SEVENTH FRAMEWORK PROGRAMME
Theme SEC-2011.2.5-1 (Cyber attacks against critical infrastructures)

D7.2 Prelminary report on host-based compromise detection

Contract No. FP7-SEC-285477-CRISALIS

Workpackage	WP7 - Host-Driven Analysis and Detection
Author	Davide Balzarotti, Damiano Bolzoni, Heiko Patzlaff
Version	1.0
Date of delivery	M24
Actual Date of Delivery	M24
Dissemination level	Public
Responsible	IEU

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement n°285477.

SEVENTH FRAMEWORK PROGRAMME
Theme SEC-2011.2.5-1 (Cyber attacks against critical infrastructures)

The CRISALIS Consortium consists of:

Symantec Ltd.	Project coordinator	Ireland
Alliander		Netherlands
Chalmers University		Sweden
ENEL Ingegneria e Innovazione		Italy
EURECOM		France
Security Matters BV		Netherlands
Siemens AG		Germany
Universiteit Twente		Netherlands

Contact information:
Dr. Matthew Elder
Symantec Limited
Ballycoolin Business Park (GA11-35)
Blanchardstown, Dublin 15
Ireland

e-mail: matthew_elder@SYMANTEC.COM

Contents

Abstract

In the context of work package 7, we propose a number of approaches and techniques to protect hosts against sophisticated targeted attacks. The goal is to use features extracted from applications, documents, process behavior, and operating systems as a whole – to design anomaly-based detection and protection systems.

In this preliminary deliverable we describe two complementary approaches, that cover threat scenarios not fully addressed by existing methodologies. In the first half of the document we present the implementation in a custom hypervisor solution of a behavioral monitor that can detect suspicious uses of different operating system resources. In the last chapter we discuss instead a novel solution to detect malicious documents, that are one of the main attack vector used by criminals to infect a system.

1 Introduction

One of the main goals of the CRISALIS project is to propose solutions to monitor the Operational Technology network infrastructure for signs of intrusions and anomalies. This is certainly one of the more accurate and less invasive approaches to prevent attacks against critical infrastructures. However, network-based approaches also suffers from a number of limitations. First, this kind of solutions lack visibility on the state of the internal operating system processes and on their mutual interactions. As a result, there is a significant loss of context information that could be used to detect attacks. Moreover, if an attacker manages to deliver a malicious payload without sending the exploit over the network (for instance by injecting it inside legitimate documents and project files) a network-based approach will be unable to report the intrusion.

The threats we are tackling in the context of CRISALIS involve sophisticated attackers with internal knowledge about the critical infrastructure's network and its components. In this scenario, a criminal may be able to gain access to its target without alerting network-based sensors. Hence, the mere deployment of network monitoring approaches is going to potentially blind a target organization from a number of attack vectors. The solution we propose in WP7 is to naturally complement network-based techniques with a number of host-based monitoring solutions.

While traditional antivirus softwares are still by far the most deployed approach in this category, due to the custom and sophistication of the attacks we need to consider, classic signatures solutions are often ineffective. For this reason, we decided to tackle the detection of attacks and host compromises by using anomaly-based approaches.

Similarly to the network domain, there exist several anomaly-based approaches that one can apply at the host level. In this deliverable we choose to focus on two complementary approaches, one that detects when the exploitation of a vulnerability is executed and the other one that detects the malicious behavior exhibited by an infected process, due to the fact that it needs to access OS resources (files, registry keys, etc.) in an abnormal way.

We believe that this multi-layer approach can provide a very high level of protection by working in parallel at two different level of abstraction: a close monitor protecting the documents that may be try to exploit the system, and a high-level monitor protecting the entire operating systems against mis-behaving programs and malicious applications.

In this deliverable we present the preliminary work we are conducting in these two

directions. First, in Chapter 2, we present our system-wide solution implemented in a custom hypervisor. Then, in Chapter 3, we describe the technique we propose to detect malicious documents that may be introduced in the system. In both cases we discuss previous work that tackles these issues, discuss their limitations and propose better approaches to detect advanced attack vectors at the host level.

2 Hypervisor-Based Behavioral Detection

In this chapter we discuss the design and implementation of AccessMiner, a system-centric behavioral malware detector. Our system, originally proposed at the ACM conference on Computer and Communications Security (CCS) in 2010, is designed to model the general interactions between benign programs and the underlying operating system (OS). In this way, AccessMiner is able to capture which, and how, OS resources are used by normal applications and detect anomalous behavior in realtime.

One of the main advantages of our approach is that it does not require to be trained on malicious samples, and therefore it is able to provide a general detection solution that can be used to protect against both known and unknown malware. However, our original system suffered from two important limitations that made it unsuitable for use in a critical infrastructure environment. First, while it was very successful in detecting both known and unknown attacks, it suffered from false positives when the software on the system in which it was installed was modified (e.g., by a software update). This limited the applicability of Accessminer in very dynamic systems. Second, the system was implemented as part of the operating system, and it was therefore unable to protect against sophisticated kernel attacks.

To make the system more resilient against tampering, we re-designed and re-implemented AccessMiner as a custom hypervisor that sits below the operating system. Our experiments show that in a stable environment AccessMiner can provide a high level of protection (around 90% detection rate with zero false positives) with an acceptable overhead - similar to the one that can be experienced in a state of the art virtual machine environment.

2.1 Introduction

The problem of detecting attacks and malicious applications at the host level has been largely studied by both the research and the industrial communities. The most common solutions are based either on matching static signatures or on using behavioral models to specify allowed or forbidden behaviors. Signatures works well to identify single malware instance but they quickly become ineffective when the attacker adopts obfuscated or polymorphic code. At the same time, most behavior-based detection techniques follow

a *program-centric* approach that focuses on modeling the execution of individual programs. These models often lack the context to capture how *generic* benign and malicious programs interact with their environment and with the underlying operating system. As a result, detectors based on program-centric behavioral techniques tend to raise alerts whenever a new program is encountered or an existing program is used in a different way. This typically leads to unacceptably high false positive rates - thus limiting the practical applicability of these approaches.

AccessMiner [28] introduced a novel *system-centric* technique to model the activity of benign programs. The main idea behind the AccessMiner approach is that, given enough training, it is possible to identify common patterns in the way benign applications interact with the operating system resources. For instance, while normal programs typically write only to their own directories (and to temporary directories), malware often attempt to tamper with other applications and critical system settings, often residing outside the normal application "scope". As a result, special *access activity models* can be derived by AccessMiner only by looking at the execution of a broad set of benign applications. Therefore, traces of malware execution, often problematic to collect from a coverage and diversity point of view, are not required to train our classifiers.

One characteristic of Accessminer that is very important for critical infrastructures is the fact that the algorithm perform best when the system on which it is installed does not change too often over time. Ideally, for systems that almost never updated, the results obtained with Accessminer are extremely good – both in terms of detection and false positive rates.

However, while our experiments showed that a system-centric approach was successful in identifying a large amount of diverse malware samples with very few false positives, a number of important points were not addressed in the original paper. In particular, the original approach was designed to be implemented as part of the Windows operating system kernel. However, the threat scenario rapidly changed in the last years, moving from traditional user-space malware toward more sophisticated rootkits techniques. The rise and nature of targeted attacks (especially in the scope of critical infrastructures) also pose new challenges that are not fully addressed by current methodologies. For example, by carefully combining a mix of social engineering, zero days exploits for unknown Windows vulnerabilities, and stolen certificates to sign kernel modules - motivated and well-funded attackers can quickly subvert the target OS and remain undetected for long period of times (as the Stuxnet [19], Duqu [46], and Flame [47] incidents have shown).

Since a successful targeted attack could easily tamper with OS-based detection mechanisms, in this chapter we present a complete re-design of AccessMiner and we describe how the same approach can be implemented as a custom hypervisor. This new solution makes the detector much more resilient to sophisticated attacks, but it also introduces

several technical problems and challenges. First, in order to collect the same information and monitor the system calls issued by each process, a hypervisor has to solve the so-called *semantic gap*. Even though many solutions exists for this problem, current hypervisor-based detection countermeasures do not scale well to several scenarios (e.g., critical infrastructures), due to their high computational requirements that conflict with the strict timing constraints of the running applications. The challenge here is to use a light-weight approach that does not impact the performance of the system in a prohibitive way.

2.2 System Call Data Collection

In this section, we discuss our efforts to collect a large and diverse set of system call traces. Our requirements are geared towards imposing the least impact on the users whose machines are part of the data collection effort. Thus, the data collection framework must have minimal impact on the performance of those machines, must operate with and without network connectivity, must ensure that private information does not leave the user's machines, and must make almost no assumptions about the run-time environment. For example, requiring that users make use of virtual machines would significantly restrict the practical applicability of our data collection. Additionally, the data collection framework must be capable of extracting a rich set of attributes for each event (i.e., system call) of interest. Unfortunately, none of the existing system call tracing tools satisfy these requirements, so we built and deployed our own data collection framework.

Our system consists of a number of software agents, which, once installed on user's machines, automatically collect, anonymize, and upload system call logs, and a central data repository, which receives logs from each machine and normalizes the data in preparation for further analysis. The software agents can be installed by users on their own machines and are mindful of system load, available disk space, and network connectivity. Furthermore, users can enable and disable the collection agent as they wish.

Our analysis and training algorithms need several information regarding each system call. Therefore, our sensors were designed to collect they syscall number and its arguments, its result (return) code, the process ID, the process name, and the parent process ID. Each log entry is represented by a tuple in the form:

$$\langle timestamp, program, pid, ppid, system\ \ call, args, result \rangle$$

This data allows us to perform our analyses within a single process, across multiple executions of the same program, or across multiple programs.

2.2.1 Raw Data Collection

The software agent that collects data is a real-time component running on each user's machine. This agent consists of a data collector and a data anonymizer. We implemented our agent for Microsoft Windows, as it is the OS targeted the most by malware. The description in the remainder of this section provides details specific to the Microsoft Windows platform. The data collector is a Microsoft Windows kernel module that traces system call events and annotates them with additional process information. The data anonymizer transforms the collected system call data according to privacy rules and uploads it to the remote, central data repository.

Kernel collector. The main goal of this component is to collect system call and process information *across the entire system.* In order to intercept and log system call information, the kernel data collector hooks the SSDT table [22]. The kernel collector logs information for 79 different system calls in five categories: 25 related to files, 23 related to registries, 25 to processes and threads, one related to networking, and five related to memory sections.

A challenge arises from the fact that the kernel collector does not necessarily observe the start of a new process. One reason is that the user can disable and re-enable the software agent at any point. Another reason is that the kernel collector is started as the last kernel module in the system boot process. This means that the kernel collector might observe system calls that refer to previously acquired resource handles, but without having any information about which resources those handles point to. As a special case, some resource handles (e.g., handles to the registry roots) are automatically provided to a process by the OS at process-creation time. Consequently, if we log only the parameters for each individual system call that we observe, we lose information about previously (or automatically) acquired resources. To address this problem, we query the open handler table for each process we have not seen before. This allows the kernel collector to retrieve the open objects already associated with a new process. We store the path names of these objects for later use, for example when we intercept a system call (such as `NtOpenKey`) that references a pre-existing handle.

Log anonymizer. To protect the privacy of our users, we obfuscate or simply remove arguments of various system calls before sending the log to the data repository. The obfuscation consists of replacing part or the entire sensitive argument value with a randomly-generated value. Every time a value repeats, it is replaced with the same randomly-generated value, so that we can recover correlations between system call arguments. We consider as sensitive all arguments whose values specify non-system paths (e.g., paths under `C:\Documents and Settings` are sensitive), all registry keys below the user-root registry key (`HKLM`), and all IP addresses. Furthermore, we remove all buffers

read, written, sent, or received, thus both providing privacy protection and reducing the communication to the data repository. The data repository indexes the logs by the primary MAC address of each machine.

