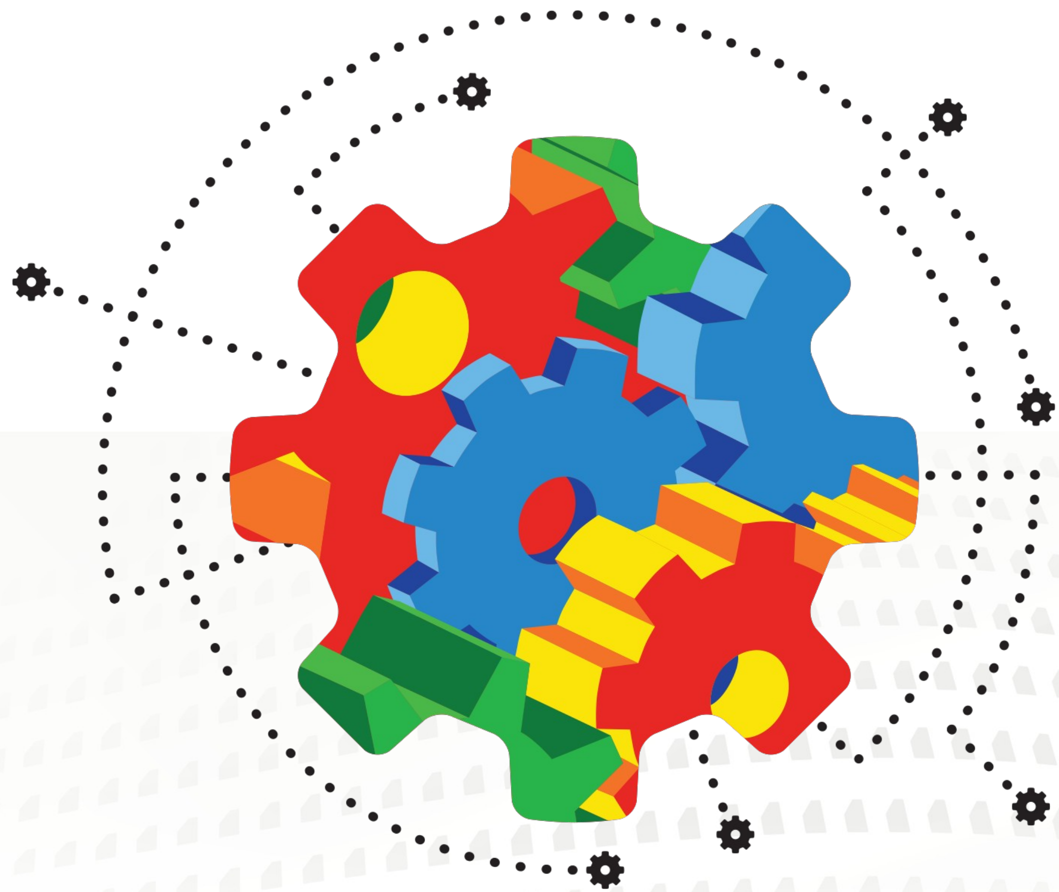
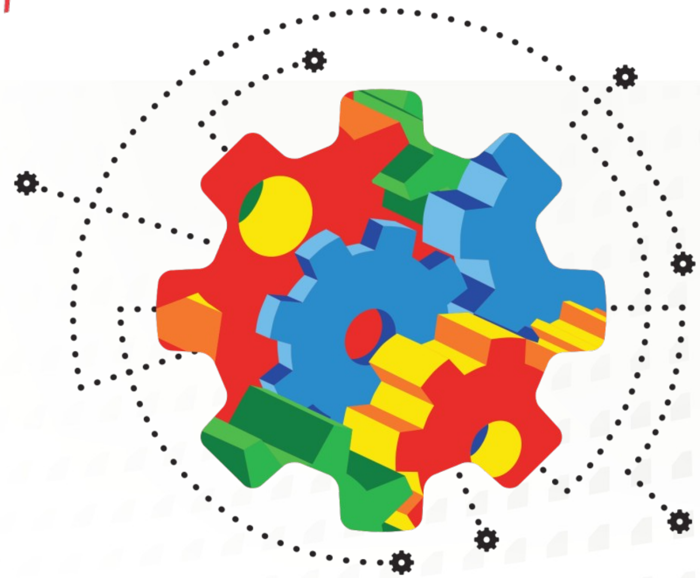


High Performance Fuzzing

Richard Johnson | Hushcon 2015

Introduction

- Whoami
  - →Richard Johnson / @richinseattle
  - →Research Manager, Vulnerability Development
  - →Cisco, Talos Security Intelligence and Research Group
- Agenda
  - →Why Performance Matters
  - →Targeting & Input Selection
  - →Engine Design
  - →Host Configuration



Why Performance Matters

Why Performance Matters

- Mutational fuzzing almost seems too easy
  →Just throw some hardware at the problem
- Majority of CPU cycles are wasted
  →Program load time vs file parsing time
  →Fuzzing requires high I/O, blocking CPU
  →Mutations on large files are inefficient

- Quantitatively analyze fuzzer designs
- Qualitatively analyze fuzzer strategies

Microsoft SDL Verification Guidance

- Fuzzing is a requirement of SDLC Verification:

> "Where input to file parsing code could have crossed a trust boundary, file fuzzing must be performed on that code. All issues must be fixed as described in the Security Development Lifecycle (SDL) Bug Bar. **Each file parser is required to be fuzzed** using a recommended tool."
>
> https://msdn.microsoft.com/en-us/library/windows/desktop/cc307418.asp

Microsoft SDL Verification Guidance

- Fuzzing is a requirement of SDL Verification:

> "Win32/64/Mac: An **Optimized set of templates must be used**. Template optimization is based on the maximum amount of code coverage of the parser with the minimum number of templates. **Optimized templates** have been shown to **double fuzzing effectiveness** in studies. A **minimum of 500,000 iterations**, and have fuzzed at least **250,000 iterations since the last bug found**/fixed that meets the SDL Bug Bar"
>
> https://msdn.microsoft.com/en-us/library/windows/desktop/cc307418.asp

Microsoft SDL Verification Guidance

- Required fuzzing is a good thing
- How did they calibrate?
  - →Iterations limited by practical resources
  - →Parsers with greater complexity require more resources
  - →Iterations is a poor choice for defining guidance
-
- Questions:
  - →What properties define the theoretical limit of available resources
  - →What are the best practices for fuzzing optimize effectiveness

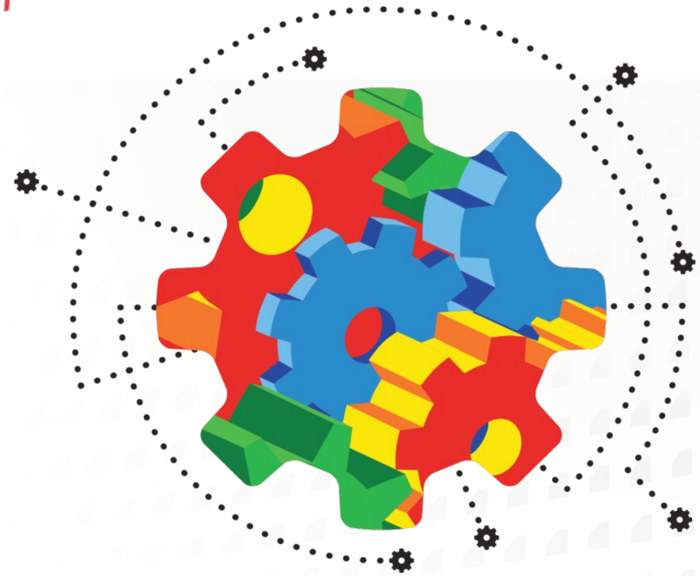
Historical Performance Stats

- Microsoft Windows Vista 2006
  - →350mil iterations, 250+ file parsers
    - ~1.4mil iterations per parser (on average)
  - →300+ issues fixed (1 bug / 1.16 million tests)
- Microsoft Office 2010
  - →800 million iterations, 400 file parsers
  - →1800 bugs fixed (1 bug / 44444 tests)
    - → http://blogs.technet.com/b/office2010/archive/2010/05/11/how-the-sdl-helped-improve-security-in-office-2010.aspx http://www.computerworld.com/article/2516563/security0/microsoft-runs-fuzzing-botnet--finds-1-800-office-bugs.html
- Charlie Miller 2010
  - →7mil iterations, 4 parsers
    - ~1.8m iterations per parser (on average)
  - →320 - 470 unique crashes (1 bug / 14893 - 21875 tests)

Historical Performance Stats (cmiller)

- Charlie Miller intentionally went with a poor design
  - →5-lines of python to mutate input
  - →AppleScript to iterate files with system handler
  - →Microsoft minifuzz is equally stupid
- Input Selection
  - →80,000 PDFs reduced to 1515 via code coverage minset

Input	Software	Count	avg time
PDF	Adobe Reader 9.2.0	3M	5.35s
PDF	Apple Preview (OS X 10.6.1)	2.8M	7.68s
PPT	OpenOffice Impress 3.3.1	610k	32s+
PPT	MSOffice PowerPoint 2008 Mac	595k	32s



Targeting and Input Selection

Target Selection

- 64-bit vs 32-bit applications (x86 architecture)
  - →64-bit binaries are fatter than 32-bit
  - →64-bit runtime memory usage is greater than 32-bit
  - →64-bit OSs take more memory and disk for your VMs

  - →Some software only comes compiled as 32-bit binaries
  - →Some fuzzers and debuggers only support 32-bit

  - →64-bit CPUs have more registers to increase performance
    - Optimization depends on compiler

Target Selection

- So are 64-bit programs faster?
  - →On x64? It varies either way to a small degree
    - Chrome - Negligible
      - http://www.7tutorials.com/google-chrome-64-bit-it-better-32-bit-version
    - Photoshop - YES?
      - 8-12% (but talks about unrelated disk i/o optimizations)
      - https://helpx.adobe.com/photoshop/kb/64-bit-os-benefits-limitations.html
  - →On SPARC? NO
    - True story, but who cares
      - http://www.osnews.com/story/5768/Are_64-bit_Binaries_Really_Slower_than_32-bit_Binaries_/page3/

Target Selection

- Much more important: Minimize lines of code
  - →What is the ratio of time spent initializing program and executing actual parser code
- Optimization strategy
  - →Target libraries directly
  - →Write thin wrappers for each API
    - This allows feature targeting
  - →Patch target to eliminate costly checksums / compression
    - This is what flayer is all about (Drewery & Ormandy WOOT'07)
  - →Instrument target for in-memory fuzzing

Input Selection

- Input is a numerical set
- Input parsers are (should be) state machines
  - →Specifications described using FSM
  - →Actual parser code typically not implemented using FSM
  - →LangSec Paper on high performance FSM Parsers
    - http://www.cs.dartmouth.edu/~pete/pubs/LangSec-2014-fsm-parsers.pdf
- Goal: search space and discover new transitions
- Each search is computationally expensive
  - →We need to optimize for time

