

Host Based Intrusion Detection Methods

Michael Aiello

Polytechnic University of Brooklyn

Prepared for Joel Wein's Fall 2005 Operating Systems I (CS623)

1 Abstract

This paper evaluates three approaches to intrusion detection described at the USENIX 2005 conference. The first paper describes an extrusion based approach that monitors patterns of initiated connections. The second paper presents a method for emulating vulnerable sections of code in order to detect memory corruption and runtime errors that may be caused by attackers. The final paper presents a method for automatically exploiting pitfalls of a system call monitoring based approach. The author then notes that current host based intrusion detection systems limit their scopes of monitored events and suggests an improvement which combines the methods of two of the evaluated papers.

2 Introduction

Host based intrusion detection is the description of a variety of methods that attempt to detect malicious behavior on a host by monitoring changes to its internal state and operations performed on behalf of processes. Items that may be monitored include; network connections initiated by processes, system call sequences, changes to files (specifically system configuration files), changes to system state, changes to specific sections of memory and application log files. The fundamental assumption of a host based intrusion detection system is that malicious system behavior is definable and is also distinguishable from normal system behavior.

Host based intrusion detection systems have been available for quite some time labeled as anti virus

and host firewall products. These systems can be broken down into three general categories.

1. Signature based methods which filter incoming network traffic and monitor applications and file systems. These are generally described as anti virus methods and have a detection mechanism which is limited to a set of known signatures.
2. Methods in which users or administrators define a list of applications which are allowed to communicate externally and block all others.
3. A combination of the above

Several attacks on the above methods have been implemented by attackers (i.e. self morphing code and malicious white listing).

Three papers at the 2005 USENIX conference present methods for detecting intrusions on a host level without maintaining a malicious control or data signature database. Two of them describe detection methods which “learn” expected system behavior and attempt to identify anomalies.

When constructing a host based intrusion detection system, three major decisions must be made; what items will be monitored by the system, how the baseline “normal behavior” will be defined, and how to distinguish normal behavior from the malicious behavior. If a detection system can be defeated in any of these areas, it will fail at detecting intrusions.

This paper will evaluate three approaches to intrusion detection described at the USENIX 2005 conference. The first paper describes an extrusion

based approach that monitors patterns of initiated connections. The second paper presents a method for emulating vulnerable sections of code in order to detect memory corruption and runtime errors that may be caused by attackers. The final paper presents a method for automatically exploiting pitfalls of a system call monitoring based approach.

Finally, the author notes that current host based intrusion detection systems limit their scope of items which are monitored and suggests an improvement which combines the methods of two of the evaluated papers.

3 BINDER: An Extrusion-based Break-In Detector for Personal Computers

Binder attempts to detect intrusions based on the assumption that a compromised machine will produce “stealthy malicious outgoing network traffic” [4] which is uncharacteristic of the system. The focus of the paper is on user-end computers (as opposed to servers) because their method relies on correlating user input to triggered network events. The problem Binder attempts to solve is detecting intrusions that result in uncharacteristic network connection patterns due to activities of malicious software running due to an intrusion. It is a reactive approach to intrusion detection that is described as “extrusion based.” “The goal of our work is to detect worms, spyware and adware on personal computers” [4].

3.1 Why the problem is difficult

The problem is difficult because defining acceptable usage patterns in an automated manner for the resources they are monitoring is difficult. It is also difficult to then algorithmically detect anomalies in this usage pattern without generating large amounts of false positives. Fundamentally, it is difficult to describe system behavior in terms of extrusions.

3.2 Method

In this system, the items monitored are

1. User input (keyboard and mouse events)
2. Processes (process start and end events)

3. Network activity (connection request, data arrival and domain name lookup events)
4. Time intervals between the above

To collect “baseline behavior” the authors suggest allowing BINDER to run on uninfected machines for an extended period of time in order to best determine unique thresholds for each parameter to the detection algorithm. Once a baseline behavior is determined, the system then monitors network events and correlates them to previous user input. If a correlation can not be determined, the network event is deemed as rogue. “BINDER can detect the break-in by capturing the first connection made by its malicious processes as an extrusion. This is because their parent processes do not receive any user input before they are created. And they do not receive any user input after they are created” [4]. The system also includes white lists in which applications can freely use network resources without having previous user input. Three types of situations are tracked:

1. dNEW is the amount of time that elapses after a child process is spawned and a new connection request takes place.
2. dOLD is the amount of time that elapses after a user interacts with a process and a connection request takes place
3. dPREV is the amount of time that elapses after a request is made to host A from process P and another request is made to the same host A from process P.

