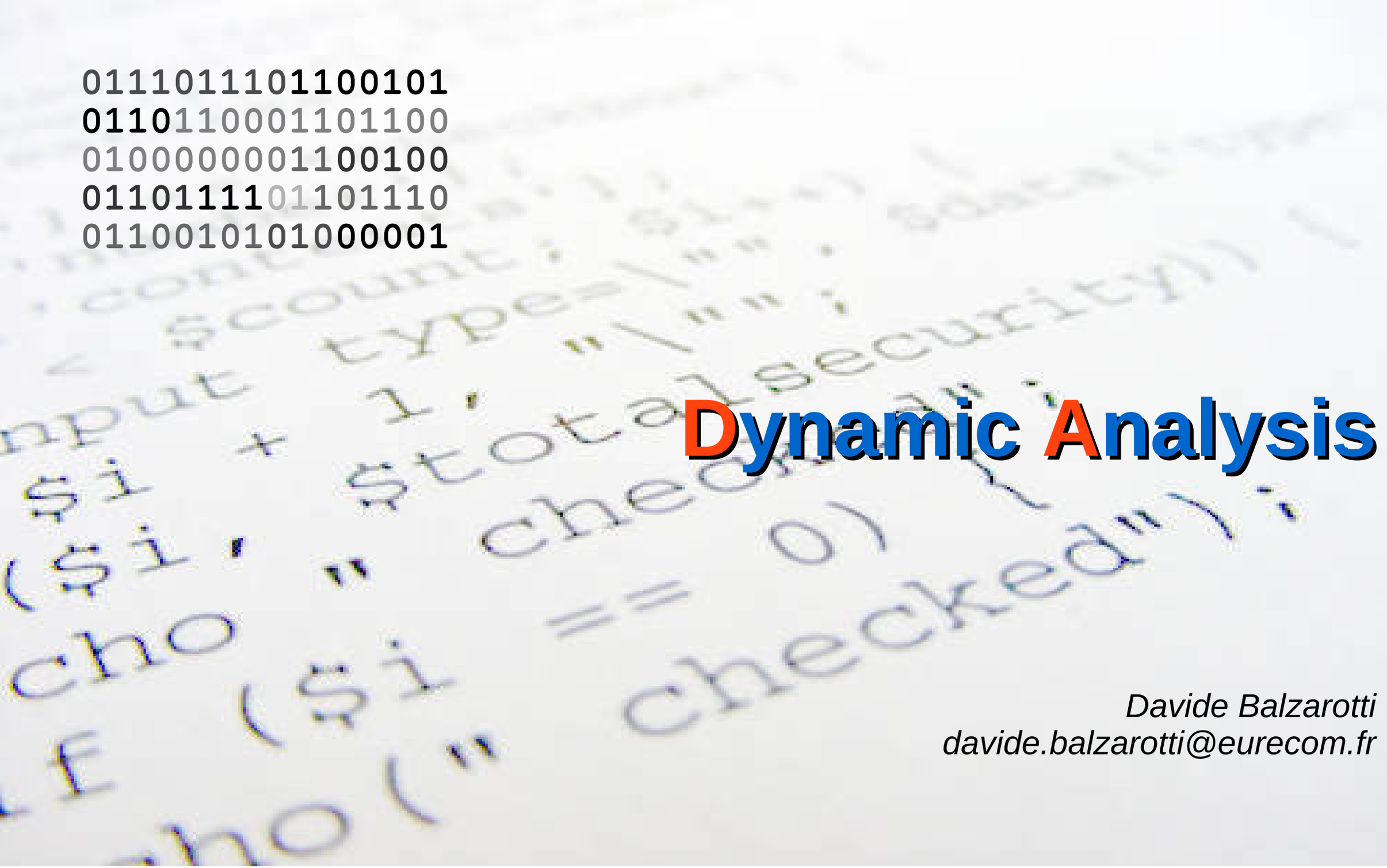
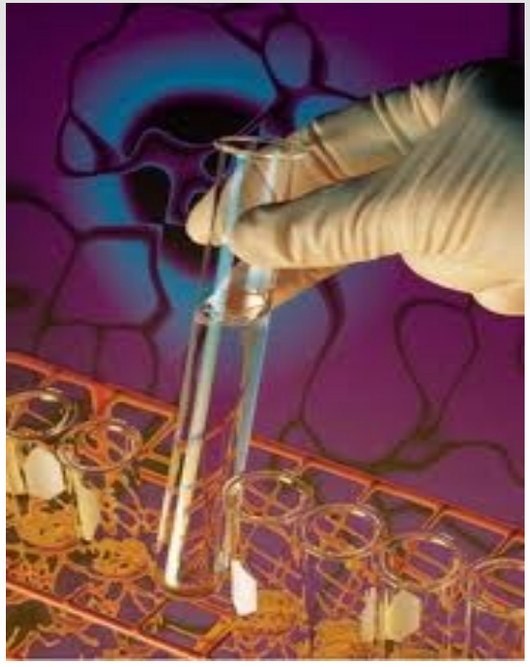


