Monday, January 1, 2018

Kernel Exploitation

# [Kernel Exploitation] 1: Setting up the environment

*The HackSysExtremeVulnerableDriver by HackSysTeam always interested me and I got positive feedback on writing about it, so here we are.*

*Repo with all code can be found here.*

---

This N-part tutorial will walk you through the kernel exploit development cycle. It's important to notice that we will be dealing with known vulnerabilities, no reversing is needed (for the driver at least).

By the end of tutorial, you should be familiar with common vulnerability classes and how they're exploited, able to port exploits from x86 to x64 arch (if possible) and be familiar with newer mitigations in Windows 10.

## What's kernel debugging?

Unlike user-mode debugging where you can pause execution of a single process, kernel-mode debugging breaks on the entire system, meaning you won't be able to use it at all. A debugger machine is needed so you can communicate with the debuggee, observe memory or kernel data structures, or catch a crash.

Reading material:

- User mode and kernel mode
- Understanding User and Kernel Mode

## What's kernel exploitation?

Something more fun than user-mode exploitation ;)

The main goal is to gain execution with kernel-mode context. A successful exploit could result in elevated permissions and what you can do is only bound by your imagination (anywhere from cool homebrew to APT-sponsored malware).

Goal for this tutorial is getting a shell with SYSTEM permissions.

## How will this tutorial be organized?

- Part 1: Setting up the environment
  - Configure the 3 VMs + debuggee machine.
  - Configure WinDBG.
- Part 2: Payloads
  - Placeholder for common payloads to be used later, this will allow us to focus on vulnerability-specific details in future posts and refer to this post when needed.
- Part 3-N:
  - One or more post per vulnerability.

## Kernel exploit development lifecycle

1. **Finding a vulnerability**: This won't be covered in this tutorial as we already know exactly where the vulnerabilities are.
2. **Hijacking execution flow**: Some vulnerabilities allow code execution, others require more than that.
3. **Privilege escalation**: Main goal will be to get a shell with SYSTEM privileges.
4. **Restore execution flow**: Uncaught exceptions in kernel-mode result in a system crash. Unless a DoS exploit makes you sleep at night we need to address this.

## What are the targets?

Exploitation will be attempted on the following targets (no specific version is needed yet):

1. a Win7 x86 VM
2. a Win7 x64 VM
3. a Win10 x64 VM

Normally we'll start with the x86 machine, followed by porting it to the Win7 x64 one. Some exploits won't run on the Win10 machine due to some newer mitigations that are added. We'll either have to tweak the exploit or come up with an entirely different approach.

## What software do I need?

- Hyper-visor software, lots of options.
- Windows 7 x86 VM
- Windows 7 x64 VM
- Windows 10 x64 VM
- VirtualKD
- OSR Driver Loader

---

## Setting up the debuggee machines

The debuggee is our guinea pig. We'll use it to load the vulnerable driver and communicate with it. This machine will crash a lot as most exceptions in kernel will result in BSOD. Make sure you give it enough RAM.

Per debuggee:

1. Inside the VirtualKD folder, run `target\vminstall.exe`. This will add a boot entry that has debugging enabled and connects automatically to the VirtualKD server on the debugger machine.
   For the Windows 10 VM, you need to enable test signing. This allows you to load unsigned drivers into the kernel.
   Running `bcdedit /set testsinging on` and rebooting will show "Test Mode" on the desktop.

   *NOTE: Windows 10 supports communicating through the network and in my experience is usually faster. To do that, follow this.*

2. Run the OSR Driver Loader, register the service then start it. You may need to reboot.

3. Optional: Install VM guest additions.

4. Set up a low-priv account, this should be used while exploiting.

   ```
   C:\Windows\system32>net user low low /add
   The command completed successfully.
   ```

---

## Setting up the debugger machine

This machine will be the one debugging the debuggee machine through WinDBG. You'll be able to inspect memory and data structures and manipulate them if needed. Having a remote debugging session running when the debuggee crashes allows us to break into the VM and/or analyze a crash.

VirtualKD host will automatically communicate with a named pipe instead of setting it up manually. If you're network debugging the Win10 VM, you'll need to test the connection manually.

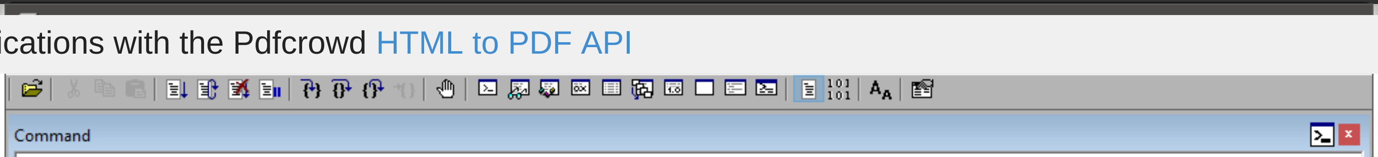
1. Install the Windows SDK (link). You can select the "Debugging Tools for Windows" only.
2. Verify that WinDBG is installed, Win10 SDK is by default installed in `C:\Program Files (x86)\Windows Kits\10\Debuggers`.
   Add it to the system path and set up the debugger path in VirtualKD.

