Effective Bug Discovery

Kernel-Mode Coverage Analysis

---

vf
vf@nologin.org

Last modified: 9/23/2006

If we knew what it was we were doing, it would not be called research, would it?
Albert Einstein

Contents

Chapter 1

Foreword

Abstract: Sophisticated methods are currently being developed and implemented for mitigating the risk of exploitable bugs. The process of researching and discovering these vulnerabilities in modern code will, in time, become less extensive. Therefore, research technologies are changing to accommodate the shift in vulnerability discovery. Code coverage analysis implemented in conjunction with fuzz testing reveals faults within a binary file that would have otherwise remained undiscovered by either method alone. This paper suggests a research method for more effective runtime binary analysis using the aforementioned strategy. This study presents empirical evidence that despite the fact that bug detection will become increasingly difficult in the future, analysis techniques have an opportunity to evolve intelligently.

Prerequisites: For an in-depth understanding of the concepts presented in this paper, a familiarity with Microsoft Windows device drivers, working with x86 assembler, debugging fundamentals, and the Windows kernel debugger is required. A brief introduction to the current state of code coverage analysis, including related uses, is introduced to support information presented within this paper. However, to implement the practices within this paper a deeper knowledge of aforementioned vulnerability discovery methods and methodolo-

gies are required. The following software and knowledge of its use is required to follow along with the discussion: IDAPro, Debugging tools for Windows, Debug Stalk, and a virtual machine such as VMware or Virtual PC.

Thanks: The author would like to thank west, icer, skape, Uninformed, and mom.

Chapter 2

Introduction

2.1 The status of vulnerability research

Researchers employ a myriad of investigative techniques in the quest for application vulnerabilities. In any case, there exists no silver bullet for the discovery of security related software bugs, not to mention the fact that several new security oriented kernel-mode components have recently been integrated into Microsoft operating systems. Although these systems are never failsafe, they do help to make comprehensive vulnerability investigations more difficult to perform. Vista, particularly on the 64-bit edition, is integrating several mechanisms including driver signing, Secure Bootup using a TPM hardware chip, PatchGuard, kernel-mode integrity checks, and restricted user-mode access to `\Device\PhysicalMemory`. The Vista kernel also has an improved Low Fragmentation Heap and Address Space Layout Randomization. In later days, bugs were revealed via dumb fuzzing techniques, whereas this year more complicated bugs are indicating that knowledge of the format would require advanced understanding of a parser. Because of this, researchers are moving towards different discovery methods such as intelligent, rather than dumb, testing of drivers and applications.

2.2 The problem with fuzzing

To compound the conception that these environments are becoming more difficult to test, monolithic black box fuzz testing, while frequently efficacious in its purpose, has a tendency for a exhibiting a lack of potency. The term "monolithic" is included as a reference to a comprehensive execution of the entire application or driver. Fuzzing is often executed in an environment where

the tester does not know the internals of the binary in question. This leads to disadvantages in which a large number of tests must be executed to get an accurate estimate of binary's reliability. This investigation can be a daunting task if not implemented in a constructive manner. The test program and data selection should ensure independence from unrelated tests or groups of tests, thereby gaining the ability of complete coverage by reducing dependency on specific variables and their decision branching.

Another disadvantage of monolithic black box fuzz testing is that it is difficult to provide coverage analysis even though the testing selection may cover the entire suite of security testing models. A further complication in this nature of testing is of cyclic dependency causing cyclic arguments which in turn leads to a lessening of coverage assurance.

2.3 Expectations

This paper aims to educate the reader on the espousal of code coverage analysis and fuzzing philosophy presented to researchers as a means to lighten the burden of bug detection. A kernel mode device driver will be fuzzed for bugs using a standard fuzzing method. Results from the initial fuzzing test will be examined to determine coverage. The fuzz testing method will be revised to accommodate coverage concerns and an execution graph is generated to view the results of the previous testing. A comparison is then made between the two prior testing methods, proving how effective code coverage analysis through kernel mode Stalking can improve fuzzing endeavors.

Chapter 3

Q&A

Before understanding how the methods and methodologies presented in this paper can be used, a few simple definitions and descriptions are addressed for the benefit of the reader.

