

Seeing the Invisible

Forensic Uses of Anomaly Detection and Machine Learning

Federico Maggi
Dipartimento di Elettronica e
Informazione
Politecnico di Milano
Via Ponzio 34-5, I-20133
Milano, Italy
federico.maggi@polimi.it

Stefano Zanero
Dipartimento di Elettronica e
Informazione
Politecnico di Milano
Via Ponzio 34-5, I-20133
Milano, Italy
stefano.zanero@polimi.it

Vincenzo Iozzo
Dipartimento di Elettronica e
Informazione
Politecnico di Milano
Via Ponzio 34-5, I-20133
Milano, Italy
vincenzo.iozzo@mail.polimi.it

ABSTRACT

Anti-forensics is the practice of circumventing classical forensics analysis procedures, making them unreliable or impossible. In this paper we propose the use of machine learning algorithms and anomaly detection to cope with a wide class of definitive anti-forensics techniques. We test the proposed system on a dataset we created through the implementation of an innovative technique of anti-forensics, and we show that our approach yields promising results in terms of detection.

Categories and Subject Descriptors

K.5.m [Legal Aspects of Computing]: Miscellaneous—*computer forensics*; K.6.5 [Management of Computing and Information Systems]: Security and Protection—*Unauthorized access (e.g., hacking, phreaking)*

General Terms

Documentation, Experimentation, Legal Aspects

Keywords

Computer forensics; Host-based anomaly detection

1. INTRODUCTION

Computer forensics is usually defined as the process of applying scientific (repeatable) analysis processes to data and computer systems, with the objective of producing evidence that can be used in an investigation or in a court of law. More in general, it is the set of techniques that can be applied to understand if, and how, a system has been used or abused to commit mischief [?]. The increasing use of forensic techniques has led to the development of “anti-forensic” techniques that can make this process difficult, or impossible [?, ?, ?].

If the *identification* phase is targeted, we have *transient* anti-forensics techniques, which make the acquired evidence difficult to analyze with a specific tool or procedure, but not impossible to analyze in general. If instead the *acquisition* phase is targeted, we have the more effective class of *definitive* anti-forensics techniques, which effectively deny once and forever any access to the evidence. In this case, the evidence may be destroyed by the attacker, or may simply not exist on the media. This is the case of in-memory injection techniques, which we will investigate in this paper.

In particular, we propose the use of machine learning algorithms and anomaly detectors to circumvent such techniques. We illustrate a prototype of anomaly detector which analyzes the sequence and the arguments of system calls to detect intrusions. We also use this prototype to detect in-memory injections of executable code, and in-memory execution of binaries (the so-called “userland exec” technique, which we reimplement in a reliable way). This creates a usable audit trail, without needing to resort to complex memory dump and analysis operations [?, ?].

The remainder of this work is organized as follows: in Section 2 we introduce the problem of anti-forensics, and reference related works in this young area. In Section 3 we introduce the key concepts and structure of our prototype for system call anomaly detection. In Section 4 we describe the experimental setup we used to test its usefulness in a forensic environment. In Section 5 we report the results we obtained in an experimental evaluation of the usefulness of the prototype in the test environment. In Section 6 we draw our conclusions and outline future research perspectives.

2. PROBLEM STATEMENT

As we observed in Section 1, anti-forensics is defined by symmetry on the traditional definition of computer forensics: it is the set of techniques that an attacker may employ to make it difficult, or impossible, to apply scientific analysis processes to the computer systems he penetrates, in order to gather evidence [?, ?, ?]. The final objective of anti-forensics is to reduce the quantity and spoil the quality [?] of the evidence that can be retrieved by an investigation and subsequently used in a court of law.

Following the widely accepted partition of forensics [?] in *acquisition*, *identification*, *evaluation*, and *presentation*, the two phases where technology can be critically sabotaged are both acquisition and identification. Therefore, we can define “anti-forensics” as comprising all the methods that make acquisition, preservation and analysis of computer-generated and computer-stored data difficult, unreliable or meaningless for law enforcement and investigation purposes.

