Results-Oriented Security

Matt Bishop
Dept. of Computer Science
University of California at Davis
Davis, CA 95616-8562 USA
bishop@cs.ucdavis.edu

Richard Ford
Harris Institute for Assured Information
Florida Institute of Technology
150 W. University Blvd, Melbourne, FL 32901 USA
rford@fit.edu

Marco Ramilli
D.E.I.S.
University of Bologna
Via Venezia, 52 - 47023 Cesena ITALY
marco.ramilli@unibo.it

Abstract

Current security practice is to examine incoming messages, commands, data, and executing processes for attacks that can then be countered. This position paper argues that this practice is counterproductive because the number and variety of attacks are far greater than we can cope with. We propose a results-oriented approach, in which one focuses on the step of the attack that realizes the compromise. Thus, the manner *in which the compromise is effected becomes less important than the actual* result*, and prevention, detection, and recovery efforts are focused on that.*

1 Introduction

Computer security practitioners have long studied how attacks work. Their goals have been twofold. First, to understand the attack in order to detect it; what are the particular characteristics of the input (whether it be typed commands, commands from a network, malicious or malformed data, or other inputs) that allow someone to identify it as dangerous? Second, how do we foil the attack—for example, by disrupting some aspect of the attack to prevent its consummation? This "observe and analyze" approach also may reveal information about the source or the nature of the attack, such as the location of the attacker.

Beneficial as it is, this approach implies that exploitation is a discrete event. It also assumes that the characteristics of the attack *can* be detected. This means that they are either unusual in some way, or known in some way. The first (unusual) is the basis for anomaly-based intrusion detection methods, as well as the more informal "hey, this doesn't look right!" feelings that lead system managers to uncover insider attacks. The second (known) is the basis for both misuse detection, in which a mechanism looks for the "signatures" of the attack, and for specification detection, in which a mechanism looks for any unexpected behavior as evidence of an attack. In all three cases, the defenders must know what is "usual," what is "bad," or what is "good."

While one would expect these to be known in advance, such an expectation is unrealistic. New attacks evade misuse detection and incomplete or incorrect specifications cause specification-based detection to miss attacks. Establishing a baseline for "normal" or "expected" requires a set of attack-free data and an environment in which users' profiles change either predictably or not at all which typically is not true in ones where detecting attacks is critical.

An alternate approach is to look for the *results* of the attack. What does the attack do? One can consider the result at a high level of abstraction—for example, enabling an unauthorized user to obtain `root` access—and look for that. The problem is the myriad of ways that such a goal can be attained (compromise of a `sudo` configuration file, alteration of a `passwd` file, exploiting a buffer overflow, *etc.*) make monitoring the high-level goal very difficult. But monitoring a low-level goal, for example one of the ways in which the high-level goal can be attained, is eminently practical. Doing so offers several advantages over detecting attacks.

First, in order for an attack to be successful, it must attain some goal. If we can detect the attempt to attain the low-level goal, we identify that an attack is occurring. From that, we can often work backwards to determine what the attack was. Second, by focusing our resources on the goal, whether we detect the specific attack while it is under way or not is irrelevant. The key point is that , should an attacker come up with a new way to realize the goal, or modify an existing attack to evade current detection mechanisms, we lose little because we still detect the result of the attack. Third, we are forced to analyze what we are trying to protect, in order to determine the (low-level) goals. This has a salutary effect on our understanding of the security of the system.

Throughout this paper, we assume that one can describe high-level events that are considered security breaches; these events are the "goals" of the attacker. We note that characterizing (or even identifying) all such goals is a non-trivial problem endemic to all of security; security policies are rarely exhaustive *a priori*, but rather are defined *a priori* and then modified as experience demonstrates new compromises, and as new needs and requirements of the organization are identified. Thus, the application of results-oriented security must evolve to reflect the current set of security threats and requirements, and consequently the current set of attacker goals. This problem arises in other contexts, notably specification-based intrusion detection [14].

It is important to distinguish our work from deriving vulnerabilities, which has been the focus of much recent work [4–6, 18]. For us, the details of the attack are irrelevant. What matters is what the attack ultimately *does*. Knowing about vulnerabilities and attacks helps when analyzing potential goals of attacks, which augment policy and goals. However, for detection, we focus solely on the intermediate and final goals of the attackers.