3.1 What is code coverage?

Code coverage analysis, as represented by a *Control Flow Graph* (CFG), is defined as an exhaustive execution of all possible paths through code and data flow that may be relevant for revealing failures. It is used as a good metric in determining how a specific set of tests can uncover numerous faults. Techniques of proper code coverage analysis presented in this paper utilize basic mathematical properties of graph theory by including elements such as vertices, links and edges. Graph theory has lain somewhat dormant until recently being utilized by computer scientists which have subsequently defined their own sets of vocabulary for the subject. For the sake of research continuity and to link mathematical to computer science definitions, the verbiage used within this paper will equate vertices to code blocks, branches to decisions, and edges to code paths.

To support our hypothesis, the aforementioned graph theory elements are compiled into CFGs. Informally, a Control Flow Graph is directed graph composed of a finite set of vertices connected by edges indicating all possible routes a driver or application may take during execution. In other words, a CFG is merely blocks of code whose connected flow paths are determined by decisions. Block execution consists of a sequence of instructions which are free of branching or other control transfers except for the last instruction. These include branches or decisions which consist of Boolean expressions in a control structure. A path

is a sequence of nodes traveled through by a series of uninterrupted links. Paths enable flow of information or data through code. In our case, a path is an execution flow and is therefore essential to measuring code coverage. Because of this factor, this investigation focuses directly on determining which paths have been traversed, which blocks and correlating data have been executed, and which links have been followed and finally applying it to fuzzing techniques.

The purpose of code coverage analysis is ultimately to require all control decisions to be transferred. In other words, the application needs to be executed thoroughly using enough inputs that all edges in the graph are traversed at least once. These graphs will be represented as diagrams in which blocks are squares, edges are lines, and paths are colored.

Chapter 4

Hypothesis: Code Coverage and Fuzzing

In the security hemisphere, fuzzing has traditionally manifested potential security holes by throwing random garbage at a target, hoping that any given code path will fail in the process of consuming the aforementioned data. The possibility of execution flowing through a particular block in code is the sum of probabilities of the conditional branches leading to blocks. In simplicity, if there are areas of code that are never executed during typical fuzz testing, then administering code coverage methodologies will reveal those unexecuted branches. Graphical code coverage analysis using CFGs helps determine which code path has been executed even without the use of symbol tables. This process allows the tester to more easily identify branch execution, and to subsequently design fuzz testing methods to properly attain complete code coverage. Prior experiments driven at determining the effectiveness of code coverage techniques identify that ensuring branch execution coverage will improve the likelihood of discovery of binary faults.

4.1 Process and Kernel Stalking

One of the more difficult questions to answer when testing software for vulnerabilities is: "when is the testing considered finished?" How do we, as vulnerability bug hunters, know when we have completed our testing cycle by exhausting all code paths and discovering all possible bugs? Because fuzz testing can easily be random, so unpredictable, the question of when to conclude testing is often left incomplete.

Pedram Amini, who recently released "Paimei", coined the term "Process Stalk-

ing" as a set of runtime binary analysis tools intended to enhance the visual effect of runtime analysis. His tool includes an IDA Pro plug-in paired with GML graph files for easy viewing. His strategy amalgamates the processes of runtime profiling, which is the act of gathering information about application data in a development scenario. Tracing, which used often in debugging, is the automatic discovery of nodes based on execution paths, and state mapping which is a graphic model composed of behavior states of a binary. In the past, Process Stalker has been used to aid in reverse engineering Microsoft Update patches. Further implementations of the "Process Stalker" tool suite can be found on Pedram Amini's site (http://pedram.redhive.com) and on the reverse engineering site (http://www.openrce.org).

4.2 Stalking and Fuzzing Go Hand in Hand

Process Stalker was transformed by an individual into a windbg extension for use in debugging user-mode and kernel-mode scenarios. This tool was given the title "Debug Stalk," and until now this tool has remained unreleased. Process and Debug Stalker have overcome the static analysis visualization setback by implementing runtime binary analysis. Runtime analysis using Process and Debug Stalking in conjunction with mathematically enhanced CFGs exponentially improves the bug hunting mechanisms using fuzz techniques. Users can graphically determine via runtime analysis which paths have not been traversed and which blocks have not been executed. The user then has the opportunity to refine their testing approach to one that is more effective. When testing a large application, this technique dramatically reduces the overall workload of said scenarios. Therefore, iterations of the Process Stalk tool and the Debug Stalk tool will be used for investigating a faulty driver in this paper.

Debug Stalk is a Windows debugger plug-in that can be used in places where Process Stalking may not be suited, such as in a kernel-mode setting.

