Monitoring for Security Intrusion using Performance Signatures

Alberto Avritzer, Rajanikanth Tanikella, Kiran James
Siemens Corporate Research
755 College Road East
Princeton, NJ, USA 08540
alberto.avritzer@siemens.com, rajanikanth.tanikella@siemens.com, kiran-james.ext@siemens.com

Robert G. Cole
JHU/Applied Physics Laboratory
12000 Johns Hopkins Road
Laurel, MD, USA 20723
robert.cole@jhuapl.edu

Elaine J. Weyuker
AT&T Labs - Research
180 Park Avenue
Florham Park, NJ, USA 07932
weyuker@research.att.com

ABSTRACT

A new approach for detecting security attacks on software systems by monitoring the software system performance signatures is introduced. We present a proposed architecture for security intrusion detection using off-the-shelf security monitoring tools and performance signatures. Our approach relies on the assumption that the performance signature of the well-behaved system can be measured and that the performance signature of several types of attacks can be identified. This assumption has been validated for operations support systems that are used to monitor large infrastructures and receive aggregated traffic that is periodic in nature. Examples of such infrastructures include telecommunications systems, transportation systems and power generation systems. In addition, significant deviation from well-behaved system performance signatures can be used to trigger alerts about new types of security attacks. We used a custom performance benchmark and five types of security attacks to derive performance signatures for the normal mode of operation and the security attack mode of operation. We observed that one of the types of the security attacks went undetected by the off-the-shelf security monitoring tools but was detected by our approach of monitoring performance signatures. We conclude that an architecture for security intrusion detection can be effectively complemented by monitoring of performance signatures.

Categories and Subject Descriptors

C.2.3 [**Computer Communications Networks**]: Network Operations—*network monitoring*; D.2.0 [**Software Engineering**]: General—*protection mechanisms*

WOSP/SIPEW'10, January 28–30, 2010, San Jose, California, USA.

General Terms

Security, Performance, Measurement, Monitoring

1. INTRODUCTION

Software systems that are used to support high-reliability mission-critical systems are usually monitored for both software defects and performance. In addition, several layers of security defenses are commonly deployed to protect these systems from intrusion by non-authorized users.

We use a non-intrusive logging and analysis approach that analyzes system data generated by Microsoft Windows Management Instrumentation API (WMI). We present empirical results that show that security intrusion of software can leave the software in a state in which it is still operational, but the system's available capacity has been reduced. Such a condition is sometimes referred to as a *soft failure*. In earlier studies, [1, 2, 3, 4, 5] we have presented examples of soft failures that represent cases of system-wide performance problems.

In this paper we present an architecture for monitoring mission-critical systems for security intrusion that takes advantage of system-wide performance signatures. In addition, we suggest an approach to distinguish between performance signatures that can be identified as being associated with security intrusions and those that are associated with faults that will lead to soft failures.

Security has been a documented issue for several millenia, with recorded cases of ancient Egyptian and Hebrew scribes devising codes to secure their messages [11]. In modern times there has been a great deal of foundational research including pioneering work at IBM [10] that led to the invention of the US Data Encryption Standard (DES), the Diffie-Hellman algorithm [7], and the RSA algorithm that led to a practical public key system [14].

Currently, securing a computer is a multi-layered process, in which some security is provided at each layer, the edge routers, the local area networks, and at individual computers by restricting user access according to user roles. Each security layer builds towards the goal of securing a set of valuable assets according to a threat model. For example, for an Internet Service Provider (ISP), RFC 2827 (Network Ingress Filtering) ensures that the packets originating from a network have source IP addresses from the specified address

range of the network. In addition, at the Network level, an access control mechanism is implemented through which some machines are denied access to the network (blacklisting), and some IP addresses and port numbers are monitored. If needed, deep packet content inspection takes place to detect the presence of attacks. This takes place at layers 2 through 7 of the OSI network protocol.

Intrusion detection systems (IDS) and intrusion prevention systems (IPS) are deployed to analyze packet contents and to counter security threats that may occur at the network layer. At the application layer, firewalls, anti-virus, and anti-spam software provide a combination of network and application-level security.

In spite of all of these precautions, the role of user education to ensure the security of mission-critical infrastructures cannot be underestimated. For example, many security attacks succeed because a significant part of the user community continues to select weak passwords [15].

The outline of this paper is as follows: In Section 2 we provide a brief literature review of approaches to system monitoring based on performance signatures. In Section 3 we present the tool architecture used for security intrusion detection. In Section 4 we present the approach used in this paper for security testing, including test procedures. In Section 5 we present the empirical results derived from our experiments. In Section 6 we compare the effectiveness of the off-the-shelf performance monitoring tools described in this paper to our detection approach based on performance signatures. In Section 7 we present the algorithm for the detection of performance degradation using performance signatures. In Section 8 we discuss our conclusions and directions for future work.

2. RELATED WORK

In earlier work, our research group studied a number of related problems that have provided a foundation for our current research. Deterministic State Testing (DST) [1] is a load test generation and execution approach, designed for telecommunication systems. When executing load tests generated using the DST approach, we observed soft failures that caused the system-under-test to operate with reduced capacity.

[2] described an approach for non-intrusive monitoring of software components that took advantage of the performance signatures of the components, while [3] introduced the concept of software capacity restoration by the use of timely reboots of the software system. Software capacity restoration is also known as *software rejuvenation* [9]. Several faults that led to soft failures were detected by monitoring performance signatures of a large mission-critical telecommunications system [4].

In [5], we introduced an approach to ensure performance of mission-critical systems by monitoring a customer-affecting metric leading to software rejuvenation when degradation was observed.

