SEVENTH FRAMEWORK PROGRAMME
Theme ICT-1-1.4 (Secure, dependable and trusted infrastructures)

WOMBAT

WORLDWIDE OBSERVATORY OF
MALICIOUS BEHAVIORS AND ATTACK THREATS

D11 (D4.3) Intermediate Analysis Report of Structural Features

Contract No. FP7-ICT-216026-WOMBAT

Workpackage	WP4 - Data Enrichment and Characterization
Author	-
Version	1.0
Date of delivery	M21
Actual Date of Delivery	30/09/2009
Dissemination level	Public
Responsible	TUV
Data included from	POLIMI,HISPASEC,SYMANTEC,VU

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement n°216026.

SEVENTH FRAMEWORK PROGRAMME
Theme ICT-1-1.4 (Secure, dependable and trusted infrastructures)

The WOMBAT Consortium consists of:

France Telecom	Project coordinator	France
Institut Eurecom		France
Technical University Vienna		Austria
Politecnico di Milano		Italy
Vrije Universiteit Amsterdam		The Netherlands
Foundation for Research and Technology		Greece
Hispasec		Spain
Research and Academic Computer Network		Poland
Symantec Ltd.		Ireland
Institute for Infocomm Research		Singapore

Contact information:
Dr. Hervé Debar
Rue des Coutures, 42
14066 Caen
France

e-mail: herve.debar@orange-ftgroup.com
Web: http://www.wombat-project.eu
Phone: +33 23 175 92 61
Fax: +33 23 137 83 43

Contents

Abstract

This deliverable provides a preliminary discussion of structural features that can be used to characterize executable code. Furthermore, it discusses a number of techniques, based on these features, that are being developed in the context of the WOMBAT project, and aim to provide a deeper understanding of malicious code and of the relations between malicious code samples.

1 Introduction

Several indicators suggest an exponential explosion of the number of newly available malicious software (*malware*) per day. For instance, the amount of samples submitted to VirusTotal [5], an online service to analyze suspicious files with more than 30 antivirus engines, is nowadays in the order of about one million samples per month [10]. Such numbers translate into an average load of approximately 30,000 new samples per day, which need to be analyzed to provide an understanding of current threats, leading to improved detection capabilities. Several antivirus vendors also report a growing number of samples that need to be examined on a daily basis in order to keep up with the latest attacks [29, 39].

Such a large number of samples can be partially explained by the easiness with which malware writers can generate new code forks by personalizing existing code bases, or by re-packing existing binaries using code obfuscation tools [27, 30, 43]. Note that malware sample counts are biased by the increasing usage of polymorphic techniques [6]. For instance, worms such as Allaple [14] take advantage of these techniques to mutate the binary content of the executable file at each propagation attempt. As a result, one malware family can be responsible for (tens of) thousands of samples per month. In spite of this huge load, novel techniques to quickly decide whether or not a given sample is a minor variant of a well-known malware family or a novel sample need to be developed.

The purpose of this document is to present a preliminary overview of what kind of *structural features* can be taken into account to quickly assess the novelty of a given sample (this overview will be extended in deliverable 17, due month 30). In the context of WOMBAT, we need a low-cost technique allowing us to easily cope with the large number of malware instances produced by polymorphic techniques and to easily distinguish them from new malware variants. Only by discerning these two cases will the security analyst be able to prioritize the usage of more costly analysis tools on potentially new code bases. Within the WOMBAT project we develop novel techniques to utilize information about the structure of a given malware sample. This information can be used in many different ways, for example to cluster a given set of samples based on structural similarity, or by generating a behavioral fingerprint that can then be statically matched against the structure of a set of malware samples. These techniques can then for example be used to quickly determine whether a given sample needs to be analyzed by costly dynamic analysis, or whether some kind of static analysis can already extract enough useful information about the sample. In the rest of the document we provide an overview of prior work related to the research focus of the WOMBAT project and report on the progress of our work in this area.

The document is structured as follows: in Chapter 2 we provide an overview of the state of the art in the area of structural analysis of malicious code. In the last few years, researchers from both academia and industry have developed several approaches (e.g., model checking, graph analysis, and symbolic execution) to take the structural information of samples into account.

Chapter 3 is split into two section which discuss different approaches for clustering malware samples based on static, structural features of a given malware sample: Chapter 3.1 introduces a techniques that is solely based on static characteristics that can be inferred by the analysis of the Portable Executable (PE) format headers of Windows executables. The intuition is that headers of executable files are typically not modified by different samples of the same (polymorphic) malware family since even polymorphic samples do not perform any kind of relinking and thus certain static features stay constant. Chapter 3.2 introduces a complementary technique, namely content-based static malware clustering: we compute the pairwise difference between all samples and use this information as a metric to cluster malware samples into families.

Chapter 4 describes how Argos, a containment environment for worms and manual system compromises, can be extended for structural analysis of the shellcode collected during an attack. This extension is able to analyze shellcode and extract the different layers of unpacking that it employs and the final "real" shellcode that performs useful actions for the attacker.

Finally, Chapter 5 introduces a novel approach to combine static and dynamic analysis to enhance the analysis process. The basic insight is that the results of dynamic analysis can also be leveraged to enhance static analysis: based on the results of dynamic analysis we can extract specific control flow information to fingerprint the code itself. This fingerprint can then also be recognized in different, but related malicious code samples.

