On Random-Inspection-Based Intrusion Dectection

Simon P. Chung and Aloysius K. Mok*

Department of Computer Sciences,
University of Texas at Austin, Austin TX 78712, USA.
Email:{phchung,mok}@cs.utexas.edu

Keywords: mimicry attacks, intrusion detection, computer security, random inspection .

Abstract. Monitoring at the system-call-level interface has been an important tool in intrusion detection. In this paper, we identify the predictable nature of this monitoring mechanism as one root cause that makes system-call-based intrusion detection systems vulnerable to mimicry attacks. We propose random inspection as a complementary monitoring mechanism to overcome this weakness. We demonstrate that random-inspection-based intrusion detection is inherently effective against mimicry attacks targeted at system-call-based systems. Furthermore, random-inspection-based intrusion detection systems are also very strong stand-alone IDS systems. Our proposed approach is particularly suitable for implementation on the Windows operating system that is known to pose various implementation difficulties for system-call-based systems. To demonstrate the usefulness of random inspection, we have built a working prototype tool: the WindRain IDS. WindRain detects code injection attacks based on information collected at random-inspection points with acceptably low overhead. Our experiments show that WindRain is very effective in detecting several popular attacks against Windows. The performance overhead of WindRain compares favorably to many other intrusion detection systems.

1 Introduction

Ever since they were first introduced in [9, 15], system-call-based anomaly-detection systems have been considered to be an effective approach to achieve intrusion detection in computer security, but there are also weaknesses in this approach. In particular, system-call-based anomaly-detection systems have been found to be susceptible to various mimicry attacks. Examples of these attacks can be found in [31, 32, 35]. Subsequently, a lot of work has been done to make system-call-based intrusion detection systems more resilent to mimicry attacks. However, system-called-based IDS are still vulnerable to different evasion techniques for which countermeasures incur expensive run-time overheads. In this

* The research reported here is supported partially by a grant from the Office of Naval Research under contract number N00014-03-1-0705

paper, we propose an approach for intrusion detection that is based on random inspection of application code execution. Our approach is complementary to system-call-based anomaly-detection in that evasion techniques that are effective against system-call-based detection are inherently vulnerable to detection by our approach. Our approach is motivated by the following observations:

1. Vulnerability to mimicry attacks can be attributed to the predictable nature of the monitoring mechanism: system-call-interface monitoring. Knowledge of when/where checking will occur puts the attackers in a very favorable position to launch mimicry attacks because they can "cover up" to make their behavior appear "normal" before making system calls. Furthermore, this monitoring mechanism does not preclude the attackers from exploiting execution with impunity in user space. For example, the "null calls" inserted by [13] can be found by an attacker who can then make those null calls accordingly to appear "normal".
2. Mimicry attacks usually take much longer than simple attacks that achieve their goals directly. To avoid detection, mimicry attacks have to spend a lot of extra effort in mimicking the normal behavior. In other word, the deployment of system-call-based IDS has the effect of significantly increasing the complexity and length of successful attacks. The example given in [35] clearly illustrates this point; in order to evade a very primitive system like pH [30], a simple attack of 15 system-calls has to be transformed into one with more than a hundred system calls. This seems to be unavoidable for any evasion to be successful.

Based on these two observations, we propose a different monitoring mechanism: random inspection. With random-inspection-based intrusion detection, we stop the execution of the monitored program at random points and observe its behavior. Based on the data collected at these random-inspection points, we determine whether an intrusion is in progress. Two major properties of random-inspection-based IDS are as follow:

1. The monitoring mechanism used by random-inspection-based IDS are less predictable to the attackers inasmuch as they cannot predict when/where a random inspection will occur, thus making it hard for mimicry attacks to evade.
2. Random-inspection-based IDS are in general more effective against long attacks. As the attack length increases, we can expect more inspections to occur when the attack is in progress. This means more data collected about the attack, and higher detection accuracy.

These properties make random-inspection-based IDS a strong complement to system-call-based IDS. In particular, the two types of IDS together present the attackers with a dilemma: in order to evade detection by system-call-based systems, the attackers will need to "mimic" normal behavior. This will significantly increas the length of the attacks. On the other hand, to avoid detection by random-inspection-based systems, the attackers should keep their attacks as

short as possible. As a result, when random-inspection-based systems are used in conjunction with system-call-based systems, it is very difficult (if not impossible) for the attacks to evade detection.

The effectiveness against long attacks also opens up the possibility of boosting random-inspection-based IDS with a new type of obfuscation techniques. Traditional obfuscation techniques as exemplified in [1, 2, 18] are designed to thwart attacks directly by making them unportable among different machines. On the other hand, obfuscation techniques designed to complement random-inspection-based systems will not have to stop all attacks. Techniques that create an unfamiliar (but still analyzable) environment will serve the purpose. In such an environment, extensive analysis will be needed for the attacker to achieve anything "interesting". This extensive analysis will significantly increase the length and complexity of attacks, which in turn makes them very visible to our random-inspection-based system. In addition to making attacks more visible and thus improving the detection rate, these new obfuscation techniques can also help reduce the performance overhead of random-inspection-based systems. We will elaborate on this point in Sect. 5.3. In fact, Windows is by itself a very "obfuscated" system to the attackers; we shall explain why the Windows environment makes attacks inevitably long in Sect. 4. This property makes Windows an especially suitable platform for our random-inspection-based IDS. We emphasize, however, that the applicability of our approach is not limited to Windows. We can apply other obfuscation techniques for machines running other operating systems.

Finally, to demonstrate the usefulness of random-inspection-based IDS, we have built a working prototype: the WindRain (WINDows RAndom INspection) system. The WindRain system focuses on code injection attacks on Windows systems. For this prototype, we adopt a very simple approach that checks the PC values at the inspection points and determines if the observed PC value is in a code region or a data region. If what is supposedly data is being executed, WindRain will mark it as an intrusion. Because of the way it utilizes collected PC values, WindRain is currently limited to code injection attacks. It cannot detect existing code attacks. However, we emphasize that this is only a limitation of the WindRain prototype and not a limitation to the general random-inspection approach we propose. WindRain is a very simple proof-of-concept system, and is not designed to show all the potentials of random-inspection-based intrusion detection. We stress that the PC value is not the only piece of information that an IDS can utilize at random-inspection points.