The system takes “upper bound” parameters for each one of these parameters. They mention that they should be “selected on a per-user basis” [4] so that false positives are minimized. During their experiments, they take the 95th percentile values of each of the parameters based on historical data.

3.3 Experiments

The authors performed two evaluations on their system. First, an evaluation using previously collected trace data results as parameters for a real world intrusion was performed. “We collected traces of

User	90% (s)				95% (s)				99% (s)			
User	D_{old}	D_{new}	D_{prev}	# of FAs	D_{old}	D_{new}	D_{prev}	# of FAs	D_{old}	D_{new}	D_{prev}	# of FAs
A	18	11	142	15	33	15	752	5	79	21	4973	3
B	15	12	64	23	28	21	260	7	79	22	3329	5
C	14	14	28	20	25	15	134	5	74	33	2272	1
D	16	81	213	3	33	81	715	3	85	81	4611	2
E	19	12	539	5	32	14	541	4	93	90	4216	3
F	14	8	80	10	27	13	265	5	79	31	3633	2

Figure 1: Binder’s Parameter Selection and False Alarm (FA) Rates

user input, process information, and network traffic from the 6 computers that installed the prototype of BINDER” [4]. Unexpectedly, explicit parameter selections were not described in the paper. It is assumed the parameter selection came from Figure 1.

In their results, the authors do not describe which set of parameters was used for each user in their real-world experiment. They simply state that all malicious programs installed on the client machine (Gator, CNSMIN and Spydeleter) were detected after the computers were restarted by the BINDER system.

The second experiment involved controlled environment tests where parameter selection was clearly defined.

1. $d_{UpperOLD} = 30$ seconds
2. $d_{UpperNEW} = 30$ seconds
3. $d_{UpperPREV} = 800$ seconds

They test the controlled environment against the email worms Beagle, NetSky, Mydoom, and Swen. They again state that all malware programs were detected after the computers were restarted by the BINDER system.

3.4 Unique Contribution

There solution for intrusion detection unique because it examines intrusions from a purely reactive approach. It also differentiates itself from default-deny host based firewalls that ask the (often untrained) user if a process is allowed to access the network because it creates a usage profile automatically during the training phase.

3.5 Analysis

In the “attacks on binder” section, the authors describe weaknesses in their approach. These described attacks include:

1. Direct Attacks: disabling the detection system
2. Faking user input: Use the APIs that Binder monitors to send fake user input
3. Trick the user into generating input: Force a click on an malicious process’s window
4. Hiding under processes: Binder is unable to detect if a malicious library has been included by a laminate process
5. Covert channels: the malicious code uses an existing monitored process to perform its communication. (i.e. URL pre-pending and appending)
6. Using a user input to hide: Binder does not interpret the initial connection request made by a malicious process as malicious. If it is able to establish a connection “it just needs to maintain a connection to a collusive remote site to keep receiving some data packets regularly. Then BINDER would think any new connections made by it are triggered by those data packets.” [4]

I propose that the authors have left out some very important attacks:

1. A malicious application may add itself to Binder’s white list before performing any network events.

2. A malicious application may modify the dUpperOld, dUpperNew, dUpperPrev parameters in Binder.
3. A malicious application may only initiate a single connection, sends all relevant information across that connection and never run again.

I believe that the experiments performed by the authors do not wholly demonstrate their claim that “BINDER limits false alarms very well and can detect a large class of threats” [4]. False alarms are decreased by increasing dOLD dNEW and dPREV. This is an effective method, but the implications of increasing the window before a malicious connection is detected is not discussed or evaluated. Also, both the controlled and “real-world” experiments only evaluate a few known malicious applications in a controlled environment. These tests may have high internal validity for this set of attacks, but there are several other types of malicious applications which must be accounted for. The notion that a malicious program is able to have a single communication take place and still remain unnoticed is undesirable. It is possible that in that initial communication, all relevant information is sent and the attacker can accomplish their goal.

