In Proceedings of the IEEE Computational Intelligence for Security and Defense Applications (CISDA-2007), April 2007

Automatically Evading IDS Using GP Authored Attacks

H. Güneş Kayacık, A. Nur Zincir-Heywood, Malcolm I. Heywood

Dalhousie University, Faculty of Computer Science,
6050 University Avenue, Halifax, Nova Scotia. B3H 1W5

***Abstract*-A mimicry attack is a type of attack where the basic steps of a minimalist 'core' attack are used to design multiple attacks achieving the same objective from the same application. Research in mimicry attacks is valuable in determining and eliminating weaknesses of detectors. In this work, we provide a genetic programming based automated process for designing all components of a mimicry attack relative to the Stide detector under a vulnerable Traceroute application. Results indicate that the automatic process is able to generate mimicry attacks that reduce the alarm rate from ~65% of the original attack, to ~2.7%, effectively making the attack indistinguishable from normal behaviors.**

I. Introduction

Intrusion detection system (IDS) developers and Black hat attackers continually play a game of point-counterpoint when it comes to IDS technology. Black hat attackers continually develop methods to evade or bypass IDSs while IDS developers and system administrators continually counteract these methods with patches and new releases. Penetration testing for IDSs is a relatively new area compared with similar tests on cryptographic protocols. The main idea behind penetration testing is to locate vulnerabilities and holes in a system before attackers exploit them. Thus, administrators can test their IDSs to establish the robustness of such systems in real world situations by using the same tools and methodologies as used by attackers. In a way, administrators start acting as white hat attackers to evade their own systems in order to identify the vulnerabilities.

Due to the inherent complexities involved in capturing, analyzing and understanding network/host traffic there are several common techniques that can be used to exploit IDSs: detector string matching weaknesses, session assembly weaknesses and denial of service techniques. In this work, we are interested in automating evasion by making use of string matching weaknesses. The approach employed is based on the automatic generation of mimicry attacks to perform evasion. A mimicry attack is defined in the literature as an attack crafted by producing a legitimate sequence of system calls while performing malicious actions [1]. In this work, we investigate the possibility of evolving mimicry attacks using genetic programming in order to evade an anomaly host based detection system. We automate a white hat attacker against an anomaly host based detection system, Stide [2] for penetration testing. Several researchers have previously identified a number of evasion attacks on network based IDSs with misuse detection [3-6], and host based IDSs with anomaly detection [1, 7-9].

However, this work addresses two important issues that have not been considered in the related research: (i) Results used for characterizing the success of mimicry attacks in compromising detectors to date have only reported performance after the attacker gains control of the application [1, 7-9]. They therefore overlook the actions of the attacker before control is obtained, where this should also be reflected in the anomaly rate returned by the detector. We demonstrate that this provides anomaly rates unreflective of the true detection rate. (ii) Results to date generally omit the parameters of system calls for the development of mimicry attacks [1, 7-9]. We believe this is unrealistic since opening a password file is generally more significant than opening a JPG image. The typical argument for ignoring such information is that IDSs generally do not use arguments during detection. However, this rather curtails the development of detectors [10] that would be able to make performance gains by making use of this information.

Our approach for developing mimicry attacks not only automates the design of appropriate system call sequences but also provides the corresponding parameters. That is to say, system call parameters convey crucial information about the nature of a session and should therefore be incorporated into the design process. In addition, in our approach, the anomaly rates used to characterize malicious code reflect the entire session of the attacker gaining the control of the system and the attack itself.

The remainder of the paper is organized as follows. Section 2 establishes the position of this work, relative to earlier research on mimicry attack design. The methodology for automating malicious code design within a buffer overflow context is discussed in Section 3. Experimental results are presented in Section 4, and a discussion presented in Section 5. Conclusions are drawn in Section 6.

II. Related Work

Previous work in evading anomaly host based detectors can be divided into two categories: research on detectors and research on mimicry attacks.

In terms of research on detectors, Forrest *et al.* [11] employ a methodology based on immune systems, which aims to distinguish 'self' from 'non-self'. To do so, the authors employed a sliding window over the sequence of system calls that the application makes during normal use.

