Windows Security Training

Course Catalog

CodeMachine

https://codemachine.com | @codemachineinc | info@codemachine.com

Table of Contents

Title	Windows Internal Architecture
Duration	5 Days
Description	The Windows PC continues to be the primary productivity device in enterprises small and large alike. Due to its ubiquity, the Windows desktop remains the favorite target for attackers to gain initial access into an organization, move laterally, and maintain their foothold. Whether you analyze malware, perform security research, conduct forensic investigations, engage in adversary simulation, or prevent it, or build security solutions for Windows, understanding how Windows works internally is critical to be effective at your task. This unique course takes you through a journey of Windows internals as it applies to user-mode execution i.e., applications and services. Everything is examined through the lens of security both from an offense and defense perspective. For each topic that is covered, components, architecture, data structures, debugger commands, and APIs are discussed with the hands-on labs helping with observing things in action and thus solidifying the understanding of the topic. This training course focuses on security-related topics and does not cover topics related to Win32 application development.
Hands-On Labs	In the hands-on lab exercises, students dig into the user and kernel mode components of Windows using debugger (WinDBG/KD) commands and learn how to interpret their output to understand the behind-the-scenes operations of the system. Students also run various custom tools that poke at certain security features of Windows and observe their behavior. Hands-on lab exercises are performed on pre-captured memory dumps and on a live VM running the latest version of Windows 10 64-bit.
Prerequisites	Attendees must have a solid understanding of operating system concepts and have a working knowledge of Windows. This course does not require any programming knowledge.
Learning Objectives	Understand the key principles behind the design and implementation of the Windows operating system. Understand the components in the Windows operating system and the functionality they provide. Understand the functionality provided by Windows that make applications and services tick. Understand the facilities in the system that are commonly abused by malware. Understand the security mitigations available in Windows that raise the bar against exploits and malware. Be able to investigate system data structures using the debugger and interpret the output of debugger commands. Be able to navigate between different data structures using the debugger. Be more effective at analyzing malware on Windows systems. Be more effective at forensic analysis of Windows systems.
Topics	System Architecture, User Mode Execution, Memory Management, PE Files, Objects and Handles, Security, Services Infrastructure, Security Mitigations.

Module	**System Architecture**
Description	The objective of this section is to learn about the architecture of the modern Windows platform with topics such as user-mode and kernel-mode execution, user and kernel components, process and system address space, functionality provided by NTDLL, call flow from Win32 applications to the kernel, WinDBG and symbols, differences between system and process memory dumps and their contents, Hyper-V, VBS, virtual trust levels (VTLs) and trustlets.
Sub-Topics	System Architecture, User Mode Execution, Memory Management, PE Files, Objects and Handles, Security, Services Infrastructure, Security Mitigations.

Module	**Memory Management**
Description	The objective of this section is to understand the process virtual address space and its components. It covers topics such as types of ALSR, page protection flags, no access pages, guard pages, execute and no-execute pages, virtual address descriptors, usage of VAD in memory forensics, remote process attachment, reading and writing remote process memory, section objects, shared memory and memory-mapped files, executable image mapping, pagefile backed regions, thread stacks, stack cookies and process heaps.
Sub-Topics	Process address space, Address space layout randomization (ASLR), Memory protection, Data execution prevention (DEP), Virtual address descriptors (VADs), Cross process memory access, Memory-mapped files, Thread stacks, Heaps.

Module	**PE Files**
Description	The objective of this section to discuss PE files and how they are loaded into memory. It covers topics such as PE files structures, navigating PE headers, sections and permissions, .NET executable, parsing export directory, importing functions and import address table, 64-bit relocation table, NTDLL loader, PEB module list, thread local storage (TLS), TLS callbacks, kernel image notification callbacks.
Sub-Topics	PE files layout, Data directories, Debug directory, CLR Header, Export directory, Import directory, Relocations, Loader functionality, DLL load callbacks.

Module	**Objects and Handles**
Description	The objective of this section is to understand objects and handle tables. It covers topics such as kernel objects, GDI objects, user objects, global and session-specific namespace, symbolic links, object headers, optional object headers, handle tables, granted access mask, handle creation and duplication, kernel object callbacks, object reference counting, types objects and object type procedures.
Sub-Topics	Windows Objects, Object namespace, Object headers, Process handle tables, Handle duplication, Object callbacks, Object reference counting, Type objects

Module	**Security**
Description	The objective of this section to learn about how the kernel secures access to objects. It covers topics such as tokens, SIDs, well-known SIDs, privileges, restricted tokens, user account control (UAC), security descriptors, discretionary access control lists (DACLs), system access control lists (SACLs), access control entries (ACEs), access masks, ACE inheritance, mandatory integrity control, mandatory policy, ACE evaluation, impersonation tokens and impersonation levels.
Sub-Topics	Tokens, Security identifiers (SID), Privileges, Security descriptors, Access control lists, Integrity levels, Access checks, Impersonation

Module	**Services Infrastructure**
Description	The objective of this section is to understand how services work in Windows. It covers topics such as service control applications, service control manager (SCM), types of services, service dependencies, executable services, DLL based services, trusted installer service, logon architecture, logon sessions, window stations, desktops, shatter attacks, session 0, user interface privilege isolation (UIPI).
Sub-Topics	Service architecture, Service control manager, Service configuration, SVCHost.exe, Service SIDs, Trusted Installer, Logon sessions, Session isolation.

