Intrusion Detection (continued)

Why Anomaly based Intrusion Detection schemes?

Intrusion Detection Techniques can be placed in the following three main categories

- Behavior/policy based schemes
- Misuse detection based schemes
- Anomaly detection based schemes

Behavior/policy based schemes are closely related to sandboxing kind of techniques. While this kind of techniques might specify some kind of preventive action on detection of an attack, the alternate scheme based on anomaly detection does not specify any such preventive actions. Yet, it is becoming more popular today for a variety of reasons that will become clear in the discussion that follows.

Misuse detection doesn't meet the main goal of an IDS which is to be the last line of defense, i.e., that it captures attacks that may be missed by other defenses --- misuse detection is able to detect only attack types that have previously been seen, and chances are that you have other defenses already in place against these. Anomaly detection, on the other hand, is able to fulfill this role of detecting attacks that no one has thought about so far, and hence have no other defenses.

When we are talking of anomaly based attack detection, we do not know what an attack looks like. But we flag an attack when we see some kind of anomalous behavior. The assumption here being that attack => anomaly. However, anomaly needs to be defined with respect to some model of normal behavior. Typically, this is done using a training process. If attacks occur during the training period, then those attacks will become part of the definition of normal, and hence may not be detected. Secondly, anomalous behavior needs to be defined on the basis of a set of features that are captured in the normal behavior profile. It is possible that some attacks don't manifest changes in the features that are observed by an anomaly detector, thus becoming another reason for false negatives.

Of course, anomaly does not imply an attack, so there may be lots of anomalous behaviors that may not be attacks. This leads to *false positives.* Often, anomaly detection techniques have high FP rate that makes it difficult to deploy them in real-world settings, although this is slowly changing.

What is normal behavior? It is something that is learnt by the scheme. Good training data can help to build a model of the normal acceptable behavior of an application/program.

In earlier days, IDS systems were supposed to be non-intrusive. Inline techniques like those that involved modifying application code, intercepting requests made to an application etc, were considered intrusive.

Anomaly Detection Techniques

There are a variety of anomaly detection techniques and a detailed discussion of all of them is beyond the scope of this course. We shall explore one of the techniques which is based on **System call characterization** and understand how it can be applied in the context of application programs. The rest of the discussion is about anomaly based IDS for capturing attacks on application programs.

Why System calls?

A lot of knowledge about the problem domain is encoded into the application. We want to be able to leverage this knowledge about how the application behaves to detect anomalous behavior. It is therefore right to think that system calls characterize the way an application works.

Alternatively if we were to look at network data and monitor it in order to detect attacks, we wouldn't be able to leverage the above mentioned feature of application programs. Also, there is often a gap in between how applications interpret the data that flows into them and what the data looks like on the network.

Steps in building an anomaly based IDS

1. Build a behavior model
2. Detect deviations from this model and flag attacks.

Building the behavior model

The idea is to train the system to learn acceptable behavior of an application program. This training has to be done in the real world so that all scenarios that could happen in a deployment scenario can be learnt. One factor we need to consider here is that when the learning process is on, the environment should be attack free.

Why don't we consider a sandbox kind of environment for learning the behavior model? Because in a sandbox environment the model would not be complete and would only include a subset of acceptable and valid behaviors. It can be a good starting point, though --- test inputs could be used to exercise certain known aspects of program behavior in depth, so that those aspects don't have to be learnt in a deployment setting.

Good training data is one that will help maximize the learning of good behavior.

When building a model, one has to also think about the performance of the scheme which is going to use the model. The size of the model is therefore an important parameter.

Side discussion: Is replaying network traffic in a sandbox environment a good option/idea?

Network Replay is a hard problem because of many factors. E.g. Consider the current scenario where majority of the network traffic is http based and involves sessions, cookies etc. Cookies, for instance, are generated by the server, and are simply returned by the client the next time it submits a form. The replayer needs to be smart enough to recognize this, and substitute the cookies in the original trace with the cookie that is returned by the server during the replay. This is just one of the problems, and there are others. There is some ongoing research in this area, e.g., see Protocol-Independent Adaptive Replay of Application Dialog but we are far from the point where we can say that we know how to solve it.

Also, in order to replay we again need network data that is free of attacks. So we are back to the same problem of ensuring that the training data is attack free.

Research (What methods have been explored so far?)

N-gram method

- The main idea here is to look at system call sequences and mark attacks based on anomalous sequences.

- In this method, there is a sliding window of size N that is used to look at the syscall sequences, N at a time. This is done at runtime.

- The behavior model could be built using a sophisticated model like the Markov model or using data mining techniques, but the Ngram method uses a simple memory based scheme where acceptable Ngrams are memorized. It has been found that the results of memory based Ngram learning models is almost as good as those that use the more involved and sophisticated techniques (which are expensive). So Ngrams are a fairly good idea in this context.

