Cr4sh / ioctlfuzzer

Automatically exported from code.google.com/p/ioctlfuzzer

7 commits | 3 branches | 0 releases | 1 contributor

Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

Cr4sh README.TXT for version 1.3 — Latest commit 7e84fa7 on 13 Dec 2011

bin	Commit of 1.3 version, see CHANGELOG.TXT	6 years ago
src	Commit of 1.3 version, see CHANGELOG.TXT	6 years ago
CHANGELOG.TXT	README.TXT for version 1.3	6 years ago
README.TXT	README.TXT for version 1.3	6 years ago
attack-surface-analysis_NT6.1_x86.log	Commit of 1.3 version, see CHANGELOG.TXT	6 years ago

README.TXT

```
   IOCTL Fuzzer ver. 1.3
   http://code.google.com/p/ioctlfuzzer/

   (c) 2009-2011 eSage lab
   http://www.esagelab.com/
   dmitry@esagelab.com


==============================================
   Overview
==============================================

 IOCTL Fuzzer is a tool designed to automate the task of searching vulnerabilities in Windows kernel
drivers by performing fuzz tests on them.

Current OS support (x32 and x64): Windows XP, Vista, 2003 Server, 2008 Server, 7.

 The fuzzer’s own driver hooks nt!NtDeviceIoControlFile() in order to take control of all IOCTL requests
throughout the system.

 While processing IOCTLs, the fuzzer will spoof those IOCTLs conforming to conditions specified in the
configuration file. A spoofed IOCTL is identical to the original IRP in all respects except the input data,
which is changed to randomly generated fuzz.

 Besides the fuzzing functionality, monitoring mode is also available with the tool. The monitoring mode
allows logging of IRPs, optionally including their HEX dumps, into a file and/or a console.

 Also, since 1.2 version exceptions monitoring feature is available, that can be usefull while fuzzing not
only a kernel drivers. Exception monitoring is working through unexported function nt!KiDispatchException()
pathing, which address obtained from Windows kernel debug symbols (they are automatically downloading from
Microsoft's PDB server, during fuzzer initialization).

 Specific IOCTLs which are to be logged or fuzzed by the tool are defined in the XML configuration file.
IOCTLs may be filtered by the following parameters:

 * Path to executable file corresponding to a process from which an IOCTL request is sent.
 * IOCTL destination device name.
 * IOCTL destination driver name.
 * IOCTL Control Code.

 Since 1.3 version IOCTL Fuzzer dumps _all_ catched IOCTLs information in text file
%SystemDrive%\ioctls.log
 Information from this file can be used for calculating requests count for each device/driver during the
```

```
attack surface analysis (see below).


==============================================
  Command line options
==============================================

--config <path> - Specify path to the fuzzer XML configuration file. For more information about
configuration file format see example in bin/ioctlfuzzer.xml. If '--config' option is not specified -
application will start in IOCTLs monitoring mode.

--boot - Boot time fuzzing/monitoring. This option will run fuzzer with the next system reboot.

--exceptions - Enable exceptions monitoring. Notice: files dbgeng.dll, dbghelp.dll and symsrv.dll are
required for exceptions monitoring and must be placed into the same directory, as IOCTL Fuzzer executable.

--noioctls - Disable IOCTLs monitoring, show exceptions only (this option valid if '--exceptions' has been
specified).

--uninstall - Uninstall IOCTL Fuzzer kernel driver and exit.

--analyze - Attack surface analysis feature: prints list of all drivers, devices and their properties
(security settings, number of catched IOCTL requests, driver file product/vendor information, opened
handles for devices, etc.), see log file example in attack-surface-analysis_NT6.1_x86.log.

--loadlog <path> - Load catched IOCTLs information for attack surface analysis from external log file. This
option can be used only with '--analyze', as '<path>' parameter you can specify path to the catched IOCTLs
log file (%SystemDrive%\ioctls.log), which is created automatically when IOCTL Fuzzer runs in IOCTLs
