Kaizen

-a experimental antivirus engine based on infectious disease dynamics-

22nd of April – start of the project

Given that in the past 2 years I only touched Python, R and a bit of JS, I needed a refresher on C++ [1].

23rd of April

Continued reading a bit more on C++ performance, algorithms, essentials [2 - 9].

“Parameter transfer is more efficient in 64-bit mode than in 32-bit mode, and more efficient in 64-bit Linux than in 64-bit Windows. In 64-bit Linux, the first six integer parameters and the first eight floating point parameters are transferred in registers, totaling up to fourteen register parameters. In 64-bit Windows, the first four parameters are transferred in registers, regardless of whether they are integers or floating point numbers. Therefore, 64-bit Linux is more efficient than 64-bit Windows if functions have more than four parameters. There is no difference between 32-bit Linux and 32-bit Windows in this respect” [7]

We will develop our PoC on Linux. Not many AVs for Linux available, especially open source; I think it’s a battle worth fighting.

24th of April

Got me a book on CMake [10]. It’s a bit early for our project, but still.

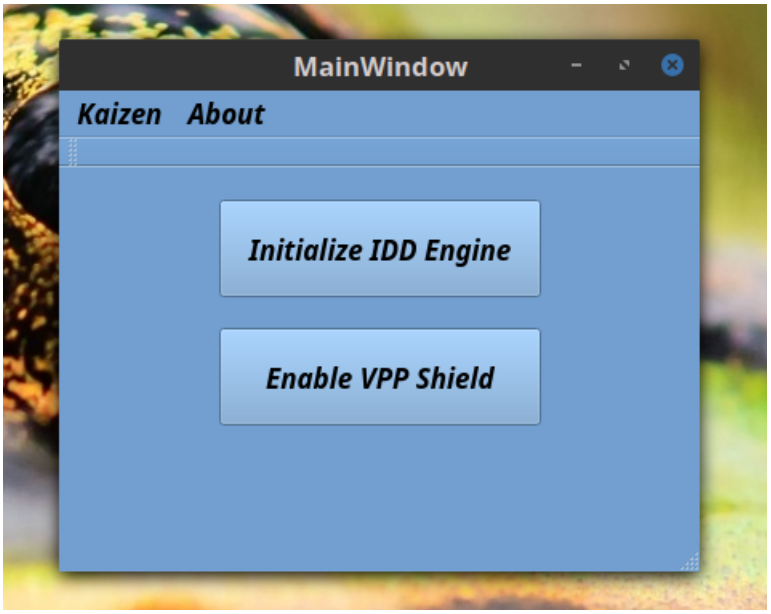
Kaizen will have two distinct components:

I. IDD, or Infectious Disease Dynamics Engine
II. VPP, or Virulent Process Protection Shield

IDD’s main objective will be reconnaissance, that is scanning all active running processes and capturing the current execution state of the machine.

VPP’s main objective will be monitoring the processes and their behavior accordingly to the mathematical model that we will define and use in the following days.

Our PoC will feature a minimalist GUI. Nothing fancy. As a matter of fact, let us prototype one in Qt Creator real quick:



This should do it just fine.

28th of April

Started my day with [12]. Ended it with [16].

Remember that my working methodology consists of 4 stages (R&D):

R I – initial research, idea, design
R II – academic research, advanced topics, mathematical model
D I – first implementation, allowed to use recipes or book adaptations
D II – final stage of the project, refined codebase, working PoC

Obviously, we are in R I now. We will step up the pace in the next week. I was busy getting some certificates in the meantime :).

Given Kaizen's nature and the the model I proposed, we will need to seriously consider the matter of **malware persistence**. Our PoC will not cover the following:

- registry
- Windows services
- startup keys
- malicious drivers

The primary reasons is that for one, we will develop it on Linux and secondly, **IDD & VPP work only at process level**.

That leaves us with tons of other issues – like process injection, APC injection, process replacement and/ or hook injection (not an issue on Linux).

More creative ways of malware evasion techniques – like anti-disassembly or anti-virtual machines and polymorphism (via metaprogramming or code expansion, etc) are out of the scope of this project.

Also, ELF files (Linux executables) are not analyzed, only the interactions between the spawned processes are scanned, after the IDD engine took the initial machine running state snapshot.

We might use decompilers like Snowman, Hex-Rays, dotPeek or iLSpy occasionally during the R I stage.

Cuckoo, ThreatAnalyzer, Joe Sandbox, Buster Sandbox Analyzer are also on our list.

VirusTotal, Malwr, Falcon Sandbox might be consulted along the way, if necessary.

Objdump, ltrace, strace, gdb, Radare2 might come in handy sometime.

When it comes to Linux memory forensics, LiMEaide will be our top choice for now, including Volatility ofc.

I'll call it a day now.

29th of April

Short recap of Linux process state:

R: running or runnable, it is just waiting for the CPU to process it

- S: Interruptible sleep, waiting for an event to complete, such as input from the terminal
- D: Uninterruptible sleep, processes that cannot be killed or interrupted with a signal, usually to make them go away you have to reboot or fix the issue
- Z: Zombie, we discussed in a previous lesson that zombies are terminated processes that are waiting to have their statuses collected
- T: Stopped, a process that has been suspended/stopped

Process info:

USER: The effective user (the one whose access we are using)

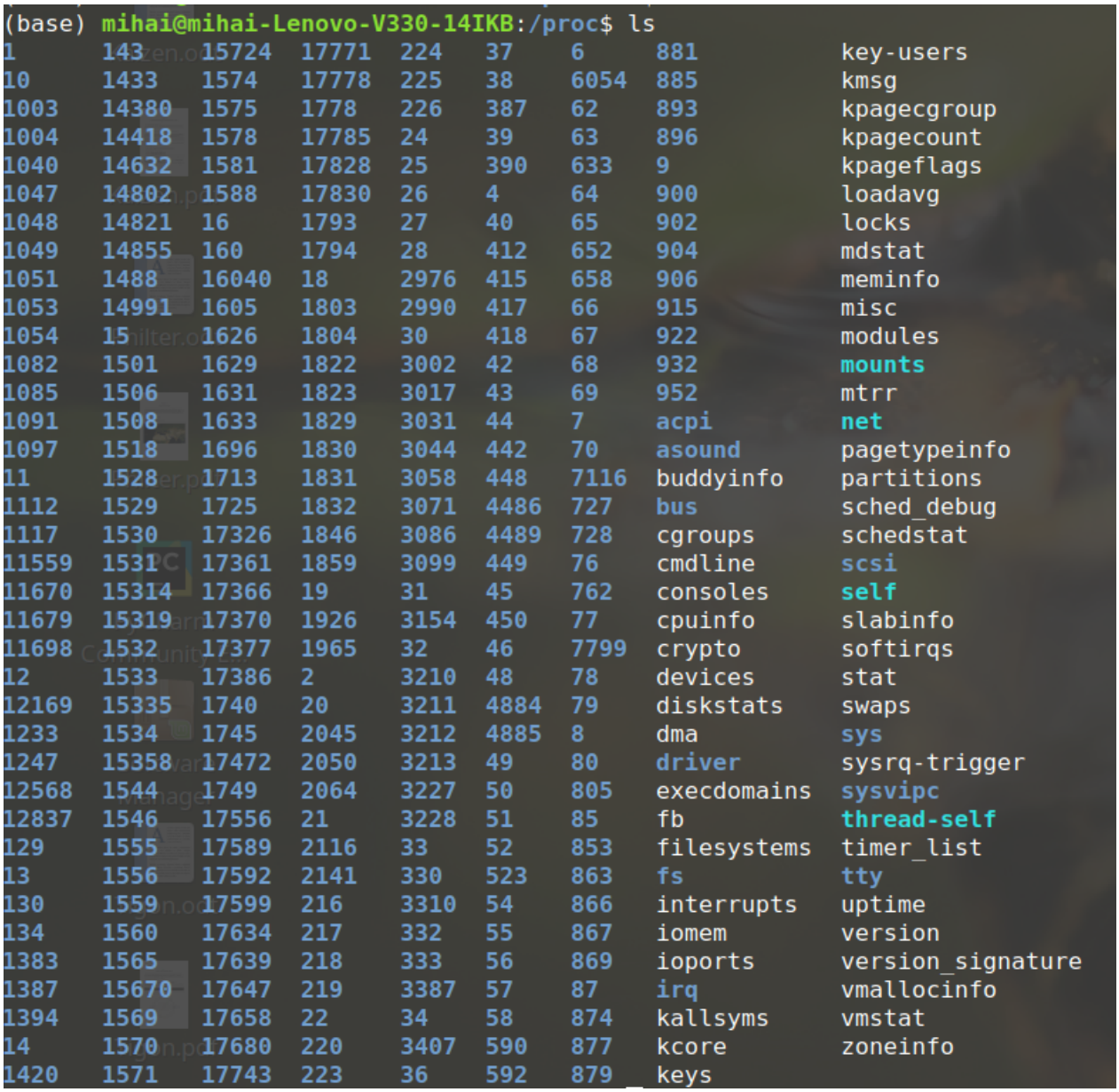
- PID: Process ID
- %CPU: CPU time used divided by the time the process has been running
- %MEM: Ratio of the process's resident set size to the physical memory on the machine
- VSZ: Virtual memory usage of the entire process
- RSS: Resident set size, the non-swapped physical memory that a task has used
- TTY: Controlling terminal associated with the process
- STAT: Process status code

- START: Start time of the process
- TIME: Total CPU usage time
- COMMAND: Name of executable/command

Process signals:

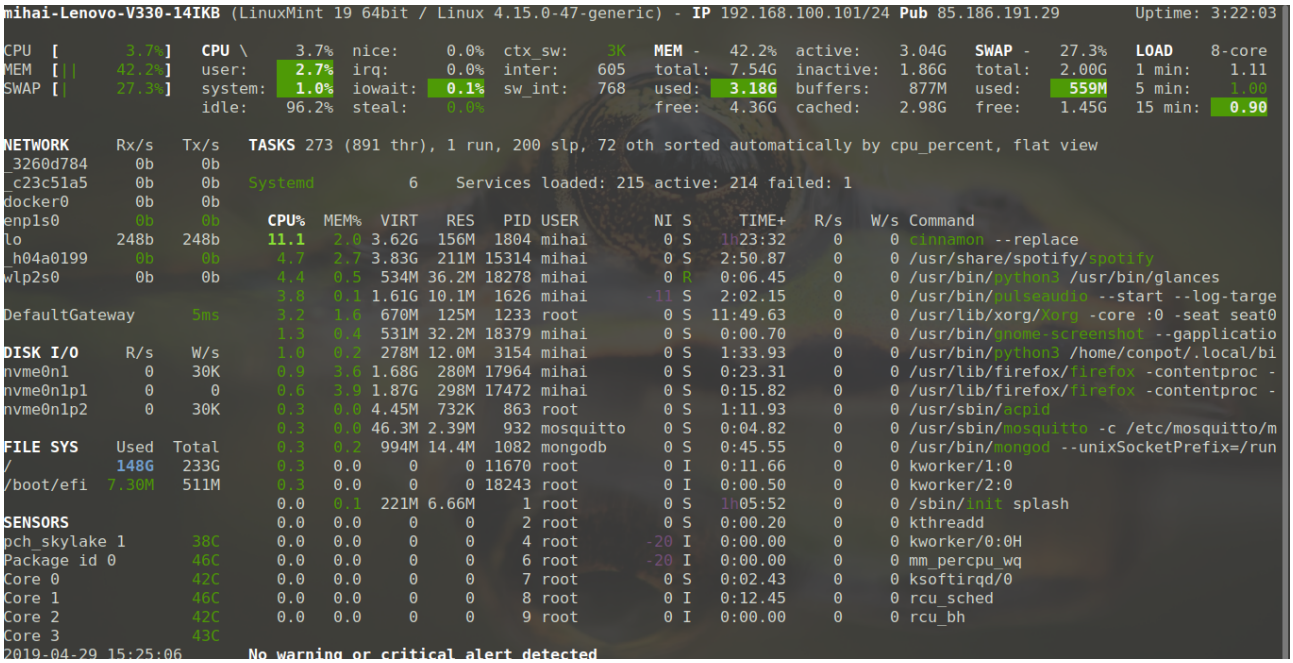
SIGHUP or HUP or 1: Hangup
- SIGINT or INT or 2: Interrupt
- SIGKILL or KILL or 9: Kill
- SIGSEGV or SEGV or 11: Segmentation fault
- SIGTERM or TERM or 15: Software termination
- SIGSTOP or STOP: Stop

Even processes are files in Linux: ls /proc



We will also need to keep daemons in mind. More on that later.

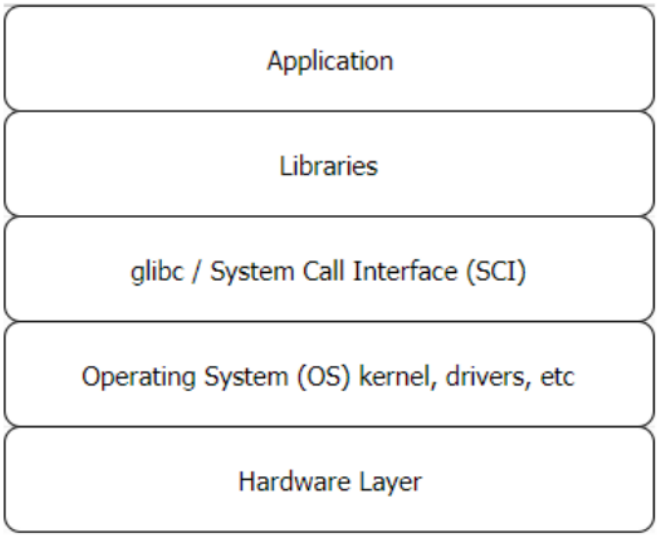
Kaizen's IDD needs to also classify processes by the parent/ child criteria.

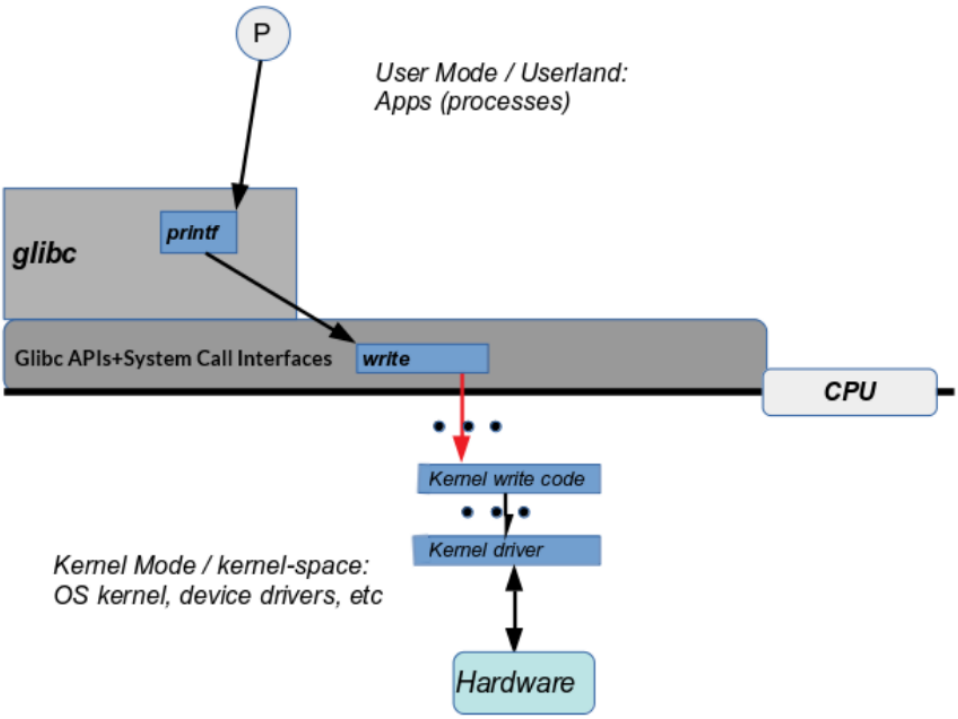


We will define all variables that IDD needs to measure in the next days. Disk I/O will certainly be a part of this list.

Got me 3 additional books on Linux System programming and the Linux Kernel. Everything is added to the bibliography section.

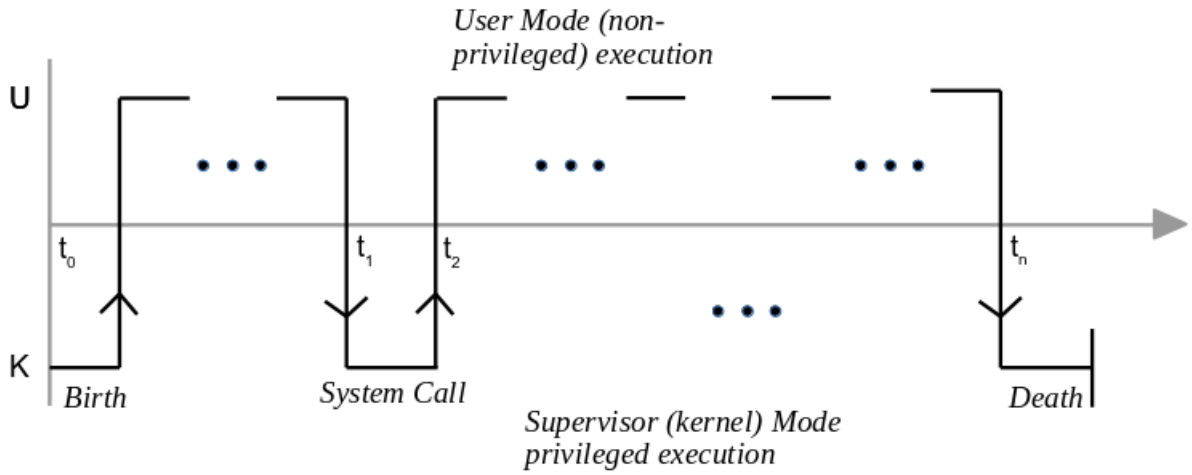
"The Unix design philosophy abstracts peripheral devices (such as the keyboard, monitor, mouse, a sensor, and touchscreen) as files – what it calls device files. By doing this, Unix allows the application programmer to conveniently ignore the details and just treat (peripheral) devices as though they are ordinary disk files. The kernel provides a layer to handle this very abstraction – it's called the ***Virtual Filesystem Switch*** *(**VFS**). So, with this in place, the application developer can open a device file and perform I/O (reads and writes) upon it, all using the usual API interfaces provided (relax, these APIs will be covered in a subsequent chapter)."*





[26]

"*To summarize, in a monolithic kernel, when a process (or thread) issues a system call, it switches to privileged Supervisor or kernel mode and runs the kernel code of the system call (working on kernel data). When done, it switches back to unprivileged User mode and continues executing userspace code (working on user data).*"



[26]

"*Technically, Linux is not considered 100 percent monolithic. It's considered to be mostly monolithic, but also modular, due to the fact that the Linux kernel supports modularization (the plugging in and out of kernel code and data, via a technology called* ***Loadable Kernel Modules*** *(****LKMs****)). Interestingly, MS Windows (specifically, from the NT kernel onward) follows a hybrid architecture that is both monolithic and microkernel.*"

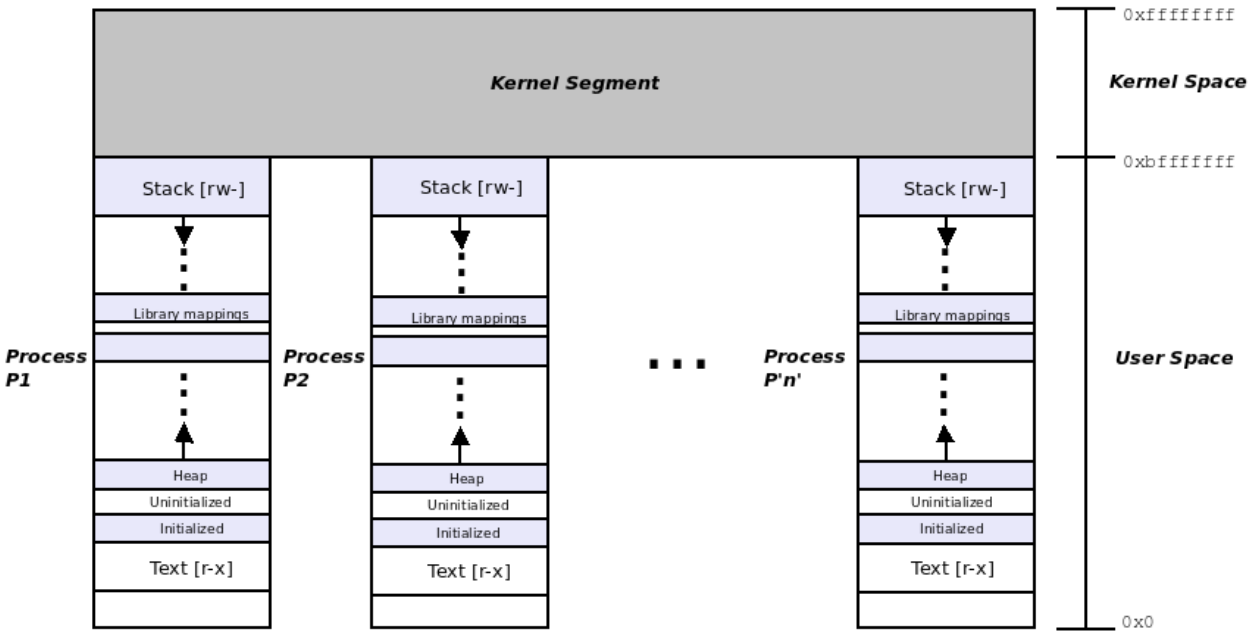
"**Process-isolation**

With virtual memory, every process runs inside a sandbox, which is the extent of its VAS. The key rule: it cannot look outside the box.