Even if more complex taxonomies have been proposed [?], we can use the traditional partition of the forensic process to distinguish among two types of anti-forensics:

- **Transient anti-forensics**, when the *identification* phase is targeted, making the acquired evidence difficult to analyze with a specific tool or procedure, but not impossible to analyze in general.
- **Definitive anti-forensics**, when the *acquisition* phase is targeted, ruining the evidence or making it impossible to acquire.

Examples of transient anti-forensics techniques are the fuzzing and abuse of filesystems in order to create malfunctions or to exploit vulnerabilities of the tools used by the analyst, or the use of log analysis tools vulnerabilities to hide or modify certain information [?, ?]. In other cases, entire filesystems have been hidden inside the metadata of other filesystems [?], but techniques have been developed to cope with such attempts [?]. Other examples are the use of steganography [?], or the modification of file metadata in order to make filetype not discoverable. In these cases the evidence is not completely unrecoverable, but it may escape any quick or superficial examination of the media: a common problem today, where investigators are overwhelmed with cases and usually undertrained, and therefore overly reliant on tools.

Definitive anti-forensics, on the other hand, effectively denies access to the evidence. The attackers may encrypt it, or securely delete it from filesystems (this process is sometimes called “counter-forensics”) with varying degrees of success [?, ?]. Access times may be rearranged to hide the time correlation that is usually exploited by analysts to reconstruct the events timeline. The final anti-forensics methodology is not to leave a trail: for instance, modern attack tools (commercial or open source) such as Metasploit [?], Mosdef or Core IMPACT[?] focus on pivoting and in-memory injection of code: in this case, nothing or almost nothing is written on disk, and therefore information on the attack will be lost as soon as it is powered down, which is usually standard operating procedure on compromised machines. These techniques are also known as “disk-avoiding” procedures.

Memory dump and analysis operations have been advocated in response to this, and tools are being built to cope with the complex task of the reliable acquisition [?, ?] and analysis [?, ?, ?] of a modern system’s memory. However, even in the case that the memory can be acquired and examined, if the process injected and launched has already terminated, once more, no trace will be found of the attack: these techniques are much more useful against in-memory resident backdoors and rootkits, which by definition are persistent.

3. SYSTEM CALL ANOMALY DETECTION USING SEQUENCE AND PARAMETERS

Most of the actions that an aggressor would try to perform (e.g., accessing the host file system, sending or receiving a packet over the network, executing another program, etc.) require the use of a system call. Thus, it is reasonable to monitor such calls in order to analyze the behavior of a process. In particular, we propose to use anomaly detection techniques to flag anomalous or suspicious executions and record them for review in order to create a trail (i.e., the alert logs) that would otherwise be lost. We will use a technique we introduced in [?, ?] which makes use of both the

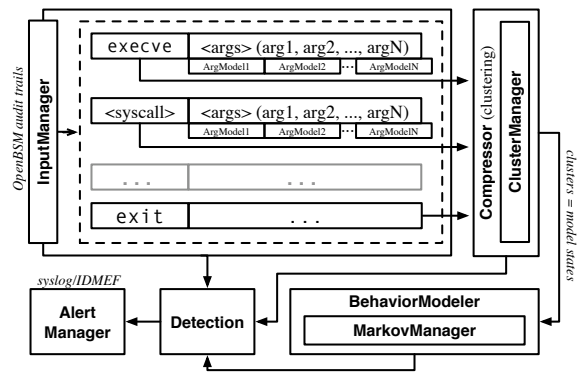


Figure 1: The architecture of our HIDS prototype

sequence and the content of system calls to detect anomalies. This has been shown to be more efficient than using sequences of syscalls only, something which has been studied for a long time since the seminal work [?]. We re-engineered and extended the proposal found in [?, ?] to use Markov models of the sequence (as in, e.g., [?]) complemented with an analysis of the arguments of the system calls.