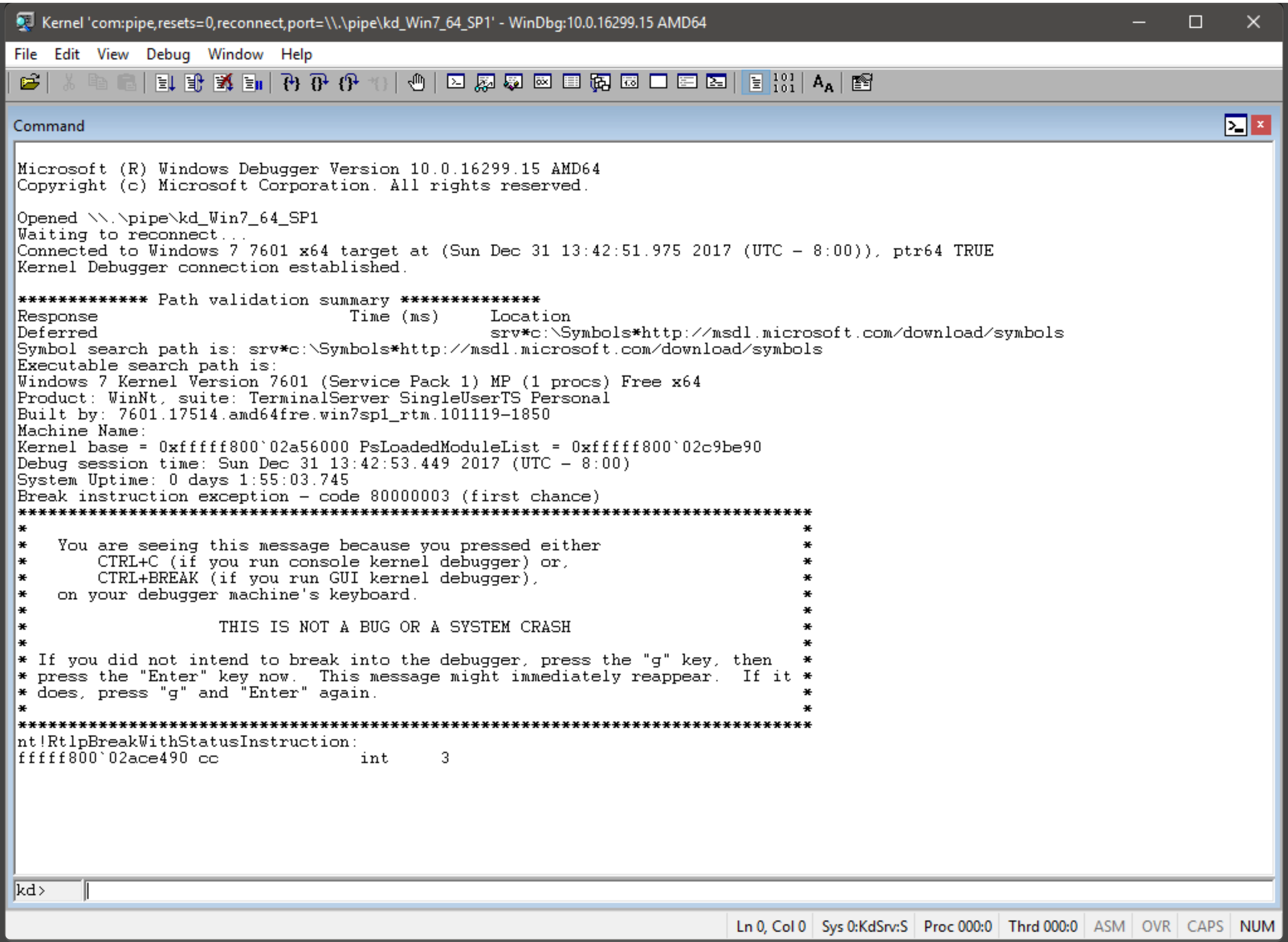
Reboot one of the debuggee machines while the VirtualKD host is running on the debugger. You should be able to start a WinDBG session.

---

## Setting up WinDBG

If everything is set up correctly, WinDBG will pause execution and print some info about the debuggee.