So, think about it, it's impossible for a process to peek or poke the memory of any other process's VAS. This helps in making the system secure and stable.

Example: we have two processes, A and B. Process A wants to write to the `0x10ea` *virtual address in process B. It cannot, even if it attempts to write to that address, all it's really doing is writing to its own virtual address,* `0x10ea`*! The same goes for reading.*

So we get process-isolation – each process is completely isolated from every other process. Virtual address X for process A is not the same as virtual address X for process B; in all likelihood, they translate to different physical addresses (via their PTs). Exploiting this property, the Android system is designed to very deliberately use the process model for Android apps: when an Android app is launched, it becomes a Linux process, which lives within its own VAS, isolated and thus protected from other Android apps (processes)!"



[26]

"Notice how the top gigabyte of VAS for every process is the same – the kernel segment. Also keep in mind that this layout is not the same on all systems – the VMSPLIT and the size of user and kernel segments varies with the CPU architecture."

"The following sums up important points to note when a predecessor process execs a successor:

- *The successor process overwrites (or overlays) the predecessor's virtual address space.*
  - *In effect, the predecessor's text, data, library, and stack segments are now replaced by that of the successor's.*
  - *The OS will take care of the size adjustments.*
- *No new process has been created—the successor now runs in the context of the old predecessor.*
  - *Several predecessor attributes (including but not limited to the PID and open files) thus get auto-inherited by the successor.*
- *On a successful exec, there is no possibility of returning to the predecessor; it's gone. Colloquially, performing an exec is like committing suicide for the predecessor: After successful execution, the successor is all that's left; returning to the predecessor is out of the question:"*

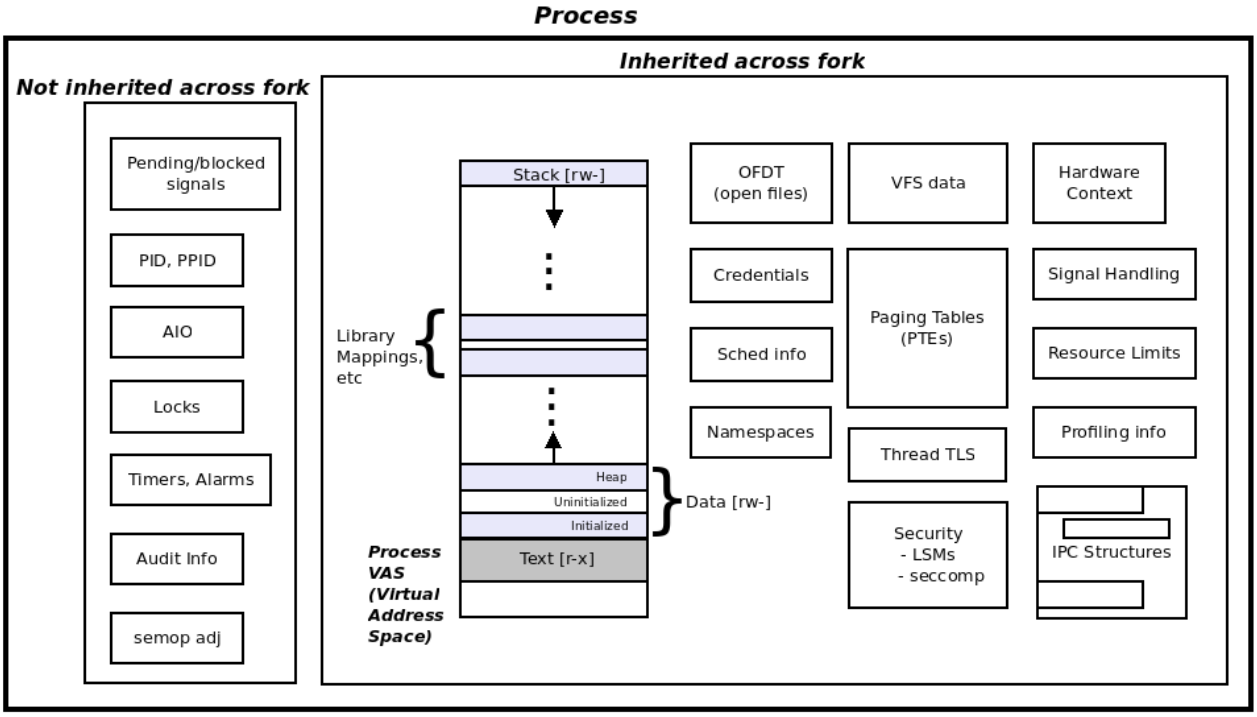
"There is more, much more, to a process than just its VAS. This includes open files, process credentials, scheduling information, filesystem structures, paging tables, namespaces (PIDs, and so on), audit information, locks, signal handling information, timers, alarms, resource limits, IPC structures, profiling (perf) information, security (LSM) pointers, seccomp, thread stacks and TLS, hardware context (CPU and other registers), and so on."

“As a first-level enumeration, the following process attributes are inherited by the child process upon fork (meaning, it-the new born child-gets a copy of the parent's attributes with the same content):

- *The**VAS**:*
    - *Text*
    - *Data:*
        - *Initialized*
        - *Uninitialized (bss)*
        - *Heap*
    - *Library segments*
    - *Other mappings (for example, shared memory regions, mmap regions, and so on)*
    - *Stack*
- *Open files*
- *Process credentials*
- *Scheduling information*
- *Filesystem (VFS) structures*
- *Paging tables*
- *Namespaces*
- *Signal dispositions*
- *Resource limits*
- *IPC structures*
- *Profiling (perf) information*
- *Security information:*
    - *Security (LSM) pointers*
    - *Seccomp*
- *Thread stacks and TLS*
- *Hardware context”*

“The following attributes of the parent process are not inherited by the child process upon forking:

- *PID, PPID*
- *Locks*
- *Pending and blocked signals (cleared for child)*
- *Timers, alarms (cleared for child)*
- *Audit information (CPU/time counters are reset for child)*
- *Semaphore adjustments made via* `semop(2)`
- ***Asynchronous IO** (**AIO**) ops and contexts”*



***Fork rule #1**: After a successful fork, execution in both the parent and child process continues at the instruction following the fork.*

***Fork rule #2**: To determine whether you are running in the parent or child process, use the fork return value: it's always 0 in the child, and the PID of the child in the parent.*

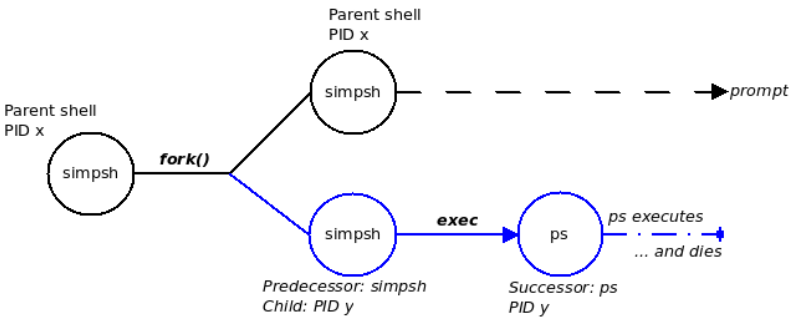
***Fork rule #3**: After a successful fork, both the parent and child process execute code in parallel.*

***Fork rule #4**: Data is copied across the fork, not shared.*

***Fork rule #5**: After the fork, the order of execution between the parent and child process is indeterminate.*

***Fork rule #6**: Open files are (loosely) shared across the fork.*

***Fork rule #7**: The parent process must wait (block) upon the termination (death) of every child, directly or indirectly*



“A signal can be defined as an asynchronous event that is delivered to a target process. Signals are delivered to the target process either by another process or the OS (the kernel) itself.

At the code level, a signal is merely an integer value; more correctly, it is a bit in a bitmask. It's important to understand that, although the signal may seem like an interrupt, it is not an interrupt. An interrupt is a hardware feature; a signal is purely a software mechanism. “

“FYI, for the default case—that is, all cases where the application developer has not installed a specific signal-handling routine (we will learn how exactly to install our own signal handlers shortly)—what exactly does the OS code that handles these cases do? Depending on the signal being processed, the OS will perform one of these five possible actions (see the following table for details):

- *Ignore the signal*
- *Stop the process*
- *Continue the (previously stopped) process*
- *Terminate the process*
- *Terminate the process and emit a core dump”*

“

Signal	***Integer value***	***Default action***	***Comment***
`SIGHUP`	*1*	*Terminate*	*Hang up detected on controlling terminal or death of controlling process*
`SIGINT`	*2*	*Terminate*	*Interrupt from keyboard :* `^C`
`SIGQUIT`	*3*	*Term&Core*	*Quit from keyboard :* `^\`
`SIGILL`	*4*	*Term&Core*	*Illegal Instruction*
`SIGABRT`	*6*	*Term&Core*	*Abort signal from abort(3)*
`SIGFPE`	*8*	*Term&Core*	*Floating-point exception*
`SIGKILL`	*9*	*Terminate*	*(Hard) kill signal*
`SIGSEGV`	*11*	*Term&Core*	*Invalid memory reference*
`SIGPIPE`	*13*	*Terminate*	*Broken pipe: write to pipe with no readers; see pipe(7)*
`SIGALRM`	*14*	*Terminate*	*Timer signal from alarm(2)*

SIGTERM	*15*	*Terminate*	*Termination signal (soft kill)*
SIGUSR1	*30,10,16*	*Terminate*	*User-defined signal 1*
SIGUSR2	*31,12,17*	*Terminate*	*User-defined signal 2*
SIGCHLD	*20,17,18*	*Ignore*	*Child stopped or terminated*
SIGCONT	*19,18,25*	*Continue*	*Continue if stopped*
SIGSTOP	*17,19,23*	*Stop*	*Stop process*
SIGTSTP	*18,20,24*	*Stop*	*Stop typed at terminal : ^Z*
SIGTTIN	*21,21,26*	*Stop*	*Terminal input for background process*
SIGTTOU	*22,22,27*	*Stop*	*Terminal output for background process*

”

“As we have seen, we can successfully handle all signals in several ways, each with their own approaches to signal-handling at high volume—pros and cons as shown in the following table:

Method	**Pros**	**Cons/Limitations**
Use `sigfillset(3)` just prior to calling `sigaction(2)` to ensure that while the signal is being handled, all other signals are blocked.	Simple and straightforward approach.	Can lead to significant (unacceptable) delays in handling and/or dropping of signals.
Setting the `SA_NODEFER` signal flag and handling all signals as they arrive.	Simple and straightforward approach.	On load, heavy stack usage, danger of stack overflow.
Use an alternate signal stack, set the `SA_NODEFER` signal flag, and handle all signals as they arrive.	Can specify alternate stack size as required.	More work to setup; must carefully test under load to determine (max) stack size to use.
Use real-time signals (covered in the following chapter).	The OS queues pending signals automatically, low stack usage, signal prioritization possible.	System-wide limit on the maximum number that can be queued (can be tuned as root).

”

“**Differences from standard signals**

So, how do the so-called real time signals differ from the regular standard signals; the following table reveals this:

Characteristic	**Standard signals**	**Real time signals**
Numbering	1 - 31 [1]	34 - 64 [2]
Standard first defined in	POSIX.1-1990 (it's old)	POSIX 1003.1b : real time Extensions to POSIX (2001)
Meaning assigned	Individual signals have a particular meaning (and are named accordingly); the exception is `SIGUSR[1|2]`	Individual RT signals have no particular meaning; their meaning is app-defined
Behavior when blocked and multiple instances of same signal continuously delivered	Out of n instances of the same signal, n-1 are lost; only 1 instance is kept pending and delivered to the target process when unblocked	All instances of RT signals are queued and delivered to the target process by the OS when unblocked (there is a system-wide upper limit [3])
Signal priority	The same: all standard signals are peers	FCFS unless pending; if pending, then signals delivered from lowest to highest numbered realtime signal [4]
Inter Process Communication (**IPC**)	Crude IPC; you can use `SIGUSR[1|2]` to communicate, but no data can be passed	Better: via the `sigqueue(3)`, a single data item, an integer or pointer value, can be sent to a peer process (which can retrieve it)

”

“Firstly, of course, we need to learn what attributes a pthread has. The following table enumerates this:

Attribute	**Meaning**	**APIs:** `pthread_attr_[...](3)`	**Values Possible**	*Linux Default*
Detach state	Create threads as joinable or detached	`pthread_attr_[get|set]detachstate`	PTHREAD_CREATE_JOINABLE PTHREAD_CREATE_DETACHED	PTHREAD_CREATE_JOINABLE
Scheduling/ contention scope	Set of threads against which we compete for resources (CPU)	`pthread_attr_[get|set]scope`	PTHREAD_SCOPE_SYSTEM PTHREAD_SCOPE_PROCESS	PTHREAD_SCOPE_SYSTEM
Scheduling/ inheritance	Determines whether scheduling attributes are inherited implicitly from calling a thread or explicitly from the attr structure	`pthread_attr_[get|set]inheritsched`	PTHREAD_INHERIT_SCHED PTHREAD_EXPLICIT_SCHED	PTHREAD_INHERIT_SCHED
Scheduling/ policy	Determines the scheduling policy of the thread being created	`pthread_attr_[get|set]schedpolicy`	SCHED_FIFO SCHED_RR SCHED_OTHER	SCHED_OTHER
Scheduling/ priority	Determines the scheduling priority of the thread being created	`pthread_attr_[get|set]schedparam`	struct sched_param holds int sched_priority	0 (non real-time)
Stack/guard region	A guard region for the thread's stack	`pthread_attr_[get|set]guardsize`	Stack guard region size in bytes	1 page

Stack/ location, size	Query or set the thread's stack location and size	`pthread_attr_[get|set]stack`
		
		`pthread_attr_[get|set]stackaddr`
			Stack address and/ or stack size, in bytes	Thread Stack Location: left to the OS
				
				Thread Stack Size: 8 MB

	`pthread_attr_[get|set]stacksize`		

As you can see, clearly understanding what exactly many of these attributes signify requires further information

”

“

Situation	**Thread : `pthread_join(3)`**	**Process: `wait[pid](2)`**
Condition	A thread being waited for must have its detached state attribute set as joinable, not detached.	None; any child process can (and in fact must) be waited upon (recall our *fork rule #7*)
Hierarchy	None: any thread can join on any other thread; there is no requirement of a parent-child relationship. In fact, we do not consider threads to live within a strict parent-child hierarchy as processes do; all threads are peers.	A strict parent-child hierarchy exists; only a parent can wait for a child process.
Order	With threads, one is forced to join (wait) upon the particular thread specified as the parameter to `pthread_join(3)`. In other words, if there are, say, three threads running and main issues the join within an ascending ordered loop, then it must wait for the death or thread #1, then thread #2, and then thread #3. If thread #2 terminates earlier, there is no help for it.	With wait, a process can wait upon the death (or stoppage) of any child, or specify a particular child process to wait for with waitpid.
Signaling	No signal is sent upon a thread's death.	Upon a process's death, the kernel sends the `SIGCHLD` signal to the parent process.

”

“

Approach to make a function thread-safe	**Comments**
Use only local variables	Naive; hard to achieve in practice.
Use global and/or static variables and protect critical sections with mutex locks	Viable but can significantly impact performance [1]
Refactor the function, making it reentrant-safe-eliminate the use of static variables in a function	Useful approach—several old `foo`

by using more parameters as required	glibc functions refactored to `foo_r`.
Thread local storage (**TLS**)	Ensures thread safety by having one copy of the variable per thread; toolchain and OS-version-dependent. Very powerful and easy to use.
Thread-specific data (**TSD**)	Same goal: make data thread-safe –older implementation, more work to use.

”

“Here are some pros of the MT model over the single-threaded process:

Context	**Multiprocess (single-threaded) model**	**Multithreaded (MT) model**
Design for parallelized workloads	• Cumbersome • Non-intuitive • Using the fork/wait semantics repeate dly (creating a large number of processes) isn't simple or intuitive either	• Lends itself to building parallelized software; calling the `pthread_create(3)` in a loop is easy and intuitive as well • Achieving a logical separation of tasks becomes easy • The OS will have threads take advantage of multicore systems implicitly; for the Linux OS, the granularity of scheduling is a thread, not a process (more on this in the next chapter) • Overlapping CPU with IO becomes easy
Creation/ destruction performance	Much slower	Much faster than processes; resource-sharing guarantees this
Context switching	Slow	Much faster between the threads of a process
Data sharing	Done via IPC (Inter-Process Communication) mechanisms; involves a learning curve, can be fairly complex; synchronization (via the semaphore) required	Inherent; all global and static data items are implicitly shared between threads of a given process; synchronization (via the mutex) is required

”

“Here are some cons of the MT model over the single-threaded process:

Context	**Multiprocess (single-threaded) model**	**Multithreaded (MT) model**

Thread-safety	No such requirement; processes always have address space separation.	The most serious downside: every function in the MT application that can be run in parallel by threads must be written, verified, and documented to be thread-safe. This includes the app code and the project libraries, as well as any third-party libraries it links into.
Application integrity	In a large MT app, if any one thread encounters a fatal error (such as a segfault), the entire app is now buggy and will have to shut down.	In a multiprocess app, *o*nly the process that encounters a fatal error will have to shut down; the rest of the project keeps running[1].
Address space constraints	On 32-bit CPUs, the VAS (virtual address space) available to user mode apps is fairly small (either 2 GB or 3 GB), but still large enough for a typical single-threaded app; on 64-bit CPUs the VAS is enormous (2^64 = 16 EB).	On a 32-bit system (still common on many embedded Linux products), the available VAS to user mode will be small (2/3 GB). Considering sophisticated MT apps with many threads, that's not a lot! In fact, it's one of the reasons embedded vendors are aggressively moving products to 64-bit systems.
The Unix everything's a file semantics	The semantic holds true: files (descriptors), devices, sockets, terminals, and so on can all be treated as files; also, each process has its own copy of a given resource.	Resource-sharing, seen as an advantage, can also be seen as a downside: • The sharing can defeat the traditional Unix model advantage • The sharing of open files, memory regions, IPC objects, paging tables, resource limits, and so on implies synchronization overhead upon access
Signal-handling	Designed for the process model.	Not designed for the MT model; can be done, but a bit clumsy to handle signals.
Designing, maintaining, and debugging	Quite straightforward compared to the MT model.	Increases complexity because the programmer has to track (in this mind) the state of several threads simultaneously, including notoriously complex locking scenarios.