2 State of the Art

Seminal studies on graph-based binary analysis are due to Halvar Flake [15, 16], who has been using graphs and similar comparison metrics to find differences between different versions of a given binary. His ideas and algorithms reveal some of the advantages in the automation of reverse engineering and code analysis [17]. In [12] Ero Carrera et al. proposed *Digital Genome Mapping*, a technique that uses graph theory to help in the analysis and identification of samples with a similar internal structure. The work introduces two types of signatures for similarity recognition. A *control flow graph (CFG) signature* which is the list of edges connecting basic blocks of a function, and a *call-tree signature* which is a fingerprint of which other functions are called inside a function body. The paper opens interesting perspectives on malware clustering and on phylogenetic classification.

The work by Kruegel et al. [25] aims at detecting polymorphic worms. The authors start from the fact that the structure of an executable is described by its CFG. They extract the CFG of a number of worms, than they create all the subgraphs of a given size of the CFG. They perform the detection doing the same with the file to analyze, and comparing all the subgraphs. Matches (i.e graph isomorphisms) are a symptom that the file is malicious. The system is made more robust by the use of a coloring system that associates a "color" to a given type of instructions inside a basic block. Match between subgraphs requires not only isomorphism but also color correspondence.

A similar approach is used by Bruschi et al. [8, 9]. First of all a disassembling phase is performed on the executable. Then a normalization phase follows aimed at reducing the effects of most of the well known mutation techniques and at unveiling the flow connection between the benign and the malicious code. Subsequent analysis is performed on the normalized code. Simple examples of normalization are removing dead and unreachable code, or rearrange control flow given by control flow instructions driven by tautologies. After normalization they use inter-procedural control flow graphs instead of simple CFGs. An inter-procedural CFG links together the CFGs of every function of a program. Moreover labeling is performed on both nodes and edges. Instructions similar from the semantic point of view, are grouped together into classes and the label assigned to each node is a number that represents the set of classes in which the instructions of the node can be grouped. Edges are labelled in the same way: possible flow transitions are grouped into classes according to the type of each transition.

Christodorescu and Jha [13] propose a technique that uses model checking to identify parts of a program that implement a previously specified malicious code template. This approach combats common virus obfuscation techniques by transforming virus source code into a malicious code automaton in order to handle inserted dead code and jumps between individual instructions. Unresolved symbols are used as placeholders for registers. If the language of the malicious code automaton has a non-empty intersection with the language of an automaton built from the program to be analyzed, then a viral code sequence is present in the program. This technique was later extended in [33], allowing more general code templates and using advanced static analysis techniques.

The approach of Shin [37] is to perform semantic analysis on malicious code. This starts by disassembling the malicious example using a commercial disassembler. The assembly code is then converted to a graph that is a hybrid of control flow and data dependence graphs. The graph is then split into subgraphs for each program subgoal (e.g., write data into a file or execute a process), and further each subgraph is converted into a finite state machine (FSM) representation. By performing inductive inference on output strings from the FSMs (which represent execution paths) they are able to develop a semantic signature for all possible malicious code in a particular class.

An approach to semantics-based malware detectors that can reliably handle malware variants derived through obfuscation or program evolution is the work of Kinder et al. [22]. Model checking is used to semantically identify malware that deviates from a temporal logic correctness specification. The potentially infected executable is disassembled and the control flow graph is extracted. The formula is then checked against the automata associated to the graph. A follow-up work [18] proposes a system for computer assisted generation of malicious code specifications in temporal logic.

Kruegel et al. [26] propose a kernel-level rootkits detection method based on symbolic execution. Kernel module binaries are symbolic executed in order locate instructions that write in memory areas of the kernel where modules usually don't write, e.g. the syscall table. A CFG is extracted from the binary with the aim of supporting the symbolic execution, for example for the detection of back edges that show the presence of a loop that requires limiting the analysis. In the paper by Kirda et al. [23] a behavioral-based approach is presented that relies heavily on static code analysis to detect Internet Explorer plug-ins that exhibit spyware-like behavior. Dynamic analysis is used to locate event handler locations in the code of the suspect spyware. Starting from those points the CFG is traversed and the reachable system calls are collected. The collected information is used in order to decide wether the executed code is malicious or not.

Wicherski developed an algorithm that transforms structural information from a sample in PE format into a hash value [41]. The basic idea is similar to the one presented in Section 3.1 and he also proposes to use this metric for clustering.

3 Static Clustering

3.1 Feature-based Clustering

We have developed a simple, but fast algorithm allowing to group together samples likely to be polymorphic instances of the same malware by looking solely at static characteristics that can be inferred by the analysis of the PE headers of Windows executables.

The intuition underlying this clustering algorithm derives from our experience in looking at the samples collected by the SGNET data set and at their features. We have seen that all the polymorphic techniques observed in the SGNET malware repository tend to randomize a limited amount of *features* of an attack event. The malware writer has to face the tradeoff between the desire of making the detection of the sample as difficult as possible and the practical cost of randomizing its different features. For instance, the polymorphic packer used by the Allaple worm [14] obfuscates and randomizes the data and code sections of each malware instance, but does not perform more expensive operations such as relinking. While the binary content changes at every propagation attempt, the headers of the executable files are not modified and have some immutable characteristics.

3.1.1 Clustering Phases

We define a simple pattern discovery technique to discover invariants among a set of features selected within the information available in the SGNET dataset. The technique is based on 4 different phases, namely feature definition, invariant discovery, pattern discovery and pattern-based classification.

