THE LINUX KERNEL HIDDEN INSIDE WINDOWS 10

BLACKHAT 2016

ALEX IONESCU

@AIONESCU

BIO

Vice President of EDR Strategy at CrowdStrike, a security startup

Previously worked at Apple on iOS Core Platform Team

Co-author of *Windows Internals 5th and 6th Editions*

Reverse engineering NT since 2000 – main kernel developer of ReactOS

Instructor of worldwide Windows Internals classes

Conference speaking:

- Infiltrate 2015
- Blackhat 2016, 2015, 2013, 2008
- SyScan 2015-2012, NoSuchCon 2014-2013, Breakpoint 2012
- Recon 2016-2010, 2006

For more info, see www.alex-ionescu.com

MJF HAS SPOKEN: YOU CAN ALL GO HOME NOW

Microsoft is revealing more details about how Bash on Windows 10 works, and the company's 'Drawbridge' pico-process work figures prominently.

By Mary Jo Foley for All About Microsoft | April 23, 2016 -- 14:28 GMT (07:28 PDT) | Topic: Windows 10

Spoiler alert: There's no secret Linux kernel hidden in Windows 10. Instead, it's the Windows Subsystem for Linux (WSL) that was developed by the Windows Kernel team is what provides the foundation that enabled the Linux binaries to run on Windows.

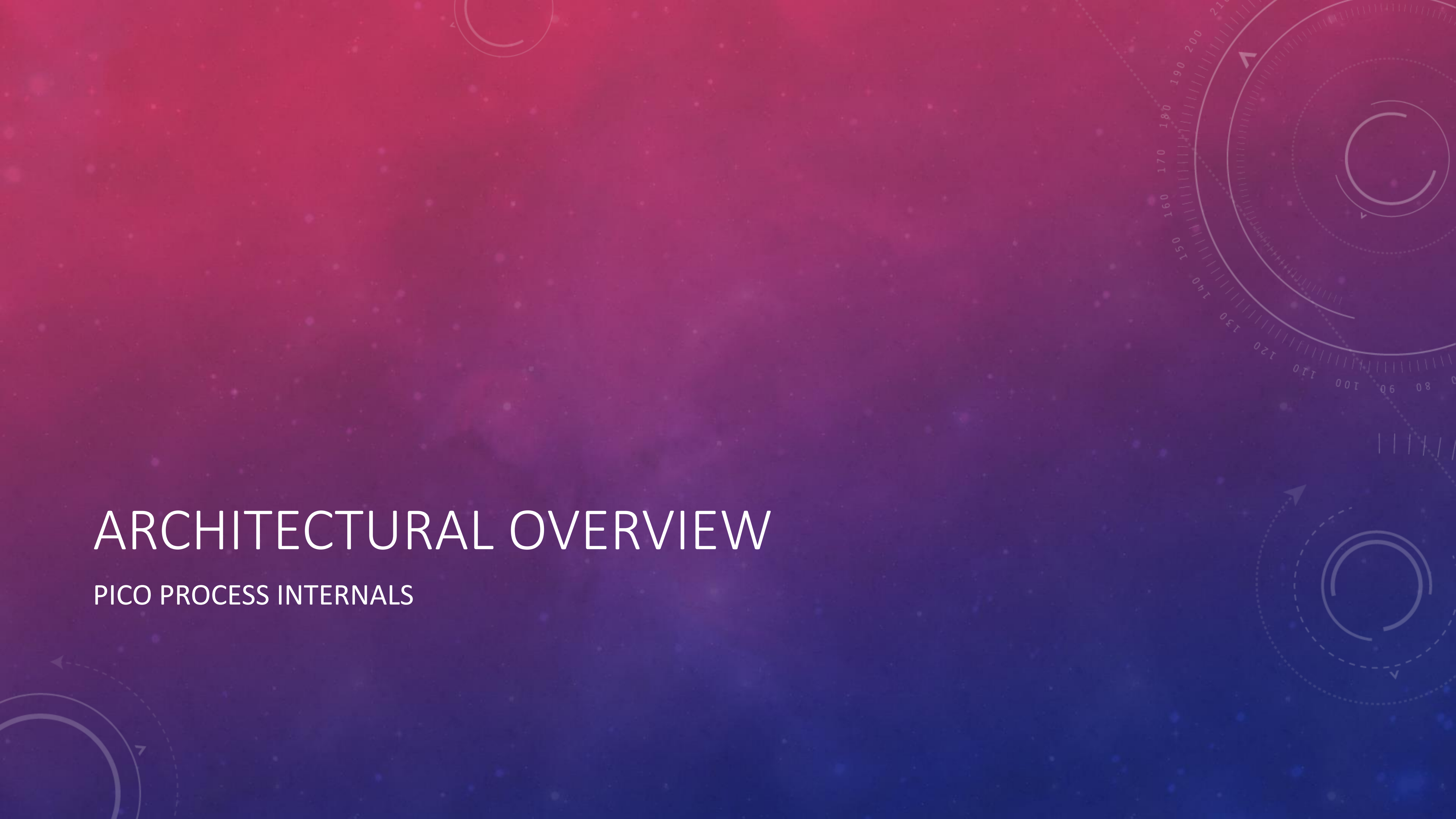
“Spoiler alert”: This is not how the English language works.

OUTLINE

- Architectural Overview
  - Minimal and Pico Processes
  - Pico Provider Interface
  - LXSS Components
    - LxCore Kernel-Mode Driver
      - Functionality, Interfaces & Attack Surface
    - LxssManager Service
      - Functionality, Interfaces & Attack Surface
  - LXSS IPC Interfaces
    - Win32-Linux Communication

OUTLINE

- Security Design Considerations
  - Before & After
  - Kernel Callout & API Behavior for Endpoint Products
  - Forensic Analysis (DbgView / WinDBG) (if time allows)
- Pico Demos (if time allows)
- Conclusion
- Q & A


ARCHITECTURAL OVERVIEW
PICO PROCESS INTERNALS

MINIMAL PROCESS

- Unofficially implemented since Windows 8.1, a minimal process is a bare-bones process with a token, a protection level, a name, and a parent.
  - Empty address space (no PEB, no NTDLL, no KUSER_SHARED_DATA)
  - No handle table
  - EPROCESS.Minimal == 1
- Threads created inside of a minimal process are called minimal threads
  - No TEB
  - No kernel-mode driven user-stack setup
  - ETHREAD.Header.Minimal == 1
- In Redstone 1 these can be created with *NtCreateProcessEx* from kernel-mode with flag 0x800
  - Built-in ones include:
    - “Memory Compression”, managed by the Store Manager
    - “Secure System”, managed by Virtual Secure Machine (VSM)

PICO PROCESS

- A Pico Process is a minimal process which has a Pico Provider (kernel-mode driver)
  - Same state as a minimal process, but EPROCESS.PicoContext is != NULL
- Both Pico and Normal Threads can exist within a Pico Process
  - ETHREAD.PicoContext is != NULL for Pico Threads
- When a minimal thread makes a system call, it's routed to a Pico Provider instead
- When ETW stack traces are being generated for a Pico Process, the Pico Provider is called
- When a user-mode exception is raised within a Pico Process, the Pico Provider is called instead
  - Also when a Probe and Lock / MDL fault is generated on a user-range
- When the name of a Pico Process is being requested, the Pico Provider is called instead
- When a handle is being opened to a Pico Process or Pico Thread, the Pico Provider is called
- When a Pico Thread and/or a Pico Process die, the Pico Provider is notified
  - When a Pico Process is being terminated, the Pico Provider is called instead