Module	**Security Mitigations**
Description	The objective of this section is to understand the different security mitigations that have been added to protect applications over the years. It covers topics such as process/thread attributes, process mitigations policies, exploit guard functionality, code integrity guard, control flow guard, enhanced control flow guard, shadow stacks, low-box tokens, app-container namespace, filtering native API calls into Win32K.sys.
Sub-Topics	Process mitigation policies, Exploit guard, Arbitrary code guard (ACG), Control flow guard (CFG), Enhanced control flow guard (xFG), Control-flow enforcement technology (CET), Win32K API call filtering.

Title	Windows Malware Techniques
Duration	5 Days
Description	User mode malware on Windows is ubiquitous and custom user mode implants are used regularly in red-team engagements. Knowledge of the latest malware techniques helps red teamers improve their custom tooling, malware analysts in taking apart malware, and anti-malware solution developers in designing behavioral solutions to detect malicious activity. The common theme amongst all Windows malware and implants is that they abuse the facilities provided by the Windows platform to achieve their objectives. Knowledge of the rich set of Windows APIs, understanding their usage in various stages of an implant, and leveraging them to detect and bypass various defenses in the system is essential for red and blue teamers. This training course takes attendees through a practical journey with a hands-on approach to teach them about the post-exploitation techniques used by PE file-based implants at every stage of their execution. Beneficial to both the offensive and the defensive side of the camp, the knowledge and hands-on experience gained in this training will help attendees with real-world red teaming engagements and in defending against both custom advanced persistent threat (APT) tooling and common-off-the-shelf (COTS) malware. Attendees will learn about how malware and implants interact with the latest version of Windows and how the different stages of malware abuse and exploit various components of Windows OS to achieve their goals and evade defenses
Hands-On Labs	In the hands-on labs, attendees implement various post-exploitation techniques used by PE file-based user-mode implants using Win32 and Native APIs in C and X64-bit assembler. All labs are performed on the latest version of Windows 10 64-bit so attendees can observe the impact of the latest defenses built into the system and learn how to evade them.
Prerequisites	Attendees must have a solid understanding of Windows internals and familiarity with user-mode development on Windows using Win32 APIs. This is a developer-oriented course and attendees are expected to have prior experience with C/C++ programming on Windows 10.
Learning Objectives	Build custom tooling for offensive operations. Build position independent shellcode using C/C++. Perform basic tasks required by user-mode implants. Inject and execute shellcode and DLLs in code in privileged processes. Perform code flow subversion through hooking and subvert anti-malware hooks. Beacon out and receive tasking from a C2 infrastructure. Exfiltrate data using protocol tunneling. Implement persistence and auto-execution to survive system reboots. Detect and evade various defensive mechanisms in the system
Topics	Shellcoding, System Interfaces, Code Injection, Hooking, Persistence, Communications, Self-Defense.

Module	**Introduction**
Description	The objective of this section is to introduce attendees to the Windows malware landscape and discuss the malware ecosystem. It covers topics such as the different stages of malware execution, common vectors through which stage zero establishes a beachhead on the system, differences between staged and stageless malware, and logging mechanisms that can be enabled to detect various execution stages.
Sub-Topics	Offense and defense, Platform mitigations, Attack execution stages, Initial access methods, Staging payloads, System logging, Ecosystem review

Module	**Shellcoding**
Description	The objective of this section is to learn about developing shellcode using raw x64 assembler and high-level languages. It covers topics such as shellcoding tools (MASM/NASM/YASM), x64 assembler limitations, methods for generating position-independent and self-contained functions, shellcode injection and execution techniques, sharing data assembler and C/C++, building trampolines to interface between x64 assembler and C/C++ and per-module/per-function level runtime dependencies.
Sub-Topics	Shellcoding tools, Shellcode injection, Position independent code, Trampolines, Compiler and linker flags, Runtime checks & dependencies.

Module	**System Interfaces**
Description	The objective of this section is to learn about the internal interfaces and mechanisms used by the system to support PE file execution and how these are exploited by malware. It covers topics such as the PEB, TEB, compiler tricks to access low-level interfaces, NTDLL loader data structures, APIs for extracting information from PE files, looking up imported functions by checksums, structured and vectored exception handlers, and exception handling in shellcode.
Sub-Topics	Module lists, Compiler intrinsics, PE parsing, Import hashing, Structured exception handling, Dynamic exception handlers

Module	**Code Injection**
Description	The objective of this section is to learn about different code injection and execution techniques. It covers topics such as various methods of injecting shellcode and PE files into remote processes, using various system execution vectors to run the injected code, implementing a custom PE loader, mapping and unmapping PE files, challenges in injecting and executing shellcode code into WoW64 processes. Commonly used process injection techniques and their variants are also covered.
Sub-Topics	Injection & execution, Process injection techniques, Classic DLL injection, Reflective injection, Process hollowing, WoW64 process injection.