Our experiments show that WindRain is very effective against some "famous" code injection attacks against Windows. We have tested WindRain on MSBlast, Welchia, Sasser, SQLSlammer and Code Red, and all attacks are detected. As for false positive rate, we found that WindRain works well with most of the programs tested without generating ANY false alarm. In terms of performance, WindRain has low runtime overhead and allows for tradeoffs. Based on these results, we believe WindRain is a very strong stand-alone IDS in addition to being an excellent complement to system-call-based systems. Finally, our pro-

totype system also demonstrates another advantage of random-inspection-based IDS: it is easier to implement on Windows systems. The proprietary nature of the Windows kernel (with an undocumented interface that changes over different Windows versions, according to [29]) tends to make system-call interposition difficult. The extensive use of dlls in Windows further complicates the implementation of system-call-based systems on Windows, since most of the current systems do not work well with dlls.

2 Related Work

The idea of anomaly detection was first proposed by [7] in the 1980's. At that time, the only known mechanism for monitoring the behavior of processes is the audit-log. The kernel and other system components are responsible for monitoring process behavior and make this result available in audit-logs. The IDS will then read the audit-log and determine whether an intrusion is observed based on what is read. A new monitoring mechanism only came on the scene when [9, 15] proposed system-call-based anomaly detection. By using system-call traces for intrusion detection, an alternative monitoring mechansim, namely the monitoring of the system-call interface is implicitly introduced. Another major contribution of [9, 15] is the introduction of black-box-profiling technique. This is a technique that allows the normal behavior of a process to be profiled by just observing its normal execution. The process is treated like a black box since the availability of the underlying code being executed is not necessary. With this normal profile, we can check the monitored behavior of a process and identify any deviation from the profile as an intrusion.

Due to the richness and timeliness of the information available at the system-call interface, system-call-based anomaly detection has become a mainstream approach in intrusion detection. A lot of work has been done in enhancing system-call-based detection [20-22, 17, 25, 27, 37]; most of them focus on the profiling technique. At the same time, black-box profiling for the traditional audit-log monitoring mechanism has also received a lot of attention [4, 5, 12, 24, 38]. Despite all the work done in enhancing both system-call-based and audit-log based anomaly detection, the underlying monitoring mechanisms have remained largely the same. Monitoring at the system-call interface and monitoring through the system audit log facility are still the two mainstream monitoring mechanisms. There are some other monitoring mechanisms proposed (implicitly with the use of new observable behavior for anomaly detection, such as [3, 16, 39]), but none of these is as general as the two traditional approaches.

On the other hand, a lot of studies [31, 32, 35] have been done to find out the limitations and weaknesses of these system-call-based IDS. A lot of evasion strategies for avoiding detection have been identified. [35] presents a systematic analysis of these evasion strategies and introduces the notion of mimicry attacks. Afterwards, a lot of work has been done to overcome the weaknesses identified. The major focus of these approaches is to improve the accuracy of the profile for normal process behavior used for anomaly detection. With an inaccurate profile,

the IDS has to be more tolerant to behavior that deviates from that predicted by the profile. Otherwise, excessive false positive will result from the misprediction of normal, valid behavior. Unfortunately, this tolerance can be exploited to the attacker's advantage. With a more accurate profile, the IDS can be stricter in its enforcement and mark any slight deviation from the normal profile as an intrusion.

Among the work done in this direction, the work in [36] is one of the most exemplary. [36] first proposed the white-box-profiling technique. Instead of treating the process being profiled as a black box, we can build the profile based on analysis of the corresponding program. They have proposed several techniques for white-box profiling, varying in the accuracy of the profile, as well as the efficiency of run time monitoring. If the analysis is done correctly, white-box profiling guarantees zero false positive. As a result, we can avoid the false-positive-false-negative tradeoff mentioned above. However, high profile accuracy comes at the cost of higher complexity in runtime monitoring. Some of the most accurate profiling techniques proposed in [36] make it extremely difficult for the attackers to evade detection. Unfortunately, the monitoring overhead based on these profiles is likely to be high, owing to the nondeterministic nature inherent in profiles generated by program analysis. In general, monitoring in this way has extremely high complexity, and is so slow that it is impractical for monitoring in real time. Requiring the availability of source code is another major drawback of this work. This makes it impossible to apply their techniques to commodity software.

Some work [8, 13, 14, 23, 40] has been done in overcoming these two drawbacks. To tackle the problem of high monitoring overhead, some tried to optimize the profile generated. There are also proposals for the monitoring of other process characteristics that allows the differentiation of states that are seemingly the same. Some other works attempt to instrument the corresponding program so that it will report the needed context information during execution time. The problem of unavailability of source code is to be solved by binary code analysis and binary code instrumentation. Also, as is pointed out in [35], both input arguments and return values of system-calls are ignored in many system-call-based anomaly detection systems. Efforts to utilize the input and output of system-calls in anomaly detection are seen in [21, 22, 13, 14].

In addition to improving both profile accuracy and monitoring efficiency, many of these works propose new kinds of inputs for anomaly detection (e.g., return address, call stack information). Many of these new types of inputs are much harder to imitate by the attackers (as compared to system-call traces). This will also make the IDS built more resilent to mimicry attacks.

In some sense, WindRain, our prototype random-inspect-based system, is also like a specification-based intrusion detection system [28, 34]. The difference between WindRain and a specification-based system is that on WindRain, we have only specified one rule to govern the behavior of the entire system. On the other hand, for specification-based systems, a very detailed rule is devised for each individual process.

Two other related areas of work are instruction-set randomization [1, 18] and Program Shepherding [19]. One can regard our WindRain system as a probabilistic implementation of some Program Shepherding policies, targeting the same attacks as [1, 18]. The main advantage of WindRain over both instruction-set randomization and Program Shepherding is its smaller runtime impact; and WindRain is by default a system-wide protection mechanism. As a result, we believe that protection provided by WindRain is stronger than the by-process protection by Program Shepherding.