Phase 1: Feature Definition

The feature definition phase consists in defining a set of features that have proven to be useful to characterize a certain activity class in our experiments. Table 3.1 shows the list of features that we have taken into consideration to characterize the malware samples. We have taken into consideration simple file properties, such as its size or its MD5 hash, as well as Portable Executable information extracted from the samples taking advantage of the PEfile [11] library.

Feature	# invariants
File MD5	57
File size in bytes	95
File type according to libmagic signatures	7
(PE header) Machine type	1
(PE header) Number of sections	8
(PE header) Number of imported DLLs	7
(PE header) OS version	1
(PE header) Linker version	7
(PE header) Names of the sections	43
(PE header) Imported DLLs	11
(PE header) List of referenced Kernel32.dll symbols	15

Table 3.1: Selected features

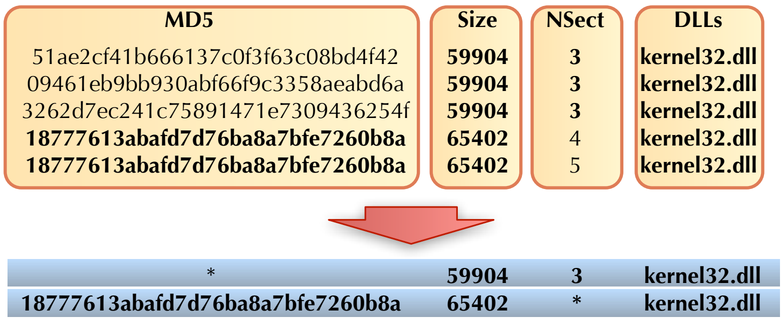
Clearly, all of the features taken into account for the classification could be easily randomized by the malware writer in order to evade this clustering technique. However, the feature selection underlines the apparent lack of interest and need in the malware community in proposing more sophisticated polymorphic approaches. More complex (and costly) polymorphic approaches might appear in the future, leading to the need to increase the clustering complexity by introducing a larger quantity of features, or by introducing more complex ones.

Phase 2: Invariant Discovery

For each of the features defined in the previous phase, the algorithm searches for all the *invariant* values. An invariant value is a the value of a non-randomized feature, that can therefore be useful as a discriminant to recognize a certain malware group.

In practice, we consider a value as invariant if a feature does not change its value across different attack instances (it is witnessed in multiple code injection attacks in the dataset), different attacking sources (the same value is used by multiple attackers in different attack instances) and different targets (multiple honeypot IPs witness the same value in different instances). An invariant value is therefore a “good” value that can be used to characterize a certain event type.

The definition of an invariant value is threshold based: we consider a value as invariant if it was seen in at least 10 different attack instances, and if it was used by at least three different attackers and witnessed on at least three honeypot IPs. Table 3.1 shows the amount of invariant values discovered for each feature.



MD5	Size	NSect	DLLs
51ae2cf41b666137c0f3f63c08bd4f42	**59904**	**3**	**kernel32.dll**
09461eb9bb930abf66f9c3358aeabd6a	**59904**	**3**	**kernel32.dll**
3262d7ec241c75891471e7309436254f	**59904**	**3**	**kernel32.dll**
18777613abafd7d76ba8a7bfe7260b8a	**65402**	4	**kernel32.dll**
18777613abafd7d76ba8a7bfe7260b8a	**65402**	5	**kernel32.dll**

*	59904	3	kernel32.dll
18777613abafd7d76ba8a7bfe7260b8a	65402	*	kernel32.dll

Figure 3.1: Pattern discovery starting from a limited number of invariants (in bold)

Phase 3: Pattern Discovery

Once the invariant values are defined on each attack feature, we look at the way in which the different features have been composed together, generating patterns of features. As exemplified in Figure 3.1, we define as pattern a tuple $T = v_1, v_2, ..., v_n$ where n is the number of features and v_i is either an invariant value for that feature or is a "don't care" value.

The pattern discovery phase looks at all the code injection attacks observed in the SGNET dataset for a certain period, and looks at all the possible value combinations for the different features, replacing all the non-invariant values with "don't care" values.

Phase 4: Pattern-based Classification

During this phase, the previously discovered patterns are used to classify all the attack instances present in the SGNET dataset. Multiple patterns could match the same instance: for example, the instance $1, 2, 3$ would be matched by both the pattern $*, 2, 3$ and the pattern $*, *, 3$. Each instance is always associated with the most specific pattern (i.e. the pattern with the smallest number of wildcards) matching its feature values. To avoid ambiguities in the case in which multiple patterns matching the same instance have the same number of wildcard, the algorithm gives priority to the pattern having higher specificity on the leftmost characteristics: $3, *, *$ has therefore priority over $*, *, 3$. However, when applying the algorithm over the SGNET dataset, such ambiguity was never observed.

# of invariants	# of clusters	# of samples
11	44	44
10	48	783
9	147	5287 (91% accumulated samples)
8	30	544
7	12	20
6	5	9
5	4	8
4	2	2
3	1	2

Table 3.2: Degrees of freedom in the clusters

3.1.2 Discussion

We have applied the malware classification algorithm previously described to all the 6699 malware samples that were collected by the SGNET deployment and that proved to be valid executable files when executed in Anubis.