Impact on performance. We designed the software agent to minimize the overhead on users' activities. The kernel module collects information only for a small subset of the 79 system calls. Log are saved locally and processed out of band before being sent to the server, when network connectivity is available. Users can turn data collection on and off, based on their needs. Local logs are uploaded to the repository when they reach 10 MB in size and logging is automatically stopped if available disk space drops below the 100 MB threshold. Each 10 MB portion of the system call log is compressed using ZIP compression, for an 95% average reduction in size (from 10 MB to 500 KB). Given these techniques, we are confident that users were able to use their computers with the data collector present as they would normally do, and thus the collected system call logs are representative of day-to-day usage.

2.2.2 Data Normalization

The purpose of this component is to process the raw system call logs and extract the fully qualified names of the accessed resources as well as the access type. For files and directories, the fully qualified name is the absolute path, while for registry keys, it is the full path from one of the root keys.

To compute fully qualified resource names, we track for each process the set of resources open at any given time, via the corresponding set of OS handles. When a resource (file or registry key) is accessed relative to another resource (either opened by the process or opened by the OS automatically for the process), we combine the resource names to obtain a fully qualified name.

Computing the access type (e.g., read, write, or execute) requires tracking the access operations performed on a resource. This is more tricky than expected. When a resource is acquired by a program (e.g, a file is opened), the program specifies a desired level of access. This information, however, is not sufficiently precise for our needs. This is because, often, programs open files and registry keys at an access level beyond their needs. For example, a program might open a file with `FULL_ACCESS` (i.e., both read and write access), but afterward, it only reads from the file. Since we are interested in the actual access type, we track all of the operations on a resource, and only when the resource is released (on `NtClose`), we compute the access type as a union of all operations on the resource.

In Microsoft Windows, there is no single system call that starts a new process from a given executable file. In order to retrieve the execution path and file name, the normalizer

Machine	*Data* (GB)	*System calls* ($\times 10^6$)	*Processes* ($\times 10^3$)	*Applications*
1	18.0	285	55.1	90
2	4.5	70	22.4	87
3	5.6	89	17.7	46
4	32.0	491	110.9	41
5	34.0	514	125.6	42
6	14.0	7	2.8	73
7	1.3	19	3.7	49
8	1.2	18	3.0	22
9	1.6	27	8.5	47
10	2.3	36	12.9	26
Total	114.5	1,556	362.6	242

Table 2.1: Characteristics of our Data Set.

needs to recognize the `NtOpenFile` system calls that belong to the process-creation task. When a process is created, the OS executes a set of system calls to allocate resources, load the binary executable, and start the new process: `NtOpenFile`, `NtCreateSection` with desired executable access, and `NtCreateThread`. Consequently, we automatically identify occurrences of this pattern and extract the executable path and file name.

2.2.3 Experimental Data Set

We deployed our data collection framework on ten different Windows machines, each belonging to a different user. The users had different levels of computing expertise and different computer usage patterns. Based on their role, the machines can be classified as follows: two were development systems, one was an office system, one was a production system, four were home PCs, and one was a computer-lab machine.

Overall we collected 114.5 GB of data, consisting of 1.556 billion of system calls, from 362,600 processes and 242 distinct applications. In table 2.1 provides detailed information for each machine.

Our system collected data from each machine at an average rate of 8.2 MB/minute, with highly used machines producing logs at 40 MB/minute and idle machines producing 1.5 MB/minute. In table 2.2, we report the logging time for the ten different machines. For each machine, we show the machine's usage profile, the size of data collected, the total time during which data was actually collected, the time period between the first

Machine	*Usage*	*Data*	*Time*		*Data rate*
			Logged	*Total*	
		(GB)	(hours)	(days)	(MB/minute)
1	office	18.0	12	3	8
2	home	4.5	4	3	6.25
3	home	5.6	3	4	7.77
4	prod.	32.0	12	3	14
5	prod.	34.0	12	3	15
6	lab	14.0	8	3	11
7	home	1.3	3	2	4
8	home	1.2	3	2	4
9	dev.	1.6	2	2	6
10	dev.	2.3	2	3	6.4

Table 2.2: Data Rates During Collection.

log entry and the last log entry, and the average data rate. For example, the fourth row indicates that machine 4 was a production server that generated 32 GB of system call logs, over a period of 3 days, during which data collection was active for 12 hours.

To test the detection rate of our approach, we introduce an additional data set that includes the execution of 10,838 malware samples. These samples were randomly chosen from the submissions received by the Anubis malware analysis service, and are a representative mix of current malware programs such as worms, bots, and file infectors.

2.3 System-Centric Models and Detection

Several studies [28, 12] have shown that models based on system call sequences (n-grams) have difficulties in distinguishing normal and malicious behaviors. One of the main problem is that while n-grams might capture well the execution of individual programs, they poorly generalize to other applications. The reason is that the model is closely tied to the execution(s) of particular applications; we refer to this as a program-centric detection approach.

In this section, we propose a model that attempts to abstract from individual program runs and that generalizes how, in general, benign programs interact with the operating system. For capturing these interactions, we focus on the file system and the registry activity of Microsoft Windows processes. More precisely, we record the files and the registry entries that Windows processes read, write, and execute (in case of files only).

Our model is based on a large number of runs of a diverse set of applications, and it combines the observations into a single model that reflects the activities of *all* programs that are observed. For this to work, we leverage the fact that we see "convergence." That is, even when we build a model from a subset of the observed processes, the activity of the remaining processes fits this model very well. Thus, by looking at program activity from a system-centric view – that is, by analyzing how benign programs interact with the OS – we can build a model that captures well the activity of these programs. Of course, this would not be sufficient by itself. To be useful, our model must also be able to identify a reasonably large fraction of malware. To demonstrate that this is indeed the case, we have performed a number of experiments that are described in more detail in Section 2.5.

2.3.1 Creating Access Activity Models

To capture normal (benign) interactions with the file system and the Windows registry, we propose the creation of *access activity models*. An access activity model specifies a set of labels for operating system resources. In our case, the OS resources are directories in the file system and sub-keys in the registry (sub-keys are the equivalent of directories in the file system). For simplicity, in the following we refer to directories and sub-keys as "folders".

Note that we do not specify labels directly on files or registry entries. The reason for this was that the resulting models are significantly smaller when looking at folders only. As a result, the model generation process is faster and "converges" quicker (i.e., less training data is required to build stable models). Moreover, in almost all cases, the labels for the folder entries (files or registry keys) would be similar to the label for that folder itself. Thus, the sacrifice in precision was minimal.

A label L is a set of access tokens $\{t_0, t_1, \ldots, t_n\}$. Each token t is a pair $\langle a, op \rangle$. The first component a represents the application that has performed the access, the second component op represents the operation itself (that is, the type of access).

In our current system, we refer to applications by name. In principle, this could be exploited by a malware process that decides to reuse the name of an existing application (that has certain privileges). In the future, we could replace application names by names that include the full path, the hash of the code that is being executed, or any other mechanism that allows us to determine the identity of the application that a process belongs to. In addition to specific application names, we use the star character (`*`) as a wildcard to match any application.

The possible values for the operation component of an access token are `read`, `write`, and `execute` for file-system resources (directories), and `read` and `write` for registry sub-keys.

2.3.2 Initial access activity model.

An initial access activity model precisely reflects all resource accesses that appear in the system-call traces of all benign processes that we monitored (we call this data set the training data). Note that for this, we merge accesses to resources that are found in different traces and even on different Windows installations. In other words, we build a "virtual" file system and registry that contains the union of the resources accessed in all traces.

Whenever an application `proc` opens or reads from an existing file `foo` in directory `C:\path\dir`, we insert the directory `dir` into our "virtual" file system, including all directories on the `path` to `dir`. When a prefix of the directories along `path` already exist in our virtual file system, then these directories are re-used. All directories that are not already present (including `dir`) are added to the virtual file system tree. Then, we add the access token $\langle$`proc`, `read`$\rangle$ to the label associated with `dir`.

When a process creates or deletes a file in a directory `dir`, or when it writes to a file, then we use the operation `write` for the access token. Similar considerations apply for read and write operations that are performed on the registry. Finally, whenever a binary is executed (loaded by the OS loader), then we add a token with `execute` to the directory that stores this binary.

For example, consider that file `C:\dir\foo` is read by `pA` on machine A, and that file `C:\dir\sub\bar` is written by `pB` on another machine B. Then, the resulting virtual file system tree would have `C:\` as its root node. From there, we have a link to the directory `dir`, which in turn has a link to `sub`. The label associated with `dir` is $\{\langle$`pA`, `read`$\rangle\}$, and the label associated with `sub` is $\{\langle$`pB`, `write`$\rangle\}$.

2.3.3 Pre-processing.

Before the model generation can proceed, there are two additional pre-processing steps that are necessary. First, we need to remove a small set of benign processes that either read or execute files in many folders. The problem is that these applications appear in many labels and could lead to an access activity model that is less tight (restrictive) than desirable. We found that such applications fall into three categories: Microsoft Windows services (such as Windows Explorer or the command shell) that are used to browse the file system and launch applications; desktop indexing programs; and anti-virus software. The number of different applications that belong to these categories is likely small enough so that a manually-created white list could cover them. In our system, we remove all applications that read or execute files in more than ten percent of the directories. We found a total of 15 applications that fit this profile: nine Windows core services,

two desktop indexing applications, and six anti-virus (AV) programs. Identifying such applications automatically is reasonable, because we assume that our training data does not contain malicious code. However, the number of white-listed applications is so small that the entries can be easily verified manually.

The second pre-processing step is needed to identify applications that start processes with different names. We consider that two processes with different names belong to the same application when their executable are located in the same directory. We have found 14 applications that start multiple processes with different names. These include well-known applications such as MS Office, Messenger, Skype, and RealPlayer. Of course, all Windows programs that are located in `C:\Windows\system32` are also aggregated (into a single meta-application that we refer to as `win_core`). Merging processes that have different names but that ultimately belong to the same application is useful to create tighter access activity models.

2.3.4 Model generalization.

Based on the initial access activity model, we perform a generalization step. This is needed because we clearly cannot assume that the training data contains all possible programs that can be installed on a Windows system, nor do we want to assume that we see all possible resource accesses of the applications that we observed. Also, the initial model does not contain labels for all folders (recall that the access is only recorded for the folder that contains the accessed entity).

The generalization step performs a post-order traversal of both the virtual file system tree and the virtual registry tree. Whenever the algorithm visits a node, it performs the following four steps:

Step 1: First, the algorithm checks the children of the current node to determine whether access tokens can be *propagated* upward in the tree. Intuitively, the idea is that whenever we inspect a folder (node) and observe that all its sub-folders are accessed by a single application only, we assume that the current folder also belongs to this application.

More formally, the upward propagation rule works as follows: For each operation `op`, we examine the labels of all child nodes and extract the access tokes that are related to `op`. This yields a set of access tokens $\{t_1, \ldots, t_n\}$. We then inspect the applications involved in these accesses (i.e., the first component of each token t_i). When we find that all accesses were performed by a *single* application `proc`, we add the access token $\langle \texttt{proc}, \texttt{op} \rangle$ to the current label.

Step 2: The upward propagation rule of Step 1 is used to identify parts of the file system or the registry that belong to a single application. However, this is problematic when considering *container* folders. A container is typically a directory that holds many

"private" folders of different applications. A private folder is a folder that is accessed by a single application only (including all its sub-folders). A well-known example of a container is the directory `C:\Program Files`, which stores the directories of many Windows programs.

Since a container holds folders owned by many different applications, its label would deny access to all sub-folders that were not seen during training. This might be more restrictive than necessary. In particular, we would like to ensure that whenever an application accesses a previously-unseen folder in a container, this should be allowed. Intuitively, the reason is that this access follows an expected "pattern," but the specific folder has not been seen during training. To handle these cases, we introduce a special flag that can be set to mark a folder as a container.

