Trustworthy Computing

Effective Fuzzing Strategies

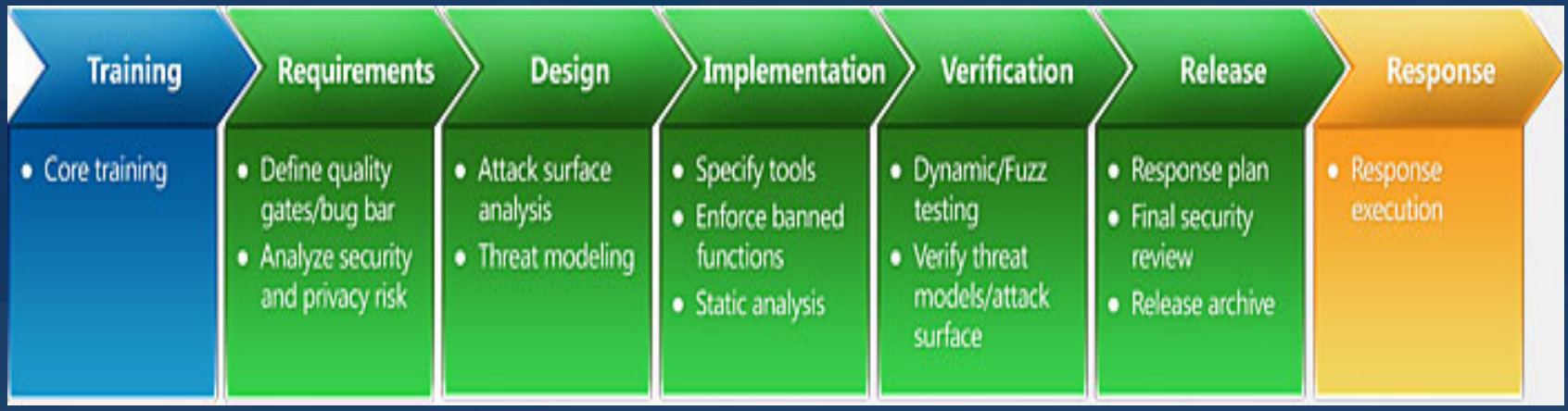
Lars Opstad
Engineering Manager
Microsoft Security Engineering Center (MSEC)

David Molnar
Researcher – Security & Privacy
Microsoft Research (MSR)

Microsoft

Security is a journey, not a destination

- The Security Development Lifecycle (SDL) was created to build security into the process of engineering software



- Part of the process is verifying that the security measures in place actually work
- Fuzzing can be used for verification

Answer These Questions First

- How do I know if my fuzzing is effective?
- Have to answer 3 other questions first
  - What approach should I take?
  - What do I look for when I fuzz?
  - How much is enough?
- Then an effective strategy can be built

Define the Target

- What are you really fuzzing?
  - Web Service
  - Protocol Parser
  - File Parser
  - Local Service
- What Type of Data is Being fuzzed?
  - Binary
  - Text
- Are there Layered Attack Surfaces?
  - Is there a wrapper?
  - Compressed?
  - Initial validation that would reject fuzzed data?

What are the Tools?

- Dumb Fuzzers
  - Easy to build and easy to use
  - Relatively low-investment to find a lot of bugs
  - Penetration may not be very deep
  - Preferred method by many in the industry
- Smart Fuzzers
  - High cost of entry
  - Format aware
  - Highly configurable
  - Better penetration in some cases
  - Find different bugs
  - “Grammar based” & “Whitebox”

Smart Fuzzing Case Study

- MS07-017 had to do with repeating ANI headers
  - 1st ANIH ☺
  - 2nd ANIH ☹
  - Wrapped by an Exception Handler
- Fuzz the framework, not just the values
- A dumb fuzzer would never find this issue
- A *grammar based* fuzzer could find it
  - Need a grammar for ANI (from where?)
  - If the grammar is too strict, it wouldn't fuzz the headers and could miss this type of issue
- The debugger has to be smart enough to catch first chance exceptions