As mentioned before, research approaches that use obfuscation/diversification techniques as a means of defense are closely related to our work. The idea of using diversity in computer systems as a defensive measure is proposed in [6, 10]. The idea is demonstrated in the instruction-set-randomization systems [1, 18] and the address obfuscation system in [2]. In the case of WindRain, though we are not introducing any diversity, we do utilize the diversity amongst Windows systems to boost the effectiveness of WindRain.

Finally, our prototype WindRain traps intrusion by catching code executing in data space. In this respect, it is similar to that of the NX (or "Execution protection") technology. According to Microsoft's Security Developer Center, the NX technology "prevents code execution from data pages, such as the default heap, various stacks, and memory pool". Since the NX technology leverages hardware support from latest CPUs (including AMD K8 and Intel Itanium families), it has the obvious advantage over WindRain in terms of performance. On the other hand, WindRain (and random-inspection-based IDS in general) is applicable to legacy systems of which there are many, and more importantly, it is flexible as we shall explain below. Since NX is built on top of hardware features available only on new CPUs, it is obvious that NX cannot support legacy hardware. The problem with legacy software needs some elaboration. Even though it is reasonable to expect that executable code will never appear in "data space", some legacy software actually violates this rule. Some examples of these offending software include the JIT compiler in many JVM, as well as WindowsMediaPlayer and WindowsExplorer (more details on these software will be given in Sect. 5.2). In order to run these software on NX-protected systems, we will have to turn off the protection for these software. Another alternative is to mark all those data pages which contain code as executable. Both proposals are very coarse-grained solutions. In contrast, with the flexibility of a software solution, we can program our IDS to recognize the offending code that got placed in data space and accept their execution as normal. In fact, this is exactly our solution for supporting WindowsMediaPlayer and WindowsExplorer under WindRain. It is also possible for random-inspection-based IDS to judge whether the execution of "data" indicates an intrusion base on some addition information. A very good example is to base such decision on the execution history of the offending program. Once again, this solution demonstrates a level of flexibility that is impossible on NX. For NX, all that is available for this decision is a single point of data: the point where "data" is executed. Thus the introduction of NX does not solve all the problems that WindRain can solve.

3 Technical Details

In this section we present our proposed system for anomaly detection based on random inspection. We first discuss how random inspection is performed. Then the implementation details of our WindRain system, which performs anomaly detection based on the PC values collected at random inspection points, will be given. In the next section, we will take a look at the environment presented by Windows to the attackers. This will reveal the problems faced by the attackers and will explain why WindRain is an effective defense against code injection attacks. We present the results of our experimental evaluation of WindRain in Sect. 5.

3.1 The Core Random Inspection

Our implementation of random inspection makes use of a common hardware feature called performance counter. Performance counters are hardware registers that can be configured to count various processor-level events (e.g. cache miss, instructions retirement, etc). This facility is mainly designed for high-precision performance monitoring and tuning. Since events are counted by the CPU in parallel to normal operations, we can expect very low overhead for the counting. Furthermore, the CPU can be configured to generate an interrupt on any performance-counter overflow. As a result, by properly initializing the performance counters, we can stop the operation of the system after a certain number of occurrences of a particular event. By resetting the counter to its initial value at each counter overflow, we can configure the system to generate an interrupt at a roughly constant frequency. This turns out to be exactly what we need for random inspection: we can perform the inspections on counter-overflow interrupt, which occurs at a constant, controllable frequency. However, the inspection frequency is constant only in a system-wide perspective. The inspection frequency observed by individual process will appear randomized, as we will show later. It is also possible to make the occurence of inspections more unpredictable by resetting the counter with random values after each overflow.

In order to perform random inspection using the performance-counter facility, two more decisions have to be made: what event to count, and what initial counter value to use. For the choice of event to count, we want an event that occurs at high frequency in both normal and injected code. Furthermore, we want this event unavoidable in the injected code. The first criterion allows us more freedom in the choice of inspection frequency. The second criterion makes random inspection more robust: the attackers cannot evade inspection by avoiding the counted event.

For our implementation, we choose to count the instruction retirement events [1] that occur in user space. We believe this event satisfies the above criteria very well. Furthermore, by counting events in user space only, we guarantee

[1] Instruction retirement marks the completion of the out of order execution of an instruction and the update of processor state with its results

that inspection will only occur in user space. This allows easier utilization of information collected at inspection points.

For the initial counter value, we make it a configuration parameter of our system. By setting different values for this parameter, we can control the inspection frequency. In the following discussion, we shall name this parameter k. In addition to being the initial value for the performance counter, k also gives the number of instruction retirements that occur between two inspections. The choice of k involves different tradeoffs between detection rate, detection latency and performance overhead. We will talk about this tradeoff in Sect. 5.

We implement our prototype system on a machine with a Pentium III CPU. We note that performance counters that generate interrupt on overflow is very common in CPUs nowadays. Thus our idea is not limited to Intel CPUs. Furthermore, we find the use of this facility is limited to profiling software only, so our implementation will not disrupt normal system operation.

Finally, we would like to point out that Windows does not save counter values during context switches. In other word, the count stored in the performance counter is a system-wide count, instead of the count for current process since its last inspection. This is both an advantage and a disadvantage of our system. On the positive side, random inspection provides protection for the entire system by default. This is because inspection can occur in any process that executes in user space, thus no process will be left unprotected. Furthermore, this introduces randomness to our system and makes inspection unpredictable. Though we perform inspections at a fixed (and even possibly known) frequency, the attacker cannot predict when an inspection will occur. This is because process scheduling is non-deterministic in general, and thus it is impossible to determine when the attacked process will be scheduled to run. This means the attacker has no way to tell what the counter value is when the injected code starts executing[2]. In other word, the attacker cannot tell when the next inspection will occur. This randomness in inspection renders even extremely short injected code susceptible to detection with non-zero probability. On the negative side, this by default system-wide inspection implies inevitable inspection on many supposedly safe processes, which leads to some inefficiency. It is also impossible to perform inspection with different frequency for different processes. This problem can be solved if we can intercept context switches in Windows.

3.2 The WindRain System

After discussing how random inspection is actually achieved, we now show our implementation of intrusion detection under the random-inspection mechanism. In the following, we present the details of our WindRain system.