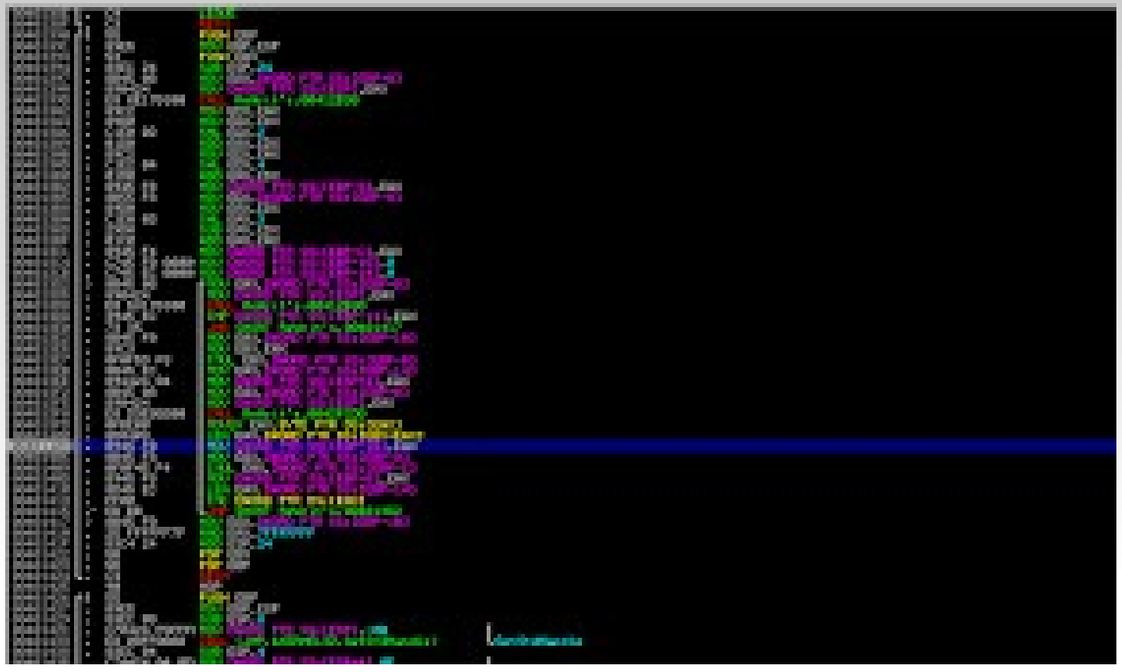
Dynamic Analysis

Davide Balzarotti
davide.balzarotti@eurecom.fr

Summary

- Fine-Grain Dynamic Analysis
  - Tracing
  - Debugging
  - Anti-Debugging
  - Record/Replay
- Large Scale Malware Analysis





Tracing

Syscall Tracing

- `strace` - linux command-line dynamic analysis tool to monitor the execution of a process and reports
  - The system calls invoked by the process
  - The signals received by the process
- `WUSSTrace` – Windows userspace tracer
  - Injects a shared library in the address space of the traced process and hooks the `KiFastSystemCall` and `KiIntSystemCall` stubs
  - Supports the majority of argument types
  - Produces easy-to-parse XML traces

strace

- Useful options
  - `-f` trace forked processes
  - `-i` print address of the instruction invoking the syscall
  - `-v` print all info
  - `-x` print non-ascii strings in hex
  - `-e read=[set]` print all the data read from file descriptors
  - `-e write=[set]` print all the data written to file descriptors
  - `-ff -o filename` save the trace to a file
  - `-s 100` print 100 characters for each string (default is 32)

Library Call Tracing

- `ltrace` - linux command-line dynamic analysis tool to monitor the execution of a process
  - Logs the library calls (and their parameters) invoked by the process
  - Works by setting a breakpoint to each symbol it needs to track (basically, it works like a little automatic debugger..)

- Windows
  - `Process Monitor` from SysInternals reports disk and registry activity of any running process
  - Several programs exist to intercept library calls (such as API Monitor)

OS Level Instrumentation

- Log syscall in Linux with `systemtap`
  - Sytstemtap provides a way to gather information about a running Linux kernel or userland application
  - The administrator/analyst writes scripts that are then automatically translated to C code, compiled, and loaded as kernel modules
  - It can also tap into userspace libraries
    - Requires some instrumentation inserted in the source code
    - Support `libc` out of the box

Instruction Tracing

- Collect the sequence of all the instructions (and potentially their parameters) executed by a program
- It is the most precise tracing information we can collect, which allows for fine-grained instruction analysis (e.g., for VM detection)
- BUT…
  - It is difficult to collect (it requires an emulator)
  - It is extremely slow to collect (several order of magnitude slow-down)
  - It is extremely verbose to store (GB for a single sample)

Debugging

Debuggers

- Debuggers are software tools designed to perform precise, manual dynamic analysis of a running process
- Originally designed to investigate bugs
- In security, two main use cases
  - To study a vulnerability and develop an exploit

    → there is still a bug to analyze
  - To analyze the behavior of a program as part of a reverse engineering process

    → there is no bug

Debugging for Reverse Engineers

- Traditional debugging
  - Normally performed by a person that understands the code
  - Source code and debugging symbols are typically available
  - The program is not trying to “hide” its behavior

Debugging for Reverse Engineers

- Traditional debugging
  - Normally performed by a person that understands the code
  - Source code and debugging symbols are typically available
  - The program is not trying to “hide” its behavior
- RE Debugging
  - Very little information available about the behavior of the program
  - No source code and debugging symbols
  - Adversarial environment: the program is often designed to make reverse engineering hard
- Different problems that share the same tool: the debugger

Debuggers

- GDB
  - Standard in Linux/BSD environments
  - Good debugger, but not designed for security purposes
- WinDbg
  - Developed by Microsoft
  - Comes for free in the “Debugging Tools for Windows” package
  - Rich features by very poor interface
  - Can be extended in C++

Debuggers

- OllyDbg
  - Recursive-traversal dissassembler
  - Nice interface
  - Plenty of security-related plugins
- Immunity Debugger
  - Modified version of Ollydbg especially designed for security
  - Scriptable, and extensible in Python
  - Command-line with windbg-aliased commands

GDB: the GNU Debugger

- The standard debugger for the GNU software system and the main debugger under linux/bsd systems
  - First written by Richard Stallman in 1986
- GDB provides an interactive environment with a command line interpreter
- GDB can also be used in batch mode
  - `gdb -batch -x cmd_file <program_name>`
  - Exit with status 0 after processing all the command specified in cmd_file
  - Exit with nonzero status if an error occurs in executing gdb

Controlling the Program Execution

(gdb) run [parameters]

- Execute the program with the specified parameters
- Without arguments, it re-uses the same arguments used in the previous run (!!)