The slides contain an illustration of how Ngrams are used.

Drawbacks A large N would ensure a more accurate model, but also would mean that the space needed is large and that the training period needed in longer. It would also require that an extremely large number of patterns be examined. All this leads to a conclusion that N should be small. However, this tradeoff will mean that long term correlations cannot be captured.

- Since acceptable Ngrams are memorized exactly as they are and not as generalizations, in order to build a complete behavior model, the time of training required is very long.

Representation o f the Behavior models

When dealing with identifying sequences of system calls, the fact that Finite State Automata renders itself naturally to identifying sequences that have no bound on their lengths comes to mind. An if-else kind of a program structure, as well as loops, can also be easily captured using an automaton. Thus FSA seem like a good way to represent program behaviors. More generally, automata models can capture control-flow behaviors of programs projected onto system calls, i.e., they can capture the sequences of system calls that can be issued by an application.

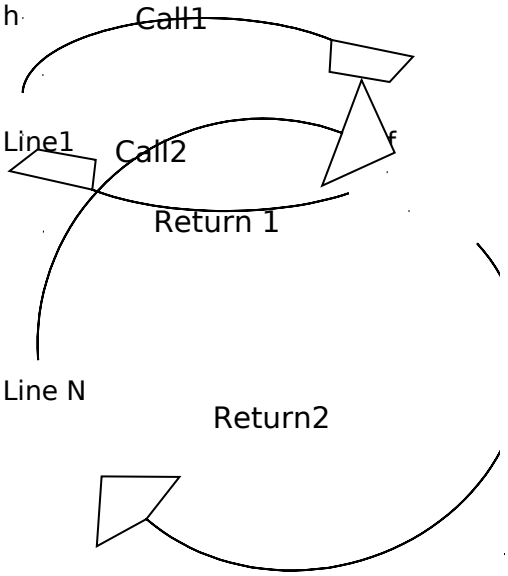
With respect to the kind of automata to be used, there are many options –

1. **Finite State Automaton**- this can capture only a regular language

2. **Extended FSA** - It provides for having some memory. Such an automaton may be necessary in some cases. E.g. A certain kind of behavior model may involve checking that the number of open system calls is equal to the number of close system calls. In this case we would need some kind of memory to be able to keep track of the number of opens seen, and then compare them with close operations.

3. **Pushdown automaton** - this can capture CFGs and allows for a stack. With this we can capture more complicated behavior models like call/returns. Some of the attacks that an FSA may miss can therefore be captured using PDA.

4. h



E.g. consider the following case where a program *h* makes calls to a function *f* at two different locations in the code. If we don't have a stack then we may not be able to detect a condition where Call 1 is followed by Return 2. The FSA will not be able to able to match call-returns and will allow the above mentioned behavior to go undetected, but the PDA will capture it.

The PDA seems like a natural way to represent program structures. Procedure call and returns can also be neatly represented using PDAs. Thus, we see that the PDA can capture all kinds of program structures. Since the behavior of functions is supposed to be independent of the location in another program from which they are called. This can be used as the basis to build *modular* models, where a model of a procedure is learnt independent of how it is used. .

What is the algorithm to extract/learn the model?

1) **Static analysis** – Constructing models by studying source code or binary code.
   Advantage– If the static analysis is sound (as they usually are), then the model will capture all control flows that are ever possible in that program. This means that you won't have false positives.
   Disadvantage – Overly permissive since many of the control flows may be ones that don't happen in usual circumstances. In fact, with sound static analysis, the only flows that are not captured are those that are incompatible with the program semantics under all executions and for all input values (usually, these analyses ignore values of program variables in order to make the analysis problem tractable). This usually means that they detect behaviors that arise due to memory corruption attacks, and nothing else.

2) **Runtime approaches using machine learning -** The space of all possible behaviors of a program is usually very large and in the real world only a few subset of them are acceptable. The task at hand for an anomaly detection based IDS scheme is to be able to detect behaviors that fall outside this subset.

   To be able to capture the runtime behavior anomalies is the motivation for this approach.

Learning sequences of strings (like in Ngrams)

Consider the scenario of learning system call sequences. Given a set of system call sequences that the program at hand has exhibited, to build an automata model from just this data is not computationally feasible. (It is of course possible to build N-gram models.) You need to have some additional information to be able to build an automata model.

Learning FSA models

An example of building an FSA model from a sample intercepted program behavior is depicted on slide 27. The slides show an analysis of an example program in terms of the system calls it makes and how an automaton is created for it. The slides also contain a graph that depicts how, with more and more training, behavior models should eventually converge to be able to generate a reasonable number of false positives. The graph on the slide gives a comparison between the Ngram and FSA based methods.