Module	**Hooking**
Description	The objective of this section is to learn about various code flow subversion techniques in EXEs and DLLs. It covers topics such as prolog and epilog hooking, evading scanners with code caves, injecting shellcode in unsigned PE files, import address table hooking, window local and global hooks, using RunDLL32 to host hook DLLs, and detecting and circumventing user-mode hooks.
Sub-Topics	Inline hooking, Code caves, Binary Trojaning, Import hooking, Windows hooks, Hook subversion.

Module	**Persistence**
Description	The objective of this section is to learn about various user-mode persistence and auto-execution vectors. It covers topics such as registry-based auto-start execution points (ASEPs), persistence using native boot executables, image file execution options (IFEO), DLL search order hijacking, DLL shimming, missing COM object hijacking, COM object search order hijacking, persistence through executable and DLL based services.
Sub-Topics	Registry ASEPs, System execution vectors, DLL hijacks, DLL proxies, COM object hijacks, Service hijacks.

Module	**Communications**
Description	The objective of this section is to learn about malware's communication with and its command and control (C2) servers. It covers topics such as WinINET APIs, using proxy aggregators and open web proxies to hide listening posts (LP), hosting C2 infrastructure, using whitelisted protocols for sending beacons and receiving tasking orders, data compression, chunking and encoding for exfiltration.
Sub-Topics	Inline hooking, Code caves, Binary Trojaning, Import hooking, Windows hooks, Hook subversion.

Module	**Self-Defense**
Description	The objective of this section is to learn about the most common detection and protection mechanisms in the latest version of Windows 10 and how malware can circumvent them. It covers topics such as detecting hostile environments, event log entries that identify malicious activity in the system, detecting endpoint security products, and common techniques for evading behavioral detection.
Sub-Topics	Environment detection, Debugger detection, VM detection, Event logging bypass, Security product detection, Evasion techniques

Title	Windows Kernel Internals
Duration	5 Days
Description	Kernel-mode software has unrestricted access to the system. This is why most anti-malware solutions and rootkits are implemented as Windows kernel modules. To analyze rootkits, identify indicators of compromise (IoC) and collect forensic evidence it is critical to have a good understanding of the architecture and internals of the Windows kernel. This course takes a deep dive into the internals of the Windows kernel from a security perspective with an emphasis on internal algorithms, data structures, debugger usage. This training course focuses on security-related topics and does not cover topics related to hardware such as plug and play, power management, BIOS, or ACPI.
Hands-On Labs	In the hands-on lab exercises, students dig into the kernel using the kernel debugger (WinDBG/KD) commands and learning how to interpret the debugger output of these commands to understand how the kernel works. Hands-on lab exercises are performed on pre-captured memory dumps and on a live VM running the latest version of Windows 10 64-bit.
Prerequisites	Attendees must have a solid understanding of operating system concepts and have a working knowledge of Windows. This course does not require any programming knowledge.
Learning Objectives	Understand the key principles behind the design and implementation of the Windows kernel. Understand the major components in the Windows Kernel and the functionality they provide. Be able to investigate system data structures using kernel debugger and interpret the output of debugger commands. Be able to navigate between different data structures in the kernel using debugger commands. Be able to locate indicators of compromise while hunting for kernel-mode malware. Be able to perform forensic analysis of the Windows kernel. Understand how kernel-mode rootkits and commercial anti-malware solutions interact with the system
Topics	Kernel Architecture, Processes and Threads, System Mechanisms, Execution Contexts, Synchronization, Memory Management, I/O Management, Kernel Security Mitigations

Module	**Kernel Architecture**
Description	The objective of this section is to learn about the architecture of the Windows kernel and key kernel-mode components. It covers topics such as privilege levels, segment registers, global descriptor table (GDT), modern PC platform, NTOSKRNL component list, HAL, Win32K.sys refactoring, kernel module list, code integrity (CI), driver load notification callbacks.
Sub-Topics	Execution rings, Platform architecture, Kernel-mode components, NTOSKRNL layers, Kernel module list, Image notification callbacks

Module	**Processes and Threads**
Description	The objective of this section to learn about how the support provided by the kernel for user-mode code execution. It covers topics such as process resources, process and thread data structures (EPROCESS/KPROCESS, EHTREAD/KTHREAD), system processes, system idle process, minimal processes, system call dispatching, user-mode and kernel-mode stacks, different lists that processes and threads are maintained in the kernel and process/thread creation and termination callbacks.
Sub-Topics	Processes and threads, Process and thread data structures, Special processes, Process resources, System calls, User kernel transition, Process lists, Process and thread callbacks