In the current paper we extend the bucket algorithm introduced in [5] to account for a multi-dimensional performance signature. Our goal is to be able to differentiate between performance degradation resulting from faults that lead to performance problems and those that result from security intrusions. The approach described in this paper is related to the algorithm presented in [6] in which a bucket algorithm

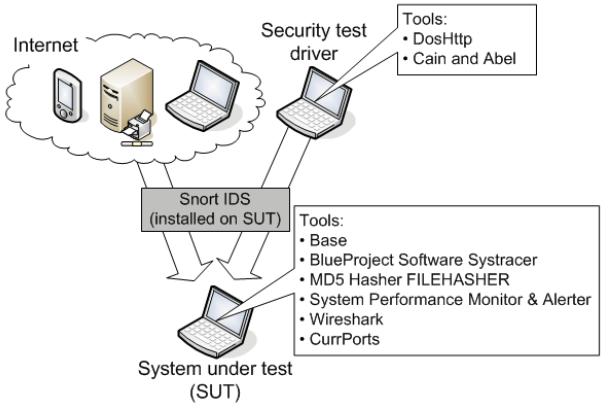

Figure 1: Tools used for security testing

combined with software was employed to thwart worm propagation in a MANET.

In [13], a brief outline of the use of performance signatures for intrusion detection was presented, based on the monitoring of time, memory, and communications requirements of certain program functions to differentiate between well-behaved and suspicious patterns of program usage.

Performance signatures were introduced in [12] as way to provide qualitative design guidelines to programmers before source code development is started. The work presented in [2], [13] and [12] is related to performance specification and performance monitoring for software components. In contrast, the work presented in the current paper is related to using performance signatures for security intrusion detection of mission-critical systems.

Other interesting research related to security enforcement using expected system usage includes [8], in which system call policies are enforced for applications by constraining the application's access to the system.

3. ARCHITECTURE FOR INTRUSION DETECTION

Several tools were deployed to drive the security tests, log test results, detect security attacks, and analyze the results. Figure 1 shows the list and locations of the tools used for security testing.

Cain & Abel is a password recovery tool for Microsoft operating systems. It allows easy recovery of various kinds of passwords by sniffing the network, cracking encrypted passwords using a dictionary, brute-force and cryptanalysis attacks, recording VoIP conversations, decoding scrambled passwords, recovering wireless network keys, revealing password boxes, uncovering cached passwords and analyzing routing protocols. The program does not exploit any software vulnerabilities or bugs that could not be fixed with a little effort. We used this tool mainly to do ARP poison routing in order to conduct man-in-the-middle attacks against the system-under-test.

DoSHTTP is a powerful HTTP flood Denial of Service (DoS) testing tool for Microsoft Windows. *DoSHTTP* includes URL verification, HTTP redirection, port designation, performance monitoring and enhanced reporting. It

uses multiple asynchronous sockets to perform an effective HTTP flood and can be used simultaneously on multiple clients to emulate a Distributed Denial of Service (DDoS) attack. We used this tool to attack a particular web service hosted on the system-under-test.

The *System Performance Monitor & Alerter* tool was deployed at the system-under-test to log security test results. It is a Java-based tool developed for the framework by our research group, which runs as a background application. It was created as a wrapper around the open source software *Hyperic Sigar* [19] to gather system performance parameters. It continuously monitors the CPU, memory usage, and interface parameters and compares the values against the values taken from the baseline profile. The values are sampled periodically. We used a frequency of 5 seconds. This helps us prevent spikes and avoid false positives.

The *Wireshark* [18] and *Snort* [16] tools were used to monitor network traffic. These were also deployed at the system-under-test. Wireshark inspects the payloads of flagged packets. Packets are usually flagged when errors are detected, such as non-compliance with a protocol or a checksum failure. Wireshark can also be used to certify that the packet has taken the correct route, and is therefore useful in the case of a man-in-the-middle attack.

Snort was the primary tool used in the analysis of the security attacks, and was deployed as an Intrusion Detection System. MySQL was installed along with Snort so that when network traffic corresponding to a Snort rule was detected, the signature match can be logged into its MySQL database for post mortem analysis.

The *Base* tool [21] provides the graphical user interface to Snort IDS, notifying the user about attacks against the host machine which match the Snort rules list.

The *Currports* tool [22] is used to scan open ports and monitor the number of established ports and connections.

The *MD5 Hasher Filehasher*, and the *Systracer* [23] tools are used to track changes to the Registry and file system of the system-under-test. *MD5 Hasher Filehasher* is a tool developed for the framework by our research group, which generates MD5 hashes of files of given types. Hashes of the files are written into a flat file for later analysis. Hashes of files taken after the exposure of the system-under-test to threats can be compared to those taken before to detect changes to those files. Similarly, the Systracer tool takes snapshots of the registry and file systems before and after threat exposure to help detect changes.

4. APPROACH FOR SECURITY TESTING AND INTRUSION DETECTION

Our objective in this paper is to assess the effectiveness of using performance signatures for security intrusion detection as compared to using off-the-shelf intrusion detection tools.

We modeled the performance characteristics of the system-under-study to generate a background load that was a reasonable representation of system usage. The background load was composed of CPU, memory, I/O, and network usage.

The steps we used to generate the background load were:

1. Create a system profile to mimic the performance characteristics of the system-under-study,
2. Customize an available shell script-based benchmark to mimic the identified user profile.

This background load was invoked along with the test set-up required to measure the security test cases. We identified the following security test cases to be used to attack the system-under-study:

- Buffer Overflow,
- Stack Overflow,
- SQL Injection,
- Denial of Service (DoS), and
- Man-in-the-Middle (MITM).

The process to generate performance signatures for security attacks consists of the following steps:

1. The system-under-test logs all data from the deployed intrusion detection tools and from Microsoft Windows Management Instrumentation API (WMI),
2. The logged data is analyzed offline to identify the performance signatures of each security attack.

Each experiment was conducted in two stages: a regular usage test stage to collect the baseline performance profile, and a security test stage to collect the attack profile. Each stage is composed of three steps:

1. System running with baseline load for five minutes ("pre-attack baseline time frame"),
2. System running with baseline load and additional load for 1 minute ("test activity time frame," wherein a regular usage test or security test is run depending on the stage), and
3. System running with baseline load for four minutes ("post-attack baseline time frame").

We now describe in detail the procedures performed during the test activity time frame for each of the five security tests.

Since the security detection framework involves both network and system level monitoring, the vulnerable applications were selected in such a way that they were readily accessible through the network.