3.5.1 Suggested Experiment

The authors claimed that all attacks were without describing how long it took to detect the malicious applications or what the “false alarm” acceptance level was set to. A holistic experiment would be one in which the authors were able to vary the methods and timing of attacks, the types of requests generated by the injected malicious code, inter-request distances of the malicious code and input parameters to BINDER. I suggest an experiment where an out-of-band controllable piece of “malicious” code is installed on a test system which is running BINDER. The malicious software should be able to generate connection attempts based on schedules defined by the examiners. False positive and detection rates could then be examined for a large range of situations, possibly eliciting a better method for describing anomalous connection requests. The authors present recommendations for preventing some of the identified attacks on BINDER:

- Restrict conditions on how a normal connection may be triggered;
- Extend the Process Monitor to Thread Monitor;
- Add a System API Monitor to BINDER.

3.5.2 Suggested Improvement

I feel it may be advantageous to monitor more types of events occurring on the system besides those which imply user interaction. These may include file read and write events, and other device events. Malware has a variety of intentions. The fundamental idea of BINDER is that malware always needs to “phone home” and communicate with another machine. Unfortunately, sometimes this is not the case (i.e. key loggers which are then physically picked up by an individual, printing of documents, adding another user to the system via /etc/passwd). Adding additional events to BINDER’s log (particularly file read and writes events) would greatly improve the robustness of its detection mechanism. I believe it is possible to implement such a change without significant effort by treating file read and write events in the same manner as the system currently treats connection requests. This would allow BINDER to monitor a broader range of malicious activities.

4 Building a reactive immune system for software services

The authors of this paper realize that it is possible to detect memory corruption based intrusions and system errors (divide by 0, segmentation fault etc.) if the instructions in “vulnerable” areas of code are fully emulated before changes they would cause are actually committed. The focus of the paper is on building a system that allows software to become self-healing through selective emulation and state restoration. However, their approach to real time intrusion detection is a unique contribution.

As opposed to BINDER, the intrusion detection methods presented in [5] are based on the assumption that the only attack vector which we wish to protect is a programming error within a running application. This type of scenario is most often found in single function server environments.

4.1 Why the problem is difficult

The problem which they are trying to solve is how to engineer a system that is able to detect and recover from serious system errors and memory corruption based intrusions. It is difficult to have a live system detect serious errors (particularly non-deterministic ones) before they occur. This is because the error generally leads to a crash or control transfer to a malicious attacker before recovery mechanisms can act.

4.2 Unique Contribution

The realization that in fully virtualized languages (such as Java) memory corruption problems are prevented by monitoring memory modification drives their proposed solution to a specific type intrusion detection. The intuition the authors present is that if sections where these type of errors can occur are pre-emulated, the errors can be detected and recovered from gracefully.

4.3 Method

In this approach, developers add instructions to a program's source code indicating "vulnerable" sections which should be emulated and monitored. "Upon entering the vulnerable section of code, the emulator snapshots the program state and executes all instructions on the virtual processor. When the program counter references the first instruction outside the bounds of emulation, the virtual processor copies its internal state back to the real CPU." [5] This creates a buffer in which sections of applications may fail and recover in emulated mode and not affect the overall application. The emulator is able to detect the following errors and attacks.

1. Computational denial of service attacks: Programmers specify a maximum number of instructions to execute for the emulated code. If the threshold is reached, it is determined that a computational denial of service is taking place and the emulator enters recovery mode.
2. Divide by zero errors: The emulator checks the operand to the div instruction. If it is a 0 before execution the emulator enters recovery mode.
3. Improper memory dereferencing: "The emulator verifies whether the source or destination addresses of any memory access point to a page that is mapped to the process address space." [5] If it does point to an out of range page, the emulator enters recovery mode.
4. Buffer overflows: The emulator pads "memory surrounding the vulnerable buffer" [5] with a one byte "canary". The emulator monitors if that byte is ever written to. If it is, then an overflow has occurred and the emulator enters recovery mode.

Once the system enters recovery mode, all memory changes since the invocation of the emulator are rolled back and a type appropriate error is returned based on the expected return type for the function (i.e. -1 for int, 0 for unsigned int). This method for emulation and recovery is labeled Selective Transactional Emulation (STEM).