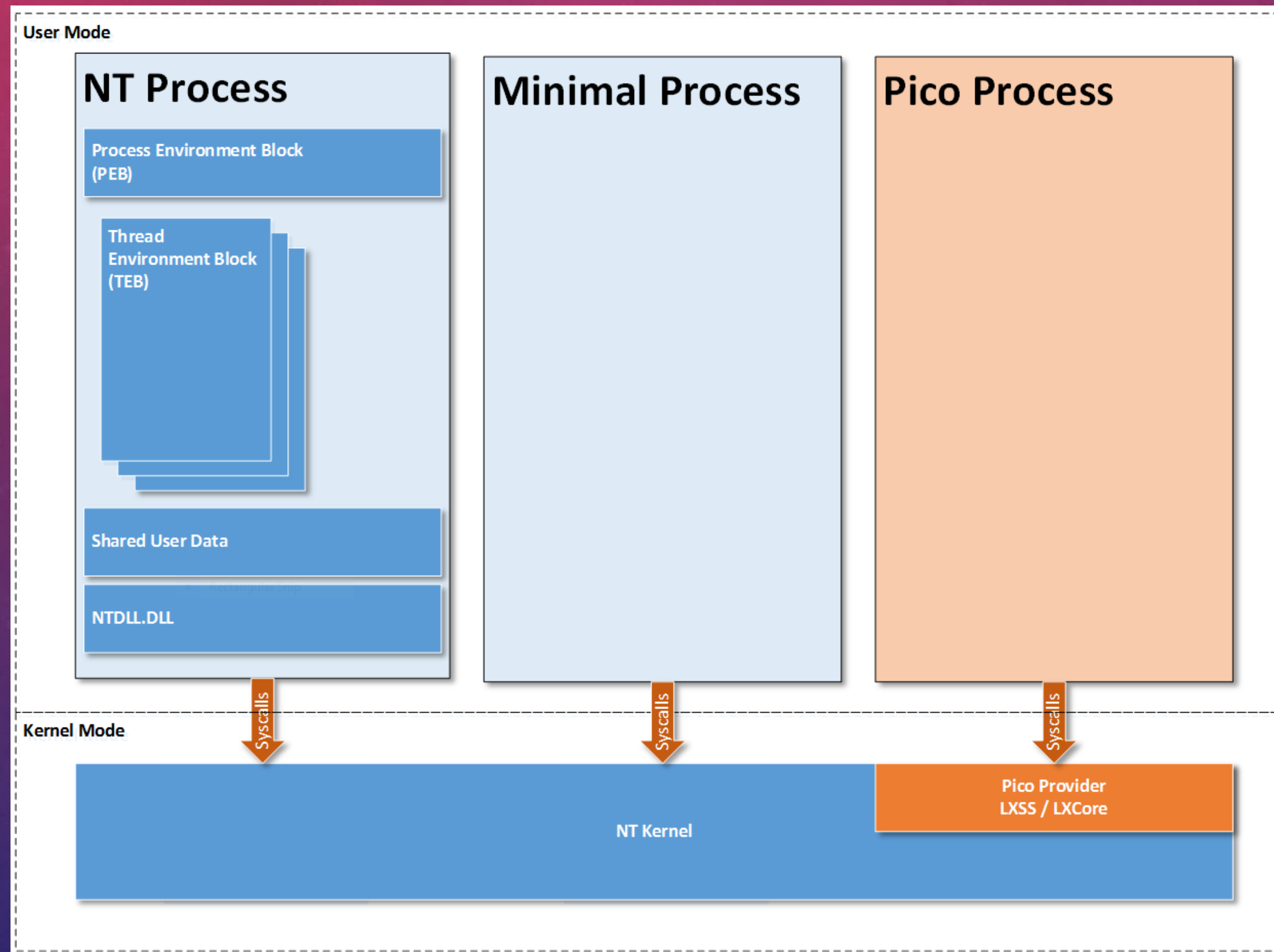
PICO PROVIDERS

- Pico providers are essentially custom written kernel modules which implement the necessary callbacks to respond to the list of possible events (shown earlier) that a Pico process can cause to arise
- NT implementation of the "Drawbridge" Microsoft Research Project
- Can be registered with *PsRegisterPicoProvider*
  - Pass in array of functions which handle the required event callbacks
  - Receive array of functions which allow creation and control of Pico processes (see next slide)
- However, only allowed if *PspPicoRegistrationDisabled* is FALSE
  - Set as soon as *PipInitializeCoreDriversByGroup* returns – before ELAM and 3rd party drivers can load
- Currently only one Pico provider can be registered, as the callbacks are simply a global function array

PICO API

- Create Pico processes and threads: *PspCreatePicoProcess, PspCreatePicoThread*
- Get and set PicoContext pointer: *PspGetPicoProcessContext, PspGetPicoThreadContext*
- Get and set CPU CONTEXT structure: *PspGetContextThreadInternal, PspSetContextThreadInternal*
- Destroy Pico processes and threads: *PspTerminateThreadByPointer, PspTerminatePicoProcess*
- Resume and suspend Pico processes and threads: *PsResumeThread, PsSuspendThread*
- Set user-mode FS and GS segment for Pico threads: *PspSetPicoThreadDescriptorBase*

BASIC BLOCK DIAGRAM



ref: https://blogs.msdn.microsoft.com/wsl/2016/05/23/pico-process-overview/ © Microsoft Corporation 2016

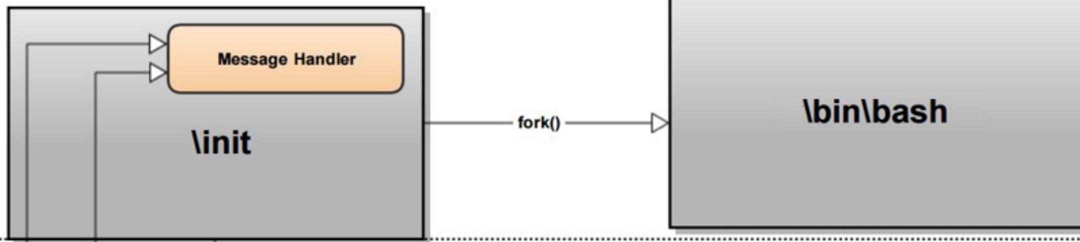
PICO PROVIDER SECURITY

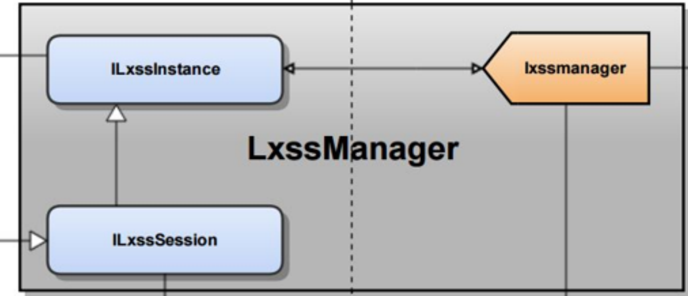
- Pico Providers also “register” with PatchGuard, providing it with their internal list of system call handlers
- Essentially, this means that the Linux system calls are protected by PatchGuard, just like the NT ones
- Additionally, attempting to register a “fake” Pico Provider, or modifying key Pico Provider state will also result in PatchGuard’s wrath
  - For example, playing with *PspPicoProviderRegistrationDisabled*
  - Even if restoring the value back to its original value
- Patching the callbacks (*PspPicoProviderRoutines*) will also result in detection by PatchGuard
- As seen before, only “core” drivers can be Pico Providers
  - Boot Loader enforces that core drivers are signed by Microsoft, and “lxss.sys” is hardcoded inside the code