In computer security and programming, a *buffer overflow*, or *overrun*, is a vulnerability in which data being written by a process exceeds the memory space allocated for it. The extra data overwrite adjacent memory, which may contain other program data, or even program instructions. This may result in erratic program behavior, including memory access errors, incorrect results, program termination (a crash), or a breach of system security (e.g., execution of arbitrary commands with the privileges of the exploited program.) This sort of breach is known as a *buffer overflow attack*.

Some programming languages including C and C++ do not have native mechanisms to check memory bounds before writing, which makes applications developed in these languages potentially vulnerable. One of the first steps in a buffer overflow attack is the attempt to access a memory location outside the memory space allocated for the data structure. This may cause a system crash and a subsequent entry in the application log.

The test procedure for buffer overflow attack security test uses a web client as the attacker and the web server as the system under test. The procedure is summarized here:

1. The attacker requests the test page (which provides an interface to a web application with a buffer overflow vulnerability) from the server,
2. The SUT returns the test page,
3. The attacker submits the query via the test page, inserting a large amount of data to cause a buffer overflow,
4. The SUT crashes, displaying an error message on the SUT console.

For *man-in-the-middle* attacks, the connection between the system-under-test and the router was hijacked by the attacking machine using a technique called ARP (Address Resolution Protocol) poisoning. Using the *Cain & Abel* tool [17] tool, the ARP table of the system-under-test was manipulated in such a way that, for packets sent to the outside network, the MAC address of the router was replaced by that of the attacking machine. This gives the attacker access to the SUT's requests, as well as the corresponding replies. Thus, the test procedure for the man-in-the-middle attack security test uses the web client as the system under test, and a man-in-the-middle machine as the attacker. The procedure is described here:

1. The ARP table of the SUT is cleared using the *Cain & Abel* tool,
2. The SUT requests a page from the Internet. In so doing it issues an ARP request to find the address of the router to the Internet,
3. The attacker, using the *Cain & Abel* tool, replies to the SUT's ARP request and provides its own MAC address. This effectively hijacks the connection,
4. The attacker forwards the SUT's request to the intended destination,
5. The intended destination responds with the requested web page,
6. The attacker receives the response and forwards it to the SUT,
7. The SUT receives and displays the requested page.

For the *stack overflow* attack, a vulnerable application was hosted on the system-under-test machine which would have resulted in a stack overflow if the application is ever accessed in a particular manner. The test procedure uses the web client as the attacker, and the web server machine as the SUT. The procedure is described here:

1. The attacker requests the test page (which provides an interface to a web application with a stack overflow vulnerability) from the server,
2. The SUT returns the test page,
3. The attacker submits the query via the test page, inserting data to cause a stack overflow,
4. The SUT processes the query:
   - If the stack space is exhausted then the SUT host responds with an error message indicating "out of stack space."),
   - else return normal application response.

The *denial of service* attack test procedure uses the web client as the attacker and the web server as the SUT. The test procedure was executed with the help of the *DoSHTTP* tool [20] running on the attacker to flood the SUT with HTTP connections. The procedure is shown here:

1. The attacker probes the URL of a resource on the SUT by making a request,
2. The SUT responds with the requested resource,
3. The attacker uses the *DoSHTTP* tool to flood the URL with connection requests,
4. The SUT receives the request and attempts to service it if connections are available:
   - If connections are available then the SUT returns the requested resource,
   - else the SUT ignores the request,
5. Failure of the SUT can be detected by the attacker by requesting the same resource. If no response is forthcoming then the denial-of-service attack is succeeding.

SQL Injection is a code injection technique that targets the database layer of an application. The vulnerability is present when user input is either incorrectly filtered for string literal escape characters embedded in SQL statements or user input is not strongly typed and thereby unexpectedly executed. Another form of attack is made in the hope of revealing some exploitable details about the database system behind the web application. Often, an attacker may conduct such an attack to obtain details about the database (e.g., brand, version, table schema) before conducting further, more focused attacks.

The test procedure for the SQL injection attack security test uses the web client as the attacker and the web server as the SUT. The SQL Injection test case was run using three different inputs. First it was run with a normal legal input to the web application. This input is in line with the expectations of the web application and produces an expected result. Second, the test case was run with an input manipulated to result in the execution of a desired attack query. This input contains an SQL query paired with a condition that evaluates to true (e.g. "' OR SELECT * FROM mytable OR 1=1".) Note that the leading "' OR" is intentionally included to preserve correct syntax and trick the web application into executing the attack query. Third, it was run with an input crafted to cause the web application to return with an error statement to the client.

The procedure is described here:

1. The attacker requests the test page (which provides an interface to a web application with an SQL injection vulnerability) from the SUT,
2. The SUT responds with the test page,
3. The attacker submits either a normal, SQL-injection, or error-causing query,
4. The SUT returns its response to the attacker:
   - For the case of a normal input, the SUT returns a normal response,

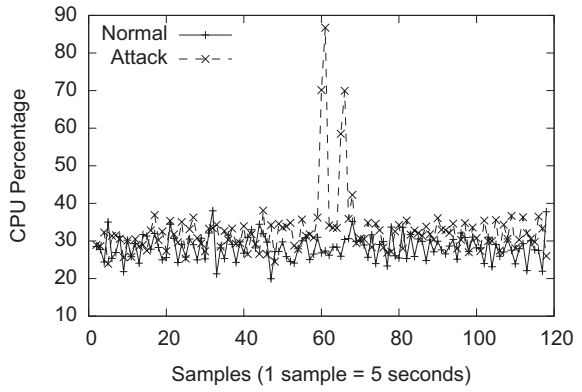

Figure 2: CPU usage during normal load and DoS attack

- For the injection case, the SUT returns the result of the injected command. In our case, all the data from a specified table in the database,
- For the error case, the SUT returns an error message issued by the database.

5. EMPIRICAL RESULTS

We executed the security test cases described in Section 4 using the tool architecture described in Section 3. In this section we show the plots obtained as a result of each security test case execution.