		Debugging deadlock (and other) situations can be quite difficult (tools such as GDB and helgrind help, but the human still needs to track things).

”

*“The first and very important concept for the developer to understand is that OSes maintain a construct called the **Kernel Schedulable Entity (KSE**). The KSE is the granularity at which the OS scheduling code operates. In effect, what object exactly does the OS schedule? Is it the application, the process, the thread? Well, the short answer is that the KSE on the Linux OS is a thread. In other words, all runnable threads compete for the CPU resource; the kernel scheduler is ultimately the arbiter that decides which thread gets which CPU core and when.*
Next, we present an overview of the process, or thread's, state machine.

The Linux process state machine

On the Linux OS, every process or thread runs through a variety of definite states, and by encoding these, we can form the state machine of a process (or thread) on the Linux OS).

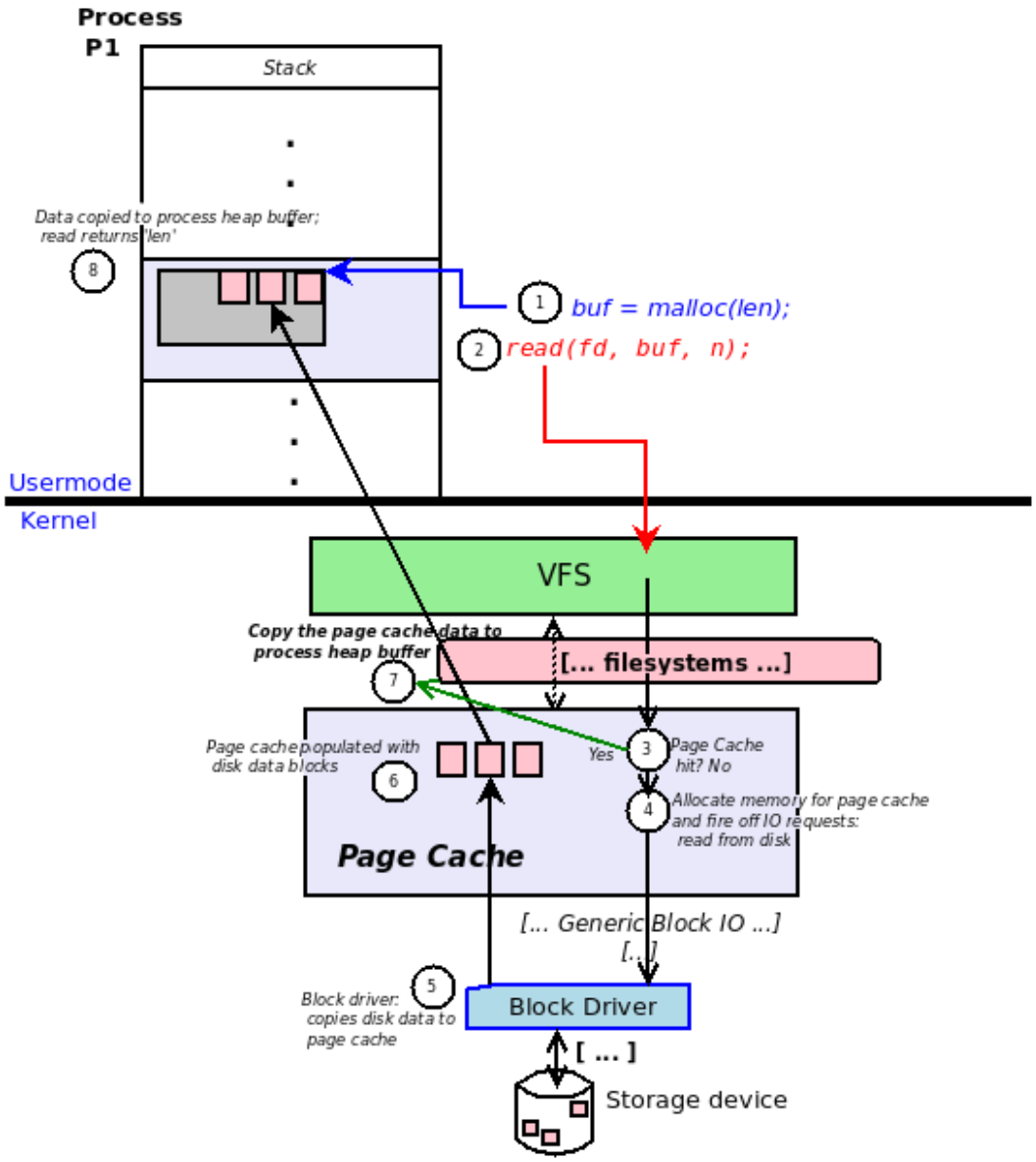
”

“

Scheduling policy	**Type**	**Priority range**
`SCHED_FIFO`	Soft real-time: Aggressive, unfair	1 to 99
`SCHED_RR`	Soft real-time: Less aggressive	1 to 99
`SCHED_OTHER`	Non real-time: Fair, time sharing; the default	Nice value (-20 to +19)

”

I’ll insert another diagram concerning Linux I/O execution:



[26]

I will also insert their epilogue:

Rule #1 : Check all APIs for their failure case.

- *Rule #2: Compile with warnings on (`-Wall -Wextra`) and eliminateallwarningsas far as is possible.*
- *Rule #3: Never trust (user) input; validate it.*
- *Rule #4: Use assertions in your code.*
- *Rule #5 : Eliminate unused (or dead) code from the codebase immediately.*
- *Rule #6 : Test thoroughly; 100% code coverage is the objective. Take the time and trouble to learn to use powerful tools: memory checkers (Valgrind, the sanitizer toolset), staticand dynamic analyzers, security checkers (checksec), fuzzers (see the following explanation).*
- *Rule #7 : Do not assume anything (assume makes an ass out of u and me).*

Also, don't forget that **Good judgment comes from experience, and experience comes from bad judgment.**

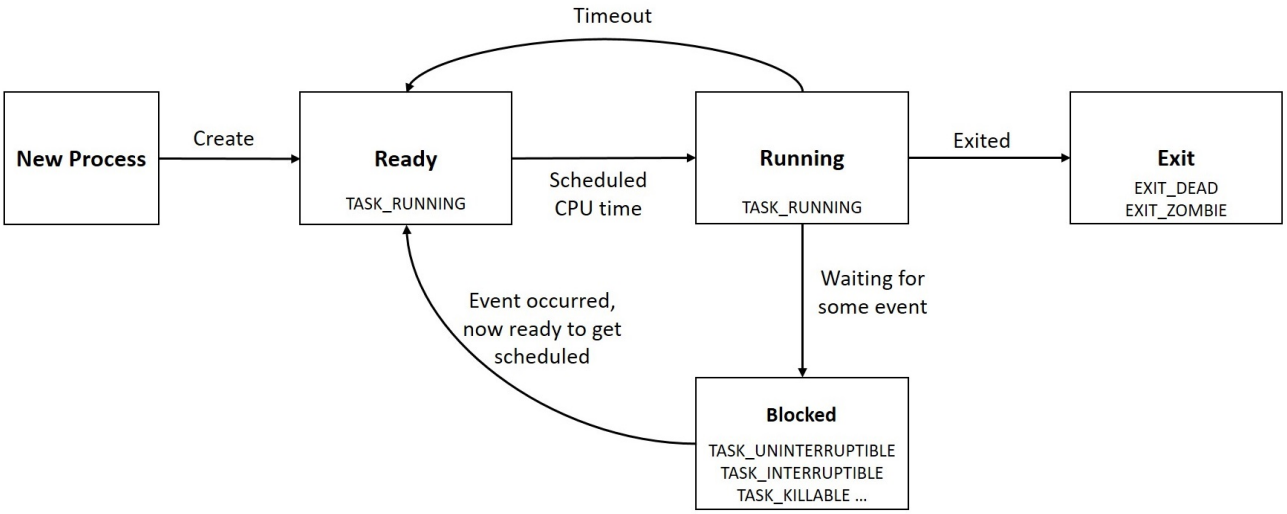
I highly recommend the following book: Hands-On System Programming with Linux **By Kaiwan N Billimoria.** [26]

Most of the previous screenshots, diagrams and text snippets were from [26].

30th of April

Got me 2 more books – [27 – 28].

Will attach here additional diagrams on Linux processes architecture from [28].



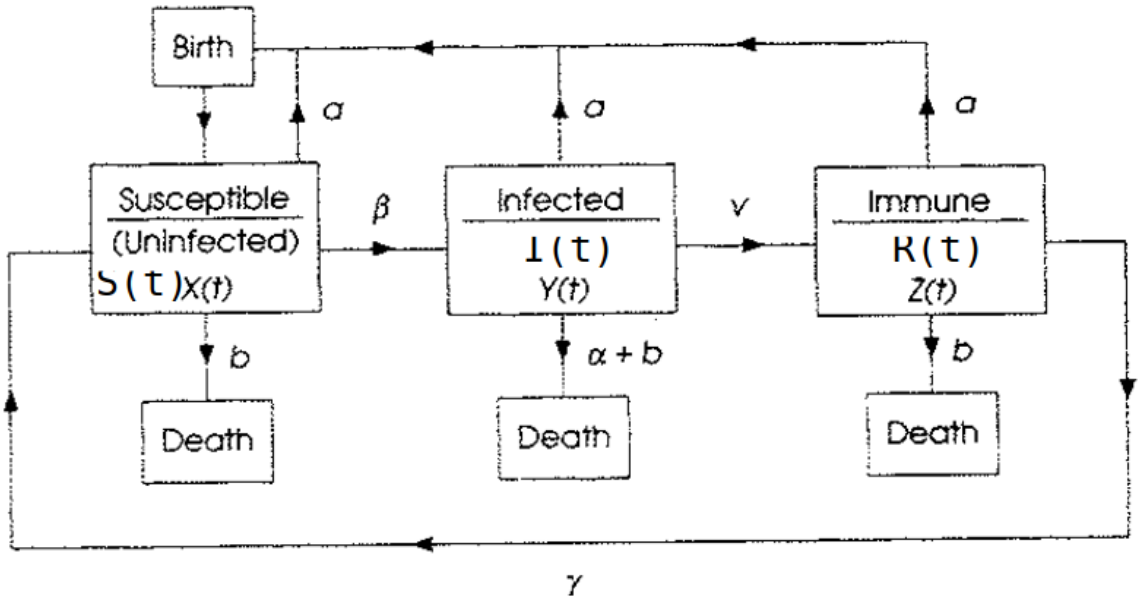
Tomorrow we will start looking into infectious disease dynamics. After that, we will finish the R I stage and enter R II where we will take a more technical approach into our project and also sketch the mathematical model that we will use.

1st of May

Today we will explore the topic of infectious disease dynamics and its applicability in our task. There are several reasons that have made me choose this mathematical model:

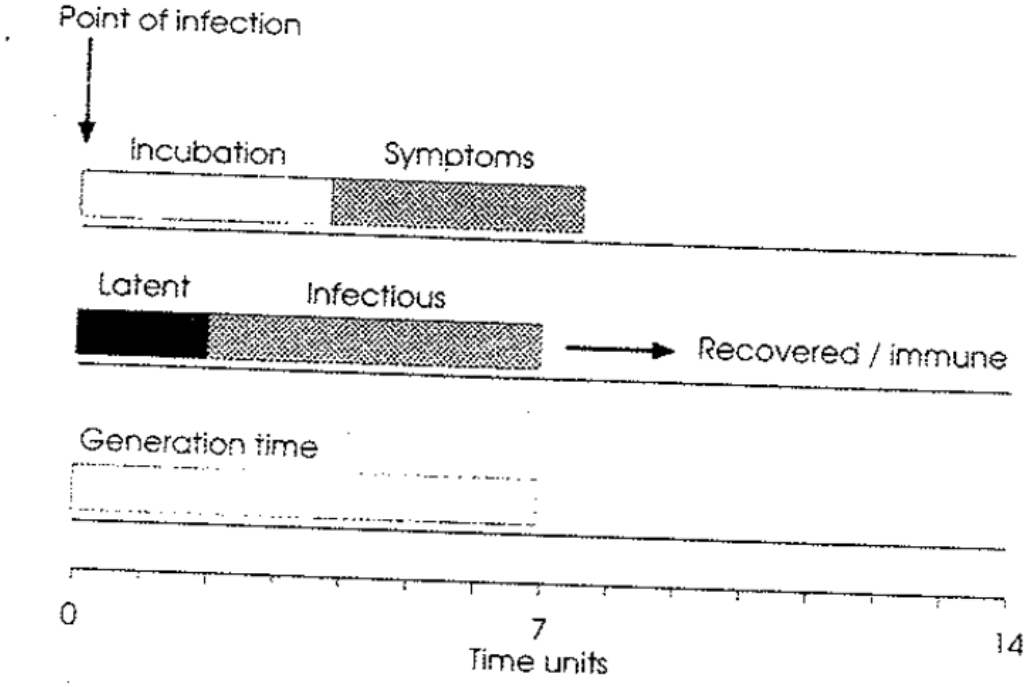
- had another project where I tried something similar, a anti spam filter
- the interactions between processes can be modeled in a reliable and efficient manner
- it will be able to detect deviant behavior like ransomware or other AV engines
- it won't need signatures nor a internet connection
- it will have a relatively low footprint on the system's overall performance
- it won't affect the host system, Kaizen will not alter the running processes in any way; its objective revolves merely around detection of an active, present rogue process

A handy graph explaining the main phases of IDD can be found in [28]:

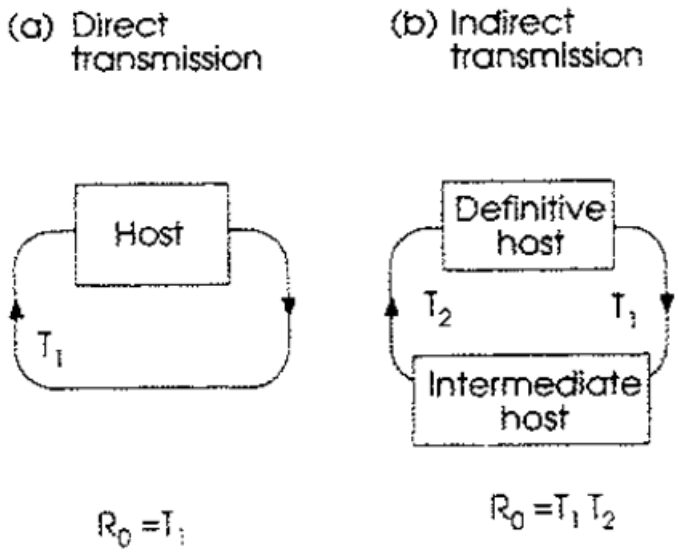


- susceptible = vulnerable process
- infected
- immune (VPP shield)

A framework for discussing the population biology of infectious diseases



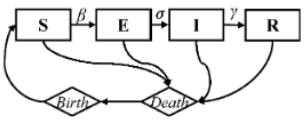
Incubation – latent or malware not active, awaiting activation or instructions



Indirect transmission can be thought of when a bad agent uses a good process to infect another, without actually damaging its integrity or functional parameters.

Another interesting concept is the matter of seasonality. In our context, seasonality can be extended to certain operations performed by the host OS – for instance booting up, updating and so on. Take a look [29]:

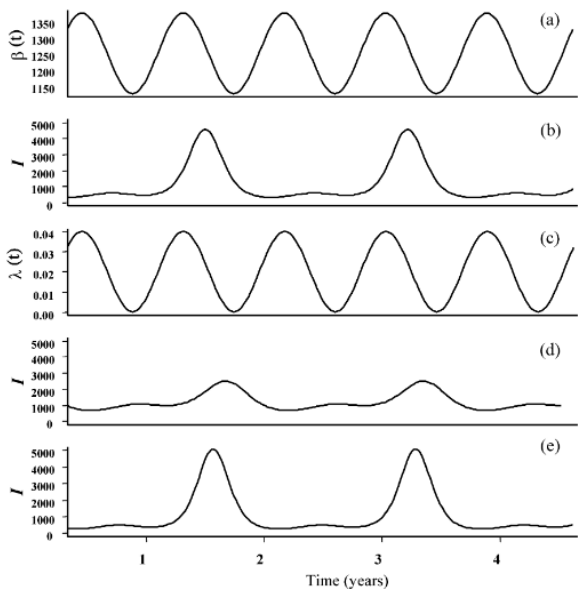
Box 1 Effects of seasonal forcing on susceptible, exposed, infected and recovered (SEIR) systems



$$\frac{dS}{dt} = \lambda(t)(N - S) + \mu I - \beta(t)S\frac{I}{N}$$
$$\frac{dE}{dt} = \beta(t)S\frac{I}{N} - \lambda(t)E - \sigma E$$
$$\frac{dI}{dt} = \sigma E - (\gamma + \mu + \lambda(t))I$$

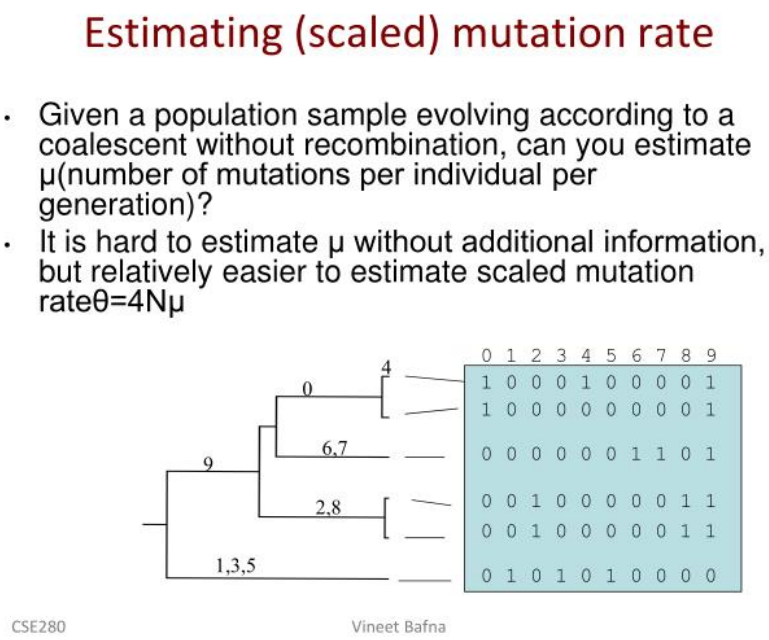
$$R = N - (S + E + I)$$
$$\beta(t) = \beta_0(1 + \beta_1 \cos(2\pi t))$$
$$\lambda(t) = \lambda_0(1 + \lambda_1 \cos(2\pi t))$$

Seasonality in either transmission or births can yield complex dynamics in SEIR systems. In the box diagram, seasonality can operate on processes including transmission β, recovery γ, birth λ and death. In this example, the nonlinearity interacts with the seasonality to produce biennial cycles. Note that the relatively strong seasonal fluctuation in births ($\lambda_1 = 1$) produces a weaker fluctuation than the lower amplitude fluctuation in the transmission rate ($\beta_1 = 0.1$), which dominates the dynamics when the two are combined.



Here, the graphical outputs illustrate (a) seasonal fluctuation in transmission rate, (b) resulting biennial dynamics in infected individuals given constant birth rate, (c) seasonal fluctuations in birth rate, (d) resulting biennial dynamics given constant transmission rate, and (e) dynamics resulting from varying both birth and transmission rates. Parameters $\lambda_0 = 0.02$, $\lambda_1 = 1$, $\beta_0 = 1250$, $\beta_1 = 0.1$, $N = 5 \times 10^6$, $\gamma = 73$, $\mu = 0$, $\sigma = 45.625$.

A concept that can be applicable when examining malware polymorphism is that of "Coalescent theory" - "A theory that describes the shape and size of genealogies that represent the shared ancestry of sampled genes. It describes how the statistical distribution of branch lengths in genealogies depends on population processes such as size change and structure." [30]. One can easily see this concept coupled with algorithms similar to ssdeep when analyzing mutation and similar malware evasion techniques.



