Dynamic Spyware Analysis

Manuel Egele, Christopher Kruegel, Engin Kirda
Secure Systems Lab
Technical University Vienna
{pizzaman,chris,ek}@seclab.tuwien.ac.at

Heng Yin
Carnegie Mellon University and College of William and Mary
hyin@ece.cmu.edu

Dawn Song
Carnegie Mellon University
dawnsong@cmu.edu

Abstract

Spyware is a class of malicious code that is surreptitiously installed on victims' machines. Once active, it silently monitors the behavior of users, records their web surfing habits, and steals their passwords. Current anti-spyware tools operate in a way similar to traditional virus scanners. That is, they check unknown programs against signatures associated with known spyware instances. Unfortunately, these techniques cannot identify novel spyware, require frequent updates to signature databases, and are easy to evade by code obfuscation.

In this paper, we present a novel dynamic analysis approach that precisely tracks the flow of sensitive information as it is processed by the web browser and any loaded browser helper objects. Using the results of our analysis, we can identify unknown components as spyware and provide comprehensive reports on their behavior. The techniques presented in this paper address limitations of our previous work on spyware detection and significantly improve the quality and richness of our analysis. In particular, our approach allows a human analyst to observe the actual flows of sensitive data in the system. Based on this information, it is possible to precisely determine which sensitive data is accessed and where this data is sent to. To demonstrate the effectiveness of the detection and the comprehensiveness of the generated reports, we evaluated our system on a substantial body of spyware and benign samples.

1 Introduction

An important security threat that affects many Internet users today is spyware [24, 25]. Spyware is malicious software that attempts to silently monitor the behavior of users, record their web surfing habits, or steal their sensitive data such as passwords. Typically, the collected information is sent back to the spyware distributor, where it is (ab)used for targeted advertisement or marketing studies. This is different from other types of malware, such as viruses and worms, which generally aim to propagate to other systems and cause damage.

As the spyware problem has intensified, a number of commercial solutions have been introduced that aim to identify and remove undesired spyware. These tools are similar to anti-virus products in that they identify *known* instances of spyware by comparing the binary image of unknown samples to a database of signatures. Often, these signatures are generated manually by analyzing known spyware samples (which is a tedious task when one considers that hundreds of new samples have to be analyzed every day). Unfortunately, spyware detection tools suffer from the known drawbacks of signature-based detectors, such as the continuous need for updates of the signature database and the inability to identify previously unknown samples. Note that a major drawback of signature-based techniques is that they are also often not able to deal with simple obfuscation techniques [3].

Because signature-based detection techniques have significant shortcomings, we previously presented a behavior-based spyware detection technique that used a combination of static and dynamic analysis to identify malicious behavior of Internet Explorer browser helper objects (BHOs) [14]. Using our previous tool, we could classify unknown components as malicious or benign. Unfortunately, our approach also has a number of limitations. First, we could only assert the *possibility* that sensitive information is leaked, but we were unable to establish *exactly what* data is collected by a spyware component. This information is required by human spyware analysts that need to understand and estimate the damage caused by a specific spyware program. Second, because of our substantial reliance on static analysis of potentially hostile code, a spyware author who is aware of our technique can use code obfuscation to attempt to evade detection (or to make detection more difficult and costly).

In this paper, we present a novel dynamic analysis approach that precisely tracks the flow of sensitive information as it is processed by the web browser and any

loaded BHOs. Based on the results of our analysis, we can classify unknown components as benign programs or spyware and provide comprehensive reports on their behavior. To identify information flows, we make use of dynamic taint analysis, which tags sensitive data elements and tracks their use as they are processed. Our taint analysis combines the traditional whole system approach (in which data is tainted at a physical level) with the ability to monitor activity within individual Windows operating system processes. This is necessary to distinguish between the use of sensitive information by the Internet Explorer and the abuse of the same data by malicious browser objects. The techniques presented in this paper address limitations with our previous approach and significantly improve the quality of our analysis reports. By tracking actual information flows, we can more precisely understand and characterize the behavior of spyware. In particular, we can determine *which* sensitive data is leaked and *where* it is sent.

The main contributions of this paper are the following:

- We introduce a dynamic analysis technique to precisely monitor the flow of sensitive data as it is processed by the web browser and browser helper objects. By tracking the actual information flow using taint analysis, we can *precisely* determine which sensitive data is collected by a spyware component. Unlike previous approaches that use dynamic taint analysis, our system not only considers data dependencies, but also control dependencies.
- We present a tool that can be used to automatically analyze the behavior of unknown BHO samples and provide comprehensive reports on their activities.
- We present experimental results on a substantial body of 21 spyware samples and 14 benign BHO samples that demonstrate the effectiveness of our approach.

2 Spyware Analysis Approach

In general, spyware refers to a category of malicious software that monitors a user's operations without her consent, typically to the benefit of a third party. Spyware exists in many forms and performs actions of different levels of maliciousness. In this paper (as well as in our previous work), we explicitly focus on spyware that exploits the hooks provided by Microsoft's Internet Explorer to monitor the actions of a user. This is done by using the browser helper object (BHO) interface. In a nutshell, browser helper objects are Windows dynamic linked libraries that are automatically loaded by the Internet Explorer when it is launched. BHOs are mostly used to extend the Internet Explorer with small, custom add-ons or utilities. Examples include helpers that block pop-ups, implement support for mouse gestures, or provide embellishments (images) for web pages. Although possible, they rarely implement more complex functionality such as multi-media extensions or Java interpreters, which are realized as Internet Explorer plug-ins. Most browser helper objects do not contain any user interface elements and work in the background, responding to browser events and user input. However, they run in the same address space as the browser and have full control over the browser's functionality.

The focus on spyware that is implemented as BHOs is justified by the fact that the large majority of spyware has a component based on this technology. This is confirmed by a recent study [26], which found that out of 120 distinct spyware programs, just under 90 used BHOs as an entry point to monitor user activity. In addition, a US CERT report [10] names BHOs as one of the most frequently used techniques employed by spyware.

In previous work, we proposed the following behavioral characterization to classify a BHO as spyware:

> "A distinctive characteristic of spyware is that a spyware component (or process) collects data about user behavior and forwards this information to a third party. Thus, a BHO is classified as spyware when it **(i)** monitors user behavior **(ii)** then leaks the gathered data to the attacker."

To determine whether an unknown component exhibits malicious behavior, we used a combination of dynamic and static code analysis techniques. The dynamic analysis identified whether a BHO calls browser functions that could be used to gather sensitive user data. The static analysis then determined whether the component contained calls that could leak this information.

Experimental evaluation demonstrated that our previous system yielded good detection results with low false positives. However, there are two significant limitations with our previous approach. One is that our approach can only identify the *possibility* that information could be leaked. We were not able to record any actual information flow. Thus, it is not possible to determine precisely *which* sensitive data is accessed or leaked. Also, it is not possible to identify precisely *where* the data is sent. Obviously, such knowledge is invaluable for a human analyst who has to manually analyze a large body of new samples every day. The second limitation is our significant dependency on static analysis, which can be exploited by a spyware author who uses code obfuscation to make it difficult to disassemble the binary [20] or hide the presence of certain function calls [5, 27]. Unfortunately, when our analysis fails to identify those function calls that are associated with malicious behavior, a spyware component is incorrectly labeled as benign. At

the same time, if a more conservative approach is used, the false positive rate increases and benign samples are falsely labeled as being malicious.