5.1 Denial of Service Attacks

The performance signature developed for the DoS security test is presented in Figures 2, 3, 4, 5, and 6. These figures plot the CPU usage, memory usage (as a percentage,) number of TCP resets, number of active threads, and number of TCP connections established for both the regular usage test stage, and the security test stage. We observe that the memory percentage usage plot, Figure 3, indicates degradation due to security attack after the test activity time frame. This degradation continues into the post-attack baseline time frame. The CPU usage plot, Figure 2, shows significant degradation in CPU usage only during the test activity time frame. In addition, the number of active threads shown in Figure 5, shows a spike at the time of execution of the security test which did not recover fully from the DoS attack. In particular, the number of active threads remaining in the post-attack baseline time frame is about 555, which is larger than the original number of baseline threads (about 540-550.)

5.2 SQL Injection

Along with the system performance monitor, we wrote Snort rules specific to the applications we were running on the system-under-test. In particular, Snort rules were created to detect specific patterns in payloads, such as SQL injection patterns. Table 1 shows a sample of the log file generated by Snort, which shows violations to the set of permissible inputs to the MySql database.

Figure 7 shows transmitted bytes/second for a normal SQL query, the malicious SQL injection, labeled "craft" in the plot, and the purposely error-causing query, labeled

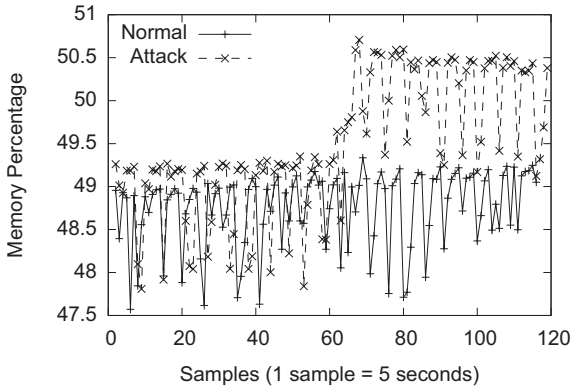

Figure 3: Percentage memory use during normal load and DoS attack

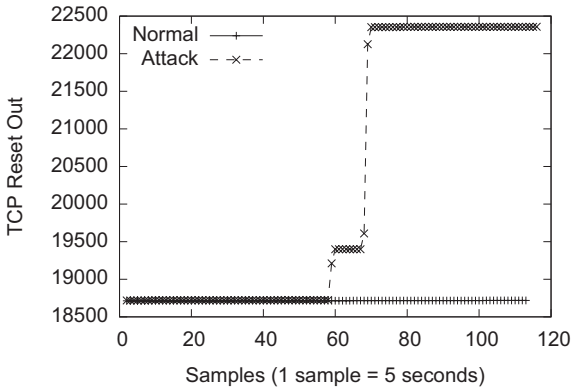

Figure 4: Number of TCP resets during normal load and DoS attack

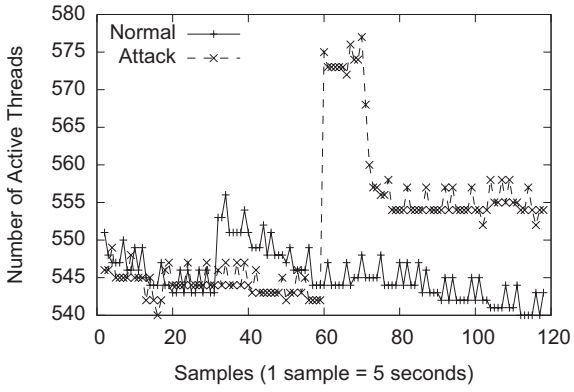

Figure 5: Number of active threads during normal load and DoS attack

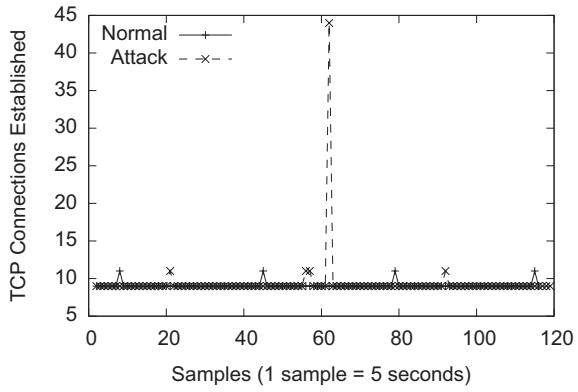

Figure 6: Number of TCP connections established during normal load and DoS attack

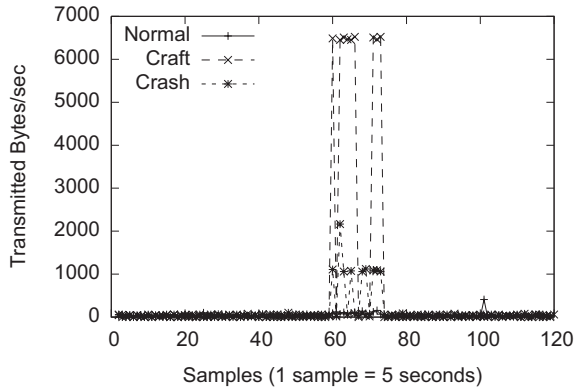

Figure 7: Transmitted bytes per second for normal, SQL-injected, and SQL crash-invoking queries

"crash" in the plot. We observe that in all three cases the performance signature of transmitted bytes/second was effective in detecting abnormal behavior.

5.3 Man-in-the-Middle

The performance signature developed for MITM security testing is presented in Figures 8, 9, 10, 11, and 12.

Figures 8 and 9 show results of the security test case that indicate higher CPU and memory usage than is normal. Figures 10, and 11 show similar trends for working set size, and number of active threads, although the former does not entirely return to pre-attack levels as the others do. We attribute this to the effort needed to compute and authenticate the fake certificate issued by the *Cain & Abel* tool. In the normal case, the certificate is actually issued by the site visited, and is likely already in the cache of the system-under-test.

A warning pop-up was issued by Internet Explorer to the system-under-test indicating the possibility of a potential threat since it could not authenticate the SSL certificate. Figure 12 shows the number of TCP Resets in both stages. The attack shows a more significant rise in the value since we observed more connection drops happening in the case of the *MITM* attack.