Table 3.2 details the distribution of the generated clusters in terms of number of invariant features associated to the pattern. For instance, a specific pattern characterized by the following features is associated to 10 invariants:

```
{
   MD5='*',
   size=59904,
   type='MS-DOS executable PE  for MS Windows (GUI) Intel 80386 32-bit',
   machinetype=332,
   nsections=3,
   ndlls=1,
   osversion=64,
   linkerversion=92,
   sectionnames='.text\x00\x00\x00,rdata\x00\x00\x00,.data\x00\x00\x00'
   importeddll='KERNEL32.dll',
   kernel32symbols='GetProcAddress,LoadLibraryA'
}
```

That is, this specific malware variant is associated with a polymorphic engine that mutates the sample content without modifying any other of its features.

Looking at Table 3.2, we can see that the vast majority of the samples taken into consideration is associated with a high number of invariants. 6114 samples, that is 91% of the total amount of samples in the SGNET dataset, are classified by patterns with 9 invariants, and are therefore associated to variability in at most 2 features.

On the other hand, we have a small number of cluster that exhibit a much higher variability. A more in-depth analysis shows that such clusters are associated with a very low number of attack events, and this prevented our algorithm from correctly discovering the invariants associated with the malware propagation. By collecting more instances of these samples, additional invariants might appear.

We have shown how the usage of simple structural features in the malware characteristics can be sufficient to discover easily polymorphic instances of the same codebases. The proposed clustering algorithm allows to quickly classify large malware sets in a short period of time: we classified over 6000 samples in less than 10 minutes on a normal PC configuration. This method provides an effective way to prioritize the analysis effort on new variants filtering out the noise generated by polymorphism. While the limited number of features taken into account here was sufficient to discover meaningful patterns on the SGNET dataset, the proposed classification leverages the tradeoff between the complexity of polymorphic techniques and their cost. As previously explained, more sophisticated malware variants might require in the future an increase in the number of considered features, and an increase in their sophistication.

3.2 Content-based Static Malware Clustering

A single approach to the malware clustering problem is not always enough. The method discussed in the previous section is effective in the case of polymorphic families, where the content of the PE sections changes dramatically from sample to sample but the PE structure remains similar. However, it is not straightforward to define relationships between the different malware groups. The static clusters generated by the feature-based clustering can be associated to simple recompilations of the same code base, such as different personalizations of the Spybot source code in order to support different Command and Control servers. They can equivalently be representing completely different malware variants, with different features and characteristics. It is difficult, or maybe impossible, to infer from simple features of the malware sample the type of relationship among the different clusters. We have therefore considered a complementary technique, able to fully inspect the malware content in order to infer the amount of overlap among different samples.

3.2.1 Measuring Sample Similarity

The first thing we need to do before we can implement an algorithm for grouping similar samples is to define a way to measure the degree of similarity between two of them. From now on, we will refer to this degree of similarity as the distance between the two samples. The more similar the samples, the smaller the distance between them.

Our problem here is finding a good way of calculating this distance. A common approach to define a distance between two arbitrary sequence of bytes is using the length of their longest common sub-sequence (LCS). But the existing algorithms to calculate the LCS are $O(n^2)$, and therefore, slow for practical purposes.

This is when delta algorithms come into play. Delta algorithms are aimed to receive two files and generate a set of instructions that can be used to convert one of the files into the other. In theory, delta algorithms are strongly related to finding the LCS of the two files, because finding the optimum set of instruction to convert a file into the other requires solving the LCS problem. In practice however, most delta algorithms sacrifice the optimality of the solution in favor of execution speed.

The algorithm used to calculate the distance between samples F_1 and F_2 in this case is inspired in xdelta [28], and generates the set of instructions to convert F_1 into F_2. We are not interested in the instructions themselves, but in the number of them, that we will call N. The distance follows the expression:

$$d = \frac{|s_1 - s_2| + 2N}{s_1 + s_2} \tag{3.1}$$

Here s_1 and s_2 are the sizes of the samples F_1 and F_2 being compared. It must be said that this distance is not symmetrical (i.e. $d(F_1, F_2) \neq d(F_2, F_1)$), although it tends towards symmetry when the samples are closer in size. For samples with very dissimilar sizes, but which are still similar in some way (consider, for example, the case of comparing a sample with a truncated version of itself), this asymmetry could lead to very different results depending on the order of comparison. To avoid this discrepancy, we decided that the biggest sample would always be the reference file F1, and the smallest one the target file F2. This is because the delta algorithm generates fewer instructions when trying to convert a large file into a smaller one than the opposite way.

3.2.2 Clustering Samples

In order to group the similar samples together the single-link clustering algorithm [19, 20] was employed. This algorithm receives a matrix with the distances between the samples of our dataset, and creates a dendogram showing the similarity relation between them.

Cluster	**# samples**
1	423
2	107
3	48
4	18
5	11
6	11
7	9
8	8
9	8
10	6
11	6
12	6
13	6
14	6
15	6
16	5
17	4
18	4
19	4
20	3
21	3
22	2
23	2
24	2
25	2
26	2
27	2
28	2
29	2
30	2
31	2
32	2
33	2
34	2
35	2
36	2
37	2
38	2
39	2

Table 3.3: Clusters generated by content-based static clustering

When applying the algorithm to a subset of 3950 samples taken from the set of 6699 analyzed in the previous section, 39 clusters were generated as shown in Table 3.3. As can be seen, the biggest generated cluster accounts for more than 10% of the samples analyzed, which is an acceptable result taking into account the polymorphic nature of the Allaple family.

