Using Machines to Exploit Machines

Harnessing AI to Accelerate Exploitation

Guy Barnhart-Magen (@barnhartguy), Ezra Caltum (@aCaltum)

April 2019

Introduction

Vulnerability research objective is often to find a method to manipulate the software to behave in a way controlled by them. The most famous example of this is to cause a seemingly benign piece of software to execute our code

A very promising way to find exploitable bugs is through fuzzing the software. By fuzzing, we mean the process of inputting random data into the software through the user accessible interfaces (API, command line etc.) in order to cause unexpected behaviour or crashes.

However, employing large scale fuzzing framework sometimes has the unpleasant side effect of being too successful, producing thousands of possible candidates for exploitation.

Given a specific crash/candidate the vulnerability researcher will conduct an analysis, sometimes following the code itself through reverse engineering, to determine if the specific crash answer two important imperatives:

1. Is it reachable from user controlled input?
2. Can it lead to an exploitation path that will achieve our required target?

Answering these questions is time and resource consuming, and very difficult to determine accurately. In essence, it is very easy to show that something is exploitable when it is (e.g. build a PoC) but the inverse isn't true (showing that something in not exploitable). There might be some other way that might lead to exploitation - and this is where the researcher's experience, creativity and determination comes into play.

In our previous work[2] we showed how ML can be subverted either through direct manipulation, or indirectly through exploiting vulnerabilities. This experience pushed to think how can we use ML in order to simplify our work.

Background

In this research we set out to find out if the process of finding suitable candidates for our research team to analyse could be accelerated through the use of tools from the domain of Machine Learning.

We wanted to build a tool (based on a machine learning model) that can outperform the best tool we could find, *exploitable*[1].

Exploitable is a tool developed to analyze crash dumps through a set of heuristics, and indicate whether it is exploitable or not. It provides three outputs once analysis is complete:

- Exploitable - this sample has a known vulnerability and it is exploitable
- Probably Exploitable - there are some indicators showing that this sample is exploitable, but it is not trivial
- Unknown - it is not clear if it is possible to exploit the sample or not

Objectives

Our objectives were to outperform *exploitable* when running on the same data set. In order to achieve this goal we broke it down into several tasks, the most important one was - can we identify exploitable samples in places where *exploitable* isn't sure or doesn't know that they are exploitable?

Data Set

In order to train our model we needed to have a sufficient number of samples that are both known to be exploitable, as well as those known to not be exploitable. We spent considerable time trying to obtain such data sets without much success.

Finally, we stumbled upon the data set provided by DARPA[3] during the Grand Cyber Challenge. In this data set, they collected a large set (634 vulnerability samples) where each one was known to be exploitable in some way. We couldn't find a proper set of similarly non-exploitable pieces of software as a control group and decided to proceed with ML tools that focus on a single class domain.

Using Machines to Exploit Machines

Harnessing AI to Accelerate Exploitation

Guy Barnhart-Magen (@barnhartguy), Ezra Caltum (@aCaltum)

April 2019

Gathering Data

Our vulnerable samples had different dependencies, and required different inputs. We strongly believe in reproducibility, and in having an easy to replicate testing environment. To be able to do so, we set up a Docker environment where we could run the vulnerable software and collect the appropriate generated artifacts.
That way, we can ensure reproducibility of these experiments and provide a common environment that researchers can use to replicate our conclusions.
We built a pipeline that achieved the following:

1. **Run The Program** - Run the vulnerable sample in a contained environment
2. **Gather Crash Data** - Allow the guest OS to capture a large amount of crash data (Core dump)
3. **Crash Analysis** - Analyze these artifacts, using *exploitable* and keeping its results
4. **Feature Extraction** - Extracting our desired features from the *exploitable* output

The results of this piplie is later fed to the ML model as their dataset.

Crash Analysis

In the traditional approach to crash analysis, we take a look at the program flow, at the output from *exploitable*, and at the information from frames, registers, threads and stack information.
We collected these core dumps, and used GDB and a couple of scripts to extract the data in a binary format, into a textual representation, thus allowing the model to replicate our traditional manual work.

```
g() {
        gdb  --batch -ex bt -ex "info all-registers"
-ex "info frame" -ex "info functions" -ex "info
registers" -ex "info stack" -ex "info threads" -ex
"exploitable" -c $1 > $1.res
}
```

Feature Extraction

Once we had the crash dumps from each sample, we wrote a python script to extract the features we deemed important.

- EAX, EBX, ECX, EDX - general purpose registers (holding values, addresses)
- ESP, EBP - Stack pointers
- ESI, EDI - Source and Destination Index (for string operations)
- EIP - Instruction pointer
- CS, SS, DS, ES, FS, GS - Segment registers

In order to simplify our model ingestion of the different values (and to increase model accuracy), we tried two approaches. The first was a rough categorization through semantics, and the other through rough binning.

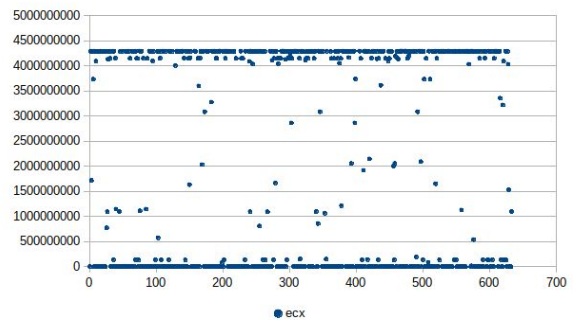

Figure 1 - Values distribution of the ECX register for our sample

Semantic

The value distribution for the registers can be roughly divided into three semantic groups:

- Low values (<1,000,000) - usually computation results or temporary values
- Mid range - usually user land addresses
- High range - (>0x1FFFFFFF) usually kernel addresses

Binning

A different approach we tried was to create 10 evenly distributed "bins" ($\frac{value_{high} - value_{low}}{10}$) or groups and fit our samples into these bins.