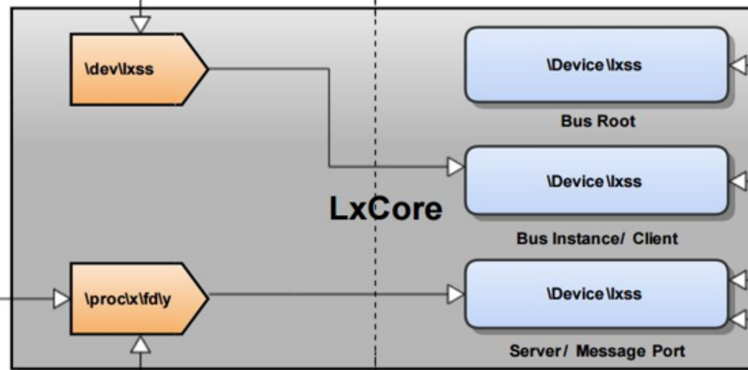
WSL COMPONENT OVERVIEW

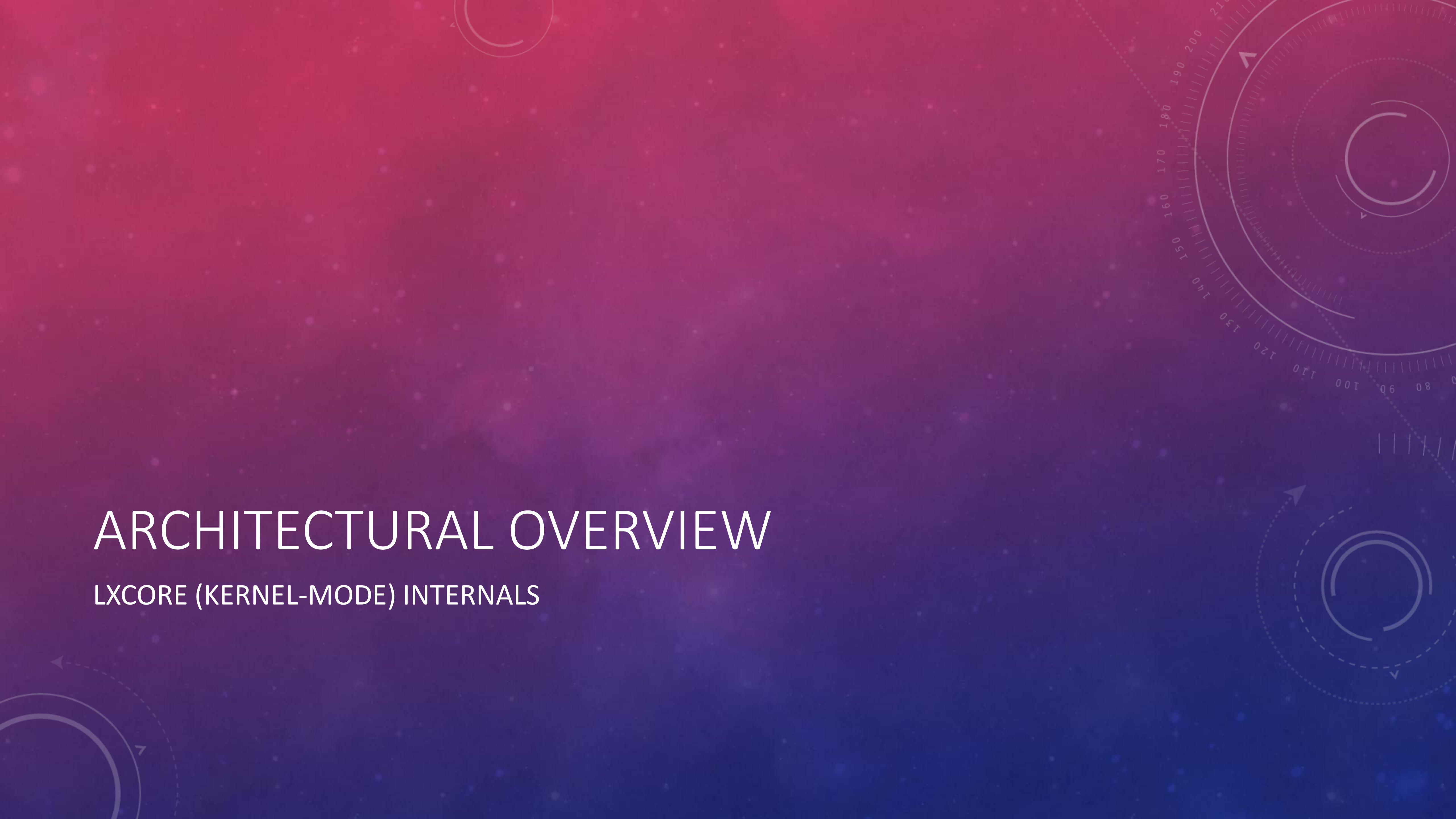
- A Pico Provider driver (LXSS.SYS / LXCORE.SYS) which provides the kernel-mode implementation of a Linux-compatible Kernel ABI and API, as well as a Device Object (\Device\lxss) for command & control.
  - It also provides a "bus" between the various WSL Instances and the NT world, and implements a virtual file system (VFS) interface, including a device inode for command & control on the Linux side.
- A user-mode management service (LxssManager), which provides an external-facing COM interface, relying information to the Pico Provider through its Device Object
- A Linux "init" daemon (like Upstart, systemd, etc…), which is created for every active WSL Instance and is the ancestor of all other Linux processes
- A Windows management process for performing updates/servicing, installation, uninstallation, and user management (LxRun.exe)
- A Windows launcher service which communicates with the LxssManager and spawns a new instance of WSL (and hence init daemon) and executes the desired process (typically "/bin/bash")

User Token
bash.exe
lx::helpers::SvcComm
Win32 World
Pico World
COM


Message Handler
\init
fork()
\bin\bash


SYSTEM Token
(Protected Process Light)
WindowsSide
ILxssInstance
lxssmanager
LxssManager
ILxssSession
Linux Side


open()/ioctl()
LinuxSide
IRP[IRP_MJ_DEVICE_CONTROL]
IRP[IRP_MJ_CREATE]
\dev\lxss
\Device\lxss
Bus Root
\Device\lxss
Bus Instance/ Client
LxCore
MARSHALING :
IRP[IRP_MJ_DEVICE_CONTROL]
(PID, Token, Section, Memory, Pipe, Console)
DATA :
IRP[IRP_MJ_READ / IRP_MJ_WRITE]
MASRHSALING :
ioctl()
PID, Token, Section, Memory, Pipe, Console)
\proc\x\fd\y
\Device\lxss
Server/ Message Port
DATA :
read()/write()
WindowsSide



ARCHITECTURAL OVERVIEW
LXCORE (KERNEL-MODE) INTERNALS

LXCORE FUNCTIONALITY

- LXCORE.SYS is the large (800KB) kernel-mode Ring 0 driver which implements all the functionality that a Linux application inside of a Pico process will see
  - In some cases, functionality is implemented from scratch (such as pipes)
  - In other cases, functionality is wrapped on top of an existing NT kernel mechanism or subsystem (scheduling)
  - In yet other cases, functionality is heavily built on top of an existing NT kernel primitive, with some from-scratch functionality on top (VFS+inodes on top of NTFS for RootFs/DriveFs, but on top of NT Process APIs for ProcFs, for example)
- Decision on how far to go down with wrapping vs. re-implementing was done based on compatibility
  - For example, Linux pipes have subtle enough differences from NT pipes that NPFS could not be used
- In some cases, compatibility isn't perfect – NTFS is not quite identical to EXT+VFS, but "close enough"
- Reliance on many key portability design decisions done by NT team from day 1 (for POSIX support)

LXCORE CUSTOMIZATION

- There are a few registry settings that can be controlled/configured to modify LxCore behavior:
  - PrintSysLevel
  - PrintLogLevel
  - KdBreakPointErrorLevel
  - TraceloggingLevel
  - KdBreakOnSyscallFailures
  - TraceLastSyscall
  - WalkPicoUserStack
  - RootAdssbusAccess
- All present in the HKLM\SYSTEM\CurrentControlSet\Services\lxss\Parameters key