(gdb) stepi [n]

- Execute N *machine* instructions, then stop and return to the debugger

(gdb) nexti [n]

- Like stepi, but it steps over function calls

(gdb) continue

- Resume the program execution

(gdb) finish

- Execute until the select stack frame returns

Breakpoints

- There are three main ways to specify conditions that determine when a program's execution should be interrupted
  - *Instruction breakpoints* specify an instruction (by its address) and stop the program before executing it
  - *Watchpoints* instructs the debugger to stop the program every time it access a specifying memory address
  - Catchpoints stops the program execution whenever a particular event occurs

Managing Breakpoints

(gdb) break <location>

- Set a breakpoint to a given location
  (the program will stop before the instruction at that location is executed)

(gdb) info <breakpoints>

- Show the list of breakpoints

(gdb) delete breakpoint

- Delete the breakpoints, watchpoints, or catchpoints specified as arguments.
- If no argument is specified, delete them all

(gdb) disable breakpoint
(gdb) enable breakpoint

- Disable (or re-enable again) a break/watch/catch point

Software Breakpoints

- Break the execution of the program at a specific address
- Implementation
  - The debugger saves the first byte of the target address and replace it with an INT 3 instruction (`0xcc`)
  - When the instruction is executed, the debugger traps the exception and replaces the trap with its original byte
  - When the user continues the execution, the debugger steps to the next instruction, restore the breakpoint, and the continues the program
- Characteristics
  - Can be created in an unlimited number
  - They require the debugger to change the program code (impossible for self-modifying code)
  - The INT 3 instruction can be detected

Hardware Breakpoints

- Use CPU specific registers to implement a breakpoint in hardware
  - The target address is set in one of the debugger registers (`DR0` through `DR3` on Intel CPUs)
  - The type of breakpoint is configured in `DR7`
    - Break on execution, break on read, break on read/write
    - Length of data item to be monitored (1, 2 or 4 bytes)
  - When the address on the bus matches those stored in the debug registers, a breakpoint signal, interrupt one (INT 1), is sent and the CPU halts the process
  - `DR6` is used as a status register to allow the debugger to know which debug register has triggered

```
(gdb) show breakpoint auto-hw
(gdb) set breakpoint auto-hw on
```

Watchpoints

- Stop the execution whenever the value of an expression changes, without having to predict a particular place where this may happen
  - The expression may be as simple as the value of a single variable, or as complex as many variables combined by operators
- Depending on the system, watchpoints may be implemented in software or hardware
  - GDB does software watchpointing by single-stepping your program and testing the variable's value each time, which is hundreds of times slower than normal execution
  - Watching complex expressions that reference many variables can quickly exhaust the resources available for hardware-assisted watchpoints because GDB needs to watch every variable in the expression with separately allocated resources
  - (gdb) show can-use-hw-watchpoints

Watchpoints

(gdb) watch <expression>

- Set a watchpoint for an expression
- GDB will break when the expression is written into by the program and its value changes.
- The most common form of expression is a program variable name

(gdb) rwatch <expression>

- Watchpoint that breaks when the expression is read by the program

(gdb) awatch <expression>

- Watchpoint that breaks when the expression is accessed by the program (either read or written)

Catchpoints

- Catchpoints tell the debugger to stop for certain kinds of program events, such as syscalls or the loading of a shared library

(gdb) catch fork

- Catch process creations

(gdb) catch syscall [which]

- Catch the invocation of (one particular) system calls
- The system call can be specified by name or by number
- Available in GDB 7.0

Break Conditions

- The simplest type of breakpoint breaks every time the program reaches a specified place
- GDB allows the user to specify additional conditions that must be true in order to stop the program
  - A condition is evaluated each time the program reaches it, and the program stops only if the condition is true

(gdb) break <location> if <expression>

(gdb) condition <bp_num> <expression>

- Set a conditional breakpoint or change the condition associated to an existing breakpoint
- Example:

```
(gdb) break *0x804be34 if $esp < 0xbfff0000
```

Breakpoint's Command List

- It is possible to specify a list of commands to execute when the program stops at a certain breakpoint
  - For example, you might want to print the values of certain expressions, or enable other breakpoints

```
(gdb) commands <breakpoint_num>
  command 1
  command 2
  ....
  end
(gdb)
```

- If the first command in a command list is silent, the usual message about stopping at a breakpoint is not printed.
  - This may be desirable for breakpoints that have to print a certain message and then continue

Breakpoint's Command List

- One application for breakpoint commands is to dynamically patch the program before continuing the execution
  - E.g., force a particular condition to happen so you can debug a particular execution path

```
(gdb) break *0xb7e55fbb
(gdb) commands
 > silent
 > set $eax=0
 > set $eflags=$eflags | 0x01
 > continue
 > end
```

Altering The Process State

- Modifying program data
  - Change the current of a register
    ```
    (gdb) set $eax = 0x10
    ```
  - Change the content of a memory location
    ```
    (gdb) set {int}0xbf830402 = 4
    ```
- Modifying program flow

  (gdb) jump <location>
  - Resume execution at the specified location
  - Equivalent to set $pc=location

  (gdb) return [value]
  - Discards the selected stack frame (and all frames within it)
  - It's like the corresponding function returned prematurely

For when You are Lost

(gdb) backtrace

- print a summary of all the frames all the way back to the *main* function

(gdb) set backtrace past-main

- Tell GDB to show also the frame of the functions before the main invocation

(gdb) x/100i [addr]

- Disassemble at the specified addr

(gdb) set disassembly-flavor intel|att

- Set the assembler syntax to intel or at&t (default)

Examining Memory

(gdb) x/nuf addr

- Examine the process memory
  - N - the repeat count. An integer (default is 1) that specifies how much memory (counting by units u) to display
  - U - the unit size
    - b: bytes
    - h: half-words (two bytes).
    - w: words (four bytes)
    - g: giant words (eight bytes).
  - F - the display format - Specify how the data must be interpreted (like the placeholders in the printf's format string)
    - x, d, u, t, c, f, s – hex, decimal, unsigned, binary, char, float, string
    - i - for machine instructions
- Example:

  (gdb) x/10wh *0xbffff567 – prints 10 words in hexadecimal

Other Useful Commands

(gdb) info registers
(gdb) info all-registers

- Print the content of (all) registers

(gdb) info proc map

- List all the memory regions

(gdb) dump binary memory [filename] [start] [end]

- Save a region of memory to disk

(gdb) info signals

- Print the signal setup

(gdb) handle [signal] [pass|nopass|stop|nostop]

- Tell GDB if it has to stop for a certain signal, and if it has to pass it back or not to the program

(gdb) signal [signal]