[31]

Variables dump from [32]:

S, I, R	Host population sizes or densities of susceptible, infected, and removed individuals
N	Total host population size or density ($N = S + I + R$ or $S + I$)
s, i	Proportion of susceptible and infected hosts ($s = S/N, i = I/N$)
b	Per capita birth rate of hosts
d	Per capita death rate of disease-free hosts
r	Intrinsic growth rate of disease-free hosts ($r = b - d$)
K	Carrying capacity of hosts
B	Population-level rate of host birth or immigration
R_0	Basic reproduction ratio
S_0	Value of S in the absence of infected hosts
N_0	Value of N in the absence of infected hosts
b_0	Value of b at low population density
d_0	Value of d at low population density
r_0	Value of r at low population density
α	Per capita disease-induced death rate of hosts
β	Infection rate constant
γ	Per capita recovery rate of hosts, from infected to removed hosts
θ	Per capita recovery rate of hosts, from infected to susceptible hosts
μ	Per capita removal rate of infected hosts ($\mu = \alpha + d + \gamma + \theta$ or $\alpha + d + \gamma$ or $\alpha + d$)
λ	Force of infection ($\lambda = \beta S$)
U	Population density of uninfected vectors
V	Population density of infected vectors
Z	Total population density of vectors ($Z = U + V$)
v	Proportion of infected vectors ($v = V/Z$)
χ	Per capita bite rate of vectors
F	Population-level rate of vector birth or immigration

f	Fitness in continuous time ($f = 0$ is neutral)
w	Fitness in discrete time ($w = 1$ is neutral)
t	Time
τ	Delay time
T	Duration of time period
a	Age
p	Probability (subscripted if necessary)
c	Cost-related constant
c_0, c_1, c_2	Arbitrary constants
$\cdot_{res}$	Trait value of resident individuals
$\cdot_{mut}$	Trait value of mutant individuals
$\cdot'$	Derivative
$\bar{\cdot}$	Average
$\cdot^*$	Equilibrium value

Found a really crazy paper called “When Zombies Attack! Mathematical Modeling of an Outbreak of zombie infection”. If you need a laugh, go check it out: “The followers of Vodou believe that a dead person can be revived by a sorcerer. After being revived, the zombies remain under the control of the sorcerer because they have no will of their own. ” :))

Today we have finished the R I stage. Next time we will start the more comprehensive research stage, R II.

8th of May

We enter R II today.

Played a bit with NetLogo. There are a couple of infectious disease models already created. Go check them out if you wish, they are free.

https://ccl.northwestern.edu/netlogo/models/

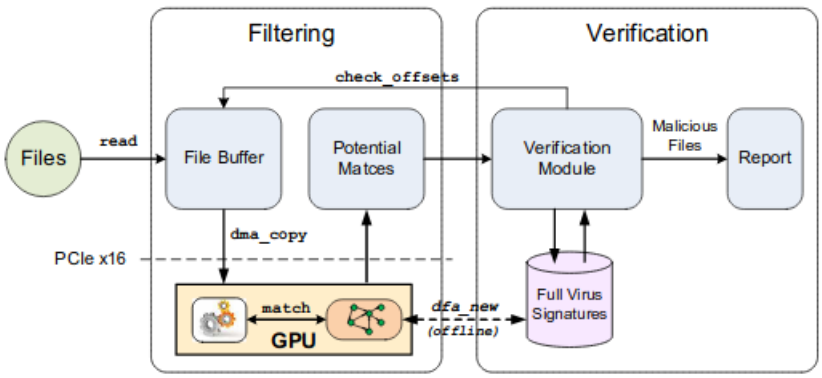
Studied a couple more biology papers [33 – 36].

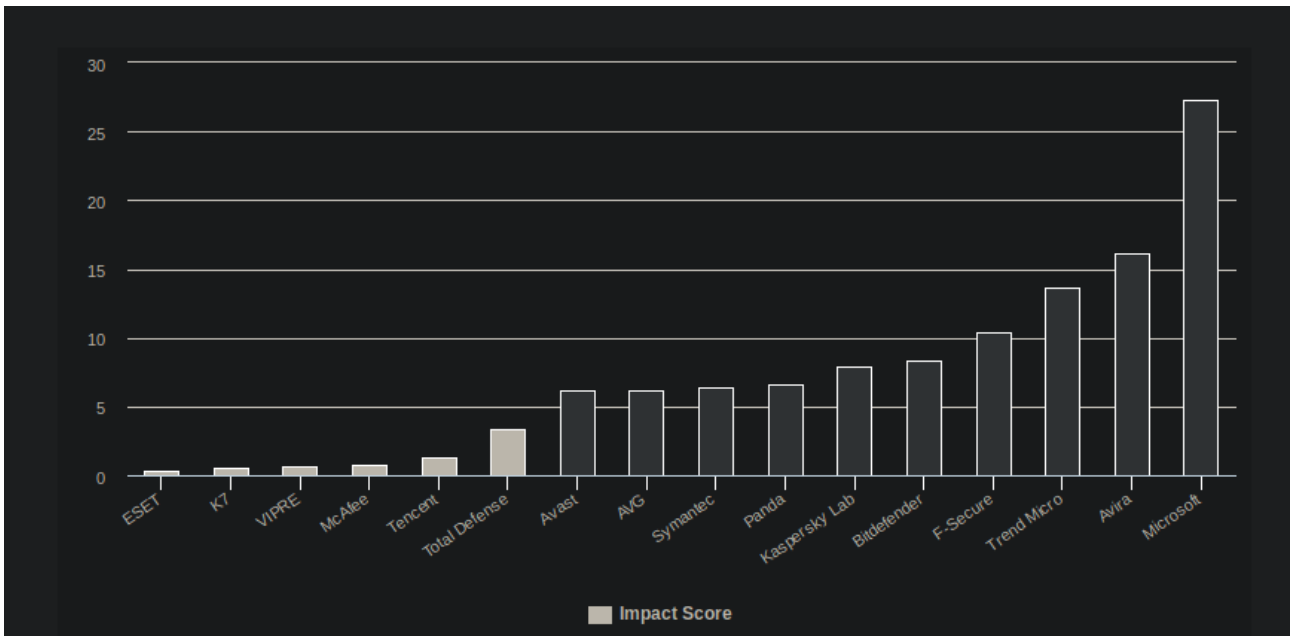
So far we have studied the Linux process architecture and the biologic foundation of disease transmission. Let us now focus on how we can package those two together in an AV engine.

“The Polymorphic Virus

With biological viruses, mutations overwhelmingly result in nonviable offspring. However, because of their sheer numbers, at least some of the mutated offspring are successful. Computer viruses cannot afford to play this type of Russian roulette. If a simple computer virus were to propagate randomly mutated copies of itself, the odds are the mutated children would fail to exhibit virus-like properties. In the most likely scenario, a mutated child virus would cause its host program to crash any time it was executed. This would immediately reveal the virus to the user and limit its ability to successfully reproduce. Because of these realities, current polymorphic computer viruses do not mutate at all; instead, they have specially designed mutation engines that simulate the process of mutation.” [37]

Gravity AV diagram [40]:





Av Comparatives performance chart, 2019.

"*On traditional desktop operating systems, processes started by one user all inherit that user's user ID (UID). As file system access controls are based on the UID, every process can access all files by every other process with the same UID ,also including all personal files by the user who started the respective processes. To introduce additional security and data privacy, and to prohibit access to any file system objects by other apps, the Android operating system breaks with the aforementioned traditional UID assignment scheme. Instead of assigning one identical UID to every process, every app is assigned its own unique UID at installation time by default [3]. The ownership of every app's working directory (located in/data/data/[package name]) and all files contained in this working directory is set to the respective app's UID. Access to files is by default only permitted to the app's UID, and thus only to the app itself. Most filesystem objects outside of apps' working directories are owned by system UIDs and thus cannot be accessed as well. Ultimately, this UID assignment scheme and connected file system access permissions establish a file system sandbox for apps [22]. They are confined in their own directory, separating each app from every other, and from operating system's files and folders. Android antivirus software, which is also just usual apps, is limited drastically by this file system-based sandboxing. As a consequence, antivirus apps cannot scan the file system on demand or monitor file system changes. Most importantly, antivirus apps cannot scan the working directories of other apps and are oblivious to any files other apps might download at runtime. Apps can do this to add functionality after installation in the form of native code, which will thus not show in package files.*" [48]

[52] is a very good read, even if not applicable in our use case.

Another great paper dealing with digital forensics, which is a great interest of mine can be found at [57].

Ended today with [64].

9th of May

I'm starting with [65] today.

Given that Android uses the Linux Kernel, studying Android HIDS and other AV attempts comes in handy:

Author	Approach	Detection Method	Platform	Description
Schmidt et al.(2008)[35]	HIDS, NIDS	Anomaly Detection	Android OS	Analyzes the security on Android smartphones from Linux-kernel view. Uses Network traffic, Kernel system calls, File system logs and Event detection modules to detect anomalies in the system.
Schmidt et al.(2009)[32]	HIDS	Signature-Based Detection	Android OS	Performs static analysis on the executables to extract function calls in Android OS using the command readelf. Function calls are compared with malware executables for classification.
Bläsing et al.(2010)[3]	HIDS	Signature-Based Detection	AndroidOS	Uses an Android Application Sandbox to perform Static and Dynamic analysis on Android applications. Static analysis scans Android source code to detect Malware patterns. Dynamic analysis executes and monitors Android applications in a totally secure environment.
Enck et al.(2010)[15]	HIDS,NIDS	Anomaly Detection	Android OS	TaintDroid is a real time monitoring system for Android OS. TaintDroid monitors Android applications and alerts the user whenever a sensitive data of the user is compromised. Uses "taint tracking" analysis to monitor privacy sensitive information.
Portolakidis et al.(2010)[29]	HIDS,NIDS	Anomaly Detection	Android OS	A remote security server in the cloud performs the Malware detection analysis. Virtual environments will be used to analyze Android mobile phone replicas.
Shabtai et al.(2010)[37]	HIDS	Anomaly Detection	Android OS	Intrusion detection for mobile devices using the knowledge-based, temporal abstraction method (KBTA) methodology. Detects suspicious temporal patterns and to issues an alert if an intrusion is found. These patterns are compatible with a set of predefined classes of malware as defined by a security expert.
Shabtai et al.(2011)[38]	HIDS	Anomaly Detection	Android OS	Host-based malware detection system that continuously monitors smartphone features and events and applies machine learning to classify the collected data as normal (benign) or abnormal (malicious) based on a already known malware and behavior.

Table 2: Android-based malware detection systems

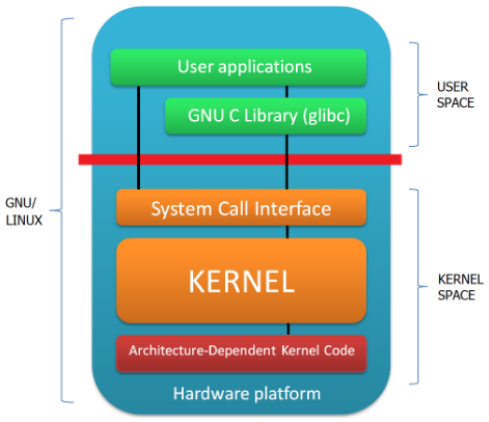


Figure 2: Linux User and Kernel space

[67]

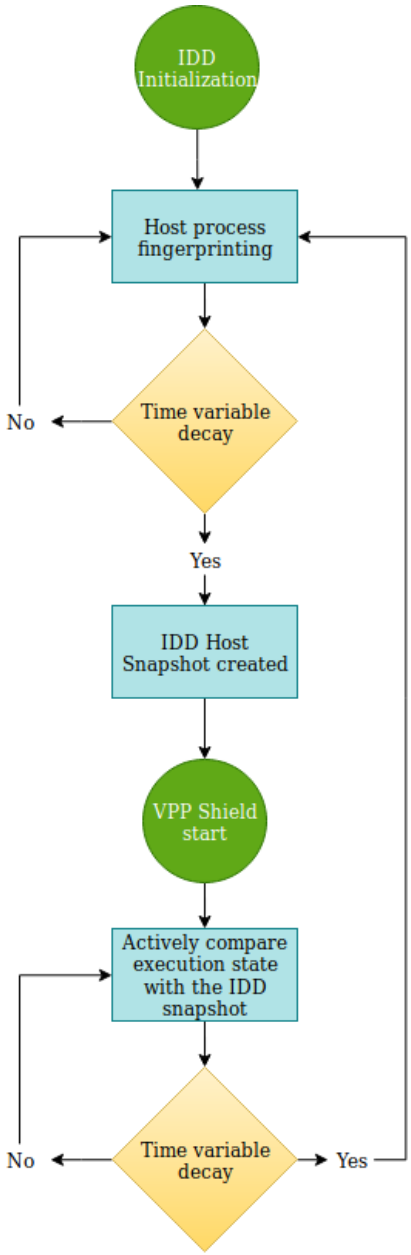
Ended my day with [71].

Next time we will start defining Kaizen's architecture and mathematical model.

13th of May

Another area where Kaizen could prove useful would be containerization where it would serve as a blueprint for the expected container behavior. Anyway, all that could prove to be just fluff in practice as there are a couple of defense mechanisms currently under development concerning microservices. Plus, don't forget that the execution of a container is quite different from a program running under a native Linux distro.

Anyway, let us now define in broad strokes the model Kaizen will use.



The time variable decay calculation will prove to be the most critical part of our project for it's there where we will need to nail the performance issue. If the time frame it's set too broad, it will flood us with false positives. If it is set too narrow, it will use too many resources. We will need to time it right.

Let us now enumerate all variables that Kaizen will use:

Process state: Susceptible (uninfected), Infected, Immune (VPP). In order to encode this information, we will create a vector called process stat with the following binary codification:

P = [1, 0, 0] = susceptible
P = [0, 1, 0] = infected
P = [0, 0, 1] = immune

The time variable decay will have the following properties, encoded in a vector as well: Duration, Incubation t0, Incubation t1, Symptoms t0, Latent t0, Latent t1, Infectious t0, Infectious t1, Immunization.

Duration = time, in seconds, that we will set manually in order to execute the snapshot recursively (a)
Incubation t0 = start of the incubation period. By incubation we mean the period where we observe a new susceptible process spawned (b0)
Incubation t1 = end of the incubation period, when the susceptible process changes status to infected or immune (b1)
Symptoms t0 = timestamp of when we start observing deviant behavior (c)
Latent t0 = UID of process that changed status from susceptible to infected (d0)
Latent t1 = UID of process that changed status from immune to susceptible (d1)
Infectious t0 = timestamp containing the start of the infectious period (e0)
Infectious t1 = timestamp containing the end of the infectious period (e1)
Immunization = binary value – 1 or 0 – yes or no respectively (f)

T = [a, b0, b1, c, d0, d1, e0, e1, f]

Virulent transmission type: direct or indirect. Indirect transmission deals with situations where a intermediate host is used. We'll use a binary value.

D = [0 , 1]

Seasonality tries to capture the manner in which the temporal dimension can be used to trigger certain events or even mask some for that matter. In order to keep it casual, for now we will just define 3 stages:
Booting up – 0 to 59.9 seconds
Updating – starts with t0 when the package update procedure begins
Post Update – ends when the update sequence ended

We will just make use of only variable to describe stage changes (Booting up, Updating, Post Update):

S = [0, 1 ,2]

The following vars will describe the host and will be used to create the IDD host fingerprint:

a = Host population size – number of active processes
b = Proportion of susceptible and infected processes
c = Population-level of spawned processes (this could capture if a large number of processes are started and ended in a quick succession)
d = Intrinsic growth rate of susceptible processes
e = Intrinsic growth rate of infected processes
f = Per-process immunization success rate
g = Per-process recovery rate (processes that change status from infected to immune)
h = Population density of susceptible processes

We will note this down with:

H= [a, b, c, d, e, f, g, h]

In regard to performance, we will write down a control cost-related variable. This will most likely be in a direct-proportional relationship with the time decay variable.

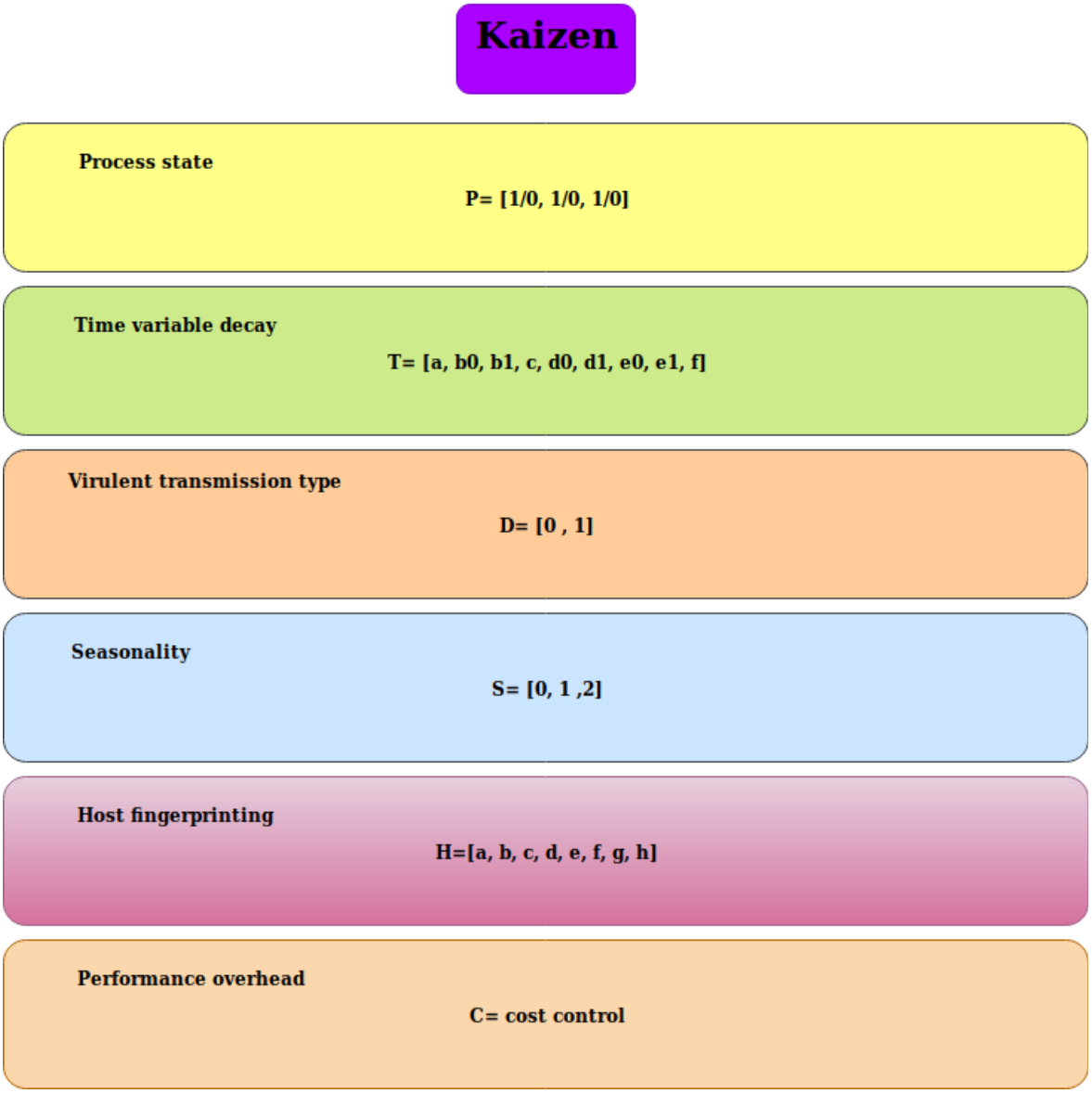
C = cost control

Let's wrap it up:

P = [1/0, 1/0, 1/0]
T = [a, b0, b1, c, d0, d1, e0, e1, f]
D = [0 , 1]
S = [0, 1 ,2]
H= [a, b, c, d, e, f, g, h]
C = cost control

On the next page I will insert the diagram containing all the variables and their respective categories.

That's it for today. Satisfied with the project so far. We are close to finishing the R II stage and starting D I . I think we will spend only one more day wrapping things up in the current stage.



