An approach to behavioral scanning malware using a sequential classification intelligent algorithm

Bodea Nicolae

Group 932

Babes Bolyai University,

Faculty of Mathematics and Informatics

Abstract

Day by day, malware becomes more intelligent and acquires more attack vectors than anytime. The reason of this happening is the more involvement of companies into the cyber-security and counter-cyber-security world, but also a better understanding of the cyber-security techniques by programmers. It is clear that nowadays, old signature-based security techniques won’t deal with the dangers from the online world, and more sophisticated techniques must be researched and developed by security teams in order to counter-attack those threats.

This paper proposes a solution to use an intelligent algorithm which classifies sequences in order to find malicious system calls made by protected applications. The solution proposed aims to scan and classify the potential malicious sequences based on their behavior. The classification involves Long-Short Term Memory networks (LSTM) based on sigmoid activation and having the Poisson loss function. This paper aims to give a more detailed explanation and implementation details of this approach.

Motivation

This solution aims to catch malware based on the system calls that a program does. Security problems tend to appear day by day, leveraging newer and more intelligent techniques. But, malware running on an Operating System, such as Windows, must leverage some Operating System calls in order to execute malicious code.

For example, any program running on a computer, needs libraries and need to allocate memory on the heap. This solution offers a way to load a library inside every protected process before any code from that program, and intercept all system calls from that the possibly malicious code does.

The neural network will then classify the sequence given by the library and if there is a problem, will instruct the library to kill the process. This is the best approach, because once the process has executed a malicious piece of code and has called a malicious sequence of system calls, there is not known how much the malware advanced, and how close it is to subjugate and compromise the whole system.

The purpose of this solution is to enhance old cyber-security techniques, by using behavioral scan, combined with neural networks in a very precise piece of anti-malware software.

Related work

1. Andromaly

Andromaly is an Android-based framework for behavioral malware detection. The framework is based on analyzing the events from the Android system, such as movement events, opening applications and network changes. This system is monitoring the default applications on the device, and, making use of a machine learning anomaly detection algorithm, detects unusual events that could appear only in malicious conditions. Their

1. DroidMat

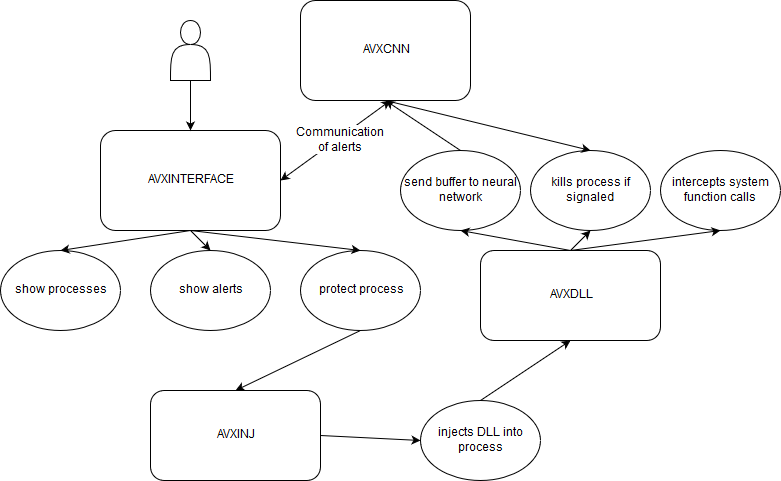
DroidMat is another Android-based protection software, which makes use of API calls tracing and the manifest of each application. Basically, the software will get the API calls of every application and, depending on its’ behavior, such as reading some sensitive information from the system, or requiring some permissions which the app shouldn’t will raise the attention of DroidMat. For detecting malicious behavior in the sequence of API calls of an application, this software uses Singular Value Decomposition for low rank approximation, and then classifying each application as malicious or benign using a kNN algorithm. The algorithm is based on training on 4000 malicious and non-malicious applications.

1. Behavioral Clustering of HTTP-Based Malware and Signature Generation Using Malicious Network Traces

This solution clusters HTTP requests based on Network API calls present on the operating system. The clustering algorithm was trained with 25000 network trace samples, by detecting the behavior of the network trace, interpreting and analyzing the HTTP requests sent or received on the machine. The results of this software are clusters of requests which are then classified as malicious or not malicious.

