Introduction

The task to bypass machine learning models I took as a personal challenge, which means I decided not to use any resources, tools, sandboxes or manpower from my employer.
I know that my free time will be very limited due to personal and professional responsibilities and I have just a few evenings to spend on the challenge. Therefore my workflow had to be time optimised.
First I checked what the challenge was about, because whenever there is a detector, the ratio of true-positives to false-positives is important. In the challenge there were 3 models. The “ember” model is quite decent (FP rate below 5% on my %ProgramFiles%) and while it’s too high for real security software, it’s acceptable for academic purposes. However the two other “malconv” models have FP rate over 70%, which makes them totally impractical.

Constrains

The competition had two main constraints - not using SFXes (self-extractors) and not using Droppers (“downloaders”).
These two constraints were important for the competition purposes, because static machine learning analysis is defenceless against them. Static analysis is done at one point in time, which means that if malicious features cannot be extracted at this time, the object will be considered clean. Therefore if Downloader is considered clean on execution time, it can freely download malicious code and execute it.
Reason to forbid “self-extractors” may not be that simple to understand at first. The problem is that “self-extractors” are very popular among clean applications (any installer), which means that training a machine learning detector you cannot train it to detect the “self-extractor itself”. It would cause too many false-positives. Self-extractors contain compressed and/or encrypted data, and from such data machine learning classifiers will extract nothing but noise.

Another constraint, that made this competition more academic than “real-world scenario”, was with the malware samples. As an attacker I would have source code of malware so I could modify it easily, but here we had to work with compiled executables, where some samples were already modified/packed by different tools. Additionally, they were executables, not DLLs, which made dynamic loading harder.

Approach to the challenge

Approach 1

The challenge was to bypass all 3 machine learning models at the same time using modified malware samples. You could do it sample-after-sample but - of course - it would be best to find a way to bypass all the samples with a single technique.

The first idea that came to my mind was finding a hole in the feature extractor. If I could force it to produce wrong results or crash, the machine learning algorithm would have no data to classify. The PE file parser used in the competition is LIEF. I had no environment set of some serious fuzzer like AFL, therefore I wrote a dummy fuzzer that was putting random bytes at random locations. I executed the fuzzer on LIEF and waited a little. It did not find any issues but waiting for the result I realised two things. The first one was, that I'm thinking too much as an attacker here, not a competition contestant. If I crash the feature extractor, the classifier won't say "the file is clean", but it will also not say "the file is malware". Therefore, depending on how the environment is prepared, it could give me zero points. Second, it would not be fair towards competition organisers because I would not try to bypass machine learning models, but I would find a hole in the competition environment. It could also influence the environment for other participants and I did not want to do this - wouldn't be really different than DDoSing the servers to block access for the others. I stopped therefore quickly and started to think about how to attack the models.

Approach 2

The first idea was to use Process Hollowing technique and there are open source tools available for it. I could create a new "good looking" executable, encrypt the malware sample and hide it in the resources, then I would extract it at run time. However here I also stopped. The competition rules clearly forbid to use self-extractors and here I would not modify the malware samples, but I would drop it just like self-extractors do. Is it really a difference if it's dropped to disk and executed or executed directly from memory? I didn't want to be disqualified for this.

Preparation of samples

Modifying the malware samples, checking if they are not detected and then running them on a sandbox to verify if the behaviour is the same is very time consuming. I needed a better approach and I found one. Why to use malware samples at all, when I can work with clean samples? I needed to find a way to trigger FP on a clean sample and from such a state find a way to bypass machine learning classifiers. This way I could very quickly and safely check if the behaviour of the sample is still correct.

First step was to modify the classifier to return the classification score. This allowed me to identify if I'm progressing with my attempts.

I packed the samples from my %ProgramFiles% with UPX, PECompact and ASPack and found out that ASPack and PECompact are detected as malicious, but UPX is not always detected. It means that this packer is very close to the classification border and small modification of the samples could move me to the required side. UPX is a very common packer, therefore just like in the case of "self-extractors", just detecting it would cause too many false-positives for practical use.
I found one application particularly useful "7za.exe" (command line version of 7zip), when UPXed was triggering FP on all 3 models.

