Seeing the Invisible

Forensic Uses of Anomaly Detection and Machine Learning

Federico Maggi
federico.maggi@polimi.it

Stefano Zanero
stefano.zanero@polimi.it

Vincenzo Iozzo
vincenzo.iozzo@mail.polimi.it

Dipartimento di Elettronica e Informazione
Politecnico di Milano
Via Ponzio 34-5, I-20133
Milano, Italy

ABSTRACT

Anti-forensics is the practice of circumventing classical forensics analysis procedures making them either 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 [26]. The increasing use of forensic techniques has led to the development of "anti-forensic" techniques that can make this process difficult, or impossible [11, 4, 16].

There are two wide classes of anti-forensics techniques: *transient* techniques make the acquired evidence difficult to analyze with a specific tool or procedure, but not impossible to analyze in general. *Definitive* anti-forensics techniques instead 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 to use 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. Afterwards, we use this prototype to detect in-memory injections of executable code, and in-memory execution of binaries. To do this, we reimplement in a reliable way the so-called "userland exec" attack technique. We demonstrate that our prototype can detect the anomalies that such techniques create in the program flow, by detecting anomalous syscalls. This creates a usable audit trail, without needing to resort to complex memory dump and analysis operations [5, 34].

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 [11, 4, 16]. The final objective of anti-forensics is to reduce the quantity and spoil the quality [14] 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 [33] 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 [16],

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 [10, 14]. In other cases, entire filesystems have been hidden inside the metadata of other filesystems [14], but techniques have been developed to cope with such attempts [32]. Other examples are the use of steganography [20], 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 [13, 12]. Access times may be rearranged to alter the activity timeline that is usually exploited by analysts to correlate events. The final anti-forensics methodology is not to leave a trail: for instance, modern attack tools (commercial or open source) such as Metasploit [2], Mosdef or Core IMPACT [8] 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 the system 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 tasks of reliable acquisition [5, 35] and analysis [5, 34, 30] of a modern system's memory. However, even if the memory can be acquired and examined, if the injected process has already terminated, no trace of the attack will be found: 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 packets over the network, executing another program, etc.) require the use of one or more system calls. Thus, it is reasonable to monitor such calls in order to analyze the behavior of a process. In particular, we propose to use anomaly

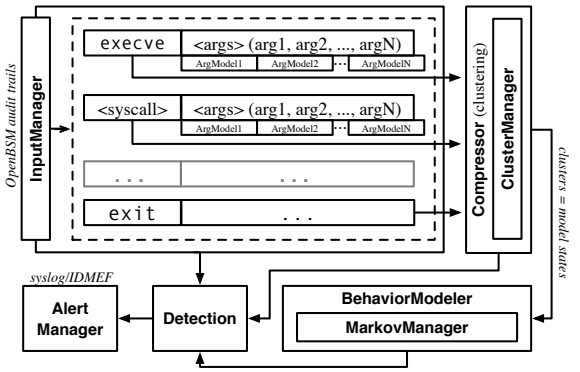

Figure 1: The architecture of our HIDS prototype

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 [23, 38] which makes use of both the 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 [9]. We reengineered and extended the proposal found in [22, 27] to use Markov models of the sequence (as in, e.g., [21]) complemented with an analysis of the arguments of the system calls.

The resulting system is shown in Figure 1. Each execution of an application is modeled as a sequence of system calls, $S = [s_1, s_2, s_3, \ldots]$, 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 distinct 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 [15] 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.

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

Syscall	Model used for the arguments
`open`	`pathname` → Path Name `flags, 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,newpath` → Path Name
`mount`	`source, target` → Path Name `flags` → Discrete Numeric
`umount`	`target,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
`mknode`	`pathname` → Path Name `mode, dev` → Discrete Numeric
`unlink, rmdir`	`pathname` → Path Name

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, depend on the type of parameter, as shown in Table 1. In our framework, the distance among two system calls, s_i and s_j, is the sum of distances between corresponding arguments $D(s_i, s_j) = \sum_{a \in \mathrm{A}_s} d_{\mathrm{model}(a)}(s_i^a, s_j^a)$ (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 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 not strictly necessary for such model, or its distance or probability functions, to be the same as the distance functions that are 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 (`UID`s and `GID`s).

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 experimentally leads to fairly good results. Thus, in Equation 1 we set δ_a to be the difference in depth. 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 if the leaves of the tree 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:

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

and K_{disc}, as usual, is a configuration parameter. In this case, the generation of the probability density function is straightforward.