2 Attack Detection

When we think about attack detection, we tend to think about determining whether something is "bad" or "good". "Good" is not designed to harm us, and "bad" is. To protect a system, simply decide what is bad and what isn't.

Unfortunately, distinguishing between "good code" and "bad code" using static analysis is anything but simple. Cohen [2] demonstrated this was equivalent to the halting problem. This is further complicated by attacks such as buffer overruns that blur the lines between executable code and data, making it unclear how the system will actually handle a particular piece of input without viewing it in context.

Despite these difficulties, our existing protection techniques generally focus on this sorting process. Virus scanners work by attempting to detect patterns of malicious behavior in emulated environments, or by looking for signatures of known-malicious sections of code (for an overview, see Szor [28]). Recently, vendors have begun leveraging the cloud and the presence of ubiquitous connectivity to see how well known a particular file is (the idea being that common files are less likely to provide the user with an unpleasant surprise), but once again, this relies on being able to tell that a piece of binary data is executable.

While these approaches provide a powerful way of reasoning about the probable danger from executable objects, many exploits do not enter the system as executable files. Thus, the protective measures must also examine data for its risk, making these defenses yet more difficult. Data is not supposed to stay the same; it changes from request to request. Data is often strangely structured. Data is, essentially, messy.

Technologies like deep packet inspection [10, 12, 22] and detection of embedded shell code [8, 23] can help, but given the intimate interaction of the data with the host system, it is very difficult to predict behavior *in situ* reliably [15, 27]. Nevertheless, our networks are architected around this "attack detection" paradigm, where we base our defenses on the fallacy that we can reliably sort good from bad at the border of our system or our site.

Given the challenges of detection based on appearance, people have put significant effort into detecting malicious behavior within services and executables based on their actions. Behavioral analysis examines what actually happens on the machine, typically on a per-process basis, to determine if an attack is underway. For many applications, this is quite effective. For example, behavioral detection works well for virus detection and prevention [7, 13].

While behavioral analysis ameliorates some of problems, the approach still focuses on sorting out the good from the bad, and detecting and stopping attacks within a narrow temporal and spatial window. The Shield system [31] provides an excellent example of what behavioral detection methods can do. Shield uses runtime instrumentation to detect when the system is in a position where a vulnerability is about to be exploited, and then modifies execution at this point. This works for known attacks, but it does not handle unknown attacks.

Finally, the success of zero-day attacks indicates that current attack detection methods are far from complete. Zero-day attacks exploit previously unknown vulnerabilities. According to McQueen *et al.* [16], the number of such vulnerabilities in existence on any given day may be as high as 2500. Not knowing these vulnerabilities, we cannot characterize attacks that exploit them.

So, instead of looking for conditions showing that a particular attack is under way, we view the system holistically, searching for changes of state we care about.

3 Result Detection

THe detection methods above depend upon the attack taking specific steps. Therein lies the problem: if the attacker can disguise these steps, the detection mechanisms become ineffective. Either they generate a large number of false positives, or are susceptible to false negatives.

Consider an attack to be a sequence of steps $A = a_1 \dots a_n$. An attack detection mechanism looks at the sequence and finds a matching subsequence $B = a_i \dots a_{i+k}$. Ideally, B is unique to A; that is, it will only occur when A occurs. We now construct a generic method for evading the detection of B. As the attack detection mechanism looks for B in a particular environment, we must disrupt the mechanism's ability to recognize B. There are two ways to do this.

First, replace B with something equivalent but sufficiently different to avoid detection. For example, computer malware that introduces variability using encryption aims at achieving this goal. The response has been to look not for the malicious code but for the decryption routines; and the counter is to make these decryption routines polymorphic.

Second, disguise B in some fashion so that the detection mechanism cannot identify B. For example, most anti-malware software looks for previously-identified patterns in single files, or subsequences of behavior (actions) in a single process or a process and its descendants. In the first case, one can simply rewrite the malware so that the subsequences are split over multiple files, and not reassembled until execution. In the second case, independent processes can execute different parts of the sequences of actions, coordinated so that the result is the same as if the sequence of actions had been executed directly. This approach is not confined to malware; Wagner and Soto [30] show how to do this to evade many host-based intrusion detection systems.

