Robust Signatures for Kernel Data Structures

Brendan Dolan-Gavitt Abhinav Srivastava Patrick Traynor Jonathon Giffin
School of Computer Science
Georgia Institute of Technology
{brendan,abhinav,traynor,giffin}@cc.gatech.edu

ABSTRACT

Kernel-mode rootkits hide objects such as processes and threads using a technique known as Direct Kernel Object Manipulation (DKOM). Many forensic analysis tools attempt to detect these hidden objects by scanning kernel memory with handmade signatures; however, such signatures are brittle and rely on non-essential features of these data structures, making them easy to evade. In this paper, we present an automated mechanism for generating signatures for kernel data structures and show that these signatures are *robust*: attempts to evade the signature by modifying the structure contents will cause the OS to consider the object invalid. Using dynamic analysis, we profile the target data structure to determine commonly used fields, and we then fuzz those fields to determine which are essential to the correct operation of the OS. These fields form the basis of a signature for the data structure. In our experiments, our new signature matched the accuracy of existing scanners for traditional malware and found processes hidden with our prototype rootkit that all current signatures missed. Our techniques significantly increase the difficulty of hiding objects from signature scanning.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Design, Security

Keywords

Data structures, memory analysis, security

1. INTRODUCTION

Many successful malware variants now employ kernel-mode rootkits to hide their presence on an infected system. A number of large botnets such as Storm, Srizbi, and Rustock have used rootkit techniques to avoid detection. This has led to an arms race between malware authors and security researchers, as each side attempts to find new methods of hiding and new detection techniques, respectively. For example, the FU Rootkit [7] introduced a means of process hiding known as Direct Kernel Object Manipulation (DKOM), which unlinks the malicious process from the list of active processes maintained by the system. In response, some forensic memory analysis tools [5, 36, 46] have started scanning kernel memory using signatures for process data structures, and comparing the results with the standard process list. Because signature-based scanning only requires access to physical memory, scanners are most useful in an offline forensic context, but can be used for live analysis as well.

However, a signature-based search can only be effective if it is difficult for an attacker to evade. As Walters and Petroni [47] note, many current signatures for process data structures in particular can easily be fooled by modifying a single bit in the process header. Although this field is normally constant, its value is irrelevant to the correct operation of the process, and so an attacker with the ability to write to kernel memory can easily modify it and evade detection. This leads naturally to the question of which fields in a given data structure are, in fact, essential to its function and would therefore be good features on which to base a signature.

In this paper, we describe a principled, automated technique for generating robust signatures for kernel data structures such as processes. We employ a feature selection process that ensures that the features chosen are those that cannot be controlled by an attacker—attempting to evade the signature by modifying the features will cause the operating system to crash or the functionality associated with the object to fail. We use our methods to derive a signature for `EPROCESS`, the data structure used to represent a running process in Windows. By construction, an attacker attempting to evade the signature by altering the fields of the process structure will harm the functionality of the OS or process. In addition, we will show conclusively that current, manually generated signatures are trivially evadable by attackers that can write to kernel memory.

Our feature selection mechanism uses two methods to determine which portions of the data structure are critical to its function. First, we monitor operating system execution and note which fields it reads and writes in the target structure. The intuition is that fields that are never accessed cannot cause a crash if modified by an attacker and hence are poor features for robust signatures. Next, we attempt to determine which fields can be modified by an attacker without preventing the data structure from working correctly. This stage of feature selection simulates the behavior of an attacker attempting to evade a signature: if an attacker can arbitrarily modify a field, then any constraint we devise for that field could be bypassed.

After robust features have been selected, we collect samples of those features in the data structure from a large number of instances in memory images. We then use a dynamic invariant detection technique [12] to find constraints on their values that can be used in a signature. Our signature generator uses these constraints to create a plugin for the Volatility memory analysis framework [46] that can find these data structures in memory.

We demonstrate the advantage of our automatically generated signatures over existing solutions by creating a prototype rootkit (based on FU [7]). This custom malware hides processes using a combination of DKOM and signature evasion techniques. By altering unused fields in the process structure that current signatures depend on, the rootkit successfully evades existing signature-based process scanners. We show that our scanner is capable of detecting processes hidden using this method in Windows memory images.

We chose to apply our technique to the problem of finding processes in Windows memory images for several reasons. Reliable identification of running programs is a basic security task that is a prerequisite for many other types of analysis. In addition, process hiding is common feature of kernel malware; a single rootkit may be used to hide the presence of wide variety of user-level malware. However, our techniques are general, and we will discuss the possible application of our techniques to other kernel data structures in Section 7.

We make the following contributions: first, we provide strong empirical evidence that existing signatures are trivially evadable. Second, we develop a *systematic* method for securely selecting features from a data structure that can be used to create highly robust signatures. Finally, we present a method for generating a signature based on robust features, and use it to create a specific signature for process objects on Windows that is as accurate as existing signatures for current malicious and non-malicious processes, but is resistant to evasion.

These results are of immediate importance to a number of security tools which rely on being able to locate data structures in kernel memory. The virtual machine-based "out-of-the-box" malware detection system proposed by Jiang et al. [17], for example, uses several invariant bytes found in the header of a process structure to find processes under Windows. Cross-view detection approaches to detect hidden processes used by memory analysis tools such as Volatility [46], memparser [5], and PTFinder [36] also make use of signatures to locate key kernel structures. Finally, virtual machine introspection libraries such as XenAccess [30] often use signature scans of guest memory to identify processes and provide user-space address translation. The ability to locate data structures such as processes, independent of any operating system-level view, is critical to the correct operation of these tools, and all of them would benefit from the use of more robust signatures.

2. RELATED WORK

Signature-based methods have repeatedly been proposed to identify particular classes of security threats, and, in general, these methods have been found vulnerable to evasion in the face of adversaries. In the area of virus detection, for example, the earliest detectors (and, indeed, many modern commercial utilities) matched byte strings found in viral code that were unlikely to occur in innocuous programs. As the volume of viral code in the wild increased, automated methods were developed to generate signatures based on known viral samples [20]. These methods were thwarted by the appearance of polymorphic and metamorphic [42] viruses; with these techniques, virus creators could transform the malicious code into a form that was functionally equivalent but had no meaningful strings in common with the original code. As a complexity theory problem, reliable detection of bounded length metamorphic viruses has been shown to be NP-complete [41]. Empirical results have confirmed the difficulty of the problem: by mutating Visual Basic viruses found in the wild using techniques similar to fuzzing [15, 24, 25] and random testing, Christodorescu and Jha [9] found that most malware detectors are vulnerable to even simple obfuscation techniques.

The response to network-based worms followed a similar path. Initial attempts to detect network worms used simple, handmade signatures for intrusion detection systems such as Snort [33] that searched for static byte patterns in the network payload of the worm. However, such manual processes did not scale to the large number of worm variants that soon appeared, and numerous systems for automatic signature generation were proposed [21, 22, 38]. These too, however, were soon defeated by polymorphic shellcode that altered the syntactic structure of the worm payload without affecting its functionality [11]. Although later signature generation systems [23,28] were able to create signatures based on invariant features in certain classes of polymorphic shellcode, Gundy et al. [16] found that there were some vulnerabilities that could not be captured by such systems. Indeed, further work by Song et al. [40] demonstrated that the general problem of modeling polymorphic shellcode was likely to be infeasible, and Fogla and Lee [14] found that detecting polymorphic blending attacks is an NP-complete problem.