Kernel 'com:pipe,resets=0,reconnect,port=\\.\pipe\kd_Win7_64_SP1' - WinDbg:10.0.16299.15 AMD64
File Edit View Debug Window Help
Command
Ln 0, Col 0 Sys 0:KdSrv:S Proc 000:0 Thrd 000:0 ASM OVR CAPS NUM

```
Microsoft (R) Windows Debugger Version 10.0.16299.15 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.

Opened \\.\pipe\kd_Win7_64_SP1
Waiting to reconnect...
Connected to Windows 7 7601 x64 target at (Sun Dec 31 13:42:51.975 2017 (UTC - 8:00)), ptr64 TRUE
Kernel Debugger connection established.

************* Path validation summary **************
Response                         Time (ms)     Location
Deferred                                       srv*c:\Symbols*http://msdl.microsoft.com/download/symbols
Symbol search path is: srv*c:\Symbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
Windows 7 Kernel Version 7601 (Service Pack 1) MP (1 procs) Free x64
Product: WinNt, suite: TerminalServer SingleUserTS Personal
Built by: 7601.17514.amd64fre.win7sp1_rtm.101119-1850
Machine Name:
Kernel base = 0xfffff800`02a56000 PsLoadedModuleList = 0xfffff800`02c9be90
Debug session time: Sun Dec 31 13:42:53.449 2017 (UTC - 8:00)
System Uptime: 0 days 1:55:03.745
Break instruction exception - code 80000003 (first chance)
*******************************************************************************
*                                                                             *
*   You are seeing this message because you pressed either                    *
*       CTRL+C (if you run console kernel debugger) or,                       *
*       CTRL+BREAK (if you run GUI kernel debugger),                          *
*   on your debugger machine's keyboard.                                      *
*                                                                             *
*                   THIS IS NOT A BUG OR A SYSTEM CRASH                       *
*                                                                             *
* If you did not intend to break into the debugger, press the "g" key, then   *
* press the "Enter" key now.  This message might immediately reappear.  If it *
* does, press "g" and "Enter" again.                                          *
*                                                                             *
*******************************************************************************
nt!RtlpBreakWithStatusInstruction:
fffff800`02ace490 cc              int     3
```

```
kd>
```

Symbols contain debugging information for lots of Windows binaries. We can get them by executing the following:

```
.sympath srv*c:\Symbols*http://msdl.microsoft.com/download/symbols;C:\HEVD
.reload /f *.*
```

Enable verbose debugging:

```
ed nt!Kd_Default_Mask 0xf
```

You should be able to find the HEVD module loaded:

```
kd> lm m HEVD
Browse full module list
start             end                 module name
fffff80b`92b50000 fffff80b`92b59000   HEVD       (deferred)
```

Save the workspace profile and any environment changes you made by selecting

```
File -> Save Workspace to File
```

This is a handy reference to commands used throughout the series.

Type `g` or hit F5 to continue execution.

---

## HEVD Driver crash course

- `DriverEntry` is the entry point of any driver.

```
NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject, IN PUNICODE_STRING RegistryPath)
    UINT32 i = 0;
    PDEVICE_OBJECT DeviceObject = NULL;
    NTSTATUS Status = STATUS_UNSUCCESSFUL;
    UNICODE_STRING DeviceName, DosDeviceName = {0};

    UNREFERENCED_PARAMETER(RegistryPath);
    PAGED_CODE();

    RtlInitUnicodeString(&DeviceName, L"\\Device\\HackSysExtremeVulnerableDriver");
    RtlInitUnicodeString(&DosDeviceName, L"\\DosDevices\\HackSysExtremeVulnerableDrive
    // Create the device
    Status = IoCreateDevice(DriverObject,
                            0,
                            &DeviceName,
                            FILE_DEVICE_UNKNOWN,
                            FILE_DEVICE_SECURE_OPEN,
                            FALSE,
                            &DeviceObject);

    ...
}
```

- This routine contains an `IoCreateDevice` call that contains the driver name we'll be using to communicate with it.
- `DriverObject` will get populated with necesarry structures and function pointers.
- What matters to us for HEVD is the function pointer assigned to `DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL]` which is routine responsible for handling IOCTLs.
- In HEVD, this function is called `IrpDeviceIoCtlHandler` and it's basically a large switch case for every IOCTL. Ever vulnerability has a unique IOCTL.

Example: `HACKSYS_EVD_IOCTL_STACK_OVERFLOW` is the IOCTL used to trigger the stack overflow vulnerability.

---

That's it! Next post will discuss payloads. For now, it'll only include a generic token-stealing payload that'll be used for part 3.

---

*I'm aware this post doesn't cover lots of issues you can run into. Due to the scope of this tutorial, you'll need to figure out a lot on your own, but you're welcome to comment with your questions if needed.*

*-Abatchy*

0 Comments abatchy17.github.io

1 Login

♡ Recommend Share

Sort by Best

Start the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Be the first to comment.