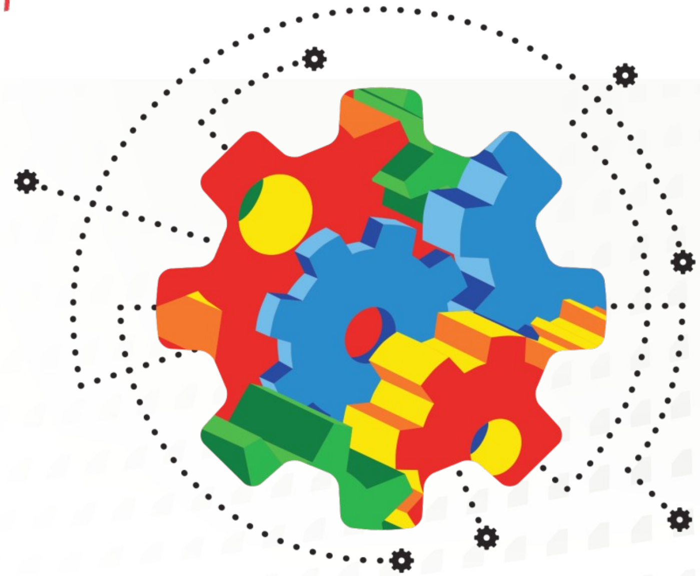
Input Selection

- Optimize input selection
  - →File size is very important
    - Mutations are more meaningful with smaller input size
    - Smaller inputs are read and parsed quicker
    - Some test generation approaches utilize large amounts of memory per-input-byte
  - →Specific feature set per input allows for focused targeting
    - Handcrafted or minimized samples
    - Feedback fuzzing or concolic testing automates creation of unique small inputs with different features

Input Selection

- CMU Coverset
  - →Optimizing Seed Selection for Fuzzing – USENIX 2014
    - https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-rebert.pdf
  - →Minset helps less than expected
  - →Unweighted Minset is the winner
  - →Conclusion: Minset is good when it's not broken
    - Peach minset tool is not minimal set algorithm
    - Peach minset performs equivalent to random selection

- We will talk more about coverage tracer perf in a bit..



Engine Design

Engine Design

- Generate new inputs
- Execute target with new input
- Detect failure conditions

Engine Design

- Generate new inputs
- Execute target with new input
- **Trace target execution**
- **Monitor trace output**
- Detect failure conditions
- **Detect non-failure conditions**

Input Generation

- Most important is the selection of mutators
  →AFL

Deterministic bitflip
    1, 2, 4, 8, 16, 32 bits
Deterministic addition/subtraction
    Values { 1 – 35 } for each byte, short, word, dword
    Little endian and big endian
Deterministic 'interesting' constant values
    27 boundary values
Dictionary keywords
Havoc
    Random bitflips, arithmetic, block move/copy, truncate
Splice
    Merge two previously generated inputs

Input Generation

- Most important is the selection of mutators
  →Radamsa

ab: enhance silly issues in ASCII string data handling
bd: drop a byte
bf: flip one bit
bi: insert a random byte
br: repeat a byte
bp: permute some bytes
bei: increment a byte by one
bed: decrement a byte by one
ber: swap a byte with a random one
sr: repeat a sequence of bytes
sd: delete a sequence of bytes
ld: delete a line

Input Generation

- Most important is the selection of mutators
  →Radamsa

```
lds: delete many lines
lr2: duplicate a line
li: copy a line closeby
lr: repeat a line
ls: swap two lines
lp: swap order of lines
lis: insert a line from elsewhere
lrs: replace a line with one from elsewhere
td: delete a node
tr2: duplicate a node
ts1: swap one node with another one
ts2: swap two nodes pairwise
```

Input Generation

- Most important is the selection of mutators
  →Radamsa

```
  tr: repeat a path of the parse tree
  uw: try to make a code point too wide
  ui: insert funny unicode
  num: try to modify a textual number
  xp: try to parse XML and mutate it
  ft: jump to a similar position in block
  fn: likely clone data between similar positions
  fo: fuse previously seen data elsewhere

Mutation patterns (-p)
  od: Mutate once
  nd: Mutate possibly many times
  bu: Make several mutations closeby once
```

Input Generation

- Deterministic mutators first
- Permutations and infinite random mode
- Stack permutations to a reasonable level

- Need feedback loop to assess effectiveness of new mutators

Execute Target

- Using an execution loop is slow
  →process creation, linking, initialization
-
- Use a fork() server
  →Skip initialization
  →Copy-on-write process cloning is very fast on Linux
  →Windows and OSX manually copy process memory
    - 30x+ performance hit over COW pages

Execute Target

- Windows black magic SUA posix fork() tangent
  - →ZwCreateProcess (NULL, …) – Windows 2000
    - No sections, threads, CSRSS, User32, etc
  - →RtlCloneUserProcess – Windows Vista
    - Works to limited extent
    - Applications cannot use Win32 API
  - →RtlCreateProcessReflection - Windows 7
    - Designed for quick full memory dump creation
    - Does not restore threads
- Windows 10 fork...