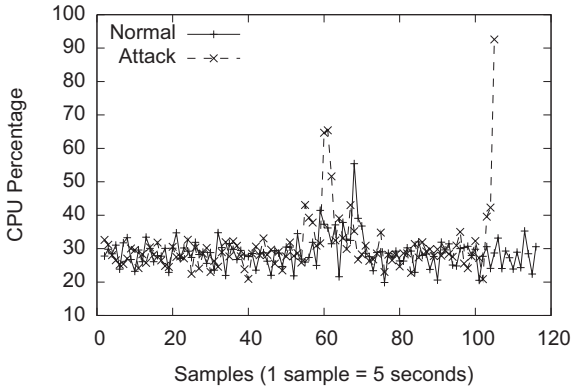

Figure 8: CPU usage during normal load and man-in-the-middle attack

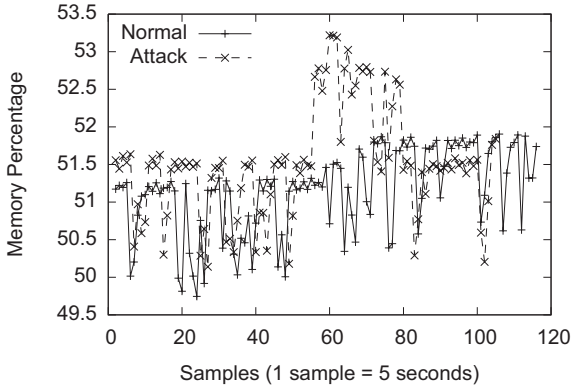

Figure 9: Percentage memory usage during normal load and man-in-the-middle attack

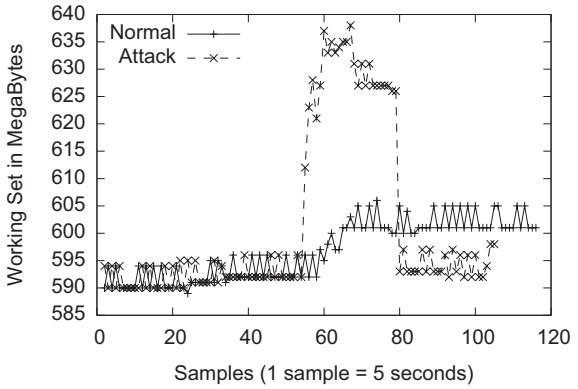

Figure 10: Process Working Set under normal load and under man-in-the-middle attack

1512-(1-1514)	[local]	[snort]	Deletion attempt	2009-04-23 11:23:20	129.73.15.15:22586	129.73.15.56:8080
1513-(1-1515)	[local]	[snort]	Deletion attempt	2009-04-23 11:49:27	129.73.15.15:22943	129.73.15.56:8080
1514-(1-1516)	[local]	[snort]	Drop table attempt	2009-04-23 12:02:09	129.73.15.15:23124	129.73.15.56:8080
1515-(1-1517)	[local]	[snort]	Command prompt attempt	2009-04-23 12:03:46	129.73.15.15:23124	129.73.15.56:8080
1516-(1-1518)	[local]	[snort]	Command prompt attempt	2009-04-23 12:07:33	129.73.15.15:23124	129.73.15.56:8080
1517-(1-1519)	[local]	[snort]	Illegal display attempt	2009-04-23 14:36:58	129.73.15.15:24815	129.73.15.56:8080
1518-(1-1520)	[local]	[snort]	Illegal display attempt	2009-04-23 14:38:06	129.73.15.15:24815	129.73.15.56:8080

Table 1: Application-specific Snort rules based on suspicious payloads (SQL Injection)

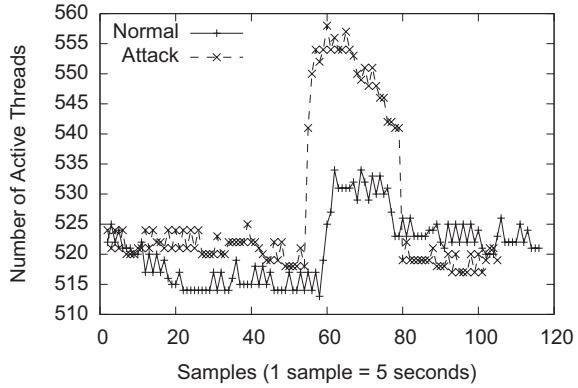

Figure 11: Number of active threads under normal load and man-in-the-middle attack

