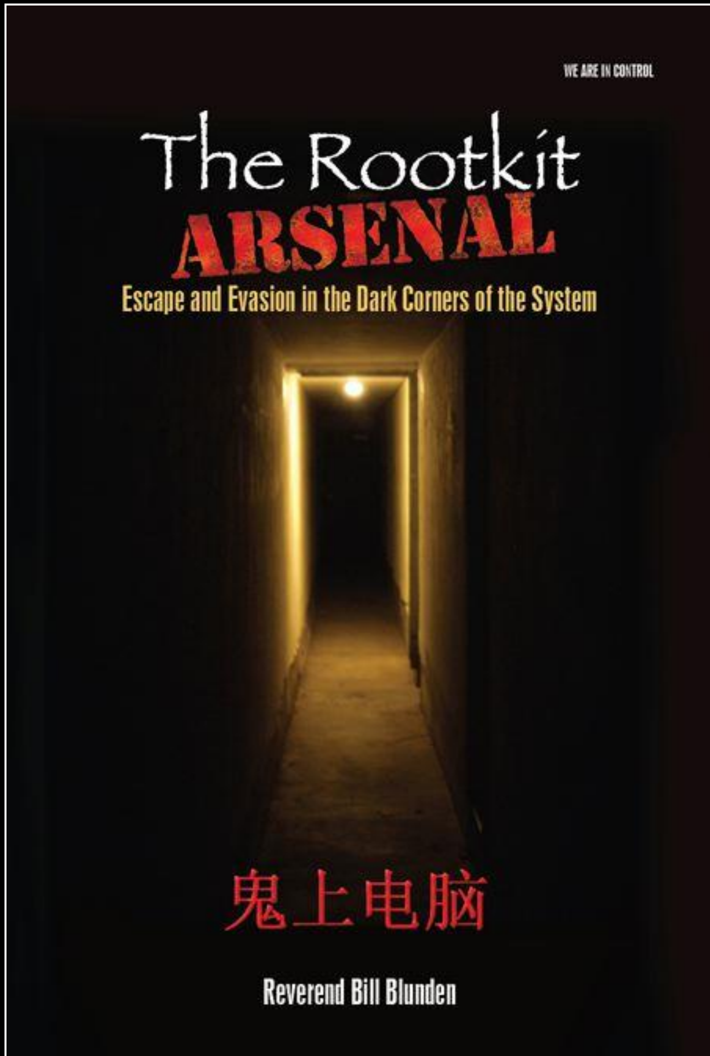
Which, in turn, translates into **runtime extensions**

Runtime Deployment

Thus far, we've loaded the rootkit by means of a user-mode exploit
A more direct alternative would be to leverage a Kernel-Mode Exploit
(Though, this depends heavily on the targeted buggy driver being present)



$$-K_B \sum_i P_i \log_e(P_i)$$



Source Code for this Presentation:
http://www.belowgotham.com/BH-DC-2010.zip

For Additional Information, See:

The Rootkit Arsenal

Jones & Bartlett Publishers
1st edition (May 4, 2009), 908 pages
ISBN-10: 1598220616
ISBN-13: 978-1598220612

Thank You For Your Time

One engineer’s secret
Is another’s implementation detail

$$-K_B \sum_i P_i \log_e(P_i)$$