The patterns formed by the sliding window are stored in a table, which establish the normal behavior model. During the detection phase, if the pattern from the sliding window is not in the normal behavior it is considered as a mismatch. This approach was implemented as the Stide detector [2]. Various improvements to the original scheme have been proposed in [12-15]. To the best of our knowledge, none of these improvements where made open source, unlike Stide [2]. Moreover, none of these methods included system arguments in the detector. However, Mutz *et al.* [10] recently proposed a host-based anomaly detection system where multiple detection models are applied to system call arguments and an overall aggregate score of these models is introduced to determine if an event is an attack or not.

On the other hand, in terms of mimicry attack research, Wagner *et al.* [7], investigated an approach to alter the system call sequence of an attack in order to render it undetectable to a specific (host based) IDS. Given a minimum sequence of malicious system calls able to support execution of a successful attack – the *core attack* – their goal was to find other sequences of system calls that avoid detection by the target IDS yet still achieves the objective of the original attack. This was achieved by manually adding system calls that have no effect on the success of the attack. Similarly, Tan *et al.* [8], aimed to undermine the anomaly based IDS Stide [2] by identifying weaknesses and modifying the malicious system call sequences to exploit these limitations. To do so, they first modified the attack by hand to change the ownership of a critical file. Secondly, they inserted system calls from data characterizing normal behavior into the malicious system call sequence. Vigna *et al.* [3] described a methodology to generate variations of an attack to test the quality of detection signatures. Stochastic modification of attack code was employed to generate variants of attacks to render the attack undetectable. Techniques such as packet splitting, evasion and polymorphic shellcode were discussed. In [1], the authors developed a static analysis tool for Intel x86 binaries in order to automatically identify instructions that can be used to redirect control flow. They use symbolic execution to achieve this. To date [1], [5] and [6] are the only works in mimicry attack research that automate the development of mimicry attacks. It should be noted here that in [5] the automation is performed against a misuse network based IDS, namely SNORT, and in [1] automation is done using a static tool at the Intel x86 assembly level against anomaly host based detection systems. Finally, a virtual vulnerability is considered in [6] and exploits evolved under the Genetic Programming paradigm using the Intel x86 assembly language. Validation of the exploits is again performed using the SNORT signature based detector.

In this work, we are using a similar approach to [5], [6], i.e. employing evolutionary computation techniques to evolve mimicry attacks, but this time against an anomaly host based detector as opposed to a misuse network based detector. Moreover, the system developed here works at the system call level as opposed to the approach taken in [1], [6], i.e. Intel x86 assembly level. In addition, all of the cases discussed above assume a methodology in which anomaly rates used to justify mimicry attacks do not include supporting code necessary to correctly set up the target application exploit. As such, detectors that recognized this behavior (i.e. before the attack was even launched) would be penalized, whereas detecting the latter behavior would enable preemption of an attack.

III. Methodology

Our objective is to develop an automated process for building "white-hat" attackers within a mimicry attack context. By 'mimicry' we assume the availability of the 'core' attack, where this establishes a series of behavioral objectives associated with the exploit. The goal of the automated white hat attacker will therefore be to establish as many specific attacks corresponding to the exploit associated with the 'core' attack. Candidate mimicry attacks will take the form of system call sequences that can avoid detection or at least minimize the anomaly rate at the corresponding detector, in this case Stide. By "white hat," we imply that the underlying objective is to use the attacks to improve the design of corresponding detectors via penetration testing.

Previous research has established the suitability of the evolutionary computation (EC) machine learning paradigm as an appropriate process for automating specific processes associated with the design of buffer overflow attacks [5], [6]. In this work, we extend the approach to a general framework for mimicry attack generation, based on the evolution of system call sequences rather than generic parameters of an attack. The general motivations for using GP within this context are as follows,

- *Goal Based Objectives*: All machine-learning algorithms require a method for expressing the suitability of the current solution. This typically takes the form of a distance metric (e.g. sum square error) evaluated over a set of training exemplars. Such an approach implies that it is possible to define the behavior relative to a set of exemplars describing input and desired output behaviors. This mode of operation has lead to the widespread use of machine learning methods within the context of IDS detectors. However, in the case of mimicry attack generation the goal is to discover programs (of system calls) that mimic the behavior of a predefined 'core' (buffer overflow) attack. Learning is therefore directed by information supplied following an interaction with an environment. The environment in this case takes two forms, the anomaly rate returned by Stide for the candidate attack (suggested by GP in this case) and the degree to which generic goals associated with achieving the 'core' attack are reached. Such a mode of operation precludes the vast majority of machine learning paradigms as they make explicit assumptions regarding the relationship between representation (the components from which a solution is built) and how the objective is specified [16]. Typical examples include the smoothness constraint associated with neural networks or kernel

methods. Conversely, GP has no limitation on the formulation of objectives.