4.4 Experiments

The authors clearly state what they attempt to accomplish with their experimentation:

- Can the system detect real attacks and faults and react to them?
- How effective is our error virtualization hypothesis as a recovery mechanism? Does it work for real software?
- What is the performance impact of emulation, and what is the gain to be had by using selective emulation?

In order to answer these questions, two tests were performed on specific, vulnerable functionality of widely used applications (SSH-1.2.27, BIND 8.2.2 and Apache2) and a performance evaluation on Apache2. The authors also emulated entire shell applications and compared their performance to non emulated versions.

In the first test, the virtualization recovery mechanism was evaluated. To accomplish this, leaf functions of each application were identified and marked for emulation. Afterward, "a script inserted an early return in all of the leaf functions, simulating an

aborted function.” [5] Invocations of the application then caused individual leaf functions to abort early and return. Each invocation was tested to see if it could successfully process a request and not crash.

Service	Success	Failed	Percentage Success
Apache	139	15	90%
SSH	72	9	89%
BIND	67	8	88%

Figure 2: Early return from leaf function success rate.

The results in figure 2 show that “error virtualization” has a high probability of returning the program to a state where it is able to continue execution.

In the second test, known exploits for each of the services were tested against the application with the vulnerable functionality emulated by STEM. All of the exploits took advantage of incorrect memory handling which lead to buffer overflows. In all 3 cases, the emulator detected that a buffer overflow was about to occur and prevented the attack from succeeding. The programs gracefully recovered and were able to serve subsequent requests.

Finally, the authors evaluated the performance impact of their emulator. They compared the number of requests the apache web server is able to process during normal execution (apache-mainloop), under full STEM emulation for the mainloop (libtasvm-mainloop), emulating only the “vulnerable sections” of the parse-uri functionality (libtasvm-parse-uri), emulating only the “vulnerable sections” of the header parser (libtasvm-header-parser) and emulation using the independent Valgrind emulator (valgrind-apache). “vulnerable sections” were those sections identified in the RATS static code analysis tool. “The majority of [these sections] contained fixed size local buffers” [5].

As seen in Figure 3, there is a significant impact when emulating the entire mainloop. However, when emulating only specific functionality, the STEM emulator performs better than other emulators. It is able to serve at least half as many of the pages as the non-emulated version. As displayed in Figure 4, full emulation of apache’s mainloop is approximately 44 times slower than normal conditions. Full emulation of entire shell ap-

Apache	trials	Mean	Std. Dev.
Normal	18	6314	847
STEM	18	277927	74488
Valgrind	18	34192	11204

Figure 4: “Timing of main request processing loop. Times are in microseconds. This table shows the overhead of running the whole primary request handling mechanism inside the emulator. In each trial a user thread issued an HTTP GET request.”[4]

plications yielded the results displayed in figure 5. These results show that there is a significant performance impact when executing entire programs in the STEM emulator. The authors point out that they did not attempt to significantly optimize the emulation through methods such as caching translated instructions.

4.5 Analysis

The unique contribution of this paper is the idea that “selective transactional emulation” can be used to detect and recover from programming errors. In terms of server based intrusion detection and recovery, the method does a good job accomplishing its goals of maintaining reliability and preventing memory based intrusions.

4.5.1 Selective Transactional Emulation

In regards to of the recovery mechanism, atomic transactions make sense and are necessary for data consistency but this type of model many not be appropriate for program state transitions. Rolling back a state “transaction” while not modifying derived program control could lead to inconsistent situation. Consider the example of authorizing a user. If the authorizing mechanism is under emulation by STEM and during emulation an error is detected, state is returned to what it was before the authorizing mechanism was invoked. If the application was designed in such a way that the authorizing mechanism ends the connection attempt when authorization fails, an attacker may be able to bypass authorization by stopping its emulation. Dangerous pseudocode example:

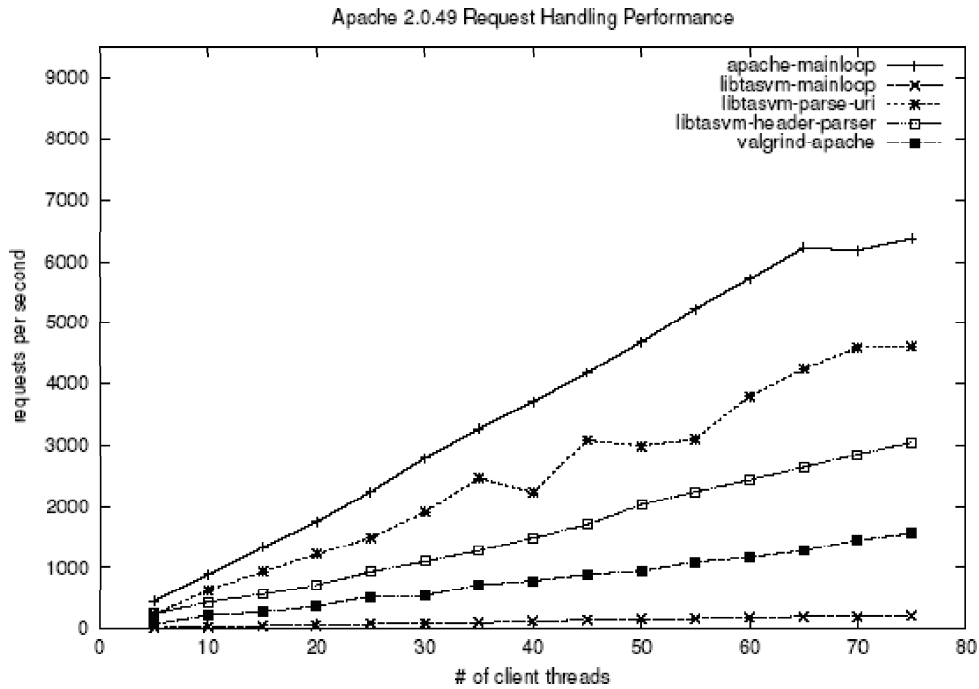


Figure 3: STEM’s Performance emulating various sections of the Apache web server compared to non-emulated apache and Valgrind

Test Type	trials	mean (s)	Std. Dev.	Min	Max	Instr. Emulated
ls (non-emu)	25	0.12	0.009	0.121	0.167	0
ls (emu)	25	42.32	0.182	42.19	43.012	18,000,000
cp (non-emu)	25	16.63	0.707	15.80	17.61	0
cp (emu)	25	21.45	0.871	20.31	23.42	2,100,000
cat (non-emu)	25	7.56	0.05	7.48	7.65	0
cat (emu)	25	8.75	0.08	8.64	8.99	947,892

Figure 5: STEM’s comparative performance emulating entire shell programs

```

Initialize()
ReadUserCradentials()
Begin STEM emulation
If ValidateUserCradentials()=False
    End Thread
End STEM emulation
PerformSensitiveOperation()

```

If there is a runtime error while validating user credentials, the validation mechanism is bypassed a sensitive operation is performed by an unauthorized user.

The current emulation cost is prohibitive to use the emulator in a production environment. Finally, in order to use STEM, it is necessary to add code that identifies vulnerable sections of applications. This may prove very difficult and time consuming in large applications.

4.5.2 Suggested Experiment

A broad range of attack methods are not tested. The authors claim that their system can prevent “computational denial of service attacks.” However, in the experimental section, only examples of memory corruption based attacks are evaluated. Solving the problem of computational denial of service attacks (forcing the program into infinite loops or excessive calculations) is difficult and should be evaluated. A formal experiment evaluating computational denial of service should be done. The experiments succeed in answering the last two questions that were posed (performance impacts and virtualization techniques validity).

4.5.3 Alternate Suggested Approach

Fundamentally, the authors have described a method for marking vulnerable sections of code and handling specific errors in those sections. This maps an infinite set of unknown runtime errors in those sections to a specific set of return values. This type of behavior could be replicated by implementing try, catch all blocks along with clever memory watching.

Their canary monitoring method for detecting memory corruption during emulation could also be replicated (without emulation) by mapping identified addresses at the ends of fixed length buffers to

exceptions. When these addresses are read or written to, an exception could be thrown.