More formally, consider a requires/provides model of attack detection [29]. In such a model, the component parts (bytes forming the attack, or actions forming the attack) are leaves in a tree. As each leaf is found, the attack meets requirements of intermediate steps, and these intermediate steps are represented as internal nodes. These intermediate steps provide additional capabilities that meet some requirements of "higher-up" intermediate goals. This continues until the final goal, represented by the root of the tree, has all its requirements satisfied.

Attack detection mechanisms work by examining the leaves (component parts), and from them inferring the intermediate nodes, and ultimately that the requirements of the root of the tree may be satisfied, thus realizing the attack. Of course, a new organization of component parts leads to a different attack tree, but an astute attacker can counter this by rearranging or repartitioning those components, leading to an "arms race" that makes defenses at best ineffective and at worst triggering alerts on so many innocuous actions that the rate of false positives renders them useless.

Rather than focus on detecting the leaves of the attack trees, we focus on the root. As that is the compromised state, we instrument the system to detect when that goal is reached. Even if the goal is reached using some different (and possibly unknown) sequence the detection method works because it looks for the *attack result* and not the attack itself.

The limitations of detecting damage at the leaves is perhaps best illustrated by an example. Imagine that we are attempting to ward off a physical attack—all majors blows are blocked by our defenses. However, smaller attacks and injuries are let through; eventually, even though no single event was sufficient to cause us serious injury, the combined impact of the small attacks will be sufficient to cause us deadly harm. Thus, it is crucial to view the combined impact on the system, not simply the individual result of any single action.

This *result detection* mechanism can be designed in one of two ways. The first is state auditing, in which the system state is examined for violations of the security policy. This method is appropriate when the components of the state affecting security are few, and can be readily checked, or when periodic scans are considered sufficient. Tripwire [11], configured to target specific files, is an example of such a mechanism.

A variant of state checking, which may be more effective in some cases, is to check state after a particular action. In this case, only those components of the state affected by the action need to be checked. For example, when an action to alter the Windows Registry is found, the system either allows the action to proceed and then checks the Registry to see if it is in a compromised state, or emulates the action and determines whether the action would put the Registry into such a state [24].

The difference between this second form of result detection and ordinary attack detection lies in what is being detected. Consider a polymorphic encrypted virus. The generic decryption technology used to detect these viruses typically loads the suspect executable into an emulator. The emulator periodically uses heuristics to analyze the memory for malware. It does not check that the malware has done anything; it looks simply for the malware [17]. Result detection would check for the result. Similarly, heuristic-based intrusion detection systems look for patterns of behavior, such as system call patterns [9] or function references [20].

To demonstrate this approach, we present three examples of attacks using malware that attack detection will fail to catch but results detection will catch.

4 Examples and Analysis

Given a signature -based attack detection mechanism (including behavioral signatures), one can construct an attack that will either not be detected by existing signatures or whose detection will trigger a large number of false positives. In other words, one may be able to identify the attack from a signature, but that same signature will not uniquely identify the attack; it will also match innocuous, and in some cases common, actions.

These attacks grew from two observations. First, signatures used by anti-malware systems must distinguish between malicious and benign programs. Second, both attack and non-attack programs and processes ultimately use the same "building blocks," namely computer instructions. Thus, signatures must distinguish attacks and non-attacks based on specific instructions ordered temporally. Unless they track *every* instruction in order of execution, anti-malware systems must make simplifying assumptions—and these assumptions can be negated.

Consider static signatures that, when found in a file, cause anti-malware systems to report that the file contains malware. Partition the file into several pieces such that no single piece contains a malware signature, but such that loading them together in memory recreates the malware. Attack detection fails because the signature will not be detected as it is never in the file [25].

Now look at behavioral signatures. Here, a series of actions provide a basis for an anti-malware system to assert that a process contains malware, because that series matches a signature of execution. The computational overhead of monitoring all processes in real time, and analyzing all possible combinations of commands that occur, is simply too high to be practical. So anti-malware systems focus on related processes—parents and descendants, or siblings. One can then take the program with the embedded malware, split it into several programs that run independently but communicate in various ways (such as IPC, which may well associate the processes enough to enable anti-malware systems to detect the co-ordination, or covert channels, which will not) so that the effect of all the processes executing is the same as the single original program. This also evades many anti-malware systems, because the signature will not be detected as it lies scattered over several processes [26].