ALSO ON **ABATCHY17.GITHUB.IO**

**How to prepare for PWK/OSCP, a noob-friendly guide**

4 comments • 9 months ago

**Navneet Soni** — why do we need to use recon-ng tool during the oscp exam?

**abatchy's blog | OSCP: Day 1**

1 comment • 9 months ago

**Mike Ship** — Did you start using MSF out of the gate or did you use other exploites? At what point if any did …

**[Kernel Exploitation] 6: NULL pointer dereference**

1 comment • 4 months ago

**Kotori** — It does not works on Windows 7 SP1, 64 bit Ultimate. NtAllocateVirtualMemory returns …

**Shellcode reduction tips (x86)**

1 comment • 9 months ago

**Cami Rodriguez** — Nice, very useful, thanks

Subscribe   Add Disqus to your site   Disqus' Privacy Policy   DISQUS

Follow @abatchy17

Follow @abatchy17

## Categories

- .Net Reversing
- Backdooring
- DefCamp CTF Qualifications 2017
- Exploit Development
- Kernel Exploitation
- Kioptrix series
- Networking
- OSCE Prep
- OSCP Prep
- OverTheWire - Bandit
- OverTheWire - Leviathan

- OverTheWire - Natas
- Powershell
- Programming
- Pwnable.kr
- SLAE
- Shellcoding
- Vulnhub Walkthrough
- rant

## Blog Archive

### January 2018

- [Kernel Exploitation] 7: Arbitrary Overwrite (Win7 x86)
- [Kernel Exploitation] 6: NULL pointer dereference
- [Kernel Exploitation] 5: Integer Overflow
- [Kernel Exploitation] 4: Stack Buffer Overflow (SMEP Bypass)
- [Kernel Exploitation] 3: Stack Buffer Overflow (Windows 7 x86/x64)
- [Kernel Exploitation] 2: Payloads
- [Kernel Exploitation] 1: Setting up the environment

### October 2017

- [DefCamp CTF Qualification 2017] Don't net, kids! (Revexp 400)
- [DefCamp CTF Qualification 2017] Buggy Bot (Misc 400)

### September 2017

- [Pwnable.kr] Toddler's Bottle: flag
- [Pwnable.kr] Toddler's Bottle: fd, collision, bof
- OverTheWire: Leviathan Walkthrough

### August 2017

- [Rant] Is this blog dead?

### June 2017

- Exploit Dev 101: Bypassing ASLR on Windows

### May 2017

- Exploit Dev 101: Jumping to Shellcode
- Introduction to Manual Backdooring
- Linux/x86 - Disable ASLR Shellcode (71 bytes)
- Analyzing Metasploit linux/x86/shell_bind_tcp_random_port module using Libemu
- Analyzing Metasploit linux/x86/exec module using Ndisasm
- Linux/x86 - Code Polymorphism examples
- Analyzing Metasploit linux/x86/adduser module using GDB
- Analyzing Metasploit linux/x86/adduser module using GDB
- ROT-N Shellcode Encoder/Generator (Linux x86)
- Skape's Egg Hunter (null-free/Linux x86)
- TCP Bind Shell in Assembly (null-free/Linux x86)

### April 2017

- Shellcode reduction tips (x86)

### March 2017

- LTR Scene 1 Walthrough (Vulnhub)
- Moria v1.1: A Boot2Root VM
- OSCE Study Plan
- Powershell Download File One-Liners
- How to prepare for PWK/OSCP, a noob-friendly guide

### February 2017

- OSCP-like Vulnhub VMs
- OSCP: Day 30
- Mr Robot Walkthrough (Vulnhub)

### January 2017

- OSCP: Day 6
- OSCP: Day 1

- Port forwarding: A practical hands-on guide
- Kioptrix 2014 (#5) Walkthrough
- Wallaby's Nightmare Walkthrough (Vulnhub)

December 2016

- Kiopritx 1.3 (#4) Walkthrough (Vulnhub)
- Kioptrix 3 Walkthrough (Vulnhub)
- Kioptrix 2 Walkthrough (Vulnhub)
- OverTheWire: Natas 17

November 2016

- OverTheWire: Natas 16
- OverTheWire: Natas 14 and 15
- Kioptrix 1 Walkthrough (Vulnhub)
- PwnLab: init Walkthrough (Vulnhub)
- OverTheWire: Natas 12
- OverTheWire: Natas 11

October 2016

- Vulnix Walthrough (Vulnhub)
- OverTheWire: Natas 6-10
- OverTheWire: Natas 0-5
- OverTheWire: Bandit 21-26
- OverTheWire: Bandit 16-20
- OverTheWire: Bandit 11-15
- OverTheWire: Bandit 6-10
- OverTheWire: Bandit 0-5
- Introduction