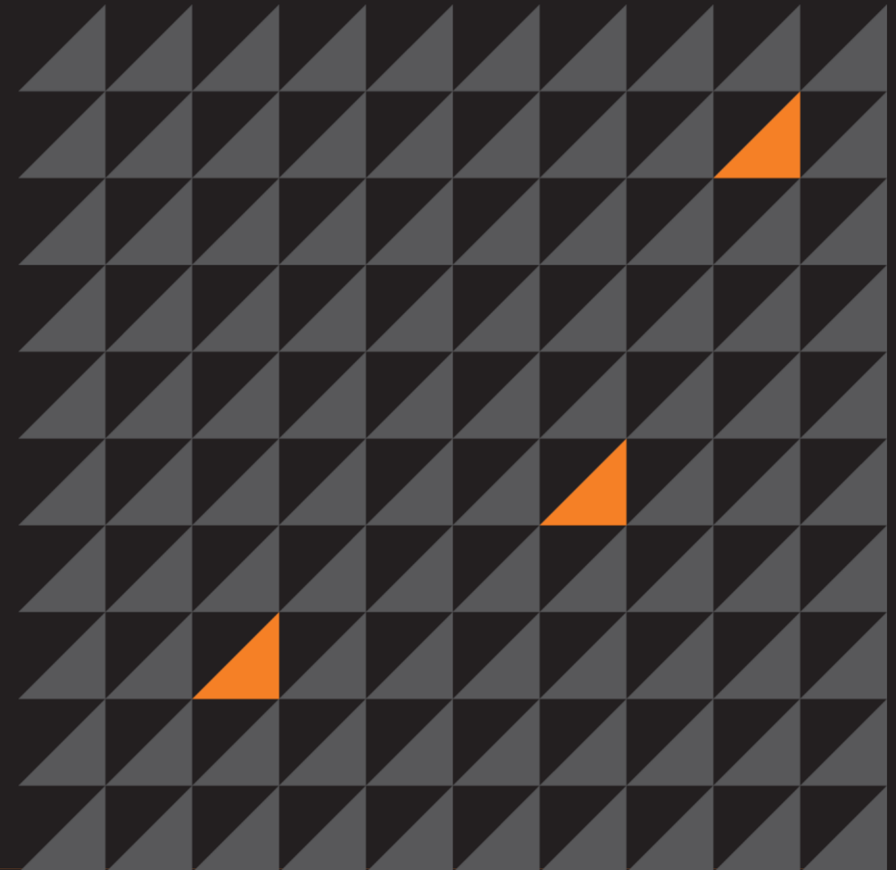
# Windows Kernel Fuzzing

Nils

T2 InfoSec

29th October 2015



# Agenda

- Introduction
- Architecture & Implementation
    - Fuzzer
    - Manager
- Tips and Tricks
- Results

# Introduction - About me

- Nils (@nils)
- Security Researcher at MWR
  - Since 2009

- Offensive research for defensive purposes
- Previous Research
  - Android, Chip&Pin, Browsers, Kernels

- Director of bytegeist GmbH
  - An MWR Company
  - Highly specialised security research projects