Although these results do not make the search for reliable kernel data structure signatures look promising, there are key differences that allow signature-based methods to be effective in this case. In the case of viruses and shellcode, the syntax of the malicious input is under the control of the attacker; only its semantics must remain the same in order to produce an effective attack. By contrast, *the syntax of kernel data structures is controlled by the code of the operating system; an attacker can only modify the data contained in the structure to the extent that the operating system will continue to treat it as a valid instance of the given type*. By identifying portions of these data structures that cannot be modified by the attacker, we are able to generate signatures that resist evasion.

Our signature generation system uses dynamic analysis to profile field usage in kernel data structures. A similar technique is employed by Chilimbi et al. [8] for a different goal; their tool, bbcache, analyzes field access patterns in user-space data structures in order to optimize cache performance. After the profiling step, we use a technique similar to fuzzing [24] to identify unused fields in kernel data structures. Fuzzing was also applied by Solar Eclipse [39] to determine which fields in the Portable Executable (PE) file format were required by the Windows loader, in order to develop ways of decreasing the size of Windows executables. Finally, our system finds invariants on the fields in the data structure and produces a Python script that can be used to find instances of the structure in memory images.

Another system that makes use of data structure invariants is Gibraltar by Baliga et al. [2]. Their system creates a graph of all kernel objects in memory and records the values of those objects' members as the system runs. The dynamic invariant detector Daikon [12] is then used to derive constraints on the objects' data. Deviations from the inferred invariants are considered to be attacks against kernel data. The goals and assumptions of our own system, however, are substantially different: whereas Gibraltar assumes that the locations of all kernel data structures can be found *a priori* and then attempts to enforce constraints on those objects, our system seeks to find features of specific data structures in order to locate them in memory. The two approaches can be seen as complementary; once security-critical objects such as processes are located using our signatures, techniques similar to Gibraltar may be employed to detect and enforce invariants on the data.

A number of approaches to detect hidden processes have been

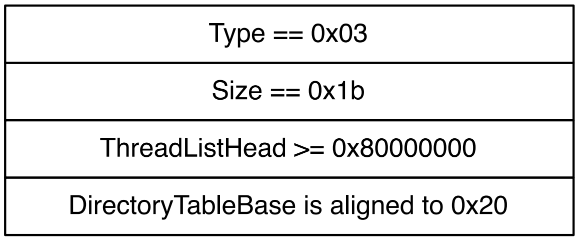

Type == 0x03
Size == 0x1b
ThreadListHead >= 0x80000000
DirectoryTableBase is aligned to 0x20

Figure 1: A naïve signature for the `EPROCESS` data structure. The constraints shown are a subset of those used in PTFinder's process signature. Because the `Size` field is not used by the operating system, an attacker can change its value, hiding the process from a scanner using this signature.

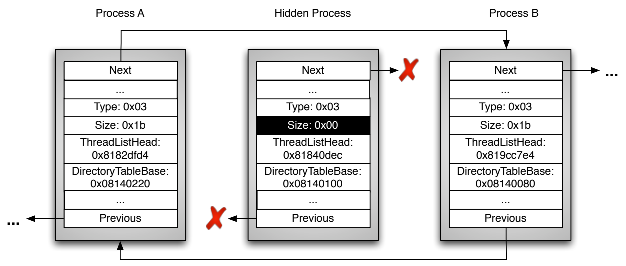

Process A
Hidden Process
Process B
Next
...
Type: 0x03
Size: 0x1b
ThreadListHead: 0x8182dfd4
DirectoryTableBase: 0x08140220
...
Previous
Next
...
Type: 0x03
Size: 0x00
ThreadListHead: 0x81840dec
DirectoryTableBase: 0x08140100
...
Previous
Next
...
Type: 0x03
Size: 0x1b
ThreadListHead: 0x819cc7e4
DirectoryTableBase: 0x08140080
...
Previous
...
...

Figure 2: A portion of the process list while a process hiding attack is underway. The hidden process has been removed from the doubly linked list, and its `Size` field has been changed to evade the signature above.

featured in other work. Antfarm [18] and Lycosid [19] track the value of the CR3 register as a virtual machine executes to identify unique virtual address spaces, which correspond to distinct processes. Although this approach is quite useful in a live environment, it cannot be used for offline forensic analysis. Some offline methods have been proposed as well: Klister [34], for example, attempted to find hidden processes by relying on the scheduler's thread list rather than the systemwide process list, which thwarts some kinds of DKOM attacks. An evasion for this kind of detection has been demonstrated [1], however: an attacker can replace the OS scheduler with a modified copy, bypassing any tool which relies on the original list. Signature scanning is less vulnerable to such attacks, as any changes an attacker makes to the layout of a data structure must be reflected in any OS code that uses the structure.

Finally, other recent work has focused on finding data structures in memory. Laika [10] infers the layout of data structures and attempts to find instances in memory using unsupervised Bayesian learning. Because their system assumes that the data structures are not known in advance, it may be useful in cases where data structure definitions are not available. This flexibility comes the cost of accuracy, however: whereas our process scanner found all instances of the structure in all tested memory images with no false positives, Laika had false positive and negative rates of 32% and 35%, respectively.

We anticipate that rootkit authors will soon add signature evasion techniques to their standard toolkits. Evasion of signatures for kernel data has already been publicly discussed: Walters and Petroni [47] demonstrated that changing a single bit in the Windows process data structure was sufficient to evade all known signatures without harming the functionality of the running process. Similarly, in response to Rutkowska's signature-based modGREPER [35], valerino [43] described an evasion technique that altered a number of fields in the driver and module structures. Finally, bugcheck [6] described a number of methods (including signatures that match fixed strings) for finding kernel data structures in Windows memory and explored several evasion techniques that could be used to hide objects from those techniques. As tools that find hidden objects through memory scans become more common, we believe malware authors will adapt by attempting to evade signatures. This threat motivates our work to generate signatures for kernel data that are resistant to evasion.

3. OVERVIEW

A signature-based scanner examines each offset in physical memory, looking for predefined patterns in the data that indicate the presence of a particular data structure. These patterns take the form of a set of *constraints* on the values of various fields in the structure. For example, Figure 1 shows a simple signature for a process data structure in Windows (called `EPROCESS`; this structure holds accounting information and metadata about a task). The constraints check to see that the `Type` and `Size` fields match predefined constants, that the `ThreadListHead` pointer is greater than a certain value, and that the `DirectoryTableBase` is aligned to 0x20 bytes. These invariants must be general enough to encompass all instances of the data structure, but specific enough to avoid matching random data in memory.

An adversary's goal is to hide the existence of a kernel data structure from a signature-based scanner while continuing to make use of that data structure. We assume that the attacker has the ability to run code in kernel-mode, can read and modify any kernel data, and cannot alter existing kernel code. This threat model represents a realistic attacker: it is increasingly common for malware to gain the ability to execute code in kernel mode [13], and there are a number of solutions available that can detect and prevent modifications to the core kernel code [31, 37], but we are not aware of any solutions that protect kernel data from malicious modification.

To carry out a process hiding attack, such as the one shown in Figure 2, an attacker conceals the process from the operating system using a technique such as DKOM. This attack works by removing the kernel data structure (`EPROCESS`) representing that process from the OS's list of running processes. The process continues to run, as its threads are known to the scheduler, but it will not be visible to the user in the task manager. However, it will still be visible to a signature-based scanner [5, 36] that searches kernel memory for process structures rather than walking the OS process list.

