Evolving Buffer Overflow Attacks with Detector Feedback

Author identities withheld

No Institute Given

Abstract. A mimicry attack is an exploit in which basic behavioral objectives 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 detector weaknesses. In this work, we provide a process for evolving all components of a mimicry attack relative to the Stide (anomaly) detector under a Traceroute exploit. To do so, feedback from the detector is directly incorporated into the fitness function, thus guiding evolution towards potential blind spots in the detector. Results indicate that we are able to evolve mimicry attacks that reduce the detector anomaly rate from ~67% of the original core exploit, to less than 3%, effectively making the attack indistinguishable from normal behaviors.

1 Introduction

Our objective is to develop an automated process for building "white-hat" attackers within a mimicry context [1,2,3,4]. By 'mimicry' we assume the availability of the 'core' attack, where this establishes a series of behavioral objectives associated with the exploit [5,6]. 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 as possible. 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. By "white hat", we imply that the underlying objective is to use the attacks to improve the design of corresponding detectors.

Previous research has established the suitability of evolutionary computation as an appropriate process for automating the parameterization of buffer overflow attacks [5], and for designing a generic buffer overflow attack itself [6]. In this work, we extend the approach to explicitly incorporate feedback from the anomaly detector. Moreover, previous instances of evolved attacks were designed within the context of a virtual vulnerability and verified against the Snort (signature based) detector post training [5,6]. Conversely, this work provides attacks that are specific to a real application vulnerability under the more advanced behavioral anomaly detection paradigm. To do so, evolution is guided towards attacks that are able to make use of unforeseen weaknesses in the detector, thus providing the basis for improvements in detector design (vulnerability testing). Relative to earlier works on mimicry attack generation (against behavioral

anomaly detectors) [1,2,3,4], the adoption of an evolutionary approach to designing attacks demonstrates that it is no longer necessary to rely on privileged detector information to establish valid attacks. As such, we believe that the ensuing attacks are more reflective of vulnerabilities that are likely to be developed by a "would be" attacker in practice.

In the following we detail the process used to configure the anomaly detector, and characterize the vulnerable application, Section 2, before introducing the evolutionary mimicry attack framework, Section 3. Results are presented in Section 4, in which attacks are successfully designed with anomaly rates less than three percent; in effect making them indistinguishable from normal behavior. Moreover, specific recommendations are made regarding the construction of appropriate search operators. Conclusions are drawn in Section 5, with the case made for the coevolution of (white hat) attacks and detectors.

2 Detector and Vulnerable Application

The general goal of this work is to demonstrate that real (white hat) exploits may be developed under an evolutionary computation paradigm given a mimicry attack model. To be of relevance to vulnerability testing of a specific detector, behavioral goals of an exploit are augmented with feedback from the detector itself. Thus, for vulnerabilities to exist in the detector we aim to evolve programs that provide the desired exploit whilst minimizing the detector anomaly rate. Unlike previous work on mimicry attack generation, no inside knowledge is utilized in identifying weaknesses in the detector [1,2,3,4]. With these general guidelines in mind, we first provide the background to the detector on which vulnerability testing will be conducted, introduce the configuration process, and establish how a successful attack will be recognized.

2.1 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 raise alarms. In this work, Stide was used as the target anomaly detector [7]; where a wide range of related research performed in vulnerability or penetration testing has employed Stide on the basis of it's open source availability and behavioral approach to anomaly detection [1,2,3,4]. That is to say, rather than taking the 'signature' based approach to detection[1], the behavioral methodology develops a model for normal behavior for specific services using *a priori* supplied system

[1] Where evading signature based detectors using mimicry methods is already considered straightforward [1].

call(s). As such the ensuing detector does not require a direct match between modeled behavior and an attack for recognition to take place (as per a 'signature' based detector), but returns a percent anomaly rate. The user is then free to interpret the anomaly rate as representing an attack or not (typically by specifying a threshold). It is this anomaly rate that will be used to guide the evolutionary process towards any weaknesses in the detector. Section 2.2 details how Stide was configured within the context of the vulnerable application.