MWR LABS

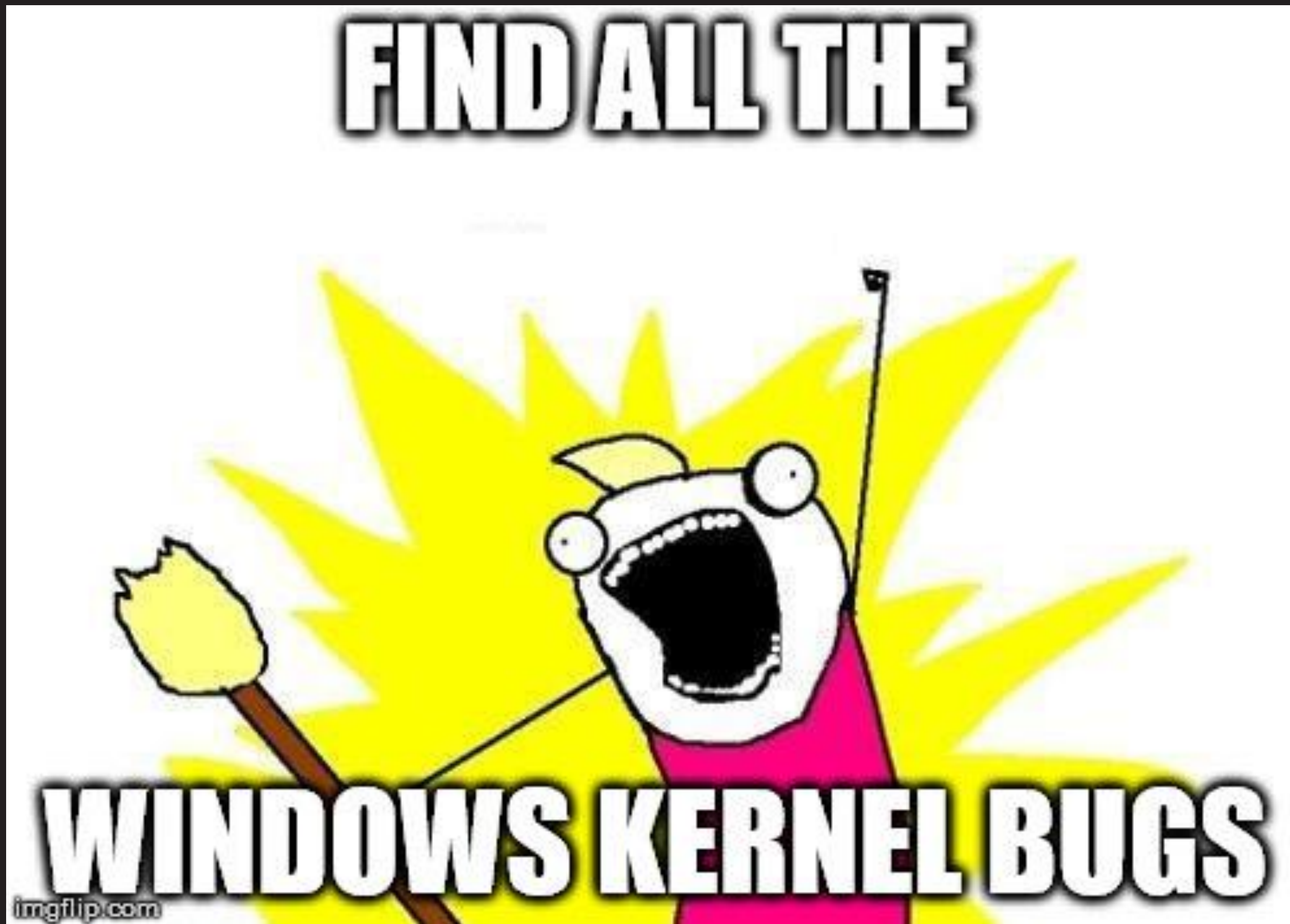
## Introduction - Motivation

- Local Privilege Escalation
  - Part of any serious in-the-wild attack
  - e.g. Sandbox breakout

- We used a win32k buffer overflow at pwn2own 2013
  - Wrote a fuzzer to find this
  - Many limitations (e.g. no repros)
- Google Project Zero provided funding:
  - Further develop the fuzzer
  - Run it at scale

MWR LABS

# Introduction - Goals



# Introduction - Goals

- Find many Windows Kernel Vulnerabilities
  – And get them fixed

- Hopefully increase cost for attackers

- And learn stuff
  – Exploitation
  – Potential Mitigations

MWR LABS

## What are we trying to find?

- CVE-2015-1701
  - win32k UAF used by APT28 (Fireeye)
  - ClientCopyImage user-mode callback
- CVE-2015-2546
  - win32k tagPOPUPMENU Use-After-Free
  - In the wild attacks (Fireeye)
- CVE-2014-4113 - win32k memory corruption
  - xxxMNFindWindowFromPoint
  - used in targeted attacks (Trend Micro)

# The Plan: To implement a Windows Kernel Fuzzer

- Fuzzes on the current state
- Logs testcases
  - Reproducible and minimisable
- Extensible and modular
  - Core kernel, win32k and other drivers
- Is able to run automated at a large scale
- And most importantly finds a lot of vulns