Both of these approaches simplify the data ingestion to the model. However, the semantic approach was too rough and did not yield good results. We hypothesized that if we did a finer grained approach we could have gotten better results, but did not attempt this.
Our best results were received through the use of binning. A possibly better approach could have been to use a binning strategy that had finer resolution at the low and high ends and lower resolution in the middle ranges, to better conform with our values distribution.

Building the Models

We built three different models, using different approaches. One of our constraints was finding models that fit our use case - a single class (as we know every sample is exploitable) and train against that data.

We built our model based on the 609 samples correctly identified by *exploitable* as exploitable and ran the other 25 samples (which *exploitable* identified as either probably exploitable or unknown) against this model to measure its accuracy. As we know these 25 samples are also exploitable, our aim was to identify as many of these as possible, thus out performing *exploitable*.
table

One Class SVM

Our first approach was to use one class support vector machines, as they are very suitable for our use case. They train on a single group or class of data, and allow us to predict if a new sample is similar to the previous samples that it trained on - which the model is built on.

Cosine Similarity

Our second approach was to use a cosine similarity metric to determine the distance (or the inverse, similarity) of each of the 25 mis-labeled samples, to the rest of the correctly labeled samples.
For distance measurements we used two different approaches, linear distance and centroid distance, comparing their results to achieve best performance.

XGBoost

We used XGBoost primarily to look into how the model is interpreting the input parameters it is trained on. The structure of the model is similar to a tree, where the root of the tree has the most contributing parameter, and each level after that represent a hierarchy of parameter contribution.
Analysing the tree gives us some insight to which parameters are the most important, and what are some of the relationships between them.

Results

Goals

Our goal was to use the data we have (634 samples) which we know are exploitable to train our model and test the cases where *exploitable* was not able to correctly identify the sample as exploitable (25 samples).

Simply speaking, we tried to categorize the 12 "Probably Exploitable" and 13 "Unknown" as "Exploitable".

OneClassSVM

Our trained model was able to correctly identify most of the samples, correctly identifying 23 samples as "Exploitable" and identifying 2 samples as "Probably Exploitable".

We were also able to show that two of the samples are outliers, meaning that they look very different than the samples the model trained on - but still identified as "Probably Exploitable".

Cosine Similarity

We first tried using the basic 9 registers and got a relatively low accuracy of ~65%, which we found too low. We were not happy with the results, so we

increases our data to 15 registers (using binning), which boosted our accuracy to ~87%.
Our model was able to correctly identify 15 more samples when using Linear Similarity, and 22 samples when using Cosine Similarity, which is the better method for our data distribution.

XGBoost

This model training is more traditional, and we split our data to 80% training (from each category, randomly selected) and 20% validation.
We were able to achieve 95%-99% accuracy. Please note, that this result is to be expected as ~96% of our dataset fits in the “Exploitable” category, and a simple “return Exploitable” classifier will be correct 96% of the time.

Our analysis of the tree we received was interesting, it seems that it is sufficient to use only two parameters in order to correctly identify 90% of our samples. While this result is interesting, we doubt its real life value, as it is highly dependant on our data set.

Comparison

In the table below we compare our results from different types of analysis models, showing that using ML has clear benefits, when used on the same data set as *exploitable*.

Type	EXP	PXP	UNK
Original	634		
exploitable	609	12	13
OneClassSVM	632	2	
Cosine Similarity	631	3	
XGBoost	632		3

Table 1 - results for different types of analysis (EXP - Exploitable, PXP - Probably Exploitable, UNK - Unknown)

Next Steps

Data

One of our biggest issues with the research project as it is, is the amount of data and the type of data that we have. Getting a reliable data set for study proved extremely difficult, however we believe that more data will enable much better results, and more robust systems.

ML

There are other ML models and techniques we could employ with the data we have, and we invite the community to improve on our work. The results we have show that such techniques can really improve on the best methods available today.

Syzkaller

Analyzing kernel crashes is more complex than analyzing user land crashes. Further - due to the popularity of syzkaller, there are multiple crash reports and test samples available at multiple mailing list. We believe that we can try a similar approach against that data set, and we will be focusing our efforts on that task.

ASAN

We want to try using AdressSANitizer[4] (ASAN) for additional data points. We believe that we could greatly improve the accuracy by using data points available in the ASAN output.

Conclusions

In conclusions, we showed that when using the same data set, we can use various ML techniques to outperform *exploitable*.

However, there are many issues that delay this project from being widely applicable.
Data Variance - as we only have data on (verifiably) exploitable samples, we are lacking a control group to validate our models.

Data Bias - as most of the data we trained on has very specific vulnerability types, the ML model is biased to recognize these types.
Data Set Size - our sample size is relatively small (~630 samples) and care should be taken when relying on these results.

Acknowledgments

We would like to acknowledge Denis Klimov who helped us with the ML part of the research.

References

[1] exploitable - https://github.com/jfoote/exploitable
[2] JARVIS never saw it coming - https://www.youtube.com/watch?v=d99QshMaGtQ
[3] Grand Cyber Challenge - https://github.com/jfoote/exploitable
[4] Address Sanitizer (ASAN) - https://en.wikipedia.org/wiki/AddressSanitizer