fuzzing or monitoring mode.


Typical usage example (run IOCTL Fuzzer with XML config and enable exceptions monitoring):

 > ioctlfuzzer.exe --config ioctlfuzzer.xml --exceptions


==============================================
  Using the fuzzer
==============================================

General algorithm for fuzz-testing an application is as follows.

1. Install target application onto a virtual machine.

2. Attach a remote debugger to the virtual machine. Notice: how to configure WinDbg remote connection to
VMware: http://silverstr.ufies.org/lotr0/windbg-vmware.html.

3. Run IOCTL Fuzzer in fuzzing mode on the guest OS.

4. Play around with target application unless an unhandled exception is displayed in the remotely attached
debugger. Notice: normally, i.e. with no debugger attached, an unhandled exception will provoke a BSOD.

5. Release code execution on the virtual machine (F5 in WinDbg) to allow guest OS generate a crash dump.

6. Analyze the crash dump. IOCTL which provoked the unhandled exception should be found at this step.

7. If necessary, manual analysis of the application binary code may be performed.


==============================================
  Using the attack surface analysis feature
==============================================

Typical attack surface analysis usage scenario:

1. Enable boot time monitoring to collect information about the most frequently-used IOCTL requests:
   > ioctlfuzzer.exe --config ioctlfuzzer.xml --boot

2. Reboot the box.

3. After reboot run attack surface analysis and pass to the IOCTL Fuzzer path of the log file, with all of
the collected IOCTLs information:

   > ioctlfuzzer.exe --analyze --loadlog %SystemDrive%\ioctls.log.


==============================================
  Using the fuzzer with
  Kernel Debugger Communication Engine
==============================================

Integration with Kernel Debugger Communicatioin Engine allows the IOCTL Fuzzer to execute any commands in
remote kernel debugger for IOCTL requests parameters, which were specified in the XML configuration file.


1. Before running fuzzer you need to load Kernel Debugger Communicatioin Engine extension in your remote
kernel debugger by executing command ".load \path\to\ioctlfuzzer\binaries\dbgcb.dll".


2. Edit "dbgcb" nodes list in XML configuration file. For example, if you want to run "kb 40" command for
each IOCTL request, that executed from the net.exe process context:

<dbgcb>
  <process val="net.exe"><![CDATA[kb 40]]></process>
</dbgcb>

Also, you can use "device" or "driver" nodes, for specifying device/driver name, and "ioctl" for I/O
Control Code value:

<dbgcb>
  <device val="\Device\Ndis"><![CDATA[kb 40]]></device>
  <driver val="\Driver\NDIS"><![CDATA[kb 40]]></driver>
  <ioctl val="0x00170014"><![CDATA[kb 40]]></ioctl>
</dbgcb>

If command value is not specified -- fuzzer just breaks into debugger when catching request with the
appropriate parameters:

<dbgcb>
  <process val="net.exe" />
</dbgcb>


3. Run fuzzer with XML configuration file:

   > ioctlfuzzer.exe --config ioctlfuzzer.xml


Note: Kernel Debugger Communication engine uses breakpoints for interaction between debugger extension and
target system (https://github.com/Cr4sh/DbgCb/blob/master/dbgcb_scheme.png), so, you can use VirtualKD
(http://virtualkd.sysprogs.org/) for better performance.


==============================================
  Building from sources
==============================================

1. Download and install Windows Driver Kit Version 7.1.0
http://www.microsoft.com/downloads/en/details.aspx?displaylang=en&FamilyID=36a2630f-5d56-43b5-b996-
7633f2ec14ff

2. Run Windows Server 2003 x86 (or x64, to build 64-bit version) Free Build Environment (Start -> «Windows
Driver Kits» -> «WDK 7600.16385.1» ->
```

«Build Environments» -> «Windows Server 2003» -> «x86 Free Build Environment»).

3. Go to the directory ./src/ and execute .\build.bat (or .\build64.bat, to build 64-bit version) from the Build Environment.