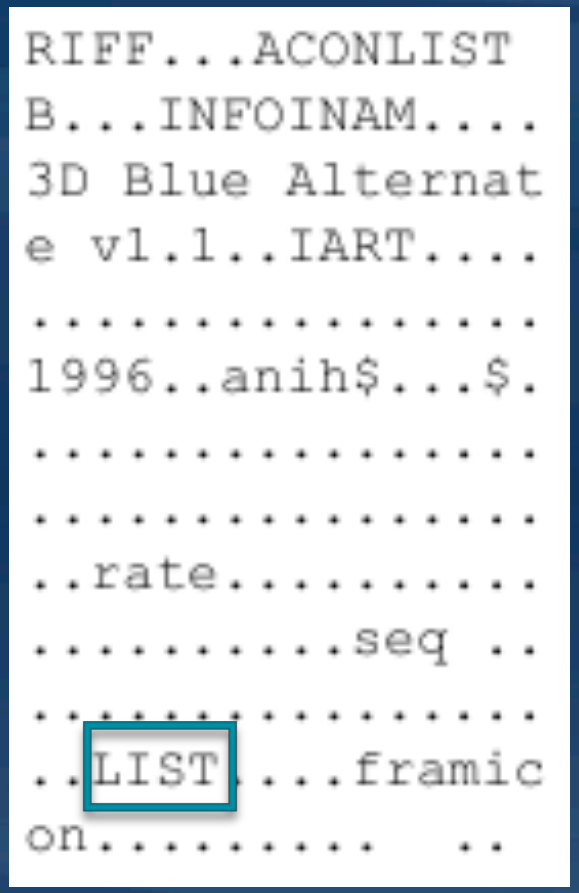
“Whitebox” fuzz testing

- Watch program run on seed file
  - Pick your favorite ANI file
- Treat program input as “tainted”
  - See program compare input bytes to ‘anih’
- Create constraints on tainted input
  - Constraint: bytes so-and-so equal to ‘anih’
- Solve for new input
  - State of the art constraint solver Z3
  - Solve for code coverage or buffer overflows

Tool: Microsoft SAGE

- "Scalable, Automated, Guided, Execution"
- Daily Win7 fuzzing on 100s of machines
- Credit due to entire SAGE team & users!
  - Center for Software Excellence
    - Michael Levin, Chris Marsh, Lei Fang, Stuart de Jong, Dennis Jeffries
  - Microsoft Research
    - Patrice Godefroid, Ella Bounimova, David Molnar, Adam Kiezun, Bassem Elkarablieh, more…
    - Solver: Nikolaj Bjorner, Leonardo de Moura
  - Windows, Office, many other users

SAGE and the ANI bug



Seed

Crash!

7 hours 36 minutes, single core 2GHz box

7706 total test cases generated

No grammar or human guidance needed

Fuzzing at Microsoft

- We Use Semi-Dumb Mutational Fuzzers First
  - Mutating existing data gives us a high probability of the fuzzed data being accepted by the target
- SAGE highly effective in Windows
  - Run after dumb fuzzing finished → multiple new bugs!
- Developing custom grammar based fuzzers has not provided a very good ROI
  - Needs domain knowledge to build, configure
  - Providing a minimal amount of initial “Fix-up” in a script is much easier than trying to define a type (e.g. CRC’s)
  - Dumb fuzzers are easy to deploy
- Research has led us to use a combined approach – Illustrated by the Fuzzing Olympics