These attacks involve executing files constructed to avoid signature detection, or running processes constructed to avoid behavioral signature detection and that result from executing downloaded files. In both cases a co-ordinator must assemble the downloaded files and execute them. But, as noted above, those files are simply collections of data and computer instructions. We suggested that they might be obtained directly from existing files or processes.

The second set of attacks does exactly this. The concept of "gadgets" has grown directly from an exploit technique known as "Return Oriented Programming" (ROP). As chipsets have become more sophisticated, various countermeasures have been put in place to make exploitation of vulnerabilities more difficult. The use of the "no execute" attribute, for example, attempts to prevent an attacker from executing data, and "stack cookies" try and detect buffer overruns that cross stack frames. Such countermeasures basically prevent typical attacks that overrun a return address on the stack (discussed, for example, by AlephOne [1]) and transfer control to the attacker-supplied buffer.

Given that buffer overruns remain a common programming error, attackers began exploring new ways to exploit them, and eventually settled on ROP. Here, an exploit does not contain the actual executable code needed to carry out an action, but instead leverages code already extant on the system. Sometimes ROP uses well-known API calls to carry out its work, at other times it carries out its mission using small fragments of code (called "gadgets").

Defending against this technique focuses on disallowing usable code fragments at build time [19]. Nevertheless, the idea of gadgets does raise some interesting questions, and highlights the importance of results analysis. Arguably, especially for a Trojan Horse, the vast majority of the code that executes the attack could actually be code that already exists on the system. In fact, one can create an exploit using ROP that is never executed, but only alters the execution path of code already resident on the system.

This leads us to an interesting discussion. Consider a piece of stand-alone malware. This attack code is sent in its entirety to the host machine, and when run formats the first fixed disk. Clearly, it is malware, and should be classified as such. Now consider data input which overruns a buffer adjacent to a function pointer, directing flow to an-already extant piece of code on the system that formats the first fixed disk. The exploit contained no code; it just contains some padding data (which is typically irrelevant) and the address of a function. Is this something which we should detect (is it malware?) and more importantly does it make sense to even try and detect it? We argue that the system must be viewed holistically. The data is not malware—the *combination* of the errant data and the environment is.

We note that this concept is not new *per se*. Cohen's original work on computer viruses [2, 3] always considered a computer virus to be a combination of the environment (machine) with a set of instructions. Thus, there is nothing *technically* incorrect about considering our data-only attack to be malware, but it is not necessarily *helpful* to do so.

4.1 Bad Good Code

As touched on above, part of the challenge comes from our insistence on determining whether particular things—in isolation—are good or bad. Such classifications tend to imply intent, and determining the intent of the attacker is not really possible. The best we can do, in the absence of being able to ask the author himself, is guess.

In the case of gadgets, this classification scheme is stretched to the point of breaking, and the unnatural contortions it requires of us become evident.

Consider, for example, a piece of attack code (for illustration, we use a JPG file that triggers a buffer overrun in a viewer). This attack code makes use of ROP techniques and gadgets its author has previously identified in the viewer. Here, then, the viewer is vulnerable, but the attack itself is found in the JPG.

However, let us assume that the attacker has some (limited) control over the code for the viewer. Perhaps it is an open source project, or the attacker works for the viewer's manufacturer. Regardless, the control the attacker has over the source code in not complete, as the code for the viewer must in either case pass peer review.

Based on his analysis, the attacker determines the viewer is vulnerable to a buffer overrun, but lacks the presence of a certain gadget to make this vulnerability exploitable. He modifies the source code, changing potentially just one or two bits of the compiled output, creating the necessary preconditions for the exploit. Is the viewer now "bad"? Should it be detected, in addition to the JPG which contains the attack? Can we even now honestly say the attack is contained in the JPG, when in actuality, the attack code, at least in part, has been included in the viewer?