Module	**System Mechanisms**
Description	The objective of this section to discuss the foundational building blocks of the system that kernel components rely on. It covers topics such as Zw/Nt APIs, model-specific registers, dispatching native API to NTOSKRNL.exe and Win32K.sys, 64-bit SSDT, machine frames, trap frames, .PDATA section, runtime image info structures, exception handling, KPCR, KPRCB, TEB, IRQLs, and DISPATCH_LEVEL restrictions.
Sub-Topics	Native APIs, Model-specific registers (MSRs), System service dispatching, Trap frames Exception handling, Kernel process control region (KPCR), Interrupt request levels (IRQL)

Module	**Execution Contexts**
Description	The objective of this section is to learn about the different mechanisms available for kernel-mode code execution. It covers topics such as kernel timers, executive timers, DPCs, user APCs, kernel APCs, special kernel APCs, process/thread suspend/resume, system worker threads, work items, executive work queues, custom driver worker threads.
Sub-Topics	Kernel timers, Deferred procedure calls (DPC), Asynchronous procedure calls (APC), Thread suspend/resume, System worker threads, Work items, Custom driver threads

Module	**Synchronization**
Description	The objective of this section is to learn about the different synchronization primitives available in the Windows kernel. It covers topics such as dispatcher objects, thread waitlists, interlocked operations, critical regions, mutually exclusive locks vs reader-writer locks, mutexes, fast mutexes, high IRQL synchronization, spin-locks, in-stack queued spin-locks, reader-writer spin-locks, and the considerations when selecting a synchronization mechanism
Sub-Topics	Dispatcher objects, Interlocked operations, Mutexes, Critical regions, Executive resources, Push-locks, Spin-locks

Module	**Memory Management**
Description	The objective of this section is to understand how kernel memory is managed by Windows. It covers topics such as physical and virtual address translation, page table entries (PTEs), physical page management, kernel virtual address space (KVAS) layout, page table space, session space, thread kernel stacks, stack jumping, pool types, small and large pool allocations, lookaside lists, usage of MDLs for memory mapping.
Sub-Topics	Address translation, Kernel virtual address space, Page frame number (PFN) database, Session space, Kernel stacks, Kernel pools, Memory Descriptor Lists (MDL)

Module	**I/O Management**
Description	The objective of this section is to understand how drivers interface with the Windows kernel. It covers topics such as driver dispatch entry points, driver objects, device objects, file objects, symbolic links, driver types (function, bus, filter), device types (FDO, PDO, FiDO), driver layering, device attachment/detachment, IRPs, I/O stack locations, IRP processing, I/O completion routines, I/O cancellation, I/O requests filtering.
Sub-Topics	Driver architecture, I/O manager data structures, Driver types, Device object types, IRPs and I/O stack locations (IOSLs), I/O processing, Filter drivers

Module	**Kernel Security Mitigations**
Description	The objective of this section is to understand the different exploit mitigations and anti-rootkit features that have been added to the Windows kernel over the course of its lifetime. It covers topics such as kernel attack surface, GS cookies, NULL page allocation prevention, safe linking and unlinking, executable and non-executable (NX) pools, kernel ASLR, page table base randomization, driver signature enforcement, attestation signing, PatchGuard, meltdown mitigations, software SMEP, KVA shadowing.
Sub-Topics	Kernel exploitation, Kernel data execution prevention (DEP), Kernel address layout randomization (ASLR), Supervisor mode execution prevention (SMEP), Kernel-mode code signing (KMCS), Kernel patch protection (PatchGuard), Kernel virtual address shadowing.

Title	Windows Kernel Development
Duration	5 Days
Description	Most security software on Windows runs in kernel mode. This training provides students a jumpstart into the world of Windows kernel-mode software development through a practical hands-on approach using the latest version of Visual Studio, Windows Driver Kit, and WinDBG. The topics covered in this course go much deeper than the information available in the WDK documentation or in public forums. Coverage includes use cases of various APIs, their applicability to security, behind the scenes working of functions, and common usage pitfalls. The discussions do not shy away from undocumented features and techniques that are essential when building kernel-mode drivers related to security functionality but considered out of scope by generic driver development training courses in the industry. This training course focuses on security-related topics and does not cover topics related to hardware drivers such as plug and play, power management, hardware busses (PCI, USB, Bluetooth), or the kernel-mode driver framework (KMDF).
Hands-On Labs	In the hands-on lab exercises, students develop and build drivers in C/C++ and then deploy, test, and debug these drivers on the latest build of Windows 10 64-bit running in a Hyper-V VM. Students also learn various techniques to debug these drivers using WinDBG.
Prerequisites	Attendees must be proficient in C programming. Attendees must have a good working knowledge of the windows kernel. The CodeMachine **Windows Kernel Internals** course provides the prerequisite Windows kernel knowledge required to get the maximum value from this course.
Learning Objectives	Be able to use tools such as Visual Studio and WDK or EWDK to develop and build kernel-mode software. Be able to deploy, test, and debug kernel-mode software. Be able to use the debugger effectively to perform live debugging of kernel-mode software. Understand the various options available to implement features in kernel-mode software. Be able to perform common programming tasks required by kernel-mode software. Be able to develop reasonably complex security functionality in kernel-mode. Be able to use tools other than the debugger to debug issues with kernel-mode software. Understand the intricacies of kernel-mode software development. Understand the best practices and common pitfalls when developing Windows kernel-mode software.
Topics	Driver Development Environment, Driver Programming Basics, Driver Debugging, I/O Processing, Kernel Execution, Kernel Synchronization, System Enumeration, Common Tasks