To evade such scanners, the attacker must modify one of the fields in the process data structure so that some constraint used by the signature no longer holds. The field must also be carefully chosen so that the modification does not cause the OS to crash or the malicious program to stop working. In the example shown in Figure 2, the attacker zeroes the `Size` field, which has no effect on the execution of his malicious process, but which effectively hides the data structure from the scanner.

In order to defend against these kinds of attacks, signatures for data structures must be based on invariants that the attacker cannot violate without crashing the OS or causing the malicious program to stop working. The signature's constraints, then, should be placed only on those fields of the data structure that are critical to the correct operation of the operating system. Rather than relying on human judgement, which is prone to errors, our solution profiles OS execution in order to determine the most frequently accessed fields, and then actively tries to modify their contents to determine which are critical to the correct functioning of the system. Such fields will be difficult for an attacker to modify without crashing the system, and are good candidates for robust signatures.

Finally, we will demonstrate that it is possible to automatically infer invariants on these robust fields and construct a scanner that is resistant to evasion attacks.

4. ARCHITECTURE

Our system architecture generates signatures using a three step process. We first *profile* the data structure we wish to model to determine which fields are most commonly accessed by the operating system (Section 4.1). This is done to narrow the domain of the data that we must test in the fuzzing stage: if a field is never accessed in the course of the normal operation of the OS, it is safe to assume that it can be modified without adversely affecting OS functionality. Next, the most frequently accessed fields are *fuzzed* (Section 4.2) to determine which can be modified without causing a crash or otherwise preventing the structure from serving its intended purpose. Finally, we collect known-good instances of the data structure, and build a signature based on these instances that depends only on the features that could not be safely modified during fuzzing (Section 4.3).

Profiling and fuzzing are both essentially forms of *feature selection*. Each tests features of the data structure to determine their suitability for use in a signature. Features that are unused by the operating system or are modifiable without negative consequences are assumed to be under the control of the attacker and eliminated from consideration. Including such weak features would allow an attacker to introduce *false negatives* by violating the constraints of a signature that used them, in the same way that a polymorphic virus evades an overly specific antivirus signature. At the other end of the spectrum, if too few features remain at the end of feature selection, the resulting signature may not be specific enough and may match random data, creating *false positives*.

The profiling and fuzzing stages are implemented using the Xen hypervisor [3] and VMware Server [45], respectively. Because profiling requires the ability to monitor memory access, we chose to use Xen, which is open source and allowed us to make the necessary changes to the hypervisor to support this monitoring. However, Xen lacks the ability to save and restore system snapshots, a feature needed for reliable fuzz testing, so we use VMware Server for this stage. Also, because VMware's snapshots save the contents of physical memory to disk, we were able to easily modify the memory of the guest OS by altering the on-disk snapshot file.

4.1 Data Structure Profiling

In the profiling stage (shown in Figure 3), we attempt to determine which structure fields are most commonly accessed by the operating system during normal operation. Fields which are accessed by the OS frequently are stronger candidates for use in a signature because it is more likely that correct behavior of the system depends upon them. By contrast, fields which are rarely accessed are most likely available to the attacker for arbitrary modification; if the OS never accesses a particular field in the data structure, its value cannot influence the flow of execution.

In our implementation, we make use of a modified Xen hypervisor and the "stealth breakpoint" technique described by Vasudevan and Yerraballi [44] to profile access to the data structure. Stealth breakpoints on memory regions work by marking the memory page that contains the data to be monitored as "not present" by clearing the Present bit in the corresponding page table entry. When the guest OS makes any access to the page, the page fault handler is triggered, an event which can be caught by the hypervisor. The hypervisor then logs the virtual address that was accessed (available in the `CR2` register), emulates the instruction that caused the fault, and allows the guest to continue. These logs can later be examined to determine what fields were accessed, and how often.

For example, to monitor the fields of the Windows `EPROCESS` data structure, we launch a process and determine the address in memory of the structure. We then instruct the hypervisor to log all access to that page, and then allow the process to run for some time. Finally, the logs are examined and matched against the structure's definition to determine how often individual fields were read or written. This process is repeated using several different applications; only the fields that are accessed during the execution of *every* program will be used as input for the fuzzing stage.

We note in passing that determining the precise field accessed requires access to the data structure's definition. On open source operating systems, this information is easy to come by, but for closed source OSes such as Windows it may be more difficult to obtain. For our implementation, which targets Windows XP, we used the debugging symbols provided by Microsoft; these symbols include structure definitions for many kernel structures, including `EPROCESS`.

4.2 Fuzzing

Although a field that is accessed frequently is a stronger candidate than one which is never accessed, this condition alone is not sufficient to identify truly robust features for use in signatures. For example, the operating system may update a field representing a performance counter quite frequently, but its value is not significant to the correct operation of the OS. Thus, to be confident that a signature based on a particular field will be robust against evasion attacks, we must ensure that the field cannot be arbitrarily modified.

The actual fuzzing (shown in Figure 4) is done by running the target operating system inside VMware Server [45]. As in the profiling stage, we first create a valid instance of the data structure. Next, the state of the guest VM is saved so that it can be easily restored after each test. For each field, we then replace its contents with test data from one of several classes:

1. **Zero**: null bytes. This is used because zero is often a significant special case; e.g., many functions check if a pointer is NULL before dereferencing.

2. **Random**: n random bytes from a uniform distribution, where n is the size of the field.

3. **Random primitive type**: a random value appropriate to the given primitive type. In particular, pointer fields are fuzzed using valid pointers to kernel memory.

4. **Random aggregate type**: a random value appropriate to the given aggregate type (i.e., structure). Embedded structures are replaced by other valid instances of that structure, and pointers to structures of a given type are replaced by pointers to that same type. Currently implemented as a random choice of value from that field in other instances of the target data structure.

After the data is modified, we resume the guest VM and monitor the operating system to observe the effects of our modifications. To determine whether our modification was harmful, we must construct a test, ϕ, which examines the guest and checks if the OS is still working and the functionality associated with the target data structure instance is still intact.

For a process data structure, ϕ could be a test that first checks to see if the OS is running, and then determines whether the associated

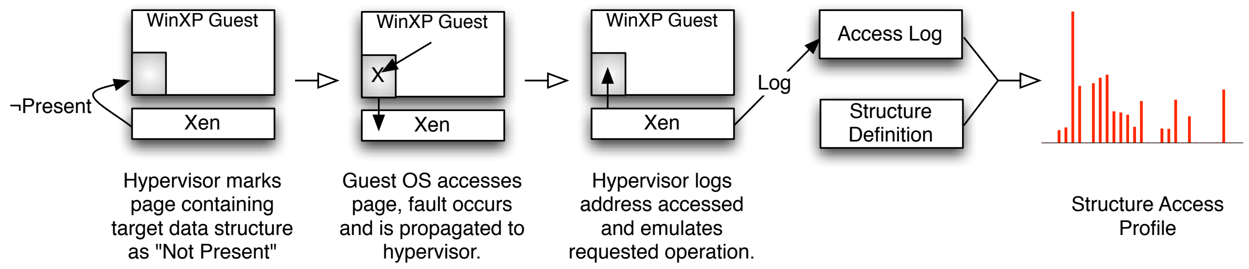

WinXP Guest
¬Present
Xen
Hypervisor marks page containing target data structure as "Not Present"
WinXP Guest
X
Xen
Guest OS accesses page, fault occurs and is propagated to hypervisor.
WinXP Guest
Xen
Log
Hypervisor logs address accessed and emulates requested operation.
Access Log
Structure Definition
Structure Access Profile

Figure 3: The profiling stage of our signature generation system. As the OS executes, accesses to the target data structure are logged. These logs are then combined with knowledge of the field layout in the structure to produce an access profile.