5 Automating Mimicry attacks using static binary analysis

5.1 Background

The traditional approach to untrained host based intrusion detection is to presume that in order for an attacker to perform malicious behavior, fundamental control of the system must be obtained. Once this occurs, attackers will then execute their own sequence of malicious instructions. In “A Sense of Self for UNIX Processes” [2], Forrest proposes that sequences of system calls can successfully describe the behavior of a system. The paper suggests that if these sequences can be recorded during normal system behavior, the resultant set of valid calls can be compared to future sequences to detect malicious behavior. If an out of sequence call occurs, the behavior could then be considered suspicious and cause an alarm. While successful at detecting simple exploits, several problems were discovered in the approach. Foremost, attacks can be created that “mimic” sequences of common system calls on most systems and can therefore evade detection. Other weaknesses include

1. Attacking limitations on the length of system call sequences recorded by the system [6]. [2] suggests a system call sequence length of 5 to 10 during training to get an optimal false positive rate. Unfortunately, an attacker is able to create a sequence of system calls that may perform complex operations by forcing their malicious code to use more than 10 system calls (while maintaining the order of a known sequence of calls).
2. Parameters to the system calls are not recorded in the sequence database. An attacker is able to change parameters to system calls without raising attention. This is particularly dangerous with calls such as `execve` Consider:

```

execve("ls -l /home/")
vs
execve("rm -rf /")

```

In order to combat these downfalls, [3] introduces a method which adds “call contexts” to the data sets of sequences. These call contexts keep track of stack information, the program counter and addresses of calling instructions. Validating this information “makes sure that [a] system call was made by the application code and after [the] system call has finished, control is guaranteed to return to the original application code” [1]. This makes the problem of arbitrary code execution by maliciously altering program control flow far more complex. Using this context validation method, attackers must not only follow a known sequence of system calls, but the invocations of system calls also be from the correct “contexts”.

5.2 Automating mimicry attacks using static binary analysis

In “Automating Mimicry Attacks Using Static Binary Analysis” [1], the authors demonstrate that, even under these highly monitored circumstances, it is possible to execute malicious instructions by manipulating parts of the system that can not be monitored efficiently.

5.3 Why the problem is difficult

The goal of [1] is to defeat a system which is specifically designed to thwart mimicry attacks by requiring a correct context. The authors of [3] attempt to make complicate mimicry attacks by requiring calls to be made from specific call contexts. In order to defeat such a scheme, an attacker must appropriately read and duplicate (or predict) call contexts without using system calls before jumping to malicious code. Essentially, for an attack to be successful, it must be able manipulate program control by only using instructions already defined in accessible memory areas.

5.4 Unique Contribution

The authors of [1] clearly describe their unique contributions:

- “We describe novel attack techniques against two well-known intrusion detection systems that evade the extended detection features and

reduce the task of the intruder to a traditional mimicry attack.” The detection systems are those described in [2] and [3].

- “We implemented a tool that allows the automated application of our techniques by statically analyzing the victim binary.”
- “We present experiments where our tool was used to generate exploits against vulnerable sample programs. In addition, our system was run on real-world applications to demonstrate the practical applicability of our techniques.” [1]

5.5 Method

Their method is based off of the fact that it is possible for an attacker to invoke a single system call by not modifying existing program context. They then demonstrate how it is possible to continuously regain control of the system by carefully modifying items in memory (usually using move instructions that modify the procedure linkage table) without using system calls. By modifying these function pointers, an attacker potentially force the application to invoke malicious code from a valid context. This is accomplished by replacing the actual function pointer with the address of the attacker’s code or the known address of a system call in a linked library. Using a series of modified function and library pointers, one is able to create a sequence of system calls (from their original contexts) without modifying any of the related items presented in [3].

The authors present a tool which automates the identification of sequences of pointer replacements that would result in a context appropriate mimicry attack. The tool symbolically executes compiled binaries to elicit the replacements.

5.6 Symbolic Execution

“Symbolic execution is a technique that interpretatively executes a program, using symbolic expressions instead of real values as input.” [1]. The authors maintain a history of program state throughout the execution and identify items (register values, and recently accessed memory values) that change between each state. A class of instructions that are

able to “modify code pointers to point to the attacker code” [1] are identified and stored by the symbolic executor. Essentially, the symbolic executor is identifying potential configurations that allow a jump target address to be modified. If an attacker is able to control these addresses, the attacker is able to manipulate program control.

