Genetically modified honeypot

Wim Mees

March 15, 2003

1 Introduction

So you bought one of Lance Spitzner's books, read the `securityfocus.com` articles on honeypots, and now you want to start your own honeypot. You have decided to use honeyd in order to be able to adopt any platform identity you like, and to use simple scripts to emulate some popular services.

But after a while, running a fixed configuration honeypot gets boring. Your honeypot cannot really be compromised since you are only using simple scripts that emulate specific services yet are not really vulnerable to the corresponding security holes. Furthermore, if you were interested in analyzing how crackers compromise a honeypot, you would be using a different honeypot setup anyway.

What might really be interesting is if you would be able to configure a range of virtual honeypots, each one listening on a different IP address and emulating a different version of for instance sendmail. If you would then analyze the data and look for instance for which version numbers the incoming connections most often go beyond grabbing the banner and continue the session, whereas for others connections tend to break of after receiving the banner, you would be able to conclude which version(s) of sendmail the attackers are after at the moment.

You could for instance configure all recent and even some future version numbers and look for peeks of activity as a function of the version number advertised by the service emulator script. This would allow you to know of new vulnerabilities as soon as they are getting known in the blackhat community and well before they are officially published.

In order to implement such a "version number popularity contest" using honeyd, you would have to write a modified service emulation script that can distinguish between a banner grab and a continued connection, as well as develop some further processing scripts to identify those version numbers where there are significantly more continued connections than banner grabs.

Unfortunately, you do not have enough IP address space for such a setup, and you're wary of having to manage all this. This is were the **genetically modified honeypot** comes in. The idea behind it is not new. In fact, it is based on the theory of evolution, formalized by Charles Darwin who wrote already in 1859:

> The affinities of all the beings of the same class have sometimes been represented by a great tree. I believe this simile largely speaks the truth. The green and budding twigs may represent existing species; and those produced during each former year may represent the long succession of extinct species.
>
> ...
>
> The limbs divided into great branches, and these into lesser and lesser branches, were themselves once, when the tree was small, budding twigs; and this connexion of the former and present buds by ramifying branches may well represent the classification of all extinct and living species in groups subordinate to groups.
>
> ...
>
> From the first growth of the tree, many a limb and branch has decayed and dropped off, and these lost branches of various sizes may represent those whole orders, families, and genera which have now no living representatives, and which are known to us only from having been found in a fossil state.
>
> ...

> As buds give rise by growth to fresh buds, and these, if vigorous, branch out and overtop on all a feebler branch, so by generation I believe it has been with the Tree of Life, which fills with its dead and broken branches the crust of the earth, and covers the surface with its ever branching and beautiful ramifications.

The concept of applying the theory of evolution to an engineering problem dates back to 1964, when Ingo Rechenberg applied the biologic principles of mutation and selection to an optimisation problem in his work *"Evolutionsstrategie"*.

In the 1970s, John Holland, assisted by students and colleagues, developed the techniques of *"genetic algorithms (GAs)"* and they are nowadays well known and widely used in the artificial intelligence community. To our knowledge, this concept has however never been used for building an "intelligent" honeypot.

In our genetically modified honeypot, we will create a random initial population of version numbers (the version number here represents the genome of an individual in a biological population), that will evolve in time. From generation to generation, following changes will occur:

- some random individuals will undergo a **mutation**, for instance version number 8.9.1 will become 8.8.1 when the middle digit was changed because of a mutation.
- some random pairs of individuals will engage in a **cross-over**, where for instance (7.8.1 ; 8.3.4) becomes (7.3.4 ; 8.8.1) when the cross-over occurs right after the first digit.
- some random individuals of the population will die and be replaced by newborns. These newborns will inherit their genetic material - their version number - from existing individuals in the population. This will happen in such a way that those existing individuals that are best adapted to their environment have the highest probability of donating their genetic information to the newborns. This is called **selection**.

For each new incoming connection, a random individual of the population is selected. So in practice, each time you connect, you may see a different version number published by the service in its banner.

In our application, we define that "being adapted to one's environment" for a version number means that the client is willing to go beyond grabbing the banner.