The resulting system is shown in Figure 1, and it is synthetically described in the following. Each execution of an application is modeled as a sequence of system calls, $S = [s_1, s_2, s_3, \dots]$, logged by the operating system auditing facilities. Each system call s_i is characterized by a *type* (e.g. `read`, `write`, `exec`, etc.), a list of *arguments* (e.g., the path of the file to be opened by `open`), a *return value*, and a *timestamp*. The return value is not taken into account, neither the *absolute* timestamp (the sequence of the system calls is considered instead).

Our system must be trained in order to “learn” a model of the normal behavior of the monitored applications. During this phase, the system builds a profile for each application (e.g. `sendmail`, `telnetd`, etc.). A two-phase process of machine learning is then applied to each type of system call separately. Firstly, a single-linkage, bottom-up agglomerative hierarchical clustering algorithm [?] is used to find, for each type of system call, sub-clusters of invocations with similar arguments. We are interested in creating models on these clusters, and not on the general system call, in order to better capture normality and deviations on a more compact input space. This is important because some system calls, most notably `open`, are used in very different ways. Indeed, `open` is probably the most used system call on UNIX-like systems, since it opens files or devices in the file system creating a descriptor for further use. Only by careful aggregation over its parameters (i.e., the file path, a set of flags indicating the type of operation, and an opening mode) we can de-multiplex the general system call into “sub-groups” that are specific to a single function. In order to do this, we must define a way to measure “distance” among arguments, as we will show.

Afterwards, the system builds models of the parameters inside each cluster. The type of models, as well as the type of distances used for agglomeration, depends on the type of parameter, as shown in Table 1. In our framework, the

Table 1: Association of models to Syscall arguments in our prototype

SYSCALL	MODEL USED FOR THE ARGUMENTS
open	pathname → Path Name flags → Discrete Numeric mode → Discrete Numeric
execve	filename → Path Name argv → Execution Argument
setuid, setgid	uid, gid → User/Group
setreuid, setregid	ruid, euid → User/Group
setresuid, setresgid	ruid, euid, suid → User/Group
symlink, link rename	oldpath → Path Name newpath → Path Name
mount	source, target → Path Name flags → Discrete Numeric
umount	target → Path Name flags → Path Name
exit	status → Discrete Numeric
chown lchown	path → Path Name group, owner → User/Group
chmod, mkdir creat	path → Path Name mode → Discrete Numeric
mknod	pathname → Path Name mode, dev → Discrete Numeric
unlink, rmdir	pathname → Path Name

distance among two system calls, s_i and s_j , is the sum of distances between corresponding arguments $D(s_i, s_j) = \sum_{a \in A_s} d_{\text{model}(a)}(s_i^a, s_j^a)$ (being A_s the shared set of system call arguments). For each couple of corresponding arguments a we compute the distance as:

$$d_a = \begin{cases} K_{(\cdot)} + \alpha_{(\cdot)} \delta_{(\cdot)} & \text{if the elements are different} \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

where $K_{(\cdot)}$ is a fixed quantity which creates a “step” between different elements, while the second term is the real distance between the arguments $\delta_{(\cdot)}$, normalized by a parameter $\alpha_{(\cdot)}$. We use “ (\cdot) ” to denote that such variables are parametric w.r.t. the type of argument.

Since hierarchical clustering does not offer a concept analogous to the “centroid” of partitioning algorithms that can be used for classifying new inputs, we also created, for each cluster, a stochastic model that can be used to cluster (or classify) further inputs. These models generate a *probability density function* that can be used to state the probability with which the input belongs to the model. It is important to note that it is not strictly necessary for the model and its distance or probability functions to be the same distance function that is used for clustering purposes.

As can be seen in Table 1, at least 4 different types of arguments are passed to system calls: path names and file names, discrete numeric values, arguments passed to programs for execution, users and group identifiers (UIDs and GIDs).