Components of the solution

This section contains the detailed explanation of which components are present in the implementation of this software, and everything of which each component is capable of. It also comprises implementation details about each of the components and how they work together and what are their inter-dependencies. For this purpose, this presentation contains an integration diagram:



1. AVXDLL

This component is the one that is responsible for System API call tracing. It is written in C++ and Assembly code. The assembly code is used for saving the context registers of the original functions when detouring the system functions’ code. For disassembling functions, this library is linked with another open-source library, named ZyDis. The disassembling functionality is needed for hooking, which will be discussed in the following paragraph. The library gets injected into a target protected process on demand, by calling the AVXINJ component.

The library has the following functionalities:

1. Detour the system call functions

This means that this library will overwrite the system functions in the following way: Firstly, will get the first instructions from the system function and save them. Then we overwrite the first instructions with a jump to our trampoline code. From our code we save the context of the program (e.g. registers, stack, etc.) and then will call the library to notify a new system API call. The system API calls are kept internally in the library as ordinals (each function has a unique number, starting with 0, from which the library identifies which call was done). This ordinal is pushed on the stack before jumping to our trampoline code.

1. Gather information about system calls

After a call is made, the library must acknowledge the call. As discussed before, this is done by ordinals. The library will find the function in an internally-kept hook list, which contains all the detours which have been made with success. After the hook is made, the library knows exactly which function was called, and has the original code. The library will call a callback, depending on what it is needed to do with this function call. For this purpose, every function call will get into a callback which will add the function to a buffer of called functions, which is kept into a dynamically allocated vector.

1. Consult AVXCNN component on a sequence of calls

The library will consult the neural network when the number of calls in the buffer is great enough (e.g. 40 calls). If the neural network will tell that it is a malicious component, the library will instantly call ExitProcess system call, which will kill the process, thus not letting the malicious code execute anymore.

1. AVXCNN
2. Deciding on the method used for learning

The architecture of the neural network has been decided experimentally, based on how the layers best fit the data analyzed by the network. It comprises three layers of neurons in a sequential layer model.

Firstly, in order to achieve a model for sequential classification, the Sequential model from Keras was used. The sequential model uses a linear stack of layers for learning, having the first layer of the model receiving the input shape. For this purpose, an Embedded input layer was used, which turns positive integers into dense vectors of fixed size. The embedded layer expects us to give the input dimension, the output dimension and optionally, the maximum input length. For the neural network to behave well, the input training data and input testing data must be padded with zeros, in order to achieve a uniform input length. For this neural network, the input length was fixed to 500, as it is the maximum length of any training data/test data array entry. The input dimension of the embedded layer must be the maximum value plus one of the data on which the neural network trains.

On the second layer, the model contains a Long-Short Term Memory layer, which is very good in practice on sequential classification. The LSTM layer must be given a parameter, the number of output nodes to generate. LSTM networks are a special kind of Recurrent Neural Networks, capable of learning based on long-term dependencies between features. The advantage of these kind of networks is the fact that they keep a long-term memory, as this is their default behavior, and these networks achieve very easily this feature. LSTM consists of a chained structure, much like other RNN-based networks are, but their structure is different, because instead of having one single layer based on a *tanh* function, there are four sub-layers which interact in a very interesting way, as presented in the following diagram:



The Long-Short Term Memory layer is a very good technique for sequential classification, in particular, for this solution, a long-term memory of known malware is an advantage. This is due to the fact that a malicious behavior will most likely be malicious anytime, even if there is a chance of false positives, it is always better to except false positive in the cyber-security environment than to have false negative.

Finally, on the third layer, a Dense model was used. This was done due to the fact that this is a binary classification, and the Dense layer model is the best-suited layer type for this kind of classification, along with a sigmoidal activation. This layer will have a single output, which consists of the answer of the classification, mainly the output of the third layer will be the answer to the question “what is the probability of the current sample to be malicious?”.