2.1 Novel Analysis Approach

To address the aforementioned shortcomings, this paper introduces a novel dynamic analysis approach. The goal of our analysis is to precisely track the flow of sensitive data as it is processed by the web browser and any loaded BHOs. By monitoring the actual information flow, we can answer the question of which sensitive data is collected by a spyware component. For example, we can determine whether the spyware only records the URLs that a user navigates to, or whether parts of the visited web pages are read as well. In addition, we can determine how this information is eventually leaked. For example, data could be sent directly over the network, or first stored in a file that is later retrieved by an independent spyware process. Moreover, some spyware components are equipped with a list of URLs. Whenever the user enters a URL, it is compared to all entries in this list. When the URL matches, the BHO triggers certain actions (e.g., display an advertisement in a pop-up window). By monitoring how sensitive information is processed, such checks can be identified. In some cases, it is even possible to reconstruct the static URL list.

Dynamic Taint Analysis. Our dynamic analysis uses tainting to track the flow of sensitive information as it propagates through the system. Tainting refers to a process in which data of interest is first labeled and then tracked as it is processed by the system. With our dynamic analysis, sensitive data such as URL and web page information is tainted. Then, we track the use of this data by the Internet Explorer and its loaded BHOs. When a BHO attempts to leak any sensitive data outside of the address space of the browser (e.g., by writing data to disk or sending it over the network), this action is recorded and the component is classified as spyware. This is because according to our definition of spyware, the leaking of sensitive information is considered malicious.

Our taint analysis takes into account both data dependencies and control dependencies. A data dependency captures the fact that the result of an operation (or assignment) depends on its source operands. However, information flows can also be introduced when the execution of an operation depends on the condition of a particular variable. In this case, there is a dependency between the result of this operation and the variable that controls whether it is executed or not. Current spyware programs can be detected by only taking into account data flow dependencies. However, it is easy to develop a spyware BHO that uses control flow dependencies to propagate tainted values in a way that is not captured by data flow dependencies (an example is shown in Section 3.2). In this fashion, tainted values can be laundered and detection is evaded. To address this threat, we believe that it is necessary to stay ahead of spyware authors and already consider control dependencies.

In addition to the precise tracking of sensitive data within the Internet Explorer, we are also interested in following this data once it has left the browser's address space. The fact that tainted information was leaked is sufficient to classify a component as spyware, but it is usually helpful for an analyst to be able to further track the information flow. For example, when data is written into the memory image of a spyware helper process, the additional information that this process later sends the data to a remote server would be valuable. To track such inter-process communication and data flows, we perform whole system analysis.

Operating System Awareness. One problem for our analysis is that it needs to distinguish between actions that are performed by the Internet Explorer and those that are performed by its browser helper objects; a problem that is complicated by the fact that the browser and its components are executing in the same process. The distinction is necessary to correctly attribute sensitive information flows either to normal browser operation or to malicious activity of a spyware component. Otherwise, it would not be possible to differentiate between the Internet Explorer writing a page to its temporary cache directory or a spyware saving the same information to a hidden log. A similar problem arises when a URL is written to the browser's history file, a normal operation performed by the Internet Explorer. To summarize, the mere fact that sensitive information is written out of the address space of the Internet Explorer is *not* sufficient to characterize a BHO as spyware. The BHO is spyware only when it initiates the sensitive information flow. To distinguish between sensitive data processed by the Internet Explorer and sensitive data processed by a BHO, our analysis requires a view that is aware of operating system processes and their loaded components.

Browser Session Recording and Replaying. A fundamental challenge faced by dynamic analysis approaches is test coverage. When exposing a component to a set of test cases, one cannot be certain that these tests cover the complete functionality. As a result, it is possible that some interesting behavior is not observed. In our context, this could lead to false negatives. To increase the coverage of our analysis, we have to ensure that we expose a BHO to realistic and sufficient user interaction. To this end, we developed a test system that records the actions of a user who is surfing the web. These actions include navigation to web pages and filling in form fields. Later, during our analysis, a recorded browser session can be

replayed to the BHO. The goal is to have the Internet Explorer visit a large number of pages with different content such that a spyware component will eventually trigger and reveal its malicious behavior. This allows us to test and classify samples without manual intervention.

3 System Design & Implementation

3.1 System Overview

Our dynamic taint analysis is built on top of Qemu [1], a generic and open source system emulator. Using Qemu's emulation of an Intel x86 system, we installed Windows 2000 as guest operating system (with no service packs). The choice of Windows and the Intel x86 architecture is motivated by the fact that our analysis focuses on spyware components that are implemented as BHOs for the Internet Explorer. An overview of the system and the analysis process is shown in Figure 1.

When an unknown BHO is analyzed, it is first installed on the guest OS. Then, the Internet Explorer is launched, loading the BHO component on startup. Also, the test generator is started. The task of the test generator is to simulate a surfing user by replaying a previously recorded browsing session. When sensitive data (such as a URL that the test generator navigates to) enters the Internet Explorer process, it is marked as tainted. From this point on, the taint engine tracks how the information is processed by the browser and the BHO. To be able to distinguish between actions by the Internet Explorer and those by the BHO, the taint engine differentiates between code that is executed by the Internet Explorer and code run on behalf of the BHO. The taint engine also monitors when (and where) tainted data exits the address space of the browser. When the Internet Explorer writes out tainted data because of regular browser activity, the flow is recorded as benign. When tainted information leaks because of activity on behalf of the BHO, the information flow is recorded as malicious. In this case, the analysis engine classifies the BHO as spyware.

To keep track of the taint status of data processed by the system, we introduced a shadow memory. This shadow memory holds one byte for each byte of emulated physical memory, and also covers the eight general purpose registers of the Intel x86 CPU. The decision to use one byte for each byte of the main memory and the registers allows us to not only record whether a certain location is tainted or not, but also to assign different taint labels to each location. This assignment is helpful in tagging data elements with different taint labels depending on their origin, or to distinguish between data that is processed by the Internet Explorer and data that was touched by the BHO. To propagate taint information, we had to extend Qemu's micro instructions accordingly.

3.2 Dynamic Taint Propagation

Data Dependencies. Tainting allows to tag data elements of interest and track their propagation throughout the system. Similar to a number of previous systems that use taint propagation [2, 6, 7, 22, 23], our taint analysis is capable of tracking *data dependencies*. To this end, the taint engine marks all bytes of the output of an operation as tainted whenever any byte of any input operand is tainted. This correctly propagates taint information in those cases in which a tainted value is used as source operand in an arithmetic or logic operation, or on the right-hand side of an assignment. Note that operand values can be either taken from processor registers or fetched from memory. Unfortunately, the propagation rule outline above does not take into account the taint status of a value that is used to calculate the *address* of an operand, as only the taint status of the operand itself is relevant. This can lead to problems when tainted input is used as an index into a table (or an array). In such cases, the result of a table lookup is not labeled as tainted, and its relationship with the input value is lost. Interestingly, such lookup operations are frequently used for converting user input (for example, to convert ASCII to Unicode characters in Windows, or to map keyboard scan codes to keystrokes in Linux). Thus, we also taint the output of an operation whenever a tainted value is involved in the address computation of a source memory operand (regardless of the taint status of the memory operand that is referenced).