As a result of the selection, each following generation will have more and more individuals that have those version numbers that the attack scripts are after. As a result of the mutations and cross-overs on the other hand, we will always have some "exotic" individuals that explore new values for the version number and can - when all of a sudden they become very popular - rapidly increase in numbers and eventually take over a population for a while.

It is precisely this rise of popularity of a specific version number or family of version numbers that we are interested in. Such an event signals the apparition of a new attack-script or worm that hunts for vulnerable servers having this version installed.

2 Realization

For the moment, our development is limited to a genetically modified service, that runs behind an unmodified version of honeyd. In term, we would however like to extend this to the emulation of a population of platform identities (e.g. "Win2K - IIS" versus "Linux 2.4 - Apache"), which will require a hook into honeyd to dynamically modify this identity from connection to connection.

We have developed a generic service emulator in Java that provides the necessary functionalities for emulating different application-level protocols using genetically evolving characteristics (which can be more than just version numbers).

In the context of a service emulation on a honeypot, where you typically won't have huge transfers of information and a large numbers of connections, performance is not an issue. We therefore choose Java as a programming language.

The application protocol that the service emulator must speak, is modeled using a finite state machine (FSM). This FSM is defined in a configuration file in XML format. A DTD is used to protect to a certain degree the correctness of the configuration file. Any XML editor can be used to edit the configuration file.

The configuration consists of:

- **state** elements.
  They are in turn defined using following sub-elements:
  - **label** (mandatory): used for labeling the state in a graphical representation, as well as further down for defining the source and target states of a transition.
  - **species** (optional): to define a population of individuals (e.g. version numbers) that will evolve. A species is defined by an upper-bound and a lower-bound genome, for instance [0.0.0-3.9.9] means that the first digit of the version number can vary from 0 to 3, the second from 0 to 9 and the third from 0 to 9 as well.
  - **individual** (optional): when this state becomes active, a random individual from the species will be selected and its genome (e.g. version number) stored in a variable.
  - **operation** (mandatory): defines what is happening in this state. Following operations are possible: ”put” to indicate that a welcome message or a reply is written from the server to the client, ”get” to indicate that a request is read from the client, ”nop” when no interaction with the client happens in this state, ”begin” to indicate a begin state (must be unique) and ”end” to indicate a state where processing ends (of which there may be more than one).

    It should be noted here that during a ”get” operation the command from the client is parsed and the resulting elements are stored in variables, which can thereafter be used in a ”put” command in order to write an appropriate status message or in a transition condition in order to decide whether a transition triggers or not. Different types of variables are available (”cstring”, ”string”, ”float”, ”int”). The above extracted individual from a population is accessed in the same way as any other variable.

    Finally, there are also ”macro’s” available for obtaining up to date information from the system, for instance the date and time.

- **transition** elements.
  They are defined using following elements:
  - **label** (mandatory): used for labeling the transition in a graphical representation.
  - **from** (mandatory): the source state for this transition.
  - **to** (mandatory): the target state to which this transition leads.
  - **condition** (optional): a condition that defines when this transition triggers and the target state becomes active.

    When a state has been processed, all its outgoing transitions are evaluated.

    At most one outgoing transition with an associated condition may then fire. If there is one such transition, it is triggered and its target state becomes active. For defining a transition condition, the variables used when defining the states can be used, as well as constants. Following operators are available: less than (”<”), greater then (”>”), equal (”=”), logical AND (”ˆ ”), and logical OR (”|”). Parentheses should be used to define the order of evaluation in more complex expressions.

    If there is no transition with an associated condition that triggers, the system looks for an outgoing transition without an associated condition. These transitions are called the ”default transitions”. When there is such a transition that originates in the currently active state, its target state becomes the next active state.

3 Usage

see README file

4 Future work

We would like to extend the genetically modified honeypot so that it automatically learns its finite-state-machine models from parcing a capture file of a connection.

We would furthermore like to add a graphical FSM editor in order to manage the xml files.

It would also be useful to be able to have a field in a ”put” state that allows us to write out a complete file, in order to easily emulate a mini-webserver.

And finally, all feature proposals are obviously welcome.