The most important component of the WindRain system is a device driver that runs on Windows systems. We have also written an application that loads the driver and displays data received from the driver in a timely manner (most

[2] Intel CPUs of P6 family or later can be configured so that performance counter values are readable only in kernel mode

importantly, notification about intrusions). The driver is responsible for setting up the system to perform random inspection, i.e., configuring the performance-counter facility. It also registers an interrupt-service routine to handle performance-counter overflow. This interrupt service routine is the part that actually performs intrusion detection.

On performance counter overflow, an interrupt is generated and the interrupt service routine registered will be called. The interrupt service routine starts by restoring the performance counter to its initial value, -k. It will then clear some flags so that the counter can start upon return to the user space. After that, the real intrusion detection starts. Among the arguments passed to the interrupt service routine is the PC value of the interrupted instruction. WindRain will determine whether that PC value corresponds to a memory location that holds code or one that holds data (in the latter case, WindRain will mark it as an intrusion). The decision is made by looking up a Windows internal data structure called Virtual Address Descriptor tree.

To keep track of the usage of the virtual address space in each process, Windows records information about each allocated (or "reserved") virtual memory region in a data structure called Virtual Address Descriptor (VAD). Among the information stored in the VAD are the start address, end address and the protection attribute for the corresponding memory region. To facilitate fast look-up, all VADs for a process are arranged as a self-balancing binary tree. Memory regions allocated for code usually have very different protection attributes from those for data (usually memory for code are copy-on-write, while memory for data are simply writable). As a result, given a PC value, we can search through the VAD tree of the corresponding process in an efficient manner. From the protection attribute of the VAD found, we determine whether that address contains code or data. If a PC value observed at an inspection point corresponds to a data region in memory, WindRain will mark it an intrusion. Currently, WindRain is a purely detection system, it does not have any capability to stop any intrusion from proceeding. Upon detecting an intrusion, the interrupt service routine will notify the application part of WindRain to display some information about the intrusion on the screen. Due to its inability to respond to attacks detected, WindRain is quite susceptible to DoS attacks. In other words, the attacker can try to turn off WindRain. We believe WindRain can perform reasonable self-defense when equipped with certain auto-response capability. Nonetheless, we believe the most ideal protection for WindRain (and possible any IDS) is from the underlying OS: having Windows consider WindRain as a core component (like lsass.exe, the termination/failure of which will lead to a system crash).

4 Analysis: Why WindRain Works?

Before we present the results of our experimental evaluation on WindRain, we first analyze the probability of WindRain detecting different code injection attacks. We will also discuss what makes it so likely for WindRain to detect intrusions.

The simplest way to perform this analysis is to consider inspection as a Poisson process, and calculate the probability that one or more inspection will occur during the entire execution of the injected code. Suppose we are performing inspection every k instructions (with $800 \leq k \leq 2400$), and the injected code requires the execution of y instructions. The probability of detection is then $P_d = 1 - P(0) = 1 - e^{-\frac{y}{k}}$.

The above analysis does not assume continuous execution of the injected code. Therefore the probability computed is valid even if context switching occurs during the execution of the injected code. It also applies to the case where the injected code calls some Windows library from time to time. A point worth noting here is that if an inspection occurs during the execution of a library function on behalf of the injected code, the intrusion will not be detected. Another very important point is that the above analysis is only valid if the attacker cannot predict when the next inspection will occur. Otherwise, it is (in theory) possible for the attacker to evade detection by calling certain library functions when an inspection is expected.

We should note that the Poisson-based analysis is overly pessimistic. Suppose the injected code executes without making library calls for an interval that we call “very visible period” (VVP). Let us make the following assumptions about this VVP:

1. this interval is more than k instructions long
2. context switch occurs in the first k instructions of this VVP with probability less than 1%

With these two assumptions, we argue that the actual detection probability $P_{d1} \geq 0.99 + 0.01 * P_d$, where P_d is the detection probability predicted for the corresponding k and y by our initial Poisson analysis. This is because in 99% of time, no context switch occurs in the first k instructions of the VVP. Since the injected code is “trapped” in the VVP for more than k instructions, we can guarantee an inspection will occur while the injected code is executing in “data space”. In this case, WindRain will detect the attack with probability one. The second term of P_{d1} accounts for the remaining 1% of time where a context switch does occur in the VVP and we have to fall back to our Poisson analysis.

In the following, we shall validate our two assumptions about the VVP and thus show that $P_{d1} \geq 0.99$.

We start with defending our first assumption. From our study of Windows shellcode, we find that they usually arrive encoded. This helps the shellcode evading signature based IDS and systems like [33]. As a result, before performing any “interesting” activities, the injected code has to decode itself first. This decoding has complexity linear to the injected code’s length, and can take up a few hundred instructions. Since there appears no library function for this decoding process, the injected code will not execute any library function during the decode phase.

A more important reason why the injected code does not execute any library functions is that it may not know the address of any library functions. Due to the extensive use of dlls in Windows, the addresses of library functions vary

across different machines. This is a very well known fact in the black-hat society [29]. As a result of the dynamic nature of library loading, static address values cannot be used for library calls. Otherwise, there will be portability issues for the resulting shellcode. As a result, in order to execute any library functions, the injected code has to dynamically search for the needed function addresses. As discussed in [29], in order to do this in a portable manner, the complexity of the library-function-locating process is usually linear to both the number of functions in the desired library and the length of each function name. Such complexity will imply a very significant number of instructions executed before finding the address of one single library function.

From our discussion above, portability is the major issue that "traps" the injected code in its VVP for an extensive amount of time. So a natural question is: is it possible for the shellcode to sacrifice certain portability to speed up this process and evade detection by WindRain? At first sight, it appears to be a feasible solution for the attacker: certain library functions do stay in the same address across a large number of machines. Furthermore, there are various values related to function addresses that are static over different Windows versions. It is thus possible to utilize these static values to speed up the process to constant time and evade detection. In fact, the IAT technique given in [29] implements this idea.

However, we argue that any approach of this kind can be thwarted with simple obfuscation techniques. This is because Windows does not depend on these values being static to function properly. As a result, any obfuscation of these values can impose serious portability problem in the shellcode, without adversely affecting the operations of Windows. For example, any shellcode that uses hard-coded address for library functions can be thwarted by a simple application that rebase every library on the system. In this case, a shellcode that works for one machine will almost guarantee to fail on another.