Control Dependencies. A system that can handle only data flow dependencies provides a spyware author with a simple opportunity to evade detection. Figure 2 provides an example that illustrates the problem. On the left side of this figure, a code fragment is shown where two conditional branches are used to "assign" the value of the tainted variable `t` to the variable `clean`, assuming that `t` only takes on the values 'a', 'b', or 'c'. Because there is no direct data dependency between `t` and `clean`, the variable `clean` will hold the same value as `t` after the execution of the code fragment, but it is not tainted.[1] Clearly, this approach can be generalized to launder arbitrary information. To mitigate this weakness, our taint analysis also considers *direct control dependencies*. To correctly handle control dependencies, the result of an operation has to be tainted whenever the execution of this operation depends on the value of a tainted variable (e.g., when an operation is guarded by an `if`-branch that tests a tainted variable, or when an operation is executed in a case branch when the corresponding switch statement used a tainted argument). Note that the result of any such

[1] Note that this example is shown in C code, although our system operates directly on x86 binaries.

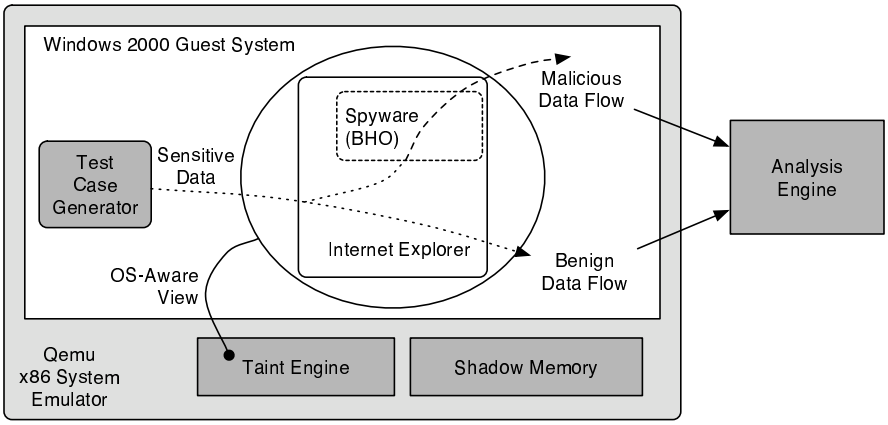

Figure 1: System Overview.

operation is tainted independently of the taint status of the source operands. Revisiting the example shown in Figure 2, and using a system that can track direct control dependencies, we observe that variable `clean` will be tainted whenever `t` is tainted. This is because the execution of any assignment operation depends on the value of `t`, and thus, there are control dependencies between `t` and the results of these assignments.

To handle control dependencies, the taint engine examines all conditional branch instructions that are encountered during execution. When such an instruction has at least one tainted operand, the taint engine has to identify all instructions whose execution is conditionally dependent on the result of the branch.[2] Using an analogy from imperative programming languages, the task is to determine the scope of a conditional branch such that this scope encloses all instructions that depend on the outcome of the branch. The results of all operations that are then executed within this scope need to be tainted.

To find all instructions that belong to the scope of a branch, static analysis is necessary. The reason is that we have to find the first instruction in the control flow graph that is executed independent of whether the conditional branch is taken or not. More formally, this instruction is the immediate post-dominator of the branch operation in the program's control flow graph. Intuitively, it is the point where the two possible execution paths after the branch operation merge. At this instruction, the scope of the branch statement ends, and it is no longer necessary to taint the results of all operations. To find this instruction, two (or more) possible execution paths need to be explored. This can only be done statically, because there is only a single path executed dynamically. As an example of a branch instruction with its corresponding scope and post-dominator node, consider the right side of Figure 2. The graph represents the control flow of the code fragment on the left. It can be seen that the last node (where 0 is assigned to `x`) is the point where the two branches of the first `if`-statement merge.

The first step in finding the instruction that ends a scope is to build a (partial) control flow graph (CFG) of the program. The control flow graph starts at the branch instruction and needs to cover all paths until the merging point. Of course, this merging point is not known *a priori*. Thus, we extend the control flow graph until we reach instructions where the disassembly process cannot continue (typically, these are function return instructions, but also indirect jumps whose target cannot be resolved statically). To build the CFG, our system uses a recursive disassembler [17]. Because we do not continue the disassembly process after instructions whose targets we cannot determine with certainty, we obtain a control flow graph that contains only instructions that are reachable during runtime. This assumes that the code is not self-modifying. Fortunately, our dynamic analysis can easily identify attempts of a BHO to modify its own code by monitoring the target of memory write operations and ensuring that no code regions are altered. Any attempt of a BHO to modify its own code is flagged as malicious.

The fact that our partial control graph is guaranteed to contain only instructions that are reachable during runtime is important, as there are a number of ways in which the attacker could attempt to thwart static analysis and the disassembly process using code obfuscation [20]. Because we use a simple analysis approach that explores paths only as long as successor instructions can be iden-

[2] Actually, the situation is a little more complicated with the x86 instruction set. The reason is that conditional jumps do not have operands themselves, but use the processor flags set by a previous compare operation to decide which branch to take. Thus, our system links the execution of an instruction that compares (or tests) tainted data with a subsequent conditional jump to identify those branches that operate on tainted data.

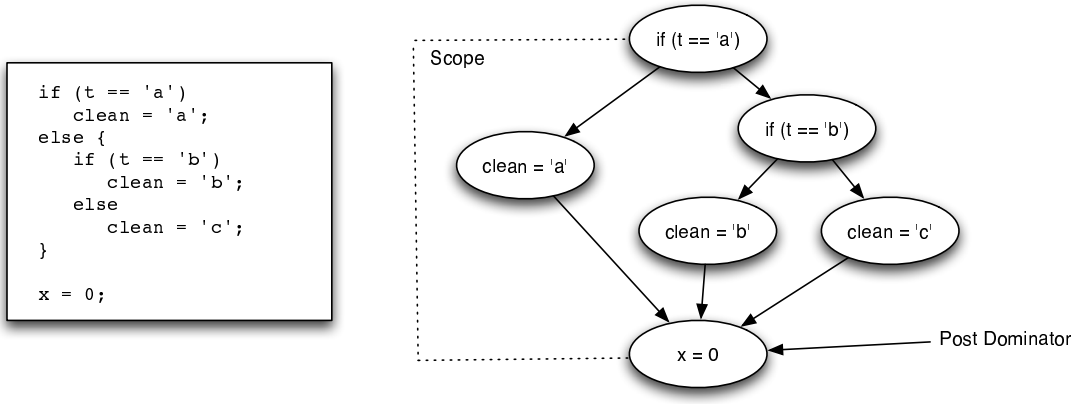

Figure 2: Control Dependency and Scope.

tified with certainty, our static analysis step is immune to these obfuscation techniques. This is a major improvement over the significantly more complex static analysis described in our previous work [14], where the complete binary is disassembled and analyzed. Of course, the control flow graph that we extract is not necessarily complete. This is typically due to the problem of indirect jump or call instructions whose targets cannot be resolved statically (e.g., in the presence of function pointers or jump tables). We recognize and handle this problem in the following step.