Execute Target

- Are you forking kidding me??

linux
    10000 fork()
        0.763s → 13106 exec/sec
    10000 fork/exec(/bin/false)
        2.761s → 3621 exec/sec
    10000 fork/exec(/bin/false) w/ taskset
        2.073s → 4823 exec/sec

cygwin
    10000 fork()
        29.954s → 333 exec/sec
    10000 fork/exec(/bin/false)
        63.898s → 156 exec/sec

Fork/exec(/bin/false) (machine #2)

linux
10000 – 0m2.263s: 4419 exec/sec

msys
10000 – 1m14.952s: 135 exec/sec

cygwin
10000 – 0m46.972s: 213 exec/sec

Native (RtlCloneUserProcess)
10000 – 0m17.457s: 574 exec/sec

Trace Target Execution

- Feedback loop fuzzing finally realized with AFL
  - →Allows qualitative assessment of fuzzing strategy
  - →Optimized instrumentation strategy
  - →Optimized feedback signal
  - →Source code only**
- Previous attempts at binary feedback were too slow
  - →EFS was overly complicated and used PaiMei
  - →BCCF uses COSEINC code coverage Pintool
  - →Honggfuzz uses BTS

Trace Target Execution

- Binary hooking engine selection is critical
  →Pin / DynamoRIO are slow

Cover trace - pngtest single shot

Native	real	0m0.121s	
PIN	real	0m0.898s	7.4x
DynamoRIO	real	0m0.466s	3.8x

Cover trace - pngtest * 551 files

Native	real	0m5.652s	
PIN	real	4m35.170s	48.6x
DynamoRIO	real	1m9.840s	12.3x

Trace Target Execution

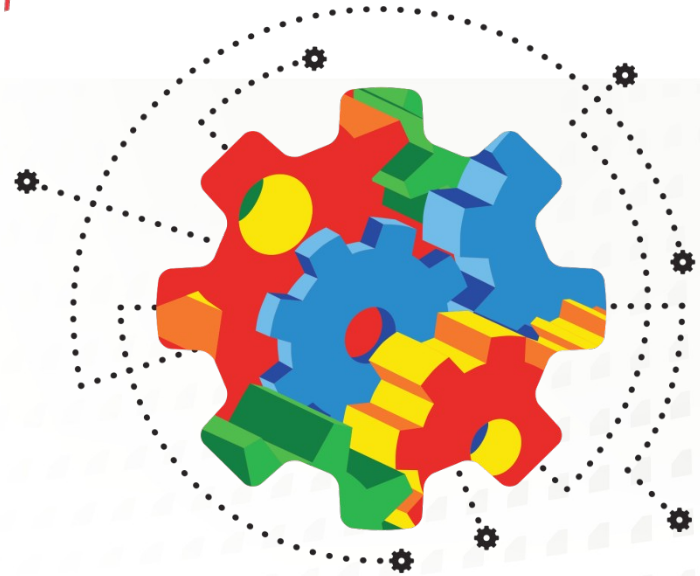
Binary hooking engine selection is critical

→Pin / DynamoRIO are slow

Skip instrumentation and hooking with a fork server!

TurboTrace:

1. Fork self in LD_PRELOADed library.
2. ptrace the forked child.
3. Break on _start
4. Inject a call to the actual function that will be doing repeated fork()ing.
5. Step over a call.
6. Repair the _start and resume execution.
    6a) On each iteration fixup argv



TurboTracer Demo

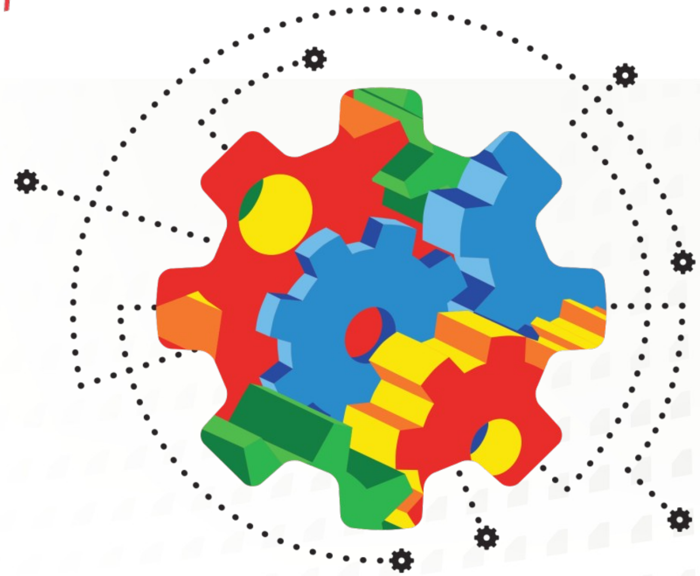
Trace Target Execution

- Binary hooking engine selection is critical
  →TurboTrace performance, 100 iterations
  - 20 – 50% speed increase

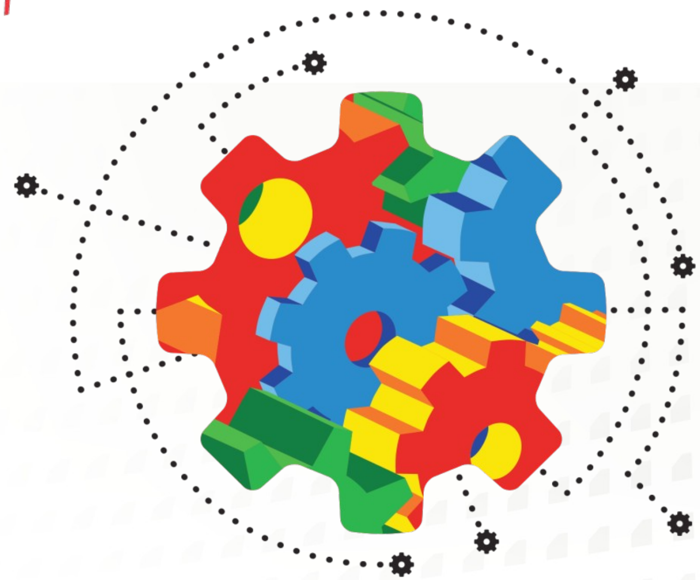
PIN initialization only:	
Pin	without pintool on test_png : 55.03 seconds
Turbotrace	without pintool on test_png : 37.24 seconds
bblocks pintool:	
Pin	bblocks pintool on test_png : 72.62 seconds
Turbotrace	bblocks pintool on test_png : 51.07 seconds
calltrace pintool:	
Pin	calltrace pintool on test_png : 106.19 seconds
Turbotrace	calltrace pintool on test_png : 85.24 seconds