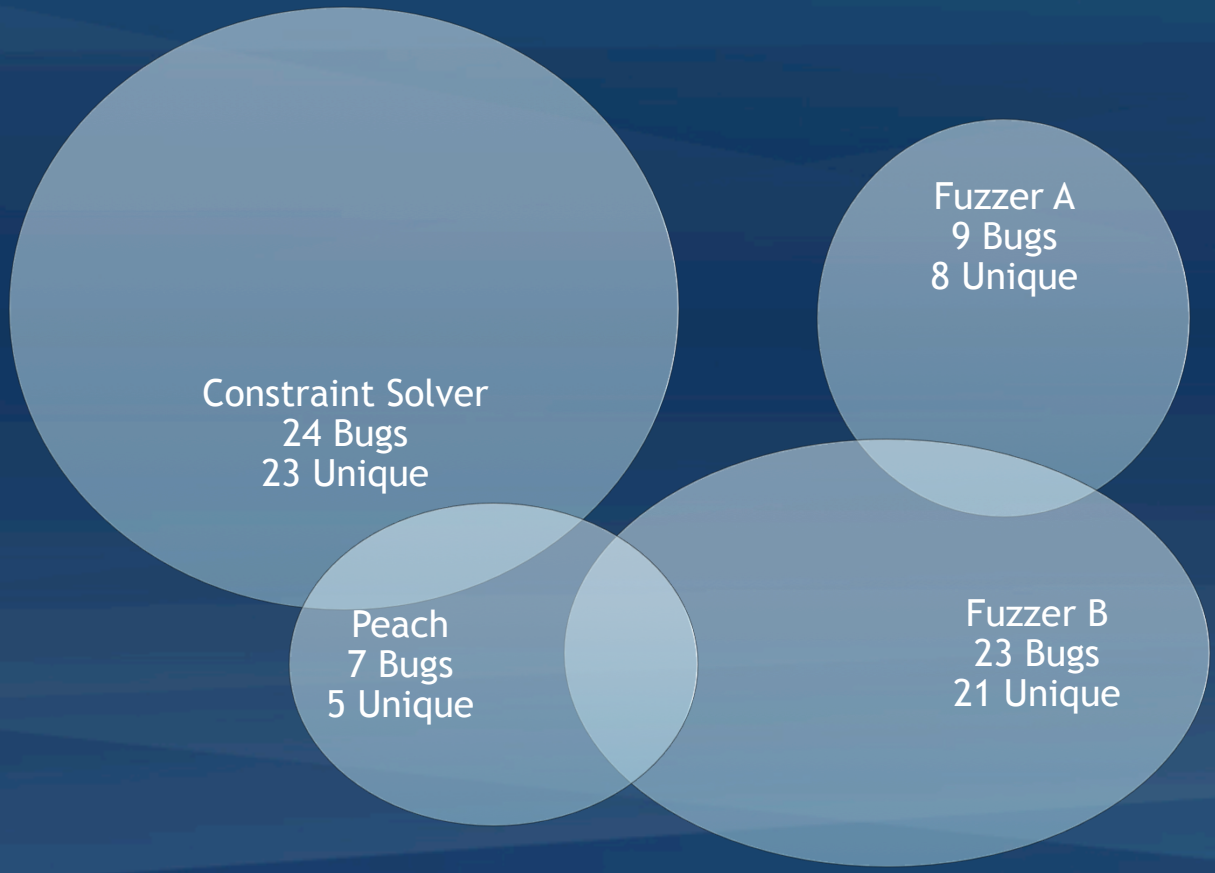
Microsoft Fuzzing Olympics

- Competition held for Bluehat 8 (Fall 2008)
- Several tools competed head-to-head
  - Several Internal Mutational Fuzzers
  - SAGE constraint solver
  - Peach – An External Mutational/Generation
- Level playing field
  - Same timeframe
  - Same targets
    - 1 text parser, 1 binary parser – both previously untested

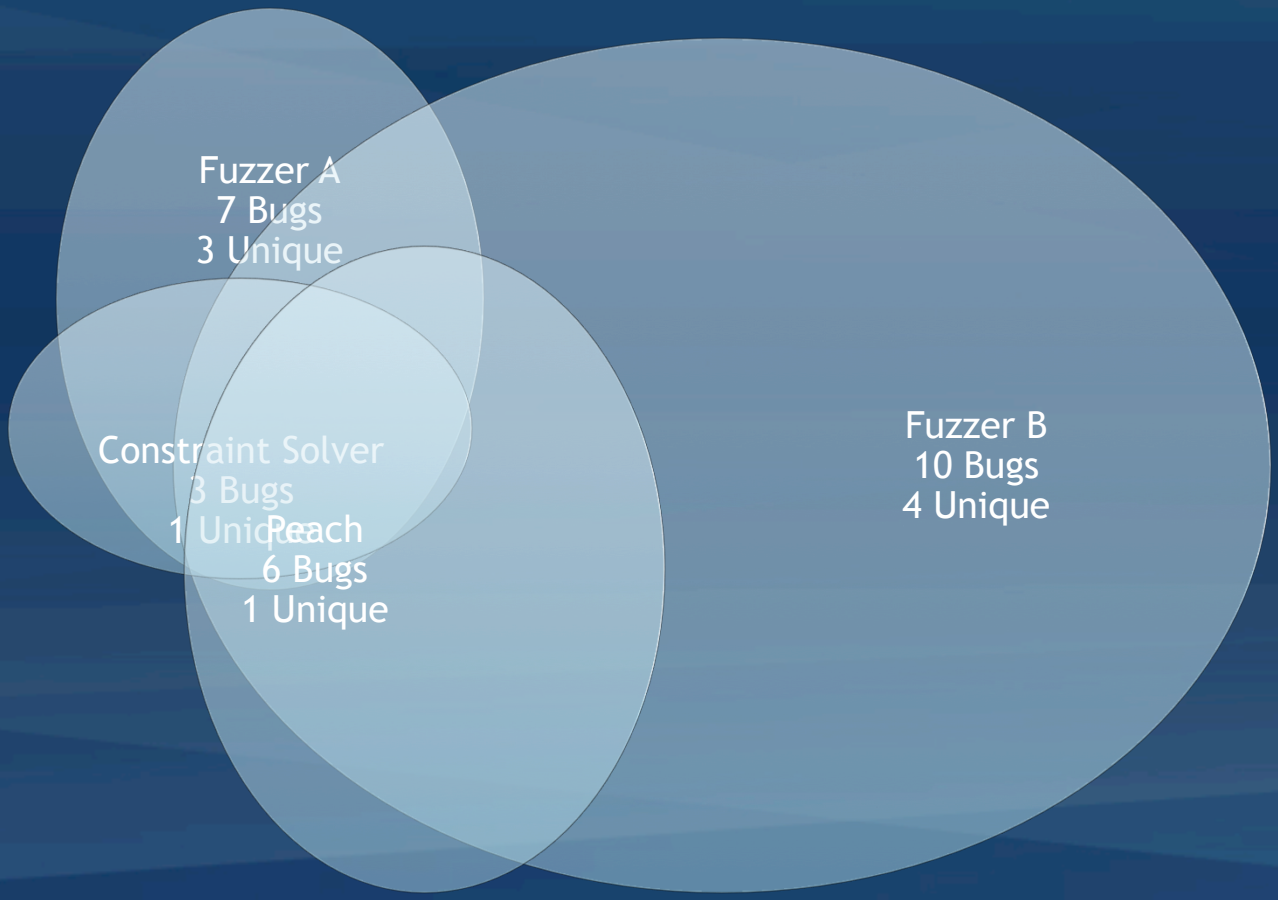
Olympics Findings

- No one fuzzer found ALL of the bugs
  - There was a lot of overlap in the bugs found
  - Many bugs were discovered, including one MSRC grade issue
  - Many of the bugs found were in close proximity to others in the code (major hashes)
- Developing custom grammars did not appear to provide a very good ROI
  - Dumb Fuzzing found the majority of the bugs
  - Other internal fuzzing efforts support this as well
- SAGE found more bugs (minor hashes) than any other fuzzer, but it's more complicated than that…

Olympic Results – Text Parser
60 distinct crashes



Olympic Results – Text Parser
16 underlying bugs



Diversify!