Chapter 5

Implementation

For the mere sake of simple illustration, several tools have been created for testing our code coverage theories. Some of the test cases have been exaggerated and are not real world examples. This testing implementation is broken down into three parts: Part I includes sending garbage to the device driver with dumb fuzzing; Part II will include smarter fuzzing; Part III is a breakdown of how an intelligent level of fuzzing helps improve code coverage while testing. First, a very simple device driver named pluto.sys was created for the purpose of this paper. It contains several blocks of code with decision based branching that will be fuzzed. The fuzzer will send random iterations of data to pluto.sys. After fuzzing has completed, a post-analysis tool will review executed code blocks within the driver. Part II will contain the same process as Part I, however, it will include an updated fuzzer based on our Part I post-analysis that will allow the driver to call into a previously unexecuted code region. Part III uses the data collected in Parts I and II as illustrative example of a proof of a beneficiary code coverage thesis.

5.1 Stalking Setup

Several software components need to be acquired before Stalking can begin: the Debug Stalk extension, Pedram's Process Stalker, Python, and the GoVisual Diagram Editor (GDE). Pedram's Stalker is listed on both his blog and on the OpenRCE website. The Process Stalker contains files such as the IDA Pro plug-in, and Python scripts that generate the GML graph files that will be imported into GDE. GDE provides a functional mechanism for editing and positioning of graphs including clustered graphing, creation and deletion of nodes, zooming and scrolling, automatic graph layout. Components can be obtained at the following locations:

GDE: http://www.oreas.com/gde_en.php
Python: http://www.python.org/download
Proc Stalker: http://www.openrce.org/downloads/details/171/ProcessStalker
Debug Stalk: http://www.nologin.org/code

5.2 Installing the Stalker

A walkthrough of installation for Process Stalker and required components will be covered briefly in this document, however, more detailed steps and descriptions are provided in Pedram's supporting manual. The .bpl file generated by the IDA plug-in will spit out a breakpoint list for entries within each block. The IDA plug-in process_stalker.plw must be inserted into the IDA Pro plug-ins directory. Restarting IDA will allow the application to load the plug-in. A successful installation of the IDA plug-in in the log window will be similar to the following:

```
[*] pStalker> Process Stalker  Profiler
[*] pStalker> Pedram Amini <pedram.amini@gmail.com>
[*] pStalker > Compiled on Sep 21 2006
```

Generating a .bpl file can be started by pressing Alt+5 within the IDA application. A dialog appears. Make sure that "Enable Instruction Colors," "Enable Comments," and "Allow Self Loops" are all selected. Pressing OK will prompt for a "Save as" dialog. The .bpl file must be named relative to its given name. For example, if calc.exe is being watched, the file name must be calc.exe.bpl. In our case, pluto.sys is being watched, so the file name must be pluto.sys.bpl. A successful generation of a .bpl file will produce the following output in the log window:

```
[*] pStalker> Profile analysis 25% complete.
[*] pStalker> Profile analysis 50% complete.
[*] pStalker> Profile analysis 7% complete.
[*] pStalker> Profile analysis 100% complete.
```

Opening the pluto.sys.bpl file will show that records are colon delimited:

```
pluto.sys:0000002e:0000002e
pluto.sys:0000006a:0000006a
pluto.sys:0000007c:0000007c
```

5.3 Installing Debug Stalk

The Debug Stalk extension can be built as follows. Open the Windows 2003 Server Build Environment window. Set the DBGSDK_INC_PATH and DBGSDK_LIB_PATH environment variables to specify the paths to the debugger SDK headers and the debugger SDK libraries, respectively. If the SDK is installed at `c:\WINDBGSDK`, the following would work:

```
set DBGSDK_INC_PATH=c:\WINDBGSDK\inc
set DBGSDK_LIB_PATH=c:\WINDBGSDK\lib
```

This may vary depending on where the SDK is installed. The directory name must not contain a space (' ') in its path. The next step is to change directories to the project directory. If Debug Stalk source code is placed within the samples directory within the SDK (located at `c:\\WINDBGSDK`), then the following should work:

```
cd c:\WINDBGSDK\samples\dbgstalk-0.0.18
```

Typing `build -cg` at the command line to build the Debug Stalk project. Copy the dbgstalk.dll module from within this distribution to the root folder of the Debugging Tools for Windows root directory. This is the folder containing programs like cdb.exe and windbg.exe. If you have a default installation of "Debugging tools for Windows" already installed, the following should work:

```
copy dbgstalk.dll "c:\Program Files\Debugging Tools for Windows\"
```

The debugger plug-in should be installed at this point. It is important to note that Debug Stalk is a fairly new tool and has some reliability issues. It is a bit flakey and some hacking may be necessary in order to get it running properly.

