Extemporaneous formula

This method consists of obtaining and indexing data from the main processor or others. components of the hardware to generate an array of data, so that by segmenting it and applying differential method it can be indirectly known if we are exposed to a malicious process or device.

Since the maximum calculation speed achievable on our host is determined by the processor and its associated components, any event or process that interacts with it will end up significantly altering the correlation of data that our app generates during the different processes that run

The formula to fix this data is presented below, being this (base formula) of indexing and being able to add more parameters according to the level of security required. The array of data or segments derived from this process is our (key) to generate the encryption engine. Once this process is finished, this engine is only valid for a type of fractal or data arrangement. Even if we execute parallel processes and the data is the same, the result will be different. This in turn serves as a unique identifier of authenticity and data integrity.

For this example our parameters to consider are:

Vcore: processor voltage
clk: Speed clock (processor speed in Hz)
Rtc: system time

With the above we will conform our temporal reference (Trf).

Trf = [n1,n2,n3,n4] + [cl1,cl2,cl3,cl4] + [v1,v2,v3,v4]

where:

(n1 → n4) = the last 4 digits of our (Rtc) expressed in milliseconds.
(cl1 → cl4) = the last 4 digits of our (clk) expressed in Hertz.
(v1 → v4) = the last 4 digits of the processor's floating voltage expressed in millivolts.

The last 4 digits correspond to the most right, these being the maximum (rms) possible to acquire by our system.

We start:

Trf1: a = (cl4 + n4); b = (cl3 + n3); c = (cl2 + n2); d = (cl1 + n1)

Our formula (poisoned) is:

Example (1):

a1, b1 → (Ti)1 → (Ti)1, c1) → d1
c1 ←---------- (fx)
c0 ----------→ ↑

where:

a1 = (cl4); b1 = (n4); (Ti)1 = (a1 + b1); c0 = (v4); (fx) = (kpn → a1); c1 = (c0 + fx); d1 = (Ti)1 + c1)

Calc:

a1 = 7; b1 = 3; c0 = 5 [7,3 → (1) → (1,7) → (8)]

a1 = 6; b1 = 3; c0 = 5 [6,3 → (9) → (9,8) → (8)]

a1 = 7; b1 = 5; c0 = 5 [7,5 → (3) → (3,7) → (1)]

a1 = 7; b1 = 3; c0 = 5 [7,3 → (1) → (1,7) → (8)]

a1 = 7; b1 = 3; c0 = 8 [7,3 → (1) → (1,1) → (2)]

a1 = 7; b1 = 3; c0 = 3 [7,3 → (1) → (1,5) → (6)]

example (2):

a1, b1 → (Ti)1 → (Ti)1, c1) → d1
c1 ←---------- (fx)
c0 ----------→ ↑

where:

a1 = (cl4); b1 = (n4); (Ti)1 = (a1 + b1); c0 = (kpn → b1); (fx) = v4; d1 = (Ti)1 + c1)

Calc:

a1 = 7; b1 = 3; c0 = 6; v4 = 5 [7,3 → (1) → (1,2) → (3)]

a1 = 6; b1 = 3; c0 = 6; v4 = 5 [6,3 → (9) → (9,2) → (2)]

a1 = 7; b1 = 5; c0 = 4; v4 = 5 [7,5 → (3) → (3,9) → (3)]

a1 = 7; b1 = 3; c0 = 6; v4 = 8 [7,3 → (1) → (1,5) → (6)]

In example (1) a1 transfers its (Kpn) to (fx) to modify (c0) and obtain (c1)
for example (2) the (Kpn → b1) generates the parameter (c0) which is modified by (v4), that is, sample (fx). The (fx) is the factor we use to tell a value to behave like another.
We call this poison the equation and it is very useful for generating dynamic encryption.
This is because, for the purposes of an auditor, the body of the program is observed in the correct way but we generate an entry point [EP] in which our program once our conditional is fulfilled takes a certain value and calculates it using our (fx) in this way a parallel program or data is generated within its body. This is not visible unless you have the parameters to verify the [PE]. Trying to find this value by brute force is almost impossible, since as we show in the previous examples for different factors we obtain the same product. Then our equation is easy to verify in the forward direction (calculation) but is ambiguous in reverse, unless the starting values are possessed.

This equation differs from those of (compression) characterization of data since in this case the opposite is required. In other words, to characterize data our possibility of collision and or ambiguity must be (0) otherwise we would obtain data without coherence. The details of this last case are omitted for copyright and commercial use reasons.

So:

the result of these 4 logical operations make up our wachtdog [WD]

(Trf4, Trf3, Trf2, Trf1) = WD1

the [DIFF] →] between [WD] 1] + [WD] 2] must always be> than [WD] 3] and this cycle is repeated by shifting the value to be calculated using the last 3 [WD] always.

The foregoing generates a plot that allows our program to predict or confirm any abnormal operation in relation to debugging the process or snorting for reverse engineering or malicious purposes with our app.

Since the above parameters are taken on the motherboard at a low level, any interaction with our app generates a correlation break in our plot, but allows the system to operate normally with others (Pid’s). Then the more effort (interaction or calculation) make an application to our program. Faster our condition is met (halt) [HLT]
leading to the implementation of countermeasures (several).

In a value confirmation equation we have:

a,b → (Ti)1 → (Ti)1,c) = d
  c

Example:

we justify (confirm) value (b)

our factors would be (b, c); a = (Kpn → c); d = our result.

b = 3; c = 7 [2,3 → (5) →(5,7) = (3)

b = 8; c = 7 [2,8 → (1) →(1,7) = (8)

b = 2; c = 8 [1,2 → (3) →(3,8) = (2)

b = 9; c = 1 [8,9 → (8) →(8,1) = (9)

we justify (confirm) value (c)

our factors would be (b, c); a = (Kpn → b); d = our result.

b = 3; c = 7 [6,3 → (9) →(9,7) = (7)

b = 8; c = 7 [1,8 → (9) →(9,7) = (7)

b = 2; c = 8 [7,2 → (9) →(9,8) = (8)

b = 9; c = 1 [9,9→ (9) →(9,1) = (1)

so:

We can affirm that the process of poisoning an equation consists in modifying the relationship between the expected value for (b) in position (a) and the Kpn of (b), (c) as the case may be.

There are several adulterated (poisoned) models, this being the basic model of two components or primary factors, but as many as required to form complex arrays can be indexed. As well as cross (exchange) values between them in a dynamic way making our final data more robust in relation to reversion.

Julio C. Gómez
io-exchange.com (c)