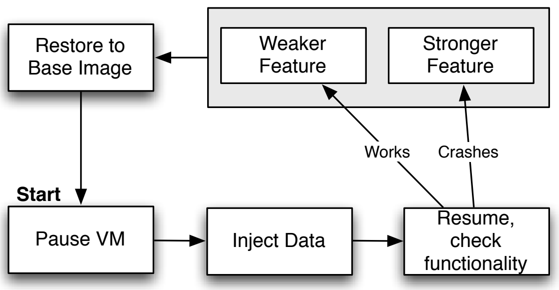


Figure 4: The architecture of the fuzzing stage. Starting from some baseline state, a test pattern is written into a field in the target data structure in the memory of the virtual machine. The VM is then resumed and tested to see if its functionality is intact. If so, the modification was not harmful and we consider the field a weaker candidate for a signature.

```
class Scan(RobustPsScanner,
        PoolScanProcessFast2.Scan):
    def __init__(self, poffset, outer):
        RobustPsScanner.__init__(self,
            poffset, outer)
        self.add_constraint(
            self.check_objecttable
        )
        self.add_constraint(
            self.check_grantedaccess
        )

    [...]

    def check_objecttable(self, buf, off):
        val = read_obj_from_buf(buf,
            types, ['_EPROCESS',
            'ObjectTable'], off)
        res = (val == 0 or
            (val & 0xe0000000 ==
                0xe0000000 and
            val % 0x8 == 0))
        return res

    def check_grantedaccess(self, buf, off):
        val = read_obj_from_buf(buf,
            types, '_EPROCESS',
            'GrantedAccess'], off)
        res = val & 0x1f07fb == 0x1f07fb
        return res
```

Figure 5: Two sample constraints found by our signature generator. If all constraints match for a given data buffer, the plugin will report that the corresponding location in memory contains an `EPROCESS` instance.

program is still running. This check may be simpler to program if the behavior of the application in question is well-known. In our experiments (described in Section 5.1), we used an application we had written that performed some simple functions such as creating a file on the hard drive. This allowed ϕ to check if the program had continued to run successfully by simply testing for the existence of the file created by the program.

To actually inject the data, we pause the virtual machine, which (in VMware) writes the memory out to a file. We then use the memory analysis framework Volatility [46] (which we modified to support writing to the image) to locate the target instance and modify the appropriate field with our random data. Volatility was ideal for this purpose, because it has the ability to locate a number of kernel data structures in images of Windows memory and provides a way to access the data inside the structures by field name. The modifications to allow writing to the image (a feature not normally supported by Volatility, as it is primarily a forensics tool) required 303 lines of additional code.

Finally, we resume the virtual machine and check to see if ϕ indicates the system is still functioning correctly after some time interval. This interval is currently set to 30 seconds to allow time for the VM to resume and any crashes to occur. Software engineering studies [26] have found that crashes typically occur within an average of one million instructions from the occurrence of memory corruption; thus, given current CPU speeds, it is reasonable to assume that this 30 second delay will usually be sufficient to determine whether the alteration was harmful to program functionality. The result of the test is logged, and we restore the saved virtual machine state before running the next test. Any fields whose modification consistently caused ϕ to indicate failure are used to generate a signature for the data structure.

4.3 Signature Generation

The final signature generation step is performed using a simplified version of dynamic invariant detection [12]. For each field identified by the feature selection as robust, we first gather a large number of representative values from all instances of the target data structure in our corpus of memory images. Then, for each field, we test several constraint templates to see if any produce invariants that apply to all known values of that field. The templates checked are:

- **Zero subset**: check if there is a subset of the values that is zero. If so, ignore these values for the remaining checks.
- **Constant**: check if the field takes on a constant value.

- **Bitwise AND**: check if performing a bitwise AND of all values results in a non-zero value. This effectively checks whether all values have any bits in common.
- **Alignment**: check if there is a power of two (other than 1) on which all values are aligned.

First, because many fields use zero as a special case to indicate that the field is not in use, we check if any of the instances are zero, and then remove these from the set to be examined. Constraints are then inferred on the remaining fields, and zero will be included as a disjunctive (OR) case in the final signature. The other templates will produce conjunctive constraints on the non-zero field values.

The *constant* template determines whether a field always takes on a particular value. This is useful, for example, for data structures that have a "magic" number that identifies them uniquely. Because the features that are used for signature generation are known to be robust (as these were the selection criteria described in Sections 4.1 and 4.2), we can have some confidence that the operating system performs sanity checking on such constant values.

The two remaining tests are particularly useful for finding constraints on pointer values. The *bitwise AND* test simply performs a bitwise AND of all values observed. In many operating systems, kernel memory resides in a specific portion of virtual address space, such as (in Windows) the upper 2 gigabytes. One can determine if a 32-bit pointer always points to kernel memory, then, by simply checking that the highest bit is set.

Finally, the *alignment* test attempts to find a natural alignment for all values. As an optimization, many OS memory managers allocate memory aligned on a natural processor boundary, such as eight bytes. As a result, most pointers to kernel objects will likewise have some natural alignment that we can discover and use as a constraint.

Our signature generator takes as input a comma-separated file, where each row gives the field name and a list of values observed for that field. For each field, it applies the constraint templates to the values listed and determines a boolean expression that is true for every value. It then outputs a plugin for Volatility, written in Python, that can be used to scan for the target data structure in a memory image. An excerpt of the plugin generated to search for instances of the `EPROCESS` data structure is given in Figure 5.

The signature generation mechanism produces extremely robust results in practice: as we describe in Section 6.3, the signature we generated for Windows process data structures found all instances of the data structure in memory with no false positives or negatives. Should this technique prove insufficient for some data structure, however (for example, if only a few features are robust enough to use in a signature), more heavyweight techniques such as dynamic heap type inference [32] could be used.

4.4 Discussion

Our experiments (described in Section 6) show that these techniques can be used to derive highly accurate signatures for kernel data structures that are simultaneously difficult to evade. There are, however, certain drawbacks to using probabilistic methods such as dynamic analysis and fuzz testing. In particular, both techniques may suffer from *coverage* problems. In the profiling stage, it is highly unlikely that every field used by the operating system will actually be accessed; there are many fields that may only be used in special cases. Likewise, during fuzzing, it is possible that although the operating system did not crash during the 30 seconds of testing, it might fail later on, or in some special circumstances.

In both of these cases, however, we note that these omissions will only cause us to ignore potentially robust features, rather than accidentally including weak ones. Moreover, from an attacker's point of view, the malware need not work perfectly, or run in every special case: sacrificing correct operation in a tiny fraction of configurations may be worth the increased stealth afforded by modifying these fields. Thus, a short time interval for testing is *conservative*: it will never cause a weak feature to be used in a signature, as only features whose modification consistently causes OS crashes form the basis of signatures. However, it may cause fields to be eliminated that would, in fact, have been acceptable to use in a signature. If too many fields are eliminated, the resulting signature may match random data in memory, creating false positives. In any case, this limitation is easily overcome by increasing the amount of time the fuzzer waits before testing the OS functionality, or by exercising the guest OS more strenuously during that time period.