From the above analysis, we see that is it very likely that an injected code will execute more than k instructions without executing any library calls. We will further validate this assumption with our experimental results in the next section. We now move on to the second assumption: context switch occurs very rarely in the VVP where no library calls are made.

Since the injected code is not making any library call, it is impossible for it to get blocked. Thus the only reason for a context switch is the expiration of a time slice. Now consider the following very conservative figures:

1. time slice in Windows ranges from 10ms to 200ms
2. Intel Pentium processor achieves 90 million instructions per second

From these two figures, we can assume that at least 900000 instructions will be executed on any Windows machine before a time slice expires. Let us model time-slice expiration as a Poisson process; the probability of expiration is 1/900000 at any time. The probability that a context switch will occur in the first k instructions of the VVP is then given by $P_{switch}(0) = 1 - e^{-\frac{k}{900000}}$. With $k \leq 2400$, we have $P_{switch}(0) \leq 0.01$. Thus, we have validated our second assumption.

As a result, we have shown that any injected code that executes more than k instructions in their VVP will be detected by WindRain with probability close to one. Our argument also shows that this is usually true for injected code.

5 Experimental Results

In this section, we will present the results of our experiments on WindRain. The experiments attempt to evaluate WindRain at different inspection frequencies. The evaluations focus on the following three aspects: false negative rate, false positive rate and performance overhead.

5.1 False Negative Rates

We have tested WindRain's ability to detect MSBlast, Sasser, SQLSlammer, Code Red and Welchia (aka Nachi). The experiments are carried out at three different inspection frequencies: once every 800 instructions, once every 1600 instructions and once every 2400 instructions. For each inspection frequency, we repeated each attack 5 times, and WindRain is able to detect all the attacks for all three configurations. In addition to testing whether WindRain can detect the attack attempts, we are also interested in verifying our assumption in the previous section, namely, that injected code executes a large number of instructions in their VVP, without executing any library calls. We validate this assumption by noting when WindRain first detect each of the 15 attack trails. The results of our experiments are presented in Table 1.

Table 1. The following table shows when WindRain first detects the attacks when configured at different inspection frequencies. The three rows show the results for three different inspection frequencies: once every 800, 1600 and 2400 instructions respectively. For the entries of each row, "Decode" means WindRain detects the attack when the injected code is decoding itself. "FindLib" means the attack is detected when the injected code is resolving the addresses of library functions needed. "Spread" means the attack is detected when it tries to infect other hosts. Each attack is repeated five times for each inspection frequency, the number in the bracket indicates how many times the attack is detected in the particular stage.

	MSBlast	Welchia	Sasser	SQLSlammer	Code Red
800	FindLib(5)	Decode(3), FindLib(2)	Decode(3), FindLib(2)	Spread(5)	FindLib(5)
1600	Decode(2), FindLib(3)	FindLib(5)	Decode(3), FindLib(2)	Spread(5)	FindLib(5)
2400	Decode(1), FindLib(4)	Decode(2), FindLib(3)	Decode(1), FindLib(4)	Spread(5)	FindLib(5)

From our analysis of the above data, we are certain that the VVP of Welchia, Sasser and CodeRed contain more than 2400 instructions. This is because both "Decode" and "FindLib" for these worms are used exclusively in their VVP

(while "FindLib" is used outside the VVP of MSBlast also). According to our analysis in Sect. 4, this implies a detection probability close to 1 when WindRain performs at least one inspection every 2400 instructions executed. We are also pretty certain that WindRain cannot detect SQLSlammer in its VVP. However, we are very certain that WindRain will detect any instance of SQLSlammer with probability one. This is because the injected code is the entirety of the SQLSlammer payload and runs on the stack indefinitely long. In fact, this is also the case for CodeRed.

However, since both MSBlast and Welchia use their library-function-locating code more than once, it is possible for future variants to use this code more efficiently to evade detection. This is because once the "GetProcAddress" function in kernel32.dll is located, address of any other library functions can be resolved using this function (in fact, this is the method used by Sasser and CodeRed). In order to clear such doubt, we experimented with the library-function-locating code of the two worms. In our experiments, we copied the piece of code under concern (with arguments for searching the "GetProcAddress" function) onto the stack and execute them. We repeated each experiment 5 times, with WindRain performing an inspection every 2400 instructions, and see if it can detect the "attack". WindRain successfully detects all the 10 "attacks". Thus we are pretty certain that WindRain can detect both MSBlast and Welchia in their VVP, even if they are modified to make more efficient use of their library-function-locating code.

Another way to increase our confidence in WindRain's ability to detect the five worms is to decrease the inspection frequency and see if it can still achieve 100% detection rate. Since SQLSlammer and CodeRed execute in "data space" forever, we find it unnecessary to perform such test for these worms. For both MSBlast and Welchia, we find that WindRain still achieves 100% detection when configured to perform an inspection every 24000 instructions executed (as before, we repeated each attacks 5 times). However, for Sasser, we tested WindRain by increasing the interval between two inspections with step of 800 instructions. We start with an inspection frequency of once every 2400 instrctions. We find WindRain miss the first attack when performing inspection once every 7200 instructions. Among the five attacks tried at this frequency, only one is missed.

In conclusion, we are very confident that WindRain can detect the five worms tested with probability very close to one. This is true even when WindRain is performing inspection at a low frequency of once every 2400 instructions. Furthermore, by detecting MSBlast, Welchia, Sasser and CodeRed in their VVP, WindRain can guarantee to detect these attacks before they can cause any real damage to the system. This is because in Windows, kernel services are only accessible through library functions. Thus the Windows kernel is inaccessible to the injected code when it is still in its VVP.

Finally, it appears possible to shorten the VVP of the tested worms by improving their implementation. Nonetheless, we believe the underlying decoding and library-function-locating algorithms will continue to have linear complexity. With this observation, an inspection frequency of once every 800 instructions

should be sufficient to detect future injected code that is optimized to have short VVP. However, we shall argue in Sect. 5.3 that the best way to guard against these threats is to complement our system with obfuscation techniques.