When the disassembler finishes, it has extracted a control flow graph that contains all instructions that are definitely reachable starting from the branch instruction. We then apply the well-known Lengauer Tarjan [19] algorithm to compute a dominator tree for this graph. This dominator tree allows us to the find the node that immediately post-dominates the branch instruction, and thus, represents the instruction that ends the scope. However, as mentioned before, the control flow graph might be incomplete. In this case, it is possible that there are multiple nodes that post-dominate the branch instruction. Hence, whenever this situation occurs, we take a safe approach and assume that the BHO contains code to thwart analysis, and label the BHO as malicious.

Note that our technique to track control dependencies is conservative, as it taints the results of all operations executed within a tainted scope. Thus, it is possible that our system introduces incorrect dependencies between variables and raises false positives. To address this issue, we only track control flow dependencies when executing code inside the BHO. The rationale is that the attacker can only control the BHO, and we assume that the Internet Explorer itself does not contain code that deliberately attempts to hide data dependencies via the control flow. Also, observe that our static analysis is only invoked when the dynamic analysis actually encounters a conditional branch instruction with tainted operands.

Previous systems that use data tainting were not able to take into account control dependencies because this conservative propagation policy typically resulted in too many tainted values (a phenomenon often referred to as taint label explosion). The fact that we only track control flow dependencies when executing code inside the BHO is very helpful to ensure that our system does not suffer from this problem. In addition, we observed in our experiments that tainted data is only used very rarely in control flow decisions, further mitigating the problem of label explosion. However, there could be cases in which BHOs process tainted data such that many more control flow decisions are based on tainted input. An example would be a Java interpreter that executes Java code loaded by the browser. In such cases, it is likely that we also suffer from memory regions that are incorrectly tainted, leading to false positives. Fortunately, such functionality is typically not realized in BHOs.

Untainting. In addition to a mechanism that flags registers and memory locations as tainted, there must also be a way to clear their taint status. In the simplest case, a register or a memory location loses its taint status when it is overwritten with an untainted value. Immediate instruction operands (constant values) are always considered untainted. In addition, one has also to take into account constant functions, which denote code sequences that always produce the same output regardless of their input. For example, the following operation is used frequently on the Intel i386 architecture to set a register to zero.[3]

```
xor %eax, %eax; // %eax = %eax^%eax
```

[3]RISC chips, in contrast, typically provide a register that is hardwired to 0.

Because this instruction always sets the register to zero, the output should not be tainted, even when the input `%eax` is tainted. Note that another variant of the same function uses a `sub` instruction instead of the `xor`. We support simple constant functions that consist of a single `xor` or `sub` instruction. However, one needs to be aware that more complex versions of constant functions may exist that are not detected. In these cases, the system could incorrectly label certain data elements as tainted, which might result in false positives. In our experiments, however, we did not observe any problems stemming from this limitation.

3.3 Bridging the Semantic Gap

A whole system emulator, such as Qemu, only provides a hardware-level view of the guest system, including physical memory, CPU registers, and I/O device status. However, for the purpose of meaningful analysis, a view at the operating system level is necessary. In other words, we have to bridge the semantic gap between these two views. In particular, we need to address two problems: (1) identifying operating system processes, so we know when the Internet Explorer is executing; (2) distinguishing what actions are performed in the context of the BHO. These problems are not entirely trivial, especially for a closed source operating system such as Microsoft Windows.

Identifying Operating System Processes. To identify operating system processes, we leverage the mechanism that Windows uses for virtual memory management (on the x86 architecture). In particular, we make use of the fact that for the current process, the `CR3` processor register stores the physical address of its page table directory. This address is unique for all running processes. To obtain the page directory address that belongs to a process, we exploit the facts that Windows stores this address as an attribute of the `EPROCESS` structure, and that a pointer to this is always mapped to the same, well-known virtual address.

Of course, our analysis has to determine the `CR3` value for the Internet Explorer before it can execute any user mode instructions. We decided to hook the Windows system call that is responsible for creating new processes (called `NtCreateProcess`). Hooking is performed by checking the processor's instruction pointer at the beginning of each operation and comparing it to the address of the `NtCreateProcess` function. This address can be obtained from the kernel symbol table that comes with each Windows distribution, usually for debugging kernel device drivers. Whenever a new process is created by invoking `NtCreateProcess`, we check the process list for the new entry and compare its name to the program(s) that we wish to monitor.[4] When the names match, the `CR3` value is extracted and the process is monitored.

Identifying Actions in the Context of the BHO. The ability to identify Windows operating system processes allows us to distinguish the operations that are performed by the Internet Explorer from those of other processes. Unfortunately, this is not sufficient. The reason is that we also need to determine which code *within* the Internet Explorer process is run because of regular browser activity, and which code is executed on behalf of a BHO. As discussed in Section 2, this is important to correctly attribute monitored behavior either to the Internet Explorer or to one of the loaded components.

Obviously, all instructions that are located directly in the code segment of the BHO are considered to run on its behalf. However, we also wish to cover the case in which the BHO code calls another function that is located elsewhere (in the Internet Explorer or any other loaded library). To correctly identify all instructions that are executed *in the context of the BHO*, the following algorithm is used:

1. Whenever execution is transferred from the Internet Explorer to the code of the BHO, record the value of the current stack pointer. This transition is recognized by observing the execution of an instruction that is located in the code segment of the BHO. Then, goto Step 2.

2. For every further instruction, check if the current value of the stack pointer is below the value stored in Step 1. If so, the instruction is executed in the context of the BHO; else it is not, and we restart with Step 1.

The rationale behind this technique is as follows: Whenever code in the BHO is called, we record the location of the current stack frame on the stack. When the BHO itself calls other functions, additional stack frames are pushed onto the stack. Because the stack grows towards smaller addresses on the x86 architecture, the stack pointer remains below the stored stack pointer. Only when all functions have returned and the BHO invokes a return operation, the stack frame of the BHO is popped from the stack and the value of the stack pointer exceeds the one stored. One problem that complicates our approach is the presence of threads. The reason is that, for each thread, the operating system allocates a different stack region in the process' virtual address space. Thus, the value of the stack pointer is only meaningful in the context of a certain thread, and switches between

[4] To be precise, we check the process list when `NtCreateProcess` returns, because at the time the function is called, the `EPROCESS` structure does not exists yet.

threads have to be identified. To do so, we examine the current identifier of the executing thread (which is located at a well-known address in the KTHREAD structure) whenever execution returns from the kernel.

Based on the knowledge of which code is executed in the context of the BHO, we now have the means to differentiate between data that is written by the Internet Explorer and data that is leaked on behalf of the BHO. To this end, we extend our taint propagation policy: Whenever an instruction that is executed in the context of the BHO writes tainted data, the label of this data receives a *suspicious* flag. From now on, this data is clearly identified as sensitive data that has been processed by the BHO. Whenever tainted data with a suspicious flag is later processed by other instructions, even when these instructions are not run on behalf of the browser component, they retain their flag. Also, whenever any operand of an instruction has the suspicious flag, the output is labeled suspicious as well. Data labeled with the suspicious flag must no longer leak from the Internet Explorer process. Otherwise, the BHO is considered spyware.