We also noticed that *execution arguments* (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:

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

denoting with $|x|$ the length of x and with K_{arg} a configuration parameter. In this way, arguments with the same length are clustered together. For each cluster, we compute

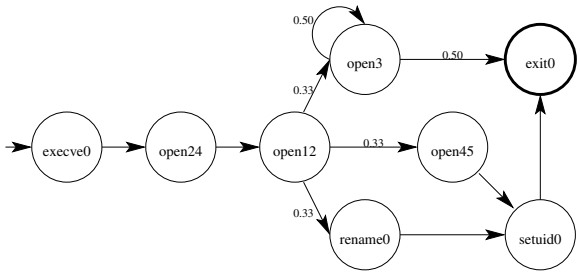

Figure 2: A sample of the resulting Markov model with the clusters of system calls as states

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 *user and group* 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 `UID`s and `GID`s 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 `UID`s 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 Markov chain (i.e. a first order Markov model) 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 have five states: $\texttt{open}_1$, $\texttt{open}_2$, $\texttt{open}_3$, $\texttt{execve}_1$, $\texttt{execve}_2$. 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 [6, 7, 18, 31, 19, 21], 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: $\max(\prod_{i \in \mathrm{M}} P_i)$. The probabilities of the Markov model are then straightforward to compute.

Since training should happen, ideally, on the machine which will be monitored, it is important to notice that the prototype is resistant to the presence of a limited number of outliers (e.g. abruptly terminated executions or attacks) in the training set, because the resulting transition probabilities will drop near zero. For the same reason, it is also resistant to the presence of any cluster of anomalous invocations created by the clustering phase. Therefore, the presence of a minority of attacks in the training set will not adversely affect the learning phase, which in turn does not require an attack-free training set, and thus it can be performed on the deployment machine.

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 ($P_c = \max(\prod_{i \in \mathrm{M}} P_i)$) is chosen as the "system call class". Three distinct probabilities can be taken into account in order to build 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 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": $P_s(l) = \sqrt[2l]{\prod_{i=1}^{l} P_p(i)^i}$ (where l is the sequence length). In this case, we demonstrated [23] that $\mathrm{P}\left[\lim_{l \to +\infty} P_s(l) = 0\right] = 1$, but it converges more slowly. Experimentally, this latter scaling function leads to much better results in terms of false positive rate.

Our current prototype is implemented in C. Both the clustering phase and the behavioral analysis are multithreaded, and the results of both procedures are stored in a binary format but can be dumped in XML for manual inspection, if needed. At runtime, the prototype dinamically loads the program profiles it needs, and stores them in memory. The prototype can send output to standard output, syslog facilities, and/or to log files in IDMEF format.

We profiled the code with `gprof` and `valgrind` for CPU and memory requirements. The throughput for the training phase varies between 6120 and 10228 syscalls per second. The training phase is also memory consuming, with

a worst-case peak during our tests of about 700 MB. The performance observed in the detection phase varies between 12395 and 22266 syscalls/sec. Considering that the kernel of a typical machine running services such as HTTP/FTP on average executes system calls in the order of thousands per second (e.g., around 2000 system calls per second for `wu-ftpd` [27]), the overhead introduced by our IDS is noticeable but does not severely impact system operations.

Of course, for using the prototype in a real system, there is an issue of survivability, i.e. wheteher or not an intruder can compromise the auditing system. This is a common issue for any type of logging system: as soon as the host is compromised at root level, any running auditing program cannot be trusted anymore. However, compromising our prototype would entail uploading a training file which accepts the attacker's actions as normal. This is definitely nontrivial: the best choice for an attacker would probably be to deactivate our system altogether. However, this can happen only in the post-exploitation phase, whereas detection hopefully happens during exploitation. If the logger is configured to forward alerts to a remote syslog server, the attacker would not be able to easily circumvent it.

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 datasets" or IDEVAL [1]. 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 [25, 24]. 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 anomaly models. Various other anomalies we found are reported in [23, 38].

We tried to avoid repeating such shortcomings in our experiments. In order to show that our system is 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 for two console applications: `bsdtar` 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 system accepts input in the BSM format, we used OpenBSM [37] to collect audit trails (i.e. system calls sequences and their details). We audited vulnerable releases of `eject` and `bsdtar`, namely `mcweject 0.9` (which is an alternative to the `eject` command bundled with FreeBSD 6.2) and the release of `bsdtar` 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 various 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 `bsdtar` 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 `bsdtar 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 [36], we prepared a shell script which emulates the pseudo-random behavior of an user who creates or extracts archives. The randomization takes into account the different way in which users pick flags: 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, the script 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.