Scripting GDB with Python

- GDB 7.0 introduced the ability to run python code inside the GDB environment
- Easy-to-write python scripts can be used to enhance GDB and add / improve functionalities required for reverse engineering
  - E.g., search memory with regular expressions, trace instructions, compare memory, show fancy disassembly, ….
- You can write your own code, or use some already existing ones
  - PEDA (Python Exploit Development Assistance for GDB) is a good starting point

```
[----------------------------------registers-----------------------------------]
EAX: 0x2
EBX: 0xb7fc6ff4 --> 0x1a0d7c
ECX: 0xbffff3b4 --> 0xbffff519 ("/home/ldlong/workshop/stack1")
EDX: 0xbffff344 --> 0xb7fc6ff4 --> 0x1a0d7c
ESI: 0x0
EDI: 0x0
EBP: 0xbffff318 --> 0x0
ESP: 0xbffff318 --> 0x0
EIP: 0x80484e5 (<main+3>:       and    esp,0xfffffff0)
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80484e1 <getpath+109>:     ret
   0x80484e2 <main>:    push   ebp
   0x80484e3 <main+1>:  mov    ebp,esp
=> 0x80484e5 <main+3>:  and    esp,0xfffffff0
   0x80484e8 <main+6>:  sub    esp,0x10
   0x80484eb <main+9>:  cmp    DWORD PTR [ebp+0x8],0x0
   0x80484ef <main+13>: jg     0x804850a <main+40>
   0x80484f1 <main+15>: mov    eax,0x8048609
[-------------------------------------stack-------------------------------------]
0000| 0xbffff318 --> 0x0
0004| 0xbffff31c --> 0xb7e3f4d3 (<__libc_start_main+243>:       mov    DWORD PTR [esp],eax)
0008| 0xbffff320 --> 0x2
0012| 0xbffff324 --> 0xbffff3b4 --> 0xbffff519 ("/home/ldlong/workshop/stack1")
0016| 0xbffff328 --> 0xbffff3c0 --> 0xbffff59b ("SSH_AGENT_PID=1299")
0020| 0xbffff32c --> 0xb7fdc858 --> 0xb7e26000 --> 0x464c457f
0024| 0xbffff330 --> 0x0
0028| 0xbffff334 --> 0xbffff31c --> 0xb7e3f4d3 (<__libc_start_main+243>:       mov    DWORD PTR [esp]
,eax)
[-----------------------------------------------------------------------------]
Legend: code, data, rodata, value

Temporary breakpoint 5, 0x080484e5 in main ()
```

Anti-Debugging

Anti-Debugging

- Goal: making the debugging process as difficult as possible
- There is almost always a way to go around each anti-debugging trick...
  - But there are many of them to deal with, and the process can be painful... successfully slowing down the binary analysis task
- Different classes of techniques
  - API-based
  - Direct access to process or thread information
  - Breakpoint detection
  - Exception-based
- Many techniques for Windows, much simpler on Linux

BP Detection

- Breakpoint Detection
  - Hardware Breakpoints by checking the hardware registers
    - But they are not accessible from ring 3...
    - In Windows there are workarounds based on exception handlers
  - Software breakpoint by checksumming the code or by looking for INT3 opcodes
- Time-based Trace detection
  - If a process is executed in single-step mode it is extremely slow
  - Almost any timing mechanism can be used to detect the anomalously slow execution speed

Exception Traps

- INT3 trap
  - Used by the debugger to implement software breakpoints
  - A program can intentionally call the int 3 instruction
    - If the debugger is not attached, the program will raise an exception

```
#include <signal.h>
static int debugged = -1;
int handler(int signo)
{
    debugged = 0;
    return 0;
}

int main()
{
    signal(SIGTRAP, handler);
    debugged = 1;
    __asm__("int3");
    if (debugged)
}
```

Debugging Itself

- Only one debugger can be attached to a program at any given time
- A program can try to debug itself (or fork and do that from another process)
  - If the call fails, another debugger is already associated to that process

```
int main()
{
    if (ptrace(PTRACE_TRACEME, 0, 1, 0) < 0)
    {
        printf("DEBUGGING... Bye\n");
        return 1;
    }
    printf("Hello\n");
    return 0;
}
```

Debugger Side Effects (bugs)

- For several years, GDB had a little bug that left file descriptors open before starting a new process

```
int detect_gdb()
{
    FILE *fd = fopen("/tmp", "r");

    if (fileno(fd) > 5)
        return 1;

    fclose(fd);
    return 0;
}
```

MS Windows

- Different classes of techniques
  - API-based
  - Direct access to process or thread information
  - Breakpoint detection
  - Exception-based

API-Based Techniques

- `IsDebuggerPresent()`
  - Check the Process Environment Block (PEB) for the `DebuggerPresent` flag
- `CheckRemoteDebuggerPresent()`
  - Check the Process Environment Block (PEB) for the `BeingDebugged` flag
- `OutputDebugString()`
  - Calling this function when there is no debugger attached to the process return an error
- `NtQueryInformationProcess()`
  - Native call to retrieve information about a process

```
NtQueryInformationProcess(-1, 0x07, &ret, 4, NULL)
```

Direct Access To Thread & Process Info

- PEB Flags
- IsDebuggerPresent Hooks
  - Many debuggers hook this call .. and the hook can be detected
- Windows Vista System DLL Name
  - For normal processes, the system DLL in the TEB is set to `kernel32.dll`. When a process is created by a debugger is set instead to `HookSwitchHookEnabledEvent`
- Heap Allocation Algorithm
  - Heaps that are created under a debugger use a different algorithm used for validation and to detect heap corruption
  - Many values in the PEB and in memory can be used to identify this condition

Exception Traps

- `INT3` trap
  - Used by the debugger to implement software breakpoints
  - A program can intentionally call the `int 3` instruction
    - If the debugger is not attached, the program will raise an exception
- CloseHandle Exception trap
  - Calling `CloseHandle(..)` with an invalid handle generates an exception if a debugger is attached, and return an error code otherwise
- `LOCK+CMPXCHG8B` trap
  - The `CMPXCHG8B` does not work if used with the lock prefix
  - If such combination is detected, a debugger would kill the process. If no debugger is present, the program can catch the exception and continue

Debugging Itself

- Only one debugger can be attached to a program at any given time
- A program can fork and try to attach a debugger to its child through the `DebugActiveProcess` function.
  - If the call fails, another debugger is already associated to that process

For a more exhaustive list of the known anti-debugging tricks you can check:

- http://www.symantec.com/connect/articles/windows-anti-debug-reference

Solutions

- Reduce the Debugger visibility
  - Mostly related to Windows data structures
- Use the appropriate breakpoint technique
  - Hardware is preferred but it is not always the stealthier option
- Intercept debugging-related API functions and return fake results
- Single-step through the problematic part and manually disable the anti-debugging checks