- A Primary Rule of Fuzzing:
  - Change your approach, find different bugs
- Try a different method
  - Mutational
  - Generation
  - Sequential
  - Constraint Solving
- Fuzzing with a second approach measurably increased effectiveness
  - 10%-300% in this case

Make the Most of Your Tools

- Check for penetration
  - Validate code coverage
  - Consider bypassing or proxying any tricky authentications, and test those separately
- Create custom fuzzers for small hard-to reach areas
- Template Optimization (Mutation only)
  - Using the smallest number of templates with the maximum amount of Coverage
  - Template Optimization increases effectiveness by ~100%*

*Based on research by Gavin Thomas, MSRC Engineering

Template Optimization Detail

- Measure code coverage for each template
- Use the following algorithm ("Greedy Search")

```
while (FullTemplateList.count>0) {
        BestTemplate=FullTemplateList.TemplateWithHighestCoverage();
        OptimalTemplateList.Add(BestTemplate);
        FullTemplateList.Remove(BestTemplate);
        foreach (Template t in FullTemplateList) {
                t.Coverage.Exclude(BestTemplate.Coverage);
                if (t.BlocksCovered == 0)
                        FullTemplateList.Remove(t);
        }
}
```

- Use resulting templates for mutation fuzzing
- Double effectiveness in many experiments

WHAT DO I LOOK FOR WHEN I FUZZ?

Scaling a Difficult Problem

- Problems exist with identifying unique crashes
  - The same issue can arise multiple times
  - The same issue can arise through multiple code paths
  - The same issue can be found across multiple machines
- Classifying the crashes is another issue entirely
  - Manual inspection of crash dumps does not scale
  - Identifying security issues takes experienced resources
  - Takes a lot of time to manually analyze the crash
- Testing produces more crashes than there are resources to triage
  - Automation can help trim down the triaging
  - Grouping crashes by location in code helps

!exploitable Crash Analyzer

- What is it?
  - Windows debugger extension (Windbg.exe)
  - Provides automated crash analysis
  - Provides security risk assessment
- How does it work?
  - A live crash or dump is examined using a debugger on Windows
  - !exploitable analyzes crash data
  - Identifies the uniqueness of each crash
  - Provides reliable guidance on exploitability
- What is the output? (Bucketizing)
  - An exploitability indicator identifies whether the crash is:
    - Exploitable
    - Probably Exploitable
    - Probably Not Exploitable
    - Unknown
  - A set of identifying uniqueness indicators
    - Hashes