Trace Target Execution

- Binary hooking engine selection is critical
  - →QEMU
    - Uses QEMU userland block tracing
    - Statically compiled binaries
    - Linux only
    - Readpng: ~860 ex/s vs ~3800 afl-gcc – 4.5x slower
  - →DynInst
    - Static binary rewriting
    - Dynamically compiled binaries
    - Linux only for now (windows port in progress)
    - Readpng: ~2400 ex/s vs ~3300 afl-gcc – 1.3x slower



AFL-DYNINST DEMO



But wait there's more!

Hushcon Special Preview Demo

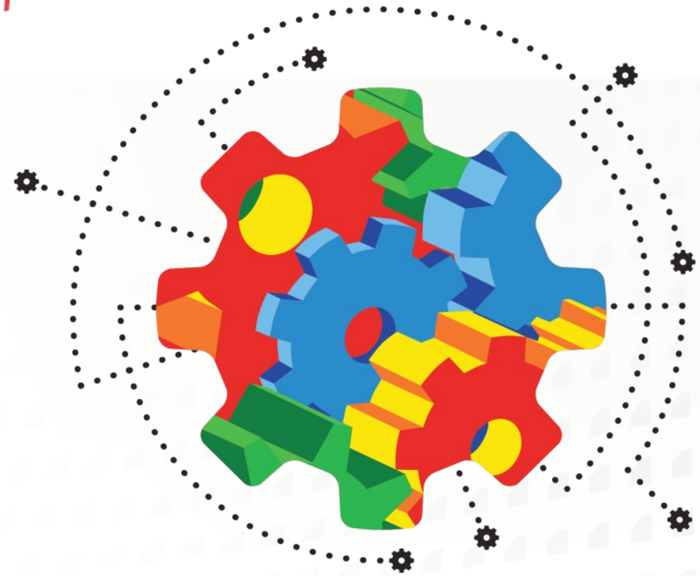
Monitor Trace Output

- Logging is critical, tracers perform way too much I/O
  →Only store enough for feedback signal
- Block coverage is weak, edge transitions are better
- Use shared memory

```
cur_location = (block_address >> 4) ^ (block_address << 8);
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

Detect Failure / Non-Failure

- Failure
  - →Linux
    - #define WTERMSIG(status) ((status) & 0x7f)
  - →Windows
    - Debugger is the only option
- Non-Failure
  - →Timeout
    - Self calibrate
    - Lowest possible timeout,
  - →CPU Usage
    - If CPU utilization drops to near zero for X millisec