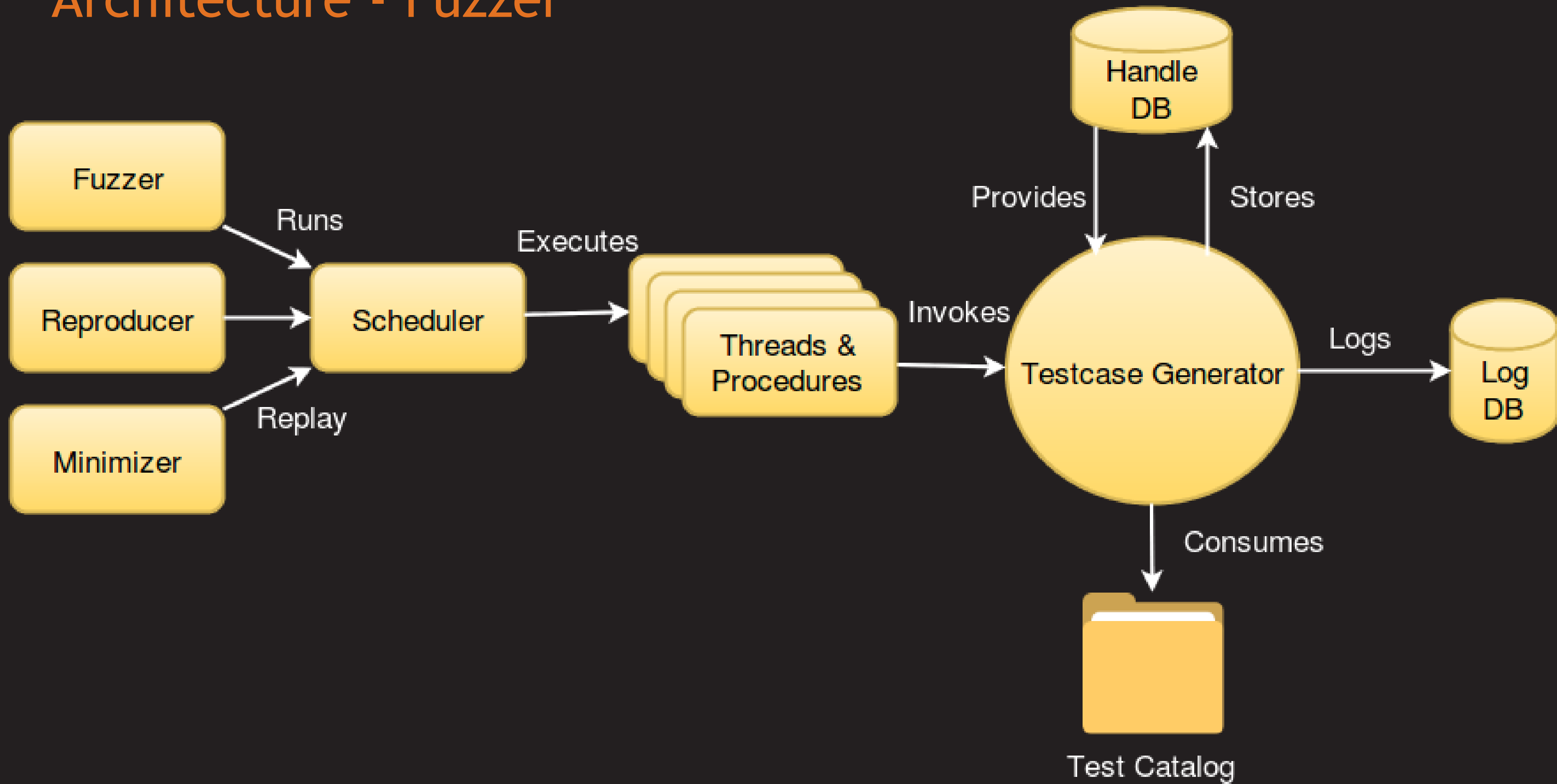
- Many implementation ideas borrowed from browser fuzzing

MWR LABS

# Fuzzer

## Design and Implementation

# Architecture - Fuzzer



MWR LABS

# Implementation

- Everything implemented in Python
- Extensive use of ctypes
  - Dirty^H custom assembler for system calls

- Allows for rapid developed and extension

## Test Catalog

- Stored knowledge on how to interact with Kernel
  – Programmatically
  – From Reversing, Googling, MSDN, ReactOS, etc
- E.g. System and Library calls
  – Arguments, return values
- Most of the work went into developing this

- We could just fire random system call # and arguments
  – Unlikely to get a good coverage
  – Even at scale

MWR LABS

## Test Catalog - Implementation

- Each test is a Python class implementing:
  - generate_arguments()
    Generates random arguments for the current test
  - run()
    Executes the current test using arguments

# Test Catalog – Example – Long Form

```
class GDI32_CreateSolidBrush(TestCase):
    def generate_arguments(self):
        self.args=[]
        color = arguments.HexArg(self.fuzzer.R(0xffffffff))
        self.args.append(color)
        return True

    def run(self):
        rv = ntypes.gdi32.CreateSolidBrush(self.args[0].value)
        self.addhandle("hbrush", rv)
```

# Test Catalog – Example – Short Form

```
class GDI32_CreateSolidBrush(SimpleTestCase):
    function = ntypes.gdi32.CreateSolidBrush
    arguments = [[SimpleTestCase.randomhexarg, 0xffffffff]]
    returnhandle = "hbrush"
```

# Test Catalog – Example – Complex

```
class COMPLEX_NotepadWindow(Testcase):
    def generatearguments(self):
        self.args = []
        name = "n" + hex(self.dr.R(0x7fffffff))
        self.stringarg(name)
        return True

    def run(self):
        open(self.args[0].value, "w").close()
        ps = subprocess.Popen(["c:\\windows\\system32\\notepad.exe",
        windowname = self.args[0].value + " - Notepad"
        time.sleep(1)
        hwnd = ntypes.user32.FindWindowA(0, windowname)
        self.addhandle("hwnd", hwnd)
        hmenu = ntypes.user32.GetMenu(hwnd)
        self.addhandle("hmenu", hmenu)
        self.addhandle("pid", ps.pid)
```

# Handle DB

Handles are references to objects in the Kernel

Returned by system calls

Consumed as arguments to system calls

Handle DB stores returned handles and provided random handles to tests

MWR LABS

## Fuzzing Run

1. Fuzzer selects random test from catalog

2. Generates arguments by calling generate_arguments()

3. Serialises arguments according to type
   ( numbers, strings, handles, return buffer )

4. Logs test name, arguments, procedure and thread

5. Executes test by calling run()

# Procedures

- Execution of tests in different contexts
  – Threads, Callbacks, Window Procedures

- “Kernel Attacks through User-Mode Callbacks”
  – Excellent paper by Tarjei Mandt