However, there are some coverage issues that could result in weak signatures. Because fuzzing is a dynamic process, it is possible to only inject a subset of values that causes the OS to crash, while there exists some other set of values that can be used without any negative effects. In this case, we may conclude that a given feature is robust when in fact the attacker can modify it at will. For most fields it is not practical to test every possible value (for example, assuming each test takes only five seconds, it would still require over 680 years to exhaustively test a 32-bit integer). In Section 8, we will consider future enhancements to the fuzzing stage that may improve coverage.

Finally, we note that although the features selected using our method are likely to be difficult to modify, there is no guarantee that they will be usable in a signature. For example, although our testing found that the field containing the process ID is difficult to modify, it could still be any value, and examining a large number of process IDs will not turn up any constraints on the value. In practice, though, we found that most of the "robust" features identified were fairly simple to incorporate into a signature, and we expect that this will be true for most data structures.

5. METHODOLOGY

Signature search is essentially a classification problem: given an unknown piece of data, we wish to classify it as an instance of our data type or as something else. Our experiments, therefore, attempt to measure the performance of the signatures using standard classification metrics: false positives and negatives. A false positive in this case is a piece of data that matches the signature but would not be considered a valid instance of the data structure by the operating system. Conversely, a false negative is a valid instance that is not classified as such by our signature. False negatives represent cases where the attacker could successfully evade the signature, whereas false positives could introduce noise and make it difficult for to tell what processes are actually running.

For our purposes, we only consider false positives that are syntactically invalid. We note that an attacker could generate any number of false positives by simply making copies of existing kernel data structures. These structures would be semantically invalid (the operating system would have no knowledge of their existence), but would be detected by a signature scanner. The possibility of such "dummy" structures is a known weakness of signature-based methods for finding kernel data structures [47]; however, a solution to this problem is outside the scope of this work.

For our experiments, we chose to generate a signature for the Windows `EPROCESS` data structure, which holds information related to each running process. This structure was chosen because it is the most commonly hidden by malicious software, and there are a number of existing signature-based tools that attempt to locate this data structure in memory [5, 36, 46]. We compare the success

System Utilities

Name	Version	Fields
Telnet	5.1.2600.5512	112
Command shell	5.1.2600.5512	135
NTFS Defragment	5.1.2600.5512	123
Explorer	6.0.2900.5512	143

Browsers

Name	Version	Fields
Internet Explorer	7.0.5730.13	153
Mozilla Firefox	3.0.5	147

Games

Name	Version	Fields
WinQuake	1.06	129
Minesweeper	5.1.2600.0	108

Editor

Name	Version	Fields
Notepad	5.6.2600.5512	151

Debugger

Name	Version	Fields
Notepad (debugged)	5.6.2600.5512	145
Windbg (debugging)	6.9.0003.113 x86	146

Communications

Name	Version	Fields
Outlook Express	6.00.2900.5512	148
Pidgin	2.5.3	143

Installer

Name	Version	Fields
Pidgin Installer	2.4.0	188

Antivirus / Antispyware

Name	Version	Fields
Avira AntiVir	8.2.0.337	130
Spybot Search & Destroy	1.6.0.0	136

Network Servers

Name	Version	Fields
Apache HTTPd	2.2.11	108
network_listener	N/A	139

Multimedia

Name	Version	Fields
Windows Media Player	9.00.00.4503	142
iTunes	8.0.2.20	142

Table 1: List of applications profiled, along with the number of fields in `EPROCESS` accessed.

of our signature with these tools. However, our work can also be applied to generate signatures for other data structures.

5.1 Profile Generation and Fuzzing

During the profiling stage, we examined access patterns for fields in the `EPROCESS` data structure. To ensure that our data represented a wide range of possible application-level behavior, we chose twenty different programs that performed a variety of tasks (see Table 1 for a full list). To obtain a profile, we first launched the application and noted the address of its associated `EPROCESS` structure using the kernel debugger, WinDbg. We then instructed the Xen hypervisor to monitor access to the page, and used the application for a minimum of five minutes.

We note that in addition to differences caused by the unique function performed by each application, other activities occurring on the system may cause different parts of the data structure to be exercised. In an attempt to isolate the effects caused by differences in program behavior, as each profile was generated we also used the system to launch several new tasks (Notepad and the command shell, `cmd.exe`), switch between interactive programs, and move and minimize the window of the application being profiled.

After profiling the applications, we picked only the features that were accessed in all twenty applications. This choice is conservative: if there are applications which do not cause a particular field to be exercised, then it may be possible for an attacker to design a program that never causes the OS to access that field. The attacker would then be able to modify the field's value at will and evade any signature that used constraints on its value.

As described in Section 4.2, features that were accessed by all programs profiled were fuzzed to ensure that they were difficult to modify. Checking that the `EPROCESS` data structure is still functioning after each fuzz test is much simpler if the associated program has known, well defined behavior. For this reason, we chose to create a program called `network_listener` that opens a network socket on TCP port 31337, waits for a connection, creates a file on the hard drive, and finally exits successfully. The baseline snapshot was taken just after launching `network_listener` inside the guest VM.

Because the program behavior is known in advance, the test to see if the OS and program are still working correctly (ϕ) becomes simple. From the host, we perform the following tests on the virtual machine:

1. Determine if the virtual machine responds to pings.
2. Check that the program is still accepting connections on port 31337.
3. Check for the existence of the file written by the application (using the VMware Tools API).

If all tests pass, then ϕ returns true, indicating that the modification was accomplished without harming OS or program functionality. If instead ϕ returns false, then the OS has crashed or some aspect of the program associated with our `EPROCESS` instance has stopped functioning. This latter case indicates that the OS will not accept arbitrary values for the field, and provides evidence that we can safely build a signature based on the field.

5.2 Signature Generation and Evaluation

Finally, we generated a signature using the method described in Section 4.3. The features chosen were the 15 most robust, as measured by the tests done during the fuzzing stage. For each of these fields, we extracted from our corpus of memory images (our training set) a list of the values it contained for all processes found in the image. The four images in the training set were not infected by malware, and were taken from systems running the 32-bit version of Windows XP, Service Pack 2. Processes were located in the memory image by walking the operating system's process list; in the absence of maliciously hidden processes, this serves as "ground truth" for the list of valid process data structures. We then used our signature generator to find constraints on the observed values. The signature generator outputs a plugin for Volatility that can be used to search for a data structure matching the constraints found in a memory image.

The generated scan plugin was used to search for processes in a number of memory images. For this purpose, we used two images provided by the NIST Computer Forensic Reference Data Sets (CFReDS) project [27] and a paused virtual machine on which a process had been hidden by our own custom rootkit. The number of false positives and negatives were measured for each test

Field	Used by
ThreadListHead.Blink	Volatility (psscan2)
Pcb.Header.Type	PTFinder
Pcb.Header.Size	PTFinder
WorkingSetLock.Header.Type	PTFinder
WorkingSetLock.Header.Size	PTFinder
AddressCreationLock.Header.Type	PTFinder
AddressCreationLock.Header.Size	PTFinder

Table 2: Fields zeroed by our modified FU Rootkit, along with the scanners that depend on that field.

image, and compared against two existing signature-based tools, PTFinder [36] and Volatility's `psscan2` module.[1]

Our custom malware, which is a slightly modified version of the FU Rootkit [7], hides processes using DKOM (as in the original FU), and additionally attempts to evade known process signatures by zeroing non-essential fields in the process data structure. The fields modified, shown in Table 2, were chosen by finding those fields that were used by common scanners but that our initial structure profiling indicated were unused by the OS.