2.2 Vulnerable Application and Configuration of Stide

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 was 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 [8]; hereafter the 'core' attack. The attack exploits a vulnerability in malloc chunk, and then uses a debugger to determine the correct return address to take control of the program. As indicated above, an anomaly detector is used in conjunction with a threshold (anomaly rate) such that detection and false positive rates are optimized. The objective of the attacker is to build attacks that return anomaly rates below the threshold characterizing normal behavior. One approach to establishing a safe detector threshold might be to set the threshold to zero. However, this would result in far too many false positive alarms. That is to say, in practice, the normal behavior model of the detector cannot cover all possible user scenarios, as configuration is only conducted over a subset of behavioral traces. However, we also note that the available configurations of the Traceroute application is also limited; thus only a small number of use cases are sufficient to provide a complete set of system call sequences to characterize normal behavior[2]. In this we consider two scenarios. Scenario 1 configures Stide using a single use case, the 'nist' domain, as per previous research in mimicry research [1,2,3,4]. The principle motivation being that if attacks can be designed against a minimalist Stide configuration, then designing attacks for a typical configuration will be easier. Scenario 2 configures Stide using 5 use cases, as follows: search engines; local servers; a non existent host; the local host; and the application help screen. The motivation in this case being that an attacker would not have access to the database of normal behaviors that Stide uses to characterize normal behavior, an assumption made by all the aforementioned works on mimicry attack generation. Thus, establishing whether a detector can be defeated under a typical configuration and without access to the internal detector data structures is also of practical interest.

[2] Stide builds a behavioral model based on system call sequences alone; no use is made of arguments, thus avoiding any sensitivity to specific system call parameters [7].

3 Evolutionary Framework for Mimicry Attack Generation

The case for using evolutionary computation in a mimicry attack context has previously been made with respect to: the utility of code bloat for obfuscation of malicious code; freedom in defining fitness functions most appropriate to the application domain; and solutions taking the direct form of the attack itself [6]. However, the feasibility of evolving attacks was previously established in terms of a hypothetical application, and did not incorporate the detector in any way. As a consequence, vulnerability testing is only appropriate under a signature based detection paradigm.

Behavioral based anomaly detectors are configured with respect to specific applications. Thus, vulnerability testing is also carried out with respect to a specific (vulnerable) application. The first step of our framework for evolving attacks against such behavioral detectors is to identify an instruction set that is not likely to be immediately recognized as anomalous by the detector. Secondly, a fitness function needs crafting that focus on the relevant behavioral properties of the 'core' exploit (Traceroute in this case). Finally, we need to define the mechanism for integrating both the behavioral components and the detector feedback into an overall fitness function. Subsections 3.1 and 3.2 detail the framework used to address these points, with Subsection 3.3 summarizing the evolutionary model employed in this work.

3.1 Identifying Instruction Set

In order to minimize the likelihood of the exploit being detected, we restrict the instruction set from which attacks are evolved to those appearing in the target application (Traceroute). Table 1 details the frequency of the top twenty system calls executed by the Traceroute application. This accounts for over 90% of the normal instruction set. The system calls used to construct attacks will therefore consist of the top 15 from this list. Note that such an approach does not require any knowledge of the detector. Establishing the system calls associated with the application merely implies that a diagnostic tool[3] is deployed to identify those sent to the operating system by the vulnerable application during execution.

3.2 Fitness Function

The original attack contains a standard shellcode, which uses the execve system call to spawn a UNIX shell upon successful execution. Since traceroute never uses an execve system call (Table 1), the original attack can be easily detected. 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 1 demonstrates that traceroute

[3] In this case the Strace diagnostic tool is employed, http://sourceforge.net/projects/strace/

Table 1. Frequency of top 20 system calls.