## Procedures – Example Window Proc

- Python wrapper functions
  – Set current function name
  – Execute main fuzzing loop
  – Pseudo code:

```
wrapper = functools.partial(wnd_proc,fuzzer,name)
wndclass.lpfnWndProc = ntypes.WNDPROC(wrapper)

def wnd_proc(fuzzer,name,hwnd,msg, lparam,wparam):
    oldfunc_name = fuzzer.get_func_name()
    fuzzer.set_func_name(name)
    fuzzer.fuzz()
    fuzzer.set_func_name(oldfunc_name)
```

MWR LABS

## Procedures – User Mode Callbacks

- Introducing "Bambi" the hooker
  - Hooks user-mode callbacks
- Again implemented as test
  - Hooks *only* the next execution
  - Unhooks automatically
- Some ctypes hacks for hooking
  - Small basic assembler for trampolines

- API:

```
bambi.hook(index, function)
```

# Threads

- Threads are just a special case of procedures
  - executed in run() of threading.Thread

- Storage of current thread and function name in TLS
  - Retrieved by the logger

# Logging

```
t0:main:SC_NtGdiCreateMetafileDC(H[0x0])
t0:main:rc => HANDLE[ID{o0}:0x2d2108c1]
t0:main:GDI32_CreateSolidBrush(H[0xde9c7010L])
t0:main:rc => HANDLE[ID{o1}:0x91008d5]
t0:main:SC_NtGdiCreateMetafileDC(HANDLE[ID{o0}:0x2d2108c1])
t0:main:rc => HANDLE[ID{o2}:0x1121088c]
t0:main:SC_NtGdiSelectPen(HANDLE[ID{o2}:0x1121088c],…
t0:main:User32_CreateMenu()
t0:main:rc => HANDLE[ID{o3}:0x60227]
t0:main:User32_AppendMenuString(HANDLE[ID{o3}:0x60227],H[0x2]
,HANDLE[ID{o3}:0x60227],S['m'])
```

## Reproducing Testcases

- Parsing the logs:

```
t0:main:SC_NtGdiCreateMetafileDC(H[0x0])
t0:main:rc => HANDLE[ID{o0}:0x2d2108c1]
```

- Thread Name
- Function Name
- Test Name
- Arguments

- Potentially return value(s)

## Reproducing Testcases

- We get:

```
threads = {
      "t0": {
            "function": [
                  ("testname", [arg1, arg2,…]),
                  …
            ],
            …
      }
}
```

## Fuzzing Run - Reminder

1. Fuzzer selects random test from catalog

2. Generates arguments by calling generate_arguments()

3. Serialises arguments according to type
   ( numbers, strings, handles, return buffer )

4. Logs test name, arguments, procedure and thread

5. Executes test by calling run()

# Repro Run

1. Fuzzer selects current test for thread & function

2. Fuzzer selects arguments for current test

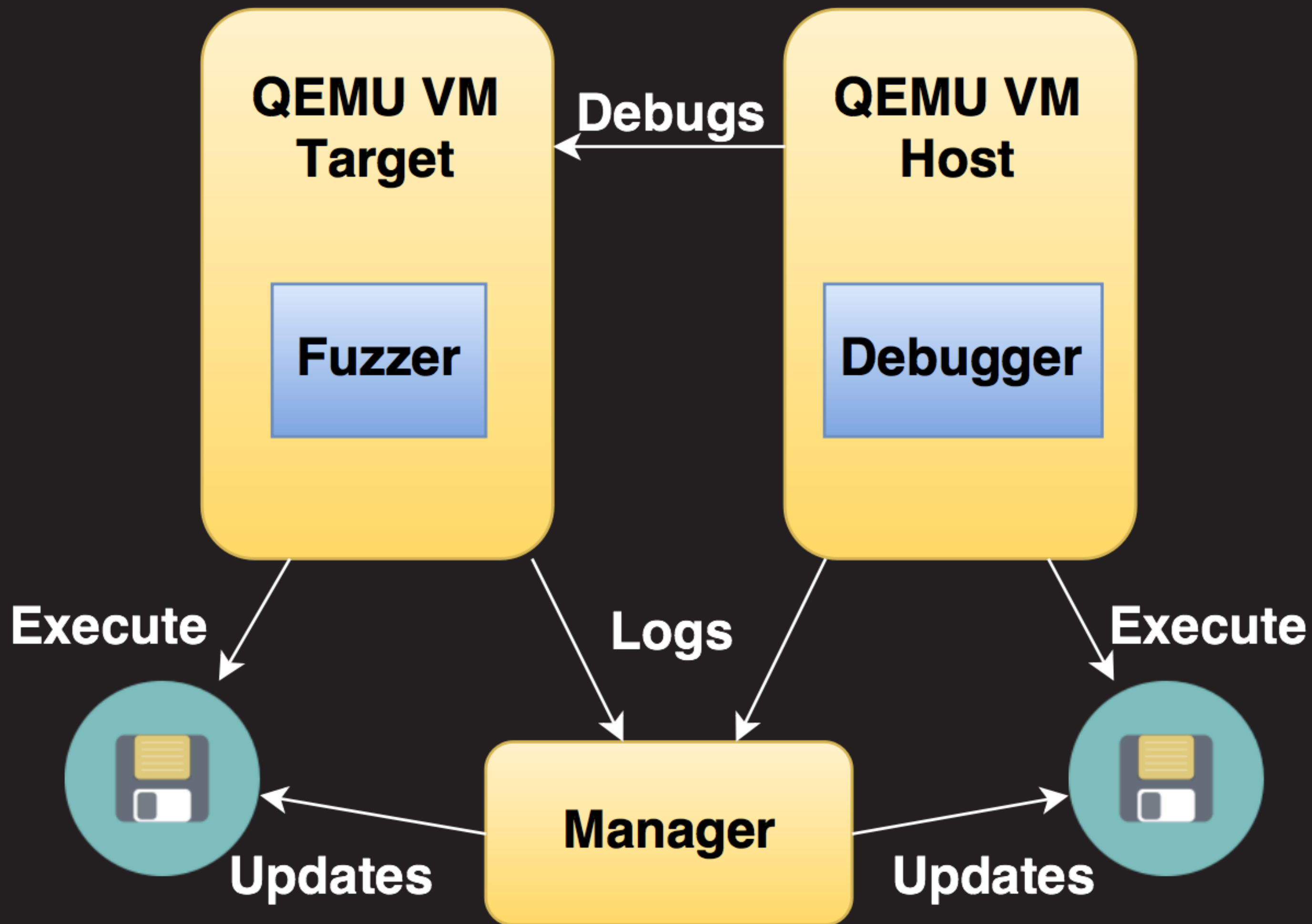
3. Executes test by calling run()