- *Representation*: Given that the objective is to design mimicry attacks, it follows that the representation utilized by the machine learning methodology must take the form of the system call sequences that explicitly correspond to the attack. This requirement precludes the utility of any other machine learning technique. For example, kernel and neural network models are based on an abstract connectionist representation, broadly applicable to data driven classification, regression, and clustering problem domains. Decision tree methods provide solutions that take the form of a set of partitioning rules. In essence all these methods utilize a representation motivated by a bias to a data driven model of learning.
- *Intron Code*: Solutions from GP take the form of a program in a predefined language. However, not all instructions comprising a solution necessarily contribute to the underlying operation of the individual, where this artifact is synonymous with 'introns' of biological genomes. Such introns result from the stochastic action of the search operators (crossover and mutation) and might account for as much as 80% of the instructions in an individual. Typically, instructions corresponding to intron behavior are removed post learning, as they make no contribution to the (functional) operation of the individual. In the context of this work, however, intron code aids the obfuscation of the real intent of the code. Thus, although not contributing to the design of a valid exploit, code corresponding to intron behavior will aid the minimization of detector anomaly rates.

In the following, we introduce the characteristics of the detector and application before detailing the GP framework utilized for automating mimicry attack generation under a core buffer overflow attack.

A. Anomaly Detector

Anomaly detection systems attempt to build models of normal user behavior and use this as the basis for detecting suspicious activities. This way, known and unknown (i.e. new) attacks can be detected as long as the attack behavior deviates sufficiently from the normal behavior. Needless to say, if the attack is sufficiently similar to the normal behavior, it may not be detected. However, user behavior itself is not constant, thus even the normal activities of a user may start raising alarms.

In this work, Stide [2] is used as the target anomaly detector, since it is the only anomaly host based detector that is currently available as open source. It is installed/configured using the same parameters as previous mimicry attack research [1, 7-9]. Moreover, a wide range of related research performed in vulnerability or penetration testing employ Stide as discussed in Section 2.

B. Vulnerable Application

In the following, Traceroute is employed as the vulnerable application. Traceroute is used to determine the routing path between a source and destination by sending a set of control packets to the destination with increasing time-to-live values. A typical use of traceroute involves providing the destination IP, whereas the application returns information on the route taken between source and destination.

Redhat 6.2 is shipped with Traceroute version 1.4a5, where this is susceptible to a local buffer overflow exploit that provides a local user with super-user access [17]. The attack exploits vulnerability in malloc chunk, and then uses a debugger to determine the correct return address to take control of the program.

When anomaly detectors such as Stide [2] are being developed in real-world conditions an acceptable anomaly rate should be established to minimize the false positive rate. Therefore the objective of the attacker is to reduce the anomaly rate below an acceptable limit. Although such acceptable limits vary between applications, it is reasonable to assume that it is non-zero (i.e. in practice, normal behavior model of the detector cannot cover all possible user scenarios, as training is only conducted over a subset of behaviors).

In previous work [1, 7-9], the only normal condition used in order to analyze the traceroute behavior was one normal use case scenario as follows:

Traceroute nis.nsf.net

In other words, in all of the previous work Stide was trained on only the above use case scenario. The argument was that if a mimicry attack could be developed for such a constrained case, then it was only natural to think that it could be developed for more general cases, i.e. Stide configured under a more representative set of more training use cases. In this work, we consider a different approach and developed 6 use cases (first 5 are all normal whereas the sixth denotes the original attack), Table I. In doing so, our objective is not to create an exhaustive list of normal use cases, but rather just to investigate how anomaly rates changes in a bigger training set as opposed to the one used in previous work.