3.2.3 Limitations

The big disadvantage of this clustering algorithm is its performance. The number of sample comparisons that must be done to fill the initial distance matrix for the clustering algorithm is $\frac{N(N-1)}{2}$. As the number of samples grows linearly, the number of comparisons grows quadratically. The sample comparison algorithm is not very costly in itself, it only depends linearly on the size of the files being compared, but it requires the content of both files to be read completely, incurring a large number of I/O disk operations. With modern hardware, applying this algorithm to up to 5000 samples is still affordable, but above this number the time required to complete the process make it unpractical.

4 Deriving the Structure of Shellcode

The Argos emulator [32] is used by various projects to detect zero-day attacks. Some advanced versions of Argos, like Prospector [38], go beyond basic detection by identifying the bytes that are responsible for buffer overflows and matching these bytes with protocol fields of network traffic to serve as signatures. However, even the most advanced version of Argos to date stops at the shellcode. In fact, we have carefully tried to avoid executing even a single instruction of the attacker's code.

On the one hand, this design decision has helped deployment. Administrators are less reluctant to host a complex piece of intrusion detection software if they know that malicious code will not be executed. As such, it has been one of the key selling points of Argos.

On the other hand, stopping at the very first instruction limits our options for further analysis. It is hard to determine what the shellcode does, which server it tries to contact, and which binary it downloads, unless we execute part of the shellcode. A simple analysis of the instructions in memory reveals very little as shellcode is typically obfuscated by means of several layers of packing (see Figure 4.1 for a common shellcode structure). Without running the unpacking routines, the bytes in memory are mostly meaningless.

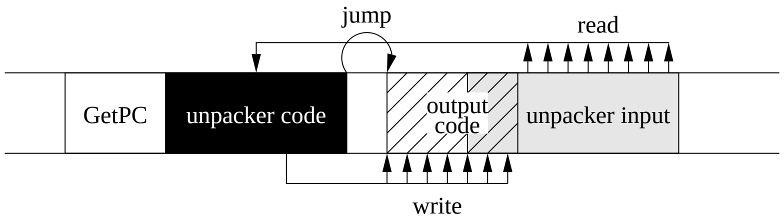


Figure 4.1: Common shellcode: a GetPC routine obtains the current program counter, which is then used by an unpacker that decodes the real shellcode in memory

In addition, preventing any instruction from being executed is probably overly strict. In almost all practical cases, it is safe to execute instructions until a system call is encountered. The reason is that without system calls the potential damage that can be caused by the shellcode is limited to the process' memory. It should not propagate and it cannot compromise permanent state. The only exception is one in which the victim process shares memory with another process and the shellcode corrupts this memory directly. However, not only is this behavior rare for shellcode, in the case of tightly controlled honeypot systems we can also use the detection of an exploit to trigger a reboot of the honeypot after executing a set of shellcode instructions (perhaps up to a predefined maximum). The reboot removes any trace of the attack from the target system.

4.1 Overview

In this section, we describe a new version of Argos that allows administrators to specify (via a command line option) that Argos should not stop at exploit detection, but rather continue executing the shellcode. The work described in this section straddles several subtasks of Workpackage 4. The main requirements for the extension are that it should (a) track shellcode while it executes, including all the bytes that it uses to unpack the shellcode, (b) note the entry points of all levels of unpacking as well as that of the final (real) shellcode, and (c) extract contextual information such as the machine from which to download the malware binary. The entry point chain forms part of the *structure* of the exploit, while the way in which the packers read and write from the process' address space was specified in the technical annex as part of the *behavioral* analysis. Finally, information extracted from the exploit typically fits in the *context* of the attack.

Scattering small bits of explanation over two or three deliverables does not help readability of the text. For this reason, we will concentrate most of the explanation of Argos shellcode analysis in D4.2. In this deliverable we will merely describe what is relevant for structural analysis of the shellcode. At the risk of some repetition, we strive to make this section as self-contained as possible.

To do so, we start by summarizing Argos and then discuss, at a fairly high level, the notion of shellcode execution in Argos. The focus of the discussion will be on structural aspects. Details about the way we implemented the analysis, and how we overcame shortcomings in the underlying Qemu emulator that prevented common shellcode from executing can be found in D4.2.

4.2 Argos Primer

Argos is an emulator based on Qemu that employs dynamic taint analysis to detect attacks. It maintains a separate shadow memory that is not accessible to the guest operating system or its applications. Argos uses the shadow memory to store taint tags. Briefly, it tags all data from a suspect source (typically the network) as tainted.

Whenever such data is copied, or used in a calculation, the destination address or register is also tainted. Conversely, certain CPU instructions implicitly clean the tags. For instance, storing a constant or clean value at a memory address that was previously tagged as tainted, will clean the tag.

Argos offers different flavors of tags, but the most common ones are 32 bit tags for every byte of guest memory in use[1]. If the byte is tainted, the tag represents an offset into a dump of the network traffic. Phrased differently, for every tainted byte in memory, we track the corresponding bytes in the network stream.

As an aside, it is clear that encrypted channels make it much harder to trace the bytes in memory back to bytes in the network trace. In HaSSLe [40], we have shown how we can still apply tracking in Argos, even in the face of encryption.