Taking this a step further, the attacker could actually embed all the functionality he needs in the viewer, except for the actual redirection that sets the chain in motion. Thus, our "exploit" code is actually just a buffer overrun and one associated address. All other functionality now exists in the viewer. Here, most researchers would instantly argue the viewer was in fact a Trojan Horse, and the JPG simply the trigger.

These two points represent two ends of a continuum. As the attacker deliberately builds in gadgets, the attack uses more code already extant on the system. Looking at the binary, or possibly even the source, it is impossible to tell that the code has been written to facilitate the exploitation of a vulnerability elsewhere in the codebase.

This thought experiment is interesting for two reasons. First, it highlights the intimate linkage between the JPG and the application or service it exploits. Second, it illustrates the futility of attempting to search incoming information (be it data or code) for exploits.

From an attackers perspective, the idea of deliberately embedding gadgets is attractive. They will not be found by traditionally vulnerability scanning tools that focus on source code analysis (indeed, they may not "exist" in source form at all, but in the compiled code). With care, the code that enables them will pass a code review by another coder. Finally, this code can be spatially and logically separated within the binary, like tumblers in a combination lock. They will be hard to find by fuzzing or other automated testing techniques, and represent a beautiful latent back door that can be opened at will.

Finally, some of this functionality may not be in the code at all. Careful (and slight) misdocumentation of API side effects allows the attacker to have the code appear to be perfect based on documented functionality. Thus, detection by inspection (as opposed to behavior) requires considerable context within the system.

4.2 Result Detection

These attacks techniques focus on the attack detection method, and illustrate the difficulty of anticipating all possible attack vectors. An attacker with knowledge of the defenses can examine ways to evade those mechanisms. In general, detection mechanisms work in three ways:

1. Catch attacks at the perimeter, when they are injected into the system. As noted above, these can be evaded. In particular, the evasion mechanisms need to distinguish attacks from non-attacks with an acceptable rate of false positives—a rate that does not overwhelm the analysts.
2. Catch the attacks during execution. This problem is similar, except that instead of instructions, the detection mechanisms look for temporal sequences of actions in related processes. By distributing the actions over multiple unrelated processes, the attack detection mechanism can be evaded.
3. Catch the *results* of the attack. Here, the focus is not on the attack but on how it causes a deviation from correct behavior.

Table 1 summarizes the primary differences in these approaches. Each technique has advantages and disadvantages. None is necessarily a replacement for the others, but combinations should provide robust protection.

When malware executes, it may not violate any part of the security policy. However, when the attacker carries out an action that *does* violate the policy, a result-oriented approach will detect it. Centering detection on goals we care about should provide protection of the things that matter.

The need is even more striking with gadgets. The "attack code" is already resident on the system, in innocuous forms; indeed, eliminating it would break harmless, and in some cases essential, programs. So detection would have to focus on executing that code. But given the complexity of distinguishing between "good executions" and "bad executions," this seems infeasible. Extending the analysis to the limit leads to a situation in which any execution and any program is suspect, producing an overwhelming number of false positives.

Consider another example, that of network-based data exfiltration. Attack detection methods would look for processes that might open files containing sensitive data, or scan files looking for sequences of code that could open such files. The result detection approach may take one of two forms.

First is simply to monitor the boundaries that sensitive data should not cross (noting, of course, that these boundaries may be different for each sensitive datum). When the defensive mechanism detects the crossing, it blocks the flow of the sensitive information, and reports the problem. This approach, which is essentially to define the set of messages that may cross the boundary, is imperfect. For example, encrypted data requires special handling. Variants of this method (such as isolating networks on which the data lies) improve technological defenses, but leave procedural defenses grounded on imperfect human beings.

A second form of the result detection approach is to monitor all accesses to the sensitive files through the use of watchdogs or some similar technology. That the file be opened and the data read is a necessary step in the attack. Filters and other mechanisms can reduce the number of false positives, but as with all security mechanisms, imperfections remain. Yet this method will produce no false negatives, because if the data is not accessed, it cannot be exfiltrated.

Developing detection mechanisms that analyze attack results rather than attacks requires a discipline of analysis. First, the specific compromised states must be identified; then, the specific commands, system calls, or other actions must be enumerated. A goal-based analysis approach similar to that for developing systems designed for forensic analysis [21] is appropriate here.

