Windows User Mode Exploit Development

Offensive Security

Copyright © 2021 Offensive Security Ltd.

All rights reserved. No part of this publication, in whole or in part, may be reproduced, copied, transferred or any other right reserved to its copyright owner, including photocopying and all other copying, any transfer or transmission using any network or other means of communication, any broadcast for distant learning, in any form or by any means such as any information storage, transmission or retrieval system, without prior written permission from the author.

Table of Contents

1 Windows User Mode Exploit Development: General Course Information
  1.1 About the EXP-301 Course
  1.2 Provided Materials
    1.2.1 EXP-301 Course Materials
    1.2.2 Access to the Internal VPN Lab Network
    1.2.3 The Offensive Security Student Forum
    1.2.4 Live Support and RocketChat
    1.2.5 OSED Exam Attempt
  1.3 Overall Strategies for Approaching the Course
    1.3.1 Welcome and Course Information Emails
    1.3.2 Course Materials
    1.3.3 Course Exercises
  1.4 About the EXP-301 VPN Labs
    1.4.1 Control Panel
    1.4.2 Reverts
    1.4.3 Kali Virtual Machine
    1.4.4 Lab Behavior and Lab Restrictions
  1.5 About the OSED Exam
  1.6 Wrapping Up
2 WinDbg and x86 Architecture
  2.1 Introduction to x86 Architecture
    2.1.1 Program Memory
    2.1.2 CPU Registers
  2.2 Introduction to Windows Debugger
    2.2.1 What is a Debugger?
    2.2.2 WinDbg Interface
    2.2.3 Understanding the Workspace
    2.2.4 Debugging Symbols
  2.3 Accessing and Manipulating Memory from WinDbg
    2.3.1 Unassemble from Memory
    2.3.2 Reading from Memory
    2.3.3 Dumping Structures from Memory
    2.3.4 Writing to Memory
    2.3.5 Searching the Memory Space

2.3.6 Inspecting and Editing CPU Registers in WinDbg
2.4 Controlling the Program Execution in WinDbg
2.4.1 Software Breakpoints
2.4.2 Unresolved Function Breakpoint
2.4.3 Breakpoint-Based Actions
2.4.4 Hardware Breakpoints
2.4.5 Stepping Through the Code
2.5 Additional WinDbg Features
2.5.1 Listing Modules and Symbols in WinDbg
2.5.2 Using WinDbg as a Calculator
2.5.3 Data Output Format
2.5.4 Pseudo Registers
2.6 Wrapping Up
3 Exploiting Stack Overflows
3.1 Stack Oveflows Introduction
3.2 Installing the Sync Breeze Application
3.3 Crashing the Sync Breeze Application
3.4 Win32 Buffer Overflow Exploitation
3.4.1 A Word About DEP, ASLR, and CFG
3.4.2 Controlling EIP
3.4.3 Locating Space for Our Shellcode
3.4.4 Checking for Bad Characters
3.4.5 Redirecting the Execution Flow
3.4.6 Finding a Return Address
3.4.7 Generating Shellcode with Metasploit
3.4.8 Getting a Shell
3.4.9 Improving the Exploit
3.5 Wrapping Up
4 Exploiting SEH Overflows
4.1 Installing the Sync Breeze Application
4.2 Crashing Sync Breeze
4.3 Analyzing the Crash in WinDbg
4.4 Introduction to Structured Exception Handling
4.4.1 Understanding SEH
4.4.2 SEH Validation

4.5 Structured Exception Handler Overflows
4.5.1 Gaining Code Execution
4.5.2 Detecting Bad Characters
4.5.3 Finding a P/P/R Instruction Sequence
4.5.4 Island-Hopping in Assembly
4.5.5 Obtaining a Shell
4.6 Wrapping Up
5 Introduction to IDA Pro
5.1 IDA Pro 101
5.1.1 Installing IDA Pro
5.1.2 The IDA Pro User Interface
5.1.3 Basic Functionality
5.1.4 Search Functionality
5.2 Working with IDA Pro
5.2.1 Static-Dynamic Analysis Synchronization
5.2.2 Tracing Notepad
5.3 Wrapping Up
6 Overcoming Space Restrictions: Egghunters
6.1 Crashing the Savant Web Server
6.2 Analyzing the Crash in WinDbg
6.3 Detecting Bad Characters
6.4 Gaining Code Execution
6.4.1 Partial EIP Overwrite
6.4.2 Changing the HTTP Method
6.4.3 Conditional Jumps
6.5 Finding Alternative Places to Store Large Buffers
6.5.1 The Windows Heap Memory Manager
6.6 Finding our Buffer - The Egghunter Approach
6.6.1 Keystone Engine
6.6.2 System Calls and Egghunters
6.6.3 Identifying and Addressing the Egghunter Issue
6.6.4 Obtaining a Shell
6.7 Improving the Egghunter Portability Using SEH
6.7.1 Identifying the SEH-Based Egghunter Issue
6.7.2 Porting the SEH Egghunter to Windows 10