4.3 Shellcode Execution Monitoring

Up to the point of detecting an attack, the extended version of Argos behaves just like the version described above. As soon as the exploit is detected, Argos logs the event and makes a transition to shellcode tracking mode. In this mode, the actual shellcode instructions are executed until a specific event occurs. Possible events include:

- The first call to a specific set of functions (for instance, a system call).
- A certain number of instructions executed.

Execution, however, is not the same as analysis. We are interested in the structure of the shellcode, that is, the different layers of unpacking that it employs and the final 'real' shellcode that performs useful actions for the attacker. Collectively, we refer to all the unpackers and the final shellcode as layers of execution. To find the structure, we strive to obtain the entry points of each consecutive layer of execution.

Unpackers typically work by reading bytes in memory, applying a decoding function and writing the results back to memory. Typically, both the source and the target of the decoding function are in tainted memory areas, although we do not preclude the use of clean bytes. For example, a very simple unpacker is one which simply XORs a range of bytes with a constant, as follows:

```
MOV ECX,100      ; ECX will be the loop counter
LEA ESI,[EBX+20] ; load start of input data for unpacker
MOV EDI,ESI      ; copy
```

[1]The actual memory consumption of the tags is limited as they are implemented by means of paging, so that memory that is not in active use also does not consume shadow memory.

```
L1: LODSB           ; load byte at address ESI into AL and increment ESI
    XOR AL,99h      ; decode using XOR with 0x99
    STOSB           ; store byte in AL at address EDI and increment EDI
    LOOP L1         ; Loop with ECX as counter
```

After unpacking, the shellcode *jumps* to this memory area and starts executing the bytes that it previously unpacked here as executable instructions. These instructions may not represent the final shellcode, as attacks often use various layers of packing and unpacking. The final shellcode only reveals itself by doing something useful for the attacker, such as downloading the malware binary and executing it. Doing so from a user space application requires system calls.

To determine the code structure, we execute the shellcode instructions under supervision, tracking all the bytes they read and write. We explicitly log a jump to an address that was just overwritten as the start of the a new layer of execution and the jump target as the entry point.

We keep executing and unpacking the shellcode until the code attempts to execute an event that has implications beyond the process' address space. In other words, we stop when the code makes a system call. The last entry point is then marked as the start of the real shell code and the system call as the first useful action on behalf of the attacker. We can now keep executing the shellcode in single step mode, or dump the shellcode and the newly derived meta-data to file for later analysis.

5 Hybrid Static-Dynamic Approaches

In the context of the WOMBAT project, techniques are being developed to characterize and analyze malicious code on the basis of its behavior (*dynamic analysis*) and on the basis of its structure (*static analysis*). As we will discuss in this chapter, these two approaches have complementary strengths and weaknesses. It is therefore clearly advantageous to develop techniques that can combine static and dynamic analysis of malware to provide a richer understanding of malicious code.

5.1 Static and Dynamic Analysis: Strengths and Weaknesses

Dynamic Analysis

Dynamic analysis of malicious code works by executing the program in a controlled environment and observing its behavior. To perform its intended behavior, malware needs to go through the operating system interface. For instance, to modify the file system, read private data from the file system or to send data over the network, a user-land program needs to make use of specific system calls. Because the interface between user-land processes and the operating system is relatively simple and well-understood, a dynamic analysis system such as ANUBIS [1] or CWSandbox [3, 42] is capable of reliably detecting the effects produced by a program on the system it is running on.

Unfortunately, dynamic analysis can only observe behavior that the malware performs while running inside the analysis sandbox. This weakness has been exploited by malware that simply stops execution when it detects it is running inside a sandbox. According to a recent study [6], between 0.3% and 12.5% of samples submitted to ANUBIS detect the ANUBIS sandbox and refuse to run. While these sandbox detection techniques can in principle be thwarted by perfectly emulating a real system, there are other reasons why a malware sample may not perform all of the behavior it is capable of during analysis.

Early malicious software, such as file-infecting viruses and network worms, was designed to operate autonomously and perform a pre-programmed behavior on all systems it managed to infect. Modern malware, however, is usually remote-controlled over some kind of Command-and-Control (C&C) channel. For instance, by analyzing the source code of a variant of the rBot bot, we discovered that this malware version supports more than one hundred different commands. Over an IRC-based C&C channel, it can be instructed to perform a variety of DOS attacks, to attempt different exploits against other systems, and to capture sensitive information from the infected system such as keypresses, network traffic, screen captures and even to record video using a webcam. However, in the few minutes that a malware sample is analyzed in ANUBIS it is highly unlikely that a bot master would send many of these commands to his botnet. Dynamic analysis of modern malware may therefore reveal only a small fraction of the functionality implemented by the malicious code.

Static Analysis

Using dynamic analysis it is trivial to recognize the high-level, operating system behavior of an executed sample, but this is not the case with static analysis. To interact with the operating system interface, software typically invokes APIs provided by system shared libraries. Since the addresses at which shared libraries are loaded may vary with each execution, these functions are invoked through pointers that are initialized during program startup. To recognize specific API calls, disassembly tools such as IDA Pro [4] rely on knowledge of standard ways of performing this initialization. If a custom mechanism is used instead, static analysis may be unable to recognize API calls. Even if the calls to API functions can be recognized, it may be impossible to understand their high level semantics unless the values of the parameters passed to these functions can be statically inferred.