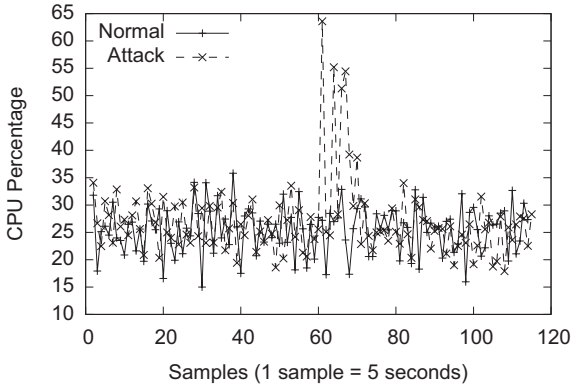

Figure 13: CPU usage during normal and buffer overflow attack

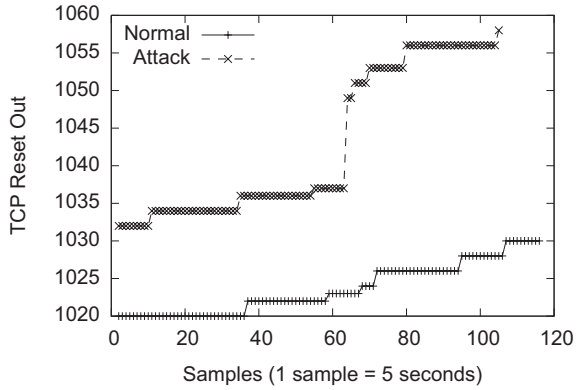

Figure 12: Number of TCP resets under normal load and man-in-the-middle attack

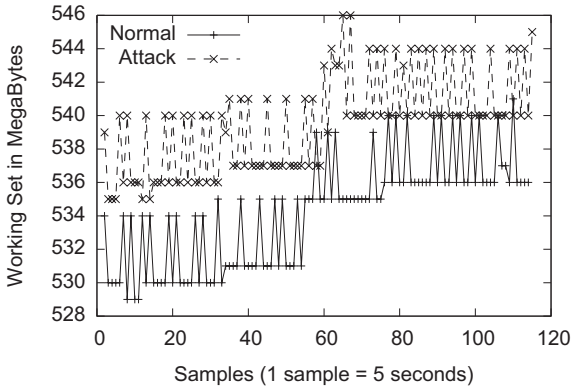

Figure 14: Working set during normal and buffer overflow attack

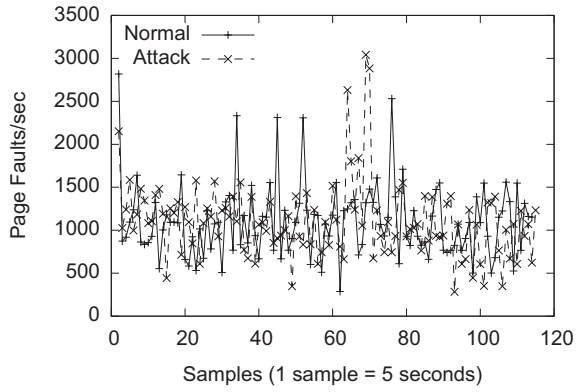

Figure 15: Page faults per second during normal and buffer overflow attack

5.4 Buffer Overflow

The performance signature developed for the buffer overflow security test is presented in Figures 13, 14, and 15.

Figures 13, 14, and 15 show the CPU percentage usage, working set size, and page faults per second for both stages of the Buffer Overflow security test. All of these graphs show significantly higher activity in the attack graph when compared to the normal traffic graph. The illegal access of memory space can explain these elevated parameters. During this test the system application log entry generated an illegal memory access alert. The System Log Monitor shown in Figure 21 is the presentation layer of the proposed attack detection architecture, and is used by the system administrator to look at the log files.

5.5 Stack Overflow

The performance signatures developed for Stack Overflow security testing are presented in Figures 16, 17, and 18.

Figure 16 shows a considerable increase in the number of active threads in the case of a *Stack Overflow* attack. The program used up the stack before crashing it and a large number of threads were used for this purpose thus explaining the increase in this parameter. Figures 17 and 18 show that when the program was exploited to reserve large chunks of memory, the swap memory percentage and the virtual memory usage elevated dramatically.

The interesting aspect of the test case *Stack Overflow* was that none of the off-the-shelf products detected the attack.

6. ANALYSIS OF EFFECTIVENESS OF PERFORMANCE SIGNATURE IN DETECTING SECURITY INTRUSIONS

The performance signatures derived from the execution of the security tests are presented in Table 2. In Table 3 the intrusion detection ability of the off-the-shelf tools composing the security infrastructure is shown.

Here a 'Y' indicates that the performance metric for that row was effective in indicating the presence of the specific attack defined for that column. An 'N' indicates that the performance metric gave no discernible indication of the attack. Ideally we would like to discover that one, or a small handful of, performance metrics would be 100% effective in

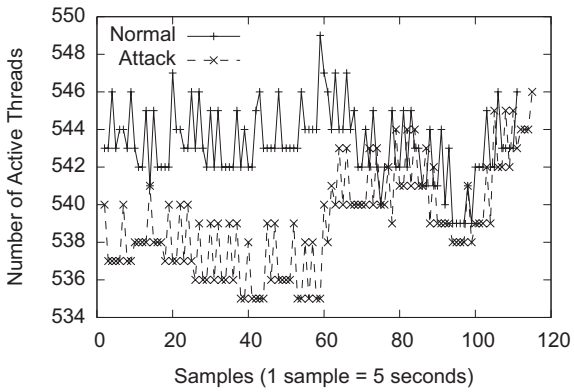

Figure 16: Number of active threads during normal load and stack overflow attack

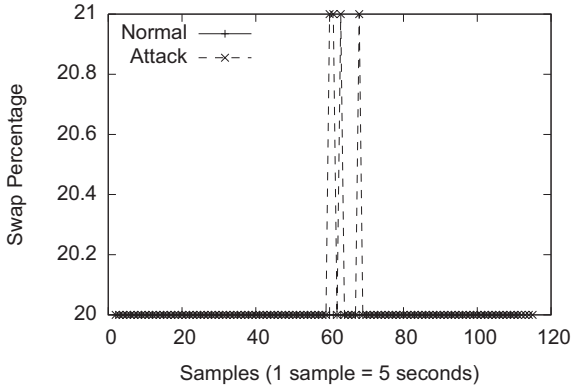

Figure 17: Swap memory usage during normal load and stack overflow attack

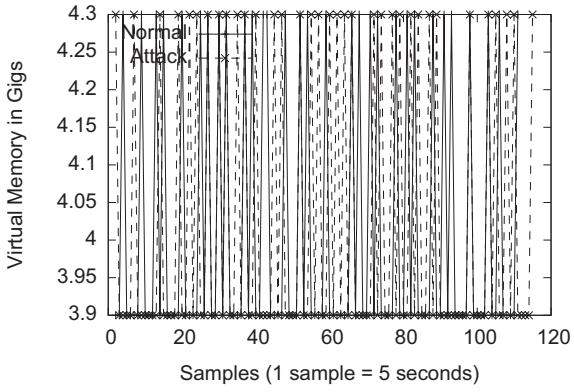

Figure 18: Virtual memory usage during normal load and stack overflow attack

Metric	DoS	Buffer Overflow	Stack Overflow	Man-in-the-Middle	SQL Injection	Success
Active threads	Y	Y	Y	Y	N	4
CPU Percentage	Y	Y	Y	Y	N	4
Memory Percentage	Y	N	N	Y	N	2
Interface Received bytes per sec	Y	N	N	N	N	1
Swap Percentage	Y	N	Y	Y	N	3
TCP Connections established	Y	N	N	N	N	1
TCP Resets	Y	N	N	Y	N	2
Interface Transmitted bytes per sec	Y	N	N	N	Y	2
Virtual Memory	N	N	Y	N	N	1
Working Set	Y	Y	N	Y	N	3