14th of May

Today I have reviewed all the progress made so far and I am confident enough to end the R II stage. Next time we will enter D I stage, development time. I am pretty sure some of our theoretical assumptions and ideas will suffer minor modifications when turned into practice.

16th of May

Today we start the D I stage.

We will begin building the IDD engine, our first objective being host fingerprinting. We will divide this task in several smaller functions and go from there.

In order to gain confidence we will tackle one of the Kaizen's layers at the time. I'll choose seasonality. As a reminder, this layer simply indicates 3 distinct time periods and state of the Linux distro – booting up, updating and post-update.

I will develop Kaizen's PoC for Debian-based systems.

(base) mihai@mihai-Lenovo-V330-14IKB:~/Kaizen/seasonality$ ./a.out
The local date and time is: Thu May 16 17:46:26 2019
(base) mihai@mihai-Lenovo-V330-14IKB:~/Kaizen/seasonality$

Used thread/chrono for setting up the timer:

```
(base) mihai@mihai-Lenovo-V330-14IKB:~/Kaizen/seasonality$ ls
a.out  bootingUp.cpp
(base) mihai@mihai-Lenovo-V330-14IKB:~/Kaizen/seasonality$ g++ bootingUp.cpp
(base) mihai@mihai-Lenovo-V330-14IKB:~/Kaizen/seasonality$ ./a.out
The local date and time is: Thu May 16 17:59:15 2019

Booting up sequence complete. Thu May 16 18:00:15 2019
```

Will continue next time. We need to detect the apt package manager in order to finish the seasonality aspect of our little AV.

20th of May

After additional research, I have decided to automatically update the distro and change the status to 'updated' afterwards instead of detecting apt instance as I intended initially. This way is more efficient, simple and secure.

sudo dpkg --configure -a
sudo apt-get install -f
sudo apt-get update
sudo apt-get upgrade
sudo apt-get dist-upgrade
sudo apt-get --purge autoremove
sudo apt-get autoclean

Added this to a neatly edited bash script file.

Today we have completed the "seasonality" layer. :)

21st of May

Yesterday we have finished one of the six Kaizen layers. Today we'll continue with another one. I choose the performance overhead layer. This layer is responsible with profiling our AVs performance in regard to its aggressiveness, which in itself is directly proportional with the time variable decay.

To put it simply, the smaller the time decay variable, the more often our AV will try to fingerprint the host, thus use more resources. Pretty intuitive. But how can we measure that?

https://dev.to/etcwilde/perf---perfect-profiling-of-cc-on-linux-of

http://valgrind.org/docs/manual/cl-manual.html

https://www.gnu.org/software/gdb/

http://www.brendangregg.com/perf.html

https://github.com/namhyung/uftrace

After carefully analyzing all our options, I have decided that Kaizen's performance overhead layer will be accomplished via a 3rd party performance auditing tool.

I have settled for uftrace. Let's try to run it on our small seasonality bootingUp script. We first need to change our compiling flag to -pg:

```
(base) mihai@mihai-Lenovo-V330-14IKB:~/Kaizen/seasonality$ uftrace a.out
uftrace: /home/mihai/uftrace/cmds/record.c:1618:check_binary
 ERROR: Can't find 'mcount' symbol in the 'a.out'.
        It seems not to be compiled with -pg or -finstrument-functions flag
        which generates traceable code.  Please check your binary file.
```

```
(base) mihai@mihai-Lenovo-V330-14IKB:~/Kaizen/seasonality$ g++ -pg bootingUp.cpp
(base) mihai@mihai-Lenovo-V330-14IKB:~/Kaizen/seasonality$ ./a.out
The local date and time is: Tue May 21 12:31:27 2019

Booting up sequence complete. Tue May 21 12:32:27 2019

(base) mihai@mihai-Lenovo-V330-14IKB:~/Kaizen/seasonality$ uftrace a.out
The local date and time is: Tue May 21 12:34:25 2019

Booting up sequence complete. Tue May 21 12:35:25 2019

# DURATION     TID     FUNCTION
            [ 15353] | _GLOBAL__sub_I_main() {
            [ 15353] |   __static_initialization_and_destruction_0() {
 221.392 us [ 15353] |     std::ios_base::Init::Init();
   0.610 us [ 15353] |     __cxa_atexit();
 225.921 us [ 15353] |   } /* __static_initialization_and_destruction_0 */
 227.442 us [ 15353] | } /* _GLOBAL__sub_I_main */
            [ 15353] | main() {
   0.363 us [ 15353] |   time();
  76.947 us [ 15353] |   ctime();
  23.029 us [ 15353] |   std::operator<<();
  20.911 us [ 15353] |   std::operator<<();
  18.294 us [ 15353] |   std::basic_ostream::operator<<();
```

```
  0.262 us [ 15353] |   std::chrono::duration::duration();
            [ 15353] |   std::this_thread::sleep_for() {
            [ 15353] |     std::chrono::duration::zero() {
  0.201 us [ 15353] |       std::chrono::duration_values::zero();
  0.208 us [ 15353] |       std::chrono::duration::duration();
  1.600 us [ 15353] |     } /* std::chrono::duration::zero */
            [ 15353] |     std::chrono::operator<=() {
            [ 15353] |       std::chrono::operator<() {
  0.165 us [ 15353] |         std::chrono::duration::count();
  0.165 us [ 15353] |         std::chrono::duration::count();
  1.321 us [ 15353] |       } /* std::chrono::operator< */
  1.790 us [ 15353] |     } /* std::chrono::operator<= */
            [ 15353] |     std::chrono::duration_cast() {
            [ 15353] |       std::chrono::__duration_cast_impl::__cast() {
  0.160 us [ 15353] |         std::chrono::duration::count();
:...skipping...
# DURATION     TID     FUNCTION
            [ 15353] | _GLOBAL__sub_I_main() {
            [ 15353] |   __static_initialization_and_destruction_0() {
 221.392 us [ 15353] |     std::ios_base::Init::Init();
  0.610 us [ 15353] |     __cxa_atexit();
 225.921 us [ 15353] |   } /* __static_initialization_and_destruction_0 */
 227.442 us [ 15353] | } /* _GLOBAL__sub_I_main */
            [ 15353] | main() {
  0.363 us [ 15353] |   time();
 76.947 us [ 15353] |   ctime();
 23.029 us [ 15353] |   std::operator<<();
 20.911 us [ 15353] |   std::operator<<();
 18.294 us [ 15353] |   std::basic_ostream::operator<<();
  0.262 us [ 15353] |   std::chrono::duration::duration();
            [ 15353] |   std::this_thread::sleep_for() {
            [ 15353] |     std::chrono::duration::zero() {
  0.201 us [ 15353] |       std::chrono::duration_values::zero();
  0.208 us [ 15353] |       std::chrono::duration::duration();
  1.600 us [ 15353] |     } /* std::chrono::duration::zero */
            [ 15353] |     std::chrono::operator<=() {
            [ 15353] |       std::chrono::operator<() {
  0.165 us [ 15353] |         std::chrono::duration::count();
  0.165 us [ 15353] |         std::chrono::duration::count();
  1.321 us [ 15353] |       } /* std::chrono::operator< */
  1.790 us [ 15353] |     } /* std::chrono::operator<= */
            [ 15353] |     std::chrono::duration_cast() {
            [ 15353] |       std::chrono::__duration_cast_impl::__cast() {
  0.160 us [ 15353] |         std::chrono::duration::count();
  0.166 us [ 15353] |         std::chrono::duration::duration();
  1.320 us [ 15353] |       } /* std::chrono::__duration_cast_impl::__cast */
  1.724 us [ 15353] |     } /* std::chrono::duration_cast */
            [ 15353] |     std::chrono::operator-() {
  0.167 us [ 15353] |       std::chrono::duration::count();
            [ 15353] |       std::chrono::duration::duration() {
```

```
          [ 15353] |         std::chrono::duration_cast() {
          [ 15353] |           std::chrono::__duration_cast_impl::__cast() {
  0.161 us [ 15353] |             std::chrono::duration::count();
  0.169 us [ 15353] |             std::chrono::duration::duration();
  1.443 us [ 15353] |           } /* std::chrono::__duration_cast_impl::__cast */
  1.887 us [ 15353] |         } /* std::chrono::duration_cast */
  0.160 us [ 15353] |         std::chrono::duration::count();
  2.861 us [ 15353] |       } /* std::chrono::duration::duration */
  0.153 us [ 15353] |       std::chrono::duration::count();
  0.159 us [ 15353] |       std::chrono::duration::duration();
  4.845 us [ 15353] |     } /* std::chrono::operator- */
          [ 15353] |     std::chrono::duration_cast() {
          [ 15353] |       std::chrono::__duration_cast_impl::__cast() {
  0.156 us [ 15353] |         std::chrono::duration::count();
  0.155 us [ 15353] |         std::chrono::duration::duration();
  1.304 us [ 15353] |       } /* std::chrono::__duration_cast_impl::__cast */
  1.718 us [ 15353] |     } /* std::chrono::duration_cast */
  0.159 us [ 15353] |     std::chrono::duration::count();
  0.186 us [ 15353] |     std::chrono::duration::count();
  1.000  m [ 15353] |     nanosleep();
  1.000  m [ 15353] |   } /* std::this_thread::sleep_for */
  0.715 us [ 15353] |   time();
 49.596 us [ 15353] |   ctime();
  6.947 us [ 15353] |   std::operator<<();
 27.353 us [ 15353] |   std::operator<<();
  7.928 us [ 15353] |   std::basic_ostream::operator<<();
  1.000  m [ 15353] | } /* main */
```

Let’s go further:

```
(base) mihai@mihai-Lenovo-V330-14IKB:~/Kaizen/seasonality$ uftrace report a.out
  Total time   Self time       Calls  Function
  ==========  ==========  ==========  ====================
    1.000  m    5.697 us           1  main
    1.000  m    3.961 us           1  std::this_thread::sleep_for
    1.000  m    1.000  m           1  nanosleep
   87.243 us    0.547 us           1  _GLOBAL__sub_I_main
   86.696 us    1.345 us           1  __static_initialization_and_destruction_0
   85.145 us   85.145 us           1  std::ios_base::Init::Init
   79.074 us   79.074 us           2  ctime
   46.637 us   46.637 us           4  std::operator<<
   14.325 us   14.325 us           2  std::basic_ostream::operator<<
    2.065 us    0.551 us           3  std::chrono::duration_cast
    1.889 us    0.591 us           1  std::chrono::operator-
    1.570 us    0.766 us           7  std::chrono::duration::duration
    1.514 us    1.137 us           3  std::chrono::__duration_cast_impl::__cast
    1.024 us    1.024 us           2  time
    0.729 us    0.189 us           1  std::chrono::operator<=
    0.638 us    0.638 us          10  std::chrono::duration::count
```

```
   0.598 us    0.442 us          1  std::chrono::duration::zero
   0.540 us    0.401 us          1  std::chrono::operator<
   0.206 us    0.206 us          1  __cxa_atexit
   0.077 us    0.077 us          1  std::chrono::duration_values::zero
```

System info:

```
# system information
# ==================
# program version     : v0.9.2-146-geab2 ( python perf sched )
# recorded on         : Tue May 21 12:41:25 2019
# cmdline             : uftrace record a.out
# cpu info            : Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
# number of cpus      : 8 / 8 (online / possible)
# memory info         : 0.9 / 7.5 GB (free / total)
# system load         : 0.41 / 0.75 / 0.83 (1 / 5 / 15 min)
# kernel version      : Linux 4.15.0-50-generic
# hostname            : mihai-Lenovo-V330-14IKB
# distro              : "Linux Mint 19"
#
# process information
# ===================
# number of tasks     : 1
# task list           : 15591(a.out)
# exe image           : /home/mihai/Kaizen/seasonality/a.out
# build id            : 9417a5ee5b395eeafd750b683e869269ed7d3d0a
# pattern             : regex
# exit status         : exited with code: 0
# elapsed time        : 60.003014473 sec
# cpu time            : 0.001 / 0.002 sec (sys / user)
# context switch      : 2 / 0 (voluntary / involuntary)
# max rss             : 4112 KB
# page fault          : 0 / 209 (major / minor)
# disk iops           : 0 / 8 (read / write)
```

Dump:

```
uftrace file header: magic         = 4674726163652100
uftrace file header: version       = 4
uftrace file header: header size   = 40
uftrace file header: endian        = 1 (little)
uftrace file header: class         = 2 (64 bit)
uftrace file header: features      = 0x263 (PLTHOOK | TASK_SESSION | SYM_REL_ADDR |
MAX_STACK | AUTO_ARGS)
uftrace file header: info          = 0x3bff

reading 15591.dat
7693.675214926  15591: [entry] _GLOBAL__sub_I_main(562eed0c2cc1) depth: 0
```

```
7693.675215239  15591: [entry] __static_initialization_and_destruction_0(562eed0c2c76) depth: 1
7693.675215548  15591: [entry] std::ios_base::Init::Init(562eed0c29d0) depth: 2
7693.675300693  15591: [exit ] std::ios_base::Init::Init(562eed0c29d0) depth: 2
7693.675301496  15591: [entry] __cxa_atexit(562eed0c2970) depth: 2
7693.675301702  15591: [exit ] __cxa_atexit(562eed0c2970) depth: 2
7693.675301935  15591: [exit ] __static_initialization_and_destruction_0(562eed0c2c76) depth: 1
7693.675302169  15591: [exit ] _GLOBAL__sub_I_main(562eed0c2cc1) depth: 0
7693.675302542  15591: [entry] main(562eed0c2b68) depth: 0
7693.675302657  15591: [entry] time(562eed0c2980) depth: 1
7693.675302772  15591: [exit ] time(562eed0c2980) depth: 1
7693.675303007  15591: [entry] ctime(562eed0c2960) depth: 1
7693.675334689  15591: [exit ] ctime(562eed0c2960) depth: 1
7693.675335072  15591: [entry] std::operator<<(562eed0c2990) depth: 1
7693.675343726  15591: [exit ] std::operator<<(562eed0c2990) depth: 1
7693.675343902  15591: [entry] std::operator<<(562eed0c2990) depth: 1
7693.675350995  15591: [exit ] std::operator<<(562eed0c2990) depth: 1
7693.675351183  15591: [entry] std::basic_ostream::operator<<(562eed0c29b0) depth: 1
7693.675357447  15591: [exit ] std::basic_ostream::operator<<(562eed0c29b0) depth: 1
7693.675357616  15591: [entry] std::chrono::duration::duration(562eed0c2d78) depth: 1
7693.675357737  15591: [exit ] std::chrono::duration::duration(562eed0c2d78) depth: 1
7693.675357890  15591: [entry] std::this_thread::sleep_for(562eed0c3186) depth: 1
7693.675357969  15591: [entry] std::chrono::duration::zero(562eed0c2dc9) depth: 2
7693.675358038  15591: [entry] std::chrono::duration_values::zero(562eed0c2d03) depth: 3
7693.675358115  15591: [exit ] std::chrono::duration_values::zero(562eed0c2d03) depth: 3
7693.675358339  15591: [entry] std::chrono::duration::duration(562eed0c2da2) depth: 3
7693.675358418  15591: [exit ] std::chrono::duration::duration(562eed0c2da2) depth: 3
7693.675358567  15591: [exit ] std::chrono::duration::zero(562eed0c2dc9) depth: 2
7693.675358705  15591: [entry] std::chrono::operator<=(562eed0c2e1c) depth: 2
7693.675358775  15591: [entry] std::chrono::operator<(562eed0c2e66) depth: 3
7693.675358855  15591: [entry] std::chrono::duration::count(562eed0c2e4a) depth: 4
7693.675358930  15591: [exit ] std::chrono::duration::count(562eed0c2e4a) depth: 4
7693.675359092  15591: [entry] std::chrono::duration::count(562eed0c2e4a) depth: 4
7693.675359156  15591: [exit ] std::chrono::duration::count(562eed0c2e4a) depth: 4
7693.675359315  15591: [exit ] std::chrono::operator<(562eed0c2e66) depth: 3
7693.675359434  15591: [exit ] std::chrono::operator<=(562eed0c2e1c) depth: 2
7693.675359570  15591: [entry] std::chrono::duration_cast(562eed0c2f5f) depth: 2
7693.675359641  15591: [entry] std::chrono::__duration_cast_impl::__cast(562eed0c2edd) depth: 3
7693.675359708  15591: [entry] std::chrono::duration::count(562eed0c2e4a) depth: 4
7693.675359768  15591: [exit ] std::chrono::duration::count(562eed0c2e4a) depth: 4
7693.675359936  15591: [entry] std::chrono::duration::duration(562eed0c2d34) depth: 4
7693.675360008  15591: [exit ] std::chrono::duration::duration(562eed0c2d34) depth: 4
7693.675360129  15591: [exit ] std::chrono::__duration_cast_impl::__cast(562eed0c2edd) depth: 3
7693.675360243  15591: [exit ] std::chrono::duration_cast(562eed0c2f5f) depth: 2
7693.675360357  15591: [entry] std::chrono::operator-(562eed0c3067) depth: 2
7693.675360426  15591: [entry] std::chrono::duration::count(562eed0c2e4a) depth: 3
7693.675360486  15591: [exit ] std::chrono::duration::count(562eed0c2e4a) depth: 3
7693.675360651  15591: [entry] std::chrono::duration::duration(562eed0c3004) depth: 3
7693.675360717  15591: [entry] std::chrono::duration_cast(562eed0c2fe4) depth: 4
7693.675360790  15591: [entry] std::chrono::__duration_cast_impl::__cast(562eed0c2f7f) depth: 5
7693.675360854  15591: [entry] std::chrono::duration::count(562eed0c2d5c) depth: 6
```

```
7693.675360922  15591: [exit ] std::chrono::duration::count(562eed0c2d5c) depth: 6
7693.675361151  15591: [entry] std::chrono::duration::duration(562eed0c2da2) depth: 6
7693.675361211  15591: [exit ] std::chrono::duration::duration(562eed0c2da2) depth: 6
7693.675361348  15591: [exit ] std::chrono::__duration_cast_impl::__cast(562eed0c2f7f) depth: 5
7693.675361461  15591: [exit ] std::chrono::duration_cast(562eed0c2fe4) depth: 4
7693.675361571  15591: [entry] std::chrono::duration::count(562eed0c2e4a) depth: 4
7693.675361631  15591: [exit ] std::chrono::duration::count(562eed0c2e4a) depth: 4
7693.675361767  15591: [exit ] std::chrono::duration::duration(562eed0c3004) depth: 3
7693.675361884  15591: [entry] std::chrono::duration::count(562eed0c2e4a) depth: 3
7693.675361944  15591: [exit ] std::chrono::duration::count(562eed0c2e4a) depth: 3
7693.675362057  15591: [entry] std::chrono::duration::duration(562eed0c2da2) depth: 3
7693.675362119  15591: [exit ] std::chrono::duration::duration(562eed0c2da2) depth: 3
7693.675362246  15591: [exit ] std::chrono::operator-(562eed0c3067) depth: 2
7693.675362358  15591: [entry] std::chrono::duration_cast(562eed0c3166) depth: 2
7693.675362424  15591: [entry] std::chrono::__duration_cast_impl::__cast(562eed0c3101) depth: 3
7693.675362490  15591: [entry] std::chrono::duration::count(562eed0c2e4a) depth: 4
7693.675362547  15591: [exit ] std::chrono::duration::count(562eed0c2e4a) depth: 4
7693.675362706  15591: [entry] std::chrono::duration::duration(562eed0c2ce0) depth: 4
7693.675362766  15591: [exit ] std::chrono::duration::duration(562eed0c2ce0) depth: 4
7693.675362892  15591: [exit ] std::chrono::__duration_cast_impl::__cast(562eed0c3101) depth: 3
7693.675363006  15591: [exit ] std::chrono::duration_cast(562eed0c3166) depth: 2
7693.675363122  15591: [entry] std::chrono::duration::count(562eed0c2d5c) depth: 2
7693.675363184  15591: [exit ] std::chrono::duration::count(562eed0c2d5c) depth: 2
7693.675363294  15591: [entry] std::chrono::duration::count(562eed0c2d18) depth: 2
7693.675363366  15591: [exit ] std::chrono::duration::count(562eed0c2d18) depth: 2
7693.675363536  15591: [entry] nanosleep(562eed0c2950) depth: 2
7753.675461542  15591: [exit ] nanosleep(562eed0c2950) depth: 2
7753.675464528  15591: [exit ] std::this_thread::sleep_for(562eed0c3186) depth: 1
7753.675466334  15591: [entry] time(562eed0c2980) depth: 1
7753.675467243  15591: [exit ] time(562eed0c2980) depth: 1
7753.675467836  15591: [entry] ctime(562eed0c2960) depth: 1
7753.675515228  15591: [exit ] ctime(562eed0c2960) depth: 1
7753.675515737  15591: [entry] std::operator<<(562eed0c2990) depth: 1
7753.675522129  15591: [exit ] std::operator<<(562eed0c2990) depth: 1
7753.675522504  15591: [entry] std::operator<<(562eed0c2990) depth: 1
7753.675547002  15591: [exit ] std::operator<<(562eed0c2990) depth: 1
7753.675547455  15591: [entry] std::basic_ostream::operator<<(562eed0c29b0) depth: 1
7753.675555516  15591: [exit ] std::basic_ostream::operator<<(562eed0c29b0) depth: 1
7753.675556058  15591: [exit ] main(562eed0c2b68) depth: 0
```