However, even when it cannot recover the operating system semantics of analyzed code, static analysis can provide valuable insight. The control flow graph (CFG) of a program is a directed graph representing all of the possible paths that can be traversed during execution of the program. A node of the CFG is a basic block: a group of instruction that are always executed in sequence, without any conditions or jumps. An edge in the CFG represents a possible jump between two basic blocks. Recent research has shown that the CFG is a distinguishing characteristic of binary code that may persist across recompilation, and even across polymorphic mutation. Flake presented a technique which uses the CFG and the callgraph of two versions of a program (before and after a patch) to automatically identify the functions that were patched [17]. This idea has been implemented in the commercial tool bindiff [2]. Kruegel et al. utilize a similar idea: the CFG of binary code being transmitted on the network is used to automatically generate a CFG-based signature for a network worm [25]. In a related work, Bruschi et al. use CFG-based signatures to detect self-mutating malware [8].

Packing

Techniques developed to analyze malicious code must be developed under different requirements than techniques aimed at analyzing ordinary, non-obfuscated code. Malware analysis operates in an adversarial context, because the authors of malicious programs have an economic incentive to make it harder to detect or even just to understand their code. Purely static approaches to the analysis of malicious code are defeated by the widespread use of off-the-shelf obfuscation technology such as packing [34]. For instance, about 40% of the samples analyzed by ANUBIS [1] are obfuscated using a known packer [6].

Fortunately, most types of packing can be defeated by generic unpacking techniques [21, 34] that are based on dynamic analysis. The basic principle of these techniques is that the packed binary is executed until it unpacks itself. The unpacked code can then be analyzed using conventional static analysis. Currently, we use existing unpackers for this purpose, but the ANUBIS system could be extended to better perform this task. One of the challenges of generic unpacking is that the unpacker has to decide when the analyzed code is done unpacking itself. Since obfuscated code may use multiple layers of packing, this is far from trivial. ANUBIS, on the other hand, executes analyzed code for several minutes, which is normally more than enough time for malware to unpack itself. In fact, a simple dump of the memory used by the malicious code at the end of ANUBIS analysis will typically contain the unpacked malware code. Therefore, ANUBIS provides an ideal platform for the unpacking of malicious code.

There are however packing techniques that cannot be defeated by existing generic unpacking methods. These include emulation-based packing [35] and conditional code obfuscation [36]. In emulation-based packing, each malware instance includes an emulator implementing a randomly selected instruction set. The malware code is then compiled to this instruction set. Recent techniques show some promise in defeating this type of obfuscations [35]. In conditional code obfuscation, each conditional branch of a program is separately encrypted with a key that depends on the branch condition. In some cases, this can provide cryptographically strong guarantees that the code cannot be analyzed statically. Despite the limitations of generic unpacking, our experience with the malware dataset collected by ANUBIS indicates that current techniques are able to unpack most malicious code currently in the wild.

5.2 Hybrid Analysis

Techniques for the analysis of malicious code that combine static and dynamic analysis may be able to leverage the complementary strengths of these approaches. In this section, we will briefly discuss the current direction of research in this area within the WOMBAT project. This work is currently in its early stages. The results of this work will be presented in more detail in deliverable D17 (D4.4) "Final analysis report of structural features".

Mapping Behavior to Code

As we discussed in Section 5.1, the control flow graph (CFG) of code can be used as a fingerprint or signature for the code itself. Using techniques from Kruegel et al. [25], in particular, we are able to fingerprint subgraphs of the CFG of a program, and recognize the occurrence of the same subgraphs in different but related malicious code. This information could be used to classify malicious code samples into clusters of related code, and produce a structural phylogeny of malicious code, similar to the behavioral phylogeny provided by the ANUBIS clustering feature [7] that was developed within WOMBAT.

The focus of our current work, however, is to understand what a CFG fingerprint corresponds to in terms of high level, security level malware behavior. To this end, we are developing techniques to map malware behavior observed during dynamic analysis to the code that is chiefly responsible for that behavior. For this purpose, we first map a behavior to the system or API calls that are responsible for it. To identify the actual code that leads to those system calls, or processes the results of those system calls, we plan to leverage taint tracing [31] of data originating from these calls, as well as the program slicing techniques we developed recently [24].

Once a high level behavior has been mapped to the code implementing it, we can extract the CFG-based fingerprints of this code. Searching for the same fingerprints in other (unpacked) malware samples should allow us to recognize a known behavior even in malware that does not execute it during dynamic analysis. This would alleviate the main weakness of dynamic analysis that we discussed in Section 5.1. That is, the fact that dynamic analysis can give no information on parts of the analyzed code that are not executed during analysis. At the same time, it would work around the difficulty of understanding the high level behavior of code using static analysis only.

Bibliography

[1] Anubis. http://anubis.iseclab.org/.

[2] zynamics BinDiff. http://www.zynamics.com/bindiff.html.

[3] CWSandbox. http://www.cwsandbox.org/, 2008.

[4] IDA Pro. http://www.hex-rays.com/idapro/, 2008.

[5] Virus Total. http://www.virustotal.com/, 2008.

[6] U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, and C. Kruegel. Insights into current malware behavior. In *LEET'09: 2nd USENIX Workshop on Large-Scale Exploits and Emergent Threats, April 21, 2009, Boston, MA, USA*, Apr 2009.

[7] U. Bayer, P. Milani Comparetti, C. Kruegel, and E. Kirda. Scalable, Behavior-Based Malware Clustering. In *16th Symposium on Network and Distributed System Security (NDSS) (to appear)*, 2009.