System Call	Occurence	Frequency	System Call	Occurence	Frequency
gettimeofday	220	16.73%	mprotect	34	2.59%
write	142	10.8%	socket	29	2.21%
mmap	113	8.59%	recvfrom	28	2.13%
select	99	7.53%	brk	27	2.05%
sendto	99	7.53%	fcntl	26	1.98%
close	93	7.07%	connect	20	1.52%
open	86	6.54%	ioctl	15	1.14%
read	75	5.7%	uname	14	1.06%
fstat	73	5.55%	getpid	12	0.91%
munmap	49	3.73%	time	10	0.76%

frequently uses open / write / close system calls. We therefore recognize that performing the following three steps establish the goals of the original shell code 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.

The objective of the evolutionary search process 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 from 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 exploit:

1. Contain open ("/etc/passwd");
2. Contain write ("toor::0:0:root:/root:/bin/bash")[4];
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.

This leads to the final composition of the fitness function, Algorithm 1. A total of 5 'points' are awarded for establishing the above components of the 'core' attack. A further 5 'points' are awarded for minimizing the anomaly rate provided by the Stide detector. A perfect individual would therefore have a fitness of '10'.

[4] Creates a user 'toor' with super-user privileges, who can connect remotely without supplying a password.

Algorithm 1 Generic Fitness Function

1. Fitness = 0;
   (a) IF ($\{open('/etc/passwd')\} \in$ sequence)THEN (Fitness += 1);
   (b) IF ($\{write('toor :: 0 : 0 : root : /root : /bin/bash')\} \in$ sequence) THEN (Fitness += 1);
   (c) IF ($\{close('/etc/passwd')\} \in$ sequence) THEN (Fitness += 1);
   (d) IF ($'open'$ precedes $'write'$) THEN (Fitness += 1);
   (e) IF ($'write'$ precedes $'close'$) THEN (Fitness += 1);
   (f) $Fitness+ = \frac{100 - Anomaly\ Rate}{20}$

3.3 Evolutionary Model

Table 2 defines the instruction set architecture (and parameter types) as per the earlier analysis of application behavior; thus the instruction set consists of the fifteen most frequently occurring system calls characterizing normal behavior, Table 1. Individuals are defined using a fixed length format, with solutions taking the form of sequences of system calls. No registers are required to store state, thus, strictly speaking, this is a Genetic Algorithm as opposed to a (linear) Genetic Program.

Search operators are used independently (children result from any combination of the three operators) and take three forms: crossover, instruction mutation, and instruction swap, Table 3. Crossover takes the form of single point crossover, with the same crossover point utilized in both individuals. The swap operator selects two instructions from the same individual with equal probability and interchanges their respective positions; thus providing the basis to investigate different permutations of the same instructions. In the case of mutation, three forms are investigated.

- *Individual-wise mutation*: selects a single instruction with uniform probability and replaces it with a different instruction from the instruction set, again chosen with uniform probability.
- *Instruction-wise mutation*: tests each instruction independently for the application of the mutation operator. Following a positive test, the instruction is again replaced with another from the instruction set (uniform probability).
- *Greedy mutation*: the current best case individual is selected and the single best one instruction modification accepted. This implies that all 14 alternative instructions are evaluated at each instruction position. Given the computational cost of accessing such an operator the test is only applied every 1,000 tournaments.

In the case of the individual-wise and instruction-wise mutation operators, a linear annealing schedule is employed such that at the last tournament, the mutation probability is zero, decaying linearly with increasing tournament count. The basic motivation being to enable the crossover operator to investigate different contexts of population material as the tournaments advance.

The selection operator takes the form of a steady state tournament, thus the population is inherently elitist with the best individuals always surviving.