Host Configuration

System Cache

- Windows
  - →Pre-Windows 7 used only 8 MB memory for filesystem cache
    - HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management
    - Set value LargeSystemCache = 1
  - →Enable disk write caching in disk properties

System Cache

- Linux
  - →Enables large system cache by default
  - →/sbin/hdparm -W 1 /dev/hda 1 Enable write caching
  - →$ sysctl -a | grep dirty
    - vm.dirty_background_ratio = 10
    - vm.dirty_background_bytes = 0
    - vm.dirty_ratio = 20
    - vm.dirty_bytes = 0
    - vm.dirty_writeback_centisecs = 500
    - vm.dirty_expire_centisecs = 3000

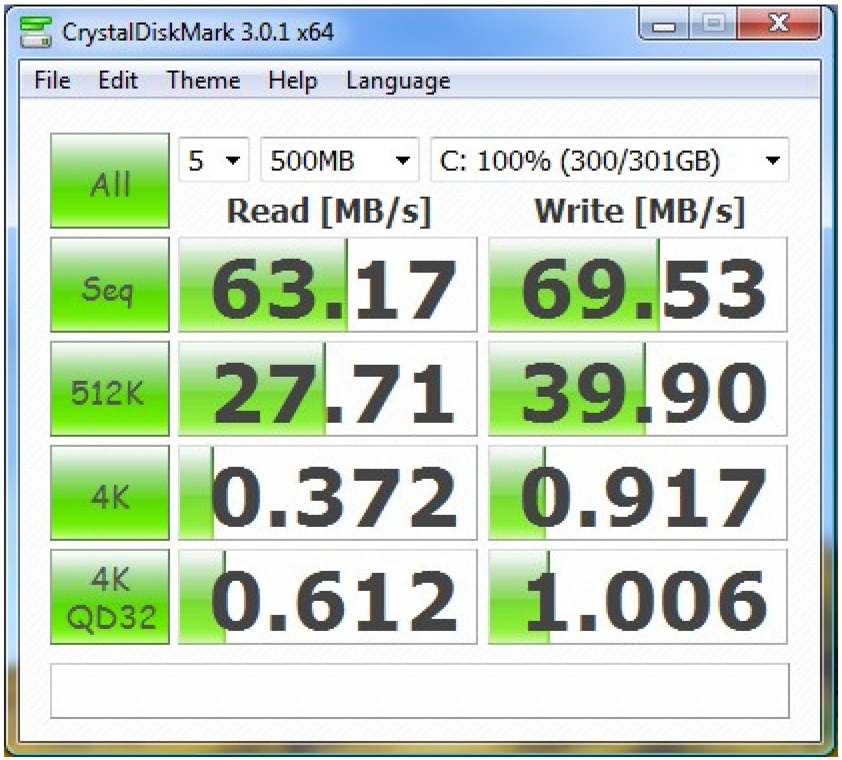
Storage: HDD

- ~100 MB/s
- Cache commonly used programs proactively
  - →Windows Superfetch (default)
  - →Linux Preload
    - http://techthrob.com/tech/preload_files/graph.png
- Features are most useful in low memory availability scenarios
  - →Typical for fuzzing w/ 1-2gb memory per VM

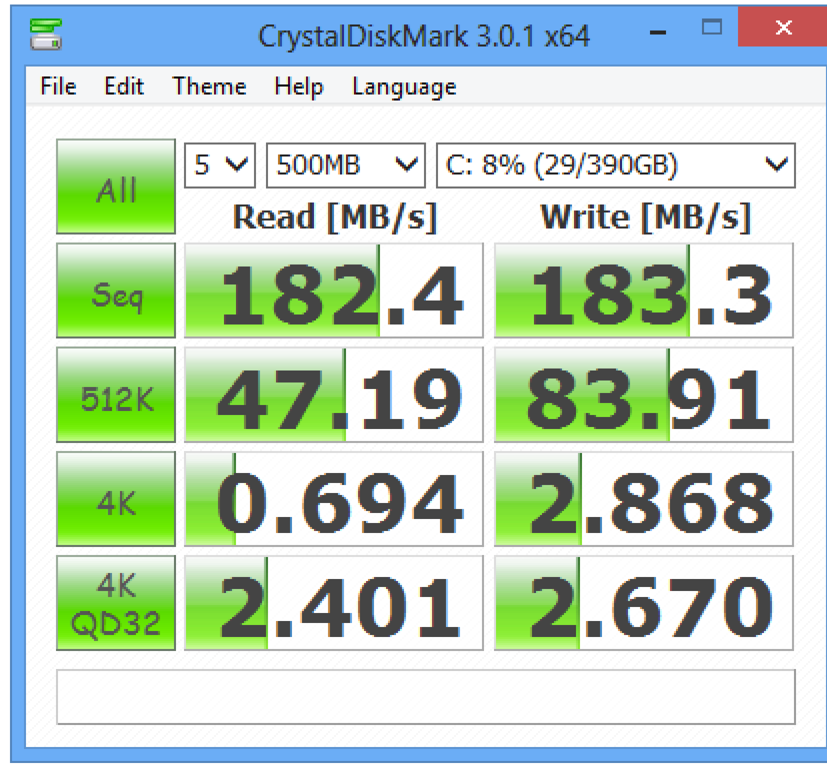
Storage: HDD

- Use a solid state USB drive for cache
  - →Benefit is low latency, not high bandwidth