Module	**Development Environment**
Description	The objective of this section is to learn about the Windows driver development toolchain and how to use the tools effectively to code, build, deploy, test, and debug drivers. It covers topics such as header files and libraries required to build drivers, various options supported by the build process, service control manager and native APIs to load and unload drivers, debugger symbols and associated kernel debugger commands, the pros and cons of using C vs C++ to develop drivers and the effect of compiler and linker flags on code generation.
Sub-Topics	Enterprise Windows Driver Kit, Targets, Platforms & Configurations, Build Customization, Driver Registration and Loading, Driver Symbols and Source Code, Code Analysis (PreFast), Source Annotation Language (SAL), Kernel Mode Code Signing (KMCS)

Module	**Driver Development Basics**
Description	The objective of this section is to learn about the unique aspects of Windows driver development. It covers topics such as various methods of determining kernel version, header file versioning, statically & dynamically binding to APIs, handling kernel API failures, allocating memory from pools & lookaside lists, manipulating UNICODE strings.
Sub-Topics	Windows version APIs, WDK headers & libraries, Static and dynamic linking, NTSTATUS codes, Dynamic memory allocation, Lookaside lists, Kernel CRT support, Unicode string.

Module	**Driver Debugging**
Description	The objective of this section is to learn about tools and techniques to perform live debugging of kernel-mode drivers. It covers topics such as configuring kernel debugging settings, controlling driver debug output, break-pointing techniques that go beyond just the commands bp/bu/ba, etc., using various tools to find subtle bugs in kernel-mode drivers, and configuring the system to collect instrumentation data.
Sub-Topics	Debug prints, Driver replacement maps, Live debugging techniques, Common debugging tasks, Driver verifier, Special pool & pool tracking, Stack tracing, GFlags kernel settings

Module	**I/O Processing**
Description	The objective of this section is to learn about how drivers process I/O requests from user-mode applications. It covers topics such as driver/device/file objects, symbolic links, Win32 I/O APIs, driver entry points, dispatch routines, IRPs and I/O stack locations, synchronous and asynchronous processing of IRPs, completion routines, cancel routines, IOCTL codes, and data transfer mechanisms between user mode and kernel mode.
Sub-Topics	I/O manager objects, Device namespaces, User/kernel interface, Dispatch routines, IRP processing, Device I/O control, Buffering methods, Accessing user buffers, Building IRPs, Memory mapping

Module	**Kernel Execution**
Description	The objective of this section is to learn about the different code execution mechanisms available in the kernel, their use cases, and constraints imposed on them. It covers topics such as the IRQLs, process and thread contexts, DPC routines and their triggers, system worker threads, executive work queues and worker routines, creating and managing driver threads, object reference counting and reference tracking, various mechanisms to query time in the kernel, their sources and precision.
Sub-Topics	IRQL management, System time, Kernel and executive timers, DPC poutines, APC routines, Worker routines, Object lifetime management, Kernel events, Driver managed threads, Exception handling

Module	**Kernel Synchronization**
Description	The objective of this section is to learn about the synchronization primitives available to drivers to perform multiprocessor safe operations such as mutexes, ERESOURCES, push-locks, mutually exclusive spin-locks, reader-writer spin-locks, and rundown protection. It covers topics such as the different types of linked lists available to drivers and their use cases, implications of IRQL on synchronization, considerations when selecting a locking mechanism, correct usage of interlocked operations to perform low overhead synchronization, and the safe unloading of drivers from memory.
Sub-Topics	Linked lists, Waitable locks, Spin locks, Reader-writer locks, Critical and guarded regions, Recursive locks, Rundown protection, Interlocked operations

Module	**System Enumeration**
Description	The objective of this section is to learn about programming tasks that are common when developing drivers that provide security-related functionality. It covers topics such as memory mapping techniques, accessing process memory from kernel mode, options for building and sending IRPs to drivers, considerations when performing registry and file access in kernel mode, various run-time instrumentation techniques, and best practices when developing drivers.
Sub-Topics	Native API, Registry & file access, Processes & threads, Handle tables, Module list, PE file parsing, Object namespace, Physical memory ranges