The following rule is used to mark a folder as a container: Similar to before, we examine the labels of all child nodes and extract the access tokes that are related to each operation `op`. We then inspect the set of access tokens that is extracted $\{t_1, \ldots, t_n\}$. When the applications in these accesses are different, but there is *no wildcard* present in any access token, then the folder is marked as container. We explain the implications of a *container flag* for detection in Section 2.3.5.

Step 3: Next, the access tokens in the label associated with the current node are *merged*. To this end, the algorithm first finds all access tokens that share the same operation `op` (second component). Then, it checks their application names (first components). When all tokens share the same application name, they are all identical, and we keep a single copy. When the application names are different, or one token contains the wildcard, then the tokens are replaced by a single token in the form $\{\langle *, \texttt{op}\rangle\}$. Merging is useful to generalize cases in which we have seen multiple applications that perform identical operations in a particular folder, and we assume that other applications (which we have not seen) are also permitted similar access.

Step 4: Finally, the algorithm adds access tokens that were likely missed because of the fact that the training data is not complete. More precisely, for each access token that is related to a *write* operation, we check whether there exists a corresponding *read* token. That is, for all applications that have written to a folder, we check whether they have also performed read operations. If no such token can be found, we add it to the label. The rationale for this step is that an application that can write to resources in a folder can very likely also perform read operations. While it is possible to configure files and directories for write-only access, this is very rare. On the other hand, adding read tokens allows us to avoid false positives in the more frequent case where we have simply not seen (legitimate) read operations in the training data.

When the generalization algorithm completes, all nodes in the virtual file system and

the registry tree have a (possibly empty) label associated with them.

Note that, for building the access activity model, we do not require any knowledge about malicious processes. That is, the model is solely built from generalizing observed, good behavior.

2.3.5 Model Enforcement and Detection

Once an access activity model M is built, we can deploy it in a detector. More precisely, a detector can use M to check processes that attempt to read, write, or execute files in directories or that read or write keys from the registry.

The basic detection algorithm is simple. Assume that an application `proc` attempts to perform operation `op` on resource `r` located in `\path\dir`. We first find the longest prefix P shared between the path to the resource (i.e., `\path\dir`) and the folders in the virtual tree stored by M. For example, when the virtual file system tree contains the directory `C:\dir\sub\foo` and the accessed resource is located in `C:\dir\sub\bar`, the longest common prefix P would be `C:\dir\sub`. We then retrieve the label L_P associated with this prefix and check for all access tokens that are related to operation `op` (actually, after generalization, there will be at most one such token, or none). When no token is found, the model raises an alert. When a token is found, its first component is compared with `proc`. When the application names match or when the first component is `*`, the access succeeds. Otherwise, an alert is raised.

The situation is slightly more complicated when the folder that corresponds to the prefix P is marked as *container*. In this case, we have the situation that a process accesses a sub-folder of a container that was not present in the training data. For example, this could be a program installed under `C:\Program Files` that was not seen during training. In this case, the access is *permitted*. Moreover, the model is dynamically extended with the full path to the resource, and all new folders receive labels that indicate that application `proc` is its owner. More precisely, we add to each label access tokens in form of $\langle$`proc`, `op`$\rangle$ for all operations. This ensures that from now on, no other process can access these newly "discovered" folders. This makes sense, because it reflects the semantics of a container (which is a folder that stores sub-folders that are only accessed by their respective owners).

Whenever an alert is raised, we have several options. It is possible to simply log the event, deny that particular access, or terminate the offending process.

2.4 Hypervisor Framework Design

In this section we present the design and of a hypervisor based detector that implements the system centric technique presented in the previous section.

Our enforcement model exploits hardware virtualization support available in commodity x86 CPU [4, 34]. Leveraging hardware-assisted virtualization technology, we design a tamper-resistant and efficient detector that is able to take over the OS operations and verify the policies derived from the AccessMiner system.

2.4.1 Technology Overview

Before presenting the details of our detector implementation, we provide a brief introduction on Intel Virtualization Technology (VT-x) [34].

The main characteristic of Intel VT-x is the support for two new VMX modes of operation. When VMX is enabled, the processor can be either in *VMX root mode* or in *VMX non-root* mode. The behavior of the processor in VMX root mode is similar to classic protected mode, except for the availability of a new set of instructions, called VMX instructions. Non-root mode is, instead, limited, even when the CPU is running in ring 0. Thanks to this, the virtual machine monitor (VMM) can inspect and intercept operations on critical resources without modifying the code of the guest OS (i.e., the *virtualized* OS). Moreover, because non-root mode operation supports all four IA-32 privilege levels, guest software can run in the original ring it was designed for.

A processor which has been turned on in normal mode can be switched to VMX root operation by executing a `vmxon` operation. The VMM running in root mode sets up the environment and initiates the virtual machine by executing the `vmlaunch` instruction.

Intel VT-x technology defines a data structure called virtual machine control structure (VMCS), that embeds all the information and the configuration needed to capture the state of the virtual machine, or resume its execution. The various control fields determine the conditions under which control leaves the virtual machine (VM exit) and returns to the VMM, and define the actions that need to be performed during VM entry and VM exit operations.

Various events may cause a VM exit, and can be configured with a very fine precision by the hypervisor (e.g., exceptions, I/O operations). Furthermore, the processor can also exit from the virtual machine *explicitly* by executing a `vmcall` instruction.

2.4.2 Threat Model

The threat model we adopt in our work considers a very powerful attacker who can operate with kernel-level privileges. On the other side, the attacker does not have physical access to the machine and, therefore, cannot perform any hardware-based attack (e.g., a DMA attack [55]) and he cannot tamper with the hypervisor operations. We assume that our hypervisor starts during the boot process of the machine and it is the most privileged hypervisor on the system.

Is it also possible to leverage *late-launching* [34] to load AccessMiner hypervisor after the boot. For this to be feasible, however, we have to relax our threat model a little. Indeed, we must assume that either there is no malware on the machine *before* we launch AccessMiner or that we leverage an integrity checking technique to ensure that the hypervisor is not altered at load time [32, 33]. Despite this requirement, an hot-bootable hypervisor can be quite useful in scenarios in which it is not possible to restart the machine (e.g., when it provides some critical service).

2.4.3 Hypervisor architecture

The Detector system is composed by three components: a *system call interceptor*, a *policy matcher*, and a *process revealer*. The outputs of all the components are combined together to check the policies derived by AccessMiner system. In Figure 2.1, we depict a scheme of the overall architecture.

Detection & Management Mode

Our hypervisor supports two operation modes: Detection and Management mode. The first is responsible for the detection of attacks against the system, while the latter is used to configure the hypervisor from an external machine using a dedicated network protocol. To this end, we insert two software components inside the hypervisor: a network driver and a keyboard handler, designed to intercept a particular keystroke sequence.

Switching from Detection to Management mode is done by pressing a keystroke sequence. The hypervisor can easily intercept every keystroke and, if the sequence is correct, it runs a communication procedure to receive and update its configuration. The communication protocol we implemented supports two types of commands: one to receive policies configuration and one to load the new set of policies and reset the state of the detector. Both commands are identified by a special identification number. After setting up the configuration, the hypervisor automatically switches from Management to Detection mode and it continues to perform its real-time detection task. When running in detection mode, the hypervisor is configured to intercept and handle two privileged

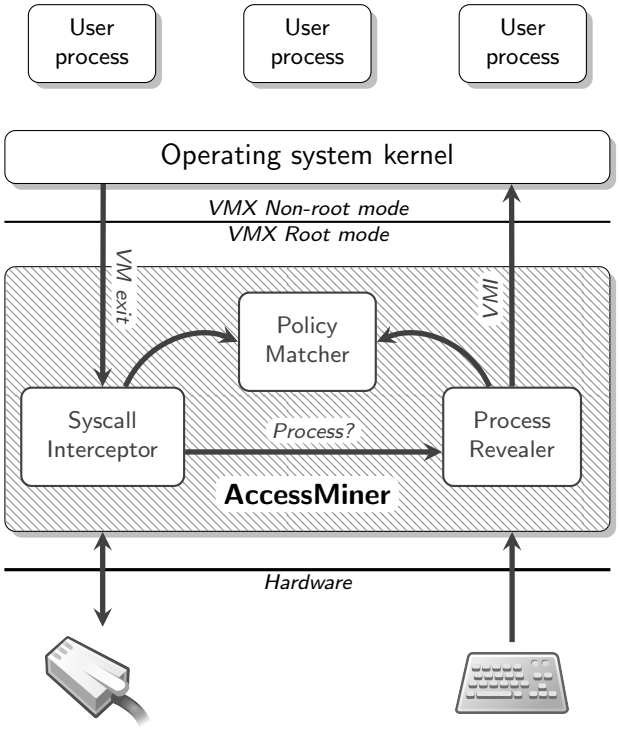


Figure 2.1: Hypervisor Architecture.

instructions: write operations on Control Registers and on alterations to Machine Status Registers (MSR). This allows us to detect context switches among processes and to verify the trusted path of our system call hooking mechanism. More details about the monitoring task are provided in the next section.

System Call Tracer

The core of the system is represented by the System Call Tracer component. Its goal is to intercept the operations performed by the OS, in terms of system call type, parameters and return values. All these information will be used by the Policy Matcher, to verify the right permissions on a certain resources on behalf of the process. There are two main requirements for this component: (R1) The interception mechanism must provide a trusted path between the invocation of the system call and its own termination. In particular the system needs to provide a secure hooking mechanism for intercepting the invocation and termination of OS operation. (R2) The overhead of the interception mechanism must be kept as low as possible.

In order to retrieve all system call information, we need to monitor the invocation of the operation along with its own termination. Whenever a system call is issued by a process, a `sysenter` instruction is invoked. The `sysenter` instruction refers to the SYSENTER_EIP MSR, that contains the address of the system call handler. In order to bring the execution flow inside the hypervisor, we need to switch from VMX non-root mode to VMX root mode. For this reason, we overwrite the SYSENTER_EIP MSR so that it points to a `vmcall` instruction. By using this hooking technique the hypervisor is able to intercept all the system calls performed by the OS and to parse the parameters according to their type. Note that any change of the MSR value on behalf of the system is intercepted and denied by the hypervisor. In this way, the system is able to protect the system call interception mechanism (requirement R1). Furthermore, we hide this change to *in-guest* software, by intercepting read accesses to this register too.

Before passing the information to the Policy Matcher, the system also needs to check whether the operation is successful or not and to collect its return value. For this purpose, our hypervisor is able to intercept a `sysexit` instruction by substituting it with a `vmcall`. Any attempt to re-write the VMX instruction is prevented by the hypervisor through a memory page protection mechanism (requirement R1). To verify the trusted path of the system call, our hypervisor also implements a simple automaton that checks the correctness of the system call execution flow. Every time a `sysenter` is intercepted an opening bracket "(" transition is triggered to indicate which system call was invoked. Every time a `sysexit` is intercepted, the hypervisor verifies that the watchpoint was expected, given the invoked system call. It performs this step repeatedly

until it sees the watchpoint ")", corresponding to the end of the system call request. Any unknown state is reported as a system anomaly. If the operation succeeded, the System Call Tracer invokes the Policy Matcher component and provides all the information on the system call type, parameters, and return value.

Since the hypervisor is intercepting a high number of system calls, the hooking mechanism is a critical component from a performance point of view. Consequently, to improve performances, we devise two modifications to our original implementation. First, the system allocates a protected memory page that contains a short control code and some data about the monitored system calls—such as the system call types and the memory handler code address. Based on the system call type, the code decides whether to invoke an hypercall to switch to monitor mode or to leave the control-flow to the default system call handler. By using this technique we are able to exclude the non-monitored system calls and reduce the overhead of the whole hypervisor system (requirement R2). More details about performance evaluation are reported in the Section 2.6.