6. RESULTS

The experimental results are given below. We describe the outcome of profiling twenty different applications and present the results of the fuzzing stage. These features are used by the signature generator to find constraints and create a new process scan module for Volatility. Finally, we compare the accuracy of our scanner with other popular scanners.

Throughout, we also consider what our results tell us about the features used by another popular signature (PTFinder's signature for `EPROCESS`). We find that after fuzzing and profiling, only two of its nine features are resistant to evasion; the remaining invariants are not sufficient to avoid matching random portions of memory.

6.1 Profiling

After profiling the twenty applications described in Section 5.1, we can confirm our hypothesis that some fields are accessed only rarely, if ever. Of the 221 fields in the `EPROCESS` data structure, 32 were never accessed during the execution of the profiled programs. At the other extreme, 72 were accessed for every application and are thus strong candidates for a process signature. In between are 117 fields that were accessed by some programs, but not others; Figure 6(a) gives a histogram detailing precisely how many programs accessed each field.

Included in the 32 fields that were never accessed are three of the nine used by PTFinder to locate processes in memory dumps; a further four are only accessed by a subset of the programs profiled (the profiling results for the fields used by PTFinder are shown in Figure 6(b)). Because the signature used in PTFinder is conjunctive (all of its constraints must be met in order to report a match), and the attacker has complete control over three of the fields used in the signature, *we can conclude that this signature can be trivially evaded*. The features chosen by PTFinder's author did not correspond to those used by the OS, demonstrating that human judgment may not be sufficient to determine what fields are appropriate for use in data structure signatures.

9 [1]Note that Volatility also includes a process scanner called `psscan`. This scanner uses the same constraints as PTFinder, and hence is vulnerable to the same evasions, so we do not consider it here.

Field	Z	R	P	A	Total
ActiveProcessLinks.Flink	5	5	5	5	20
Pcb.DirectoryTableBase[0]	5	5	5	5	20
Pcb.ThreadListHead.Flink	5	5	5	5	20
Token.Value	5	5	5	3	18
Token.Object	5	5	5	1	16
VadHint	5	5	2	0	12
UniqueProcessId	1	5	5	1	12

Table 3: Selected `EPROCESS` fields and the results of fuzzing them. The values indicate the number of times a given test caused ϕ to return false, indicating that the OS or program had stopped working correctly. The columns indicate number of OS crashes when testing with the Zero, Random, Random Primitive, and Random Aggregate patterns.

6.2 Fuzzing

We then took the 72 fields identified as always accessed during the profiling stage and fuzzed them using the four different data patterns (zero, random, random primitive, and random aggregate), modifying each field with each pattern five times, for a total of 1,440 distinct tests. The overall number of failed tests for each field is shown in Figure 7. However, this does not provide a full picture of the fuzzing results, as it is also important to note which data patterns caused the OS to fail. It may be acceptable to use a field in a signature even if it is possible to write zeroes to that field, for example, because the constraint could include zero as a special case. We have, therefore, included several sample fields in Table 3, in order to give an idea of what the result data looks like.

We find, as expected, that there are many "essential" fields upon which we may base our signature. Five fields failed every attempt at manipulation, and a further 12 failed more than half of the tests (i.e., more than 10). This will give us a set of robust features that is large enough to ensure that the number of false positives found by the signature is minimized.

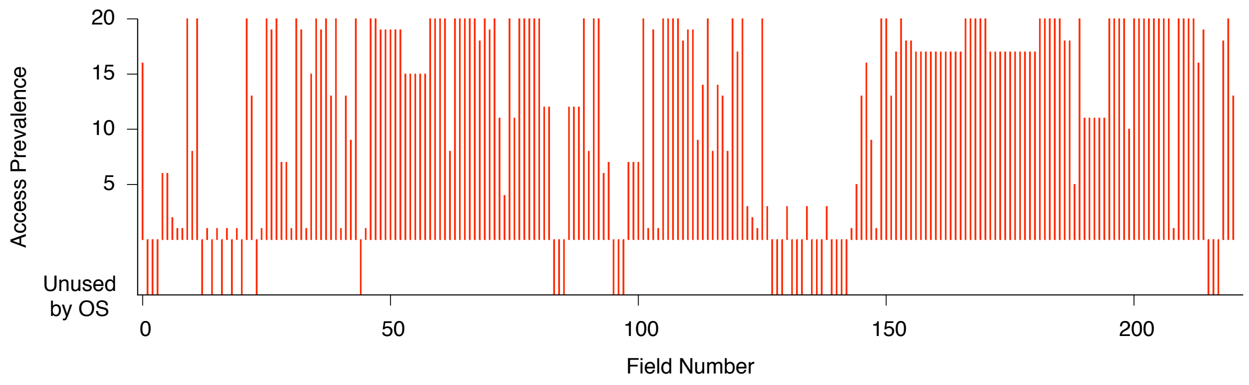
As in the profiling stage, we also note that these results give us a very strong indication of what fields *not* to choose. Of the 72 fields from profiling, 29 passed every test (their modification did not result in any loss of functionality); these, again, would be a poor basis for a signature as their values can be controlled by an attacker.

6.3 Signature Accuracy

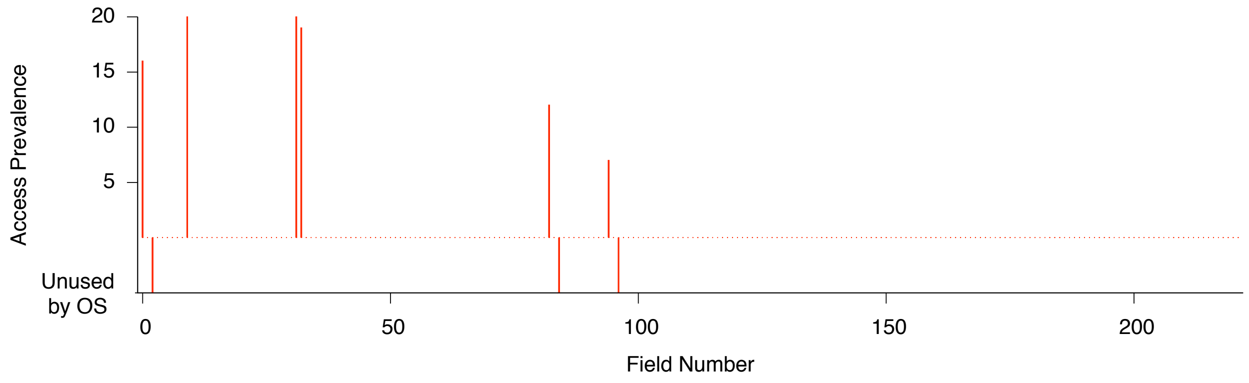
With a list of robust features in hand, we used our signature generator to find constraints on the values of each feature. The field values were collected from 184 processes across the four images in our training set and constraints were inferred using the templates described in Section 5.2, producing the constraints shown in Table 4. The signature generator produced a plugin for Volatility that uses the constraints found to search for `EPROCESS` instances in memory images.

We then evaluated the accuracy of three process scanners: our own automatically generated scanner, Volatility's `psscan2` module, and PTFinder [36]. Using each scanner, we searched for instances of the `EPROCESS` data structure in the three memory images listed in Section 5.2. The output of each tool was compared against the OS's list of running processes (found by walking the linked list of process data structures using Volatility's `pslist` module). In the case of the non-NIST image, we also checked for the presence of our hidden process, which was not visible in the standard OS process list.