Graph:

```
# Function Call Graph for 'a.out' (session: dd168ff0e9f16e19)
========== FUNCTION CALL GRAPH ==========
# TOTAL TIME   FUNCTION
   1.000  m : (1) a.out
  87.243 us :  +-(1) _GLOBAL__sub_I_main
  86.696 us :  | (1) __static_initialization_and_destruction_0
  85.145 us :  |  +-(1) std::ios_base::Init::Init
            :  |  |
```

```
  0.206 us :  |  +-(1) __cxa_atexit
           :  |
  1.000  m :  +-(1) main
  1.024 us :     +-(2) time
           :     |
 79.074 us :     +-(2) ctime
           :     |
 46.637 us :     +-(4) std::operator<<
           :     |
 14.325 us :     +-(2) std::basic_ostream::operator<<
           :     |
  0.121 us :     +-(1) std::chrono::duration::duration
           :     |
  1.000  m :     +-(1) std::this_thread::sleep_for
  0.598 us :        +-(1) std::chrono::duration::zero
  0.077 us :        |  +-(1) std::chrono::duration_values::zero
           :        | |
  0.079 us :        |  +-(1) std::chrono::duration::duration
           :        |
  0.729 us :        +-(1) std::chrono::operator<=
  0.540 us :        | (1) std::chrono::operator<
  0.139 us :        | (2) std::chrono::duration::count
           :        |
  1.321 us :        +-(2) std::chrono::duration_cast
  0.956 us :        | (2) std::chrono::__duration_cast_impl::__cast
  0.117 us :        |  +-(2) std::chrono::duration::count
           :        | |
  0.132 us :        |  +-(2) std::chrono::duration::duration
           :        |
  1.889 us :        +-(1) std::chrono::operator-
  0.120 us :        |  +-(2) std::chrono::duration::count
           :        | |
  1.178 us :        |  +-(2) std::chrono::duration::duration
  0.744 us :        |     +-(1) std::chrono::duration_cast
  0.558 us :        |     | (1) std::chrono::__duration_cast_impl::__cast
  0.068 us :        |     |  +-(1) std::chrono::duration::count
           :        |     | |
  0.060 us :        |     |  +-(1) std::chrono::duration::duration
           :        |     |
  0.060 us :        |     +-(1) std::chrono::duration::count
           :        |
  0.134 us :        +-(2) std::chrono::duration::count
           :        |
  1.000  m :        +-(1) nanosleep
```

Well, there you have it, our 2nd layer is covered. 4 more to go :).

22nd of May

Next up we will tackle the process state layer.

As a reminder, it simply needs to switch between 3 states – susceptible, immune or infected. This can be easily done with a vector, 3 of them to be specific. Each vector will contain the PID of the process. I think this is the simplest thing to do.

Until we tackle the host fingerprinting layer we cannot decide which data structure to settle for.

29th of July

Hi again! After quite a lot of time I am back! In the meantime I've got myself a bunch of certificates and enrolled back at the university; I even managed to get a scholarship, being in the top 10% of the students admitted. Great news!

Now, I want to refresh my mind a bit for today so I'll try to buy me some more books on malware analysis if something new popped up. Tomorrow I should be ready to continue from where we left off.

I got me "Mastering Malware Analysis by Alexey Kleymenov, Amr Thabet". This should get me back on track. The 2 months break starts to show its effects :).

Learned a bit more about threat intelligence techniques such as using the TimeDateStamp info from the file's header. This can leak information about the country of origin of the said piece of malware – one can analyze the working hours and patterns of compilation timestamps and try to correlate that with the usual 9 to 5 work schedule.

Spent some time on packers as well.

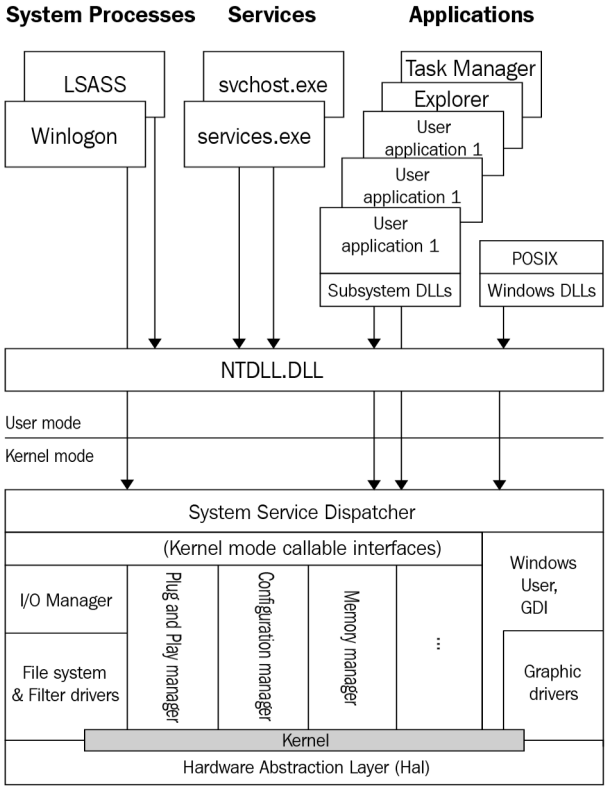
Found some great info on detecting sandboxes and virtual machines:

"**CPUID hypervisor bit**: The CPUID instruction returns information about the CPU and provides a leaf/ID of this information in `eax`. For leaf `0x01 (eax = 1)`, the CPUID sets bit 31 to 1, indicating that the operating system is running inside a virtual machine or a hypervisor.

- **Virtualization brand**: With the CPUID instruction, for some virtualization tools, given `eax = 0x40000000`, it could return the name of the virtualization tool, such as Microsoft HV or VMware in EBX, EDX, and ECX.

- **MMX registers**: MMX registers are a set of registers that were introduced by Intel that help speed up graphics calculations. Some virtualization tools don't support them. Some malware or packers use them for unpacking in order to detect or avoid running on a virtual machine. " [73]

As discussed, we are only focusing on Linux yet I have discovered a very clean diagram describing Windows' design schema. I need to save it here for later :D . [73]



I have dedicated a couple of minutes to studying the Android security model as well as runtime environment.

I will stop here today. We will resume next time with the D I stage. As a reminder, we have finished 2 out of 6 modules.

30th of July

Today we will take our first shot at the host fingerprinting module.

a = Host population size – number of active processes
b = Proportion of susceptible and infected processes
c = Population-level of spawned processes (this could capture if a large number of processes are started and ended in a quick succession)
d = Intrinsic growth rate of susceptible processes
e = Intrinsic growth rate of infected processes
f = Per-process immunization success rate
g = Per-process recovery rate (processes that change status from infected to immune)
h = Population density of susceptible processes

Let's see how can we make this happen.

I struggled for about 3 hours to no avail. I am trying to parse the proc folder and list all folders that contain digits in it. Pretty frustrating so far.

By the end of the day, I posted one question on stackoverflow… Wish me luck :D .

1st of August

After more research, I have failed to find a reliable C++ library to get info on Linux processes. I did find some in C though yet the one that caught my attention as the most usable one was a python library with extended support for Linux…

Don't know what to do at this point… Should I switch to python the whole project… ?

As of today, I have decided to switch the project to Python. All previous progress will remain in a separate folder "oldKaizen" just for reference. This decision will make Kaizen's development much more easier and extend its capabilities without any significant downsides.

2nd of August

Today is a fresh new start for Kaizen. The library I'm talking about is psutil.

I have followed the basic "get started" snippets and managed to detect a process by its name, in our case, Firefox:

```
(base) mihai@mihai-Lenovo-V330-14IKB:~/Kaizen/newKaizen$ python hostFingerprinting.py
*** Check if a process is running or not ***
Firefox instance running
*** Find PIDs of a running process by Name ***
Process Exists | PID and other details are
(7989, 'firefox', '2019-08-02 15:32:42')
** Find running process by name using List comprehension **
psutil.Process(pid=7989, name='firefox', started='15:32:42')
```

Good start so far, way better than what I had to deal with in C or C++…

12th of August

Played a bit with Sophos' Linux AV version, ran some minor tests. Today I have added the disinfection capability, namely killing a process tree.

13th of August

Added CRC capabilities for signature creation & identification.

Seems like this was a stupid idea… CRC appears to be biased.

Check this out:

https://eklitzke.org/crcs-vs-hash-functions

“It’s inappropriate to use a CRC in place of a general purpose hash function because CRCs usually have biased output. It’s equally inappropriate to use a general purpose hash function in place of a CRC because general purpose hash functions usually do not make any guarantees on the conditions under which hash collisions can occur.”

I think I’ll settle for Blake2b.

https://blake2.net/

In the end I choose SHA1, SHA256, SHA512.

14th of August

Added host fingerprinting (lists all active processes) and process sniper (search for a specific process).

```
*** Create a list of all running processes ***
{'name': 'systemd', 'cpu_percent': 0.0, 'pid': 1}
{'name': 'kthreadd', 'cpu_percent': 0.0, 'pid': 2}
{'name': 'kworker/0:0H', 'cpu_percent': 0.0, 'pid': 4}
{'name': 'mm_percpu_wq', 'cpu_percent': 0.0, 'pid': 6}
{'name': 'ksoftirqd/0', 'cpu_percent': 0.0, 'pid': 7}
{'name': 'rcu_sched', 'cpu_percent': 0.0, 'pid': 8}
{'name': 'rcu_bh', 'cpu_percent': 0.0, 'pid': 9}
{'name': 'migration/0', 'cpu_percent': 0.0, 'pid': 10}
{'name': 'watchdog/0', 'cpu_percent': 0.0, 'pid': 11}
{'name': 'cpuhp/0', 'cpu_percent': 0.0, 'pid': 12}
{'name': 'cpuhp/1', 'cpu_percent': 0.0, 'pid': 13}
{'name': 'watchdog/1', 'cpu_percent': 0.0, 'pid': 14}
{'name': 'migration/1', 'cpu_percent': 0.0, 'pid': 15}
{'name': 'ksoftirqd/1', 'cpu_percent': 0.0, 'pid': 16}
{'name': 'kworker/1:0', 'cpu_percent': 0.0, 'pid': 17}
{'name': 'kworker/1:0H', 'cpu_percent': 0.0, 'pid': 18}
{'name': 'cpuhp/2', 'cpu_percent': 0.0, 'pid': 19}
{'name': 'watchdog/2', 'cpu_percent': 0.0, 'pid': 20}
{'name': 'migration/2', 'cpu_percent': 0.0, 'pid': 21}
{'name': 'ksoftirqd/2', 'cpu_percent': 0.0, 'pid': 22}
{'name': 'kworker/2:0H', 'cpu_percent': 0.0, 'pid': 24}
{'name': 'cpuhp/3', 'cpu_percent': 0.0, 'pid': 25}
{'name': 'watchdog/3', 'cpu_percent': 0.0, 'pid': 26}
{'name': 'migration/3', 'cpu_percent': 0.0, 'pid': 27}
{'name': 'ksoftirqd/3', 'cpu_percent': 0.0, 'pid': 28}
{'name': 'kworker/3:0H', 'cpu_percent': 0.0, 'pid': 30}
{'name': 'cpuhp/4', 'cpu_percent': 0.0, 'pid': 31}
{'name': 'watchdog/4', 'cpu_percent': 0.0, 'pid': 32}
{'name': 'migration/4', 'cpu_percent': 0.0, 'pid': 33}
{'name': 'ksoftirqd/4', 'cpu_percent': 0.0, 'pid': 34}
{'name': 'kworker/4:0H', 'cpu_percent': 0.0, 'pid': 36}
{'name': 'cpuhp/5', 'cpu_percent': 0.0, 'pid': 37}
{'name': 'watchdog/5', 'cpu_percent': 0.0, 'pid': 38}
{'name': 'migration/5', 'cpu_percent': 0.0, 'pid': 39}
{'name': 'ksoftirqd/5', 'cpu_percent': 0.0, 'pid': 40}
```

{'name': 'kworker/5:0H', 'cpu_percent': 0.0, 'pid': 42}
{'name': 'cpuhp/6', 'cpu_percent': 0.0, 'pid': 43}
{'name': 'watchdog/6', 'cpu_percent': 0.0, 'pid': 44}
{'name': 'migration/6', 'cpu_percent': 0.0, 'pid': 45}
{'name': 'ksoftirqd/6', 'cpu_percent': 0.0, 'pid': 46}
{'name': 'kworker/6:0H', 'cpu_percent': 0.0, 'pid': 48}
{'name': 'cpuhp/7', 'cpu_percent': 0.0, 'pid': 49}
{'name': 'watchdog/7', 'cpu_percent': 0.0, 'pid': 50}
{'name': 'migration/7', 'cpu_percent': 0.0, 'pid': 51}
{'name': 'ksoftirqd/7', 'cpu_percent': 0.0, 'pid': 52}
{'name': 'kworker/7:0H', 'cpu_percent': 0.0, 'pid': 54}
{'name': 'kdevtmpfs', 'cpu_percent': 0.0, 'pid': 55}
{'name': 'netns', 'cpu_percent': 0.0, 'pid': 56}
{'name': 'rcu_tasks_kthre', 'cpu_percent': 0.0, 'pid': 57}
{'name': 'kauditd', 'cpu_percent': 0.0, 'pid': 58}
{'name': 'kworker/0:1', 'cpu_percent': 0.0, 'pid': 59}
{'name': 'khungtaskd', 'cpu_percent': 0.0, 'pid': 62}
{'name': 'oom_reaper', 'cpu_percent': 0.0, 'pid': 63}
{'name': 'writeback', 'cpu_percent': 0.0, 'pid': 64}
{'name': 'kcompactd0', 'cpu_percent': 0.0, 'pid': 65}
{'name': 'ksmd', 'cpu_percent': 0.0, 'pid': 66}
{'name': 'khugepaged', 'cpu_percent': 0.0, 'pid': 67}
{'name': 'crypto', 'cpu_percent': 0.0, 'pid': 68}
{'name': 'kintegrityd', 'cpu_percent': 0.0, 'pid': 69}
{'name': 'kblockd', 'cpu_percent': 0.0, 'pid': 70}
{'name': 'kworker/4:1', 'cpu_percent': 0.0, 'pid': 72}
{'name': 'kworker/5:1', 'cpu_percent': 0.0, 'pid': 73}
{'name': 'ata_sff', 'cpu_percent': 0.0, 'pid': 76}
{'name': 'md', 'cpu_percent': 0.0, 'pid': 77}
{'name': 'edac-poller', 'cpu_percent': 0.0, 'pid': 78}
{'name': 'devfreq_wq', 'cpu_percent': 0.0, 'pid': 79}
{'name': 'watchdogd', 'cpu_percent': 0.0, 'pid': 80}
{'name': 'kswapd0', 'cpu_percent': 0.0, 'pid': 83}
{'name': 'kworker/u17:0', 'cpu_percent': 0.0, 'pid': 84}
{'name': 'ecryptfs-kthrea', 'cpu_percent': 0.0, 'pid': 85}
{'name': 'kthrotld', 'cpu_percent': 0.0, 'pid': 127}
{'name': 'acpi_thermal_pm', 'cpu_percent': 0.0, 'pid': 128}
{'name': 'ipv6_addrconf', 'cpu_percent': 0.0, 'pid': 132}
{'name': 'kstrp', 'cpu_percent': 0.0, 'pid': 141}
{'name': 'charger_manager', 'cpu_percent': 0.0, 'pid': 158}
{'name': 'nvme-wq', 'cpu_percent': 0.0, 'pid': 215}
{'name': 'scsi_eh_0', 'cpu_percent': 0.0, 'pid': 216}
{'name': 'scsi_tmf_0', 'cpu_percent': 0.0, 'pid': 217}
{'name': 'scsi_eh_1', 'cpu_percent': 0.0, 'pid': 218}
{'name': 'scsi_tmf_1', 'cpu_percent': 0.0, 'pid': 219}
{'name': 'kworker/u16:3', 'cpu_percent': 0.0, 'pid': 221}
{'name': 'i915/signal:0', 'cpu_percent': 0.0, 'pid': 222}
{'name': 'i915/signal:1', 'cpu_percent': 0.0, 'pid': 223}
{'name': 'i915/signal:2', 'cpu_percent': 0.0, 'pid': 224}
{'name': 'i915/signal:4', 'cpu_percent': 0.0, 'pid': 225}
{'name': 'raid5wq', 'cpu_percent': 0.0, 'pid': 304}
{'name': 'kworker/5:1H', 'cpu_percent': 0.0, 'pid': 358}
{'name': 'jbd2/nvme0n1p2-', 'cpu_percent': 0.0, 'pid': 360}
{'name': 'ext4-rsv-conver', 'cpu_percent': 0.0, 'pid': 361}
{'name': 'systemd-journald', 'cpu_percent': 0.0, 'pid': 418}
{'name': 'kworker/2:1H', 'cpu_percent': 0.0, 'pid': 433}
{'name': 'kworker/7:1H', 'cpu_percent': 0.0, 'pid': 449}
{'name': 'lvmetad', 'cpu_percent': 0.0, 'pid': 453}
{'name': 'systemd-udevd', 'cpu_percent': 0.0, 'pid': 473}
{'name': 'kworker/3:1H', 'cpu_percent': 0.0, 'pid': 475}