Table II summarizes the significance of training Stide on each of the five 'use cases' denoting normal behavior, with testing performed over the remaining four normal use cases and one attack. The last column 'All 5' represents the case in which Stide training is conducted over all five use cases associated with normal behavior. The last row '6' represents the case in which original attack is tested on Stide trained with each of the five use cases indicated in the columns. All of the test results are given in terms of percentages where 0% indicates normal and 100% indicates attack behavior. Thus, the aim of the attacker is to raise as low an anomaly rate as possible, i.e. closer to 0%, to minimize the alarms.

TABLE I
USE CASES

Use Case	Command
1. Target Google	Traceroute google.com
2. Target FCS	Traceroute cs.dal.ca
3. Target a non existent host	Traceroute [randomcharacters].com
4. Target localhost	Traceroute localhost
5. Help screen	Traceroute –h
6. Original Attack	./traceroot 2 0x0804c7c4

TABLE II
ANOMALY RATE REPORTED BY STIDE UNDER DIFFERENT COMBINATIONS OF TEST AND TRAINING USE CASES

		Training Sets					
		1	2	3	4	5	All 5
Test Sets	1	**0.0%**	56.0%	78.1%	73.9%	94.3%	**0.0%**
	2	2.8%	**0.0%**	40.4%	31.3%	83.5%	**0.0%**
	3	7.4%	7.4%	**0.0%**	35.1%	71.6%	**0.0%**
	4	9.5%	7.3%	48.9%	**0.0%**	84.7%	**0.0%**
	5	15.8%	15.8%	15.8%	15.8%	**0.0%**	**0.0%**
	6	62.8%	62.8%	64.8%	66.1%	76.6%	**62.8%**

Naturally, anomaly rate is sensitive to a wide range of behavioral properties. For example in the case of host name (use case 2), Stide produces a 56% anomaly rate when it is trained on the 'google' trace (use case 1). On instances of Stide trained with a single trace file anomaly rates on normal use cases varies between 2.8% to 94.3%. Needless to say, training Stide over *all* the normal uses cases (normal behavior), resulted in a zero anomaly rate on normal use cases, see "All 5" column in Table II. The attack produces a 62.8% anomaly rate under this Stide model, where this also happens to be the joint minimum anomaly rate for the original attack. This establishes the minimum performance target for the mimicry Traceroute attack under the Stide detector. That is to say, any mimicry attack developed with an anomaly rate less than 62.8% will be considered as an improvement over the original attack. However, in order to be effective in practice we require anomaly rates of attacks to approach that of those returned by Stide under normal use cases. This precludes the use of thresholds to distinguish anomalous behaviours from attacks. Moreover, unlike previous work [1, 7-9], we also recognize that this anomaly rate should include actions of *both* the attack script and the execution of the shellcode after the script gains control of the traceroute application.

Table III details the occurrence of system calls in the normal data. It is apparent that 8 system calls can cover over 82% of the normal data set, thus the proposed scheme for automating mimicry attack generation will evolve system call sequences using the same system calls. We also observe that the application frequently executed memory allocation and I/O system calls where these are also appropriate for obfuscation of mimicry attacks.

TABLE III
SYSTEM CALL FREQUENCY OF THE TRACEROUTE APPLICATION

System Call	Occurrence	Percentage
gettimeofday	190	25.85%
write	123	16.73%
select	90	12.24%
sendto	90	12.24%
old_mmap	33	4.49%
close	30	4.08%
open	27	3.67%
read	24	3.27%
fstat	23	3.13%
munmap	15	2.04%
recvfrom	13	1.77%
socket	11	1.50%
fcntl	10	1.36%
mprotect	9	1.22%
connect	8	1.09%

C. Linear Genetic Programming

As indicated above, the process for designing mimicry attacks is automated using Genetic Programming (GP). GP differs from most machine learning methodologies in that a 'population' of candidate solutions are maintained concurrently throughout the search process. Each candidate solution, or individual, takes the form of a program. Programs are represented (in this case) as a sequence of system calls, where the set of permitted system calls is predefined by the user. The search process progresses through the iterative application of a selection operator, evaluation of performance associated with a subset of individuals, and application of search operators. Specifically, the selection operator in this work takes the form of a steady state tournament. This means that an even number of individuals (4 in this work) is selected from the population of individuals with uniform probability. Performance (fitness) of this subset of individuals is evaluated, ranking the individuals participating in the tournament relative to each other. Search operators are then applied to the better performing half of the tournament, resulting in children. The children overwrite the individuals of the worst half of the tournament, taking their place in the population. Such a scheme is inherently elitist with the best individuals always surviving.