The model was compiled with a Poisson loss function, of the form:

C:\Users\nbodea\AppData\Local\Microsoft\Windows\INetCache\Content.Word\psm_nonlinear-2.png

The first loss function that was used was a binary cross-entropy function, as this is the most logic and most used loss function for binary classification. But, experimentally, Poisson proven to be a better loss function for this kind of problem, mainly because it is better on the detection rate. This is due the fact that, even if the loss becomes lower slowly, the network learns better the current sample. That means that, if the answer was that the current sequence is malicious, the next time the network encounters a sample which is approximatively the same, it will know that it is malicious.

The optimizer used for this model was an Adam optimizer, the name being derived from adaptive moment estimation. By using an optimizer, running the training of neural network can make the difference between days of training and hours. Adam optimization algorithm is used instead of the classical stochastic gradient descent in order to update the weights, by using different learning rates based on the neurons. This fastens the process of learning very much, as some neurons can tend to the perfect weights for the current problem very fast, instead of just iterating again and again through all nodes and updating the weights. Adam optimizer adapts to the current training set and will make some nodes train faster than usually, which greatly improves overall performance.

1. Training and testing data

Training and testing was done on the CSDMC\_API data set. The training data set consists of 387 sequences of system calls having different length, ranging from 600 calls per sequence, to 18672 calls. Test data has 376 sequences, having the same length range as the training set.

The training set consists of about 80% malicious sequences and 20% non-malicious sequences, and the testing set is 100% non-malicious sequences. Due to these inequalities between the sets, the data set must be normalized prior to start training. For this purpose, a five-fold cross-validation strategy was taken into consideration. Firstly, all of the data was put in a single array, then, for a chunk of sequences, we must select a chunk of 5 sequences having 1 malicious (positive answer) and 4 non-malicious (negative answers). This happens with a chance of 50% for every chunk, in order to keep the testing set having a lower size than the training set.

The second problem was the wide range of the sequences’ length. For this purpose, every sequence with length greater than 500 was divided in smaller sequences of length 500 and added to the data set, being annotated with the same value as the bigger sequence. This also augments the data set and normalizes the lengths of the sequences, improving the overall accuracy of the testing set. Having the length of a pre-defined fixed size will help when padding the sequences, and also, this is a real-world scenario, as the AVXDLL component will only send sequences of fixed size.

After training, the neural network will save the serialized model into a JSON file, and the weights will be serialized in a file with the H5 format, which is known to be a good format for this purpose.

When testing the data, the neural network reloads the file from the disk, then recompiles the model and predicts the testing data, verifying if the prediction is the same as expected.

1. Communicating with other components

The neural network communicates with AVXDLL through a socket. It receives the length of a sequence and then the sequence elements. The neural network will respond through the same socket with “0” if non-malicious and “1” if malicious.

If the neural network responds with 1, then it will also send the process identifier to the interface, in order to correlate the alert events.

1. AVXINJ

AVXINJ component is responsible with deploying the AVXDLL component library to the protected processes. It will make use of Windows System API Calls in order to inject the path to the DLL to the process and to create a remote thread in the targeted process, which calls LoadLibrary on the given path. AVXDLL will then load and the process will be protected.

This component is called with the command-line avxinj.exe <process id of the targeted process>.

1. AVXINTERFACE

The interface consists in a Windows Form view written in C#. The form view has two DataGridViews, which show to the user two tables, the process table and the alert table. The process table will be initialized with the started processes on the system when the interface is running, having the following information on each process: process name, process identifier and a button to protect the process. The alerts table will contain the process identifiers for the processes which triggered an alert and were killed by the neural network. One can reload at any time the user process list by clicking the reload button under the process table. The process table gives the possibility to the user to sort by name or process identifier the processes which are currently running on the system.

Avxinterface is the entry point for using the system and does some initializations prior to letting the user do any actions. It starts the neural network and makes a connection to it through a socket on port 50055. This connection will be used for sending malicious process identifiers when an alert is triggered. The interface will then initialize its components, mainly the process table.

When a user is clicking the “Protect” button for a process, the AVXINJ component is called, which loads the library as described.