It is important to underline that normal users rarely choose random names for their files and directories, they usually prefer to use common words 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 [28] 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 [17] which we modified slightly to suit our purposes and execute our own payload. The attack against `bsdtar` is based on a publicly disclosed vulnerability in the PAX handling functions of `libarchive 2.2.3` and earlier [29], 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

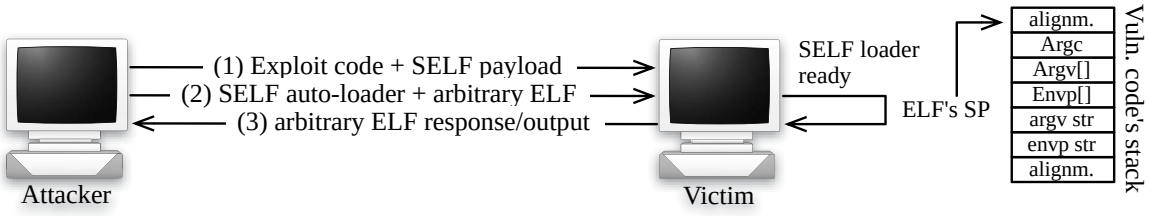

Figure 3: An illustration of the in-memory execution technique we developed and used for this paper

a pointer to the shellcode in place of the correct function pointer.

In the tests we also used a modified version of SELF [3], 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 accomplished 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.

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`, randomized as described above. 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. In the latter we injected four different statically built payloads (`sash`, `links`, `fget`, and the `sudoku` command line game); in the case of `bsdtar` we gathered 8 executions by invoking `bsdtar` with two different command line option sets; in the case of `eject` we injected 8 different payload (the same used for `bsdtar` plus `portsentry`, `tree`, `pstree`, `less`) and we audited eight different executions.

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. It must be noted that, in the case of userland execution with SELF, we were able of reaching 100% because our IDS is easily triggered by in memory attacks; in fact, executing the injected payload significantly modifies the normal behavior of the process more than a classic exploit does. Also note that to test the accuracy of the prototype we used attack-free data.

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 processes.

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 it is not part of standard operating procedures, and it 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 a 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.

Details	*Detection accuracy*		
PROGRAMS	DR % = $\frac{TP}{TP+FN}$%		FPR % = $\frac{FP}{FP+TN}$%
Test env.:	**(a) w/o userland execution**	**(b) w/ userland execution (SELF)**	**(c) Attack-free data**
`eject` (no. of execs.)	**75%** = $\frac{6}{6+2}$%	**100%** = $\frac{8}{8+0}$%	0% = $\frac{0}{0+404}$%
`bsdtar` (no. of execs.)	**70.6%** = $\frac{12}{12+5}$%	**100%** = $\frac{4 \cdot 2}{4 \cdot 2+0}$%	7.81% = $\frac{20}{20+236}$%

Table 2: Experimental results for DR with a (a) regular shellcode and (b) with our userland exec implementation (SELF). Test environment (c) is related to the data used to compute the FPR.

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

[1] Darpa intrusion detection evaluation datasets. available online at `http://www.ll.mit.edu/IST/ideval/data/dataindex.html`.

[2] Mafia: Metasploits anti forensics investigation arsenal. available online at `http://metasploit.com/projects/antiforensics/`.

[3] Advanced antiforensics self. available online at `http://www.phrack.org/issues.html?issue=63&id=11`, 2005.

[4] H. Berghel. Hiding data, forensics, and anti-forensics. *Commun. ACM*, 50(4):15–20, 2007.

[5] M. Burdach. In-memory forensics tools. available online at `http://forensic.seccure.net/`.

[6] J. B. D. Cabrera, L. Lewis, and R. Mehara. Detection and classification of intrusion and faults using sequences of system calls. *ACM SIGMOD Record*, 30(4), 2001.

[7] G. Casas-Garriga, P. Díaz, and J. Balcázar. ISSA: An integrated system for sequence analysis. Technical Report DELIS-TR-0103, Universitat Paderborn, 2005.

[8] Core Security Technologies. CORE Impact. `http://www.coresecurity.com/?module=ContentMod&action=item&id=32`.

[9] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for Unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, Washington, DC, USA, 1996. IEEE Computer Society.

[10] J. Foster and V. Liu. Catch me if you can.... In *Blackhat Briefings 2005*, Las Vegas, NV, August 2005.

[11] S. Garfinkel. Anti-Forensics: Techniques, Detection and Countermeasures. In *Proceedings of the 2nd International Conference on i-Warfare and Security (ICIW)*, pages 8–9, 2007.

[12] S. Garfinkel and A. Shelat. Remembrance of data passed: a study of disk sanitization practices. *Security & Privacy Magazine, IEEE*, 1(1):17–27, 2003.

[13] M. Geiger. Evaluating Commercial Counter-Forensic Tools. In *Proceedings of the 5th Annual Digital Forensic Research Workshop*.