The specific representation utilized in this work defines instructions as a 2-byte opcode with two operands (each 1-byte) i.e. all instructions have the same number of bytes. Table IV defines the instruction set architecture as per the earlier analysis of application behavior; thus the instruction set consists of the seven most frequently occurring system calls characterizing normal behavior. That is to say, of the eight most frequent cases, 'old_mmap' requires knowledge of valid memory address ranges, thus artificially increasing the complexity of designing valid exploits. Table V defines the supporting parameter types.

Individuals are defined using a fixed length format. Search operators take three forms: two point crossover, instruction mutation, and instruction swap, Table VI. Crossover is therefore constrained to exchanging an equal number of instructions (a page) between two individuals. The number of instructions per page is allowed to vary from 1 instruction to "max instructions per page" as the fitness function reaches a new plateau, as in the page-based GP framework [18]. Mutation selects a single instruction with uniform probability and replaces it with a different instruction from the instruction set, Table IV. The swap operator selects two instructions from the same individual with equal probability and interchanges their respective positions.

The details of the linear Genetic Programming methodology itself is not particularly important for this paper, however, previous work utilizing Grammatical Evolution (GE) indicated that the linear representation provides a more direct method for successfully evolving buffer overflow attacks [5].

TABLE IV
LINEAR GP INSTRUCTION SET

System Call	Parameter 1	Parameter 2
gettimeofday	N/A	N/A
select	N/A	N/A
sendto	N/A	N/A
open	FILE	N/A
close	FILE	N/A
read	FILE	ADDR
write	FILE	TEXT

TABLE V
INSTRUCTION PARAMETERIZATION

Parameter Type	Options
FILE	"/etc/passwd", "/tmp/dummy"
ADDR	Address of the 4 byte space allocated within the shellcode.
TEXT	"toor::0:0:root:/root:/bin/bash",[1] "Hello, World!"

TABLE VI
GENERIC LINEAR GP PARAMETERS

Parameter	Settings
Crossover	0.9
Mutation	0.5
Swap	0.5
Selection	Tournament of 4 individuals
Stop Criteria	At the end of 50,000 tournaments.
Population	500 individuals with different sizes.

D. Fitness Function

The original attack contains a standard shellcode, which uses the execve[2] system call to spawn a UNIX shell upon the successful execution. Since traceroute never uses an execve system call (as shown in Table III), the original attack can be easily detected, Table II. To this end, we employ a different attack strategy by eliminating the need to spawn a UNIX shell. Most programs typically perform I/O operations, in particular open, write to / read from and close files. Table III demonstrates that traceroute frequently uses open / write / close system calls. We therefore recognize that performing the following three steps mimics the behavior of the original attack:

1. Open the UNIX password file *(/etc/passwd*).
2. Write a line, which provides the attacker a super-user account that can login without a password.
3. Close the file.

Therefore, the objective of our GP based automatic attack development system is to discover a sequence of system calls that perform the above three steps in the correct order (i.e. the attack cannot write to a file that it has not opened) while minimizing the anomaly rate of Stide. Hence the fitness function has two objectives: evolving successful as well as undetectable attacks. In particular, the shellcode must contain the following sequence of 'core' components in order to conduct the attack:

1. Contain open ("*/etc/passwd*").
2. Contain write ("*toor::0:0:root:/root:/bin/bash*").
3. Contain close ("/etc/passwd").
4. Execute close after write and open before write
5. When the system call sequence is fed to Stide, anomaly rate should be as low as possible.

Two fitness functions are considered, incremental and concurrent. Both award a total of 5 'points' for establishing the above components of the 'core' attack and minimizing the anomaly rate. A perfect individual would therefore have a fitness of '10'.