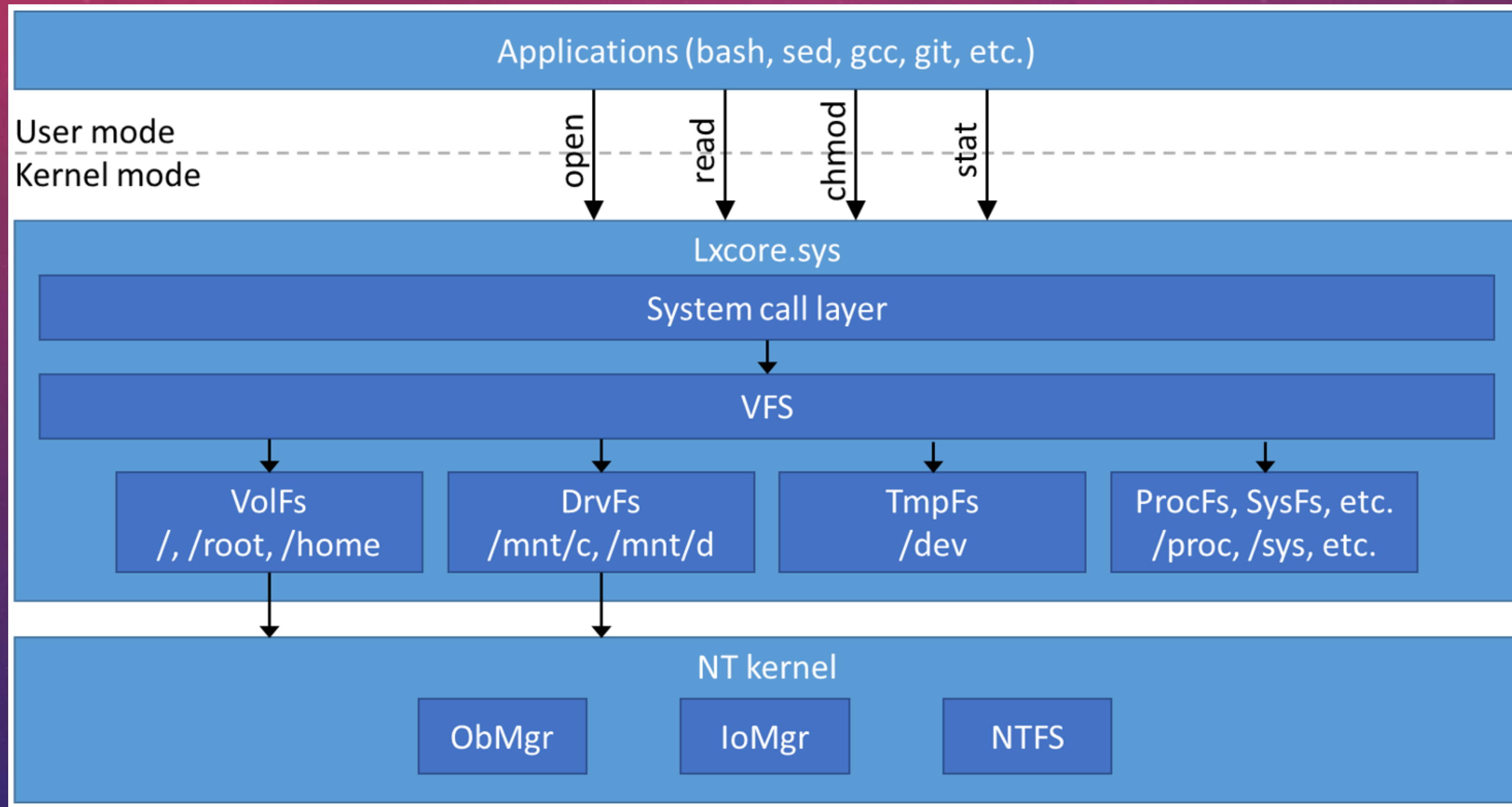
SYSTEM CALLS

- Full "official" list available from Microsoft: https://msdn.microsoft.com/en-us/commandline/wsl/release_notes
- 216 implemented as of the release build
- Called by *LxpSysDispatch*
- Support for ptrace() tracing of system calls
- Can enable breaking on system call failure, for debugging
- *LxpTraceSyscall* used for both ETW logging (LxpPrintSysLevel) and DbgView (LxpTraceLastSyscall)

VFS / FILE SYSTEM

- Implements a “VolFs” system, which is the Linux-facing file system with support for unix rights, etc
  - Implemented on top of Alternate Data Streams and Extended Attributes part of NTFS
  - Not interoperable with Win32 applications
- Also implements a “DrvFs” system, which is the Linux-facing file system that maps Win32 drives
  - Implemented as mount points that map to actual NTFS data on the machine
  - Rights are dependent on the token that was used to launch the initial WSL instance
- Provides emulation/wrapping/actual implementation for various virtual file system concepts:
  - ProcFs mostly works (calls kernel query APIs as needed) or just already has the data
  - TmpFs also exists and implements various devices, including the Binder (key Android IPC from the BeOS days)
  - SysFs is there, etc…

VFS BLOCK DIAGRAM



ref: https://blogs.msdn.microsoft.com/wsl/2016/06/15/wsl-file-system-support/ © Microsoft Corporation 2016

LXCORE INITIALIZATION

- When LxCore initializes *(LxInitialize)*, it first
  - Registers itself as a Pico Provider
  - Then creates an IRP queue for the IPC mechanism,
  - Sets up Event Log tracing,
  - And creates its \Device\lxss Device Object with the appropriate Security Descriptor
- No other work is performed until IRPs are received
  - Create/Close/Cleanup for managing handles to its various entities
  - DeviceControl for sending actual command & control requests
  - Read/Write for the IPC mechanism

DEVICE OBJECT INTERFACES

- Although \Device\lxss is a single Device Object, it implements 5 different interfaces on top of it.
- A Root Bus Interface – used by LxssManager to create new Instances. Only accepts the single IOCTL.
- A Bus Instance Interface – automatically opened by the Root Bus Interface when a new Instance is created. Internally represented as \Device\lxss\{Instance-GUID}. Acts as the command and control interface for the Instance.
- A Bus Client Interface – used by LxssManager whenever an IPC Bus Server is registered or connected to, such as when talking with the init daemon. Represented as \Device\lxss\{Instance-GUID}\Client
- An IPC Server Port Interface – used when an IPC Bus Server is registered. Represented by \Device\lxss\{Instance-GUID}\ServerPort
- An IPC Message Port Interface – used when an IPC Bus Client connects to its Server. Represented by \Device\lxss\{Instance-GUID}\MessagePort

ROOT BUS / INSTANCE CREATION

- The Root Bus implements a single IOCTL:
  - 0x22006F = IOCTL_ADSS_CONTROL_DEVICE_CREATE_INSTANCE
- A handle is obtained by directly opening \Device\lxss with no other parameter or EA
- LxssInstance::_StartInstance called by LxssInstance::StartSelf is normally the only path here
- Creating an instance requires
  - An instance GUID
  - A root directory handle (RootFs)
  - A temporary directory handle (TmpFs)
  - A job object handle for the init process
  - A token handle for the init process
  - Information on which paths to map with DriveFs
  - An instance termination event handle