{'name': 'kworker/6:1H', 'cpu_percent': 0.0, 'pid': 476}
{'name': 'kworker/4:1H', 'cpu_percent': 0.0, 'pid': 479}
{'name': 'kworker/0:1H', 'cpu_percent': 0.0, 'pid': 501}
{'name': 'kworker/1:1H', 'cpu_percent': 0.0, 'pid': 502}
{'name': 'cfg80211', 'cpu_percent': 0.0, 'pid': 625}
{'name': 'kworker/2:2', 'cpu_percent': 0.0, 'pid': 633}
{'name': 'kworker/u17:1', 'cpu_percent': 0.0, 'pid': 642}
{'name': 'ath10k_wq', 'cpu_percent': 0.0, 'pid': 652}
{'name': 'ath10k_aux_wq', 'cpu_percent': 0.0, 'pid': 654}
{'name': 'irq/135-mei_me', 'cpu_percent': 0.0, 'pid': 672}
{'name': 'irq/80-ELAN0612', 'cpu_percent': 0.0, 'pid': 716}
{'name': 'systemd-timesyncd', 'cpu_percent': 0.0, 'pid': 790}
{'name': 'systemd-resolved', 'cpu_percent': 0.0, 'pid': 796}
{'name': 'auditd', 'cpu_percent': 0.0, 'pid': 807}
{'name': 'udisksd', 'cpu_percent': 0.0, 'pid': 915}
{'name': 'systemd-logind', 'cpu_percent': 0.0, 'pid': 926}
{'name': 'rsyslogd', 'cpu_percent': 0.0, 'pid': 930}
{'name': 'acpid', 'cpu_percent': 0.0, 'pid': 933}
{'name': 'bluetoothd', 'cpu_percent': 0.0, 'pid': 937}
{'name': 'accounts-daemon', 'cpu_percent': 0.0, 'pid': 939}
{'name': 'ModemManager', 'cpu_percent': 0.0, 'pid': 941}
{'name': 'thermald', 'cpu_percent': 0.0, 'pid': 942}
{'name': 'dbus-daemon', 'cpu_percent': 0.0, 'pid': 944}
{'name': 'wpa_supplicant', 'cpu_percent': 0.0, 'pid': 983}
{'name': 'freshclam', 'cpu_percent': 0.0, 'pid': 993}
{'name': 'avahi-daemon', 'cpu_percent': 0.0, 'pid': 999}
{'name': 'irqbalance', 'cpu_percent': 0.0, 'pid': 1015}
{'name': 'networkd-dispat', 'cpu_percent': 0.0, 'pid': 1016}
{'name': 'NetworkManager', 'cpu_percent': 0.0, 'pid': 1018}
{'name': 'cron', 'cpu_percent': 0.0, 'pid': 1023}
{'name': 'avahi-daemon', 'cpu_percent': 0.0, 'pid': 1028}
{'name': 'clamd', 'cpu_percent': 0.0, 'pid': 1033}
{'name': 'mosquitto', 'cpu_percent': 0.0, 'pid': 1038}
{'name': 'polkitd', 'cpu_percent': 0.0, 'pid': 1058}
{'name': 'savd', 'cpu_percent': 0.0, 'pid': 1092}
{'name': 'postgres', 'cpu_percent': 0.0, 'pid': 1099}
{'name': 'wicd', 'cpu_percent': 0.0, 'pid': 1110}
{'name': 'postgres', 'cpu_percent': 0.0, 'pid': 1117}
{'name': 'postgres', 'cpu_percent': 0.0, 'pid': 1118}
{'name': 'postgres', 'cpu_percent': 0.0, 'pid': 1119}
{'name': 'postgres', 'cpu_percent': 0.0, 'pid': 1121}
{'name': 'postgres', 'cpu_percent': 0.0, 'pid': 1123}
{'name': 'postgres', 'cpu_percent': 0.0, 'pid': 1124}
{'name': 'wicd-monitor', 'cpu_percent': 0.0, 'pid': 1136}
{'name': 'libvirtd', 'cpu_percent': 0.0, 'pid': 1160}
{'name': 'mongod', 'cpu_percent': 0.0, 'pid': 1164}
{'name': 'vnstatd', 'cpu_percent': 0.0, 'pid': 1169}
{'name': 'containerd', 'cpu_percent': 0.0, 'pid': 1191}
{'name': 'chronyd', 'cpu_percent': 0.0, 'pid': 1193}
{'name': 'lightdm', 'cpu_percent': 0.0, 'pid': 1194}
{'name': 'Xorg', 'cpu_percent': 0.0, 'pid': 1307}
{'name': 'agetty', 'cpu_percent': 0.0, 'pid': 1326}
{'name': 'tor', 'cpu_percent': 0.0, 'pid': 1360}
{'name': 'lightdm', 'cpu_percent': 0.0, 'pid': 1452}
{'name': 'systemd', 'cpu_percent': 0.0, 'pid': 1459}
{'name': '(sd-pam)', 'cpu_percent': 0.0, 'pid': 1463}
{'name': 'cinnamon-session', 'cpu_percent': 0.0, 'pid': 1492}
{'name': 'dbus-daemon', 'cpu_percent': 0.0, 'pid': 1505}
{'name': 'ssh-agent', 'cpu_percent': 0.0, 'pid': 1557}
{'name': 'at-spi-bus-launcher', 'cpu_percent': 0.0, 'pid': 1573}

{'name': 'dbus-daemon', 'cpu_percent': 0.0, 'pid': 1578}
{'name': 'at-spi2-registryd', 'cpu_percent': 0.0, 'pid': 1580}
{'name': 'gnome-keyring-daemon', 'cpu_percent': 0.0, 'pid': 1591}
{'name': 'csd-media-keys', 'cpu_percent': 0.0, 'pid': 1599}
{'name': 'csd-power', 'cpu_percent': 0.0, 'pid': 1600}
{'name': 'csd-keyboard', 'cpu_percent': 0.0, 'pid': 1601}
{'name': 'csd-print-notifications', 'cpu_percent': 0.0, 'pid': 1602}
{'name': 'csd-color', 'cpu_percent': 0.0, 'pid': 1603}
{'name': 'csd-housekeeping', 'cpu_percent': 0.0, 'pid': 1604}
{'name': 'csd-mouse', 'cpu_percent': 0.0, 'pid': 1605}
{'name': 'csd-background', 'cpu_percent': 0.0, 'pid': 1608}
{'name': 'csd-xrandr', 'cpu_percent': 0.0, 'pid': 1609}
{'name': 'csd-cursor', 'cpu_percent': 0.0, 'pid': 1612}
{'name': 'csd-a11y-keyboard', 'cpu_percent': 0.0, 'pid': 1624}
{'name': 'csd-clipboard', 'cpu_percent': 0.0, 'pid': 1634}
{'name': 'csd-automount', 'cpu_percent': 0.0, 'pid': 1636}
{'name': 'csd-orientation', 'cpu_percent': 0.0, 'pid': 1639}
{'name': 'csd-sound', 'cpu_percent': 0.0, 'pid': 1641}
{'name': 'csd-xsettings', 'cpu_percent': 0.0, 'pid': 1642}
{'name': 'csd-a11y-settings', 'cpu_percent': 0.0, 'pid': 1643}
{'name': 'csd-wacom', 'cpu_percent': 0.0, 'pid': 1645}
{'name': 'csd-screensaver-proxy', 'cpu_percent': 0.0, 'pid': 1649}
{'name': 'gvfsd', 'cpu_percent': 0.0, 'pid': 1656}
{'name': 'csd-locate-pointer', 'cpu_percent': 0.0, 'pid': 1657}
{'name': 'gvfsd-fuse', 'cpu_percent': 0.0, 'pid': 1663}
{'name': 'pulseaudio', 'cpu_percent': 0.0, 'pid': 1677}
{'name': 'upowerd', 'cpu_percent': 0.0, 'pid': 1680}
{'name': 'rtkit-daemon', 'cpu_percent': 0.0, 'pid': 1681}
{'name': 'colord', 'cpu_percent': 0.0, 'pid': 1701}
{'name': 'csd-printer', 'cpu_percent': 0.0, 'pid': 1715}
{'name': 'gvfs-udisks2-volume-monitor', 'cpu_percent': 0.0, 'pid': 1730}
{'name': 'dconf-service', 'cpu_percent': 0.0, 'pid': 1747}
{'name': 'gvfs-gphoto2-volume-monitor', 'cpu_percent': 0.0, 'pid': 1772}
{'name': 'gvfs-mtp-volume-monitor', 'cpu_percent': 0.0, 'pid': 1783}
{'name': 'gvfs-goa-volume-monitor', 'cpu_percent': 0.0, 'pid': 1789}
{'name': 'goa-daemon', 'cpu_percent': 0.0, 'pid': 1793}
{'name': 'goa-identity-service', 'cpu_percent': 0.0, 'pid': 1813}
{'name': 'gvfs-afc-volume-monitor', 'cpu_percent': 0.0, 'pid': 1819}
{'name': 'krfcommd', 'cpu_percent': 0.0, 'pid': 1832}
{'name': 'cinnamon-launch', 'cpu_percent': 0.0, 'pid': 1834}
{'name': 'cinnamon', 'cpu_percent': 0.0, 'pid': 1847}
{'name': 'wicd-client', 'cpu_percent': 0.0, 'pid': 1871}
{'name': 'polkit-gnome-authentication-agent-1', 'cpu_percent': 0.0, 'pid': 1872}
{'name': 'blueberry-obex-agent', 'cpu_percent': 0.0, 'pid': 1878}
{'name': 'nemo-desktop', 'cpu_percent': 0.0, 'pid': 1879}
{'name': 'nm-applet', 'cpu_percent': 0.0, 'pid': 1880}
{'name': 'cinnamon-killer', 'cpu_percent': 0.0, 'pid': 1881}
{'name': 'obexd', 'cpu_percent': 0.0, 'pid': 1899}
{'name': 'gvfsd-metadata', 'cpu_percent': 0.0, 'pid': 1908}
{'name': 'gvfsd-trash', 'cpu_percent': 0.0, 'pid': 1940}
{'name': 'dhclient', 'cpu_percent': 0.0, 'pid': 2004}
{'name': 'dockerd', 'cpu_percent': 0.0, 'pid': 2057}
{'name': 'kerneloops', 'cpu_percent': 0.0, 'pid': 2066}
{'name': 'kerneloops', 'cpu_percent': 0.0, 'pid': 2098}
{'name': 'iprt-VBoxWQueue', 'cpu_percent': 0.0, 'pid': 2163}
{'name': 'iprt-VBoxTscThr', 'cpu_percent': 0.0, 'pid': 2191}
{'name': 'collectl', 'cpu_percent': 0.0, 'pid': 2524}
{'name': 'blueberry-tray', 'cpu_percent': 0.0, 'pid': 2605}
{'name': 'cinnamon-screensaver', 'cpu_percent': 0.0, 'pid': 2608}
{'name': 'sh', 'cpu_percent': 0.0, 'pid': 2634}

{'name': 'python2', 'cpu_percent': 0.0, 'pid': 2636}
{'name': 'python2', 'cpu_percent': 0.0, 'pid': 2745}
{'name': 'rfkill', 'cpu_percent': 0.0, 'pid': 2751}
{'name': 'docker-proxy', 'cpu_percent': 0.0, 'pid': 2791}
{'name': 'docker-proxy', 'cpu_percent': 0.0, 'pid': 2805}
{'name': 'docker-proxy', 'cpu_percent': 0.0, 'pid': 2818}
{'name': 'docker-proxy', 'cpu_percent': 0.0, 'pid': 2831}
{'name': 'docker-proxy', 'cpu_percent': 0.0, 'pid': 2845}
{'name': 'docker-proxy', 'cpu_percent': 0.0, 'pid': 2859}
{'name': 'docker-proxy', 'cpu_percent': 0.0, 'pid': 2873}
{'name': 'docker-proxy', 'cpu_percent': 0.0, 'pid': 2886}
{'name': 'docker-proxy', 'cpu_percent': 0.0, 'pid': 2898}
{'name': 'containerd-shim', 'cpu_percent': 0.0, 'pid': 2905}
{'name': 'conpot', 'cpu_percent': 0.0, 'pid': 2930}
{'name': 'mintUpdate', 'cpu_percent': 0.0, 'pid': 3315}
{'name': 'applet.py', 'cpu_percent': 0.0, 'pid': 3471}
{'name': 'exim4', 'cpu_percent': 0.0, 'pid': 3481}
{'name': 'cupsd', 'cpu_percent': 0.0, 'pid': 3958}
{'name': 'cups-browsed', 'cpu_percent': 0.0, 'pid': 3960}
{'name': 'flatpak-session-helper', 'cpu_percent': 0.0, 'pid': 4175}
{'name': 'xdg-document-portal', 'cpu_percent': 0.0, 'pid': 4180}
{'name': 'xdg-permission-store', 'cpu_percent': 0.0, 'pid': 4183}
{'name': 'spotify', 'cpu_percent': 0.0, 'pid': 4442}
{'name': 'spotify', 'cpu_percent': 0.0, 'pid': 4446}
{'name': 'spotify', 'cpu_percent': 0.0, 'pid': 4458}
{'name': 'spotify', 'cpu_percent': 0.0, 'pid': 4484}
{'name': 'savscand', 'cpu_percent': 0.0, 'pid': 4576}
{'name': 'dbus-launch', 'cpu_percent': 0.0, 'pid': 4677}
{'name': 'dbus-daemon', 'cpu_percent': 0.0, 'pid': 4678}
{'name': 'packagekitd', 'cpu_percent': 0.0, 'pid': 5987}
{'name': 'kworker/3:0', 'cpu_percent': 0.0, 'pid': 6873}
{'name': 'kworker/7:1', 'cpu_percent': 0.0, 'pid': 7218}
{'name': 'kworker/u16:2', 'cpu_percent': 0.0, 'pid': 7725}
{'name': 'spotify', 'cpu_percent': 0.0, 'pid': 8097}
{'name': 'kworker/2:1', 'cpu_percent': 0.0, 'pid': 8293}
{'name': 'kworker/6:0', 'cpu_percent': 0.0, 'pid': 8314}
{'name': 'kworker/u16:0', 'cpu_percent': 0.0, 'pid': 8353}
{'name': 'kworker/4:2', 'cpu_percent': 0.0, 'pid': 8389}
{'name': 'kworker/7:0', 'cpu_percent': 0.0, 'pid': 8453}
{'name': 'kworker/6:2', 'cpu_percent': 0.0, 'pid': 8466}
{'name': 'kworker/0:2', 'cpu_percent': 0.0, 'pid': 8485}
{'name': 'kworker/3:2', 'cpu_percent': 0.0, 'pid': 8509}
{'name': 'kworker/1:2', 'cpu_percent': 0.0, 'pid': 8510}
{'name': 'kworker/u16:1', 'cpu_percent': 0.0, 'pid': 8582}
{'name': 'kworker/5:2', 'cpu_percent': 0.0, 'pid': 8584}
{'name': 'kworker/4:0', 'cpu_percent': 0.0, 'pid': 8640}
{'name': 'kworker/7:2', 'cpu_percent': 0.0, 'pid': 8659}
{'name': 'kworker/6:1', 'cpu_percent': 0.0, 'pid': 8682}
{'name': 'kworker/0:0', 'cpu_percent': 0.0, 'pid': 8759}
{'name': 'kworker/u16:4', 'cpu_percent': 0.0, 'pid': 8824}
{'name': 'kworker/1:1', 'cpu_percent': 0.0, 'pid': 8826}
{'name': 'oosplash', 'cpu_percent': 0.0, 'pid': 8836}
{'name': 'soffice.bin', 'cpu_percent': 0.0, 'pid': 8854}
{'name': 'kworker/3:1', 'cpu_percent': 0.0, 'pid': 8984}
{'name': 'kworker/5:0', 'cpu_percent': 0.0, 'pid': 8986}
{'name': 'nemo', 'cpu_percent': 0.0, 'pid': 9001}
{'name': 'gnome-terminal-server', 'cpu_percent': 0.0, 'pid': 9030}
{'name': 'bash', 'cpu_percent': 0.0, 'pid': 9037}
{'name': 'python', 'cpu_percent': 0.0, 'pid': 9060}
*** Top 5 process with highest memory usage ***

```
{'name': 'spotify', 'username': 'mihai', 'pid': 4442, 'vms': 4671.3671875}
{'name': 'cinnamon', 'username': 'mihai', 'pid': 1847, 'vms': 3666.9375}
{'name': 'spotify', 'username': 'mihai', 'pid': 4484, 'vms': 2271.76171875}
{'name': 'dockerd', 'username': 'root', 'pid': 2057, 'vms': 1964.625}
{'name': 'pulseaudio', 'username': 'mihai', 'pid': 1677, 'vms': 1905.6171875}
```

That's it for today.

16th of August

Improved a bit the process sniper. Works as expected.

Finished the disinfection module, we can now successfully kill a existing process tree.

```
{'name': 'WebExtensions', 'username': 'mihai', 'pid': 6707, 'vms': 21226.1640625
}
{'name': 'spotify', 'username': 'mihai', 'pid': 9386, 'vms': 3958.625}
{'name': 'cinnamon', 'username': 'mihai', 'pid': 1969, 'vms': 3655.12109375}
{'name': 'firefox', 'username': 'mihai', 'pid': 6599, 'vms': 3348.59765625}
{'name': 'Web Content', 'username': 'mihai', 'pid': 7542, 'vms': 2882.55078125}
```

```
Enter pid of the process tree you wish to terminate: 9386
Process tree terminated.
```

Tested it on Spotify. If the player is rendering music, it will freeze for a couple of seconds before closing the GUI. If only the interface is open, it will close gracefully.

In order to accurately test what "gracefully" means, I have added a timer. I tested killing the Spotify process tree when listening to music (thus more children processes active) and when only the GUI is active. The difference is quite dramatic:

```
(base) mihai@mihai-Lenovo-V330-14IKB:~/Kaizen/newKaizen$ python disinfection.py
Enter pid of the process tree you wish to terminate: 9491
Process tree terminated.
44.98892879486084
(base) mihai@mihai-Lenovo-V330-14IKB:~/Kaizen/newKaizen$ python disinfection.py
Enter pid of the process tree you wish to terminate: 11983
Process tree terminated.
5.416476249694824
```

It took almost 8 times longer to kill the entire tree. Reminds me of the countless times when my antivirus software freezes the Windows testing environment when trying out new malware.

I am quite satisfied with our progress made today.

19th of August

All computer security companies claim to offer extraordinary behavioral based detection technologies and close to 100% detection rates yet for the most part, the malware identification process itself is still based on manual reverse engineering, good old R&D.

Another issue that one has to keep in mind is the instruction set, that is the actual hardware that the malware piece will execute on. In our case, we focused on one operating system yet in the case of IoT where multiple processor manufacturers are competing for the market, things are bit more complicated than just the OS options (Riot OS for instance or other RTOS flavors).

With that in mind, we will only focus on a very limited use case – Linux based operating systems (Debian based to be more specific) and a 8th generation Intel processor instruction set. ELF files are the main actors on this scene.

In order to emphasize the variety of platforms the "bad guys" can use, check it out at: https://onlinedisassembler.com/static/home/index.html

Notice the variety of ARM, AVR etc.

The most commonly used tools include: objdump, ndisasm, ODA, Radare2, RetDec, Snowman, Ghidra, Vivisect, lida, Relyze, Binary ninja, Hopper, IDA.

My go-to choice for reverse engineering is https://github.com/cea-sec/miasm

Right now, Kaizen has the capability of updating the host machine, creating a fingerprint of the running processes, sniping and disinfecting (killing) a given process tree and hash files (for identification or as a rudimentary virus signature file).

I have completely given up the machine snapshot feature that I intended to use initially given the tremendous overhead that it would add – keep in mind that just killing a multiple process tree takes half a minute. This is partly Python's fault but also the lack of extensive support for Linux libraries…

So… What's left for our project? We are now closing the D I stage and entering D II, the final stage of development.

20th of August

Today I played a bit with YARA in order to generate virus signatures and rules. This seems to be quite popular among AV and threat intelligence companies. Learned a bit more about file signatures and headers as well.

https://github.com/virustotal/yara/releases

Besides signature generation, I was also thinking about some form of sandboxing - chroot jail. Check this out:

https://olivier.sessink.nl/jailkit/index.html#download

https://dwheeler.com/secure-programs/Secure-Programs-HOWTO/minimize-privileges.html

Finished my day researching a bit on the academic side, especially signature generation and malware families aggregation/ classification algorithms.