5.7 Determining if overwriting a jump target address results in the ability to execute malicious code

The authors of [1] realize that it is possible to represent these program control flow as a system of linear equations. “A symbolic term [in an equation] expresses the current value of a register or a memory location as a function of the initial values ... the symbolic expressions are integer polynomials over variables that describe the initial state of the system” [1]. If a constraint is then applied to this equation such that the resultant target address is the address of the attacker’s code and the equation is solvable, a configuration can be generated so that control of the program is directed to the attacker’s code. This is because “the configuration fulfills the path constraints of the current symbolic execution thread, the actual execution will follow the path of this thread” [1].

5.8 Experiments

The authors wish to demonstrate that their method is able to generate configurations that result in the execution of attacker code after a system call takes place.

They first evaluate their method on the host based IDS implemented in [3] and [2]. The experiment is deemed successful if an attacker is able execute system calls (in context) without triggering the detection systems. Three simple applications with easily identifiable memory corruption errors were constructed. Using the symbolic executor, all possible paths where the attacker could regain program control were identified. In all three test cases, the binary analysis was able to determine a configuration which would allow the attacker to regain control without triggering either of the host based IDSs in [3] and [2]. Malicious code which did not violate

the call sequence was then able to perform actions on behalf of the attacker, thus reducing the problem to a traditional mimicry attack.

In their second experiment, the symbolic execution tool examined 3 widely used applications (apache2, netkit ftpd and imapd) “To determine whether it is possible to find a configuration and a sequence of instruction such that the control flow can be diverted to an arbitrary address.” This is determined by the ability of the symbolic execution to direct program control to 100 randomly selected addresses within the program.

Program	Instr.	Success	Failed	
			Return	Exhaust
apache2	51,862	83	12	5
ftpd	9,127	93	7	0
imapd	133,427	88	11	1

Figure 6: Symbolic execution results for real world applications. “Return” is the number of addresses that failed because the program returned immediately after executing the instruction. “Exhaust” is the number of addresses that the system was unable to find configurations that lead to execution at that address.

The real word application results show that it is highly probable that if an attacker is able to inject instructions into memory, it is possible to invoke malicious system calls from correct contexts, again reducing the attack to a simple mimicry attack.

Finally, total execution time using their symbolic execution tool for each real world application analysis is presented. The times range from 0.3 to 12.4 seconds.

6 Analysis

The authors present and demonstrate a successful method for defeating a leading context sensitive system call based intrusion detection system. In their experiments, both known and unknown (real-world) applications are examined. The results demonstrate that complications imposed on an attacker by including program context into the system call database can be bypassed. Their method is generic (and

unique) in its ability to detect configurations which result in the direction of program control flow to an attacker's code. It successfully defeats the intrusion detection methods presented in [3] and [2].

6.1 Suggested Experiment

An additional experiment in which the techniques in [3] monitor a previous release of a real world application that has known vulnerabilities and then attacked using a configuration generated by the symbolic executor, would be a stronger demonstration than the 100 random addresses control modification experiment presented in the paper. This would show, definitively, that attacks on system call sequence host based intrusion detection systems can be automated.

6.2 Other applications of symbolic execution

The authors have created a method to determine if there is any condition that results in program control being transferred to a specific address. The authors propose that memory corruption attacks could be detected by identifying configurations "which the current functions return address is overwritten" [1]. In addition, complete control path histories are provided if any configuration is discovered that allows instructions at specified address to be executed. I feel that these histories could be used to help developers reduce control paths to sections of code that require authorization or special permissioning.

In terms of host based intrusion detection, I feel that it may be possible to automatically define "inappropriate" execution paths using this tool. It would be an approach similar to system call monitoring; inappropriate paths could be defined as those in which major process jump or branch target address changes are not part of the baseline state change operation. Jump and branch target addresses (as defined by initial symbolic execution) would then be monitored by the system and any inappropriate change would result in an alarm.