Table 2. Instruction Set

System Call	Parameter 1	Parameter 2
open	{"/etc/passwd", "/tmp/dummy"}	n/a
close	{"/etc/passwd", "/tmp/dummy"}	n/a
read	{"/etc/passwd", "/tmp/dummy"}	4 byte space address
write	{"/etc/passwd", "/tmp/dummy"}	{"toor::0:0:root:/root:/bin/bash", "Hello, world!"}
other	n/a	n/a

Table 3. Parameters for Evolutionary Search

Parameter	Value
Population	500
Crossover	0.9
Mutation (individual wise)	0.5 with linear decay
Mutation (instruction wise)	0.001 with linear decay
Greedy Mutation	Every 1 000 tournaments
Swap	0.5
Tournament Size	4
Stop Criterion	100 000 tournaments

4 Experiments

We begin by establishing the anomaly rate for the original 'core' attack [8] on the two use cases used to configure Stide. This defines the minimum performance for any attack we evolve. Both configurations return an anomaly rate of approximately 65%; thus in order for evolved mimicry attacks to represent an improvement over the original attack, they should return an anomaly rate significantly lower than this.

The principle evolutionary parameter of interest in this work is the significance of the mutation operators employed. To this end, three scenarios are considered of increasing complexity: Individual-wise mutation; Instruction-wise mutation; Instruction-wise mutation with greedy mutation. In all three cases both crossover and swap operators appear, Table 3. Figures 1 and 2 detail the corresponding percent anomaly rate over 10 runs for Stide configured under scenario 1 (a single use trace) and scenario 2 (five use traces) respectively. For completeness we also summarize the anomaly rate of best case attacks, Table 4.

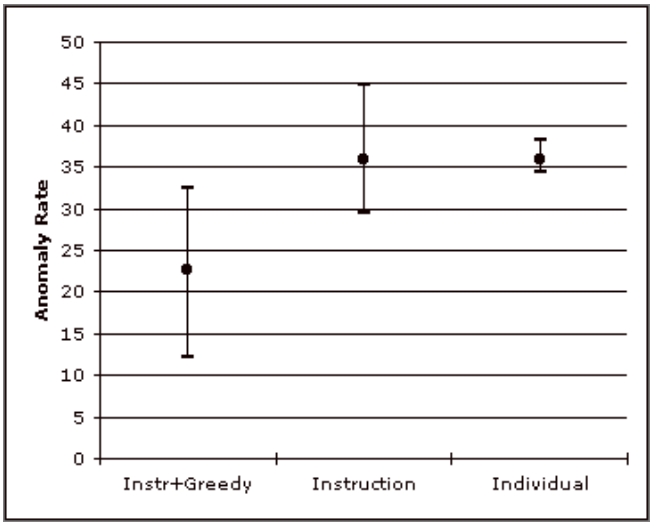

Fig. 1. Anomaly rate of attacks evolved against Stide configured over scenario 1 use case 'nist'.

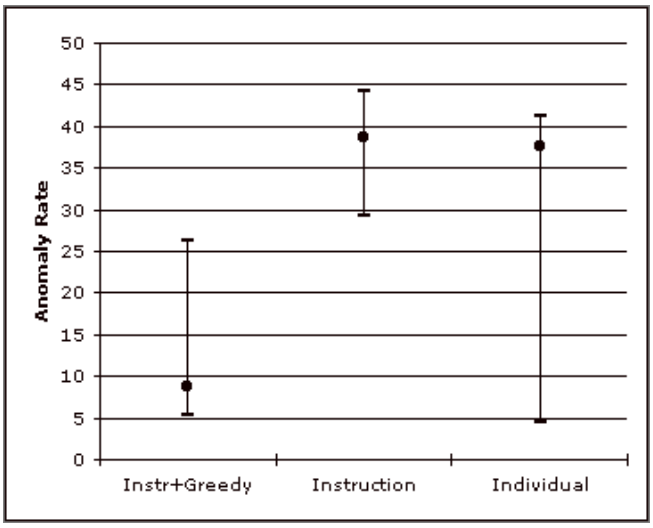

Fig. 2. Anomaly rate of attacks evolved against Stide configured over scenario 2 use case "all five".