6.8 Wrapping Up
7 Creating Custom Shellcode
7.1 Calling Conventions on x86
7.2 The System Call Problem
7.3 Finding kernel32.dll
7.3.1 PEB Method
7.3.2 Assembling the Shellcode
7.4 Resolving Symbols
7.4.1 Export Directory Table
7.4.2 Working with the Export Names Array
7.4.3 Computing Function Name Hashes
7.4.4 Fetching the VMA of a Function
7.5 NULL-Free Position-Independent Shellcode (PIC)
7.5.1 Avoiding NULL Bytes
7.5.2 Position-Independent Shellcode
7.6 Reverse Shell
7.6.1 Loading ws2_32.dll and Resolving Symbols
7.6.2 Calling WSAStartup
7.6.3 Calling WSASocket
7.6.4 Calling WSAConnect
7.6.5 Calling CreateProcessA
7.7 Wrapping Up
8 Reverse Engineering for Bugs
8.1 Installation and Enumeration
8.1.1 Installing Tivoli Storage Manager
8.1.2 Enumerating an Application
8.2 Interacting with Tivoli Storage Manager
8.2.1 Hooking the recv API
8.2.2 Synchronizing WinDbg and IDA Pro
8.2.3 Tracing the Input
8.2.4 Checksum, Please
8.3 Reverse Engineering the Protocol
8.3.1 Header-Data Separation
8.3.2 Reversing the Header
8.3.3 Exploiting Memcpy

8.3.4 Getting EIP Control
8.4 Digging Deeper to Find More Bugs
8.4.1 Switching Execution
8.4.2 Going Down 0x534
8.5 Wrapping Up
9 Stack Overflows and DEP Bypass
9.1 Data Execution Prevention
9.1.1 DEP Theory
9.1.2 Windows Defender Exploit Guard
9.2 Return Oriented Programming
9.2.1 Origins of Return Oriented Programming Exploitation
9.2.2 Return Oriented Programming Evolution
9.3 Gadget Selection
9.3.1 Debugger Automation: Pykd
9.3.2 Optimized Gadget Discovery: RP++
9.4 Bypassing DEP
9.4.1 Getting The Offset
9.4.2 Locating Gadgets
9.4.3 Preparing the Battlefield
9.4.4 Making ROP's Acquaintance
9.4.5 Obtaining VirtualAlloc Address
9.4.6 Patching the Return Address
9.4.7 Patching Arguments
9.4.8 Executing VirtualAlloc
9.4.9 Getting a Reverse Shell
9.5 Wrapping Up
10 Stack Overflows and ASLR Bypass
10.1 ASLR Introduction
10.1.1 ASLR Implementation
10.1.2 ASLR Bypass Theory
10.1.3 Windows Defender Exploit Guard and ASLR
10.2 Finding Hidden Gems
10.2.1 FXCLI_DebugDispatch
10.2.2 Arbitrary Symbol Resolution
10.2.3 Returning the Goods

10.3 Expanding our Exploit (ASLR Bypass)
10.3.1 Leaking an IBM Module
10.3.2 Is That a Bad Character?
10.4 Bypassing DEP with WriteProcessMemory
10.4.1 WriteProcessMemory
10.4.2 Getting Our Shell
10.4.3 Handmade ROP Decoder
10.4.4 Automating the Shellcode Encoding
10.4.5 Automating the ROP Decoder
10.5 Wrapping Up
11 Format String Specifier Attack Part I
11.1 Format String Attacks
11.1.1 Format String Theory
11.1.2 Exploiting Format String Specifiers
11.2 Attacking IBM Tivoli FastBackServer
11.2.1 Investigating the EventLog Function
11.2.2 Reverse Engineering a Path
11.2.3 Invoke the Specifiers
11.3 Reading the Event Log
11.3.1 The Tivoli Event Log
11.3.2 Remote Event Log Service
11.3.3 Read From an Index
11.3.4 Read From the Log
11.3.5 Return the Log Content
11.4 Bypassing ASLR with Format Strings
11.4.1 Parsing the Event Log
11.4.2 Leak Stack Address Remotely
11.4.3 Saving the Stack
11.4.4 Bypassing ASLR
11.5 Wrapping Up
12 Format String Specifier Attack Part II
12.1 Write Primitive with Format Strings
12.1.1 Format String Specifiers Revisited
12.1.2 Overcoming Limitations
12.1.3 Write to the Stack

12.1.4 Going for a DWORD
12.2 Overwriting EIP with Format Strings
12.2.1 Locating a Target
12.2.2 Obtaining EIP Control
12.3 Locating Storage Space
12.3.1 Finding Buffers
12.3.2 Stack Pivot
12.4 Getting Code Execution
12.4.1 ROP Limitations
12.4.2 Getting a Shell
12.5 Wrapping Up
13 Trying Harder: The Labs
13.1 Challenge 1
13.2 Challenge 2
13.3 Challenge 3
13.4 Wrapping Up