5.4 Stalking with Kernel Debug

5.4.1 Part I

For testing purposes, a Microsoft Operating System needs to be set up inside of a Virtual PC environment. Load the pluto.sys driver inside of the Virtual PC and attach a debug session via Kernel Debug (kd). Once kd is loaded and attached to a process within the Virtual Machine, Debug Stalk can be invoked by calling "!dbgstalk.dbgstalk [switches] [.bpl file path]" at the kd console. For example:

```
C:\Uninformed>kd -k com:port=\\.\pipe\woo,pipe

Microsoft (R) Windows Debugger Version 6.6.0007.5
Copyright (c) Microsoft Corporation. All rights reserved.

Opened \\.\pipe\woo
Waiting to reconnect...
Connected to Windows XP 2600 x86 compatible target, ptr64 FALSE
Kernel Debugger connection established.
Windows XP Kernel Version 2600 (Service Pack 2) UP Free x86 compatible
Product: WinNt, suite: TerminalServer SingleUserTS
Built by: 2600.xpsp_sp2_rtm.040803-2158
Kernel base = 0x804d7000 PsLoadedModuleList = 0x8055ab20
Debug session time: Sat Sep 23 14:40:24.522 2006 (GMT-7)
System Uptime: 0 days 0:06:50.610
Break instruction exception - code 80000003 (first chance)
nt!DbgBreakPointWithStatus+0x4:
804e3b25 cc              int     3
kd> .reload
Connected to Windows XP 2600 x86 compatible target, ptr64 FALSE
Loading Kernel Symbols
...................................................
Loading User Symbols

Loading unloaded module list
...........
kd> !dbgstalk.dbgstalk -o -b c:\Uninformed\pluto.sys.bpl
[*] - Entering Stalker
[*] - Break Point List.....: c:\Uninformed\pluto.sys.bpl
[*] - Breakpoint Restore...: OFF
[*] - Register Enumerate...: ON
[*] - Kernel Stalking:.....: ON

current context:

eax=00000001 ebx=ffdff980 ecx=8055192c edx=000003f8 esi=00000000 edi=f4be2de0
eip=804e3b25 esp=80550830 ebp=80550840 iopl=0         nv up ei pl nz na po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000202
nt!RtlpBreakWithStatusInstruction:
804e3b25 cc              int     3

commands:

        [m] module list         [0-9] enter recorder modes
        [x] stop recording      [v] toggle verbosity
        [q] quit/close
```

Once Debug Stalk is loaded, a list of commands is available to the user. A breakdown of the command line options offered by Debug Stalk is as follows:

```
[m]    module list
[0-9]  enter recorder modes
[x]    stop recording
[v]    toggle verbosity
[q]    quit/close
```