Evasion. A spyware author who is aware of our technique to identify actions on behalf of the BHO might attempt to evade detection. The goal for the attacker is to leak sensitive information, but let it appear as regular browser activity. One possibility is to modify the code of the Internet Explorer such that malicious actions are performed when regular browser code is executed. Another possibility is to inject new code into the address space that our analysis does not associate with the BHO. Then, some code pointer in the Internet Explorer must be redirected to point to this injected code region. Both threats can be countered by using the fact that our analysis engine has complete control of the execution of the browser and the BHO. This allows us to ensure that only those instructions are executed that are in known code regions. To prevent a malicious component from altering the contents of legitimate code regions, we can ensure that the BHO cannot remove their memory write protection (by hooking the appropriate system call). Moreover, note that evasion is not possible for the attacker by executing a statement in the BHO that pushes the stack pointer above the limit stored in Step 1. The reason is that, in this case, the instruction following the stack pointer modification is again recognized as belonging to the code segment of the BHO. Thus, the new value of the stack pointer is saved and execution continues on behalf of the BHO.

3.4 Detection & Analysis

In this section, we discuss when information is tainted and then explain when and where the use of tainted data is suspicious.

Taint Sources. A taint source can be any part in the system that precisely defines a portion of data that we wish to track. On one hand, this can be memory locations where the hardware stores information (such as buffers that hold network packets or keyboard scan codes). On the other hand, we can taint the arguments of certain functions. Currently, we use two taint sources. One taint source is used to taint all URL strings of the pages that a users visits. This can either happen by typing the URL directly into the browser's address bar or by clicking a link on a page. The other taint source taints the information that the Internet Explorer receives in response to its requests. This includes both HTTP pages and files that are downloaded. The reason for selecting these taint sources is that we consider both the URL and the content of the page as sensitive information. Whenever this information is leaked on behalf of a browser component, this component is classified as spyware. Note that it would also be interesting to taint the data that a user enters into web forms. This would allow us to identify BHOs that attempt to steal user passwords (and other private information). Including additional taint sources is quite straightforward, and as part of our future work, we are planning to taint user input as well. To taint the URLs, we hook the `Navigate` function of the Internet explorer and taint the string argument that represents the URL. Note that the hooking of an Internet Explorer function works similar to the hooking of a system call.

To taint the data that is retrieved by the Internet Explorer, we mark the return data buffer of the Windows equivalent of the Unix `receive` system call, which is called `NtDeviceIoControlFile`. Whenever this function is invoked, we first wait until it returns and then consult the return code. When data was successfully received, the appropriate buffer is tainted. We assign different taint labels to the URL and the page data to be able to distinguish between them.

Taint Sinks. When input data becomes tainted, taint information is automatically propagated by our system. The goal is to determine whether this data is eventually used in a fashion that would reveal spyware-like behavior of a browser component. According to our definition of spyware, such behavior is present in situations where tainted data is leaked by the BHO. Recall that we are not interested in writes of tainted data in general. Only a flow of information that is explicitly labeled suspicious leads to the classification of a component as spyware To detect such flows, our system monitors the interfaces that can be used to write information out of a process for the presence of suspicious information. Currently, we monitor communication over the network, writes to the file system, accesses to the registry, and communication with other processes via shared memory. While we believe that our set of sensitive sinks is comprehensive,

it is possible that we have missed a vector that a BHO could use to leak sensitive information. However, adding additional vectors to our system is straightforward, and merely a matter of monitoring the appropriate arguments of the relevant system calls.

To monitor whether information is leaked over the network, we monitor the data buffer argument to the system call `NtDeviceIoControlFile`. This system call acts as a funnel for higher-level network calls and is responsible for receiving and sending data over both UDP and TCP. To differentiate between the different roles of `NtDeviceIoControlFile`, its first parameter must be evaluated. To check for writes to files, we monitor the `NtWriteFile` system call. Also, we hook the `NtCreateFile` function to be able to later associate the file name with the file handle that is used for file access calls. Similar hooks are inserted to monitor the system calls that are responsible for writing keys and values to the registry. Note that it is typically not sufficient to check the arguments of the `NtWriteFile` system call to cover all file accesses. The reason is that files can also be memory mapped. In this case, (parts of) the contents of a file are mapped into the virtual address space of a process. Then, the file can be accessed by regular memory read and write operations. To detect tainted data that is written into memory mapped files, the system call that performs the mapping is intercepted. Whenever a monitored process maps a file into its address space, the corresponding memory regions are recorded. On any subsequent write to these monitored ranges, our analysis can derive that a file was written. For this, it is necessary to check the target addresses of every write operation. This check is performed as part of the taint propagation logic.

Note that it might be overly conservative to consider as suspicious the fact that a BHO saves data to disk. Although we have not encountered legitimate BHOs in our experiments that write URLs or web page data to a file, it is conceivable that certain legitimate applications might do so (e.g., bookmark managers). In this case, the system could be extended so that it does not immediately report a BHO as spyware that writes to disk, but instead continues to monitor what happens to the sensitive data. When, at one point, another process accesses the file, reads the sensitive information, and sends it over the network, the BHO would be classified as spyware. Otherwise, the write would be considered benign. While this extension has not been implemented, our system already supports this kind of analysis in principle (i.e., the tainting engine can track tainted data in multiple processes, and we record which bytes are tainted in a file when sensitive data is saved).

Detailed Analysis. To improve the quality of our analysis reports, we also record in more detail how code that is executed in the context of a BHO handles tainted data. One piece of information that we are interested in is whether the BHO reads tainted data at all. If a BHO never touches any sensitive information, our confidence increases that the component is not spyware. On the other hand, if tainted data is accessed, we are particularly interested in those reads where subsequent bytes of the input are accessed. This could indicate that parts of the sensitive data are copied for further processing.

Another interesting indicator to better understand the behavior of spyware is whether the monitored component performs compare operations where one of the operands is tainted. For example, when spyware compares the current URL with its own list of interesting URLs, or when the page is scanned for the presence of certain keywords, we would expect to see a number of consecutive compare instructions with tainted operands that are executed in the context of the BHO. By recording which values are compared, it is even possible for a human analyst to derive which keywords or URLs the spyware is searching for. Deriving more information about the values that the BHO is looking for can be done especially well when an x86 string compare instruction such as `cmps` is used. In this case, the operands of the instruction point to the two complete strings that are compared. Also, we check for sequences of compare operations that refer to consecutive memory locations. This allows us to identify (some) string matching routines that perform byte-by-byte comparisons.

Automated Browser Testing. When using dynamic approaches, it is very difficult to be certain that the complete range of functionality of a component is analyzed. Thus, the number of web pages visited and the interaction during the browsing phase is an important factor for the quality of the results. Clearly, requiring the human analyst to manually visit pages and fill out forms is tedious for large test sets and also prevents the system from being integrated into an automated tool-chain for spyware analysis. To address this problem, we developed a browser testing tool that allows us to automate our analysis by mimicking the surfing behavior of users. The tool can record the web interactions of a user and later "replay" them to make the Internet Explorer visit a large number of web pages without manual intervention. It also supports user input that is inserted into form fields.