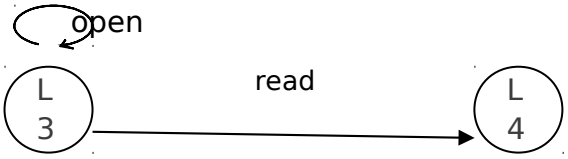
To build the model, we picked up the system call and the location in the program from which the system call was made. When using a system call interception technique like ptrace, we can find the location from where the system call was made by knowing the address of the int80 based software interrupt instruction that is executed to make the system call. However the problem here is that this location will be inside libc, since all syscalls are made via libc. In order to find the actual location from which a syscall was made, you need to do more – you need to examine the runtime stack of a program to trace back the call to a location within the executable segment of the process, and use this address as the location of the syscall and this is the technique commonly employed.

Learning PDA models
As in case of FSAs, since a PDA builds a call return model, we would again need to examine the runtime stack to be able to match up call and return addresses.

PDA versus FSA

- FSA does not capture call-returns accurately since both are captured as control transfers. This might result in the FSA not being able to detect a control path anomaly where the call happens from the call site of the first invocation of a method and the return happens to the call site of the second/other invocation. This, in literature, is called as an “impossible path problem”.

- FSA technique leads to inaccuracies due to system calls made within libraries. We recollect here, the approach of moving up the stack till a return address was found within the executable region and then using this as the location of the system call. The idea that inaccuracies can occur can be best understood with an example. Consider a program that makes a call to a routine in libc and assume that this routine in libc makes two system calls *open* followed by *read*. If L is the location in the program from where this routine in libc was called, we would end up with a self loop that depicts the behavior as shown below. Note that this model will allow an arbitrary number of open calls to be made from location L, as opposed to the single call that was in the training data.



Mimicry attacks

It is an evasion attack where the attacker has knowledge of the kind of behavior that is acceptable to an IDS. Such an attack needs to be constructed with knowledge of the model constructed by the IDS.

As one can expect, the coarser a model is, easier it is to make mimicry attacks. They are easiest with Ngrams and most difficult with PDA.

Note that in a mimicry attack, an attacker may want to make an exec call, but the model may not permit it in the current state of the model. To construct a successful attack, it is necessary to examine the model, and construct a sequence of system calls that take you from the current state to another state in the automaton, and this sequence should end with the desired system call (i.e., execve in this example). Thus, a mimicry attack involves making a sequence of system calls. If the IDS is not examining the call stack, then injected code can be used to make this sequence of calls. If it examines the stack, then any calls made from injected code can be identified by the IDS (which expects all returns addresses on the stack to point to code regions rather than data regions.) This problem can be avoided if the attack requires only a single call S': the code can set up the parameters, and then jump to another location in the program that calls S'. Now there is no return address of injected code that is left on the stack, as it used a jump rather than a call. However, if a sequence of calls needs to be made, this does not work. This is because when the system call returns, control won't return back to the injected code. Here is an interesting paper that shows how this can be done:

Automating Mimicry Attacks Using Static Binary Analysis

(This is in case you are really interested – the work is far too specialized to be of general interest in a course like ours.)

It is reasonable to say that evasion attacks like mimicry attacks seem like a problem only when there is a possibility of injected code due to typical memory corruption kind of attacks.

Learning system call arguments

When only system call names are examined to build behavior models, there are many aspects of program execution that cannot be captured. E.g. exec of shell would be considered same as exec of another process like */s*. We can see that examining system call arguments is clearly going to make the behavior model more powerful.

In the context of system call argument examination, we can think of 2 categories –

- One of them is focused on inputs and outputs. E.g. if you think of a server, a strong behavior model would be able to capture the structure of inputs and outputs of the server program. This would be called the content based intrusion detection scheme.

- The other category includes everything else other than data that is read and written by a program. e.g. the files that are opened. It is often easier to work with this category.

The latter category is what the rest of this discussion is about. The main idea is to be able to understand the properties of the arguments to system calls. E.g. We may want to specify that the filename argument associated with an open syscall is of the form "/home/user/* " (prefix property).

You can have a layered approach where with a control flow FSA, you can layer the argument learning algorithm. In practical scenarios this is what is done.

E.g. Open system call may be made from states L1 and L2, so you could define that in state L1, Open takes an argument with prefix /home/folder1/___ and from L2, it takes an argument with prefix /home/folder2

One type of property that you may learn is the prefix property. Another type of property you may learn is about relationships between arguments.

There are two types of relationships:

- Unary relations – In which we look at a single system call at a time and then try to learn some behavior for its arguments. E.g. the first argument has a value X always.

- Binary relations – Looks at two system calls and the relationship between the arguments of these system calls. E.g. you may want to specify that the file descriptor returned by the open system call is the same that is used by the read system call.

Learning Binary Relationships is often very useful and powerful in being able to detect attacks. E.g. if we consider a tar program, then we may want to specify that the directory argument with which the tar program is called is the same as the prefix of all files that are opened by the program to add to the tar.