- *Incremental Fitness Function* (Algorithm 1): Assumes a step-by-step approach to designing an attack, thus fitness is first awarded for steps associated with building a successful attacks, Steps (a) to (e). Only when all the components associated with designing the 'core' attack are present is the individual rewarded for minimizing the anomaly rate, Step (f).
- *Concurrent Fitness Function* (Algorithm 2): In this case both objectives are evaluated independently. That is to say, individuals are rewarded for minimizing the anomaly rate at the same time as being rewarded for building a successful attack.

Algorithm 1. Incremental fitness function

Fitness = 0
(a) IF the sequence contains *open* (*"/etc/passwd"*) THEN Fitness += 1
(b) IF the sequence contains *write* (*"toor::0:0:root:/root:/bin/bash"*) THEN Fitness += 1
(c) IF the sequence contains *close* (*"/etc/passwd"*) THEN Fitness += 1
(d) IF *open* precedes *write* THEN Fitness += 1
(e) IF *write* precedes *close* THEN Fitness += 1
(f) If (Fitness == 5) Fitness += (100 – *AnomRate*)/20

[1] This text creates a user "toor" with super-user privileges, who can connect remotely without supplying a password.
[2] Execve is a system call, which executes the program given as the first argument.

Algorithm 2. Concurrent fitness function

Fitness = 0

(a) IF the sequence contains *open* (*"/etc/passwd"*) THEN Fitness += 1

(b) IF the sequence contains *write* (*"toor::0:0:root:/root:/bin/bash"*) THEN Fitness += 1

(c) IF the sequence contains *close* (*"/etc/passwd"*) THEN Fitness += 1

(d) IF *open* precedes *write* THEN Fitness += 1

(e) IF *write* precedes *close* THEN Fitness += 1

(f) Fitness += $(100 - AnomRate)/20$

IV. Results

Previous research employing the same vulnerable application, *i.e.* Traceroute v1.4a5, configures Stide with only one normal use case (Section 3) [1, 7-9]. Thus, the performance of the detector when trained under different use cases was not reported. Furthermore, results in [1, 7-9] were not expressed in terms of anomaly rates. The approach taken in those work was to analyze the attack by determining whether an attack provides a match in the normal (configuration) database or not. On the other hand, in this work, we run each attack through the detector and obtain the corresponding anomaly rate. We believe this way of checking the validity of the attack is more representative of the behavior of an attacker given that it is more likely that the attacker will not be able to know all the details of the detector under attack. Thus, results given in Tables VII and VIII show the anomaly rates of the best attacks generated by different GP parameterizations.

The principle free parameters of GP take the form of the program length limit and population initialization. Results are therefore reported using best individual returned over 20 initializations per program length limit, where the process is repeated over each Fitness Function, Tables VII and VIII. Anomaly rate is obtained by testing the attack designed by GP against Stide, where this is equivalent to previous practice in mimicry attack evaluation [1, 7-9].

We also report *adjusted anomaly rate* where this includes activities targeted at obtaining the control of the program as well. What we mean here is that, since previous work only analyzed the attacked developed for a match or no match, they implicitly assume that the attacker has already gained control on the victim system. As such although the attack, they developed, does not raise an alarm (no match means zero anomaly rate), no consideration is given as to whether the combination of the attacked developed and the process of attacker gaining control will increase the anomaly rate. We call measuring the anomaly rate over the process of gaining control and the attack itself, as the "Adjusted Anomaly Rate"; whereas the anomaly rate associated with the exploit alone is denoted by "Anomaly Rate". We believe that the "Adjusted anomaly Rate" reflects the condition in practice more. As seen from the results, Tables VII and VIII, indeed the adjusted anomaly rate is greater than measuring/analyzing the anomaly rate just over the attack developed. This implies that while developing mimicry attacks automatically, the evasion system should consider the conditions under which an attacker gains control of the victim system.

Moreover, the results indicate that as the attack gets longer GP becomes increasingly effective at hiding the attack, resulting in an anomaly rate down to 1.69% (adjusted anomaly rate of 2.97%). The second most significant GP parameter appears to be the 'page size' where as long as the page size is six or more instructions; anomaly rates are lower than five per cent.