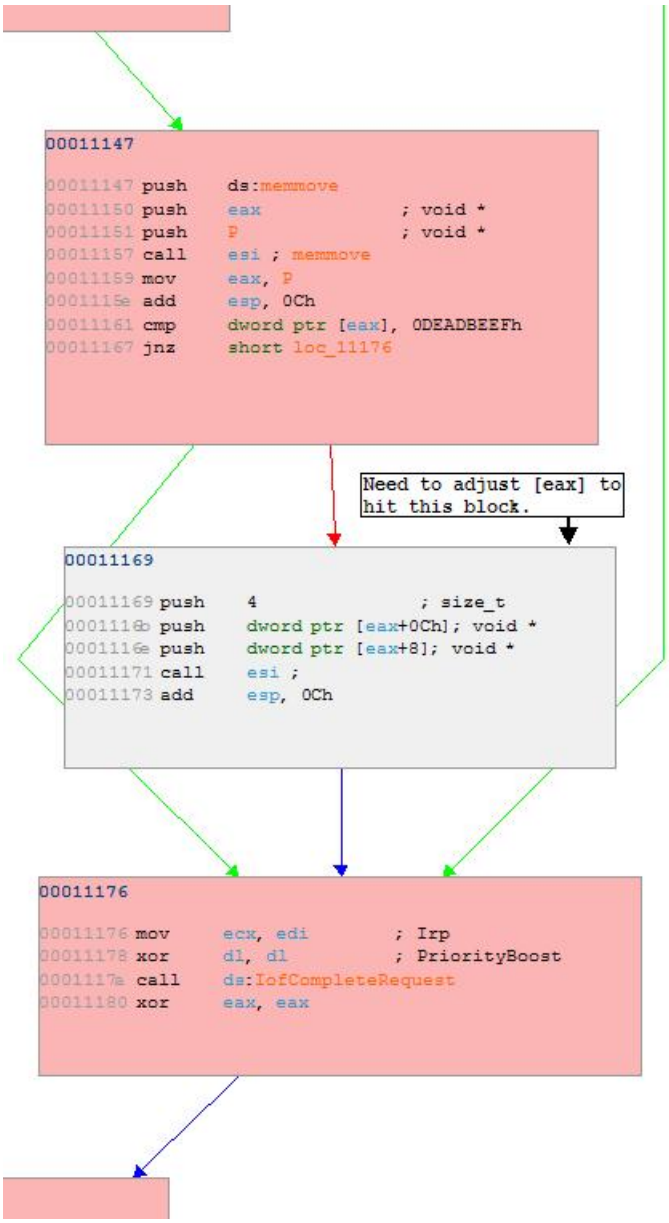
At this point, the fuzz tool needs to be executed to send random arbitrary data to the device driver. While the fuzzer is running, Debug Stalk will print out information to kd. Pressing 'g' at the command line prompt will resume execution of the target machine. This invocation will look something like this:

```
kd> g
[*] - Recorder Opened......: pluto.sys.0
[*] - Recorder Opened......: pluto.sys-regs.0
Modload: Processing breakpoints for module pluto.sys at f7a7f000
Modload: Done. 46 of 46 breakpoints were set.
0034c883 T:00000001 [bp] f7a83000 a10020a8f7      mov     eax,dword ptr [pluto+0x3000 (f7a82000)]
0034ed70 T:00000001 [bp] f7a8300e 3bc1            cmp     eax,ecx
0034eded T:00000001 [bp] f7a83012 a12810a8f7      mov     eax,dword ptr [pluto+0x2028 (f7a81028)]
0034ee89 T:00000001 [bp] f7a8302b e9aed1ffff      jmp     pluto+0x11de (f7a801de)
0034ef16 T:00000001 [bp] f7a801de 55              push    ebp
0034ef93 T:00000001 [bp] f7a80219 8b45fc          mov     eax,dword ptr [ebp-4]
0034f03f T:00000001 [bp] f7a80253 6844646b20      push    206B6444h
0034f0cb T:00000001 [bp] f7a802a2 b980000000      mov     ecx,80h
0034f148 T:00000001 [bp] f7a802ab 5f              pop     edi
00359086 T:00000001 [bp] f7a8006a 8b4c2408        mov     ecx,dword ptr [esp+8]
0035920c T:00000001 [bp] f7a800f6 833d0420a8f700  cmp     dword ptr [pluto+0x3004 (f7a82004)],0
003592a9 T:00000001 [bp] f7a8010c 8b7760          mov     esi,dword ptr [edi+60h]
00359345 T:00000001 [bp] f7a80114 8b4704          mov     eax,dword ptr [edi+4]
003593e1 T:00000001 [bp] f7a80122 6a10            push    10h
0035945e T:00000001 [bp] f7a80133 85c0            test    eax,eax
003594eb T:00000001 [bp] f7a80147 ff7604          push    dword ptr [esi+4]
00359587 T:00000001 [bp] f7a80176 8bcf            mov     ecx,edi
00359614 T:00000001 [bp] f7a80182 5f              pop     edi
0035ac5b T:00000001 [bp] f7a8002e 55              push    ebp

current context:

eax=00000001 ebx=0000c271 ecx=8055192c edx=000003f8 esi=00000001 edi=291f0c30
eip=804e3b25 esp=80550830 ebp=80550840 iopl=0         nv up ei pl nz na po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000202
nt!RtlpBreakWithStatusInstruction:
804e3b25 cc              int     3


commands:

        [m] module list         [0-9] enter recorder modes
        [x] stop recording      [v] toggle verbosity
        [q] quit/close

kd> q
[*] - Exiting Stalker
q
```

Debug Stalk has finished Stalking the points in the driver allowed by the fuzzer. Files named "pluto.sys.0," "pluto.sys-regs.0 (optional)," have been saved to the current working directory.