Table 2: Performance Signatures intrusion detection ability. Y = yes, N = no

Metric	DoS	Buffer Overflow	Stack Overflow	Man-in-the-Middle	SQL Injection	Success
System Log	N	Y	N	N	N	1
Browser Warning	N	N	N	Y	N	1
Snort Logs	Y	N	N	N	Y	2

Table 3: off-the-shelf security intrusion detection ability. Y = yes, N = no

indicating a broad set of security attacks. From Table 2, we see that most attacks were flagged by the Active Threads and the CPU Percentage performance signatures, while only the Interface Transmitted Bytes per Second (IfXmit) performance signature flagged the presence of the SQL Injection attack. If we evaluate the metrics by their number of successes in detecting attacks (indicated by the last column in the table), then we see that the Active Threads, CPU, then SWAP Percentage, Process Working Set, Memory Percentage, TCP Resets and IfXmit metrics were most successful on average in detecting attacks. A small and robust set of performance signatures for general security attack detection from the information in this table might include (a) CPU Percentage, (b) Active Threads, (c) SWAP Percentage, and (d) IfXmit signatures. Clearly more empirical studies are required.

Our initial study just scratches the surface of the total set of empirical studies necessary to make strong recommendations on an appropriate set of performance signatures for security attack detection. However, we are encouraged by our initial results.

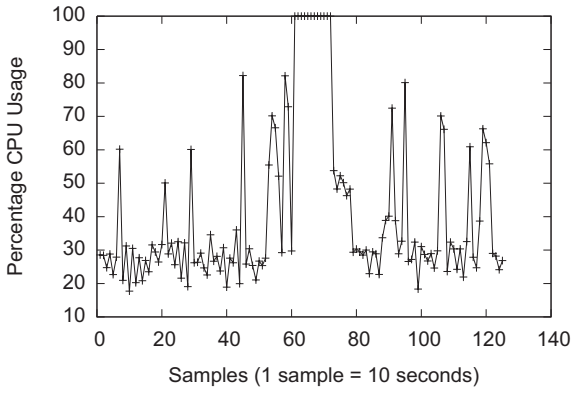

Figure 19: Increase in CPU usage due to a soft failure

7. DISTINGUISHING BETWEEN SOFT FAILURES AND SECURITY INTRUSIONS

The data analysis of performance signatures presented in Table 2 has motivated us to compare the performance signatures obtained from our security tests with some of the performance signatures we obtained due to non-security faults in earlier work [4]. When analyzing performance signatures that were derived from failure events, we have observed significant degradation with CPU spikes to 100% and constant increase in memory usage. In contrast, when security intrusions have occurred, we have observed smaller spikes in usage that are usually within a well-defined range. Figures 19 and 20 show examples of CPU and memory degradation due to a non-security fault.

Motivated by these observations, we present an extension of the bucket algorithm introduced in [5]. This will be used for detection of a multi-dimensional performance signature that can distinguish between performance issues and security intrusions. The monitoring infrastructure uses WMI to capture the performance signature periodically. We used five second periods in our security testing experiments. At each sampling period, the value of each component i of the performance signature $S_{N[i]}$ is estimated by counting the recent number of occurrences, d, of sample values that exceed $\overline{x[i]} + N[i]\sigma[i]$ where $\overline{x[i]}$ is the reference average expected value of the performance signature component i, $N[i]$, (i= 0,1,2,...,I), is the index to the current bucket of the i_{th} component of the performance signature, and $\sigma[i]$ is the reference expected standard deviation of the performance signature

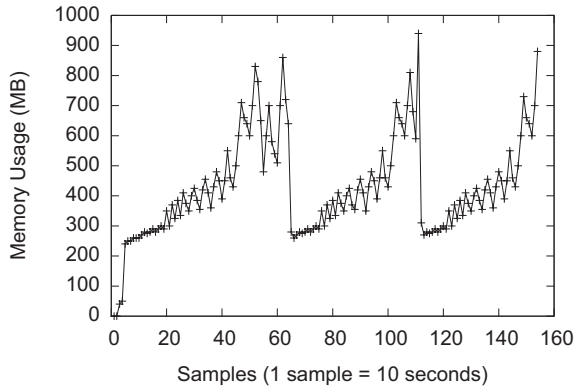

Figure 20: Decrease in available memory due to a soft failure

component i. $KIDS$ represents the total number of buckets required for a single component i of the algorithm for security intrusion detection. $KSOFT$ represents the total number of buckets required for a single component i of the algorithm for detection of a soft failure. $KSOFT > KIDS$. $D_{N[i]}$ represents the depth of bucket $N[i]$.

If an overflow is observed in any of the last available buckets i, the algorithm signals the detection of a soft failure or a security intrusion. The algorithm works by tracking the levels of K contiguous buckets for each component i. Therefore K times i buckets are monitored. At any given time, the level $d[i]$ of only the Nth bucket of each component i is considered. $N[i]$ is incremented when the current bucket overflows, i.e. when $d[i]$ first exceeds $D_{N[i]}$, and is decremented when the current bucket is emptied, i.e., when $d[i]$ next takes the value zero. Whenever the $Nth[i]$ bucket overflows, the depth $D_{N[i]+1}$ of the next bucket will be computed as $D_{N[i]+1} = D_{MAX[i]}/(\ S_{N[i]}\ -(\overline{x[i]}\ +\ N[i]\sigma[i]\ ))$, where $D_{MAX[i]}$ is the maximum depth configured for the first bucket of each component and I is the dimension of the performance signature vector.