Another relevant source of overhead is related to possible multiple repetitions of the same system call from the same process. For example, during a file copy operation, the same read and write operations are repeated multiple times, according to the size of disk blocks and of copied file. Since there is no reason to check the permissions for each operation, our system is designed to verify only the first occurrence of the operation and run the other operations natively. The overhead caused by the repetition of these operations is thus avoided.

This is implemented by introducing a small cache that contains an checksum based on the system call number, its parameters, and the value of the CR3 register of the process who is performing the operation. Every time the system discovers a new operation, we insert it into the cache and when the operation is not likely to be repeated (e.g., the corresponding process terminates, or the file is closed), we flush the cache entry related to that operation. In this way we only check the first operation and we skip possible repetitions (requirement R2). We report a measurement of the effectiveness of our cache in Section 2.6.

Process Revealer

The goal of this component is twofold: First, it extracts and provides the name of the process that is performing the actual operation (i.e., a system call) through Virtual Machine Introspection [48] and, second, it caches this information to reduce the system overhead. The component keeps a cache that allows to lookup the name of the process given a certain CR3 value. The cache is updated every time a process is created or destroyed, by properly intercepting and analyzing process-related system calls.

Policy Checker

The goal of this component is to check AccessMiner policies and to generate an alert in case some of them are violated.

We recognize two main phases for the Policies Checker task: Initialization and Detection phase. The initialization phase is responsible to create the memory structures that will be used for the detection phase. In particular, to check the filesystem and registry policies, we adopt an hash table memory structure where the name of each resource is used as key and the name of process with its own permissions on that resource is stored as value. During the initialization phase, the hypervisor receives the signatures using the ad-hoc network communication protocol we briefly mentioned above. Then, whenever a signature is loaded, the full-pathname of the corresponding resource is extracted and inserted in the memory structure as a key of the hash table. The list of the processes that can get access to the resource along with their own access permission are inserted as elements of such a key.

Another important memory structures used by the policy matcher is the *file/registry handles* structure. Since most of the filesystem and registry system calls operate on handles, while our policy system works with full-pathname resources, the system needs to keep the association between an handle number and the resource full pathname. For this reason, we use a dynamic memory structure that tracks this association. During the monitoring of the system, every time a resource is created or opened, the system retrieves the handle associated to the resource full pathname and it registers it in the structure. Afterwards, when a system call operates on the same handle, the corresponding object is retrieved from the handle structure. Every time an handle is closed, the system removes it from the handles memory structure.

To protect the policy information loaded during the initialization phase, the network driver that receives commands is only enabled when the hypervisor is in Management mode—in our prototype, this is triggered by using a special keystroke sequence. On the other hand, to protect the policies from network attacks, a signature scheme between the hypervisor and the management console is provided. In this way, we can assure that no one is able to tamper the hypervisor configuration information, according to our thread model.

During the detection phase, the System Call Tracer invokes the Policy Checker with the relevant system call information. At this point, the Policy Checker, by using the resources full-pathname as a key of the hash table, retrieves the list of the processes along their permissions. It also queries the process Revealer component in order to retrieve the processes name that acts as a subject of the operation. Once all the information are obtained, it scans the list of the processes to search the process name. If the process is

Machine	*Detection rate*	*False positive rate*	*Adjusted detection rate*	*Rates of detected access violations*			*Detection rate (only writes)*
				Read	*Write*	*Execute*	
1	0.656	0.225	0.906	0.000	0.022	0.222	0.864
3	0.657	0.154	0.907	0.000	0.130	0.043	0.902
4	0.657	0.156	0.907	0.024	0.049	0.122	0.902
5	0.657	0.143	0.907	0.024	0.024	0.095	0.902
6	0.635	0.242	0.877	0.014	0.055	0.242	0.868
7	0.657	0.267	0.907	0.020	0.041	0.265	0.901
8	0.657	0.045	0.907	0.000	0.045	0.000	0.902
9	0.657	0.025	0.907	0.000	0.025	0.000	0.902
10	0.657	0.050	0.907	0.000	0.038	0.038	0.902
Average	0.655	0.148	0.904	0.008	0.044	0.137	0.895

Table 2.3: Partial Detection Based on our Filesystem Access Activity Model.

not allowed to perform the operation, the Policy Checker raise an alert and blocks the operation. Otherwise, it permits the operation and then returns to non-root mode.

2.5 Detection Results

In this section, we evaluate the effectiveness of our system in detecting malicious activities on real systems.

More precisely, we conducted ten experiments. For each one, we selected one of the machines and we used the system calls recorded on the other nine hosts to generate the access activity model, as described in Section 2.3. Finally, we used this model for detection by checking the resource accesses performed by all processes on the machine that was *not* used for model generation. Then, we examine the accesses performed by the malware samples. For each experiment, we evaluate the detection capabilities and false positives of the file system model alone, the registry model alone, and both models combined.

When computing false positives, we used the number of different applications as a basis, and not the number of processes. That is, when the detector raises alerts for ten processes, but all these processes were running the same application, we count this as a single false positive. The reasons for this decision are the following: First, we did not want to bias the results to the popularity of a certain application and the number of times that it appears in our data set. Second, when a detector raises frequent alerts for the same application, it is easy to white-list this particular program

Machine	*FP rate*	*Final det.rate*
1	0.0	0.864
2	0.0	0.902
3	0.0	0.902
4	0.0	0.902
5	0.0	0.902
6	0.0	0.868
7	0.0	0.901
8	0.0	0.902
9	0.0	0.902
10	0.0	0.902
Average	0.0	0.895

Table 2.4: Final Detection Based on our Filesystem Access Activity Model.

2.5.1 File system access activity model.

On average, the file system access activity model contains about 100 labels. These labels contain tokens that restrict read access to about 70 directories, write access to about 80 directories, and execute access to about 30 directories. The results for the file system model are shown in Table 2.3. In this table, we see a number of different columns for the detection rates and the false positive rates. These are discussed in the following paragraphs.

When using the original model to check all read, write, and execute accesses, we see an average detection rate of 66% for the malware samples (column *Detection rate*) and a false positive rate of almost 15% (column *False positive rate*). Note that the false positive rates are computed on the basis of single applications and not on a process basis.

At first glance, the results appear sobering. However, a closer examination of the result reveals interesting insights. First, we decided to investigate the false negative rate in more detail. When looking at the execution traces of the malware programs, we observed that many samples did not get far in their execution but quickly exited or crashed. Interestingly, a substantial fraction of malicious samples never wrote to the file system or the registry, and they did not open any network connections. It is difficult to confirm that these samples exhibit any malicious activity at all. As a result, we decided to remove from our malware data sets all samples that never perform a write operation or open a network connection. This decreases our malware data set to 7,847 samples that exhibit at least some kind of activity. It also improves our detection rate to more than 90%, as reported in column *Adjusted detection rate* of Table 2.3. For the remainder of this chapter, all reported detection rates are computed based on the adjusted malware

data set.

In the next step, we investigated the false positives in more detail. Table 2.3 shows the access violations for each machine, divided into violations due to read (column *Read*), write (column *Write*), and execute (*Execute*) access attempts. It can be seen that execute violations account for a significant majority of false positives. However, we also found that they are only marginally important for detection. Thus, for the next experiment, we decided to use only the access tokens that refer to `write` operations. This is justified by the fact that we are most interested in preserving the integrity of the operating system resources. The detection results for the new *write-only* detection approach are presented in column *Detection rate (only writes)* of Table 2.3. As can be seen, the numbers remain high with 89.5% 2.4. This confirms that write access violations are a good indicator for malicious activity. With this approach, the false positives are identical to the write violations, which are shown in column *Write*.

We further examined the reasons for the remaining write violations. It turned out that these violations were due to two root causes. One set of false positives was caused by our own system-call logging component that wrote temporary files directly into the `C:\` directory before sending the data over the network. The second reasons was due to software updates. More precisely, we detected a number of cases in which an application was writing to its folder in `C:\Program Files`. Of course, only this program had read/execute access to that directory. However, we never saw a write access during training, and as a result, the directory was considered read-only. To accommodate for updates, we manually added a rule to the model that would grant write permission to applications that "own" directories in `C:\Program Files`. Moreover, we granted our component write access to `C:`. With more extensive training, both access activities would have very likely been added automatically. The model that incorporated our minor adjustments generated no more false positives, as shown in Table 2.4. However, the detection capabilities of the model remain basically unchanged, as shown in Table 2.4.

2.5.2 Registry access activity model.

In our experiments, the registry access activity model contained in average about 3,000 labels, significantly more than the file-system model. In particular, the labels contained tokens that restrict read access to about 1,600 keys and write access to about 2,800 keys (*execute* is not defined for registry keys).

The results for the registry model are shown in Table 2.5. The columns *Detection rate* and *False positive rate* show the detection rates and the false positive rate, respectively, for the original model. It can be seen that both the detection rate and the false positive rates are lower than for the file system model. We also examined the detection rate and

Machine	*Detection rate*	*False positive rate*	*Det. rate (only writes)*	*FP rate (only writes)*	*Final det. rate*
1	0.567	0.063	0.530	0.063	0.521
2	0.557	0.107	0.540	0.053	0.521
3	0.566	0.179	0.530	0.128	0.062
4	0.557	0.000	0.530	0.000	0.540
5	0.557	0.000	0.530	0.000	0.540
6	0.557	0.015	0.530	0.000	0.540
7	0.597	0.133	0.530	0.000	0.540
8	0.557	0.067	0.530	0.067	0.537
9	0.561	0.100	0.530	0.025	0.521
10	0.557	0.000	0.530	0.000	0.540
Average	0.563	0.066	0.530	0.034	0.486

Table 2.5: Detection Based on our Registry Access Activity Model.

the false positive rate when considering only write operations (columns *Det. rate (only writes)* and *FP rate (only writes)*). Similar to the file system case, the false positive rate drops significantly; there are five runs in which no false positives were reported at all. However, the detection rate remains (relatively) high.

We also examined the cases for which the registry access model raises false positives. We found that all registry write access violations can be attributed to the sub-tree `HKEY_USERS\Software\ Microsoft`. While this is an important part of the registry that contains a number of security settings, we wanted to understand the detection capabilities of a model that permits write access to these keys. To this end, we added a manual rule to allow writes to this sub-tree and re-run the experiments on the malware data set. We see that the model is still effective and achieves a detection rate of over 48% (shown in column *Final det. rate* of Table 2.5) with no false positives. Considering the significantly larger size of the registry models compared to the ones for the file system, we expect that a larger training set would be required to effectively capture legitimate writes to the `Software\Microsoft` sub-tree.

2.5.3 Full access activity model.

For the final experiment, we combined those improved file system and registry models that yielded zero false positives. The combined detection rate improves compared to the file system model alone, but only slightly (between 1% and 2% for all of the ten runs). The average detection improved from 89.5% to 91% (of course, with no false positives).

2.5.4 Discussion.

When focusing on write operations only, our access activity model achieves a good detection rate (more than 90%) with a very low false positive rate. The false positive rate even drops to zero with minor manual adjustments that compensate for deficiencies in the training data, while still retaining its detection capabilities. This suggests that a system-centric approach is suitable for distinguishing between benign and malicious activity, and it handles well even applications not seen previously. This is because most benign applications are written to be good operating system "citizens" that access and manage resources (files and registry entries) in the way that they are supposed to.

Malicious programs frequently violate good behavior, often because their goals inevitably necessitate tampering with system binaries, application programs, and registry settings. Of course, we cannot expect to detect all possible types of malicious activity. In particular, our detection approach will fail to identify malware programs that ignore other applications and the OS (e.g., the malware does not attempt to hide its presence or to gain control of the OS) and that carry out malicious operations only over the network. For these types of malicious code, it will be necessary to include also network-related policies.

2.6 Performance Measurements and Future Work

At the moment we are performing a number of micro and macro benchmarks to demonstrate the efficiency of our system in a real deployment. For our experiments we chose to use the Passmark Performance Test suite [45] in four different test environments: on a physical machine (PM), inside a guest VMWare virtual machine (VM), on physical machine with AccessMiner (AM), and on physical machine running the Hypersight (RTD) [35] real-time rootkit detector.