Title	Windows Kernel Rootkits
Duration	5 Days
Description	To achieve maximum stealth and obtain unabated access to the system, rootkits execute in kernel mode. This course focuses on the kernel interfaces (APIs), data structures and mechanisms that are exploited by rootkits to achieve their goals at every stage of their execution. Kernel security enhancements that have been progressively added from Windows 7 to the latest version of Windows are discussed along with some circumvention techniques. This advanced course provides a comprehensive end-to-end view of the modus-operandi of rootkits by taking an in-depth look at behind the scenes working of the Windows kernel and how these mechanisms are exploited by malware through hands-on labs and real-world case studies. Kernel security enhancements that have been progressively added to Windows are discussed along with some circumvention techniques. Attendees will study key techniques used by rootkits to understand the real-world applicability of these concepts for offensive and defensive purposes. This training is beneficial to anyone responsible for developing, detecting, analyzing, and defending against rootkits and other Windows kernel post-exploitation techniques including EPP/EDR software developers, anti-malware engineers, security researchers, red/blue/purple teamers.
Hands-On Labs	Every topic in this course is accompanied by hands-on labs where attendees get to implement key components of a rootkit and test them on 64-bit Windows systems to reinforce their understanding of the theory.
Prerequisites	Attendees must be proficient in C/C++ programming, have a good understanding of Windows kernel internals and APIs, and be able to use the kernel debugger (WinDBG) to debug kernel modules. CodeMachine's **Windows Kernel Internals** and **Windows Kernel Development** courses provide the Windows kernel knowledge required to get full value from this course.
Learning Objectives	Understand vulnerabilities in the Windows kernel and device drivers. Be able to write and modify kernel-mode exploits. Understand the security enhancements that have been added to the Windows kernel over time. Be able to bypass some of the security mitigations in recent versions of Windows. Understand the post-exploitation steps performed by kernel-mode rootkits. Understand the techniques used by real-world rootkits. Understand how rootkits hide their presence in the system. Understand how rootkits intercept systemwide networking activity. Be able to identify malicious behavior and defend against rootkits.
Topics	Kernel Attacks, Kernel Shellcoding, Kernel Hooking and Injection, Kernel Callbacks, Kernel Filtering, Kernel Networking, Virtualization Based Security

Module	**Kernel Attacks**
Description	The objective of this section is to learn about vulnerabilities in kernel-mode drivers and how they are exploited by attackers to escalate privilege and gain code execution. It covers topics such as phases of rootkit execution, remote code execution, local code execution, hostile environment detection, kernel exploitation primitives, exploiting vulnerable drivers, determining kernel version, privilege escalation methods, internal kernel structure manipulation, and methods of controlling the kernel-mode instruction pointer. System defenses such as kernel-mode address space layout randomization (KASLR), supervisor mode execution prevention (SMEP), and kernel virtual address shadowing (KVAS) are also covered.
Sub-Topics	Kernel attack workflow, Types of vulnerabilities, Environment detection, Exploiting drivers, Direct kernel object manipulation (DKOM), Privilege escalation, Kernel execution vectors

Module	**Kernel Shellcoding**
Description	The objective of this section is to learn about developing kernel-mode shellcode using raw x64 assembler and high-level languages. It covers topics such as kernel-mode shellcode consideration and constraints, shellcoding tools (MASM/NASM/YASM), x64 assembler limitations and workarounds, developing kernel-mode shellcode in C/C++, leveraging the PE file .pdata section, shellcode injection and execution, reflective driver loading, methods for bypassing write protection in kernel-mode, calling kernel internal (un-exported) functions, accessing kernel internal global data structures, removing shellcode execution artifacts. System defenses such as driver signature enforcement (KMCS), kernel-mode DEP, and non-executable non-paged pool (NonPagedPoolNx) and are also covered.
Sub-Topics	Kernel mode shellcode, Shellcoding tools, Shellcoding in C/C++, PE exception table, Calling non-exported functions, Kernel Payload Loader, Circumventing memory protection

Module	**Kernel Hooking and Injection**
Description	The objective of this section is to learn about code flow subversion techniques in the kernel and methods to inject and executed code into user-mode processes from kernel-mode. It covers topics such as function prolog/epilog hooking, trampolines, multi-processor synchronization, code caves, kernel function tables, function pointer hooking, stealth filtering through data structure redirection, user-mode code-injection, user-mode APCs, thread register context manipulation, user-mode callbacks, and hook detection methods. System defenses such as kernel control flow guard (KCFG), kernel patch protection (PatchGuard) are also covered.
Sub-Topics	Code flow subversion methods, Function hooking, Function pointer hijack, Import hooking Data structure hooking, Code injection and execution, Hook detection