21st of August

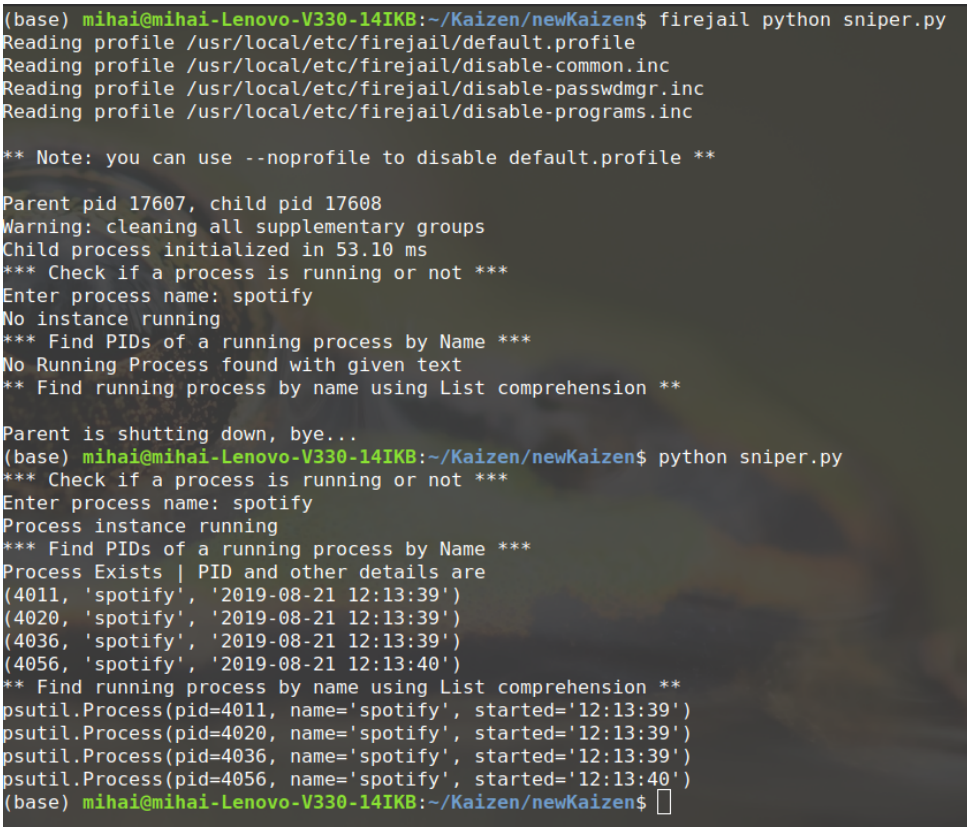
Found an interesting approach on sandboxing using containers:

https://github.com/isidentical/pysandbox/tree/master/evality

I also played a bit with PyPy.

Tried flatpak and checked out bubblewrap.

I'll have a look at firejail now. I am quite satisfied with this one, pretty intuitive. I am attaching a sandboxing proof using our process sniper:



Let's try the host fingerprinting feature:

(base) mihai@mihai-Lenovo-V330-14IKB:~/Kaizen/newKaizen$ firejail python hostFingerprinting.py
Reading profile /usr/local/etc/firejail/default.profile
Reading profile /usr/local/etc/firejail/disable-common.inc
Reading profile /usr/local/etc/firejail/disable-passwdmgr.inc
Reading profile /usr/local/etc/firejail/disable-programs.inc

** Note: you can use --noprofile to disable default.profile **

Parent pid 18030, child pid 18031

```
Warning: cleaning all supplementary groups
Child process initialized in 76.76 ms
*** Iterate over all running process and print process ID & Name ***
firejail  :::  1
python  :::  3
*** Create a list of all running processes ***
{'cpu_percent': 0.0, 'pid': 1, 'name': 'firejail'}
{'cpu_percent': 0.0, 'pid': 3, 'name': 'python'}
*** Top 5 process with highest memory usage ***
{'username': 'mihai', 'pid': 3, 'name': 'python', 'vms': 70.43359375}
{'username': 'mihai', 'pid': 1, 'name': 'firejail', 'vms': 20.58203125}

Parent is shutting down, bye…
```

I was curious to see what will happen if we try to run the bash update script:

```
(base) mihai@mihai-Lenovo-V330-14IKB:~/Kaizen/newKaizen$ firejail update.sh
Reading profile /usr/local/etc/firejail/default.profile
Reading profile /usr/local/etc/firejail/disable-common.inc
Reading profile /usr/local/etc/firejail/disable-passwdmgr.inc
Reading profile /usr/local/etc/firejail/disable-programs.inc

** Note: you can use --noprofile to disable default.profile **

Parent pid 18077, child pid 18078
Warning: cleaning all supplementary groups
Child process initialized in 76.15 ms
/bin/bash: update.sh: command not found

Parent is shutting down, bye...
```

Hmmm let us try sandboxing a browser and run a fingerprinting test. I am really curious to see the difference:

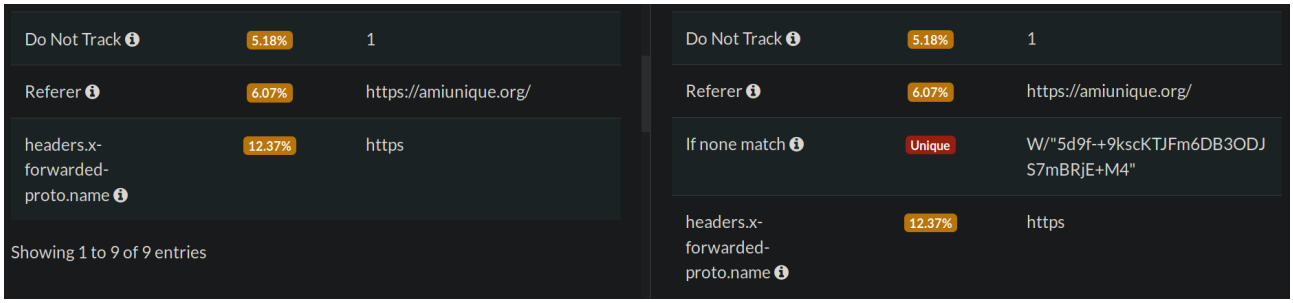
Browser Plugin Details	16.6	99643.0	Plugin 0: true; true; true; (true; true; true) (true; true; true) (true; true; true) (true; true; true) (true; true; true). Plugin 1: true; true; true; (true; true; true) (true; true; true) (true; true; true). Plugin 2: true; true; true; (true; true; true) (true; true; true) (true; true; true). Plugin 3: true; true; true; (true; true; true) (true; true; true) (true; true; true).
Time Zone	14.43	22142.89	90
Screen Size and Color Depth	9.03	523.06	NaNx1080xNaN
System Fonts	6.61	97.88	Arial, Arial Narrow, Bitstream Vera Sans Mono, Courier New, Times New Roman, Wingdings 2, Wingdings 3 (via javascript)
Are Cookies Enabled?	0.24	1.18	Yes
Limited supercookie test	0.36	1.29	DOM localStorage: Yes, DOM sessionStorage: Yes, IE userData: No
Hash of canvas fingerprint	6.63	98.95	f62f5e66c2081afc626390f1383d062b

Browser Plugin Details	6.47	88.57	Plugin 0: Shockwave Flash; Shockwave Flash 32.0 r0; libflashplayer.so; (Shockwave Flash; application/x-shockwave-flash; swf) (FutureSplash Player; application/futuresplash; spl).
Time Zone	4.1	17.2	-180
Screen Size and Color Depth	2.72	6.57	1920x1080x24
System Fonts	6.61	97.93	Arial, Arial Narrow, Bitstream Vera Sans Mono, Courier New, Times New Roman, Wingdings 2, Wingdings 3 (via javascript)
Are Cookies Enabled?	0.24	1.18	Yes
Limited supercookie test	0.36	1.29	DOM localStorage: Yes, DOM sessionStorage: Yes, IE userData: No
Hash of canvas fingerprint	6.63	99.0	f62f5e66c2081afc626390f1383d062b
Hash of WebGL fingerprint	3.63	12.42	undetermined
DNT Header Enabled?	0.79	1.73	True

Looks like the resolution and plugins list vary. Not much else however.

Check the language and platform tabs :) .

Ran one more test:



Right side is the sandboxed one. The rest of the specs are identical.

That’s it for today.

23rd of August

Given the degree of manual reverse engineering involved (even at big AV companies, virus analysis is still manual labour) in analyzing malware samples and the lack of any benefits pertained by behavioral based heuristic detection without signatures, I have decided to stop our R&D project here. I think we have touched the major points of interest required to understand and perform security analysis on any given malicious code sample.

The security industry is mainly a business of risk and reward, a world where aggressive capitalism and misleading claims meet with personal freedoms and oppressive governments. The era of information is mainly an era of noise and passive aggressiveness. Where should we draw the line between peaceful existence and fake news? Between an individual’s right of expressing his personal views without censorship (see the Honk Kong protests) and counteracting disinformation campaigns ran by foreign interest groups? Should we give up our digital self? Should we give up our fabricated “perfect life” image for something more substantial? Or are those just idealistic goals without any hopes of materialization?

Well, that’s up to you. All one can do is just take a small step towards the greater good. This will motivate and keep you humble at the same time. So, next time you are reverse engineering a piece of malware targeting a hospital, keep in mind that, some day in the near future, you might need that help too :) .

Take care & have fun.

Bibliography:

1. C++ Fundamentals - Antonio Mallia, Francesco Zoffoli
2. https://users.ece.utexas.edu/~adnan/pike.html
3. https://cs.unc.edu/~anderson/teach/comp790/papers/mars_pathfinder_long_version.html
4. https://preshing.com/
5. Hands-On High Performance Programming with Qt 5 - Marek Krajewski
6. https://queue.acm.org/detail.cfm?id=3220266
7. Optimizing software in C++ - Agner Fog
8. https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md
9. Hands-On System Programming with C++ - Dr. Rian Quinn
10. CMake Cookbook - Radovan Bast, Roberto Di Remigio
11. Qt 5 Projects - Marco Piccolino
12. C++ High Performance - Viktor Sehr, Björn Andrist
13. Boost C++ Application Development Cookbook - Second Edition - Antony Polukhin
14. Mastering C++ Multithreading - Maya Posch
15. C++17 STL Cookbook - Jacek Galowicz
16. Puppet 4.10 Beginner's Guide - Second Edition - John Arundel
17. Mastering Reverse Engineering - Reginald Wong
18. Python Digital Forensics - Daryl Bennett
19. Practical Linux Security Cookbook - Second Edition - Tajinder Kalsi
20. Learning Malware Analysis - Monnappa K A
21. Mastering Linux Security and Hardening - Donald A. Tevault
22. Digital Forensics with Kali Linux - Shiva V. N Parasram
23. Windows Malware Analysis Essentials - Victor Marak
24. Practical Windows Forensics - Ayman Shaaban, Konstantin Sapronov
25. Practical Digital Forensics - Richard Boddington
26. Hands-On System Programming with Linux - Kaiwan N Billimoria
27. CompTIA Linux+ Certification Guide - Philip Inshanally
28. Infectious Diseases of Humans, Dynamics and control – Roy M. Anderson, Robert M. May
29. Seasonality and the dynamics of infectious diseases - Sonia Altizer, Andrew Dobson, Parviez Hosseini, Peter Hudson, Mercedes Pascual and Pejman Rohani
30. Evolutionary analysis of the dynamics of viral infectious disease - Oliver G. Pybus and Andrew Rambaut
31. Coalescent theory – Vineet Bafna
32. Adaptive Dynamics of Infectious Diseases: In Pursuit of Virulence Management - Ulf Dieckmann, Johan A.J. Metz, Maurice W. Sabelis, and Karl Sigmund
33. Xu Y, Ren J (2016) Propagation Effect of a Virus Outbreak on a Network with Limited Anti-Virus Ability. PLoS ONE 11(10): e0164415. https://doi.org/10.1371/journal.pone.0164415
34. Stanoev A, Trpevski D, Kocarev L (2014) Modeling the Spread of Multiple Concurrent Contagions on Networks. PLoS ONE 9(6): e95669. https://doi.org/10.1371/journal.pone.0095669
35. Yang L-X, Draief M, Yang X (2015) The Impact of the Network Topology on the Viral Prevalence: A Node-Based Approach. PLoS ONE 10(7): e0134507. https://doi.org/10.1371/journal.pone.0134507
36. Yang L-X, Yang X (2015) A Novel Virus-Patch Dynamic Model. PLoS ONE 10(9): e0137858. https://doi.org/10.1371/journal.pone.0137858
37. Computer Virus Coevolution – Carey Nachenberg
38. CloudAV: N-Version Antivirus in the Network Cloud - Jon Oberheide, Evan Cooke, Farnam Jahanian

39. The Underground Economy of Fake Antivirus Software - Brett Stone-Gross, Ryan Abman, Richard A. Kemmerer, Christopher Kruegel,Douglas G. Steigerwald and Giovanni Vigna
40. GrAVity: A Massively Parallel Antivirus Engine - Giorgos Vasiliadis and Sotiris Ioannidis
41. Attacking Antivirus – Feng Xue (Nevis Labs)
42. Rethinking Antivirus: Executable Analysis in the Network Cloud - Jon Oberheide, Evan Cooke, Farnam Jahanian
43. AV-Meter: An Evaluation of Antivirus Scans and Labels - Aziz Mohaisen, Omar Alrawi
44. Characterizing Antivirus Workload Execution - Derek Uluski, Micha Moffie and David Kaeli
45. Gashi, I., Stankovic, V., Leita, C. and Thonnard, O. (2009). An Experimental Study of Diversity with Off-The-Shelf AntiVirus Engines. Paper presented at the Eighth IEEE International Symposium on Network Computing and Applications, 9 - 11 July 2009, Cambridge, MA, USA
46. Improving antivirus accuracy with hypervisor assisted analysis - Daniel Quist, Lorie Liebrock, Joshua Neil
47. The Antivirus Hacker's Handbook - Joxeaan Koret, Elias Bachhaalany
48. An Antivirus API for Android Malware Recognition - Rafael Fedler, Marcel Kulicke, Julian Schutte
49. Towards A Methodical Evaluation of Antivirus Scans and Labels - Aziz Mohaisen, Omar Alrawi, Matt Larson, and Danny McPherson
50. Formalizing and Verification of an Antivirus Protection Service using Model Checking - Adalat Safarkhanlou, Alireza Souri, Monire Norouzi, SeyedHassan Es.haghi Sardroud
51. AVLeak:Fingerprinting Antivirus Emulators Through Black-Box Testing - Jeremy Blackthorne, Alexei Bulazel, Andrew Fasano, Patrick Biernat, Bülent Yener
52. Powerslave: Analyzing the Energy Consumption of Mobile Antivirus Software - Iasonas Polakis, Michalis Diamantaris, Thanasis Petsas,Federico Maggi, and Sotiris Ioannidis
53. From the design of a generic metamorphic engine to a black-box classification of antivirus detection techniques - Jean-Marie Borello, Éric Filiol, Ludovic Mé
54. RAVE: Replicated AntiVirus Engine - Carlos Silva, Paulo Sousa, Paulo Veríssimo
55. ROPInjector: Using Return Oriented Programming for Polymorphism and Antivirus Evasion - Giorgos Poulios, Christoforos Ntantogian, Christos Xenakis
56. Software Vulnerability vs. Critical Infrastructure - a Case Study of Antivirus Software - Juhani Eronen, Kati Karjalainen, Rauli Puuperä, Erno Kuusela, Kimmo Halunen, Marko Laakso, Juha Röning
57. The Impact of the Antivirus on the Digital Evidence - Mohammed I. Al-Saleh
58. One packer to rule them all: Empirical identification, comparison and circumvention of current Antivirus detection techniques - Arne Swinnen, Alaeddine Mesbahi
59. Bypass Antivirus Dynamic Analysis - Emeric Nasi
60. Overview of Real-Time Antivirus Scanning Engines - L. Radvilavicius, L. Marozas and A. Cenys
61. Apoc@lypse: The end of antivirus - Rodrigo Ruiz, Rogério Winter, Kil Park, Fernando Amatte
62. Comparison of Firewall, Intrusion Prevention and Antivirus Technologies - Juan Pablo Pereira Technical Marketing Manager
63. Sophail: Applied attacks against Sophos Antivirus - Tavis Ormandy
64. Are current antivirus programs able to detect complex metamorphic malware? - Jean-Marie Borello1, ́Eric Filiol2, and Ludovic Me
65. Detecting unknown computer worm activity via support vector machines and active learning - Nir Nissim, Robert Moskovitch, Lior Rokach, Yuval Elovici
66. Crowdroid: Behavior-Based Malware Detection Systemfor Android - Iker Burguera and Urko Zurutuza, Simin Nadjm-Tehrani
67. Malware Analysis & Antivirus Signature Creation - Alan Martin Sweeney

68. A Model Checking Framework For Developing Scalable Antivirus Systems - E. Osaghae, S.C.Chiemeke PhD
69. Development Of A Heuristic Antivirus Scanner Based On The File's PE-Structure Analysis – S.Yu. Gavrylenko, M.S.Melnyk, V. V.Chelak
70. Testing antivirus engines to determine their effectiveness as a security layer - Jameel Haffejee, Barry Irwin
71. Malware Obfuscation Techniques: A Brief Survey - Ilsun You, Kangbin Yim
72. http://pramodkumbhar.com/2017/04/summary-of-profiling-tools/ (accessed on 21st of May)
73. Mastering Malware Analysis - Alexey Kleymenov, Amr Thabet
74. https://linuxgazette.net/133/saha.html
75. https://eklitzke.org/crcs-vs-hash-functions
76. Daniel J. Bernstein. ChaCha, a variant of Salsa20.http://cr.yp.to/chacha.html
77. Orr Dunkelman and Dmitry Khovratovich. Iterative differentials, symmetries, and mes-sage modification in BLAKE-256. InECRYPT2 Hash Workshop, 2011
78. Samuel Neves and Jean-Philippe Aumasson. BLAKE and 256-bit advanced vector ex-tensions. InThe Third SHA-3 Candidate Conference, March 2012
79. https://seanthegeek.net/257/install-yara-write-yara-rules/
80. https://en.wikipedia.org/wiki/Magic_number_(programming)#Magic_numbers_in_files
81. https://en.wikipedia.org/wiki/List_of_file_signatures
82. Practical Malware Analysis – Michael Sikorski, Andrew Honig
83. Evolution of Computer Virus Concealment and Anti-Virus Techniques: A Short Survey - Babak Bashari Rad, Maslin Masrom and Suhaimi Ibrahim
84. Review of Viruses and Antivirus Patterns - Muchelule Yusuf Wanjala & Neyole Misiko Jacob
85. DeepSign: Deep Learning for Automatic MalwareSignature Generation and Classification - Eli (Omid) David, Nathan S. Netanyahu
86. https://wiki.python.org/moin/Asking%20for%20Help/How%20can%20I%20run%20an%20untrusted%20Python%20script%20safely%20%28i.e.%20Sandbox%29
87. http://www.wiredyne.com/index.html
88. On Studying the Antivirus Behavior on Kernel Activities - Mohammed I. Al-Saleh and Hanan M. Hamdan
89. ANTIVIRUS SOFTWARE AND INDUSTRIAL CYBER SECURITY
SYSTEM CERTIFICATION IN RUSSIA - M.A. Nazarenko, A.I. Gorobets, D.V. Miskov, V.V. Muravyev, A.S. Novikov
90. Anti-forensic techniques deployed by custom developed malware in evading Anti-virus detection - Ivica Stipovic
91. Joint Detection of Malicious Domains and Infected Clients - Paul Prasse, Rene Knaebel, Lukas Machlica·Tomas Pevny ·Tobias Scheffer
92. Privacy and Security Risks of “Not-a-Virus” Bundled Adware:The Wajam Case - Xavier de Carné de CarnavaletMohammad Mannan
93. Machine Learning With Feature Selection Using Principal Component Analysis for Malware Detection: A Case Study - Jason Zhang
94. Learning to Evade Static PE Machine Learning Malware Modelsvia Reinforcement Learning - Hyrum S. Anderson, Anant Kharkar, Bobby Filar, David Evans, Phil Roth
95. The damage inflicted by a computer virus: A new estimation method - Jichao Bi, Lu-Xing Yang, Xiaofan Yang, , Yingbo Wu, Yuan Yan Tang
96. DETECT KERNEL-MODE ROOTKITS VIA REAL TIME LOGGING & CONTROLLING MEMORY ACCESS - Satoshi Tanda, Igor Korkin
97. Concolic Execution as a General Method of DeterminingLocal Malware Signatures - Aubrey Alston

98. Control Flow Change in Assembly as a Classifier in Malware Analysis - Andree Linke, Nhien-An Le-Khac
99. A Game Theoretic Model forNetwork Virus Protection - Iyed Khammassi, Rachid Elazouzi, Majed Haddad and Issam Mabrouki
100. Eradicating Computer Viruses on Networks - Jinyu Huang
101. Towards Metamorphic Virus Recognition Using Eigenviruses - Moustafa Saleh

Chelalau Ionut Mihai, R&D Engineer
c.ionutmihai@protonmail.ch