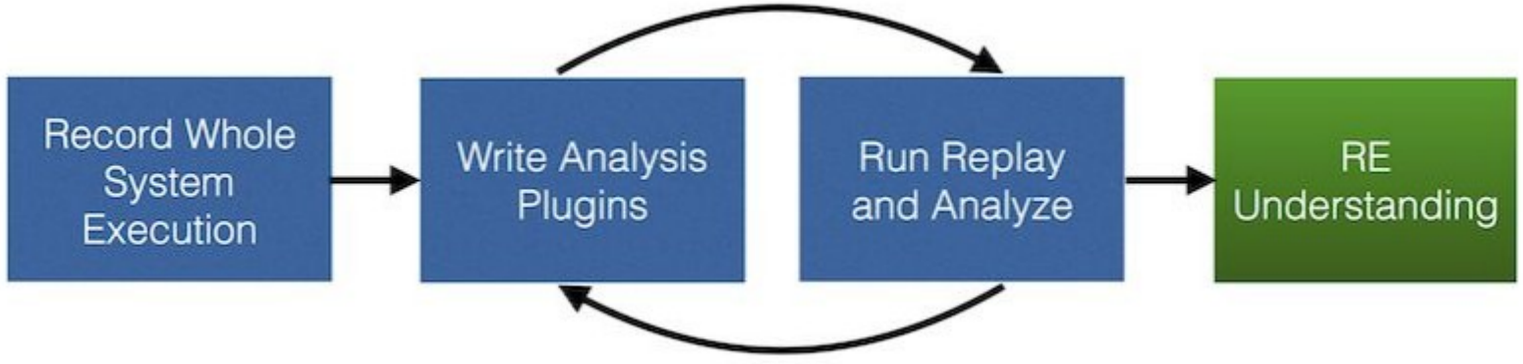
Record / Replay

Problems

- Reproducibility
  - Big problem in Malware analysis as the execution often depends on external components that only operate for a limited amount of time
  - Re-running malware months after they have been captured often return no results
- Complex Analysis
  - Are too slow to be executed in realtime
    (e.g., taint analysis)
  - Produce too much data to store for each sample
    (e.g., full instruction traces)

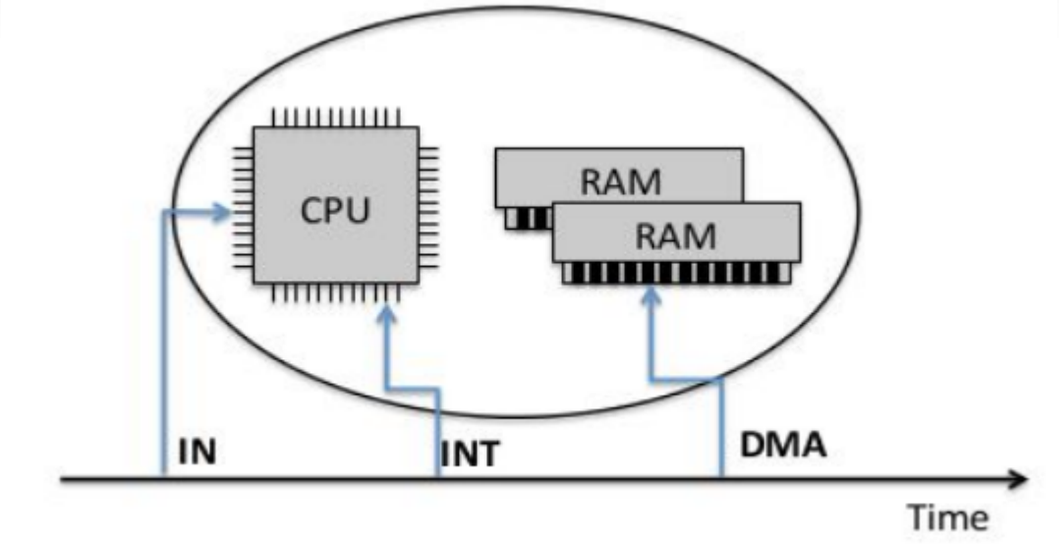
PANDA

- **P**latform for **A**rchitecture **N**eutral **D**ynamic **A**nalysis
- Based on QEMU and LLVM
- Execute a binary (actually the entire operating system) while recording any non-deterministic input in a compressed trace
- Allows to replay the recording (re-executing the system by providing the same inputs) while performing complex analysis



Recording

- PANDA records:
  - Any data entering the CPU
  - Any hardware interrupt and parameters
  - Any data written to RAM by DMA peripherals