BUS INSTANCES

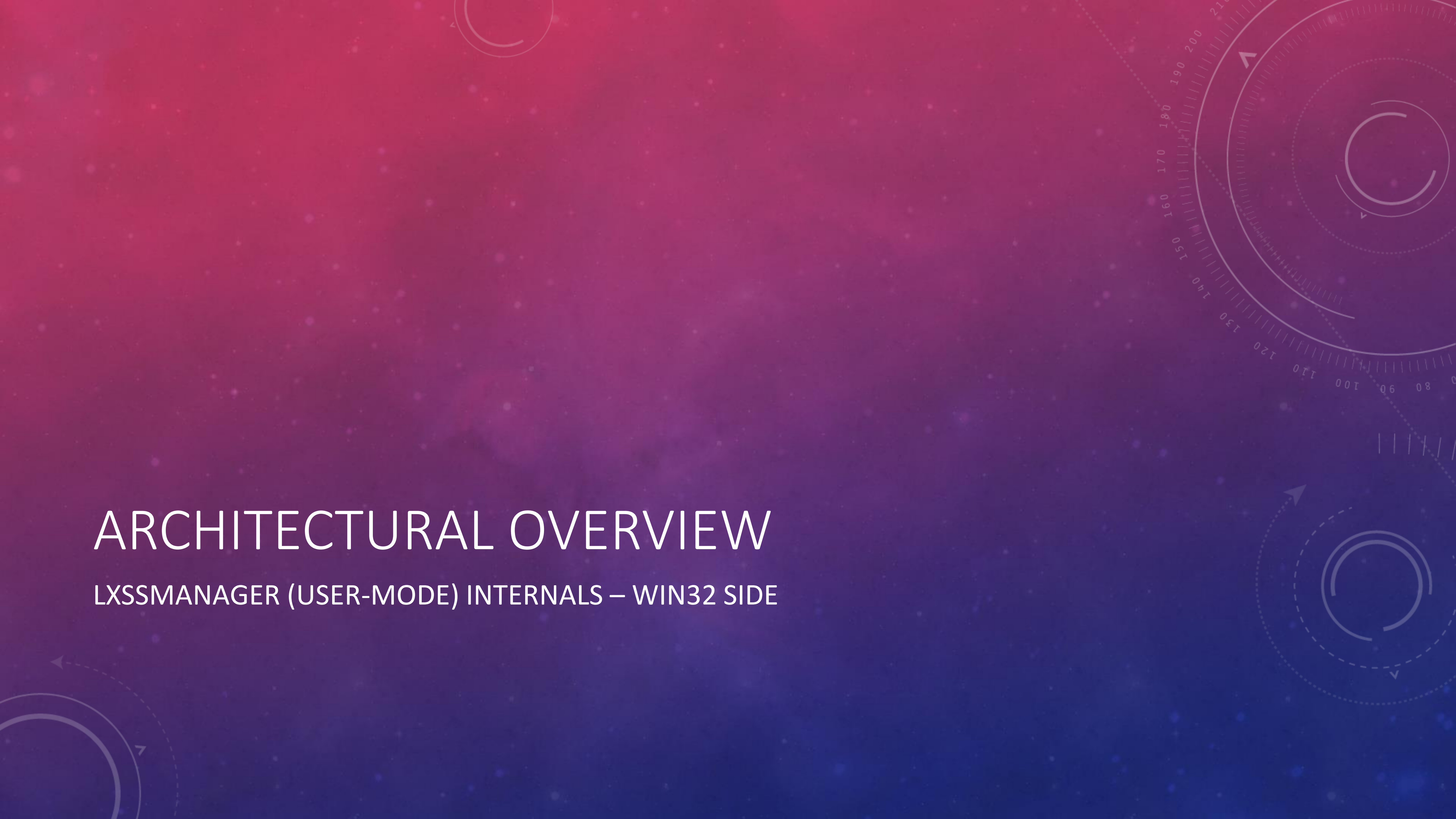
- Once an instance is created, LxCore will automatically open a handle to its corresponding file object under the \Device\lxss namespace, which will be represented by its GUID
  - Cannot arbitrarily attempt to open \Device\lxss\{GUID} from user-mode
- Accepts the following IOCTLs:
  - 0x220073 = IOCTL_ADSS_SET_INSTANCE_STATE
  - 0x22007B = IOCTL_ADSS_BUS_MAP_PATH
  - 0x22007F = IOCTL_ADSS_BUS_MAP_PATH_2
  - 0x220083 = IOCTL_ADSS_UPDATE_NETWORK
  - 0x220087 = IOCTL_ADSS_CLIENT_OPEN
- Starting an instance will cause the creation of the initial thread group & process for the init daemon
- Other IOCTLs used for DriveFs mappings and for updating the NICs exposed to the WSL Instance

BUS CLIENTS

- Automatically opened by the Bus Instance when IOCTL_ADSS_CLIENT_OPEN is used, creating a File Object with the "\Client" suffix under the Instance GUID
- Accepts the following IOCTLs:
  - 0x20002B = IOCTL_ADSS_REGISTER_SERVER
  - 0x22002F = IOCTL_ADSS_CONNECT_SERVER
  - 0x22003F = IOCTL_ADSS_ENLIGHTENED_FORK
  - 0x22004B = IOCTL_ADSS_ENLIGHTENED_FORK_CALLBACK
  - 0x22004F = IOCTL_ADSS_ENLIGHTENED_FORK_CALLBACK_STATUS
  - 0x22008F = IOCTL_ADSS_FILE_SYSTEM_CALLBACK
  - 0x220093 = IOCTL_ADSS_FILE_SYSTEM_CALLBACK_STATUS
- Additionally, exposed to the WSL Instance itself in the Linux world under \dev\lxss
  - Normally only the init daemon (PID == 1) is allowed a handle to this

SERVER /MESSAGE PORTS

- A server port is automatically created as a \ServerPort file object under the Instance GUID when IOCTL_ADSS_REGISTER_SERVER is sent from a Bus Client
- Similarily, a message port (\MessagePort) is created when IOCTL_ADSS_IPC_SERVER_WAIT_FOR_CONNECTION is used (and a connection is established)
- Support for all the marshalling and unmarshalling IOCTLs is implemented here
- Also supports map/unmap memory IOCTLs
- Allows registration and signaling of a port-associated event
- And finally registration/unregistration and create/open of shared memory sections
- More in the IPC section…


ARCHITECTURAL OVERVIEW
LXSSMANAGER (USER-MODE) INTERNALS – WIN32 SIDE

LXSSMANAGER INITIALIZATION

- LxssManager (LXSSMANAGER.DLL) is a service-hosted service living inside of a SVCHOST.EXE process
- Runs as a Protected Process Light (PPL) at the Windows Level (0x51)
  - Will discuss in the security section
- Registers a COM Class (CLSID_LxssUserSession) which implements the IID_ILxssSession interface

LXSSMANAGER INTERFACES

- LxssManager has internal classes for managing the state of WSL in general (Filesystem::, LxssNetworking::/LxssLinuxTimezone::LifetimeManager::)
- It exposes LxssSession:: over COM
  - GetCurrentInstance(), StartDefaultInstance(), SetState(), QueryState(), InitializeFileSystem()
- Once an instance is started or queried, an IID_ILxssInstance interface is exposed (LxssInstance::)
  - GetConfiguration(), GetId(), QueryState(), SetState(), CreateLxProcess(), RegisterAdssBusServer(), ConnectAdssBusServer(), GetState(), StartSelf(), StopSelf(), GetSuspendState()
- Some requests can be done from any arbitrary user (CreateLxProcess) while others require Admin rights (RegisterAdssBusServer)
- Two in-box components communicate with it: LXRUN.EXE and BASH.EXE

LXRUN INTERFACE

- LxRun (LXRUN.EXE) is responsible for User Management, Installation and Uninstallation of WSL, Servicing/Updates.
  - Functionality exposed through command-line interface: /install, /uninstall, /setdefaultuser, /update
- Uses the lx::helpers::SvcComm class to communicate with LxssManager
- Uses lx::helpers::LxUser to provide user management
  - Essentially done by launching the /usr/sbin/usermod, /usr/bin/id, /usr/bin/passwd, /usr/sbin/addgroup, /usr/sbin/deluser, /usr/sbin/adduser binaries in an invisible WSL Instance
- Users lx::run (DoUpdate, DoInstall, HashFile, InstallCleanup, RemoveFolderStructure, AcquireFile)
  - Again, relies on Linux binaries such as /usr/lib/update-notifier/apt-check, /usr/bin/apt, /usr/sbin/update-locale

BASH.EXE INTERFACE

- BASH.EXE is responsible for being the Win32-friendly launcher of WSL Instances
- lx::bash::DoLaunch is used to either launch /bin/bash or another binary, if BASH.EXE –c “valid path” was used
  - NOTE: BASH.EXE –c “man 2 open” will actually launch “/usr/bin/man” and not “/bin/bash –c /usr/bin/man”
- Launch is done by using the same lx::helpers::SvcComm class as LXRUN.EXE
- Reads the DefaultUid in the user’s registry settings to associate the correct Linux user with which to spawn
  - HKCU\Software\Microsoft\Windows\CurrentVersion\Lxss\