Preliminary results seem to confirm that Accessminer introduces an acceptable overhead, in line with other virtual machine monitors on the market.

During the next year we plan to conclude our benchmark experiments and also add some fine grain measurements to precisely assess the impact of our tool on different low-level aspects of the operating system.

3 Detecting malicious electronic documents

Malicious electronic documents are a common attack vector in regular IT and backoffice environments. Attackers can easily deliver via email infected electronic documents which contain malicious code that will take over the targeted system, extract relevant (confidential) information and ship that back to the attacker, or turn the system into a bot that executes illegal activities such as spamming or Denial of Service attacks.

While this typical IT scenario is less relevant to ICS environments, due to the fewer backoffice-like activities being performed there and the best practices used for network segmentation, during our interviews with stakeholders we learnt that it is anyway common to exchange user manuals in the form of PDFs, not to mention the project files that are being loaded onto engineering workstations to program PLCs and RTUs. There exist several real-life examples of such potential attack vectors [1].

There is extensive prior art regarding the detection of malicious documents such as PDFs, but little research has been performed in the area of "non-standard" electronic documents, such as engineering project files. In recent years, security researchers have shown how it is possible to divert and take control of an engineering workstation by loading a specifically crafted engineering project file. Thus, it is important to take into account also this attack vector.

On the other hand, most of the approaches developed for detecting "common" malicious electronic documents are based on the assumptions that once the malicious code is executed, it will perform anomalous operations such as, for instance, connecting to an Internet service to download malicious updates. While such action would be possible also in an ICS environment, the threats we wish to counter are not likely going to perform actions other than those required to disrupt their final target, or alter the underlying processes. Hence, current techniques are not said to detect targeted attacks.

In the remaining of the documents we will describe current exploitation techniques and approaches for detecting malicious electronic documents (mainly for PDFs), then we will present our approach and the benchmarks of preliminary tests.

[1] `http://ics-cert.us-cert.gov/alerts/ICS-ALERT-11-283-01` and `https://ics-cert.us-cert.gov/advisories/ICSA-12-102-03`

3.1 General exploitation techniques

In this section we describe current exploitation techniques commonly found in malicious documents.
For a successful exploitation of an application, some conditions must hold. First, the attacker needs to be able to provide some form of input to the application (for example, a document to an electronic document reader, font files, images, data over a network stream, etc.). Second, the application, or any third-party module involved in processing the user input must contain a vulnerability that gives the attacker control over the content of the memory, and eventually control over the execution flow of the application. Once the attacker has control over the memory, she can change it in such a way that the application executes instructions of the attacker's choosing.

3.1.1 Shellcode

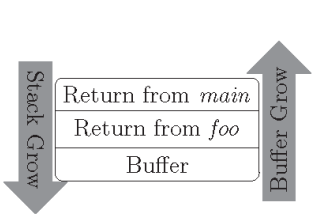
One way of executing malicious instructions is by loading *shellcode* into the application's memory, and diverting the execution flow of the application to this shellcode (Section 3.1.2 and 3.1.2). There may be size restrictions on the memory region that the attacker controls. Therefore, the shellcode is usually a small piece of code that starts a *reverse-shell*[2] (hence the name, *shell*code) or downloads and executes additional components from the Internet, which is referred to as *dropping*.

In some cases, the shellcode can be delivered to the application through normal user input (such as a network stream). It can even be part of the input that triggers the vulnerability to divert execution flow to the shellcode. As application hardening techniques became more commonly implemented over time, loading shellcode has shifted towards *heap spraying* which will be discussed in Section 3.2.1.

3.1.2 Buffer Overflows

An easy to exploit, and prevalent type of vulnerability is a (stack-based) buffer overflow. Buffer overflows occur when the application copies input from the user into a buffer that is not large enough to hold the input. Figure 3.1 shows a simplified x86 stack. A 200 byte buffer is allocated on *foo's* stack frame and user-input is copied into this buffer without respecting the 200 byte limit. Since the buffer grows upwards in the Figure (Buffer[0] is at the bottom, Buffer[199] at the top), the attacker can overwrite the return address that was pushed onto the stack before the buffer ('*Return from foo*') [3]. On

[2]When a reverse-shell is started on a victim's computer, it listens for incoming connections from the attacker. Once connected, the attacker can execute arbitrary commands on the victim's computer




```
void foo(const char* UserInput)
{
    char Buffer[200];
    // Copy until null-terminator
    strcpy(Buffer, UserInput);
}
```

Figure 3.1: Example of a stack-based buffer overflow

x86 architecture, whenever a CALL instruction is executed, the memory address of the instruction following the CALL instruction is pushed onto the stack. Then, when the CPU next executes a '**RET**' instruction, this address, whose value is now controlled by the attacker, is popped from the stack and execution is resumed at this address. Since the attacker controls the stack, execution is resumed at an attacker-controlled memory location, such as previously loaded shellcode.
In a correct, non-vulnerable version of the example, the '*strcpy*' call would need to be replaced with a call to '*strncpy(Buffer, UserInput, 200)*'. This would limit the number of bytes copied into *Buffer* to 200, preventing '*Return from foo*' to be overwritten when *UserInput* is larger than 200 bytes.

One technique to prevent the execution flow from diverting to an attacker controlled location is by performing sanity checks on the return addresses on the stack. This is typically done by placing a 'canary' value below a return address on the stack [7]. Each time the CPU executes a RET instruction, it checks the validity of the canary value before diverting execution flow to the return address. Clearly, the canary value must be unpredictable.

If in the example the buffer would be large enough to contain the shellcode, the attacker could divert control to the same buffer and the shellcode would be executed on the stack. With Data Execution Prevention (DEP)[14], regions in memory where executable code is not expected are marked as Non-Executable. It is uncommon for a typical application to execute instructions that are located on the stack, therefore the stack can be marked as Non-Executable in most typical applications. This does not prevent the attacker from overwriting return addresses on the stack, but she can no longer execute shellcode that is in the same buffer as the one used to overwrite the return address.

DEP may cause compatibility problems with existing applications, it is therefore possible to programmatically disable DEP which will be discussed in Section 3.2.1 on Heap Spraying.

Integer overflow

In some cases, a prerequisite to a buffer overflow is that an integer value overflows. Computers are natively only able to express numbers up to a certain value. For example, on 32-bit systems the maximum value of an unsigned integer is $2^{32}-1$ (or '4.294.967.295'). Integer overflow [17] vulnerabilities are particularly difficult to detect [10] as integer overflows cannot (conveniently) be detected after they have happened.

Integer overflows do not allow for direct modification of the application's memory, or for control over the execution flow. A common case where integer overflows can enable buffer overflow vulnerabilities is when arithmetics cause an overflow and the result is used to determine the buffer size. Figure 3.2 shows an example of how an integer overflow can enable a buffer overflow vulnerability.

```
1  int* copyArray( int* input, int count )
2  {
3      int* result = (int*)malloc( count * sizeof( int ) );
4      if( NULL == result )
5          return NULL;
6      for( int i = 0; i < count; ++i )
7          result[ i ] = input[ i ];
8
9      return result;
10 }
```

Figure 3.2: Buffer overflow with integer overflow prerequisite

If an attacker controls *count*, she directly controls how much memory is allocated for *result*. By choosing a sufficiently large value for *count*, the multiplication '*count * sizeof(int)*' will overflow and become a different, smaller number. Since the overflow is undetected, the call to *malloc* will return successfully, but allocate less memory than expected, and the for loop will write past the end of *result*, resulting in a heap overflow.

Other execution flow diversion techniques

In the section on buffer overflows, the return addresses present on the stack was overwritten to divert the execution flow of the application. This is the most generally applicable technique, but others exist. Consider an artificial application as shown in Figure 3.3. A function pointer, pointing to the *foo* function is placed on the stack before a buffer overflow vulnerability. After input from the command line has been copied to the 200 byte buffer, *foo* is called through the function pointer *pFoo*. Exactly as with overwriting

```
1   int main( int argc, const char** argv )
2   {
3       char Buffer[ 200 ];
4       void (*pFoo)( const char* ) = &foo;
5
6       // Copy until null-terminator
7       strcpy( Buffer, argv[1] );
8
9       pFoo( Buffer );
10      return 0;
11  }
```

Figure 3.3: Overwriting function pointer to divert execution flow

```
1   int main( int argc, const char** argv )
2   {
3       char Buffer[ 200 ];
4       __try {
5           // Copy until null-terminator
6           strcpy( Buffer, argv[1] );
7       } __except( GetExceptionCode() == EXCEPTION_INT_DIVIDE_BY_ZERO ?
8           EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH ) {
9          // Handle exception
10      }
11      return 0;
12  }
```

Figure 3.4: Overwriting SEH-chain entry to divert execution flow

the return address in the previous example, the attacker can now overwrite the value of the *pFoo* pointer, and thus directly control the execution flow of the application. Note that stack canaries will not be effective against this type of vulnerability as the attacker does not need to overwrite the return address. Some compilers implement array variable reordering [18] which would cause '*Buffer*' to be allocated at the top of the stack frame, in which case overflowing '*Buffer*' does not overwrite '*pFoo*'.

A similar but possibly more prevalent method is to overwrite SEH (Structured Exception Handler) pointers in Windows applications. In short, when a Windows application uses exceptions as shown in Figure 3.4, a SEH-chain is pushed onto the stack so that in case of an exception, execution flow can be diverted to a suitable exception handling routine. Again, by overwriting one of the SEH-chain entries with attacker controlled

values she can divert execution flow by triggering an exception (such as divide by zero or illegal instruction) which causes the control flow to divert to the attacker controlled value.

Use-after-free

```
1 A* pA = new A;
2 delete pA; // pA is now a 'dangling pointer'
3
4 B* pB = new B; // sizeof(A) == sizeof(B)
5 // It is likely that pA == pB here
6
7 // Here, B's vtable is used for the function call.
8 pA->virtual_function_call();
```

Figure 3.5: Artificial example of use-after-free attack

Use-after-free vulnerabilities are very application-specific, and difficult to exploit [2, 50]. In general, the structure of the attack is as follows:

1. the application allocates an object 'A' on the heap;
2. the object 'A' is freed, any pointers to 'A' are now considered *dangling*, meaning that they refer to an object that no longer exists;
3. the attacker allocates one or more objects 'B' of the same size as 'A';
4. with a high probability, one of the 'B' objects will be allocated in the previous heap location of 'A';
5. the application calls a virtual function on the 'A' object, whose memory location is currently filled be the attacker-controlled 'B'.

Figure 3.5 shows in pseudo-code what happens in a use-after-free attack. Since the vtable of 'B' is used in the call on line 8, the attacker gains control of the application's control flow under the right conditions.
In object oriented languages such as C++, the vtable is used to determine the memory location of a function at run-time. This run-time, as opposed to compile time dependency allows classes to form a hierarchy, and overload functions of their base classes. We can think of the vtable as a list of 'pFoo' pointers as seen in Figure 3.3. So naturally, if the

attacker control the vtable of an object, she controls the application's execution flow whenever a virtual function of this object is called.

3.2 Return Oriented Programming (ROP)

A technique to circumvent Data Execution Prevention is not to overwrite one return address, but rather to place a chain of return addresses on the stack in such a way that each 'RET' instruction executes a small portion of the malicious code. Since the attacker can not execute code on the stack, she needs to find 'gadgets'[3] in the application's code section and push the addresses of these gadgets on the stack. It has been shown that in a typical application, ROP can be used to execute arbitrary code [42]. When the attacker uses gadgets from the standard-C library, which is present in most applications, this is called a *return-to-libc* attack.

Consider that an attacker wishes to increment the EAX and EBX registers. She needs to find one or more executable locations in memory that effectively perform *inc EAX, inc EBX*, followed by a RET instruction, and place the addresses of these instructions on the stack.