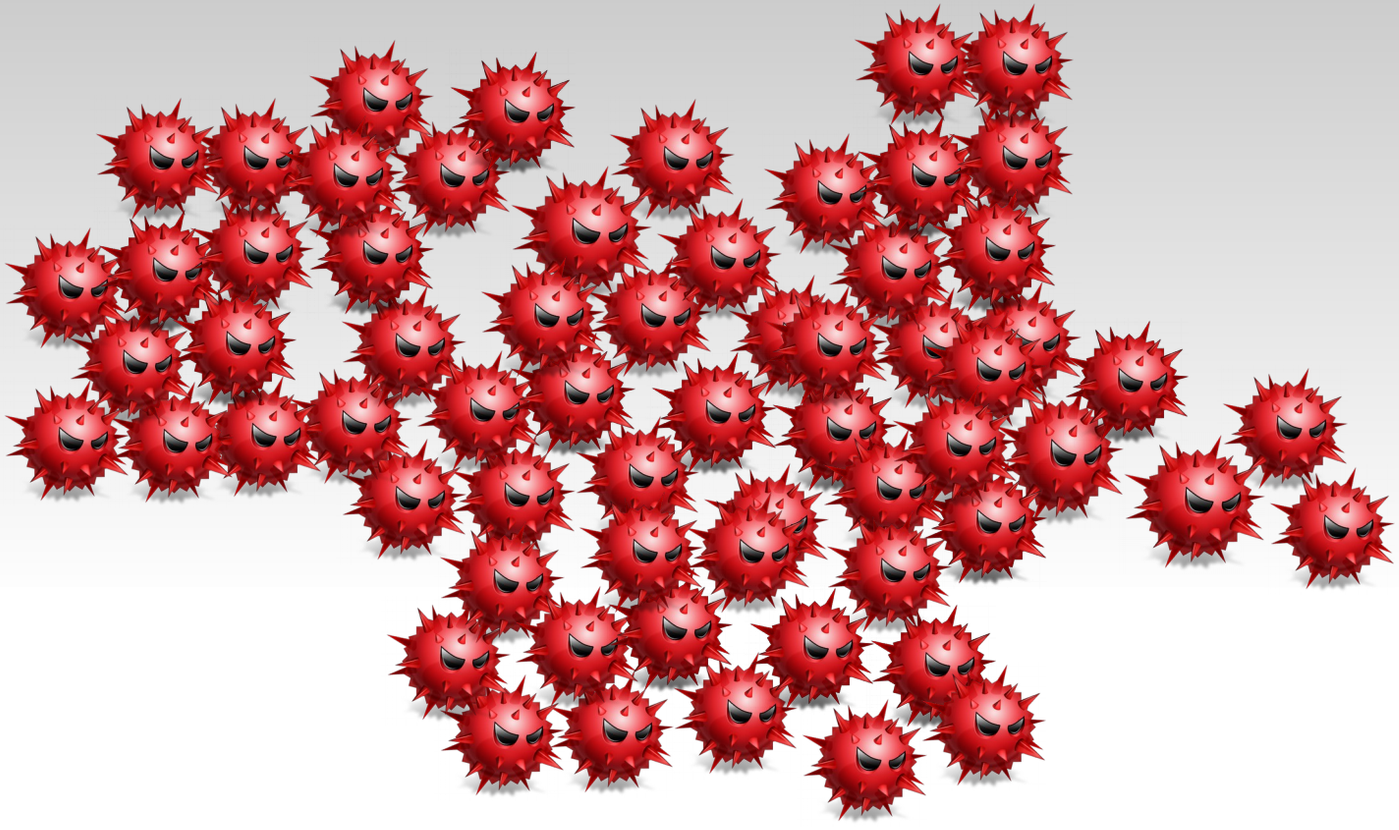


Replay

- When the system is re-played, it is completely self-contained
  - Reproducible results
  - Perfect isolation

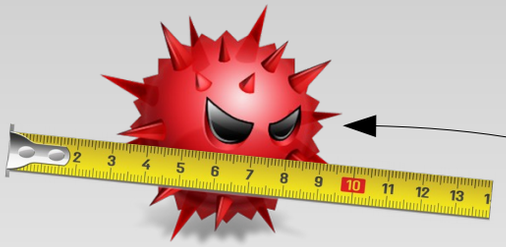
- During replay, the analyst can load analysis plugins that can take action at various instrumentation points

Panda Documentation:
https://github.com/moyix/panda/blob/master/docs/manual.md

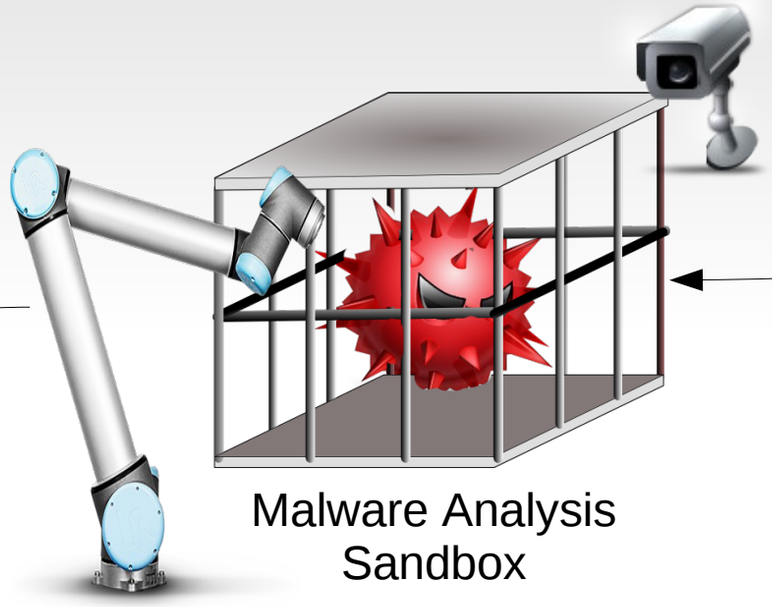


Large Scale Analysis

Is it known ??

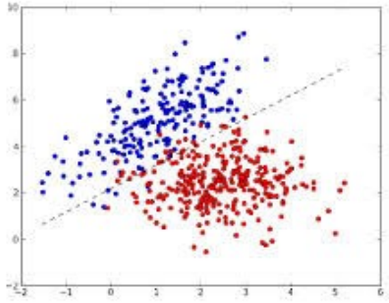


Does it match a known signature?