We found that all three scanners had equal detection performance



9(a) Number of profiled programs in which `EPROCESS` fields were accessed. Only fields accessed by all 20 programs provide the strongest assurance for use in a signature.



9(b) Access prevalence of fields used by PTFinder's `EPROCESS` signature. Note that the signature relies on field values never used by the OS, so an attacker can safely change these values to evade the signature.

Figure 6: Access prevalence for `EPROCESS` for profiled applications.

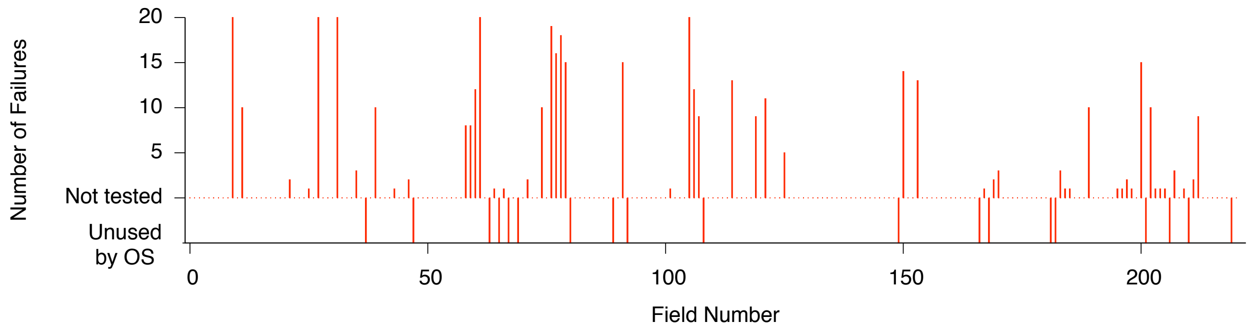


Figure 7: Fuzzing results for `EPROCESS`. The y-axis represents the total number of tests for which ϕ returned false, indicating that the process was no longer functioning correctly. Higher bars indicate stronger features.

on the NIST images and found every process data structure with no false positives. However, only our own scanner was able to detect the hidden process in the third image, demonstrating that an attacker could potentially evade both psscan2 and PTFinder with minimal effort. We believe our signature will also prove resistant to evasion against real-world attackers, as the features it uses are demonstrably difficult for an attacker to alter.

Aside from the active processes in the images, we also noted some discrepancies between the three scanners with respect to terminated processes whose `EPROCESS` structure was still in memory and had not yet been overwritten. Although PTFinder and psscan2 were vulnerable to the evasion by our custom malware, they also found these terminated processes, which our scanner missed.

As terminated processes could be of forensic interest, we checked whether there was some subset of our "robust" features that would find such processes without introducing false positives. By modifying our scanner to report the result of each constraint for the terminated processes, we found that Windows appears to zero the

Field	Constraint
Pcb.ReadyListHead.Flink	`val & 0x80000000 == 0x80000000 && val % 0x8 == 0`
Pcb.ThreadListHead.Flink	`val & 0x80000000 == 0x80000000 && val % 0x8 == 0`
WorkingSetLock.Count	`val == 1 && val & 0x1 == 0x1`
Vm.VmWorkingSetList	`val & 0xc0003000 == 0xc0003000 && val % 0x1000 == 0`
VadRoot	`val == 0 || (val & 0x80000000 == 0x80000000 && val % 0x8 == 0)`
Token.Value	`val & 0xe0000000 == 0xe0000000`
AddressCreationLock.Count	`val == 1 && val & 0x1 == 0x1`
VadHint	`val == 0 || (val & 0x80000000 == 0x80000000 && val % 0x8 == 0)`
Token.Object	`val & 0xe0000000 == 0xe0000000`
QuotaBlock	`val & 0x80000000 == 0x80000000 && val % 0x8 == 0`
ObjectTable	`val == 0 || (val & 0xe0000000 == 0xe0000000 && val % 0x8 == 0)`
GrantedAccess	`val & 0x1f07fb == 0x1f07fb`
ActiveProcessLinks.Flink	`val & 0x80000000 == 0x80000000 && val % 0x8 == 0`
Peb	`val == 0 || (val & 0x7ffd0000 == 0x7ffd0000 && val % 0x1000 == 0)`
Pcb.DirectoryTableBase.0	`val % 0x20 == 0`

Table 4: Constraints found for "robust" fields in the `EPROCESS` data structure. The operators shown have the same meaning as in C; % stands for the mod operation, and & represents bitwise AND. && and || are the boolean operators for "and" and "or", respectively.

`Token.Object` and `Token.Value` fields, which refer to the process's security token, when the process exits. Once we removed these constraints from our signature, we were able to find the terminated processes reported by other tools without introducing false positives. We note that our scanner remains resistant to evasions, as the remaining fields are all robust. The terminated processes demonstrate the importance of generating signatures from a training set that represents the full range of objects one wishes to detect.

7. OTHER STRUCTURES

Although our experiments have only been run on `EPROCESS`, we are confident that the technique will generalize to other data structures. Certain structures in particular, such as threads (represented by `ETHREAD` in Windows) and files (`FILE_OBJECT`), would be good candidates for signature generation, as they contain a wealth of information about the runtime state of the system that is useful for forensic analysis. We will briefly consider what changes might be needed to generate signatures for these structures.

The profiling stage is essentially the same for any structure: the objects are created by some user-level program (i.e., by spawning a thread or opening a file), their location in memory is determined, and the memory region is monitored to log access to the structure. In the fuzzing stage, the only significant challenge is creating an appropriate functionality test ϕ. As threads contain executable code, one could simply use the same test as for processes: attempt to create a file and ensure that the file is created successfully. For file objects, one could test functionality by performing a range of operations on the open file, such as reading, writing, seeking, and closing the file. Finally, our signature generator is not specific to any one object type and could be used as-is: the only input required is a list of observed values for each field in the data structure.

One final complication may arise if the target structure is fairly small. In this case, it may be that after eliminating weak features, there will not be enough left to create a reliable signature (in the sense of having few false positives). In this case, we might employ a more sophisticated search technique: rather than simply using basic pattern matching to find instances of the structure, we could take advantage of information such as the types of objects to which it points. This technique has previously been used successfully in other work to identify the types of objects on the heap [32], and this additional contextual information could improve signature accuracy.

8. FUTURE WORK

As discussed in Section 4.4, obtaining full coverage during fuzzing is impractical; however, but it may be possible to improve our coverage through more judicious selection of random data. For example, we might incorporate *mutation fuzzing* [29], which generates fuzz data by creating small, random variations on existing values. This would help us more efficiently explore the space of possible values, as for many fields legal values will be clustered fairly close together.

The profiling stage could also be made more accurate by switching from simply monitoring whether a field is accessed to attempting to determine how it is used. This would involve the use of taint tracking [4] to find out whether the value of a given field actually influences the execution of the OS. We expect that this could significantly reduce the number of fields that would need to be fuzzed.

Finally, although the automatically generated signatures from our method appear to work well, they are based on dynamic analysis and may therefore suffer from coverage problems. Gaps in coverage could lead to false negatives and evasions in the signature matching process: a constraint inferred on a small number of samples may not be representative of the full range of values that field uses, and thus be overly restrictive. To improve confidence in such constraints, one could also use static analysis to attempt to prove that the inferred constraints do, indeed, hold in all cases.