Path names and file names are very frequently used in system calls. They are complex structures, rich of useful information, and therefore difficult to model properly. For the clustering phase, we chose to use a very simple model, the directory tree depth. This is easy to compute, and ex-

perimentally leads to fairly good results. Thus, in Equation 1 we set δ_a to be the difference in depth, e.g. let $K_{\text{path}} = 5$ and $\alpha_{\text{path}} = 1$; comparing `/usr/lib/libc.so` and `/etc/passwd` we obtain $d_a = 5 + (3 - 2) * 1 = 6$, while comparing `/usr/lib/libc.so` and `/usr/lib/libelf.so.1` we obtain $d_a = 0$. The stochastic model for path names is a probabilistic tree which contains all the directories involved with a probability weight for each. Filenames are often too variable to be considered, so we consider the leaves of the tree and if they are too different we simply ignore them for that specific model.

Discrete numeric values such as flags, opening modes, etc. are usually chosen from a limited set. Therefore we can store all of them along with a discrete probability. Since in this case two values can only be “equal” or “different”, we set up a binary distance model for clustering, where the distance between x and y is:

$$d_a = \begin{cases} K_{\text{disc}} & \text{if } x \neq y \\ 0 & \text{if } x = y \end{cases}$$

and K_{disc} , as usual, is a configuration parameter. In this case, the generation of probability for a new input to belong to the model is straightforward.

We also noticed that *execution argument* (i.e. the arguments passed to the `execve` syscall) are difficult to model, but we found the length to be an extremely effective indicator of similarity of use. Therefore we set up a binary distance model, where the distance between x and y is:

$$d_a = \begin{cases} K_{\text{arg}} & \text{if } |x| \neq |y| \\ 0 & \text{if } |x| = |y| \end{cases}$$

denoting with $|x|$ the length of x and with K_{arg} a configuration parameter. In this way, arguments with the same

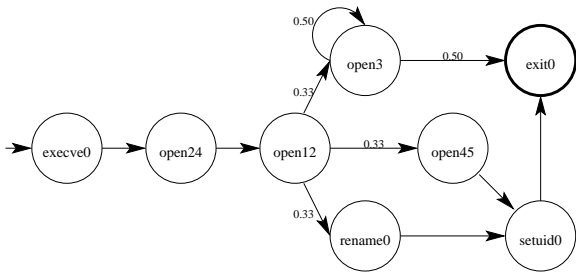


Figure 2: A sample of the resulting Markov model with the clusters of system calls as states

length are clustered together. For each cluster, we compute the minimum and maximum value of the length of arguments. Fusion of models and incorporation of new elements are straightforward. The probability for a new input to belong to the model is 1 if its length belongs to the interval, and 0 otherwise.

We developed an ad-hoc model for *users and groups* identifiers. These discrete values have three different meanings: UID 0 is reserved to the super-user, low values usually are for system special users, while real users have UIDs and GIDs above a threshold (usually 1000). So, we divided the input space in these three groups, and computed the distance for clustering using the following formula:

$$d_a = \begin{cases} K_{uid} & \text{if belonging to different groups} \\ 0 & \text{if belonging to the same group} \end{cases}$$

and K_{uid} , as usual, is a user-defined parameter. Since UIDs are limited in number, they are preserved for testing, without associating a discrete probability to them. Fusion of models and incorporation of new elements are straightforward. The probability for a new input to belong to the model is 1 if the UID belongs to the learned set, and 0 otherwise.

In order to take into account the execution *context* of each system call, we use a first order Markov chain to represent the program flow. The model states represent the system calls, or better they represent the various clusters of each system call, as detected during the clustering process. For instance, if we detected three clusters in the `open` syscall, and two in the `execve` syscall, then the model will be constituted by five states: `open1`, `open2`, `open3`, `execve1`, `execve2`. Each transition will reflect the probability of passing from one of these groups to another through the program. A sample of such a model is shown in Figure 2. This approach was investigated in former literature [?, ?, ?, ?, ?], but never in conjunction with the handling of parameters and with a clustering approach.