Malware Analysis Sandbox

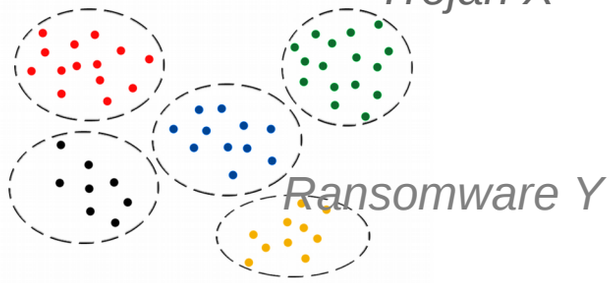
Behavioral Reports



Classifier

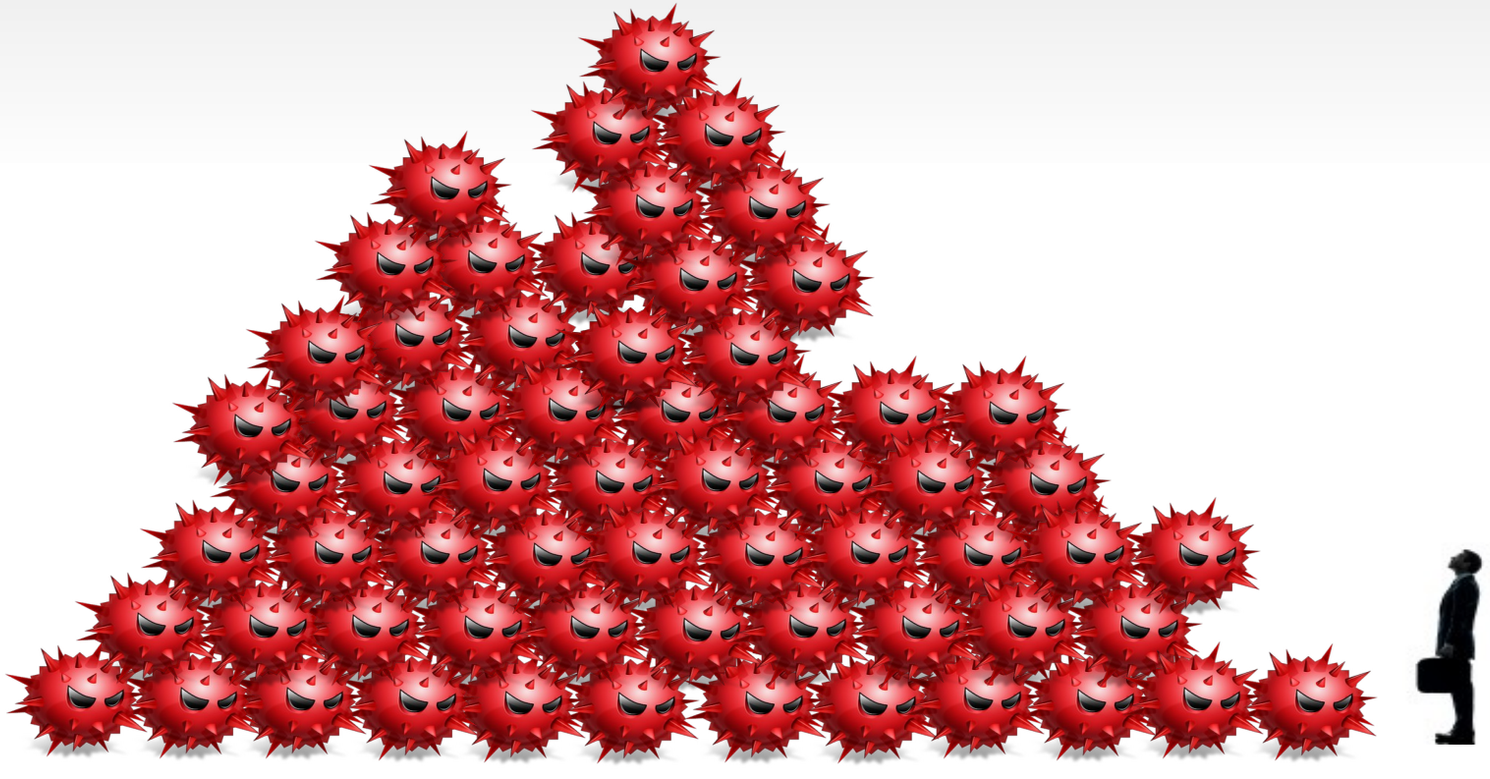
bad

good



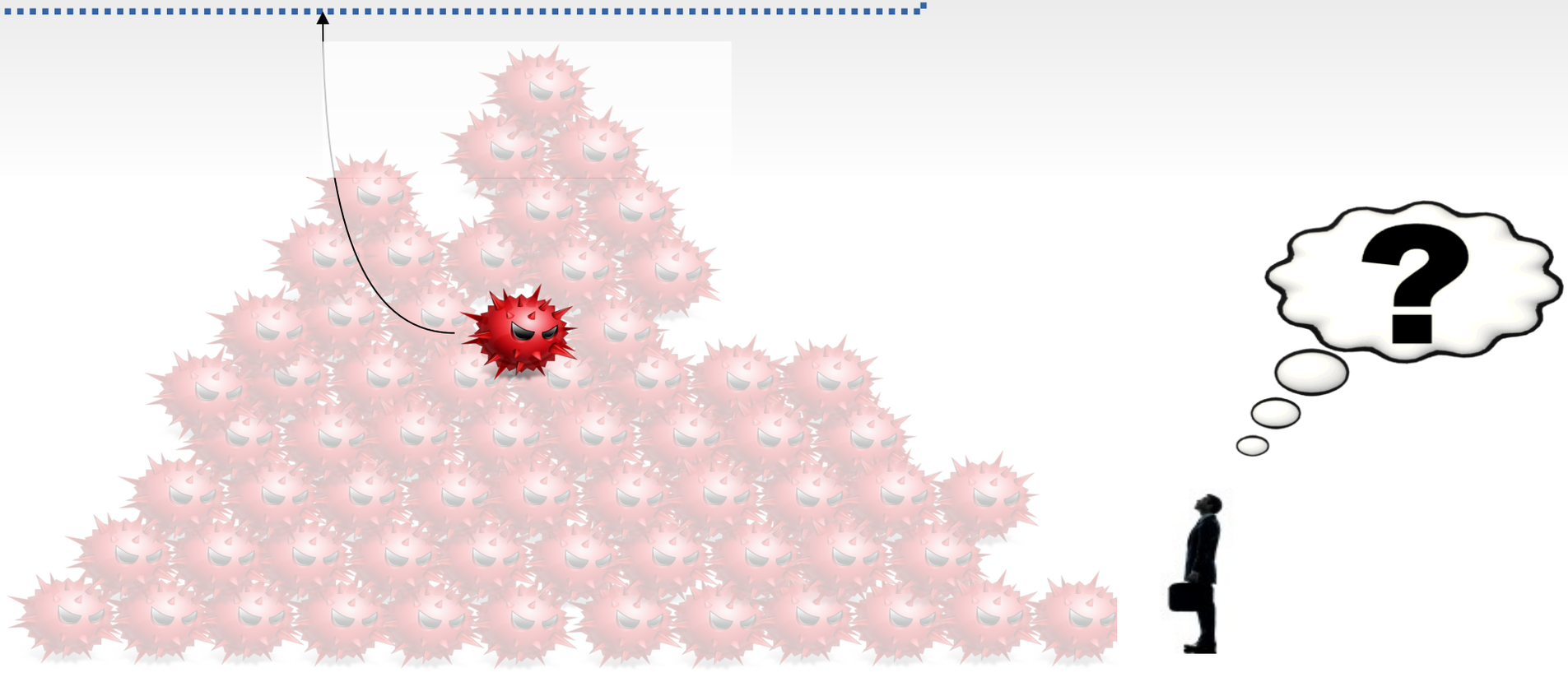
Clustering

Even IF we had a perfect malware classifier, how do we distinguish interesting samples from the rest ?