The browser automation tool consists of two components: The first component is a Mozilla Firefox extension (i.e., plug-in) that records the pages a user has visited and the input she has entered into forms. The captured data is dumped into a file so that it can be later replayed. The second component is a Microsoft Windows application that first reads the information from the capture file and then replays the surfing session to the Internet Explorer. To this end, the tool first obtains a handle to the browser. Then, it repeatedly invokes the `Navigate`

method of the browser's `IWebBrowser2` interface to visit the list of stored URLs. For every web page that is visited, the tool uses the `IHTMLDocument2` Document Object Model (DOM) interface to locate all its form elements. This allows us to automatically fill out form fields that were filled out during the recorded session (using the names of the form elements). When a form is completed, it is automatically submitted.

4 Evaluation

The goal of our system evaluation is twofold. On one hand, we wish to verify the ability of our system to classify unknown browser helper objects. To this end, we analyzed a collection of spyware and benign samples and determined the fraction of samples that were correctly identified. On the other hand, we wish to demonstrate that our system provides comprehensive reports that allow a human analyst to quickly and in detail understand the behavior exhibited by a spyware component. To this end, we selected a few samples and provide a more detailed description of our findings.

4.1 Batch Analysis

To verify the ability of our system to distinguish between spyware and benign components, we compiled a test set that contained 21 spyware and 14 benign browser helper object samples. All spyware samples were provided by an anti-virus vendor. For the benign samples, we downloaded a number of different browser helper objects from various shareware sites. Of course, we made sure that these components were indeed benign by carefully checking both anti-spyware vendor and software review web sites. The benign samples were chosen from a variety of application areas. Tables 4 and 5 in the Appendix list and describe the samples that we used during our experiments. It is often difficult to determine the name of a malware sample as these names are not unique and may vary between anti-spyware and anti-virus vendors. When naming the malware samples, we used the information we were given by the anti-virus vendor.

Using our test set, we performed a batch analysis. That is, for each sample in the set, the following steps were carried out: First, the sample is loaded into the analysis environment. Then, it is installed using the Windows `regsvr32` utility. During this installation process, BHOs register themselves with the Internet Explorer so that they are automatically loaded when the browser is launched. After that, the Internet Explorer is started and the automated test generator replays a previously recorded browsing session. For this test session, we surfed to 50 web pages. The pages were chosen from different web site categories (such as adult, news, or wall paper) to provide variety. For these categories, we decided to use the ones presented in [21], a paper in which the authors analyzed different sites for the presence of spyware. The browsing session also contains typical user interactions on the sites that might be of interest for spyware, such as Google searches for free pornography, news browsing, and visits to music sites. At the end of the test, the browser is closed, and the analysis engine analyzes the log file. If the log contains any indication that sensitive information was leaked on behalf of the component under analysis, it is classified as spyware.

The sample set we used does not contain toolbar-based spyware. This is because our automated testing infrastructure currently does not support toolbars. Toolbars introduce additional GUI elements into the web browser that are not present when the initial test session is recorded. Thus, our testing tool cannot invoke any toolbar functions that require to click on GUI elements installed by this toolbar.

Table 1 shows the results for the batch analysis of our test set. The results demonstrate that all spyware components were correctly identified. In our experiments, none of the spyware samples made use of control flow evasion techniques. Thus, all malicious BHOs can be correctly detected taking into account data dependency information only. However, writing malicious code that makes use of control flow evasion is quite simple. To demonstrate that, we developed a proof-of-concept BHO that uses a sequence of `if`-statements (similar to the code shown in Figure 2) to leak sensitive information. Using only data dependencies, this BHO is classified as benign. When control dependencies are included, our system correctly identifies the malicious data transfer.

Also, most benign samples were correctly classified. However, in accordance with the results reported in our previous paper, we found two benign samples that actually *do leak* sensitive data and thus, exhibit spyware-like behavior. In one case, closer analysis revealed that no sensitive data was sent to a third party (false positive). In the other case, however, sensitive data was indeed sent to the distributor of the BHO, although very infrequently (suspicious case). In Table 2, we show a breakdown of the different mechanisms that the analyzed spyware samples used to leak sensitive information. These results underline that spyware BHOs in the wild actually make use of a variety of techniques to send collected information back to the spyware distributor.

Performance. Even though Qemu is a fast system emulator, the complete analysis of an unknown BHO with the replaying of a browsing session can take several minutes. Thus, our system is mainly intended for analysts that have to understand and classify unknown BHOs. In addition, our tool could also be used as the analysis component in an automated spyware collection system.

	Spyware	False Negative	Benign	Suspicious	False Positive	Total
Spyware	21	0	-	-	-	21
Benign	-	-	12	1	1	14

Table 1: Results for batch analysis.

Network	File System	Registry	Shared Memory	Total
11	1	3	6	21

Table 2: Different mechanisms used by spyware to leak sensitive data.

	Min.	Max.	Average
Native Windows	0.6	2.9	1.9
Qemu	1.8	6.1	3.6
Modified Qemu	17.3	79.4	35.7

Table 3: Performance overhead.

For a more detailed overview of the incurred performance penalty, refer to Table 3. This table presents the times (in seconds) that were necessary to load web pages on our test machine (Pentium IV, 2.4 GHz with 1 GB RAM); for Windows running natively, on an unmodified Qemu emulator, and on Qemu with our modifications. These numbers show the average, minimum, and maximum load times for the web pages used in our experiments. The time required to load each page increased almost linearly with the size of the page, and the resulting work necessary for rendering. It can be seen that our system incurs an average slowdown of about a factor of ten when compared to an unmodified Qemu, with an additional factor of two when compared to native execution. In a worst-case scenario, when all memory is tainted, the slowdown could significantly increase. The reason is that all conditional branches would operate on tainted data, thus triggering the static analysis step. Fortunately, as shown by our experiments, only a small fraction of memory is typically tainted, and BHOs rarely used tainted data in control flow decisions.

Although the focus of this work was not on performance, note that this overhead could probably be improved. For example, about 30% of the overhead of our system is caused by checking, for each basic block, whether the first address corresponds to a function that represents a sensitive source or a sensitive sink. Instead of checking the instruction pointer for each basic block, the interesting code parts could be memory-protected. Whenever these code regions are later accessed, a fault is raised that can be used by our system to determine that an interesting function was called. The remaining overhead of 70% is a result of the logic that propagates the taint labels. Again, this number could be significantly reduced, for example, by selectively switching between emulation and virtualization, as discussed in [11]. The memory overhead of our system is basically constant, and dominated by the size of the shadow memory, which requires one byte for each byte of emulated physical memory. Because we reserved 128 MB of memory for Qemu, the size of the shadow memory was 128 MB as well.

4.2 Detailed Analysis

The following paragraphs describe briefly the information that is contained in our analysis reports. In addition, we discuss in more detail the false positive, the suspicious sample, and three representative spyware BHOs. This discussion underlines the richness and the level of detail of the reports that are automatically generated by our dynamic analysis.