It is immediately apparent that augmenting the search operators with increasingly sophisticated mutation operators results in a direct improvement to the median anomaly rate of the associated evolved exploits. Moreover, it is also apparent that although configuring Stide using a single use case represents a more difficult problem, all attacks returned a lower anomaly rate than the original core attack. The principle difference between the two use case scenarios appear to be manifest in the degree of variation in attack anomaly rates, with the more difficult scenario resulting in a lower variance in anomaly rates. However, there is very little variation in best case attack anomaly rates, with all search operator combinations returning attacks with anomaly rates lower than 4% under Stide configured with 5 use cases, and less than 6.5% anomaly rate under Stide configured using a single use case.

Table 4. Percent anomaly rate of best case attacks evolved.

Stide Configuration	Instruction-wise and Greedy Mutation	Instruction-wise Mutation	Individual-wise Mutation
all 5 use cases	2.11%	2.97%	3.81%
single 'nist' use case	6.36%	2.97%	5.08%

5 Conclusion

In this work, we developed an evolutionary mimicry attack approach to perform vulnerability testing on the well known Stide host based anomaly detector whilst treating the detector as a black box. That is to say, unlike previous approaches to mimicry attack generation, information from the detector is limited to that available to a "would be" attacker. Specifically, no use is made of privileged data structures internal to the detector. This means that the only feedback employed from the detector during the evolution of attacks is the detector anomaly rate, where this constitutes open information available to users as part of detector deployment. Conversely, previous approaches to mimicry attack generation have concentrated on reverse engineering the normal behavior database from the detector using an exhaustive search [1,2,3,4]. Such an approach would not be feasible without access to privileged information.

A central theme in the approach is the utilization of a Genetic Algorithm 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 exploits 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' exploit. (iii) Support for obfuscation, where in this case this is a direct side effect of the stochastic search operators inherent in an evolutionary search. (iv)

Search operators benefit from instruction wise mutation and an anealing scheme. Inclusion of a greedy instruction wise mutation operator is also beneficial, but expensive computationally on account of the number of fitness evaluations necessary to resolve a single application of the operator.

Future work will investigate the optimization of search operators for identifying detector blind spots more effectively than is currently the case. Moreover, we are interested in integrating attack evolution into a co-evolutionary context. That is to say, coevolution of attack-detector pairs will enable attacks previously unseen in the environment to be encountered and appropriate responses evolved on a continuous basis. A pre-requisite for such a system, however, requires the development of an evolutionary detection paradigm based on one-class training. Specifically, in order to avoid the issue of finding an appropriate characterization of normal behavior (an exceptionally difficult task, that typically results in system specific solutions) we recommend the utilization of classifiers trained on attack data alone. Such a class of classifier has been demonstrated using SVMs, but is still outstanding within an evolutionary computation context.

Acknowledgments

Acknowledgments to appear here.

References

1. D. Wagner and P. Soto, Mimicry attacks on host based intrusion detection systems, ACM Conference on Computer and Communications Security, pp. 255-264, 2002.
2. 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.
3. 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. 717-738, 2005.
4. K. M. C. Tan, John McHugh, Kevin S. Killourhy, Hiding Intrusions: From the Abnormal to the Normal and Beyond, Symposium on Information Hiding, pp. 1-17, 2002.
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, SIGEVO, Volume 2, ACM Press, 1667-1673, July 8-12, 2006.
7. University of New Mexico, Computer Science Department, Computer Immune Systems Data Sets and Software, http://www.cs.unm.edu/~immsec/data-sets.htm, Last accessed May 2006.
8. Securiteam Web Site, Linux Traceroute Exploit Code Released (GDB), Oct 2002, http://www.securiteam.com/exploits/6A00A1F5QM.html, Last accessed May 2006.