ARCHITECTURAL OVERVIEW
INIT (USER-MODE) INTERNALS – LINUX SIDE

INIT DAEMON INITIALIZATION (INITENTRY)

- close() every handle from 0 to 2048
- open() a handle to /dev/kmsg and then dup3() into stderr it for error logging
- open() a handle to /dev/null and then dup3() into stdin and stdout
- open() a handle to /dev/lxss
- Call *ConfigCreateResolvConfSymlinkTarget* to symlink ../run/resolveconv/resolv.conf into /etc/resolv.conf
- Call *InitConnectToServer* and send IOCTL_ADSS_CONNECT_SERVER to connect to the IPC Server “lxssmanager”
- Call *SaveSignalHandlers* and *SetSignalHandlers,* using sigaction to save and set new signal handlers as needed
- Call *ReadMessageFromServer* in a loop, calling reboot() if an invalid message is received

INIT RESPONSIBILITIES

- Through the lxssmanager IPC channel, init will listen for message requests.
- Update Network Information (*InitUpdateNetworkInformation*) -> update /etc/resolv.conf
- Update Timezone Information (*ConfigUpdateTimezoneInformation*) -> update /usr/share/zoneinfo/Msft/localtime and symlink it to /etc/localtime
- Update Hostname Information (*ConfigUpdateHostNameInformation)* -> sethostname(), setdomainname(), update /etc/hostname and create /etc/hosts
- Create process (*CreateProcess*) -> create the environment block, set the command line and working directory, setup the standard handles, then call fork(). Once the new PID exists, setup UID/GID, HOME/PATH environment variables, set the TTY SID, duplicate the standard handles and execvpe() the new process
- Create session leader (*InitCreateSessionLeader*) -> fork() another init and connection to lxssmanager

LXSS IPC INTERFACES
WIN32-LINUX COMMUNICATION

SOCKETS / FILES

- Unix sockets are supported for Linux-Linux application communication, but Internet sockets are also implemented, allowing both local and remote communications over IP
- By creating a localhost server, a Win32 application can connect to it and engage in standard socket API communications
  - Similarly, the server could be on the Win32 side, and the client is in an WSL Instance
- Raw sockets are supported, but require an Admin Token on the NT side to comply with TCPIP.SYS checks
- Using DriveFs, it should be possible for a Windows and Linux app to use read/write/epoll on a file descriptor on the Linux side, and a file object on the Windows side with Read/Write/WaitForSingleObject

BUS IPC ("ADSS" IPC)

- The only other way for a Windows and Linux application to communicate is to use the internal LxCore-provided "Bus" between clients of an instance, part of the original "ADSS" interface (Android Sub System)
- Allows a named server to be registered and connected to, after which both sides can:
  - Use Read/Write API calls for raw data
  - Use IOCTL API calls for marshal/unmarshal operations
- The Windows side needs to register a server by either sending the correct IOCTL (to the Bus Instance) or by using the ILxssSession COM Interface and calling the RegisterAdssBusServer method (Administrator only)
  - Then, can use the IOCTL_ADSS_IPC_SERVER_WAIT_FOR_CONNECTION on the returned Server Port handle
- The Linux side needs to open a handle to the Bus Instance (\dev\lxss) as root (UID 0) and then send the IOCTL_ADSS_CONNECT_SERVER IOCTL
  - Restricted to the init daemon only (PID must be 1)
  - Or, set "RootAdssbusAccess" in HKLM\CCS\Services\lxss\Parameters key to 1

BUS IPC MARSHALLING

- With a connection setup, the following types of metadata can be marshalled
  - PIDs from Linux to Win32. Used during process creation by the init process to provide the fork()'ed PID
  - Console handles from Win32 to Linux. Used during process creation to setup the TTY/SID after unmarshaling, to correspond to the Win32 console handles
  - Pipe handles from Win32 to Linux. Can be used during process creation to associated the in/out/error handles of the Linux application with a Win32 pipe handle. Allows "piping" from Linux to Win32, but sadly not exposed.
  - Tokens from Win32 to Linux. Overrides the NT token stored in the LXPROCESS structure used for the next fork() called. Used during process creation, but provides interesting capabilities.
- IOCTLs:
  - 0x220097/9B = IOCTL_ADSS_IPC_CONNECTION_MARSHAL/UNMARSHAL_PID
  - 0x22009F/A3 = IOCTL_ADSS_IPC_CONNECTION_MARSHAL/UNMARSHAL_HANDLE
  - 0x2200A7/AB = IOCTL_ADSS_IPC_CONNECTION_MARSHAL/UNMARSHAL_CONSOLE
  - 0x2200AF/B3 = IOCTL_ADSS_IPC_CONNECTION_MARSHAL/UNMARSHAL_FORK_TOKEN

BUS IPC DATA EXCHANGE

- A Linux application can share part of its virtual address space with an NT application, which can then map it
- Both sides can register/unregister and create/open shared memory sections, which appear as file descriptors on Linux and Section Object handles on Win32
- A Win32 application can register an Event Object handle and associate it with the message port, and then call WaitForSingleObject on it – the Linux side can signal it with an IOCTL
- A Linux application can use epoll() to wait on its message port file descriptor, and a Win32 application can signal it with an IOCTL
- IOCTLs:
  - 0x220037/3B = IOCTL_ADSS_IPC_CONNECTION_SHARE/UNSHARE_MEMORY
  - 0x220043/47 = IOCTL_ADSS_IPC_CONNECTION_MAP/UNMAP_MEMORY
  - 0x220053/57 = IOCTL_ADSS_IPC_CONNECTION_CREATE/OPEN_SHARED_SECTION
  - 0x22005B/5F = IOCTL_ADSS_IPC_CONNECTION_REGISTER/UNREGISTER_SHARED_SECTION
  - 0x220063/67 = IOCTL_ADSS_IPC_CONNECTION_REGISTER/SET_SIGNAL_EVENT

SECURITY DESIGN CONSIDERATIONS
BEFORE & AFTER

INITIAL ANALYSIS

- Pico Processes were originally the cornerstone behind “Project Astoria”
  - Full Android Runtime on Windows 10 Mobile / Phone, implemented as a “true” NT Subsystem
  - Only on Phone SKU, and was killed in 1511 Update RTM
- In Windows 10 Anniversary Update Previews, Pico functions were once again functional
  - Adss.sys was replaced by lxss.sys
- No more “Project Astoria”: Pico processes are now used for the re-re-reimplementation of the original POSIX subsystem from Windows NT
  - New Name: Windows Subsystem for Linux (WSL)
- Instead of running Android, the user-space environment is Ubuntu 14
- Significant improvements to run desktop-class Linux applications were made
  - Other Android-specific features removed, such as /dev/fb or /dev/adb

DESIGN ISSUES IN PREVIEW BUILDS

- WSL processes were initially invisible in Task Manager
  - Still visible in other tools
- Documented kernel API did not provide notification for Pico processes or threads
  - Invisible from endpoint security products/AV
- WSL processes and libraries (.so) are not loaded as SEC_IMAGE, so no image load notifications
  - Invisible from endpoint security products/AV
  - Completely bypasses AppLocker rules
- WSL file access and network I/O is kernel-sourced
  - Does result in minifilter and WFP callbacks, but might be “trusted” due to kernel being the caller
- *SeLocateProcessImageName* returns NULL for Pico processes
  - Cannot create firewall rules or get the name in WFP callout driver
- “Developer Mode” and “Feature Installation” were required for WSL – but driver could be communicated with from