Reports After our system has analyzed a BHO, a report is generated that describes how this BHO has handled sensitive data. For every byte of sensitive input that is accessed by the BHO, we show the value and the origin (i.e., sensitive source) of this byte. Consecutive labels are combined so that accesses to strings appear as such in the output. Of course, multiple reads of the same data are suppressed and the access is shown only once. Whenever sensitive data is used in comparison operations, we show the values and labels of those bytes involved in the comparisons, as well as the values that the input is compared to. Again, compares of multiple, consecutive labels are shown in a combined form (as discussed in Section 3.4). Finally, whenever a tainted byte is leaked via a sensitive sink, the type of the sink and the leaked bytes are displayed. In all cases, information is only displayed when the tainted data has been processed by the BHO under analysis.

In general, these reports present a significant improvement compared to our previous system [14], which could

only label a BHO as spyware or benign. Previously, a tedious and time-intensive manual process was necessary to understand why an alert was raised, for example, in case of a false positive. Furthermore, the important information about the type of data that was leaked (such as the URL, or part of the page) was not available.

False Positive. The false positive listed in Table 1 is caused by the `PrivacyBird` BHO. This component implements the client side of a privacy management standard defined by the Platform for Privacy Preferences Project (P3P). The P3P standard specifies a mechanism for users to control the disclosure of their personal information on web pages. To this end, the `PrivacyBird` BHO has to retrieve a privacy policy file (which is located at w3c/p3p.xml) for every web page. To determine the server that hosts the privacy file for the current page, the BHO reads the URL and extracts the domain name. Then, the domain name is combined with the static path to the privacy file. The resulting URL is then used to fetch the privacy file. Because this contains a part that is tainted (the domain string), we detect a malicious information flow. Our analysis shows that the BHO reads the URL of every page that is visited. In addition, we can quickly confirm that for every "malicious" request, the server that is contacted is equivalent to the domain string that is tainted in this request. This information allows a human analyst to gain confidence that the `PrivacyBird` component is indeed not sending any sensitive information to a third party. Note that it would also be possible to specify a policy that classifies as benign all information flows in which information about a URL or a page is transferred back to the host from which they are loaded. The reason is that in such cases, no sensitive information is revealed to a third party. When this policy were in effect, the `PrivacyBird` BHO would not have raised a false positive.

Suspicious Sample. The suspicious sample was the `LostGoggles` BHO, a component that embellishes Google search results by adding pictures to the search hits. To this end, the BHO downloads a piece of JavaScript code from the author's web server when a search request is sent to Google for the first time. Subsequently, this JavaScript snippet is inserted into every result page returned by Google. When the script is downloaded, the BHO sets the referrer header in the HTTP request that fetches the JavaScript file. This referrer header contains the URL of the Google search that was issued before. Thus, `LostGoggles` does leak possibly sensitive user information, although the data is probably sent inadvertently and only once when the script is obtained.

Interestingly, both the web pages of `PrivacyBird` and `LostGoggles` emphasize that the components are not spyware, even though they do send information over the network (which is behavior typically associated with spyware). Using our tool, a detailed analysis can help to provide more evidence to decide whether a component is using data as described.

Spyware Samples. `Zango` advertises its products (such as games or screen savers) as ad-supported freeware. During our analysis, we determined that the `zangohook.dll` BHO, which is shipped with the company's instant messaging client, is spyware. More precisely, our system detects that whenever a web page is visited, the BHO reads the current URL and copies it to a previously opened shared memory section. From this shared memory section, the data is later read by the spyware helper process `zango.exe`. The `Zango` example underlines the importance of monitoring shared memory areas that can be used by a BHO to write out data to other processes. Also, it demonstrates the usefulness of whole system analysis, which allows us to follow the sensitive data to the spyware helper process.

The `e2give` BHO reads the URL of every site that is visited and compares it to a list of URLs stored in the BHO. This check is implemented by consecutively matching the current URL against every item in the BHO's URL list. As our analysis checks for compare instructions that involve tainted operands, the log file contains the complete list of URLs that the BHO checks against. If any of the requested sites is found in the list, the BHO redirects this request to a different server. In this case, the original URL is passed as an argument to the redirected GET request. This constitutes a flow of sensitive data that is correctly identified by our system. The `e2give` BHO is interesting for two reasons. First, it demonstrates the ability of our tool to extract lists of URLs that a spyware monitors. Second, it underlines the importance of test coverage. When none of the URLs in the BHO list were visited, the sample would be misclassified as benign (as sensitive data is only leaked in case of a match).

Finally, our analysis detected that the `stdup.dll` BHO (i.e., Borlan) submits the URLs of all visited pages to a remote server. Thus, the sample was classified as spyware. This BHO is interesting because a scan with the latest versions (at the time of writing) of the commercial anti-spyware tools AdAware [18] and SpyBot [15] yields no detection. This demonstrates that our analysis is capable of detecting previously unknown spyware samples.

5 Related Work

Malware Detection. To combat the increasing spread of spyware, a number of commercial solutions have been developed. For example, both AdAware [18] and SpyBot [15] are popular tools that are able to remove a large

number of spyware programs. The problem with existing spyware detection tools is that they use signatures to detect known spyware instances. Thus, they require frequent updates to their signature database and cannot identify previously unseen samples.

To address the limitations of signature-based malware detection, researchers have recently proposed behavior-based techniques. These techniques attempt to characterize a program's behavior in a way that is independent of its binary representation. By doing this, it is possible to detect entire classes of malware. An example of using behavior characterization to detect malicious code is Microsoft's Strider Gatekeeper [26]. This tool monitors auto-start extensibility points (ASEPs) to determine if software that will be executed automatically at startup is being surreptitiously installed on a system. In [4], the authors characterize different variations of worms by identifying semantically equivalent operations in the malware variants. A similar approach is followed in [16], where the behavior of kernel-level rootkits is modeled.

In a previous paper [14], we introduced a behavioral approach to detect spyware. For that paper, we used the same underlying characterization of spyware as in this work (that is, a BHO is considered spyware when it leaks sensitive information). However, the analysis techniques are completely different. For the former paper, we mainly relied on binary, static analysis to identify code paths in the BHO that can leak information. A small dynamic component was used to find the entry points for the static analyzer. In this paper, we developed (from scratch) a dynamic taint analysis system that supports data and control dependencies and is operating-system aware. Using our new system, we can automatically generate rich reports that precisely identify which sensitive information a BHO touches and where it is eventually stored. This was not possible with our previous system. Also, we removed our reliance on complex binary static analysis, which is vulnerable to code obfuscation and evasion.

Virtual Machines and Taint Analysis. For this paper, we use a virtual machine (Qemu) to monitor the behavior of unknown browser helper objects. This has the benefit that our analysis runs in complete isolation from the samples that are examined, making it much harder for spyware to detect the presence of our system. Other researchers made similar use of virtual machines to detect and prevent intrusions [9, 12] and to analyze attacks [8, 13]. Also, virtual machines have been used to implement whole system analysis based on dynamic tainting. For example, a system was proposed in [2] to use taint information to track the lifetime of data. The goal was to determine the use of sensitive information by the operating system and large applications. Other researchers used taint analysis to monitor program execution for the use of tainted data as arguments to control flow instructions or systems calls [6, 7, 22, 23] (a system to perform taint propagation particularly efficient was presented in [11]). The aim of these systems is to identify exploits at runtime, and, in some cases, to create signatures for detected attacks. There are a number of differences to our work. First, we analyze malicious code that can be deliberately designed to thwart detection. Thus, it was necessary to extend our taint analysis with the capability to handle control dependencies in addition to data dependencies. Second, previous systems focus on whole system emulation only and can neither distinguish between operations performed by different operating system processes (and individual components of these processes) nor keep track of which component has accessed sensitive data. Finally, the aim of previous systems is to detect exploits, while the goal of our system is to identify spyware components and comprehensively analyze and document their behavior.