TABLE VII
CHARACTERISTICS OF BEST INDIVIDUALS UNDER THE INCREMENTAL FITNESS FUNCTION

Page Count × Page Size	Fitness	Anomaly Rate	Adjusted Anomaly Rate
40×6	9.92	1.69%	2.51%
20×12	9.92	1.69%	2.51%
20×6	9.83	3.45%	4.40%
10×12	9.83	3.45%	4.40%
40×3	9.74	5.17%	5.66%
20×3	9.73	5.36%	6.06%
10×6	9.64	7.14%	7.07%
5×12	9.64	7.14%	7.07%
80×3	9.32	13.56%	12.54%

TABLE VIII
CHARACTERISTICS OF BEST INDIVIDUALS UNDER THE CONCURRENT FITNESS FUNCTION

Page Count × Page Size	Fitness	Anomaly Rate	Adjusted Anomaly Rate
40×6	9.92	1.69%	2.51%
20×12	9.92	1.69%	2.51%
20×6	9.83	3.45%	4.40%
10×12	9.83	3.45%	4.40%
40×3	9.78	4.31%	5.03%
20×3	9.73	5.36%	6.06%
10×6	9.73	5.36%	6.06%
5×12	9.64	7.14%	7.07%
80×3	9.30	13.98%	12.90%

Figure 1 details a successful 240-system call attack with the minimum anomaly rate. System calls that are related to the attack are printed in bold. The block of 6 underlined system calls is repeated 38 times in the attack, hence is truncated in the figure for eligibility. The attack indicates that GP found a "blind spot" in Stide's normal behavior database. Specifically, GP discovered a sequence in the Stide database that both minimized the anomaly rate *and* represented a "NOP" sequence from the perspective of the attack. The sequence is naturally repeated through the action of the GP crossover operator under a (maximum) page size of six or more.

Given the significance of such repeating sequences, we also note that from the attacker's perspective, it might make

more sense to deploy such a strategy using a loop. Thus, rather than explicitly encoding 240 (480) system calls individually, an instruction loop would be utilized, reducing the code length by over 95%, and making the attack both difficult to detect and very concise.

open(1) gettimeofday() select()
write(1,r) write(2,r) gettimeofday() sendto() gettimeofday() select()
write(1,r) write(2,r) gettimeofday() sendto() gettimeofday() select()
write(1,r) **close(1)** open(2**) exit()**

Fig. 1. An attack of 240 system calls with the minimum anomaly rate (underlined block of 6 system calls is repeated 38 times in the attack).

V. Conclusion

In this work, we employed a mimicry attack approach to perform penetration testing on the well-known Stide host based anomaly detector. The mimicry attacks are evolved not manually but automatically, in this case using a linear genetic programming based approach. Our work differs in two ways from the pervious research in the area: (*i*) the anomaly rate is measured not only after the attacker gains control but over the entire attack, (*ii*) not only system call sequences but also all the associated parameters are evolved automatically.

A central theme in the approach is the utilization of GP to actually automate the process of malicious code design. To do so, a framework is utilized in which specific emphasis is placed on the: (*i*) Identification of an appropriate set of system calls from which attacks are built, in this case informed by the most frequently executed instructions from the vulnerable application. (*ii*) Identification of appropriate goals, where these take two basic forms, minimization of detector anomaly rate, whilst matching key steps in establishing the 'core' attack. (*iii*) Support for copying (duplicating) key instruction sequences once discovered, such that their utility may be investigated under different conditions by the learning algorithm. (*iv*) Support for obfuscation, where in this case this is a direct side effect of the stochastic search operators inherent in GP.

Results show that our approach can discover suitable rules for mimicry attacks where the anomaly rate is reduced to ~2.97% for the entire attack (and ~1.69% for the part after the attacker gains control). Moreover, in these experiments, the use of different fitness functions, the implications of the length of the system call sequences, and the impact of the page sizes to anomaly detection rates are investigated. In summary both fitness functions provide sequences with low anomaly rates but the concurrent fitness function provides sequences with lower anomaly rates. In general, increasing the system call length provides lower anomaly rates provided that crossover works on sequences of 6 or more instructions. This observation is related to the need to provide an efficient mechanism for duplicating 'NOP' sequences (from the attack behavior perspective). Such sequences, however, are identified within the context of conforming to normal behavior profiles as characterized by the Stide database.