Automated Crash Triage

544 Crashes

71 Unique Stacks

23 Underlying Bugs

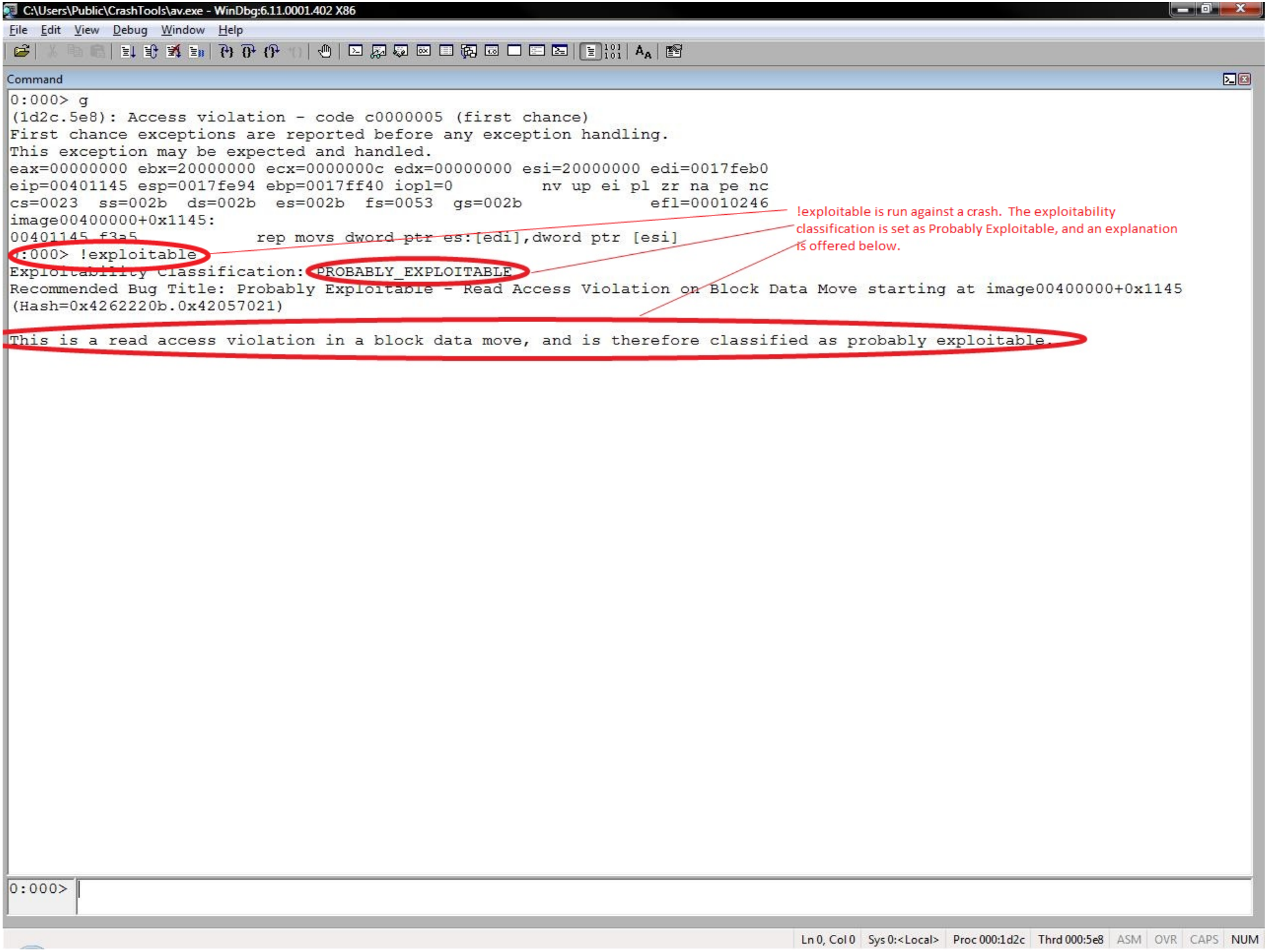
2 Issues Classified as Probably Exploitable