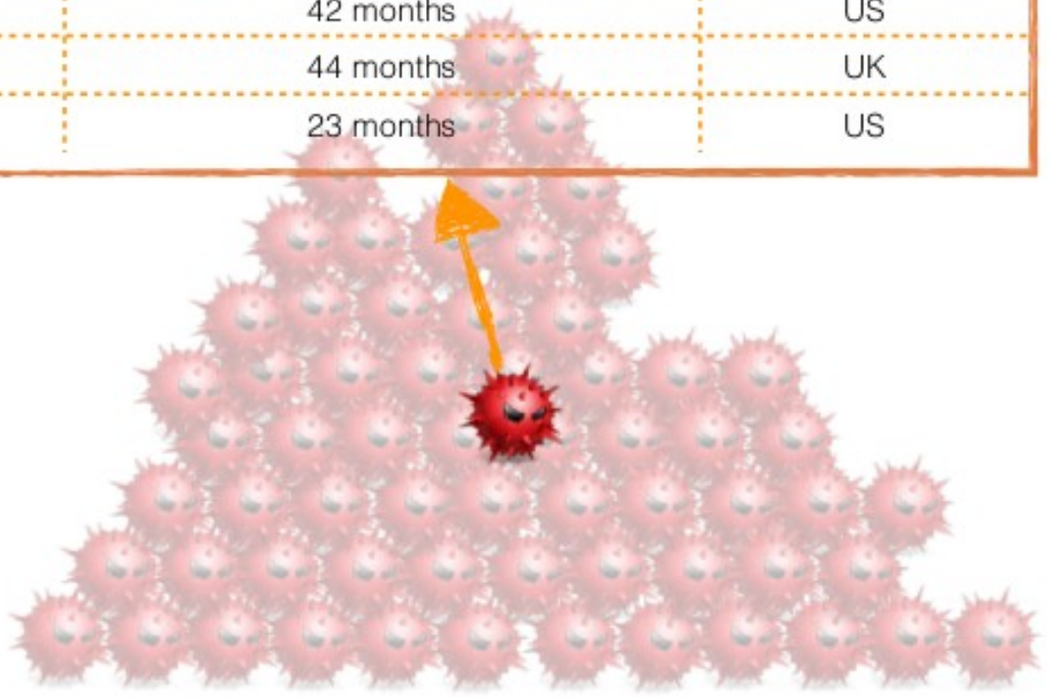


Equation Group Sample

(collected & analyzed 23 months before it was “discovered”)



CAMPAIGN	TIME BEFORE PUBLIC DISCLOSURE	SUBMITTED BY
Operation Aurora	4 months	US
Red October	8 months	Romania
APT1	43 months	US
Stuxnet	1 month	US
Beebus	22 months	Germany
LuckyCat	3 months	US
BrutePOS	5 months	France
NetTraveller	14 months	US
Pacific PluX	12 months	US
Pitty Tiger	42 months	US
Regin	44 months	UK
Equation	23 months	US



Some Figures

- The AVTest institute reports 390K new malicious samples per day
- VT receives over 1M new samples per day, out of which 400-500K are detected by at least one AV
- Symantec reported ~750M new malicious samples in 2014-2015 alone

  → We are already well over the 1B (milliard) mark !!

Some Figures

- The AVTest institute reports 390K new malicious samples per day
- VT receives over 1M new samples per day, out of which 400-500K are detected by at least one AV
- Symantec reported ~750M new malicious samples in 2014-2015 alone

  → We are already well over the 1B (milliard) mark !!

But at least, now that we have collected so much data, we can finally answer a lot of questions. Right?

How Many of the Samples we collect every day are Really Malicious ?

How Many Samples are Packed ?

How many have VM detection capabilities ?

How Many Samples are Packed ?

Panda [2007] → 79%

McAfee [2009] → 80%

Bayer [2011] → 40%

Intel [2014] → 37% (63% custom or unknown)

How many have VM detection capabilities ?

How Many Samples are Packed ?

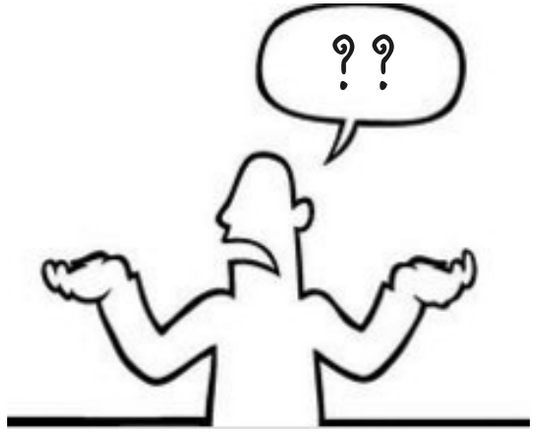
Panda [2007] → 79%

McAfee [2009] → 80%

Bayer [2011] → 40%

Intel [2014] → 37% (63% custom or unknown)

How many have VM detection capabilities ?



Bayer [2011] → < 12%

Lindorfer [2011] → 26%

Fireeye [2011] → It's a Myth!

Intel [2012] → 81%

Microsoft [2014] → 28%

Symantec [2014] → 18%-28%

How Long do we need to run each sample?

How many malicious samples also query popular domains?

What is the fraction of samples that do not belong to polymorphic families?

How prevalent is technique X?

Lots of data… plenty of statistics…
but still very few answers :(

Big Data should allow us to extract Intelligence, Analytics, discover new Correlations, observe General Trends and the evolution of the Big Picture

Big Data should allow us to extract Intelligence, Analytics, discover new Correlations, observe General Trends and the evolution of the Big Picture

... Otherwise it is just a Burden!!

Data

- Confusing marketing statistics published by companies..
  with no information about the methodology or even the meaning of the terms

- What does it even mean today to have a “*representative dataset*”?

Research

- We need help from data scientists … but they wont solve the problem alone
- In just eight years we went from 1M to 1B samples.
  Think how many 5-year-old papers are now already obsolete
- How can we practically look into these gigantic datasets?
- Automation, automation, automation …
  and manual analysis only *<u>when</u>* (??) necessary