During training, each execution of the program in the training set is considered as a sequence of observations. Using the output of the clustering process, each syscall is classified into the correct cluster, by computing the probability value for each model and choosing the cluster whose models give out the maximum composite probability along all known models: $\max(\prod_{i \in M} P_i)$. The probabilities of the Markov model are then straightforward to compute.

During the detection phase, each system call is considered in the context of the process. The cluster models are once again used to classify each syscall into the correct cluster: the probability value for each model is computed and the stored cluster whose models give out the maximum composite probability ($P_c = \max(\prod_{i \in M} P_i)$) is chosen being representative of the “system call class”. Three distinct probabilities can be taken into account in order to build proper anomaly thresholds:

- P_s , the probability of the *execution sequence* to fit the Markov model up to now;
- P_c , the probability of the *system call* to belong to the best-matching cluster;
- P_m , the *latest transition* probability in the Markov model.

We fuse the last two into a probability value of the single syscall, $P_p = P_c \cdot P_m$. A second, separate value for the *sequence probability* P_s is kept. Using the training data, appropriate threshold values are calculated by considering the lowest probability over all the dataset for that single program (for both P_s and P_p). We then choose a sensitivity parameter for scaling such value, giving the final *anomaly threshold*. A process is flagged as malicious if either P_s or $P_p = P_c \cdot P_m$ are lower than the anomaly threshold. For avoiding a P_s which quickly decreases to zero for long sequences, we introduced a “scaling” of the probability calculation based on the geometric mean, by introducing a sort of “forgetting factor”: $P_s(l) = \sqrt[l]{\prod_{i=1}^l P_p(i)}$. In this case, we demonstrated [?] that $\text{P}[\lim_{l \rightarrow +\infty} P_s(l) = 0] = 1$, but it converges much more slowly (l is the sequence length). Experimentally, this latter scaling function leads to much better results in terms of false positive rate.

4. EXPERIMENTAL SETUP

A well-known problem in IDS research is the lack of reliable sources of test data, except for the well-known and abused datasets created by the Lincoln Laboratory at MIT, also known as “DARPA IDS Evaluation dataset” [?]. These datasets contain BSM auditing data for Solaris systems, directory tree snapshots and the content of sensitive directories, and inode data where available. However, their generation method makes them unsuitable for testing systems dedicated to forensic analysis and incident response. In fact, many authors already analyzed the network datasets, finding many shortcomings and regularities [?, ?]. We analyzed host-based logs, and concluded that they are artificially simple and regular, as they contain only a handful of executions, all very similar among themselves since they are generated through simple, scripted sequences. This can cause the overfitting of any anomaly model. Various other anomalies we have found are reported in [?, ?].

We tried to avoid repeating such shortcomings in our experiments. In order to show that our system is also capable of detecting the in-memory injection of code, and of creating an audit trail which can be used for forensics purposes, while at the same time reducing the logged data to the bare minimum that is needed, we generated an experimental dataset

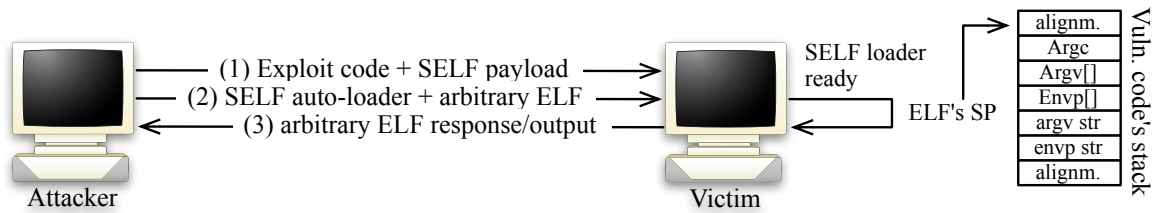


Figure 3: An illustration of the in-memory execution technique we developed and used for this paper