# Manager

## Design and Implementation

MWR LABS

# Architecture - Manager



## Manager – Example Run

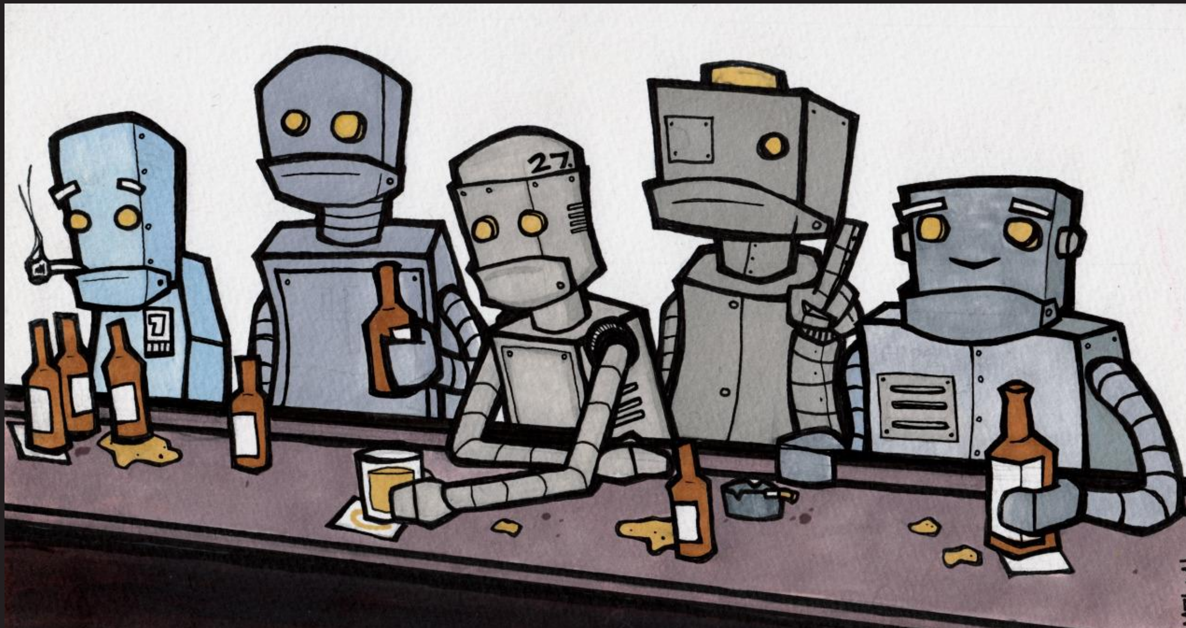
1. Start the debugger VM
2. Wait for "Waiting to reconnect…"
3. Start the target VM
4. Record testcase until crash or stall
5. In case of crash: Store debugger output and testcases
6. Kill VM's and start over at 1

# Manager – Scaling it up

- First Option: Bare metal

+ Good Performance

- High upfront cost
- Not very flexible
- Loud …



## Manager – Scaling it up

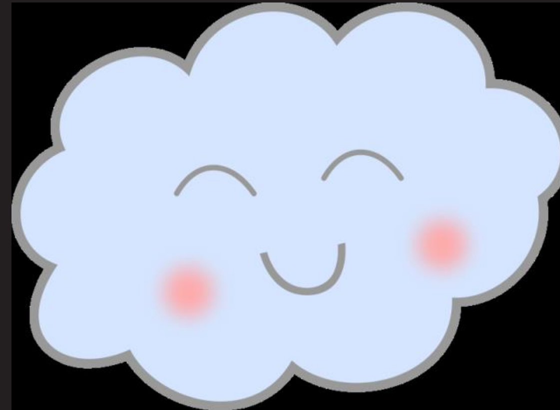
- Second option: To the cloud

+ Flexible
+ Cheap (Spot instances/Preemptible)
+ No fixed costs

- However, there is one problem:

# MULTIPLE LAYERS OF VIRTUALISATION

## Manager – Scaling it up



- We can make it work:
- QEMU/TCG
- ~10x slow down
  - We can scale against that
  - Just click that scale button at your cloud provider

- No x64 currently

# Cloud – Costs ¯_(ツ)_/¯

## Example: First week of October:

| | | |
|---|---|---|
| c4.8xlarge Linux/UNIX Spot Instance-hour in US East (Virginia) in VPC Zone #5 | 587 Hrs | $170.38 |
| m4.2xlarge Linux/UNIX Spot Instance-hour in US East (Virginia) in VPC Zone #6 | 9,807 Hrs | $522.86 |

99.588 CPU(Core) Hours

MWR LABS

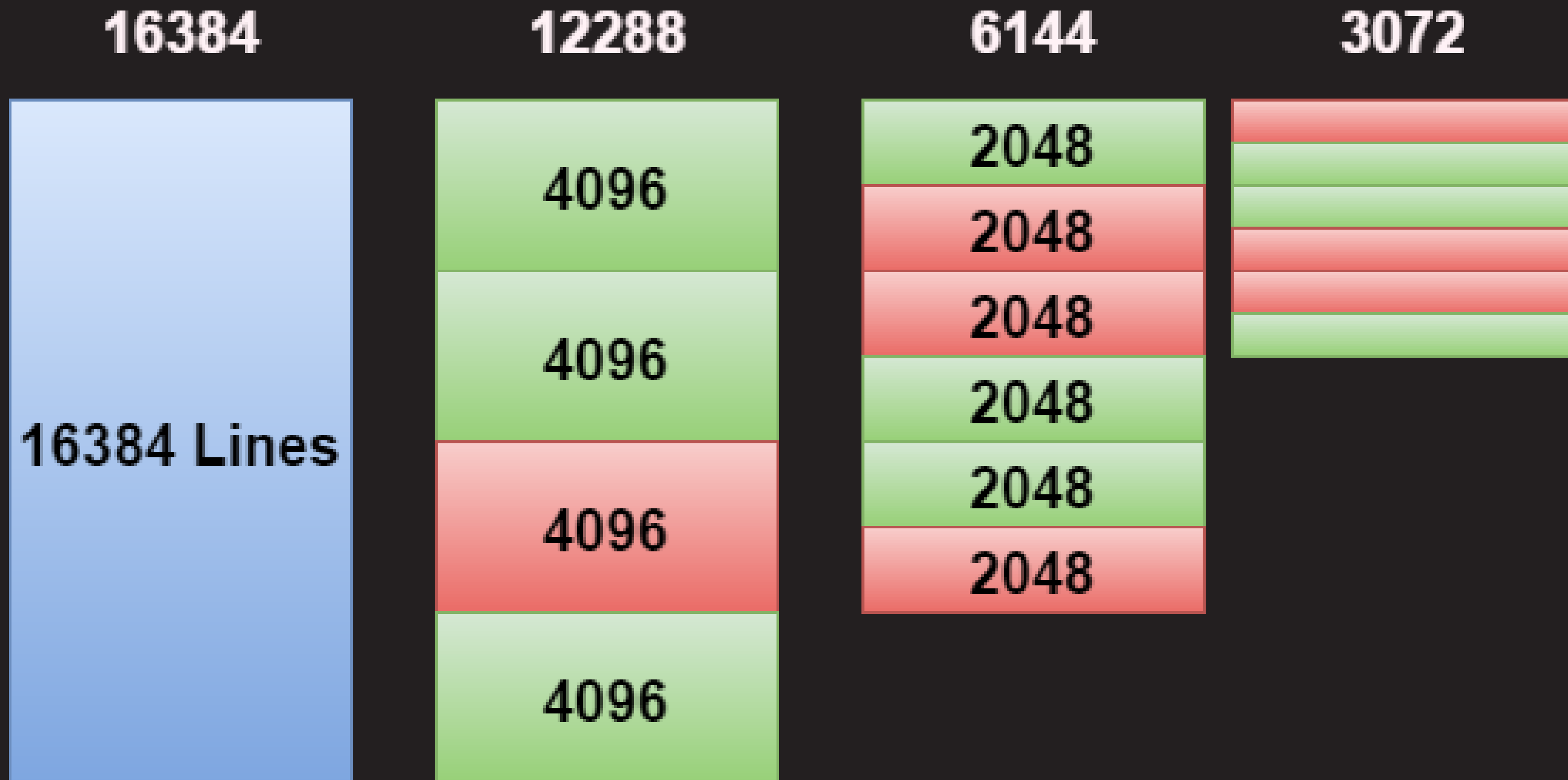
## Minimising Testcases

- Testcase
  - Ordered set of lines (tests)
  - Often > 10k , sometimes >100k

- Remove line by line
  - Not crashing => Line essential for testcase
  - Otherwise remove line

- Divide and Conquer
  - Remove blocks instead of lines and reduce blocksize

# Minimising Testcases – Divide & Conquer




- After 39 execution down to 3 lines ( very “friendly” case )