Some samples from the “malware set” (MLSEC_samples) were compressed and UPX has internal checks not to make output bigger. I grabbed the UPX source code and compiled it with all the checks removed. Controlling UPX code was potentially useful for the future, because UPX keeps some imports of the original PE file (imported API is always a strong feature of malware classifiers), and they could be hidden by a dynamic loader.

Approach 3

Using the previously prepared fuzzer I wrote a script that was fuzzing the header of the UPXed 7za.exe, checking if LIEF can parse it properly, checking if modification leads to better classification score and if yes, executing it. The problem however was that a randomly modified application was crashing. Here I used a not well known trick to handle such a crashed app and then I was able to easily identify samples that were for sure running properly after modification.
This fuzzing very quickly revealed one very strong feature in “ember” classifier:
NT Header -> Optional Header -> Data Directories -> DIRECTORY_ENTRY_SECURITY
the presence of just this entry in 90% of cases was influencing the classifier to tell that sample is not malicious anymore.

Approach 4

Considering that I had a very strong weapon against the “ember”, it was time to focus on “malconv” models. As the models are extremely aggressive, I was thinking what approach should I take: feature importance analysis, better fuzzing of not only header but also imports to find out what impacts classification the most? Transfer of API imports from a clean executable to the malicious ones?
I peeked at the code of the classifier and... wait. Are they using a neural net and in a dummy way feed the whole executable to “convolutional sliding window”? It was clearly the case, so it was worth trying a very simple solution. To explain the approach, imagine that we are classifying objects in a picture. We are trying to find if in the picture we have a malicious wolf or a cute sheep. Let's say that there is something that resembles a wolf and the neural net will return “True”. However we can expand the picture. We can add any number of very clear images of the sheep to it. What will the classifier say now? Almost for sure it will say that the picture is more “sheepy” than “wolfy”.
I classified the whole %ProgramFiles% set and found a few samples that had very good scores of being clean by both Malconv models. I attached these samples to the end of malicious samples and for the classifier they got the “clean” statuts.

Testing, scoring and technical issues

I submitted the samples.and waited for the output. Out of all the samples only 1 passed the IOC check and I got 3 points of 150. Very discouraging. Clearly modifications I did caused the malware samples to stop executing, which was surprising, because UPX is very well tested, the overlay added to PE does not influence execution and my header modification was harmless. Seems that the hard part in competition is not to bypass the models, but to assure exact behaviour in IOC matching after modification. The next evening I spent

installing and setting up a VM of x64 Win10, running a few malware samples and checking them with Process Monitor, reverting images, writing some tools to generate behavioural trace, scripting to extract the trace from VM file system for automated processing and autoreverting the VM.
I submitted slightly modified samples that should improve a few things but all submissions were stuck in processing. The next submission was also stuck. Administrator admitted that the competition environment has issues and they will be fixed in the next few days. Unfortunately, due to lack of personal time I wasn't able to return to the competition after a few days, when the problems were fixed. Organizers admitted technical issues with processing and my submitted samples were gaining more and more points, ending with a score of 141/150. Meanwhile other contestants gained 150/150 and the competition was won.
On August 26 organisers admitted another issue with scoring and the competition was set as ongoing again and about a dozen of my previously submitted samples did not get accepted.

Approach 5

After returning from work I looked at the samples that did not get previously accepted. When checking the samples I realised that a big contributing factor to classification score is entropy. Appending multiple times to files for which the score was “clean” in all 3 models I was able to quickly get to 150 points again. Almost the same time other contestants got 150 so I went to sleep. The next morning I checked the results and, well, again there was a scoring reset, because of issues with IOC processing. This time I wasn’t able to get back to the samples to fix them, because of professional responsibilities and then planned holidays (without access to a PC with the samples and scripts). Knowing that there will be scoring reset again I would resubmit all the samples with a newly discovered method (appending clean files to overlay) that would almost for sure guarantee me 150 points.

I believe that by spending time on bypassing the models instead of trying to fix non-existing issues I had a high chance to reach 150 first, especially that for a few days I was leading in the competition by a huge margin. The competition was an interesting challenge, however technical issues took a big portion of the pleasure away.