for two console applications: `bsdtdar` and `eject`. Our testing platform is an Intel x86 machine running a basic installation of FreeBSD 6.2, on which we recompiled the kernel enabling auditing capabilities. Since our systems, and other host-based anomaly detectors [?, ?], accept input in the BSM format, the OpenBSM [?] auditing tools collection has been used for collecting audit trails (i.e., system calls sequences and their details). We audited vulnerable releases of `eject` and `bsdtdar`, namely: `mcweject 0.9` (which is an alternative to the `eject` which is bundled with FreeBSD 6.2) and the release of `bsdtdar` distributed with FreeBSD 6.2.

The `eject` executable has a small set of command line option and a very plain execution flow. For the simulation of a legitimate user, we simply chose different permutations of flags and different devices. For this executable, we manually generated 10 executions, which are remarkably similar (as expected).

Creating a dataset of normal activity for the `bsdtdar` program is more challenging. It has a large set of command line options, and in general is more complex than `eject`. While the latter is generally called with an argument of `/dev/*`, the former can be invoked with any argument string, for instance `bsdtdar cf myarchive.tar /first/path /second/random/path` is a perfectly legitimate command line. Using a process similar to the one used for creating the IDEVAL dataset, and in fact used also in other works such as [?], we prepared a shell script which re-creates the pseudo-random behavior of an user who creates or extracts archives. The randomization takes into account the different usage of flags by different users: for instance, many users prefer to uncompress an archive using `tar xf archive.tar`, many others still use the dash `tar -xf archive.tar`, and may use the “verbose” option as well. Running inside a snapshot of a real-world desktop filesystem, our tool randomizes such variations and similar aspects. To simulate user activity, it randomly creates files of various size and content both around the system (in the case of superusers), and into an user’s own home directory. Once the filesystem has been populated, our tool randomly walks around the system directory tree and creates TAR archives, with random sets of command line flags and target directories. Similarly, archives are expanded using the above explained flag randomization procedures.

In such a setup, the audit trails keep changing, along with the simulated user behavior. It is important to underline that normal users would never use really random names for their files and directories, they usually prefer to use words

from their tongue plus a limited set of characters (e.g., `.`, `-`, `_`) for concatenating them. Therefore, we rely on a large dictionary of words for generating filenames.

We have chosen these two applications because they have been recently found to be vulnerable to two different buffer overflow vulnerabilities that allow to execute arbitrary code. In the case of `mcweject 0.9`, the vulnerability [?] is a very simple stack overflow, caused by improper bounds checking. By passing a long argument on the command line, an aggressor can execute arbitrary code on the system with root privileges. There is a public exploit for the vulnerability [?] which we modified slightly to suit our purposes and execute our own payload. The attack against `bsdtdar` is based on a publicly disclosed vulnerability in the PAX handling functions of `libarchive 2.2.3` and earlier [?], where a function in file `archive_read_support_format_tar.c` does not properly compute the length of a buffer when processing a malformed PAX archive extension header (i.e., it does not check the length of the header as stored in a header field), resulting in a heap overflow which allows code injection through the creation of a malformed PAX archive which is subsequently extracted by an unsuspecting user on the target machine. In this case, we developed our own exploit, as none was available online, probably due to the fact that this is a heap overflow and requires a slightly more sophisticated exploitation vector. In particular, the heap overflow allows to overwrite a pointer to a structure which contains a pointer to a function which is called soon after the overflow. So, our exploit overwrites this pointer, redirecting it to the injected buffer. In the buffer we craft a clone of the structure, which contains a pointer to the shellcode in place of the correct function pointer.