5.5 Analyzing the output

Pedram has developed a set of Python scripts to support the .bpl and recorder output file, such as adding register metadata to the graph, filtering generated breakpoint lists, additional GDE support for difficult graphs, combining multi-function graphs into a conglomerate graph, highlighting interesting blocks, importing back into the IDA changes made directly to the graph, adding function offsets to breakpoint addresses and optionally rebasing the recording addresses, and much more. Pedram provides detailed descriptions and usage of his python scripts in his manual. The Python scripts used for formatting the .gml files (for block based coverage) are ps_process_recording and ps_view_recording_funcs. The ps_process_recording script is executed first on the pluto.sys.0 which will produce another file called pluto.sys.0.BadFuzz-processed. The ps_view_recording_funcs is executed on the pluto.sys.0.BadFuzz-processed file to produce the file called BadFuzz.gml, which is the chosen name for the initial testing technique. More information on Pedram's Python scripts, reference the Process Stalking Manual. Opening the resulting .gml file will enable us to view the following graph.



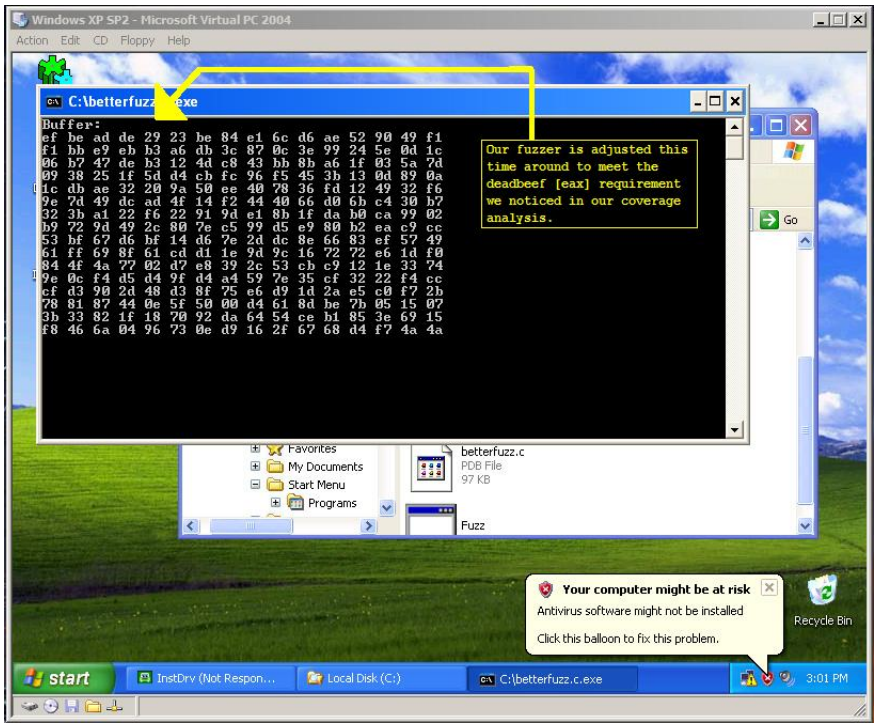
Executed blocks are available in pink, unexecuted blocks are shown as grey, paths of execution are green lines, and unexecuted paths are red lines. At this point it is important to note that the code block starting at address 00011169 does not get executed. This is detrimental to our testing process because it appears that fuzzer supplied data is passed to it and it does not appear to

get executed. Based on this evidence, we can conclude that a readjustment of our testing methodologies needs to be put in place so that we can hit that unexecuted block.

Analysis indicates that the device driver does not execute block 00011169 because a comparison is made in the block at address 00011147 which reveals that [eax] does not match a specified value. Since eax is pointing to the fuzzer supplied data, we should be able to adjust the fuzzer to meet the requirement of the 00011161 cmp dword ptr [eax], 0DEADBEEFh instruction, which will allow us to get into block 00011169. BetterFuzz.exe was improved to do complete the previous description.

5.5.1 Part II

Determining that the previous testing methodology is not effective, a re-engineering of the test case has been implemented and re-testing the driver to hit the missed block can now be accomplished. Following the steps provided in Part I, the driver is loaded into the Virtual PC, kd is attached to the driver process, and Debug Stalk has been loaded into kd and has been invoked to run by using the 'g' command. The entire process is the same except that when the new fuzz test is invoked, different output is printed to kd:

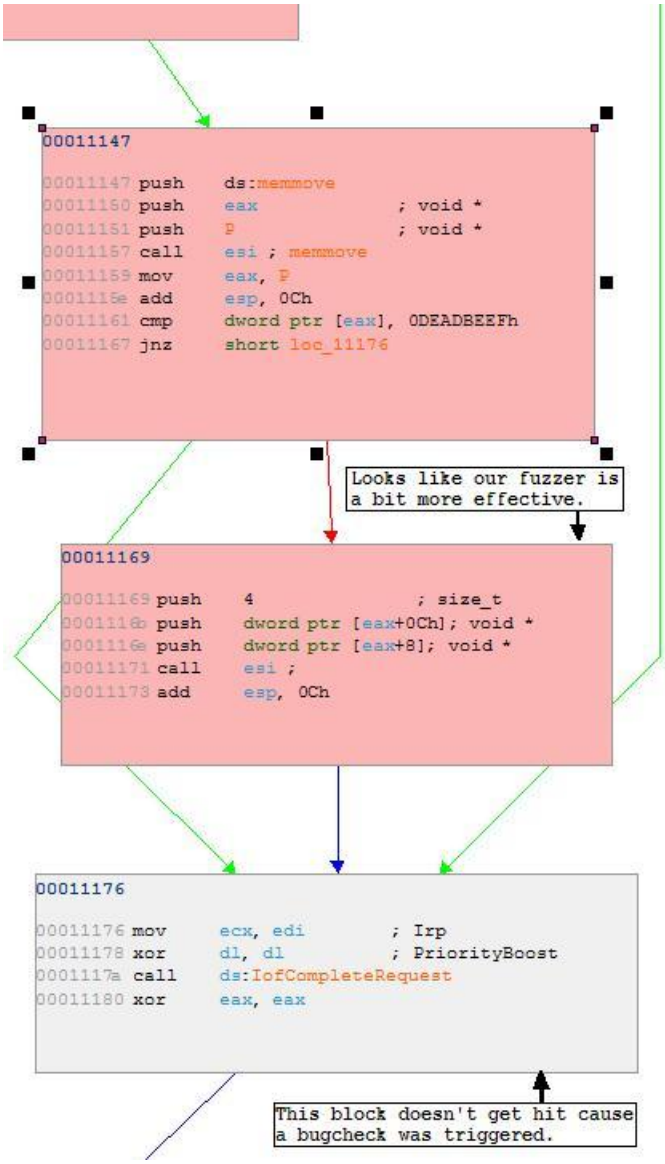



```
kd> g
[*] - Recorder Opened......: pluto.sys.0
[*] - Recorder Opened......: pluto.sys-regs.0
Modload: Processing breakpoints for module pluto.sys at f7a27000
Modload: Done. 46 of 46 breakpoints were set.
004047a0 T:00000001 [bp] f7a2b000 a100a0a2f7      mov     eax,dword ptr [pluto+0x3000 (f7a2a000)]
004052bc T:00000001 [bp] f7a2b00e 3bc1            cmp     eax,ecx
00405339 T:00000001 [bp] f7a2b012 a12890a2f7      mov     eax,dword ptr [pluto+0x2028 (f7a29028)]
004053e5 T:00000001 [bp] f7a2b02b e9aed1ffff      jmp     pluto+0x11de (f7a281de)
00405462 T:00000001 [bp] f7a281de 55              push    ebp
004054ee T:00000001 [bp] f7a28219 8b45fc          mov     eax,dword ptr [ebp-4]
0040558b T:00000001 [bp] f7a28253 6844646b20      push    206B6444h
00405617 T:00000001 [bp] f7a282a2 b980000000      mov     ecx,80h
00405694 T:00000001 [bp] f7a282ab 5f              pop     edi
00406ccc T:00000001 [bp] f7a2806a 8b4c2408        mov     ecx,dword ptr [esp+8]
00406e04 T:00000001 [bp] f7a280f6 833d04a0a2f700  cmp     dword ptr [pluto+0x3004 (f7a2a004)],0
00406eb0 T:00000001 [bp] f7a2810c 8b7760          mov     esi,dword ptr [edi+60h]
00406f4c T:00000001 [bp] f7a28114 8b4704          mov     eax,dword ptr [edi+4]
00406ff8 T:00000001 [bp] f7a28122 6a10            push    10h
00407075 T:00000001 [bp] f7a28133 85c0            test    eax,eax
00407102 T:00000001 [bp] f7a28147 ff7604          push    dword ptr [esi+4]
004071ae T:00000001 [bp] f7a28169 6a04            push    4

current context:

eax=00000003 ebx=00000000 ecx=8050589d edx=0000006a esi=00000000 edi=f1499052
```

```
eip=804e3b25 esp=f3cbe720 ebp=f3cbe768 iopl=0         nv up ei pl zr na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000246
nt!RtlpBreakWithStatusInstruction:
804e3b25 cc              int     3

commands:

        [m] module list         [0-9] enter recorder modes
        [x] stop recording      [v] toggle verbosity
        [q] quit/close

kd> k
ChildEBP RetAddr
f3c1971c 805328e7 nt!RtlpBreakWithStatusInstruction
f3c19768 805333be nt!KiBugCheckDebugBreak+0x19
f3c19b48 805339ae nt!KeBugCheck2+0x574
f3c19b68 805246fb nt!KeBugCheckEx+0x1b
f3c19bb4 804e1ff1 nt!MmAccessFault+0x6f5
f3c19bb4 804da1ee nt!KiTrap0E+0xcc
*** ERROR: Module load completed but symbols could not be loaded for pluto.sys
f3c19c48 f79f0173 nt!memmove+0x72
WARNING: Stack unwind information not available. Following frames may be wrong.
f3c19c84 8057a510 pluto+0x1173
f3c19d38 804df06b nt!NtWriteFile+0x602
f3c19d38 7c90eb94 nt!KiFastCallEntry+0xf8
0006fec0 7c90e9ff ntdll!KiFastSystemCallRet
0006fec4 7c81100e ntdll!ZwWriteFile+0xc
0006ff24 01001276 kernel32!WriteFile+0xf7
0006ff44 010013a7 betterfuzz_c!main+0xa4
0006ffc0 7c816d4f betterfuzz_c!mainCRTStartup+0x12f
0006fff0 00000000 kernel32!BaseProcessStart+0x23

current context:

eax=00000003 ebx=00000000 ecx=8050589d edx=0000006a esi=00000000 edi=f1499052
eip=804e3b25 esp=f3c19720 ebp=f3c19768 iopl=0         nv up ei pl zr na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000246
nt!RtlpBreakWithStatusInstruction:
804e3b25 cc              int     3

commands:

        [m] module list         [0-9] enter recorder modes
        [x] stop recording      [v] toggle verbosity
        [q] quit/close

kd> q
[*] - Exiting Stalker
q

C:\Uninformed>
```