5 Conclusion and Future Work

This paper argues for a focus on the outcome of the attack rather than the attack itself. This has three advantages. First, it identifies entry into a compromised state, which is after all the definition of a violation of security (and of a successful attack). Second, it reduces the complexity of the detection mechanisms because those mechanisms need not look for *potential* attacks, which requires a large number of signatures or multiple heuristic mechanisms. Detectors can either check for known good states, or states that are known to be bad. Third, it forces the developer of the detection methods to take into account the environment in which the attack is to be realized—something essential for dealing with the limiting cases of the gadgets, identified above.

Instances of this approach have always existed. For example, Tripwire [11] scanned file systems and reported anomalies that may have been the result of attacks. However, our approach has a different perspective: we look for those indications that are *specifically* tied to compromising the system rather than scanning for anomalies. This reflects the difference between anomaly-based and specification-based intrusion detection [14]. The former, and past work involving looking for results, looked for unusual events and presupposed they indicated attack. The latter, and our approach, look

Technique	Philosophical Approaches	Attacker Countermeasures	Advantages	Disadvantages
Anomaly Detection	"What's different is bad"	Blend traffic/behavior in with normal behavior. Deliberately create false positives.	Capable of detecting new attack modalities, provided they look "different" to normal behavior.	The underlying assumption (different == bad) can give rise to unacceptable false positive rates.
Signature Detection	"Find the things we know are bad"	Server-side meta- and polymorphism.	Very reliable for known attacks.	Very poor against new attacks.
Behavioral Detection	"Determine where good or bad based on actions"	Mimicry of benign operations, spread attack over multiple small objects/actions.	Quite good at detecting new attacks, acceptable false positive rates.	Too focused on behavior of individual objects instead of the system holistically.
Results Detection	"Make sure undesirable events do not happen to protected things"	Come up with goals which are consonant with the concept of system behavior.	Detects the part of the exploit we care about – the violation of the data/system.	Difficult to quantify the potential goals of the attacker for many cases.

Table 1. Table of results-oriented detection techniques

for results known to be the result of attack.

One difficulty is that many compromised states are unknown. This means that the relationship between the implementation of the system and the security policy is not fully understood. But this is essentially the same problem as not having signatures for all attacks available which is the current state of the art, as the term "zero-day vulnerability" implies. So this problem is no worse than one existing in current attack detection mechanisms, and in fact is probably more tractable.

An interesting question is how the false positives and negatives for the results-oriented approach compare to those of the attack-oriented approach. Another interesting one involves examining vectors for distributing attacks using gadgets. In a way, it is like building a combination lock into an application. Each little part looks innocuous, but given the right sequence, the lock springs open. The distributed nature has several benefits for the attacker. First, the spatial and potentially temporal distribution makes these parts very hard to find. They will resist analysis using fuzzing and static analysis tools. Second, some of the distributions could even include errors in documentation, where an API is slightly, but deliberately, misdocumented. That would allow the parts of the attack code to pass code reviews. Finally, a group of attackers might be able to use different libraries, shared objects, and compilers on a platform like Linux to develop an attack that is very difficult to detect unless the mechanisms look for the results. This is disturbing, to say the least, and calls for the attack-result detection paradigm we have developed here.

Acknowledgements: Matt Bishop thanks the National Science Foundation for the support of grants CCF-0905503 and CNS-1049738 to the University of California at Davis. All opinions are those of the authors and not necessarily those of the National Science Foundation.

References

[1] AlephOne. Smashing the stack for fun and profit. *Phrack*, 7(49), 1996.

[2] F. Cohen. Computer viruses: Theory and experiments. *Computers & Security*, 6(1):22–35, Feb. 1987.

[3] F. Cohen. *A Short Course on Computer Viruses*. John Wiley & Sons, Inc., New York, NY, USA, 1994.

[4] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: Securing software by blocking bad input. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*, pages 117–130, Dec. 2007.

[5] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containmanet of internet worm epidemics. *ACM Transactions on Computer Systems*, 26(4):9:1–9:68, Dec. 2008.

[6] J. R. Crandall, Z. Su, S. F. Wu, and F. T. Chong. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic exploits. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 235–248, 2005.