# Minimising Testcases – Example – 14 Lines

**CVE-2015-1726: win32k use-after-free in HmgAllocateObjectAttr**
**https://code.google.com/p/google-security-research/issues/detail?id=320**

```
t0:main:SC_NtGdiCreateHatchBrushInternal(H[0x4],H[0xb4],H[0x1])
t0:main:rc => HANDLE[ID{o8}:0x1410022f]
t0:main:NtGdiSetBrushAttributes(HANDLE[ID{o8}:0x1410022f],H[0x1])
t0:main:rc => HANDLE[ID{o10}:0x1490022f]
t0:main:SC_NtGdiClearBrushAttrs(HANDLE[ID{o10}:0x1490022f],H[0x1])
t0:main:rc => HANDLE[ID{o16}:0x1410022f]
t0:main:NtGdiSetBrushAttributes(HANDLE[ID{o8}:0x1410022f],H[0x1])
t0:main:SC_NtGdiClearBrushAttrs(HANDLE[ID{o10}:0x1490022f],H[0x1])
t0:main:SC_NtGdiDeleteObjectAppBrush(HANDLE[ID{o16}:0x1410022f])
t0:main:SC_NtGdiCreateHatchBrushInternal(H[0x7],H[0x72],H[0x1])
t0:main:rc => HANDLE[ID{o87}:0x81006c7]
t0:main:NtGdiSetBrushAttributes(HANDLE[ID{o87}:0x81006c7],H[0x1])
t0:main:rc => HANDLE[ID{o119}:0x89006c7]
t0:main:SC_NtGdiClearBrushAttrs(HANDLE[ID{o119}:0x89006c7],H[0x1])
```

MWR LABS

## Distributed Minimising

- Divide and Conquer not great for running in parallel
  - Especially with fuzzing testcases

- The execution of the testcase only takes a few seconds
  - Small testcase less than a second

- Distributed Minimiser
  - Starts up VMs (Debugger/Targets)
  - Upload testcases through serial port & execute
  - D&C still sequential, substantially faster
  - Implementation uses ZeroMQ PUSH/PULL

# Tips&Tricks

# Special Pool

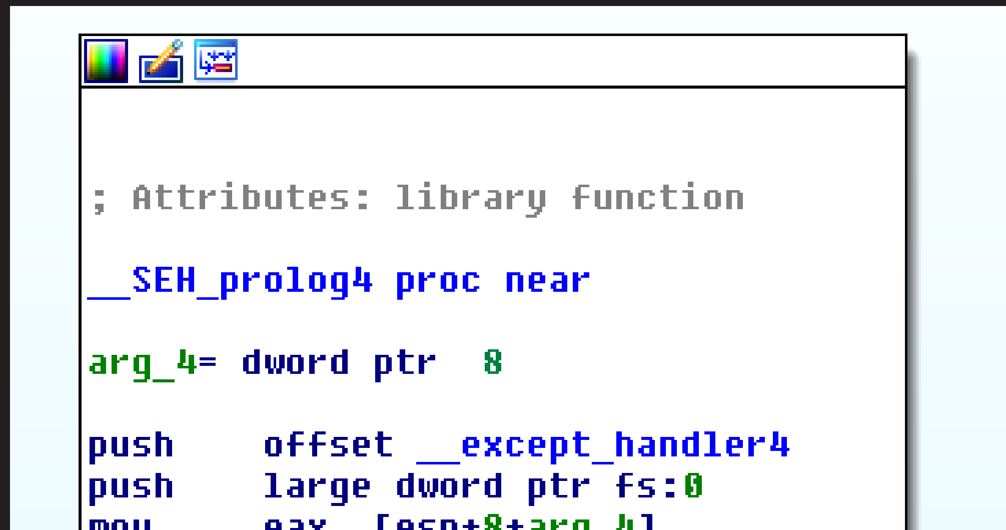
- Page heap or AddressSanitizer for kernel drivers
- Detects buffer issues and UAF’s among other things

Output:

```
DRIVER_PAGE_FAULT_IN_FREED_SPECIAL_POOL (d5)
Memory was referenced after it was freed.
This cannot be protected by try-except.
When possible, the guilty driver's name (Unicode string) …
the bugcheck screen and saved in KiBugCheckDriver.
Arguments:
Arg1: fa85efa4, memory referenced
Arg2: 00000001, value 0 = read operation, 1 = write operation
Arg3: 94596f21, if non-zero, the address which referenced memory
Arg4: 00000000, (reserved)
```

## Exception Handlers

- Most system calls start with call to __SEH_prolog4
- Basically wrapping everything in a try {} catch () {}
  - Even for access violations etc.
  - Return from system call on error
  - Masking many bugs



MWR LABS

## Exception Handlers - DISCLAIMER

- There are probably better solutions
  - e.g. https://code.google.com/p/ioctlfuzzer

- Turn away if you are easily offended by dirty hacks :P

- Ready?