9. CONCLUSIONS

We have successfully demonstrated that it is possible to automatically select robust features of data structures and generate evasion-resistant signatures based on them. More importantly, we have described a systematic way of determining which features to use when creating a data structure signature. To our knowledge, no such method was previously available, and we believe that many applications will benefit from this technique.

Our work resulted in a new signature for process data structures on Windows, which can be used immediately by applications which require the ability to locate processes in memory. We also showed that existing signatures used by memory analysis applications were vulnerable to evasion, and in the case of PTFinder we described precisely which constraints could be violated by an attacker. These concrete contributions significantly increase the difficulty of hiding process objects from signature scans on Windows systems.

Acknowledgements

We would like to thank our anonymous reviewers and Matt Fredrikson for their helpful comments on earlier drafts of this paper. This work is supported in part by NSF contract CNS-0831300 and DHS contract FA8750-08-2-0141. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the above governmental agencies or the U.S. Government.

10. REFERENCES

[1] 90210. Bypassing Klister 0.4 with no hooks or running a controlled thread scheduler. `https://www.rootkit.com/newsread.php?newsid=235`.

[2] A. Baliga, V. Ganapathy, and L. Iftode. Automatic inference and enforcement of kernel data structure invariants. In *Proceedings of the 24th Annual Computer Security Applications Conference (ACSAC)*, Anaheim, California, USA, 2008.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating systems principles (SOSP)*, Bolton Landing, NY, 2003.

[4] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, MITRE Corp., Bedford, MA, 1973.

[5] C. Betz. MemParser. `http://sourceforge.net/projects/memparser`.

[6] bugcheck. GREPEXEC: Grepping executive objects from pool memory. *Uninformed Journal*, 4, 2006.

[7] J. Butler. FU rootkit. `http://www.rootkit.com/project.php?id=12`.

[8] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation (PLDI)*, Atlanta, GA, 1999.

[9] M. Christodorescu and S. Jha. Testing malware detectors. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, Boston, MA, 2004.

[10] A. Cozzie, F. Stratton, H. Xue, and S. T. King. Digging for data structures. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[11] T. Detristan, T. Ulenspiegel, Y. Malcom, and M. S. von Underduk. Polymorphic shellcode engine using spectrum analysis. `http://www.phrack.com/issues.html?issue=61&id=9`, 2003.

[12] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3), 2007.

[13] F-Secure. A different twist on the path to the kernel. `http://www.f-secure.com/weblog/archives/00001507.html`, 2008.

[14] P. Fogla and W. Lee. Evading network anomaly detection systems: formal reasoning and practical techniques. In *Proceedings of the 13th ACM conference on Computer and communications security (CCS)*, Alexandria, VA, 2006.

[15] J. E. Forrester and B. P. Miller. An empirical study of the robustness of Windows NT applications using random testing. In *Proceedings of the 4th conference on USENIX Windows Systems Symposium (WSS)*, Seattle, WA, 2000.

[16] M. V. Gundy, H. Chen, Z. Su, and G. Vigna. Feature omission vulnerabilities: Thwarting signature generation for polymorphic worms. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC)*, Miami Beach, FL, 2007.

[17] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through VMM-based "out-of-the-box" semantic view reconstruction. In *Proceedings of the 14th ACM conference on Computer and communications security (CCS)*, Alexandria, VA, 2007.

[18] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Antfarm: tracking processes in a virtual machine environment. In *Proceedings of the USENIX Annual Technical Conference (ATEC)*, Boston, MA, 2006.

[19] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. VMM-based hidden process detection and identification using lycosid. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE)*, Seattle, WA, 2008.

[20] J. Kephart and W. Arnold. Automatic extraction of computer virus signatures. In *Proceedings of the 4th International Virus Bulletin Conference (VB)*, Jersey, Channel Islands, UK, 1994.

[21] H.-A. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *Proceedings of the 13th conference on USENIX Security Symposium*, volume 13, San Diego, CA, 2004.

[22] C. Kreibich and J. Crowcroft. Honeycomb: creating intrusion detection signatures using honeypots. *ACM SIGCOMM Computer Communication Review*, 34(1), 2004.

[23] Z. Li, M. Sanghi, Y. Chen, M.-Y. Kao, and B. Chavez. Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, Oakland, CA, 2006.

[24] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12), 1990.

[25] B. P. Miller, D. Koski, C. Pheow, and L. V. Maganty. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Technical report, 1995.

[26] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *Proceedings of the 32nd annual international symposium on Computer Architecture (ISCA)*, Washington, DC, USA, 2005.

[27] National Institute of Standards and Technology (NIST). The CFReDS project. `http://www.cfreds.nist.gov/`.

[28] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, Oakland, CA, 2005.

[29] P. Oehlert. Violating assumptions with fuzzing. *IEEE Security and Privacy*, 3(2), 2005.

[30] B. D. Payne, M. Carbone, and W. Lee. Secure and flexible monitoring of virtual machines. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC)*, Miami Beach, FL, 2007.

[31] N. L. Petroni, Jr. and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM conference on Computer and communications security (CCS)*, Alexandria, VA, 2007.

[32] M. Polishchuk, B. Liblit, and C. W. Schulze. Dynamic heap type inference for program understanding and debugging. *SIGPLAN Not.*, 42(1):39–46, 2007.

[33] M. Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX conference on System administration (LISA)*, Seattle, WA, 1999.

[34] J. Rutkowska. Klister v0.3. `https://www.rootkit.com/newsread.php?newsid=51`.

[35] J. Rutkowska. modGREPER - hidden kernel modules detector. `https://www.rootkit.com/newsread.php?newsid=315`, 2005.

[36] A. Schuster. Searching for processes and threads in Microsoft Windows memory dumps. In *Proceedings of the 6th Annual Digital Forensic Research Workshop (DFRWS)*, Lafayette, IN, 2006.

[37] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of 21st ACM SIGOPS symposium on Operating systems principles (SOSP)*, Stevenson, WA, 2007.

[38] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, 2004.

[39] Solar Eclipse. Tiny PE. `http://www.phreedom.org/solar/code/tinype/`, 2006.

[40] Y. Song, M. Locasto, A. Stavrou, A. Keromytis, and S. Stolfo. On the infeasibility of modeling polymorphic shellcode. In *Proceedings of the 14th ACM conference on Computer and communications security (CCS)*, Alexandria, VA, 2007.

[41] D. Spinellis. Reliable identification of bounded-length viruses is NP-complete. *IEEE Transactions on Information Theory*, 49(1), 2003.

[42] P. Ször and P. Ferrie. Hunting for metamorphic. In *Proceedings of the 11th International Virus Bulletin Conference*, Prague, Czech Republic, 2001.

[43] valerino. Please don't greap me! `https://www.rootkit.com/newsread.php?newsid=316`, 2005.

[44] A. Vasudevan and R. Yerraballi. Stealth breakpoints. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)*, Tucson, AZ, 2005.

[45] VMWare. VMWare Server. `http://www.vmware.com/products/server/`.

[46] A. Walters. The Volatility framework: Volatile memory artifact extraction utility framework. `https://www.volatilesystems.com/default/volatility`.

[47] A. Walters and N. Petroni. Volatools: Integrating volatile memory forensics into the digital investigation process. In *Blackhat Federal*, Washington, DC, 2007.