5.2 False Positive Rates

We have evaluated the false positive rate of WindRain by performing some daily activities with WindRain running on the background. In all of our tests, WindRain is configured to perform an inspection every 800 instructions. By experimenting WindRain at such high inspection frequency, we have established a worst-case false-positive rate. We expect the false-positive rate will only drop when we decrease the inspection frequency of WindRain. Another reason for choosing the number 800 is that we believe this inspection frequency is high enough to detect most attacks with very high probability.

The daily activities we have tested include: surfing the web (using IE), reading PDF files, creating word documents (using MSWord and Wordpad), viewing PowerPoint presentations, connecting to a remote machine (using Telnet), compiling the entire WindRain system (using the MS VisualStudio for the application part, and the MS DDK for the device driver part), file management (under WindowsExplorer), playing MP3s (using WindowsMediaPlayer) and Quicktime movies (using QTPlayer) and using a bunch of GNU tools that comes with cygwin (including all the utilities tested in the performance analysis). Finally, we have also tried compiling and running Java programs while WindRain is running.

The first false positives identified are from WindowsMediaPlayer and WindowsExplorer. We find both WindowsMediaPlayer and WindowsExplorer execute small fragments of code on the heap, which cause the false alarms. The violating code executed on the heap are thunks that pass control to some callback functions. This turns out to be a well documented “workaround” to pass the “this” pointer of C++ objects to callback functions. This technique allows instance methods to be called by the callback mechanism. To tackle this problem, we modified WindRain to recognize the structure of the thunk and make sure it is passing control to some callback function. After the modification, there are no more false alarms from these applications.

We have also observed occasional false positive when Microsoft software prompts us to activate/register their products. We believe this is a technique to avoid bypassing the corresponding check and use the software without activation/registration. The major offender in this category is winlogon.exe, which keeps prompting the user to activate Windows. No more false positives are observed after we activate Windows.

Finally, we find that both the compilation and execution of Java programs will lead to false positives in WindRain. The false positives from executing Java application is caused by the JIT compilation in the underlying JVM. When native code is generated during runtime, they are kept in writable memory areas. This is mainly for efficiency reasons and allows new native code to be written without first unprotecting the corresponding pages. The execution of these dynamically-generated code will lead to false positives from WindRain. For the

compilation of Java code, we observe that the Java compiler uses code from the JVM, which may explain the problem. We believe, in general, WindRain does not handle programs that use dynamically-generated/self-modifying code very well. A possible solution is to perform profile-based anomaly detection using the library/function-calling pattern of the monitored program. In this solution, instead of determining whether "data" is being executed, the IDS will keep track of the function-usage pattern of the monitored program. By building a normal profile of this pattern, the IDS can check if the observed function usage is normal. Any abnormal behavior is marked as intrusion. We believe this approach is also useful in detecting existing-code attacks.

One last point about our experiments is that by running WindRain in a Windows system, we have implicitly tested WindRain against those Windows system processes. For system processes, we mean processes Windows created by itself, including svchost.exe, lsass.exe, etc. In fact, it seems that these processes are the ones that needed most protection. Our results show that WindRain has zero false positive for all these processes after the user has activated Windows with Microsoft. Thus it is possible to modify WindRain to only report intrusions concerning these processes. This modification will allow WindRain to provide very useful protection to some major threats against Windows systems, while maintaining a zero false-positive rate.

5.3 Performance Overhead

Since WindRain does not need to keep any record about different processes, its memory footprint is very small and is constant. The entire device driver (including the interrupt service routine and all other code) is just 20KB. In other words , WindRain has minimal space overhead. However, the frequent execution of the interrupt service routine at inspection points can cause substantial overhead in terms of execution time. In this section, we report WindRain's runtime overhead when tested on several programs in the SPEC2000 benchmark suite. The effect of the inspection frequency on the runtime overhead is studied by running the benchmarks under 9 different WindRain configurations.

Before we present the experimental results, let us briefly describe our experiments. Each benchmark program is executed six times. They are executed on an otherwise idle Windows system with all the Windows system processes running on the background. From the execution times measured for the six runs, one outlier is removed. This helps to avoid any fluctuation in the measured values from affecting our results. We establish the base execution time of the benchmark program by averaging the remaining five data points. For each inspection frequency studied, the process is repeated with WindRain running under the corresponding configuration. Again, the execution time is measured six times for each benchmark, and one outlier is removed to obtain five data points. The averaged execution time is compared against the base execution time to obtain the overhead caused by WindRain at the inspection frequency being tested. The results of our experiments is shown in Fig. 1.

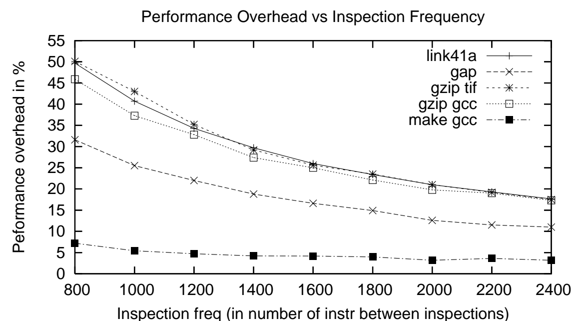

Fig. 1. Performance overhead of WindRain at different inspection frequency: y-axis is the overhead in %, x-axis gives the inspection frequency in number of instructions executed before an inspection occurs

From the results in Fig. 1, we see that the performance overhead drops quite significantly as the inspection frequency decreases. This shows a significant tradeoff between detection rate and performance overhead. Such tradeoff once again highlights the value of obfuscation techniques that lengthen the VVP of future injected code. Consider an obfuscation technique that guarantees any injected code will spend at least 2000 instructions locating the needed library functions. With such guarantee, we can perform an inspection every 2000 instructions and still guard against optimized injected code at an overhead of around 20%. Otherwise, we will have to guard against these future attacks by increasing the inspection frequency at the cost of higher performance penalty. However, even when performing random inspection once every 800 instructions, the performance overhead of WindRain still compares favorably against many system-call-based IDS. According to [8, 27], system-call-based systems typically incur more than 100% overhead in the interposition of system calls alone, unless the kernel is modified for the task.