Module	**Kernel Callbacks**
Description	The objective of this section is to learn about the different callback mechanisms available to kernel-mode drivers to intercept systemwide activity that is interesting from a security perspective. It covers topics such as usage of process callbacks to veto process creation, thread callbacks to detect remote thread injection, image notification callbacks to block driver loading, object manager callback to block process memory access, shutdown callbacks to implement self-protection, bug-check callbacks for anti-forensics, and power callbacks for user presence detection.
Sub-Topics	Process callbacks, Thread callbacks, Image notification callbacks, Object manager callbacks, Shutdown notifications, Bug-check callbacks, Power notification callbacks

Module	**Kernel Filtering**
Description	The objective of this section is to learn about the different filtering mechanisms available to kernel-mode drivers to intercept device, file access, and registry access. It covers topics such as device stacks, filter drivers, usage of IRP filters for keylogging and disk access, filter registration, dynamic device attachment/detachment, registry filters, registry key context management, hiding registry entries, filter manager, FS mini-filters, context management, hiding directory entries, finding filter driver callbacks neutering filter callbacks.
Sub-Topics	Filtering models, IRP filters, PnP hardware detection, Stealth filtering, Registry filters, File system mini-filters, Neutering filters

Module	**Kernel Networking**
Description	The objective of this section is to discuss the kernel-mode networking stack components, the interfaces to intercept networking activity in the system, and inject network data. It covers topics such as Windows networking architecture, kernel network interfaces, network packet data representation, NDIS drivers (miniport, intermediate, protocol, and filter), WFP architecture, Windows firewall, content inspection/modification of TCP streams, network layer-2 filtering, NDIS internal data structures and low-level network I/O.
Sub-Topics	Kernel network interfaces, Net buffer lists (NBL) and net buffers (NB), Windows filtering platform (WFP), WFP MAC layer filtering, NDIS driver types, NDIS lightweight filters (LWF), NDIS internal data structures and hooking

Module	**Virtualization Based Security**
Description	The objective of this section is to learn about virtualization-based security and its impact on rootkits. It covers topics such as Hyper-V platform requirements, VT-x/AMD-V, second-level address translation (SLAT), mode based execution control (MBEC), Hyper-V top-level functional specification (TLFS), virtual trust levels (VTL0/VTL1), normal kernel, secure kernel, secure kernel patch guard (SKPG) functionality, HVCI restrictions on kernel drivers, the effects of KCFG on code flow subversion, secure pools and the effects of KDP on DKOM.
Sub-Topics	Hyper-V Architecture, Virtual Trust Levels (VTL), Secure Kernel (SK), HyperGuard (SKPG), HyperVisor Protected Code Integrity (HVCI), Kernel Control Flow Graph (KCFG), Kernel Data Protection (KDP)

Title	Windows Kernel Debugging
Duration	5 Days
Description	All software has bugs. When a bug in an application manifests itself, it affects just that one application. Whereas a bug in a kernel mode driver affects the entire system and often leads to the infamous Blue Screen of Death (BSOD). The color of that infamous screen may have changed in recent versions of Windows, but the underlying causes of failure have not. This course is targeted at security researchers, software developers, support engineers who must regularly analyze and debug Windows kernel mode software. This course builds the necessary foundation for effective kernel debugging with topics such as configuring the debugger, debug symbols, performing basic debugging tasks, debugger scripting, retrieving function parameters from x64 stack, mapping x64 assembler to high-level language constructs, etc. It then dives into various techniques and strategies that can be applied to perform triaging, fault isolation, root cause analysis of crashes and hangs caused by kernel mode drivers. Also, this course touches upon the identification of malicious behavior in the kernel commonly exhibited by rootkits. This training course focuses on software and security related drivers and does NOT cover issues related to hardware drivers for PCI, USB, Bluetooth devices.
Hands-On Labs	In the hands-on lab exercises, students work on a wide variety of kernel mode crash and hang dumps that have been captured on various versions of Windows ranging from Windows XP to Windows 10. Each memory dump involves applying Windows internals knowledge and a unique set of debugging techniques to go from "!analyze-v" to isolating the module responsible for the crash or hang to determining the root cause of the problem and potential ways of fixing it.
Prerequisites	Attendees must have a good working knowledge of the windows kernel. The CodeMachine **Windows Kernel Internals** course provides the prerequisite Windows kernel knowledge required to get the maximum value from this course.
Learning Objectives	Understand the architecture and components of Windows Debugger (WinDBG/KD). Be able to use the kernel debugger commands to achieve common debugging tasks. Be able to automate common debugging tasks using debugger scripting. Be able to map X64 assembler to high-level language (C/C++) code constructs. Understand the calling convention, parameter passing, stack usage on 64-bit systems. Be able to retrieve register-based parameters from the x64 call stacks through non-volatile registers. Be able to identify symptoms of system failure/instability, perform bug triaging, and fault isolation. Be able to debug hard-to-reproduce hangs and crashes. Be able to analyze and root cause problems down to a code change in kernel modules. Be able to detect kernel mode rootkits in the system.
Topics	Windows debugger, X64 assembler, X64 call stacks, Reverse engineering, Debugger automation, Crash dump analysis, Debugging deadlocks and hangs, Advanced analysis techniques, Debugging tools