In the tests conducted we used a modified version of SELF [?], which we improved in order to reliably run under FreeBSD 6.2 and ported to a form which could be executed through code injection (i.e., to shellcode format). This tool implements a technique known as “Userland Exec”: by overwriting the program headers of any statically linked ELF binary, and by building a specially-crafted stack it allows an attacker to load and run that ELF in the memory space of a target process without calling the kernel and, more importantly, without leaving any trace on the hard disk of the attacked machine. This is done through a two-stage attack where a shellcode is injected in the vulnerable program, and then retrieves a modified ELF from a remote machine, and subsequently injects it into the memory space of the running target process, as shown schematically in Figure 3.

With a regular shellcode		
	FPR	DR
<code>eject</code>	0%	75%
<code>bsdtar</code>	7.81%	71%

With userland exec		
	FPR	DR
<code>eject</code>	0% (unchanged)	100%
<code>bsdtar</code>	7.81% (unchanged)	100%

Table 2: Experimental results with a regular shellcode and with our userland exec implementation.

5. RESULTS

In the setup detailed above, we performed several experiments with both `eject` and `bsdtar`. We trained our anomaly detector with ten different execution of `eject` and more than a hundred executions of `bsdtar` (using our random activity generator). We also audited eight instances of the activity of `eject` under attack, while for `bsdtar` we logged seven malicious executions. We repeated the tests both with a simple shellcode which opens a root shell (a simple `execve` of `/bin/sh`) and with our implementation of the userland exec technique.

The overall results are summarized in Table 2. Let us consider the effectiveness of the detection of the attacks themselves. The attacks against `eject` are detected with no false positive at all. The exploit is detected in the very beginning: since a very long argument is passed to the `execve`, this triggers the argument model. The detection accuracy is similar in the case of `bsdtar`, even if in this case there are some false positives. The detection of the shellcode happens with the first `open` of the unexpected special file `/dev/tty`. It must be underlined that most of the true alerts are correctly fired at system call level; this means that malicious *calls* are flagged by our IDS because of their unexpected arguments, for instance.

On the other hand, exploiting the “Userland Exec” an attacker launches an otherwise normal executable, but of course such executable has different system calls, in a different order, and with different arguments than the ones expected in the monitored process. This reflects in the fact that we achieved a 100% detection rate with no increase in false positives, as each executable we have run through SELF has produced a Markov model which significantly differs from the learned one for the exploited host process.

6. CONCLUSIONS

In this paper we analyzed the wide class of *definitive* anti-forensics techniques which try to eliminate evidence by avoiding disk usage. In particular, we focused on in-memory injection techniques. Such techniques are widely used by modern attack tools (both commercial and open source).

As memory dump and analysis is inconvenient to perform, often not part of standard operating procedures, and does not help except in case of in-memory resident backdoors and rootkits, we proposed an alternative approach to circumvent such techniques. We illustrated how a prototype which analyzes (using learning algorithms) the sequence and the

arguments of system calls to detect intrusions can be used to detect in-memory injections of executable code, and in-memory execution of binaries.

We proposed an experimental setup using vulnerable versions of two widely used programs on the FreeBSD platform, `eject` and `bsdtar`. We described the creation of a training and testing dataset, how we adapted or created exploits for such vulnerabilities, and how we recorded audit data. We also developed an advanced in-memory execution payload, based on SELF, which implements the “userland exec” technique through an injectable shellcode and a self-loading object (a specially-crafted, statically linked ELF file). The payload executes any statically linked binary in the memory space of a target process without calling the kernel and, more importantly, without leaving any trace on the hard disk of the attacked machine.

We performed several experiments, with excellent detection rates for the *exploits*, but even more importantly with a 100% detection rate for the in-memory execution payload itself. We can positively conclude that our technique yields promising results for creating a forensic audit trail of otherwise “invisible” injection techniques. Future developments of this work will include a more extensive testing with different anti-forensics techniques, and the development of a specifically designed forensic output option for our prototype.

Acknowledgments

The authors would like to thank, for a number of ideas and discussions, Dr. Matteo Matteucci, co-author of a related paper. We also thank Luigi Drago and Orlando Bassotto for their helpful suggestions.

7. REFERENCES