To study how much overhead is contributed by the random-inspection process alone, we studied the performance overhead of a system that performs random inspection without the PC-value checking. We compared the performance overhead of both WindRain and the “empty” system at three inspection frequencies: once every 800, 1600 and 2400 instructions respectively. Due to space limitation, we omit the raw data of our experiments and simply report our findings below.

We find that a large proportion of the overhead (more than 89% in all our experiments) comes from performing random inspection. On the other hand, the checking of PC values obtained from random inspections only slightly increases the overhead. This result demonstrates the feasibility of performing more sophisticated checking at each inspection point. For example, one would expect the checking of the return address of the current stack frame to incur very small extra overhead. This finding also allows us to conclude that the overhead is mainly contributed by the side effect of random inspection, instead of perform-

ing the PC checking. This side effect includes the flushing of pipelines and the consumption of extra instruction cache. We have also measured the effect of random inspection on different cache miss rate and the paging rate. Our experiments show no significant increase in these measures while performing random inspection. As a result, we strongly believe that the flushing of pipeline caused by the frequent performance-counter overflow and subsequent interrupt handling is the major cause of the high overhead. Pipeline flushing is also identified as a major cause of overhead in system-call interposition systems.

6 Conclusions and Future Work

In this paper, two problems of system-call-based anomaly detection systems are discussed: its inherent vulnerability to mimicry attacks and its being non-portable for the widely deployed Windows systems. These weaknesses have their roots in monitoring at the system-call interface and the predictability thereof to the attacker. Since this monitoring mechanism is shared by all system-call-based systems, it is difficult to completely overcome these difficulties without having an alternative and complementary mechanism. We propose random inspection as an alternative monitoring mechanism. We demonstrated that random inspection can be implemented on Windows without requiring knowledge or modification of the Windows kernel. Furthermore, owing to its random nature, random-inspection-based intrusion detection is inherently less susceptible to mimicry attacks. Random-inspection-based intrusion detection is a strong complement to the more traditional system-call-based intrusion detection systems. Together these two types of IDS require attackers to deal with two conflicting constraints. In order to evade detection by random-inspection-based systems, the attacks need to be short. On the other hand, to evade detection by system-call-based IDS, attacks must be more complicated and therefore take longer to execute. Random-inspection-based systems also provide a second line of defense for systems that depend on obfuscation/diversification as the main line of defense. With our random-inspection-based detection as a complement, even obfuscation/diversification techniques that are susceptible to reversal by an attacker can become very useful defense mechanisms. In particular, random-inspection-based detection will make the design of obfuscation techniques easier. In reciprocal, both traditional system-call-based systems and obfuscation techniques can complement random-inspection-based systems by forcing intruders to lengthen the attacks. This will allow random inspection to be performed at lower frequency while still maintaining a very high detection rate and a lower frequency implies a lower performance overhead.

To demonstrate the usefulness of random-inspection-based detection, we have implemented a working prototype: the WindRain intrusion detection tool. Our prototype performs random inspection on the PC value of the instruction being executed. If the inspected PC value corresponds to a region of memory that contains data, WindRain will mark it as an intrusion. Despite being a very simple system, our analysis shows that WindRain can detect most of the injected

code attacks with a very high probability. We have tested several attacks against WindRain (namely, MSBlast, Welchia, Sasser, Code Red and SQLSlammer, all famous attacks against Windows systems). We found that WindRain can detect all the attempted attacks very effectively. This is even true with the lowest inspection frequency tested. In terms of false positive, we found that WindRain generates few false alarms for all but two applications we have tested, the Java compiler and the JVM. Furthermore, WindRain was found to work well with all the Windows system processes without raising any false positive. This makes WindRain very suitable for system-wide protection. In terms of performance overhead, WindRain compares favorably against many other intrusion detection systems, even when performing inspections at a very high frequency.

We consider our work in this paper as an illustration of the usefulness of random-inspection-based intrusion detection systems. There is a lot of interesting work to be done in both enhancing the idea of random-inspection-based detection and extending the capability of WindRain.

For the improvement of WindRain, we are working on solutions that allow WindRain to work with dynamically-generated/self-modifying code (like those generated by JVM). We believe the approach outlined in the Sect. 5.2 is very promising. We are also interested in ways to turn off WindRain for non-critical processes so that only critical processes incur the performance overhead from WindRain. A possible direction would be to capture Windows context switch and reconfigure WindRain accordingly. We have some preliminary evidence of success on this. We note that the software approach we take allows us to attack these problems in ways that the inflexibility of hardware-based technology such as NX would have a much harder time to emulate.

In terms of the development of random-inspection-based systems, we are interested in studying what kind of information is available at the random-inspection points, and how to make use of it. An interesting direction of research is to design profile-based intrusion detection systems under the random-inspection mechanism. The profile-based approach will allow us to protect programs that use dynamically generated code without generating too many false positives. It is also a promising approach to tackle existing code attacks. We believe our work has opened up new directions for research of obfuscation techniques that can be used as defensive mechanisms. With the complement of random-inspection-based systems, new obfuscation techniques do not have to thwart attacks directly. They only need to make attacks significantly more complicated and visible to random-inspection-based detection. The work in [2] about address obfuscation is a very good example in this direction. Another interesting example is to reproduce the harsh Windows environment (where the kernel interface is unknown) on Linux. This can be achieved by randomizing the mapping between the system-call number and the corresponding kernel service. If we obfuscate the kernel interface, we can avoid injected code from making direct calls to the kernel. As a result, injected code will have to go through the long library-function-locating process as on Windows. Thus this obfuscation technique will

allow injected code attacks to be detected easily by random-inspection-based systems like WindRain.