7.1 Algorithm to Estimate Current Value of Monitored Performance Signature

1. for (i=0,i < I, i++) { /* do loop Steps 2-7*/
2. if (N[i] == KIDS) issue an security intrusion detection notification.
3. if (N[i] == KSOFT) issue a soft failure notification.
4. if ($S_{N[i]}$, > $\overline{x[i]}$ + N[i]σ[i])
   then
   do{ $d[i] := d[i] + 1$;}
   else
   do { $d[i] := d[i]$ - 1; }
5. if ($d[i] > D_{N[i]}$)
   do { $d[i] := 0$;
   $D_{N[i]+1} := D_{MAX[i]}$ /($S_{N[i]}$ - ($\overline{x[i]}$ + N[i]σ[i])) ;
   $N[i]:=N[i]+1$;}
6. if (($d[i] < 0$) AND ($N[i] > 0$))
   then
   do {$d[i] := D_{MAX[i]}$;
   $N[i] := N[i]$ -1;
   $D_{N[i]}: = D_{MAX[i]}$; }
7. if (($d[i] < 0$) AND ($N[i] == 0$))
   then
   do {$d[i] := 0$;}
   } /* end of do loop of Step 1 */

8. CONCLUSIONS AND FUTURE WORK

We have described the technique and the process through which we developed performance signatures for some common security tests. In addition, we have introduced an algorithm to be used in conjunction with the framework to automatically detect and distinguish between failures and security attacks.

We have successfully demonstrated that it is possible to obtain performance signatures based on security tests, system performance monitoring, and extensive logging from different security monitoring tools. We are very encouraged by these preliminary results and are planning to extend this research in the following ways:

- Analyze and tune our proposed monitoring algorithm against our set of security attacks. Our analysis will trade window size versus responsiveness, since too large a window will likely result in sluggish performance,
- Expand the definition of normal/baseline behavior for industrial software systems to accommodate varying load conditions based on time of day, day of week, or day of month. As the baseline conditions change, the variations in the relative values need to be adjusted appropriately,
- Develop performance signatures for additional security test cases,
- Develop an intelligent analyzer that can make assumptions about unknown performance signatures.

Figure 21 shows the proposed attack detection architecture. *System Performance Monitor & Alerter SPEMA* is implemented in the bucket framework with the aim of looking at the pre-defined signatures along with the other tools (*CurrPorts, Wireshark, System Log monitor*). When a signature is matched, the alerter (UI) provides live warning to the user and logs the attack. If *SPEMA* notices erratic behavior, it warns the user through the alerter and logs the possible attack (which can be further analyzed). *SPEMA* also logs the software degradation data. If the parameters are within permissible limits, it lets the data go through without logging it. Our future work will focus on fleshing out and developing this architecture.

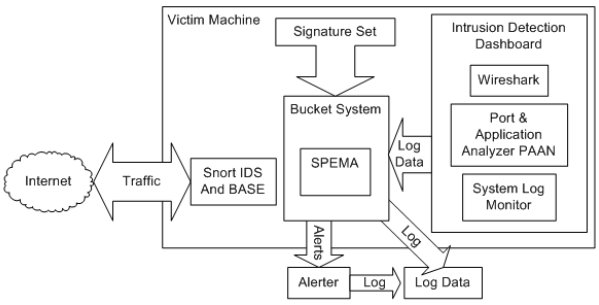

Figure 21: Proposed attack detection architecture

9. REFERENCES

[1] A. Avritzer and E. J. Weyuker. The Automatic Generation of Load Test Suites and the Assessment of the Resulting Software. *IEEE Trans. on Software Engineering*, Sept 1995, pp. 705-716.

[2] A. Avritzer and E. J. Weyuker, *Detecting failed processes using fault signatures*, International Computer Performance and Dependability Symposium, July, 1996.

[3] A. Avritzer and E. J. Weyuker, *Monitoring Smoothly Degrading Systems for Increased Dependability*, Empirical Software Engineering, Springer Netherlands, March 1997.

[4] A. Avritzer, J. P. Ros and E. J. Weyuker, *Estimating the CPU utilization of a rule-based system*, Proc. Fourth International Workshop on Software and Performance 2004, Redwood Shores, California, Jan, 2004, pp. 1–12.

[5] A. Avritzer, A. Bondi and E. J. Weyuker, *Ensuring Stable Performance for Systems that Degrade*, Proc. Fifth International Workshop on Software and Performance 2005, Palma de Mallorca, Spain, July, 2005, pp. 43–51.

[6] A. Avritzer, R. G. Cole and E. J. Weyuker, *Using performance signatures and software rejuvenation for worm mitigation in tactical MANETs*, Proc. Sixth International Workshop on Software and Performance 2007, Buenos Aires, Argentina, February, 2007, pp. 172–180.

[7] W. Diffie and M. E. Hellman, *New directions in cryptography.* IEEE Transactions on Information Theory, vol IT-22, Nov 1976, pp:644-654.

[8] S. A. Hofmeyr and S. Forrest and A. Somayaji, *Intrusion Detection Using Sequences of System Calls.* Journal of Computer Security, vol 6, No 3, 1998, pp 151-180.

[9] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton, Software rejuvenation:Analysis, module and applications. *Proc. Twenty-fifth International Symp. on Fault-Tolerant Computing*, 1995, pp. 381–390.

[10] IBM Cryptography Research Group. *http://domino.research.ibm.com/security*

[11] D. Khan. The code breakers. Macmillan, 1967.

[12] R. Mariani. Performance Signature: A qualitative approach to dependence guidance. International Computer Measurement Group Conference, pp 469-474, 2006.

[13] D. L .Oppenheimer and M. R. Martonosi, Performance Signatures: A Mechanism for Intrusion Detection. *Proceedings of the 1997 IEEE Information Survivability Workshop*, 1997. http://www.sysnet.ucsd.edu/ davi-dopp/pubs/perfsig.html.

[14] R. L. Rivest, A. Shamir and L. Adleman. *A method for obtaining digital signatures and public key cryptosystems.* Communications of the ACM, Feb. 1978, pp 120-126.

[15] http://www.wired.com/threatlevel/2009/01/professed-twitt/

[16] Snort http://www.snort.org/

[17] Cain & Abel http://www.oxid.it/cain.html

[18] Wireshark http://www.wireshark.org/

[19] Hyperic Sigar http://www.hyperic.com/products/sigar.html

[20] DoSHttp http://www.socketsoft.net/

[21] Base http://base.secureideas.net/

[22] CurrPorts http://www.nirsoft.net/utils/cports.html

[23] SysTracer http://www.blueproject.ro/systracer