  - →Windows ReadyBoost (available by default)
    - Random access is 10x faster on flash than hdd
    - http://www.7tutorials.com/files/img/readyboost_performance/readyboost_performance14.png
    - If you aren't already using a device for caching, and the new device is between 256MB and 32GB in size, has a transfer rate of 2.5MB/s or higher for random 4KB reads, and has a transfer rate of 1.75MB/s or higher for random 512KB write
      - https://technet.microsoft.com/en-us/magazine/2007.03.vistakernel.aspx
  - →Linux >3.10 bache / zfs l2arc
    - 12.2K random io/sec -> 18.5K/sec with bcache, 50% increase
      - http://bcache.evilpiepirate.org/

Host Configuration



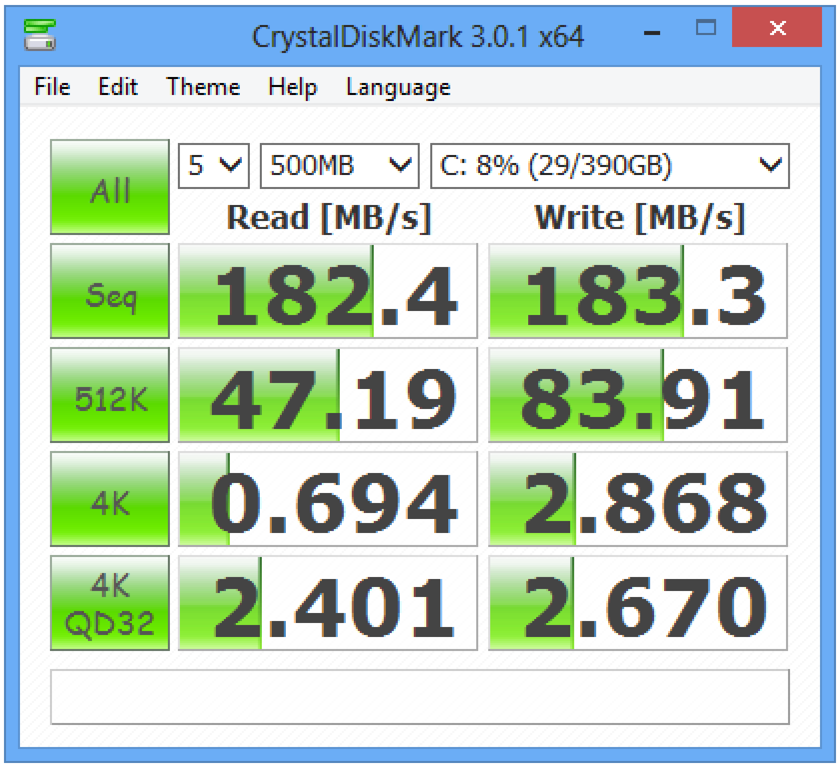
Standard HDD



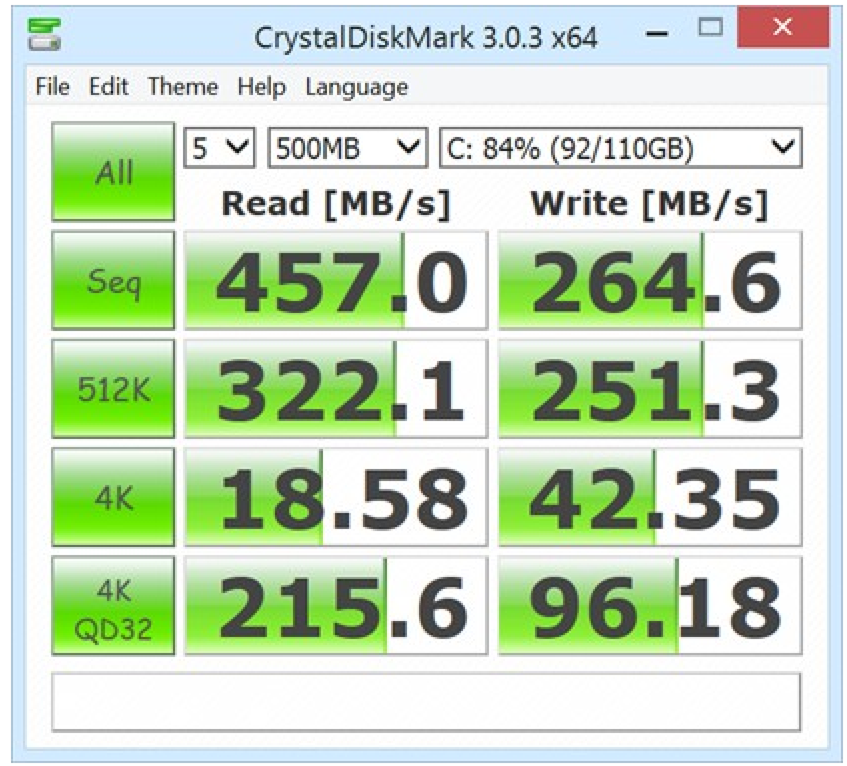
Raid 0

Storage: SSD

- Major performance gains over HDD



Raid 0

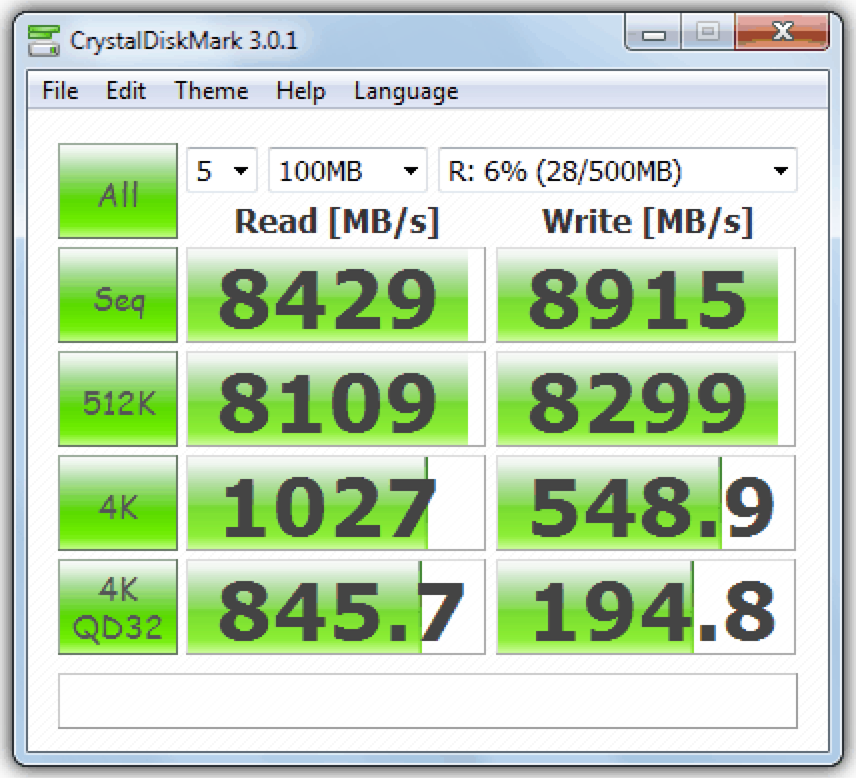


SSD

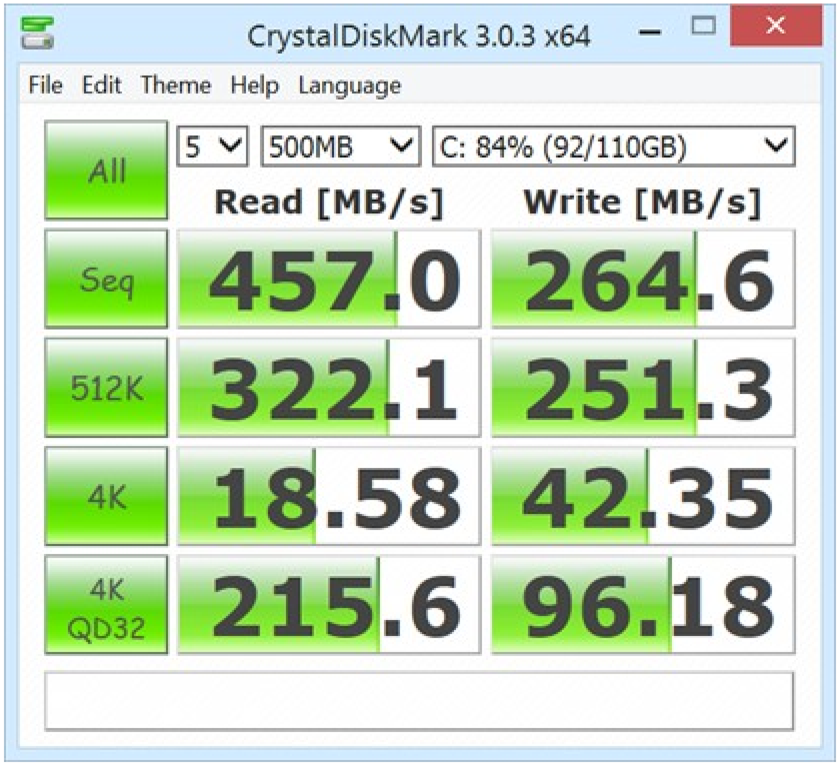
Storage: Ram Disk

- Much faster than SSD, eliminates fragmentation
  → http://superuser.com/questions/686378/can-ssd-raid-be-faster-than-ramdisk (10GB/s - 17GB/s)
- Linux - built in
  →ramfs or tmpfs
- Windows - 3rd party