MWR LABS

# Exception Handlers - Hack

On load of win32k:

eb win32k!_SEH_prolog4 68 ef be ad de
eb win32k!_SEH_prolog4_GS 68 ef be ad de

Replaces:
**push offset __except_handler_4**
With:
**push 0xdeadbeef**

!analyze –v still successful  \o/

MWR LABS

## Fuzzing with Minidumps

- Configure Windows to store Minidump
  - On virtual hard drive
  - Virtual hard drive not a snapshot
  - Can be retrieved by Manager

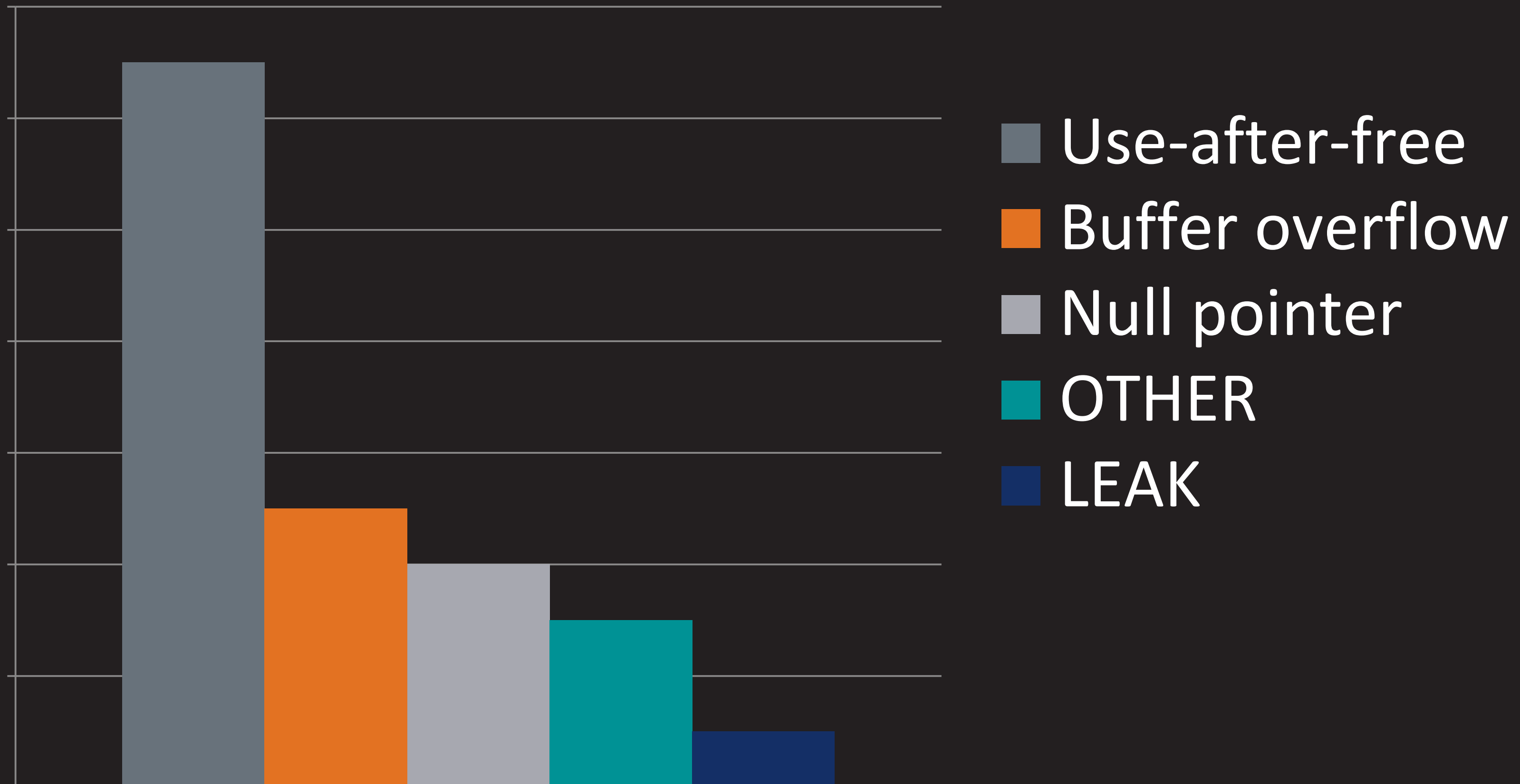
- Requires reboot
  - Minidump copied on reboot

- No real speed improvement
  - However more VM’s per Memory

# Does it work?

MWR LABS

# Results – 26 Bugs reported so far




https://code.google.com/p/google-security-research/issues/list?can=1&q=label%3AFinder-nils

# Conclusion

- It works
  - Finding and minimising bugs automated process
  - Had several runs on hundreds of cores
  - More bugs in the pipeline
- Still many more bugs to find
- All bugs also affected the latest Windows versions
  - Windows 8.1 or 10

# Future Work

- Open source the fuzzer
  - Once it stops finding bugs

- Continue adding tests
  - More Nt*
  - Ioctl's
  - Other drivers: afd.sys, DirectX and many more

- More runs at scale
  - It keeps finding new bugs without changes

A problem has been detected and Windows has been shut down to prevent damage to your computer.

ANY_QUESTIONS_PLEASE_ASK

Technical Information:

*** STOP: 0xHAMMERTIME (0xC000FFEE, 0xDEADBEEF)