How to install

System requirements: Windows 10, .NET framework at least 2010, Visual Studio Redistributable at least 2010, Python >= 3. Python must be in system PATH.

Results in testing and real-world scenarios

For measuring accuracy of the neural network, an approach of using precision and recall method based on the confusion matrix method was used. Precision was measured by computing the number of “guesses” of the neural network divided by the total number of testing samples. But this method will not give a good approximation of the real accuracy, because there are non-relevant items taken into account. The recall method is measured by computing the number of true positives guessed divided by the number of true positives + false negatives in the data set. The recall method is often named “detection rate” in cyber-security world and it is a standard for measuring anti-malware software. This is one of the most accurate method for this kind of purpose, because the false-negatives are the most relevant when taking into account detection rate of an anti-malware system.

*Accuracy results*

|  |  |  |
| --- | --- | --- |
|  | Precision accuracy | Recall accurracy |
| Training set | 92.5% | 89.5% |
| Testing set | 88.2% | 74.5% |

*False positives/negatives statistics*

|  |  |  |
| --- | --- | --- |
|  | False positives | False negatives |
| Training set | 121 | 233 |
| Testing set | 2 | 14 |

For a real world scenario, the anti-malware system was used to monitor different applications, such as Mozilla Firefox, Google Chrome, Notepad and Windows Calculator. Mozilla Firefox, because of the many detours that puts itself on the system DLLs, is almost impossible to protect, creating a very high performance impact, and also creating a very random rate of false positives and false negatives. This is due to the fact that the AVXDLL library could not extract enough information for the neural network to guess correctly. But, for the other applications, the anti-malware system presented in this paper works as expected, having a relatively low performance impact, the applications still being usable when the system is monitoring them. However, some applications created some false positives inadvertently, such as Chrome, generating as much as 4-5 false positives per hour. For a complex anti-malware system, these can easily be put into an exception from detection, and there can be a backpropagation error for the detection in this case.

For the future work, this project aims to monitor and keep performance at good standards for as much as ten application in a row, and one of the aims should be 0 false negatives and 0 false positives for the supported applications.

Conclusion

Even if the neural network used for this project is having some good results, there are some drawbacks on this solution. For example, a normal user should be able to create an exception for alerts which he knows they are false positives, or should create a policy for the system, to act on certain applications with “kill process” or with “log only”. Also, a normal user would want 0 false positive as much as possible, as one would not want to quit his browser immediately if there is a particular case that the neural network has not been prepared for.

One more drawback is that even if the system would have 100% detection rate and 0 false positive, an intelligent malware could easily take down the protection from a process, by unhooking the functions and thus, stop monitoring.

But even with this drawbacks, there are more and more malware which are polymorphic, and interchange some instructions which would have the same result as the original code, and infiltrate a system which is not updated. In most of the cases, normal signature-based anti-malware software will not catch these kind of viruses, because they know only about limited versions, and changing an instruction will change the signature. But this software, because of the fact it is monitoring behavior of the processes, rather than verifying the exact code in some part of the process, will successfully block these kinds of attacks.

Bibliography

* Brownlee, Jason. "Machine learning mastery." *URL: http://machinelearningmastery. com/discover-feature-engineering-howtoengineer-features-and-how-to-getgood-at-it* (2014).
* Müller, K. R., Krauledat, M., Dornhege, G., Curio, G., & Blankertz, B. (2004). Machine learning techniques for brain-computer interfaces.
* Wu, Dong-Jie, et al. "Droidmat: Android malware detection through manifest and api calls tracing." *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on*. IEEE, 2012.
* Firdausi, Ivan, Alva Erwin, and Anto Satriyo Nugroho. "Analysis of machine learning techniques used in behavior-based malware detection." *Advances in Computing, Control and Telecommunication Technologies (ACT), 2010 Second International Conference on*. IEEE, 2010.
* Gers, Felix A., Jürgen Schmidhuber, and Fred Cummins. "Learning to forget: Continual prediction with LSTM." (1999): 850-855.
* Shabtai, Asaf, et al. "“Andromaly”: a behavioral malware detection framework for android devices." *Journal of Intelligent Information Systems* 38.1 (2012): 161-190.