[14] Grugq. The art of defiling: defeating forensic analysis. In *Blackhat briefings 2005*, Las Vegas, NV, August 2005.

[15] J. Han and M. Kamber. *Data Mining: concepts and techniques*. Morgan-Kauffman, 2000.

[16] R. Harris. Arriving at an anti-forensics consensus: Examining how to define and control the anti-forensics problem. In *Proceedings of the 6th Annual Digital Forensic Research Workshop (DFRWS '06)*, volume 3 of *Digital Investigation*, pages 44–49, September 2006.

[17] "harry". Exploit for CVE-2007-1719. available online at `http://www.milw0rm.com/exploits/3578`.

[18] S. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6:151–180, 1998.

[19] S. Jha, K. Tan, and R. A. Maxion. Markov chains, classifiers, and intrusion detection. In *Proceedings of the 14th IEEE Workshop on Computer Security Foundations (CSFW'01)*, pages 206–219, Washington, DC, USA, June 2001. IEEE Computer Society.

[20] N. Johnson and S. Jajodia. Exploring steganography: Seeing the unseen. *COMPUTER*, 31(2):26–34, 1998.

[21] A. P. Kosoresow and S. A. Hofmeyr. Intrusion detection via system call traces. *IEEE Softw.*, 14(5):35–42, 1997.

[22] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the Detection of Anomalous System Call Arguments. In *Proceedings of the 2003 European Symposium on Research in Computer Security*, Gjøvik, Norway, October 2003.

[23] F. Maggi, M. Matteucci, and S. Zanero. Detecting intrusions through system call sequence and argument analysis. Submitted for publication, 2007.

[24] M. V. Mahoney and P. K. Chan. An analysis of the 1999 DARPA / Lincoln laboratory evaluation data for network anomaly detection. In *Proceedings of the 6th International Symposium on Recent Advances in Intrusion Detection (RAID 2003)*, pages 220–237, Pittsburgh, PA, USA, September 2003.

[25] J. McHugh. Testing intrusion detection systems: a critique of the 1998 and 1999 DARPA intrusion

detection system evaluations as performed by lincoln laboratory. *ACM Trans. on Information and System Security*, 3(4):262–294, 2000.

[26] G. M. Mohay, A. Anderson, B. Collie, R. D. McKemmish, and O. de Vel. *Computer and Intrusion Forensics*. Artech House, Inc., Norwood, MA, USA, 2003.

[27] D. Mutz, F. Valeur, G. Vigna, and C. Kruegel. Anomalous system call detection. *ACM Trans. Inf. Syst. Secur.*, 9(1):61–93, 2006.

[28] National Vulnerability Database. CVE-2007-1719. available online at `http://nvd.nist.gov/nvd.cfm?cvename=CVE-2007-1719`.

[29] National Vulnerability Database. CVE-2007-3641. available online at `http://nvd.nist.gov/nvd.cfm?cvename=CVE-2007-3641`.

[30] J. Nick L. Petroni, A. Walters, T. Fraser, and W. A. Arbaugh. Fatkit: A framework for the extraction and analysis of digital forensic data from volatile system memory. *Digital Investigation*, 3(4):197–210, december 2006.

[31] D. Ourston, S. Matzner, W. Stump, and B. Hopkins. Applications of Hidden Markov Models to detecting multi-stage network attacks. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences*, page 334, 2003.

[32] S. Piper, M. Davis, G. Manes, and S. Shenoi. *Detecting Hidden Data in Ext2/Ext3 File Systems*, volume 194 of *IFIP International Federation for Information Processing*, chapter 20, pages 245–256. Springer, Boston, 2006.

[33] M. Pollitt. Computer forensics: an approach to evidence in cyberspace. In *Proc. of the National Information Systems Security Conference*, volume II, pages 487–491, Baltimore, Maryland, 1995.

[34] S. Ring and E. Cole. Volatile Memory Computer Forensics to Detect Kernel Level Compromise. In *Proceedings of the 6th International Conference on Information And Communications Security (ICICS 2004)*, Malaga, Spain, October 2004. Springer.

[35] B. Schatz. Bodysnatcher: Towards reliable volatile memory acquisition by software. In *Proceedings of the 7th Annual Digital Forensic Research Workshop (DFRWS '07)*, volume 4 of *Digital Investigation*, pages 126–134, September 2007.

[36] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, Washington, DC, USA, 2001. IEEE Computer Society.

[37] R. Watson. Openbsm. `http://www.openbsm.org`, 2006.

[38] S. Zanero. *Unsupervised Learning Algorithms for Intrusion Detection*. PhD thesis, Politecnico di Milano T.U., Milano, Italy, May 2006.