Buffer overflows, ROP, and most offensive techniques in general, require the attacker to be aware of the location of certain objects in memory. Clearly when the attacker overwrites a return address to jump to her shellcode, she needs to know where in memory the shellcode resides. Address Space Layout Randomization (ASLR) [14, 43] is a technique that randomizes an application's address-space layout, either at compile time or at each new execution of the application. ASLR-enabled dynamic libraries will be loaded into memory at unpredictable offsets each time. Shacham et al.[43] show that ASLR can effectively be beaten on 32-bit systems by brute-force. The authors show that any buffer overflow can be modified to beat ASLR in on average 216 seconds of overhead. More importantly, even in modern operating systems, not all sections of the address space are randomized, due to executables that have fixed load addresses [21] or dynamic library incompatibilities with ASLR [15]. Finally, in some cases the base address of a loaded dynamic library can be brute forced [43] or calculated through a leaked pointer [53].

ROP attacks are made more difficult by ASLR because the location of the gadgets will be unpredictable. Though even with a limited number of ASLR-disabled modules, it is still possible to perform part of an attack using the ROP technique, such as disabling Data Execution Prevention.

[3] Small set of instructions ending with a 'RET' instruction. Each 'RET' pops the address of the next gadget from the stack and starts executing it

3.2.1 Heap Spraying

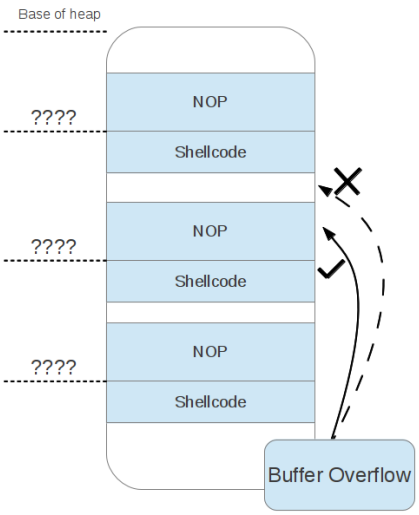


Figure 3.6: Heap littered with shellcode

With ASLR enabled, an attacker has much less certainty about where in memory her shellcode resides. This caused the rise in popularity of using heap spraying techniques. Heap spraying is a technique to litter the heap of an application with instances of the shellcode at predictable address ranges. If an application supports interpreted languages such as Javascript, then these can be used to perform the heap spray 'legitimately', i.e. without exploiting a vulnerability. Otherwise a vulnerability must exist in the application which allows the attacker to allocate large portions of heap space, and fill it with copies of the shellcode.
Once the heap spraying stage is complete, a vulnerability such as a buffer overflow can be used to divert the execution flow to a semi-random location in the application's heap which is now likely to contain the shellcode.

In practice, the shellcode is often prepended with a NOP-slide[4] to increase the likelihood of diverting the execution flow to a location that will eventually lead to execution of the shellcode. Figure 3.6 shows a heap, littered with instances of the shellcode. Somewhere in the code, a buffer overflow or other type of exploit is triggered to divert the

[4]A NOP-slide is a set of instructions that have no effect on the CPU state, e.g. *inc EAX, dec EAX*

execution flow. If a memory region marked other than '*NOP*' is hit, the application may crash.

If the heap is marked as Non-Executable by Data Execution Prevention techniques, a ROP attack is first needed to programmatically disable DEP. Techniques to disable DEP using Return Oriented Programming are described in [44]. While heap sprays in itself are not malicious, they are a crucial part of modern malware that is able to operate on systems that have been hardened using ASLR techniques. In fact, we will see in Section 3.4.1 that all recent examples of Adobe Reader exploits use the Javascript interpreted language to spray shellcode into the heap before triggering any vulnerabilities.

3.3 State of the art

In this section we discuss previous work aimed at detecting malicious electronic documents. Unfortunately, most work is aimed at malicious PDF documents. However, while the attack vector could change (i.e., an engineer project file) the main techniques to diver or take control of a targeted system are the same, therefore it is worth discussing how various attack techniques have been countered.

3.3.1 Shellcode

Shellcode detection has been a much discussed topic in the literature. Ideally one would like to scan an application's input such as a PDF document or a network stream and be able to detect shellcode without having to execute anything. This is known as static detection.

The early forms of static detection techniques were signature based, where input is simply scanned for sequences of bytes, called signatures, used previously in known attacks. If the signatures are carefully constructed to uniquely identify certain malware, the false positive rate is low which makes them preferred in Intrusion Prevention Systems, where false positives mean disruption to honest users of the system. Clearly, signature based systems require constant updates when new attacks are found in the wild and thus result in large signature databases.

Toth and Kruegel propose static techniques to search for common shellcode patterns such as NOP-slides [49] or structural similarities between different worm instances [27]. These methods have similar advantages as signature based systems, but due to their static nature they are easily evaded. For example, shellcode that modifies itself during execution [38] will evade detection as the static method is only able to scan the initial form of the shellcode, it is unable to scan the modified, malicious form of the shell-

	Positive	**Negative**
Static	Speed Undetectable Safe (nothing is executed) All code available, independent of whether it is actually executed often	Prone to obfuscation Cannot scan run-time dependent input
Dynamic	Resistant to obfuscation Run-time data available	Detectable May miss rarely executed parts of code Prone to time outs

Table 3.1: Comparison of dynamic and static detection approaches.

code. Furthermore, the shellcode can use indirect jumps to avoid detection, as the jump location depends on data that is only available at run-time [38].

Dynamic techniques attempt to disassemble input into valid CPU instructions, and execute them on an (often emulated) CPU. These techniques are more robust against shellcode whose execution flow depends on run-time data and self-modifying shellcode, as eventually the underlying malicious shellcode will be executed and analyzed. A problem with dynamic analysis is that if the shellcode is executed on an emulated CPU, the shellcode may detect imperfections in the emulation and stop execution, or it may unintentionally stop functioning due to the imperfections. Also, in targeted attacks the attacker may verify presence of specific characteristics of its target, such as a specific user name, before showing malicious behavior. Since dynamic techniques attempt to emulate the shellcode, and only a limited time frame is available for this analysis, an attacker can evade detection by inserting long running loops to reach the detector's execution threshold before executing the malicious shellcode. Finally, emulation based detection techniques may lack a view of the system's state such as the complete address space of the target application, or CPU registers. The implications can be limited by instrumenting the emulator with commonly used system libraries which the shellcode may use, resulting in an emulator that more closely resembles a real machine.

Polychronakis et al. propose a dynamic technique for network-level detection of self-decrypting polymorphic shellcode [38]. Self-decrypting polymorphic shellcode is shell-

code in encrypted form, prepended with a decryption routine to make it self-decrypting. The technique combines two heuristics to detect the decryption routine of the shellcode. First, for the decryption routine to work it must obtain the current absolute memory address of the shellcode, this is referred to as *Get Program Counter*, or *GetPC*. Three common techniques of GetPC are described in [38]. Secondly, the decryption routine will perform many read operations in the small memory region where the encrypted shellcode resides. Polychronakis et al. combine these two heuristics to effectively detect polymorphic shellcode. Since the detection technique focuses on detecting the decryption routine present in polymorphic shellcode, it does not detect plain or metamorphic[5] shellcode.

Polychronakis et al. improve on this in [39] by proposing a general purpose dynamic shellcode detection technique based on heuristics. The heuristics attempt to detect common shellcode behavior such as kernel32.dll base address resolution, and SEH-based GetPC code.

3.3.2 Use-after-free

Use-after-free vulnerabilities are difficult to detect efficiently. One way is to track whether a pointer is 'dangling' (referring to a freed object). This results in a runtime cost at every memory access and is generally unacceptable. More recently research has shifted towards more secure memory allocation algorithms [2, 31, 20] with promising results.

3.3.3 Return Oriented Programming

Detection of Return Oriented Programming attacks have received less attention in literature, but the number of attacks using ROP is expected to grow with increasing adoption of Data Execution Prevention techniques. Currently, ROP attacks as seen in the wild are known only to have been used to disable memory protection such as DEP, and to divert control to traditional shellcode [40]. This means that the attack as a whole could still be detected if the ROP stage goes unnoticed. However, it has been shown that ROP can be used to execute arbitrary code in a typical application [42]. Such a ROP-only attack would go unnoticed by shellcode detectors, as there is no shellcode.

ROP attacks can be stopped in two ways, either by reducing the number of available gadgets in an application or by scanning memory buffers that hold user input for sequences of gadget addresses. To reduce the number of available gadgets, compiler extensions [30, 36] have been proposed that specifically limit the number of usable gadgets

[5] Metamorphic shellcode is modified before execution, and thus does not contain a decryption routine

emitted by the compiler. While this is an effective technique, quick and widespread adoption is inhibited by the need to recompile existing applications. Secondly, run-time solutions as proposed in [13, 16] incur significant run-time overhead which limits their adoption. Finally, Vasilis Pappes et al [37] introduce a method to reduce the number of available gadgets on existing binaries using in-place code randomization. As opposed to other methods [1, 9, 26] that work on existing binaries, the in place code randomization does not require debugging symbols to be available. In fact, debugging symbols are typically not available in commercial software and thus would have similar drawbacks as the compiler extensions.

A second technique, dubbed **ROPScan** [40] by Polychronakis and Angelos takes a similar approach to dynamic shellcode detectors. ROPScan initializes a CPU emulator with a snapshot of the virtual memory of the application it aims to protect. Next, the input is scanned for valid addresses within the process' virtual memory. Existence of valid addresses in an input alone is not an indication of a ROP payload, as by chance data may contain valid addresses. Therefore, when a valid address is found ROPScan attempts to start execution at that address. If the execution flow resembles ROP behavior (chain of short sequences of instructions followed by 'RET'), ROPScan marks it as malicious.

3.3.4 Heap Spraying

Ratanaworabhan et al. introduce **Nozzle** [41]. Nozzle hooks into the family of memory allocation functions such as *malloc*. It keeps track of all currently allocated memory blocks, and periodically scans these blocks for valid x86 instructions. Since arbitrary data may contain many valid x86 instructions, these instructions are first analyzed to determine whether they look like a NOP slide. This is done by constructing execution flow graphs from arbitrary starting points. It is considered a NOP slide if many execution flows reach the same basic block, which may indicate the start of the shellcode. Figure 3.7 illustrates this.

When an application makes many heap allocations, the overhead of Nozzle is significant, making it impossible to deploy at an end user's machine.

3.3.5 Other general detection techniques

Besides techniques designed specifically to detect one or a small subset of exploitation techniques, more general techniques exist. A technique quite different to the previously discussed is called *automated dynamic malware analysis*. Here, the goal is to detect the observable effects of a malware infection, regardless of the exploitation technique used. Tools implementing this technique either use a fully emulated environment [8, 23] or

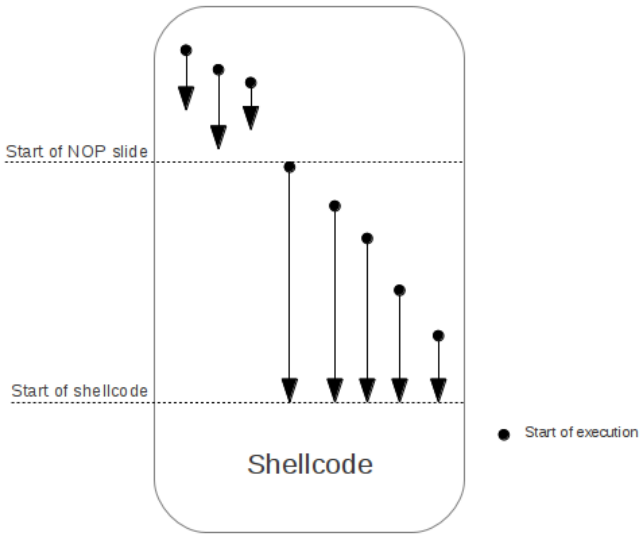


Figure 3.7: Memory block being scanned by Nozzle

instrument an environment [54] in such a way to monitor security-related events of the sample under analysis. Security-related events include *file system* and *network activity*, *registry modifications* and *process creation*.