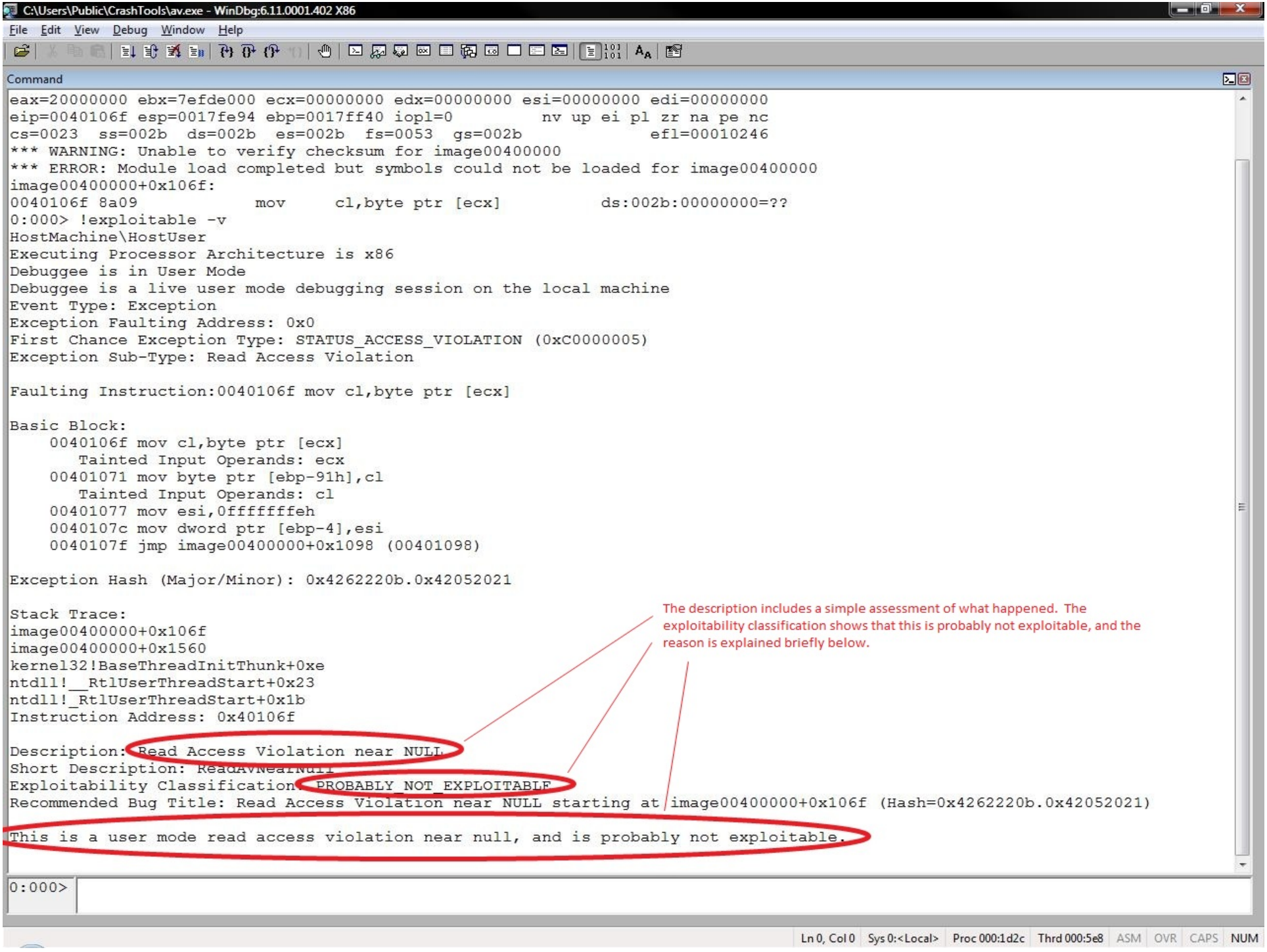
1 Critical Fix

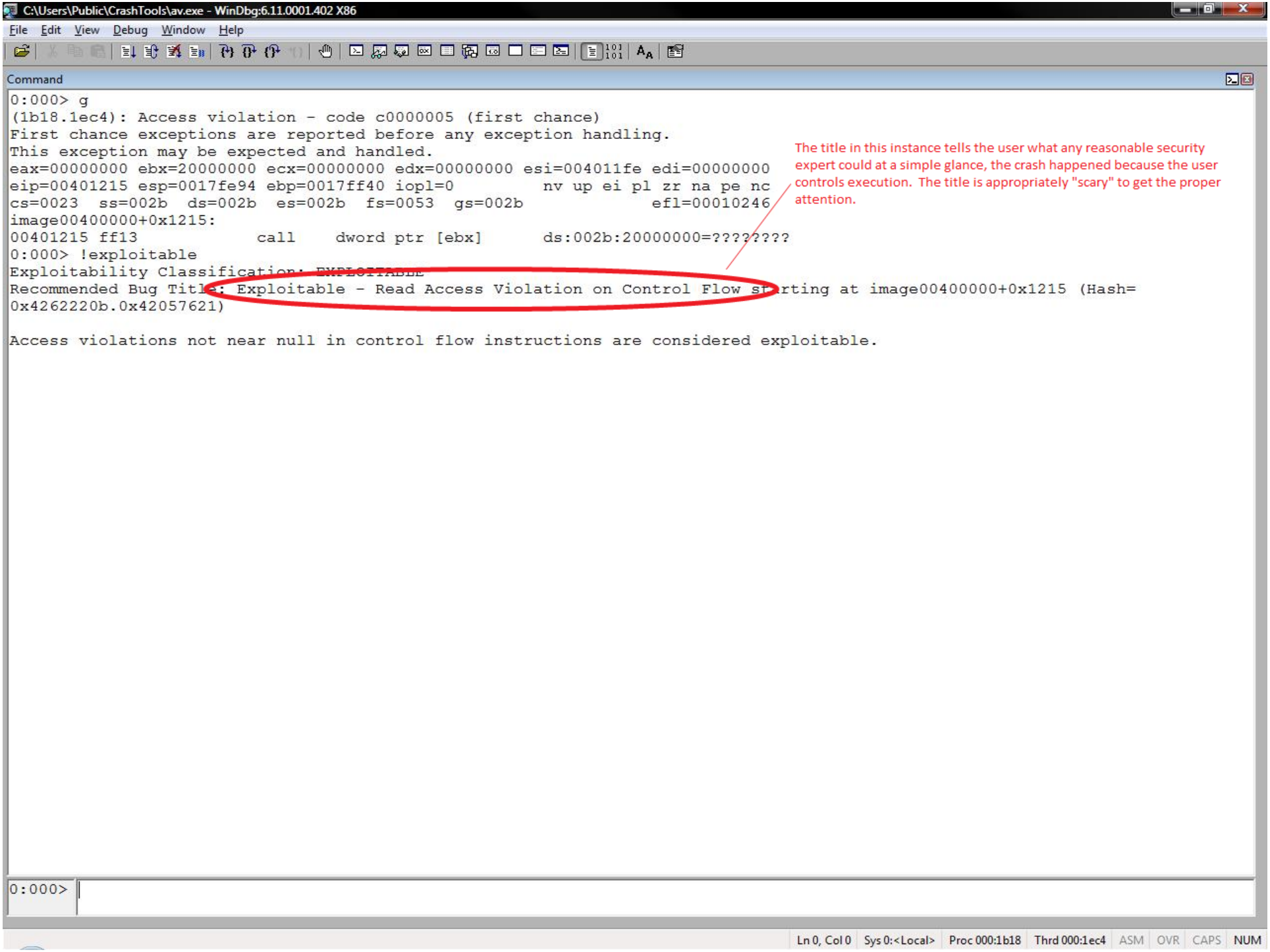
Trustworthy Computing

!exploitable Crash Analyzer

Walkthrough


C:\Users\Public\CrashTools\av.exe - WinDbg:6.11.0001.402 X86
File Edit View Debug Window Help
Command
0:000> g
(1d2c.5e8): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=20000000 ecx=0000000c edx=00000000 esi=20000000 edi=0017feb0
eip=00401145 esp=0017fe94 ebp=0017ff40 iopl=0 nv up ei pl zr na pe nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00010246
image00400000+0x1145:
00401145 f3a5 rep movs dword ptr es:[edi],dword ptr [esi]
0:000> !exploitable
Exploitability Classification: PROBABLY_EXPLOITABLE
Recommended Bug Title: Probably Exploitable - Read Access Violation on Block Data Move starting at image00400000+0x1145
(Hash=0x4262220b.0x42057021)
This is a read access violation in a block data move, and is therefore classified as probably exploitable.
!exploitable is run against a crash. The exploitability classification is set as Probably Exploitable, and an explanation is offered below.
0:000>
Ln 0, Col 0 Sys 0:<Local> Proc 000:1d2c Thrd 000:5e8 ASM OVR CAPS NUM


C:\Users\Public\CrashTools\av.exe - WinDbg:6.11.0001.402 X86
File Edit View Debug Window Help
Command
eax=20000000 ebx=7efde000 ecx=00000000 edx=00000000 esi=00000000 edi=00000000
eip=0040106f esp=0017fe94 ebp=0017ff40 iopl=0 nv up ei pl zr na pe nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00010246
*** WARNING: Unable to verify checksum for image00400000
*** ERROR: Module load completed but symbols could not be loaded for image00400000
image00400000+0x106f:
0040106f 8a09 mov cl,byte ptr [ecx] ds:002b:00000000=??
0:000> !exploitable -v
HostMachine\HostUser
Executing Processor Architecture is x86
Debuggee is in User Mode
Debuggee is a live user mode debugging session on the local machine
Event Type: Exception
Exception Faulting Address: 0x0
First Chance Exception Type: STATUS_ACCESS_VIOLATION (0xC0000005)
Exception Sub-Type: Read Access Violation
Faulting Instruction:0040106f mov cl,byte ptr [ecx]
Basic Block:
0040106f mov cl,byte ptr [ecx]