Module	**Windows Debugger**
Description	The objective of this section is to learn about the kernel debugger, debugging symbols, and debugger usage. It covers topics such as configuring a system for kernel debugging, ramifications of kernel debugging, symbol (.PDB) file contents, resolving issues related to symbol files, using third party debugger extensions, differences between the various memory dump types (complete, kernel, active, automatic) and generating process and system memory dumps manually.
Sub-Topics	Debugger architecture, Debugger extensions, Debugger command types, Symbol files Symbol server, Kernel debugging, Memory dumps, Manual memory dump generation

Module	**x64 Assembler**
Description	The objective of this section is to learn about the x64 CPU, registers, instructions, and the basics of reading and understanding x64 assembler required for reverse engineering. It covers topics such as x64 instruction set, x64 instruction encoding, absolute and relative offsets, little-endian vs big-endian, sign and zero extension, most frequently used instruction types (data transfer, arithmetic, logical, control flow, etc.), condition flags and conditional jumps, direct and indirect control transfers, basic blocks, control flow graphs.
Sub-Topics	Hardware architecture, X64 CPU registers, X64 Instruction set, Number representation, Instruction types, Control flow transfer, Function basic blocks, Common instruction patterns

Module	**x64 Call Stacks**
Description	The objective of this section is to learn how stacks work on 64-bit systems and how to use knowledge of 64-bit stack layout to retrieve information from the stack. It covers topics such as x64 calling convention, components of x64 stack frame, stack canaries, local variables and parameters, parameter homing space, interpreting the x64 call stack displayed by the debugger, retrieving register-based parameters from the stack and compiler flags affecting stack frame generation.
Sub-Topics	Function calls and stacks, Call stacks and stack frames, Calling convention, Stack pointer, Non-volatile registers, Function prolog and epilog, Interpreting call stacks, Retrieving function parameters

Module	**Reverse Engineering**
Description	The objective of this section is to learn how to map x64 assembler to high-level language (C/C++) code constructors. It covers topics such as CPU register usage, identifying access to function parameters, local variables, global variables, structures fields, array elements, list nodes, etc., determining branch conditions, backtracking execution based on register and memory state, identifying code transformation due to compiler optimizations.
Sub-Topics	Register usage, Variable types, Global variables, Local variables and parameters, Structure layout, Arrays, Compiler optimization, Control flow analysis

Module	**Debugger Automation**
Description	The objective of this section is to learn about the debugger's scripting support and how to use it to automate common debugging and analysis tasks. It covers topics such as appropriate usage of MASM and C++ expression evaluators, debugger command pipelines, rules for developing and executing debugger scripts, using pseudo-registers, and aliases to reduce script complexity, formatting debugger output and other automation techniques.
Sub-Topics	Expression evaluators, Pseudo Registers, Dereferencing Memory, Iterators, Macros, Debugger scripts, Control flow tokens, Aliases, String Comparison

Module	**Crash Dump Analysis**
Description	The objective of this section is to learn about why and how bug-checks happen, bug triaging, and fault isolation. It covers topics such as conditions leading to system bugchecks, common causes of dump generation failures, interpreting the information from debuggers' automated analysis, using bugcheck information to determine the appropriate register context, performing sanity check on registers, obtaining system state using debugger extensions to identify runtime anomalies, triaging bugs and isolating faulty modules.
Sub-Topics	System bugchecks, Crash dump generation, Types of bugchecks, Automated analysis, Module identification, Context switching, Hardware failures, Examining system state

Module	**Debugging Deadlocks and Hangs**
Description	The objective of this section is to learn about debugging and analyzing systems hangs and performance issues. It covers topics such as types of hangs, waitable locks available in kernel mode (mutexes, fast mutexes, ERESOURCES, push-locks), spin-locks (exclusive spin-locks, reader writer), causes of deadlocks, deadlock detection, dependency analysis, debugging blocked power state transitions, identifying stalls in the storage path, tracking pending I/O requests, identifying overconsumption of system resources and finding the culprits.
Sub-Topics	Causes of hangs, Classic deadlocks, Deadlock with spinlocks, Deadlock analysis, Driver power state failure, I/O request stalls, Resource exhaustion.

Module	**Advanced Analysis Techniques**
Description	The objective of this section is to learn deep analysis techniques that can be used to root cause problems into specific bugs in kernel modules. It covers topics such as recognizing stack patterns, determining debugging workflow, debugging invalid memory access at various IRQLs, common causes of pool corruption, identifying corruption patterns, debugging double faults, debugging multi-driver interactions, finding artifacts of kernel mode rootkits.
Sub-Topics	Debugging strategies, Stack patterns, Invalid memory access, Pool corruption, Structure corruption, Stack corruption and overflows, Rootkit IoCs