6 Conclusions

In this paper, we presented a novel dynamic analysis approach to classify unknown browser helper objects and capture their behavior. The goal of our system is to automatically identify spyware that is installed in the form of browser helper objects for the Microsoft Internet Explorer. To this end, we monitor the way that the Internet Explorer and installed browser helper components handle sensitive user information (such as the URL that a user visits or the content of the web pages that are loaded). A BHO is classified as spyware when it leaks sensitive information outside of the browser process. In addition to classification, the analysis also provides a rich and comprehensive description of the actions performed by BHOs. The experimental results on a substantial body of spyware and benign samples demonstrate the effectiveness of our approach.

Acknowledgments

We would like to thank our shepherd Andrew Warfield and the anonymous referees for their valuable feedback. This work was supported by the Austrian Science Foundation (FWF) under grant P18157, the FIT-IT project Pathfinder, and the Secure Business Austria competence center.

References

[1] BELLARD, F. QEMU, a Fast and Portable Dynamic Translator. In *Usenix Annual Technical Conference, Freenix Track* (2005).

[2] CHOW, J., PFAFF, B., GARFINKEL, T., CHRISTOPHER, K., AND ROSENBLUM, M. Understanding Data Lifetime via Whole System Simulation. In *Usenix Security Symposium* (2004).

[3] CHRISTODORESCU, M., AND JHA, S. Testing Malware Detectors. In *ACM International Symposium on Software Testing and Analysis (ISSTA)* (2004).

[4] CHRISTODORESCU, M., JHA, S., SESHIA, S., SONG, D., AND BRYANT, R. Semantics-Aware Malware Detection. In *IEEE Symposium on Security and Privacy (Oakland)* (2005).

[5] COLLBERG, C., THOMBORSON, C., AND LOW, D. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *Conference on Principles of Programming Languages (POPL)* (1998).

[6] COSTA, M., CROWCROFT, J., CASTRO, M., ROWSTRON, A., ZHOU, L., ZHANG, L., AND BARHAM, P. Vigilante: End-to-End Containment of Internet Worms. In *20th ACM Symposium on Operating Systems Principles (SOSP)* (2005).

[7] CRANDALL, J., AND CHONG, F. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *37th International Symposium on Microarchitecture (MICRO)* (2004).

[8] DUNLAP, G., KING, S., CINAR, S., BASRAI, M., AND CHEN, P. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Symposium on Operating Systems Design and Implementation (OSDI)* (2002).

[9] GARFINKEL, T., AND ROSENBLUM, M. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Network and Distributed Systems Security Symposium* (2003).

[10] HACKWORTH, A. Spyware. US CERT Publications, 2005.

[11] HO, A., FETTERMAN, M., CLARK, C., WARFIELD, A., AND HAND, S. Practical Taint-based Protection using Demand Emulation. In *EuroSys Conference* (2006).

[12] JOSHI, A., KING, S., DUNLAP, G., AND CHEN, P. Detecting past and present intrusions through vulnerability-specific predicates. In *Symposium on Operating Systems Principles* (2005).

[13] KING, S., AND CHEN, P. Backtracking Intrusions. In *Symposium on Operating Systems Principles (SOSP)* (2003).

[14] KIRDA, E., KRUEGEL, C., BANKS, G., VIGNA, G., AND KEMMERER, R. Behavior-Based Spyware Detection. In *Usenix Security Symposium* (2006).

[15] KOLLA, P. Spybot Search & Destroy. `http://www.safer-networking.org/`, 2006.

[16] KRUEGEL, C., ROBERTSON, W., AND VIGNA, G. Detecting Kernel-Level Rootkits Through Binary Analysis. In *Annual Computer Security Applications Conference (ACSAC)* (2004).

[17] KRUEGEL, C., VALEUR, F., ROBERTSON, W., AND VIGNA, G. Static Analysis of Obfuscated Binaries. In *Usenix Security Symposium* (2004).

[18] LAVASOFT. Ad-Aware. `http://www.lavasoftusa.com/software/adaware/`, 2006.

[19] LENGAUER, T., AND TARJAN, R. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS) 1*, 1 (1979).

[20] LINN, C., AND DEBRAY, S. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *ACM Conference on Computer and Communications Security (CCS)* (2003).

[21] MOSHCHUK, A., BRAGIN, T., GRIBBLE, S., AND LEVY, H. A Crawler-based Study of Spyware on the Web. In *Network and Distributed Systems Security Symposium (NDSS)* (2006).

[22] NEWSOME, J., AND SONG, D. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Network and Distributed System Security Symposium (NDSS)* (2005).

[23] PORTOKALIDIS, G., SLOWINSKA, A., AND BOS, H. Argos: an Emulator for Fingerprinting Zero-Day Attacks. In *ACM SIGOPS EUROSYS* (2006).

[24] SAROIU, S., GRIBBLE, S., AND LEVY, H. Measurement and Analysis of Spyware in a University Environment. In *Usenix NSDI* (2004).

[25] THOMPSON, R. Why Spyware Poses Multiple Threats to Security. *Communications of the ACM 48*, 8 (2005).

[26] WANG, Y., ROUSSEV, R., VERBOWSKI, C., JOHNSON, A., WU, M., HUANG, Y., AND KUO, S. Gatekeeper: Monitoring Auto-Start Extensibility Points (ASEPs) for Spyware Management. In *Usenix Large Installation System Administration Conference (LISA)* (2004).

[27] WROBLEWSKI, G. *General Method of Program Code Obfuscation*. PhD thesis, Wroclaw University of Technology, 2002.

Appendix

Sample Name	Description
McAfeeAntiPhishingFilter	Antiphishing solution
AT&T P3PClient	Privacy utility
LostGoggles	Search enhancing utility
Earthlink Toolbar	Scam blocker
PopUpBlocker	Utility to block popups
CookiePal	Cookie management
SpywareGuard	Spyware protection utility
ezSaveFlash	Utility to save flash files
keepit	File management utility
KillaFing3	Utility to block popups
Super Popup Blocker	Utility to block popups
SAPplayer	Plays music/video files
BookmarkBuddy	Bookmark management
Plug-In for blind users	Render page for blind users

Table 4: Benign samples.

Sample Name	Description
ZangoIM	Universal instant messaging
BargainBuddy	Bundled Spyware
RAX Search Helper	Search tool
Sitestep	Travel price comparison
Borlan (stdup.dll)	Targeted ads
HtmlEdit Module	Targeted ads
Clear Search	Search tool
IEHelper Module	Installs third-party components
Generic BHO module	Targeted ads
eUniverse	Targetede ads
eZula	URL collector
W01	URL collector
Adware.MediaPlaceTV	Targeted ads and url collector
msnetwrk	URL collector
hopster	URL collector
Replace module	URL collector
e2Give	URL collector
Generic data miner	URL collector
Adware-Click	Targeted ads
CWSMeup-B	Search string collector
HuntBar BHO	URL collector

Table 5: Spyware samples.