Advantage of these tools are that no specific exploitation technique is being targeted, thus previously unknown malware can be detected as long as it triggers some maliciously looking set of security events.
The listed security events are perfectly legitimate and performed by many legitimate applications, so often human interaction is required to determine whether the sample is malicious or not. Consider an FTP server application. Under normal circumstances it can read, create and write files, spawn sub-processes to delegate part of the work load, etc. An attacker may want to do the same things when she exploits a vulnerability in the FTP server, the only differences is that she does it without the proper credentials. In the case of electronic document readers, this is less of a problem. It is unusual for a document reader such as Adobe Reader to download and execute a binary file, thus observing this pattern indicates that the sample is malicious with high probability.

Systems like CWSandbox [54] use API hooking to log the security events, which is easily detected by the malware, allowing it to terminate or perform only benign actions in the presence of CWSandbox. TTAnalyze [8] improves on this by completely emulating a

PC in software making detection by the malware more difficult but still possible through exploiting timing differences between a real system and the emulated PC.

Techniques similar to malware analysis can be used for prevention. As noted, it is unusual for document readers to download and execute binary files and as of such, this action can be prohibited without the explicit consent of the user, to stop an infection. The process of explicitly allowing what certain applications can do, as opposed to explicitly disallowing, is called **white listing**.

3.4 Proposed detection technique

In this section we derive our novel detection technique, and present a case study of Javascript vulnerabilities in Adobe Reader.

At the root of most classes of vulnerabilities lies improper or missing validation of user input. This allows the attacker to control elements of the application that she is not meant to control, such as the number of bytes to be copied into a buffer and indirectly the return addresses resident on the stack, or pointers to (exception handling) routines. To exploit a vulnerability in the implementation of a function, malware has to supply a parameter that does not comply to the "function specifications", and this non-compliance is usually readily observable if the malware sample is compared to examples of legitimate use of the same function.
Automated tools exist [25, 29, 51] to find vulnerabilities in source code, such as invalid or missing input validation. However, these tools are cumbersome to use either because of a high false positive rate, or because they require additional annotations [29] from the developer to assist in the vulnerability analysis. Additionally, static analysis tools lack run-time context that a vulnerability may depend on, while dynamic tools may miss vulnerabilities residing in rarely executed parts of the application, where vulnerabilities are likely to reside.

It is clear that it is not sufficient to simply educate developers of potential security problems, the fact is that buffer overflow vulnerabilities still exist, even though it has been over one and a half decade since Aleph One's infamous paper '*Smashing the stack for fun and profit*' [3]. Even if an application is built from the ground up with security in mind, it is only natural that rarely used routines receive less attention from developers compared to commonly used functionality.

3.4.1 Case: Adobe Reader's Javascript interpreter

Adobe Reader[6] is an application to parse and display *Portable Document Format*(PDF) documents. The PDF specification is an open standard, therefore Adobe Reader is not the only application available to parse and display PDF documents. Other applications include *Foxit Reader*[7], *Evince*[8], Preview for Mac users [9], and many more. According to Avast! in July 2011, Adobe Reader was used by over 80% of its user base, compared to 4.8% of the users using the second most popular Foxit Reader [6]. We chose to analyze Adobe Reader because of its popularity, especially in corporate environments, and also because of the wide availability of exploit information.

The PDF specification (ISO 32000-1) [24] is an extremely complex document. The specification for the basic PDF functionality counts over 2500 pages [56]. This includes only the ISO specification, the Javascript API reference for version 8.1 and the XML Forms architecture specification. Not included are the 3D Annotation specification, XMP specification nor any of the font specifications. Many vulnerabilities have been found in the parsing logic of PDF readers, but these vulnerabilities will not be the focus of this discussion. Instead, we look at vulnerabilities involving Javascript in PDF documents.

The PDF specification allows for the use of Javascript to provide dynamic content in PDF documents, such as automated form validation. Javascript is a dynamic, weakly typed scripting language commonly used to provide dynamic content on web sites.

PDFs that contain malicious Javascript attempt to exploit vulnerabilities in the Javascript interpreter embedded in the PDF reader. In this section, we will iterate known Adobe Reader exploits that exploit vulnerabilities in the Javascript interpreter. Table 3.2 provides an overview of the most prominent vulnerabilities available today. All the listed vulnerabilities are stack-based buffer overflows, except the *media.newPlayer* vulnerability, which is a *use-after-free* vulnerability. The column *Anomaly* refers to what is unusual in a malicious call to the vulnerable function compared to a benign call. For example, an overly large buffer passed to *'Collab.getIcon'* to trigger a buffer overflow.

3.5 Approach

In this section we describe our proposed approach to detect malicious electronic documents. While we will refer mainly to malicious PDF documents, the approach can

[6] `http://www.adobe.com/products/reader.html`
[7] `http://www.foxitsoftware.com/Secure_PDF_Reader/`
[8] `http://projects.gnome.org/evince/`
[9] `http://support.apple.com/kb/HT2506`

CVE	Vulnerable Function	Anomaly	Samples
CVE-2009-0927	Collab.getIcon	Unusually large string argument '*cName*'	85
CVE-2009-1492	doc.getAnnots	Large negative integer arguments	10
CVE-2009-1493	spell.customDictionaryOpen	Unusually large string argument '*cName*'	0 (linux only)
CVE-2009-4324	media.newPlayer	NULL argument	6
CVE-2008-2992	util.printf	Unusually large floating-point argument	10
CVE-2007-5659	Collab.collectEmailInfo	Unusually large string argument '*msg*'	150

Table 3.2: Existing vulnerabilities inside Adobe Reader

be applied to other electronic documents including project engineering files, Microsoft Office Word and Excel documents.

On a high level, our approach detects anomalies in the malicious use of functions, compared to their benign use. To do this, we model the characteristics of a benign function call and compare these characteristics to future uses of the function. In the first of two phases, we process samples that are known or expected to be malware-free (benign samples). From these benign samples, we create a model of the function identifier and each function parameter, we call this the detection model. The approach is deterministic, meaning that for a given input the resulting model is always the same.

In the second phase we feed with samples that are unknown to be benign or malicious. We compare function identifiers and the function parameter characteristics from the unknown sample with the previously obtained benign detection model. Intuitively, the approach is based on self-learning white listing, as originally introduced in [11, 52], then used for network intrusion detection.
Advantages of this approach include:

Universal This approach can be applied to any kind of electronic document.

Lightweight The system does not require emulation, OS-based instrumentation or other computationally expensive analysis. Thanks to its minimal footprint, users can

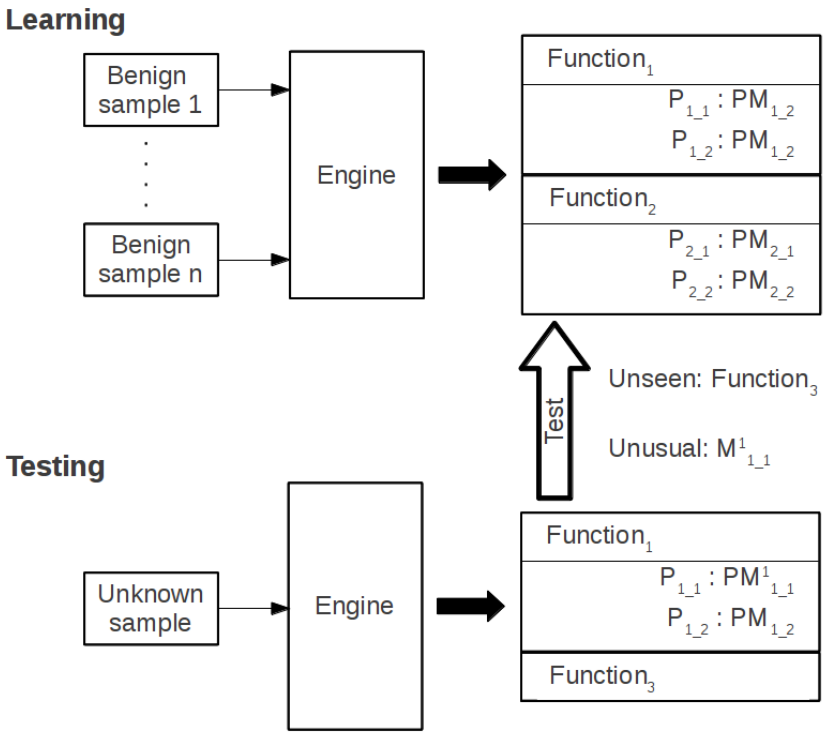


Figure 3.8: Phases of the proposed technique

run it on their systems without observing any significant degradation in performance. The only overhead is due to the interposition with actual functions for monitoring.

Amenable to privacy preserving collaborative malware detection The underlying detection model is deterministic, meaning that the outcome is easily predictable and that viewing a legitimate sample for the second time does not cause any modification in the detection model (notice that this would not be the case if we had used a machine learning technique like a neural networks). These characteristics allow the merging of legitimate learning samples, thereby decreasing the false positive rate. The merging of samples can easily be done in a privacy preserving way, as we only need to exchange function codes

together with an abstraction of their parameters (length, observed ranges, etc.). This allows detection models to be shared between different organizations.

Modular Unlike typical anomaly detection approaches, the learning and the detection phase do not have to be disjoint, this allows for a modular learning phase, in which the detection model is improved each time a new legitimate sample is detected. This can be done without stopping the detection phase. A collaborative detection model, built by multiple trusted clients could be envisioned.

With our approach, we flag a sample as malicious in two cases. Either the unknown sample contains a function identifier that is not present in the detection model, or a function is called with arguments that have unusual characteristics compared to the model.

Figure 3.8 shows the learning and testing phases visually. A model is built from a certain number of benign samples containing function calls to $Function_1(P_{1\_1}, P_{1\_2})$ and $Function_2(P_{2\_1}, P_{2\_2})$. A parameter model PM_{x_y} is associated with each parameter P_{x_y}, modeling the parameter characteristics (described below). In the testing phase, the model from the unknown sample is compared to the detection model. In this case there are two anomalies: $Function_3$ is not present in the detection model, and the parameter model $PM^1_{1\_1}$ is unusual compared to the parameter detection model. Thus the unknown sample is flagged as malicious.

In a real situation with a sufficiently large number of benign training samples this would indicate that the sample flagged as malicious is "*out of the ordinary*". A previously unseen function call could mean a call to an undocumented function that is normally never called, or a call to a rarely used documented function such as *media.newPlayer*. Differences in the function parameter characteristics indicate an "*out of the ordinary*" use of a *known* function, such as an excessively large buffer passed to *collab.getIcon*.

What follows is a description of the function parameter characteristics that we record in the model.
From the discussion on general exploitation techniques the following function parameter characteristics follow:

- Parameter length (string or buffer length, array size, ...)
- Numeric value (integer, floating point values, ...)
- String characteristics (string encoding, printable vs non-printable characters, ...)

```
// Function declarations
function Function1( arg1, arg2, arg3 );
function Function2( arg1 );

// Learning function calls
Function1( "first argument", "second argument", 3 );
Function1( ""              , "second argument", 10 );

Function2( 1024 );
Function2( 1 );

// Resulting Model
Function1 (count: 2)
{
    arg1     (string)    [0-14]
    arg2     (string)    [15-15]
    arg3     (integer)   [3-10]
}
Function2 (count: 2)
{
    arg1     (integer)   [1-1024]
}

// Example anomalous function calls:

// Parameter length 'arg2' out of range
// 0 outside of [15-15]
Function1( "first argument", "", 3 );

// Numeric value 'arg1' out of range
// 2147483647 outside of [1-1024]
Function2( 2147483647 );
```

Figure 3.9: Example functions calls, and the resulting model

Parameter length Often, an unusually large buffer is passed to a function to trigger the exploit, as is the case with the *collab.getIcon* vulnerability shown in Figure **??**. A buffer of roughly 24 kilobytes is being passed containing garbage data. *collab.getIcon* is used to retrieve an icon present in the document, based on its name. It is unreasonable that under normal circumstances, the same function will be called with such a large parameter. This would mean that the document contains an icon identified by a name of over 24000 characters.