  →High amount of variance
  - https://www.raymond.cc/blog/12-ram-disk-software-benchmarked-for-fastest-read-and-write-speed/

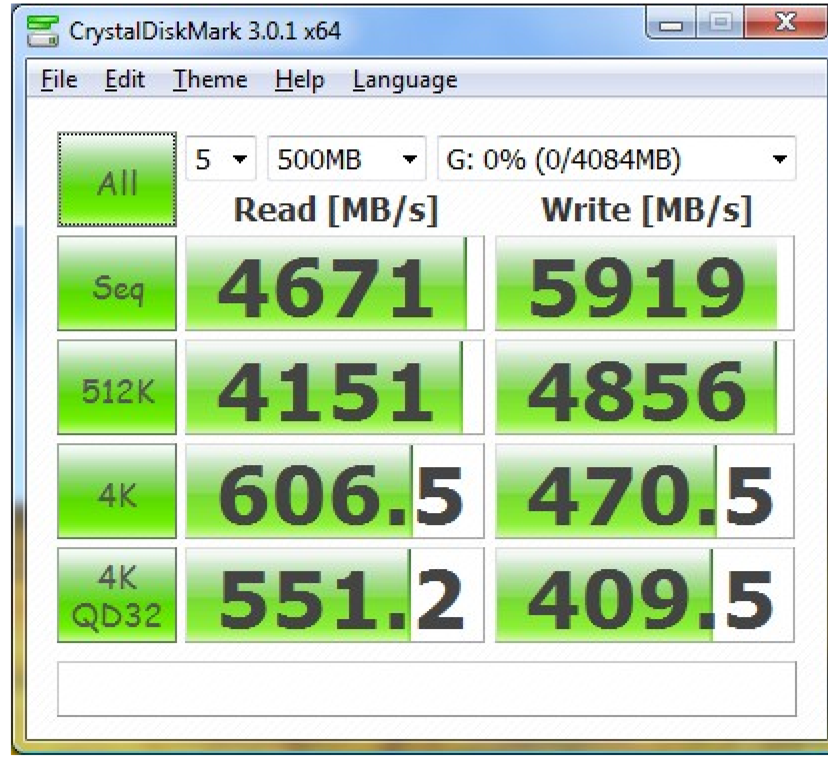
  →SoftPerfect RamDisk is winner for free software
  - https://www.softperfect.com/products/ramdisk/



Host Configuration



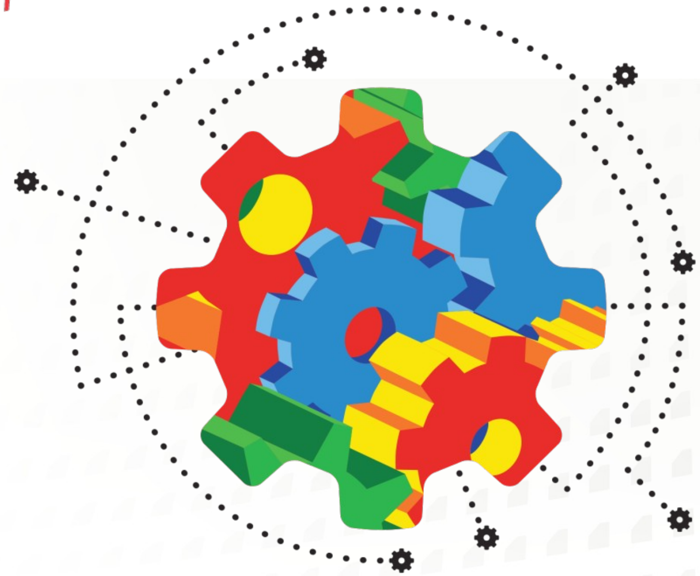
SSD



Ramdisk

Memory

- 32-bit memory limits
  →Linux - built in to PAE kernels
  →Windows
    - Limited based on SKU of your OS
    - Driver compatibility is the claimed reasoning
      – http://blogs.technet.com/b/markrussinovich/archive/2008/07/21/3092070.aspx
    - kernel patching required
      – http://www.geoffchappell.com/notes/windows/license/memory.htm
      – http://news.saferbytes.it/analisi/2012/08/x86-4gb-memory-limit-from-a-technical-perspective/
      – http://news.saferbytes.it/analisi/2013/02/saferbytes-x86-memory-bootkit-new-updated-build-is-out/



Conclusions

Thank You!

- Cisco Talos VulnDev Team
  - →Richard Johnson
    - rjohnson@sourcefire.com
    - @richinseattle
  - →Marcin Noga
  - →Yves Younan
  - →Piotr Bania
  - →Aleksandar Nikolic
  - →Ali Rizvi-Santiago