Tainted Input Operands: ecx
00401071 mov byte ptr [ebp-91h],cl
Tainted Input Operands: cl
00401077 mov esi,0fffffffeh
0040107c mov dword ptr [ebp-4],esi
0040107f jmp image00400000+0x1098 (00401098)
Exception Hash (Major/Minor): 0x4262220b.0x42052021
Stack Trace:
image00400000+0x106f
image00400000+0x1560
kernel32!BaseThreadInitThunk+0xe
ntdll!__RtlUserThreadStart+0x23
ntdll!_RtlUserThreadStart+0x1b
Instruction Address: 0x40106f
Description: Read Access Violation near NULL
Short Description: ReadAvNearNull
Exploitability Classification: PROBABLY_NOT_EXPLOITABLE
Recommended Bug Title: Read Access Violation near NULL starting at image00400000+0x106f (Hash=0x4262220b.0x42052021)
This is a user mode read access violation near null, and is probably not exploitable.
The description includes a simple assessment of what happened. The exploitability classification shows that this is probably not exploitable, and the reason is explained briefly below.
0:000>
Ln 0, Col 0 Sys 0:<Local> Proc 000:1d2c Thrd 000:5e8 ASM OVR CAPS NUM


C:\Users\Public\CrashTools\av.exe - WinDbg:6.11.0001.402 X86
File Edit View Debug Window Help
Command
0:000> g
(1b18.1ec4): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=20000000 ecx=00000000 edx=00000000 esi=004011fe edi=00000000
eip=00401215 esp=0017fe94 ebp=0017ff40 iopl=0 nv up ei pl zr na pe nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00010246
image00400000+0x1215:
00401215 ff13 call dword ptr [ebx] ds:002b:20000000=????????
0:000> !exploitable
Exploitability Classification: EXPLOITABLE
Recommended Bug Title: Exploitable - Read Access Violation on Control Flow starting at image00400000+0x1215 (Hash=
0x4262220b.0x42057621)
Access violations not near null in control flow instructions are considered exploitable.
The title in this instance tells the user what any reasonable security expert could at a simple glance, the crash happened because the user controls execution. The title is appropriately "scary" to get the proper attention.
0:000>
Ln 0, Col 0
Sys 0:<Local>
Proc 000:1b18
Thrd 000:1ec4
ASM
OVR
CAPS
NUM

Who Benefits from !exploitable?