All the listed malicious Javascript samples use heap spraying to litter the memory with instances of the shellcode, but it is not uncommon to carry the shellcode in the same buffer that is used to trigger the exploit, especially in network-level exploits [5]. Since shellcode is often larger in size than typical string parameters, such as names, this could be detected in the same way. The above is even more true for ROP payloads as these payloads can not be heap sprayed and many gadget addresses may be needed to create functioning shellcode making an unusually large buffer inevitable.

In the learning phase, we record the minimum and maximum parameter length passed to a function. In testing phase, the parameter length is compared to the benign detection model.

Numeric value The *util.printf* vulnerability shows the importance of recording the range of numeric values of parameters. The vulnerability is triggered by attempting to parse a floating point number and passing an excessively large integer ($\sim 12 \times 10^{294}$). Similarly, in the *getAnnots* vulnerability, large negative numbers are used to trigger the exploit. Again, it is clear that both large floating point numbers, and large negative numbers are not commonly found when using these functions.

In the learning phase, we record the minimum and maximum value of numeric parameters passed to a function. In testing phase, the numeric value is compared to the benign detection model.

String characteristics In some exploitations, the shellcode is passed to the function taking a buffer or string as argument. This case can be detected by recording characteristics of the string buffer in the model. Characteristics may include whether or not non-printable characters are allowed, which string encodings have been used (ASCII, UTF-8, UTF-16, ...), etc.

While it seems sufficient to flag function calls with parameter lengths or parameter values above a certain large value and check for null parameters, this does not suffice as these simple static checks prove insufficient. Consider the vulnerable function shown

```
1 #define HEADER_LENGTH 10
2 unsigned* copyBody( packet : unsigned*, packet_len : unsigned )
3 {
4     unsigned bodyLen = packet_len - HEADER_LENGTH;
5     unsigned* pArr = malloc( bodyLen );
6     // ...
7     return pArr;
8 }
```

Figure 3.10: Integer underflow example

in 3.10, in this example parameter *packet_len* with a value of 9 or lower would cause an integer underflow resulting in the *if* statement returning true. Secondly, *null* arguments are not necessarily incorrect, simply flagging them as such could therefore result in a large number of false positives.

3.6 Future work

After having developed an initial proof-of-concept for Adobe Reader, we first plan to test the accuracy in detecting malicious PDF documents of our approach. Once that has been verified, we will proceed with prototyping our approach to an engineering software. This second stage will mainly depend on the availability of vulnerabilities for testing. Some vulnerabilities have been uncovered in recent past which would allow our test.

Bibliography

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *CCS '05: Proc. 12th ACM conference on Computer and Communications Security*, pages 340–353. ACM Press, 2005.

[2] P. Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10, pages 12–12, Berkeley, CA, USA, 2010. USENIX Association.

[3] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996.

[4] AMD, Inc. AMD Virtualization. `www.amd.com/virtualization`.

[5] S. Andersson, A. Clark, and G. Mohay. Network based buffer overflow detection by exploit code analysis. In G. M. Mohay, A. J. Clark, and K. Kerr, editors, *AusCERT Asia Pacific Information Technology Security Conference: R&D Stream*, pages 39–53, Gold Coast, Australia, May 2004. University of Queensland.

[6] Avast. Six out of every ten users run vulnerable versions of Adobe Reader. `http://www.avast.com/pr-six-out-of-every-ten-users-run-vulnerable-versions-of-adobe-reader`.

[7] A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. In *In Proceedings of the USENIX Annual Technical Conference*, pages 251–262, 2000.

[8] U. Bayer, C. Kruegel, and E. Kirda. TTAnalyze: A Tool for Analyzing Malware. 2006.

[9] S. Bhatkar, R. R.S̃ekar, and D. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Sec: Proc. 14th Conference on USENIX Security Symposium*, volume 14 of *SSYM '05*, pages 17–17. USENIX Association, 2005.

[10] blexim. Basic integer overflows. *Phrack*, (60), December 2002.

[11] D. Bolzoni and S. Etalle. Boosting Web Intrusion Detection Systems by Inferring Positive Signatures. In *OTM '08: On the Move to Meaningful Internet Systems Confederated International Conferences*, volume 5332 of *LNCS*, pages 938–955. Springer-Verlag, 2008.

[12] D. Canali, A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda. A quantitative study of accuracy in system call-based malware detection. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA)*, 2012.

[13] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. DROP: Detecting Return-Oriented Programming Malicious Code. In A. Prakash and I. S. Gupta, editors, *Information Systems Security*, volume 5905 of *LNCS*, pages 163–177. Springer-Verlag, 2009.

[14] C. Cowan, F. Wagle, P. Calton, S. Beattie, and J. Walpole. Buffer overflows: attacks and defenses for the vulnerability of the decade. In *DISCEX '00: Proc. DARPA Information Survivability Conference and Exposition*, volume 2, pages 119–129, 2000.

[15] D. Dai Zovi. Practical return-oriented programming. SOURCE, 2010.

[16] L. Davi, A. Sadeghi, and M. Winandy. ROPdefender: a detection tool to defend against return-oriented programming attacks. In *ASIACCS '11: Proc. 6th ACM Symposium on Information, Computer and Communications Security*, pages 40–51. ACM Press, 2011.

[17] W. Dietz, P. Li, J. Regehr, and V. Adve. Understanding integer overflow in c/c++. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 760–770. IEEE Press, 2012.

[18] H. Etoh and K. Yoda. Protecting from stack-smashing attacks. 2000. Published on World-Wide Web at URL http://www.trl.ibm.com/projects/security/ssp/main.html.

[19] N. Falliere, L. O Murchu, and E. Chien. W32.stuxnet dossier. `http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf`, 2011. [Online; accessed 20-April-2013].

[20] Y. Feng and E. D. Berger. A locality-improving dynamic memory allocator. In *Proceedings of the 2005 workshop on Memory system performance*, MSP '05, pages 68–77. ACM, 2005.

[21] G. R. Fresi, L. Martignoni, R. Paleari, and D. Bruschi. Surgically Returning to Randomized lib©. In *ACSAC '09: Proc. 25th Annual Computer Security Applications Conference*, ACSAC '09, pages 60–69. IEEE Computer Society, 2009.

[22] G. Hoglund and J. Butler. *Rootkits: Subverting the Windows kernel*. Addison-Wesley Professional, 2005.

[23] iseclab. Anubis. `http://anubis.iseclab.org`, 2009.

[24] Document Management - Portable Document Format - Part 1:PDF 1.7, 2008.

[25] A. Iyer and L. Liebrock. Vulnerability scanning for buffer overflow. In *Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004. International Conference on*, volume 2, pages 116–117 Vol.2, 2004.

[26] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In *ACSAC '06: Proc. 22nd Annual Computer Security Applications Conference*, pages 339–348, Washington, DC, USA, 2006. IEEE Computer Society.

[27] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *RAID '06: Proc. 8th International Symposium on Recent Advances in Intrusion Detection*, LNCS, pages 207–226. Springer-Verlag, 2006.

[28] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda. AccessMiner: Using system-centric models for malware protection. In *Proceedings of the 17th Conference on Computer and Communications Security (CCS)*, 2010.

[29] J. Larus, T. Ball, M. Das, R. DeLine, M. Fahndrich, J. Pincus, S. Rajamani, and R. Venkatapathy. Righting software. *Software, IEEE*, 21(3):92–100, 2004.

[30] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with "return-less" kernels. In *EuroSys '10: Proc. 5th European Conference on Computer Systems*, pages 195–208. ACM Press, 2010.

[31] V. B. Lvin, G. Novark, E. D. Berger, and B. G. Zorn. Archipelago: trading address space for reliability and security. *SIGPLAN Not.*, 43(3):115–124, Mar. 2008.

[32] L. Martignoni, R. Paleari, and D. Bruschi. Conqueror: tamper-proof code execution on legacy systems. In *Proceedings of the 7th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, Lecture Notes in Computer Science, Bonn, Germany, 2010. Springer.

[33] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the ACM European Conference in Computer Systems (EuroSys)*, 2008.

[34] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig. Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. *Intel Technology Journal*, 10(3):167–177, August 2006.

[35] Northsecuritylabs. HyperSigh rootkit detector. `http://northsecuritylabs.com/downloads/whitepaper-html/`.

[36] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-Free: defeating return-oriented programming through gadget-less binaries. In *ACSAC '10: Proc. 26th Annual Computer Security Applications Conference*, pages 49–58. ACM Press, 2010.

[37] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. *Security and Privacy, IEEE Symposium on*, 0:601–615, 2012.

[38] M. Polychronakis, K. Anagnostakis, and E. Markatos. Network-level polymorphic shellcode detection using emulation. In *DIMVA '06: Proc. 3th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, LNCS, pages 54–73. Springer-Verlag, 2006.

[39] M. Polychronakis, K. Anagnostakis, and E. Markatos. Comprehensive shellcode detection using runtime heuristics. In *ACSAC '10: Proc. 26th Annual Computer Security Applications Conference*, pages 287–296. ACM Press, 2010.

[40] M. Polychronakis and A. Keromytis. ROP Payload Detection Using Speculative Code Execution.

[41] P. Ratanaworabhan, B. Livshits, and B. Zorn. NOZZLE: a defense against heap-spraying code injection attacks. In *USENIX Sec: Proc. 18th conference on USENIX Security Symposium*, pages 169–186. USENIX Association, 2009.

[42] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *CCS '07: Proc. 14th ACM conference on Computer and Communications Security*, pages 552–561. ACM Press, 2007.

[43] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *CCS '04: Proc. 11th ACM Conference on Computer and Communications Security*, pages 298–307. ACM Press, 2004.

[44] Skywing and M. Miller. Bypassing windows hardware-enforced data execution prevention. *Uninformed*, 2, September 2005.

[45] P. Software. PassMark Performance Test.

[46] Symantec. W32.Duqu: The Precursor to the Next Stuxnet, October 2011. `http://www.symantec.com/connect/w32_duqu_precursor_next_stuxnet`.

[47] Symantec. Flamer: Highly Sophisticated and Discreet Threat Targets the Middle East, May 2012. `http://www.symantec.com/connect/blogs/flamer-highly-sophisticated-and-discreet-threat-targets-middle-east`.

[48] G. T. and R. M. Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the Network and Distributed Systems Security Symposium, The Internet Society (2003)*, 2003.

[49] T. Toth and C. Kruegel. Accurate buffer overflow detection via abstract payload execution. In *RAID '02: Proc. 5th Internation Symposium on Recent Advances in Intrusion Detection*, pages 274–291, 2002.

[50] Use-after-free definition. `http://cwe.mitre.org/data/definitions/416.html`.

[51] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw. Its4: a static vulnerability scanner for c and c++ code. In *Computer Security Applications, 2000. ACSAC '00. 16th Annual Conference*, pages 257–267, 2000.

[52] G. Vigna, W. Robertson, and D. Balzarotti. Testing network-based intrusion detection signatures using mutant exploits. In *Proceedings of the 11th ACM conference on Computer and communications security*, CCS '04, pages 21–30. ACM, 2004.

[53] P. Vreugdenhil. Pwn2own 2010 windows 7 internet explorer 8 exploit. `http://vreugdenhilresearch.nl/Pwn2Own-2010-Windows7-InternetExplorer8.pdf`.

[54] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security and Privacy*, 5:32–39, 2007.

[55] R. Wojtczuk. Subverting the Xen hypervisor. *Black Hat USA*, 2008, 2008.

[56] J. Wolf. OMG WTF PDF. 27th Chaos Communication Congress, 2010.