[8] D. Bruschi, L. Martignoni, and M. Monga. Detecting self-mutating malware using control-flow graph matching. In *Detection of intrusions and malware and vulnerability assessment (DIMVA)*, 2006.

[9] D. Bruschi, L. Martignoni, and M. Monga. Code normalization for self-mutating malware. *IEEE Security & Privacy*, 5(2):46–54, 2007.

[10] J. Canto, M. Dacier, E. Kirda, and C. Leita. Large scale malware collection: Lessons learned. In *IEEE SRDS Workshop on Sharing Field Data and Experiment Measurements on Resilience of Distributed Computing Systems*, 2008.

[11] E. Carrera. Pefile, http://code.google.com/p/pefile/.

[12] G. Carrera E.; Erdlyi. Digital genome mapping advanced binary malware analysis. *Virus Bulletin Conference*, pages 187–197, 2004.

[13] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 12–12, Berkeley, CA, USA, 2003. USENIX Association.

[14] F-Secure. Malware information pages: Allaple.a, `http://www.f-secure.com/v-descs/allaplea.shtml`.

[15] H. Flake. Graph-Based Binary Analysis. In *Black Hat Briefings*, 2002.

[16] H. Flake. Have More fun with Graphs. In *Black Hat Federal*, 2003.

[17] H. Flake. Structural comparison of executable objects. In *Proc. of Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, 2004.

[18] A. Holzer, J. Kinder, and H. Veith. Using verification technology to specify and detect malware. In R. Moreno-Díaz, F. Pichler, and A. Quesada-Arencibia, editors, *EUROCAST*, volume 4739 of *Lecture Notes in Computer Science*, pages 497–504. Springer, 2007.

[19] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: A review, 1999.

[20] A. K. Jain, E. Topchy, M. H. C. Law, and J. M. Buhmann. Landscape of clustering algorithms. In *In Proceedings of the 17th International Conference on Pattern Recognition (ICPR*, pages 260–263, 2004.

[21] M. G. Kang, P. Poosankam, and H. Yin. Renovo: a hidden code extractor for packed executables. In *WORM '07: Proceedings of the 2007 ACM workshop on Recurring malcode*, 2007.

[22] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith. Detecting malicious code by model checking. In K. Julisch and C. Krügel, editors, *DIMVA*, volume 3548 of *Lecture Notes in Computer Science*, pages 174–187. Springer, 2005.

[23] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. A. Kemmerer. Behavior-based spyware detection. In *Usenix Security Symposium*, 2006.

[24] C. Kolbitsch, P. Milani Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and Efficient Malware Detection at the End Host. In *18th Usenix Security Symposium*, 2009.

[25] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *RAID*, 2005.

[26] C. Kruegel, W. Robertson, and G. Vigna. Detecting Kernel-Level Rootkits Through Binary Analysis. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC 2004)*, Tuscon, AZ, USA, December 2004.

[27] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *10th ACM conference on Computer and communications security*, pages 290–299. ACM New York, NY, USA, 2003.

[28] J. Macdonald. Versioned file archiving, compression, and distribution. Technical report, 1999.

[29] McAfee. Threats Report First Quarter 2009, 2009.

[30] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *23rd Annual Computer Security Applications Conference (ACSAC)*, 2007.

[31] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *12th Annual Network and Distributed System Security Symposium (NDSS)*, 2005.

[32] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an Emulator for Fingerprinting Zero-Day Attacks. In *Proc. ACM SIGOPS EUROSYS'2006*, Leuven, Belgium, April 2006.

[33] M. D. Preda, M. Christodorescu, S. Jha, and S. Debray. A semantics-based approach to malware detection. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 377–388, New York, NY, USA, 2007. ACM.

[34] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *Computer Security Applications Conference, 2006. ACSAC '06. 22nd Annual*, 2006.

[35] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Rotalume : A tool for automatic reverse engineering of malware emulators. In *SP '09: Proceedings of the 2009 IEEE Symposium on Security and Privacy*, 2009.

[36] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee. Impeding malware analysis using conditional code obfuscation. In *NDSS*, 2008.

[37] J. Shin and D. F. Spears. The basic building blocks of malware. Technical report, University of Wyoming, 2006.

[38] A. Slowinska and H. Bos. The age of data: pinpointing guilty bytes in polymorphic buffer overflows on heap or stack. In *23rd Annual Computer Security Applications Conference (ACSAC'07)*, Miami, FLA, December 2007.

[39] Symantec. Global Internet Security Threat Report: Trends for 2008, April 2009.

[40] M. Valkering, A. Slowinska, and H. Bos. Tales from the Crypt: fingerprinting attacks on encrypted channels by way of retainting. In *Proc. of 3rd European Conference on Computer Network Defense (EC2ND)*, Heraklion, Greece, October 2007.

[41] G. Wicherski. peHash: A Novel Approach to Fast Malware Clustering. In *LEET'09: 2nd USENIX Workshop on Large-Scale Exploits and Emergent Threats, April 21, 2009, Boston, MA, USA*, Apr 2009.

[42] C. Willems, T. Holz, and F. Freiling. CWSandbox: Towards automated dynamic binary analysis. *IEEE Security and Privacy*, 5(2), 2007.

[43] T. Yetiser. Polymorphic viruses - implementation, detection, and protection , `http://vx.netlux.org/lib/ayt01.html`.