References

1. Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, Trek S. Palmer, Darko Stefanovic and Dino Dai Zovi, *Randomized instruction set emulation to disrupt binary code injection attacks*, 10th ACM International Conference on Computer and Communications Security (CCS), pp. 272 - 280. October 2003.
2. Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar, *Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits*, 12th USENIX Security Symposium, 2003.
3. F. Buchholz, T. Daniels, J. Early, R. Gopalakrishna, R. Gorman, B. Kuperman, S. Nystrom, A. Schroll, and A. Smith, *Digging For Worms, Fishing For Answers*, ACSAC 2002.
4. Sung-Bae Cho, and Sang-Jun Han, *Two Sophisticated Techniques to Improve HMM-Based Intrusion Detection Systems*, RAID 2003.
5. Scott Coull, Joel Branch, Boleslaw K. Szymanski, and Eric Breimer, *Intrusion Detection: A Bioinformatics Approach*, ACSAC 2003.
6. Crispin Cowan, Calton Pu, and Heather Hinton, *Death, Taxes, and Imperfect Software: Surviving the Inevitable*, theNew Security Paradigms Workshop 1998
7. Dorothy E. Denning, *An intrusion detection model*, IEEE Transactions on Software Engineering, 13-2:222, Feb 1987.
8. Henry H. Feng, Oleg Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong, *Anomaly Detection Using Call Stack Information*, IEEE Symposium on Security and Privacy, 2003.
9. S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff, *A sense of self for UNIX processes*, IEEE Symposium on Security and Privacy, 1996.
10. S. Forrest, A. Somayaji, and D. Ackley, *Building Diverse Computer Systems*, Proceeding: 6 workshop on Hot Topics in Operating Systems, IEEE Computer Society Press, pp. 67-72.
11. Tal Gar nkel, *Traps and pitfalls: Practical problems in in system call interposition based security tools*, Proc. Network and Distributed Systems Security Symposium, February 2003.
12. Anup K. Ghosh, Christoph Michael, and Michael Schatz, *A Real-Time Intrusion Detection System Based on Learning Program Behavior*, RAID 2000.
13. Jonathon T. Giffin, Somesh Jha, and Barton P. Miller, *Detecting manipulated remote call streams*, 11th USENIX Security Symposium, 2002.
14. Jonathon T. Giffin, Somesh Jha, and Barton P. Miller, *Efficient context-sensitive intrusion detection*, 11th Network and Distributed System Security Symposium, 2004.
15. S. A. Hofmeyr, A. Somayaji, and S. Forrest, *Intrusion detection using sequences of system calls*, Journal of Computer Security, Vol. 6, 1998, pp. 151–180.
16. Ruiqi Hu and Aloysius K. Mok, *Detecting Unknown Massive Mailing Viruses Using Proactive Methods*, RAID 2004.
17. A. Jones and S. Li, *Temporal Signatures of Intrusion Detection*, ACSAC 2001.
18. Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. *Countering Code-Injection Attacks With Instruction-Set Randomization.* 10th ACM International Conference on Computer and Communications Security (CCS), pp. 272 - 280. October 2003.

19. V. Kiriansky, D. Bruening, and S. Amarasinghe, *Secure execution via program shepherding*, 11th USENIX Security Symposium, 2002.
20. C. Ko, *Logic Induction of Valid Behavior Specifications for Intrusion Detection*, IEEE Symposium on Security and Privacy, 2000.
21. C. Kruegel, D. Mutz, F. Valeur ,and G. Vigna, *On the Detection of Anomalous System Call Arguments*, 8th European Symposium on Research in Computer Security (ESORICS), 2003.
22. Christopher Kruegel, Darren Mutz, William Robertson, and Fredrik Valeur, *Bayesian Event Classification for Intrusion Detection*, ACSAC 2003.
23. Lap Chung Lam and Tzi-cker Chiueh, *Automatic Extraction of Accurate Application-Specific Sandboxing Policy*, RAID 2004.
24. T. Lane and C. Brodley, *Temporal Sequence Learning and Data Reduction for Anomaly Detection*, ACM Trans. Info. and Sys. Security, 1999.
25. W. Lee and S. Stolfo, *Data Mining Approaches for Intrusion Detection*, 7th USENIX Security Symposium, 1998.
26. p62_wbo_a@author.phrack.org, Jamie Butler, and p62_wbo_b@author.phrack.org, *Bypassing 3rd Party Windows Buffer Overflow Protection*, Phrack, Issue #62, of July 10, 2004
27. R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, *A Fast Automaton-based Method for Detecting Anomalous Program Behaviors*, Proceedings of the 2001 IEEE Symposium on Security and Privacy.
28. R. Sekar, A. Gupta, J. Frullo, T. Shanbhag, A. Tiwari, H. Yang, and S. Zhou, *Specification based anomaly detection: a new approach for detecting network intrusions*, ACM Computer and Communication Security Conference, 2002.
29. skape, *Understanding Windows Shellcode*, http://www.hick.org/code/skape/papers/win32-shellcode.pdf
30. A. Somayaji, S. Forrest, *Automated Response Using System-Call Delays*, 9th Usenix Security Symposium, 2000.
31. Kymie M. C. Tan, and Roy A. Maxion, *"Why 6?" Defining the Operational Limits of Stide, an Anomaly-Based Intrusion Detector*, IEEE Symposium on Security and Privacy 2002.
32. Kymie M. C. Tan, Kevin S. Killourhy, and Roy A. Maxion, *Undermining an Anomaly-Based Intrusion Detection System Using Common Exploits*, RAID 2002
33. Thomas Toth, Christopher Krugel, *Accurate Buffer Overflow Detection via Abstract Payload Execution*, RAID 2002.
34. P. Uppuluri and R. Sekar, *Experiences with Specification-Based Intrusion Detection*, RAID 2001
35. D. Wagner and P. Soto, *Mimicry Attacks on Host-Based Intrusion Detection Systems*, ACM Conference on Computer and Communications Security, 2002.
36. D. Wagner and D. Dean, *Intrusion Detection via Static Analysis*, IEEE Symposium on Security and Privacy, 2001.
37. Christina Warrender, Stephanie Forrest, and Barak Pearlmutter, *Detecting intrusions using system calls: alternative data models*, IEEE Symposium on Security and Privacy, 1999.
38. A. Wespi, M. Dacier and H. Debar, *Intrusion detection using variable-length audit trail patterns*, RAID, 2000.
39. Matthew M. Williamson, *Throttling Viruses: Restricting propagation to defeat malicious mobile code*, ACSAC 2002.
40. Haizhi Xu, Wenliang Du and Steve J. Chapin, *Context Sensitive Anomaly Monitoring of Process Control Flow to Detect Mimicry Attacks and Impossible Paths*, RAID 2004.