Future work will consider attack obfuscation to generate variant buffer overflows for IDS blind spot penetration testing and the implementation of automatic buffer overflows for other well-known service such as SSH or FTP. Moreover, we anticipate being able to integrate the attack generation component into a co-evolutionary context. The resulting arms race between detectors and attacks will provide detectors that incorporate parameter analysis as well as being able to preempt 'unseen' attacks. That is to say, coevolution of attack-detector pairs will enable attacks previously unseen in the environment to be encountered and appropriate responses designed.

Acknowledgements

This work was supported in part by Killam, NSERC, MITACS and the CFI New Opportunities program. All research was conducted at the Dalhousie NIMS Laboratory, http://www.cs.dal.ca/projectx/.

References

1. C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, Automating mimicry attacks using static binary analysis, Proceedings of the USENIX Security Symposium, pp. 161-176, 2005.
2. Stide, http://www.cs.unm.edu/~immsec/data-sets.htm, Last accessed May 2006.
3. Vigna, G., Robertson, W., Balzarotti D., Testing Network Based Intrusion Detection Signatures Using Mutant Exploits, ACM Conference on Computer Security, pp. 21-30, 2004.
4. T. H. Ptacek and T. N. Newsham, Insertion, Evasion and Denial of Service: Eluding Network Intrusion Detection. Technical Report, Secure Networks, January 1998.
5. H. G. Kayacik, A. N. Zincir-Heywood, M. I. Heywood, Evolving Successful Stack Overflow Attacks for Vulnerability Testing, 21st Annual Computer Security Applications Conference, pp. 225-234, 2005.
6. H. G. Kayacik, M. I. Heywood, A. N. Zincir-Heywood, On Evolving Buffer Overflow Attacks using Genetic Programming. Proceedings of the Genetic and Evolutionary Computation Conference, (GECCO'06). SIG EVO, ACM Press. pp. 1667-1673, 2006.
7. D. Wagner and P. Soto, Mimicry attacks on host based intrusion detection systems, ACM Conference on Computer and Communications Security, pp. 255-264, 2002.
8. Tan, K. M. C., Killourhy, K. S., Maxion, R. A., Undermining an Anomaly-based Intrusion Detection System using Common Exploits, RAID'2002, LNCS 2516, pp 54-73, 2002.
9. K. M. C. Tan, J. McHugh, K. S. Killourhy, Hiding Intrusions: From the Abnormal to the Normal and Beyond, Symposium on Information Hiding, pp. 1-17, 2002.
10. D. Mutz, F. Valeur, G. Vigna, C. Kruegel, Anomalous System Call Detection, ACM Transactions on Information system and Security, 9(1), pp. 61-93, Feb 2006.
11. S. Forrest, S. A. Hofmeyr, A. Somayaji, and T.A. Longstaff. A sense of self for unix processes. In Proceedings of the IEEE Symposium on Security and Privacy, pp. 120--128, 1996,
12. R. Sekar, M. Bendre, D. Dhurjati & P. Bollineni, A Fast Automation-based Method for Detecting Anomalous Program Behavior, IEEE Symposium on Security and Privacy pp. 144-155, 2001.

13. H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong, Anomaly detection using call stack information, IEEE Symposium on Security and Privacy, pp. 62-75, 2003.
14. D. Wagner and D. Dean, Intrusion detection via static analysis, IEEE Symposium on Security and Privacy, pp. 156, 2001.
15. Wespi, A., Dacier, M., and Debar, H., Intrusion Detection Using Variable-Length Audit Trail Patterns, RAID'2000, LNCS 1907, pp. 110-129, 2000.
16. Banzhaf W., Nordin P., Keller R. E., Francone F. D., Genetic Programming: An Introduction. Morgan Kaufmann, 1998.
17. Securiteam, Linux Traceroute Exploit Code Released, http://www.securiteam.com/exploits/6A00A1F5QM.htm, Last accessed May 2006.
18. M. I. Heywood, A. N. Zincir-Heywood, Dynamic Page Based Crossover in Linear Genetic Programming, IEEE Transactions on Systems, Man and Cybernetics - Part B, 32(3), pp 360-388, 2002.