DESIGN ISSUES IN PREVIEW BUILDS

- LxCore is installed **by default** in the kernel even if the feature is disabled and developer mode is turned off
- Checks were done only by LxssManager over the COM interface, but not the driver itself
  - Driver allowed Administrators to have RW Access
- As such, could completely bypass LxssManager/Developer Mode/Feature Installation and directly send commands to the driver (from Admin)
- Tweeted “PicoMe” in February before WSL was even announced

```
LXCORE!LxElfValidateHeader64:
fffff80b`05ae466c 48895c2408      mov     qword ptr [rsp+8],rbx ss:ffffc200`07d34190=0000000000000170
fffff80b`05ae4671 55              push    rbp
fffff80b`05ae4672 56              push    rsi
fffff80b`05ae4673 57              push    rdi
fffff80b`05ae4674 4154            push    r12
fffff80b`05ae4676 4155            push    r13
fffff80b`05ae4678 4156            push    r14
fffff80b`05ae467a 4157            push    r15
fffff80b`05ae467c 488d6c24e1      lea     rbp,[rsp-1Fh]
fffff80b`05ae4681 4881ecb0000000  sub     rsp,0B0h
fffff80b`05ae4688 488b05a9a9feff  mov     rax,qword ptr [LXCORE!__security_cookie (fffff80b`05acf038)]
fffff80b`05ae468f 4833c4          xor     rax,rsp
fffff80b`05ae4692 4889450f        mov     qword ptr [rbp+0Fh],rax
fffff80b`05ae4696 488b457f        mov     rax,qword ptr [rbp+7Fh]
fffff80b`05ae469a 4c8d55cf        lea     r10,[rbp-31h]
fffff80b`05ae469e 4c8bd9          mov     r11,rcx
fffff80b`05ae46a1 488945af        mov     qword ptr [rbp-51h],rax
```

Command - Kernel 'usb2:targetname=sammy' - WinDbg:10.0.11063.818 AMD64

```
2: kd> k
 # Child-SP          RetAddr           Call Site
00 ffffc200`07d34188 fffff80b`05b11057 LXCORE!LxElfValidateHeader64
01 ffffc200`07d34190 fffff80b`05b0484b LXCORE!LxpMmMapImageContextInitialize+0x57
02 ffffc200`07d34210 fffff80b`05b04a89 LXCORE!LxpThreadGroupMapImageContextInitialize+0x10f
03 ffffc200`07d342f0 fffff80b`05b0419f LXCORE!LxpThreadGroupParametersPopulate+0x2d
04 ffffc200`07d34360 fffff80b`05b04044 LXCORE!LxpThreadGroupLaunchExecute+0x5f
05 ffffc200`07d344f0 fffff80b`05aed03c LXCORE!LxpThreadGroupLaunch+0x130
06 ffffc200`07d34540 fffff80b`05aeda06 LXCORE!LxpInstanceLaunchInitProcess+0x164
07 ffffc200`07d34670 fffff80b`05aed87f LXCORE!LxpInstanceStart+0x112
08 ffffc200`07d346b0 fffff80b`05b330d0 LXCORE!LxpInstanceSetState+0xdb
09 ffffc200`07d34700 fffff80b`05b32a40 LXCORE!AdssBusSetInstanceStateIoctl+0x54
0a ffffc200`07d34740 fffff80b`05ae3194 LXCORE!AdssBusIoctl+0xac
0b ffffc200`07d34790 fffff80b`05ae37da LXCORE!LxpControlDeviceIoctlAdssBusInstance+0xd4
0c ffffc200`07d34800 fffff801`19248bf1 LXCORE!LxpControlDeviceIoctl+0xaa
0d ffffc200`07d34850 fffff801`192480a6 nt!IopXxxControlFile+0x9e1
0e ffffc200`07d34a20 fffff801`18f53e83 nt!NtDeviceIoControlFile+0x56
0f ffffc200`07d34a90 00007ffd`6fdf19a7 nt!KiSystemServiceCopyEnd+0x13
10 00000006`ed6ff398 00007ff7`f46a14b9 ntdll!ZwDeviceIoControlFile+0x17
11 00000006`ed6ff3a0 00007ff7`f46a1d84 PicoMe+0x14b9
12 00000006`ed6ff6f0 00007ff7`f46a1c8e PicoMe+0x1d84
13 00000006`ed6ff730 00007ff7`f46a1b4e PicoMe+0x1c8e
14 00000006`ed6ff790 00007ff7`f46a1d99 PicoMe+0x1b4e
15 00000006`ed6ff7c0 00007ffd`6f66c902 PicoMe+0x1d99
16 00000006`ed6ff7f0 00007ffd`6fd9bfa4 KERNEL32!BaseThreadInitThunk+0x22
17 00000006`ed6ff820 00000000`00000000 ntdll!RtlUserThreadStart+0x34
```

DESIGN ISSUES IN PREVIEW BUILDS

- All WSL process' handles were kernel handles
  - Handle table appears empty to all tools and software
  - Impossible to determine resource access
- Could inject Win32 threads into Pico processes
  - But can still change context of Pico threads
  - Kernel-mode callers can still do the above – could still cause issues
- Could inject memory/duplicate handles/read/write memory of Pico processes from Win32 processes
  - Allocations < 64KB are allowed for Pico processes, due to compatibility
- No PEB, no NTDLL, no TEB, etc… and the main executable is ELF
  - Would security products ready to handle these types of processes?
- Reached out to Microsoft in order to help address these issues and provide suggestions

STATE OF CURRENT RELEASE (THE GOOD)

- Processes now show up in Task Manager
- *SeLocateProcessImageName* now returns the correct Pico Process name
- LxCore driver is now ACLed as follows:
  - D:P(A;;GA;;;S-1-5-18)S:(**TL;;0x0;;;S-1-19-512-4096**)
  - SACL: Trust Label ACE: S-1-19-512-4096 (WINDOWS LITE)
  - In other words, only allows a Protected Process Light, at the Windows Level (5) to communicate with it
- Developer mode is now an enforcement as only way to obtain handle is through LxssManager
- Can fully see network I/O in netstat and other tools, attributed to the correct process
  - Same for file I/O in resource monitor
- *PspProcessOpen* and *PspThreadOpen* now perform similar checks for Pico processes as they do for Protected Processes – certain access rights are not permitted if one side is Win32
  - Only PROCESS_QUERY_LIMITED_INFORMATION, PROCESS_TERMINATE and SYNCHRONIZE rights allowed

STATE OF CURRENT RELEASE (THE BAD)

- Some things remain by design:
  - Because of VFS-in-kernel implementation – file handles are still kernel handles
    - Similar for any Linux synchronization file descriptors that map to NT objects
  - The reality is that Pico processes execute ELF binaries, so no PE files / image sections (aka no DLLs, etc.)
    - And hence no AppLocker
- Others are still dubious, but understandable due to compatibility concerns:
  - Still no documented API for 3rd parties to receive Pico process notifications
  - No API at all for Pico thread notifications
- Also, minor WinDbg annyances:
  - !process does not understand Pico processes (will show “System Process” for all of them)
  - No symbols for lxcore, so cannot analyze Pico processes or their state

ATTACK SURFACE ANALYSIS

- 216 system calls that can now result in possible local privilege escalation
  - Yay! An extra 700KB attack surface!
- Full network access (within firewall rules)
- Full disk access (within token rules)
  - Ransomelfware, anyone?
- BSODs were found – accidentally – during the Insider Preview
  - One NULL pointer dereference
  - One invalid pointer dereference (may have led to LPE)
  - And this is without anyone actually fuzzing this thing!
- At least the IPC interfaces are locked down… (for now)
  - <u>But an unprivileged user can replace the init daemon!</u>

SECURITY DESIGN CONSIDERATIONS

KERNEL CALLOUT & API BEHAVIOR FOR ENDPOINT PRODUCTS

PROCESS / THREAD NOTIFICATIONS & BEHAVIOR

- Pico Processes will not result in notifications registered through *PsSetCreateProcessNotifyRoutine* or *PsSetCreateProcessNotifyRoutineEx*
- Pico Threads will not result in notifications registered through *PsSetCreateThreadNotifyRoutine* or *PsSetCreateThreadNotifyRoutineEx*
- Undocumented API exists: *PsSetCreateProcessNotifyRoutineEx2* which allows requesting notifications for Pico Processes
    - The PS_CREATE_NOTIFY_INFO Flags field now contains an undocumented field to indicate this is a Pico process
    - Used by tcpip.sys, but not Windows Defender (probably would have to document it at that point)
- Essentially no documented way to have visibility into the creation/termination of Pico applications
- WARNING: No NTDLL, no PEB, no TEBs, no KUSER_SHARED_DATA, no API Set Map, no Section Object!