- !exploitable Crash Analyzer helps 3rd party software Developers and Testers working on Microsoft® platforms to manage their workload
  - Developers and Testers don't have to be security experts in order to identify many security issues
  - Can identify and categorize crashes with security implications quickly
  - Helps to prioritize work based on exploitability of crashes
    - "Exploitable" Elevation of Privilege bug may need immediate attention
    - "Probably Not Exploitable" Divide by Zero bug is likely a lower priority
  - Decreases the amount of time needed to analyze crashes for exploitability
- Security Ecosystem
  - Helps standardize exploitability reporting within companies and across the Security Ecosystem
  - Integrated into fuzzers inside and outside of Microsoft

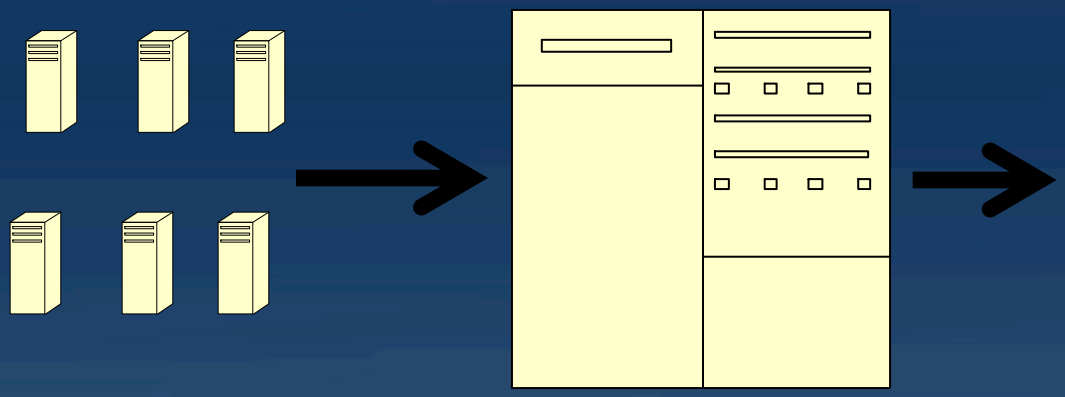
CLOUD FUZZ TESTING

Challenge: Fuzzing at Scale

- You need to try millions of test cases!
- “[R]unning peach on one laptop with 30 ninjas standing around it with IDA Pro open is not going to work.” – Ben Nagy
- Building infrastructure is expensive

Rent Scale With Cloud Providers

- Rent machines from cloud provider
- Each machine fuzzes, reports data
- Organize results, feed to your test team



Current metafuzz stats, last updated Monday 24th of August 2009 05:32:30 PM:
3053 runs in database, total of 2546056 test files created, with 6840 distinct bug stack hashes over all runs.

See the Premade VM Page for instructions on how to contribute your own results!

Current bugs with uploaded test cases (NOTE: this list is regenerated periodically and does not reflect an up to the minute list

Run UUID	Seq. No	Stack Hash	Kind	Program	Status	Fuzz Type	Test Case	Submitter
503650	0	3180556327	InvalidRead	evince	Not Yet	catchconv	Download?	premade@metafuzz.com
503650	0	1168963511	InvalidRead	evince	Not Yet	catchconv	Download?	premade@metafuzz.com
503650	0	1703376083	UninitCondition	evince	Not Yet	catchconv	Download?	premade@metafuzz.com
503650	0	3736762920	UninitCondition	evince	Not Yet	catchconv	Download?	premade@metafuzz.com
503650	0	2194644105	UninitCondition	evince	Not Yet	catchconv	Download?	premade@metafuzz.com
503650	0	2845793442	InvalidRead	evince	Not Yet	catchconv	Download?	premade@metafuzz.com
503650	0	1616894225	InvalidRead	evince	Not Yet	catchconv	Download?	premade@metafuzz.com
503650	0	2351990255	InvalidRead	evince	Not Yet	catchconv	Download?	premade@metafuzz.com
503650	0	2442096583	InvalidRead	evince	Not Yet	catchconv	Download?	premade@metafuzz.com
503650	0	3771527843	SyscallParam	evince	Not Yet	catchconv	Download?	premade@metafuzz.com

- D. Molnar PhD: http://www.metafuzz.com
  - Fuzzing on Amazon Elastic Compute Cloud
  - MySQL DB for results, PHP front end

Trustworthy Computing

Minifuzz Plus Visual Studio Team Foundation Server

Cloud Fuzzing Demo

Minifuzz:

http://edge.technet.com/Media/minifuzz-overview-and-demo/

Team Foundation Server 2008 trial:

http://www.microsoft.com/downloads/details.aspx?FamilyId=B0155166-B0A3-436E-AC95-37D7E39A440C&displaylang=en

Conclusions – A Practical Guide to Fuzzing

- Invest up front in choosing your approach
  - Identify targets
  - Choose the best tools
  - Choose optimal inputs (Template Reduction)
  - Consider leveraging Cloud resources
- Diversify
  - Consider a mix of fuzzing tools and approaches
- Use !exploitable Crash Analyzer
  - Reduces triage time
  - Highlights important security issues quickly

Links

- For more information on Microsoft's Security Science and !exploitable Crash Analyzer, please visit:
  - http://www.microsoft.com/security/**msec**/
- For more information about SAGE:
  - http://channel9.msdn.com/posts/Peli/Automated-Whitebox-Fuzz-Testing-with-SAGE/
  - http://research.microsoft.com/en-us/um/people/pg/public_psfiles/ndss2008.pdf
- And the Security Research & Defense (SRD) blog:
  - http://blogs.technet.com/srd

QUESTIONS?

Microsoft®

Your potential. Our passion.™