Generating the .gml file allows the tester to view the new execution path. In this case the block at address 00011169 is executed. All subsequent blocks underneath it are not executed because the driver BugChecks inside of this

newly hit block indicating a bug of some sort. Command 'k' in kd produces the stack unwind information and we can see that a BugCheck was initiated for an Access Violation that occurs inside of pluto.sys.



5.6 Part III

Analysis of the graph BadFuzz.gml generated in Part I indicated that the testing methods used were not effective enough to exhibit optimal code coverage of the device driver in question. Part II implemented an improved test case based on the coverage analysis used in Part I. Graph BetterFuzz.gml allowed test executers to view the improved testing methods to ensure that the missed block was reached. This process revealed a fault in block 00011169 which would have otherwise remained undetected without code coverage analysis.

Chapter 6

Conclusion and Future Work

This paper illustrated an improved testing technique by taking advantage of code coverage methods using basic graph theory. The author would like to reiterate that the driver and fuzz tool used in this paper were simple examples to illustrate the effectiveness of code coverage practices.

Finally, more research and experimentation are needed to fully implement these theorems. The question remains on how to integrate a full code coverage analysis tool and a fuzzing tool. Much work has been done on code coverage techniques and their implementations. For example, the paper entitled *Cryptographic Verification of Test Coverage Claims*, Devanbu, et al[1] presents protocols for coverage testing methods such as verifying coverage with and without source code, with just the binary which can utilize both block and branch testing (e0178[1].PDF). A tool to automate the espousal of code coverage and fuzz technologies needs to be implemented so that the two technologies may work together without manual investigation. Further research may include more sophisticated coverage techniques using graph theory such as super blocks, denominators, and applying weights to frequently used loops, paths and edges. CFGs may also benefit from Bayesian networks which are a directed cyclic graph of nodes represented as variables including distribution probability for these variables given the values of its parents. In other words, the Bayesian theory may be helpful for deterministic prediction of code execution which can in turn lead to more intelligent fuzzing. In closing, the author extends the hope that methods and methodologies shared herein can offer other ideas to researchers.

Bibliography

[1] Devanbu, T (2000). *Cryptographic Verification of Test Coverage Claims*. IEEE. 2, 178-192.