IMAGE LOAD NOTIFICATIONS & BEHAVIOR

- Nothing is ever loaded with SEC_IMAGE inside of a Pico Process – no PE files exist so no Image Section Objects can be created
- As such, no callbacks received if registered through *PsSetLoadImageNotifyRoutine*
  - Careful: many security products rely on these callbacks to either see NTDLL loading or to see the .EXE itself loading (indicates process is now running, and called in-process, vs. process was started, and called out-of-process)
- That being said, if _all_ memory mappings are enumerated (undocumented), will see SEC_FILE mappings associated with the ELF binary and the .so files which have been mapped inside of it
  - For example, see Process Hacker
- Everything is ELF.
  - PE parsers will not work/break

MINIFILTER & WFP NOTIFICATIONS AND BEHAVIOR

- Filter Manager *will* issue notifications for all File I/O issued from a Pico Process
- Unfortunately, because all I/O is done by the kernel through IoCreateFile/ZwReadFile/ZwWriteFile APIs which will set Previous Mode to kernel
  - All file handles will thus be kernel handles!
- Additionally, attempting to lookup the PID/TID will return a Pico Process – with no API to actually indicate that this is a Pico Process
  - Will probably confuse many security products and make them crash/assert
  - Additionally, if product keeps PID/TID state: won't find anything, because of lack of notifications
- QUICK DEMO: SysInternals Procmon and Pico Processes

FORENSIC ANALYSIS (EXTRA)

VISIBILITY INTO LINUX PROCESSES

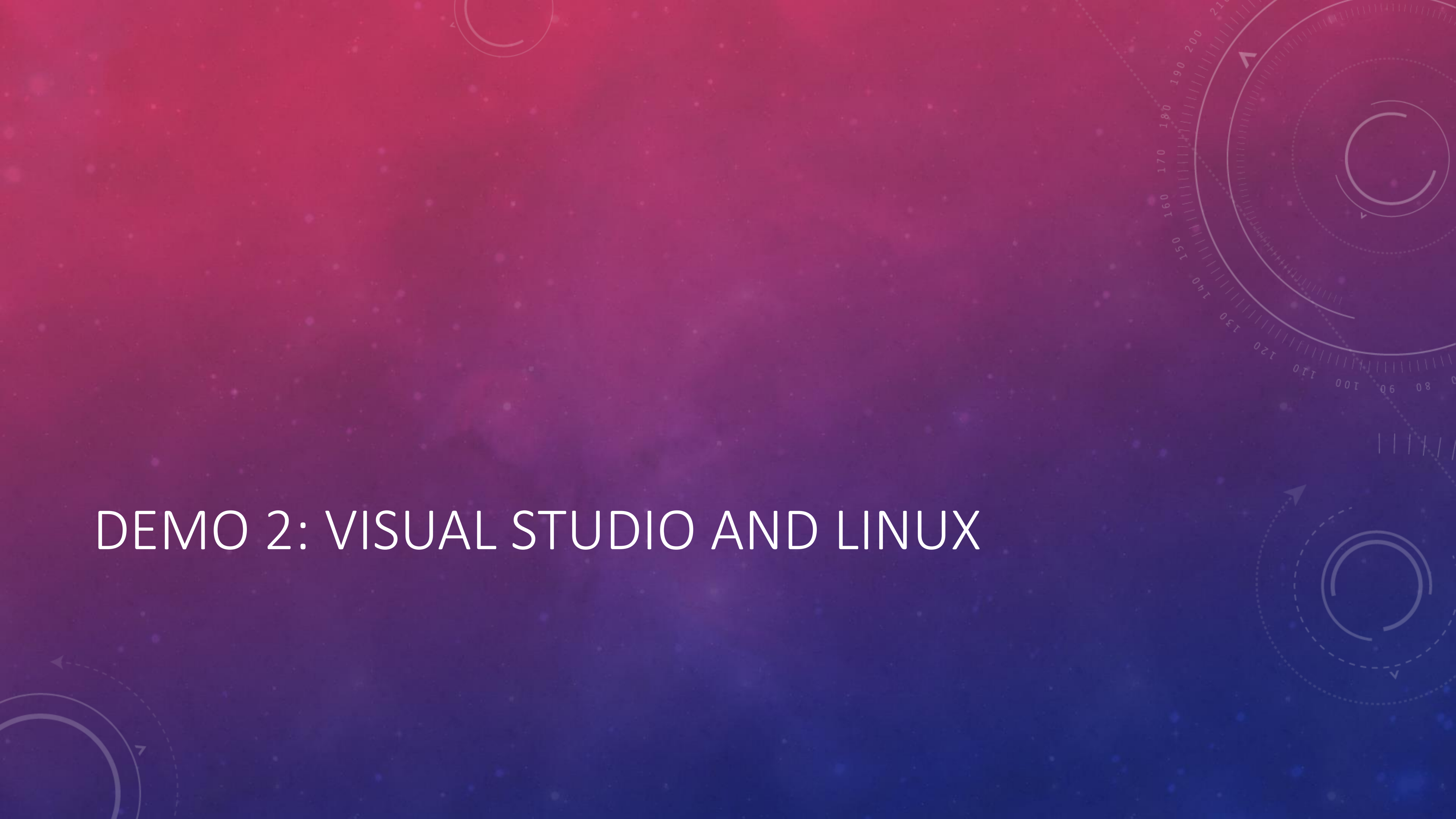
DBGVIEW

- With certain tracing settings turned on, can see significant volume of data from LxCore
  - LxpAdssBusLoggingEnabled – full view into all ADSS/BUS IPC communications
  - LxKdBreakPointErrorLevel – full view of every error/warning
    - However, will NT_ASSERT in many cases, so a debugger is needed to avoid a blue screen of death
  - LxpTraceLastSyscall – full view of every system call

WINDBG

- The debugger does not have any type information for lxss.pdb at this time
  - Requesting Microsoft to add some basic types/extensions (or use NatVis) to dump current WSL instances, thread groups, processes, and file descriptors opened
  - Will be working on writing my own scripts/extensions to dump this data – expect to publish on my GitHub (ionescu007) page at a later time
- Start with lxcore!LxGlobal
  - Contains a linked list of all currently active instances
- Dump each instance…
  - Contains linked list of all running thread groups, etc…

DEMO 1: ANALYSIS OF A PICO PROCESS



DEMO 2: VISUAL STUDIO AND LINUX

DEMO 3: IPC BETWEEN WINDOWS AND LINUX

CONCLUSION

- Microsoft took the time to both address an onslaught of user requests for more functionality (over 700 issues filed) throughout the Insider Preview as well as actually address absolutely 100% of the technical issues I privately brought up to them
- The publishing of the blog posts and videos provides useful, good, internals information for researchers, power users, and administrators
- However – have not seen actual guidance for AV vendors at this point
    - *PsSetCreateProcessNotifyRoutineEx2* remains undocumented
    - No *PsIsPicoProcess* and/or *PsIsMinimalProcess* or similar documented API
    - Most security software would probably crash/assert when hit with processes that have no PEB, NTDLL, etc…
    - Should vendors start building ELF parsers? Should they launch their Linux AV SKU (if they have one) in WSL?
- Afraid that vendors will do what they do best: hook/hack their way around, use undocumented data structures and APIs

REFERENCES & GREETZ

- Sincere thanks to the following people for their assistance, support and help with this presentation and WSL-related research:
  - Arun Kishan, Nick Judge, Jamie Schwartz, Deepu Thomas, Jason Shirk, John Lento
- Be sure to check out the official WSL Blog and GitHub as well as the Product Page (release notes, etc…)
  - https://blogs.msdn.microsoft.com/wsl
  - https://github.com/Microsoft/BashOnWindows
  - https://msdn.microsoft.com/en-us/commandline/wsl/
- https://github.com/ionescu007/lxss for presentation slides and code

THANK YOU