[7] D. R. Ellis, J. G. Aiken, K. S. Attwood, and S. D. Tenaglia. A behavioral approach to worm detection. In *Proceedings of the 2004 ACM Workshop on Rapid Malcode*, pages 43–53, 2004.

[8] B. Gu, X. Bai, Z. Yang, A. C. Champion, and D. Xuan. Malicious shellcode detection with virtual memory snapshots. In *Proceedings of the 2010 IEEE INFOCOM*, pages 1–9, Mar. 2010.

[9] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[10] D. Jain, K. V. Lakshmi, and P. Shankar. Deep packet inspection using message passing networks (extended abstract). In R. Lippmann, E. Kirda, and A. Trachtenberg, editors, *Proceedings of the 2008 Symposium on Recent Advances in Intrusion Detection*, volume 5230, pages 419–420, Sep. 2008.

[11] G. H. Kim and E. H. Spafford. The design and implementation of tripwire: A file system integrity checker. In *Proceedings of the Second ACM Conference on Computer and Communications Security*, pages 18–29, 1994.

[12] S. Kim and J.-Y. Lee. A system architecture for high-speed deep packet inspection in signature-based network intrusion prevention. *Journal of Systems Architecture*, 53(5-6):310–320, May 2007.

[13] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. A. Kemmerer. Behavior-based spyware detection. In *Proceedings of the 15th USENIX Security Symposium*, Aug. 2006.

[14] C. Ko, M. Ruschitzka, and K. Levitt. Execution monitoring of security-critical programs in distributed systems: A specification-based approach. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 175–187, May 1997.

[15] J. Mason, S. Small, F. Monrose, and G. MacManus. English shellcode. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, pages 524–533, 2009.

[16] M. A. McQueen, T. A. McQueen, W. F. Boyer, and M. R. Chaffin. Empirical estimates and observations of 0day vulnerabilities. In *Proceedings of the 42nd Haeaii International Conference on System Sciences*, pages 1–12, Jan. 2009.

[17] C. Nachenberg. Computer virus-antivirus coevolution. *Communications of the ACM*, 40(1):46–51, Jan. 1997.

[18] J. Newsome, D. B. anbd Dawn Song, J. Chamcham, and X. Kovah. Vulnerability-specific execution filtering for exploit prevention on commodity software. In *Proceedings of the 2006 Symposium on Network and Distributed Systems*, Feb. 2006.

[19] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-free: Defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 49–58, Dec. 2010.

[20] S. Peisert, M. Bishop, S. Karin, and K. Marzullo. Analysis of computer intrusions using sequences of function calls. *IEEE Transactions on Dependable and Secure Computing*, 4(2):137–150, Apr. 2007.

[21] S. Peisert, M. Bishop, and K. Marzullo. Computer forensics *in Forensis*. *SIGOPS Operating Systems Review*, 42(3):112–122, Apr. 2008.

[22] P. Piyachon and Y. Luo. Efficient memory utilization on network processors for deep packet inspection. In *Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, pages 71–80, 2006.

[23] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Comprehensive shellcode detection using runtime heuristics. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 287–296, Dec. 2010.

[24] D. Povey. Optimistic security: A new access control paradigm. In *Proceedings of the 1999 New Security Paradigms Workshop*, pages 40–45, Oct. 2000.

[25] M. Ramilli and M. Bishop. Multi-stage delivery of malware. In *Proceedings of the 5th International Conference on Malicious and Unwanted Software*, pages 91–97, Oct. 2010.

[26] M. Ramilli, M. Bishop, and S. Sun. Multiprocess malware. In *Proceedings of the 6th International Conference on Malicious and Unwanted Software*, 2011.

[27] Y. Song, M. E. Locasto, A. Stavrou, A. D. Keromytis, and S. J. Stolfo. On the infeasibility of modeling polymorphic shellcode. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 541–551, 2007.

[28] P. Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley professional, Boston, MA, USA, Feb. 2005.

[29] S. J. Templeton and K. Levitt. A requires/provides model for computer attacks. In *Proceedings of the 2000 New Security Paradigms Workshop*, pages 31–38, 2000.

[30] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 255–264, Nov. 2002.

[31] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. *ACM SIGCOMM Computer Communications Review*, 34(4):193–204, Aug. 2004.