7 Conclusions and Future Work

This paper examined three approaches to host based intrusion detection. BINDER presented an intrusion based method that correlated process activity with network activity to identify anomalies. BINDER is able to detect a specific types of malicious applications that use the network in a common way. The work in [5] presents a method for detecting specific types of programming errors that may lead to run time errors by selectively emulating and monitoring sections of the application. At great performance cost, is able to detect (and recover from) memory corruption attacks which may lead to an attacker executing malicious code. Finally, the methods presented in [2] and [3] attempt to describe the behavior of a system through system call sequences and their contexts. Using a database of valid call sequences and contexts, the system is able to detect if a specific process is behaving maliciously. While promising, this method was defeated using static binary analysis in [1].

I believe that that the work in [1] has the most impact on the host based intrusion detection research. It demonstrates that the fundamental assumption in [2] (that sequences of system calls effectively describe a program's behavior) may not be correct and a better metric for describing a system's fundamental behavior must be created.

Current approaches may be too specific in their detection methods and their definition of "normal" behavior. BINDER, for example, is unable to detect malicious applications that do not use network resources. "Normal" behavior is defined in terms of the program's usage of the network. The approaches of [3] and [5] are unable to detect if a malicious application is installed by a legitimate user (they are only able to detect if a "normal" process begins acting maliciously). In this case, "normal" behavior is defined in terms of the sequence of system calls the application uses. I believe that these definitions do not wholly encompass the behavior of a system and should be re-examined.

A complete host based intrusion detection system should be able to detect intrusions due to both attackers taking advantage of programming errors and users unknowingly installing malicious applications. This type of activity should be automatically

detected without querying the an untrained user for rule sets and without the notion of white lists. I believe that it would be possible to move closer to such a system by combining the time based extrusion detection ideas presented in BINDER [4] with system call sequence ideas presented in “A Sense of self for Unix Processes [3]”. This would define “normal” behavior in terms of the system calls a process executes as they occur in reaction to network, user input and time events.

A system that takes into account the time between system calls and network events along with a database of allowable system call sequences and contexts would successfully thwart the attack method presented in [1]. Statistics on the time between two specific system calls could be stored along with each appropriate system call sequence. The attacker must not only be able to mimic the sequence of system calls but must also predict the amount of time their operations should take to execute. This method further complicates the attacker’s ability to maintain program control. As in BINDER, network events should relate to specific system call sequences and network responses. If a network event is usually associated with a small set of system call sequences, similar network events should lead to the same set of sequences. If a sequence of system calls outside of the event’s related set begins to take place, the behavior should be considered malicious. I believe that adding network events and the time factor to the ideas presented in [3] has potential to solve the two problems with the host based intrusion detection methods presented at the USENIX 2005 conference:

1. The method presented in [3] does not successfully detect if the user has been tricked into installing a malicious program. By tracking system calls that result from network events, the system is able to detect malicious activity that results from users mistakenly installing malware.
2. The method presented in [4] does not detect malicious applications that do not interact with the network. If system calls and the time between their execution in a certain process become part of a user’s “normal” activity profile, malicious applications that perform any type

of system call (as opposed to only focusing on network events) would be identified.

References

- [1] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, Giovanni Vigna *Automating Mimicry Attacks Using Static Binary Analysis* 2005 USENIX Conference Proceedings
- [2] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, Thomas A. Longstaff *A Sense of Self for Unix Processes* Proceedings of the 1996 IEEE Symposium on Security and Privacy
- [3] Jonathon T. Giffin Somesh Jha Barton P. Miller *Efficient Context-Sensitive Intrusion Detection* In 11th Annual Network and Distributed Systems Security Symposium (NDSS), San Diego, California, February 2004.
- [4] Weidong Cuiy, Randy H. Katzy, Wai-tian Tanz *BINDER: An Extrusion-based Break-In Detector for Personal Computers* 2005 USENIX Conference Proceedings
- [5] Stelios Sidiroglou, Michael E. Locasto, Stephen W. Boyd, and Angelos D. Keromytis *Building a Reactive Immune System for Software Services* 2005 USENIX Conference Proceedings
- [6] K. Tan, K. Killourhy, and R. Maxion. *Undermining an Anomaly-Based Intrusion Detection System Using Common Exploits* 5th Symposium on Recent Advances in Intrusion Detection (RAID), 2002.