Evolving Successful Stack Overflow Attacks for Vulnerability Testing

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

Dalhousie University, Faculty of Computer Science,
6050 University Avenue, Halifax, Nova Scotia. B3H 1W5
{kayacik,zincir, mheywood}@cs.dal.ca

Abstract

The work presented in this paper is intended to test crucial system services against stack overflow vulnerabilities. The focus of the test is the user-accessible variables, that is to say, the inputs from the user as specified at the command line or in a configuration file. The tester is defined as a process for automatically generating a wide variety of user-accessible variables that result in malicious buffers (an exploit). In this work, the search for successful exploits is formulated as an optimization problem and solved using evolutionary computation. Moreover the resulting attacks are passed through the Snort misuse detection system to observe the detection (or not) of each exploit.

1 Introduction

Buffer overflow attacks aim to alter the execution of a vulnerable program by copying data to a variable in such a way that the original storage capacity is exceeded [7]. This may cause excess data to spill over the unallocated address space and overwrite the pointer to the next instruction after the function call. However, in order to deploy the attack successfully, execution must be accurately diverted to attacker's arbitrary code. To do so, the attacker might develop a program, which can assemble the different components of the malicious buffer. Moreover, because the location of the vulnerable program in address space is determined at runtime, certain characteristics of the malicious buffer should be approximated in the code.

In this work we propose the utilization of Evolutionary Computation (EC) [8] to discover the characteristics of the malicious buffer with the objective of identifying a wide range of successful attacks. The population represents the set of candidate exploits. The EC approach only requires the address of the user-accessible variable being tested and the outcome of the attack, both of which can be determined by a debugger or an executable code analyzer. Evolution is guided by the definition of a suitably informative fitness (cost) function that determines the quality of a malicious buffer exploit. Attack diversity is maintained by modifying the fitness function to incorporate fitness sharing, which discounts fitness based on the degree of similarity between individuals (exploits). The fitness function also represents the principle mechanism for incorporating *a priori* knowledge. In this case, minimizing the NoOP sled is known to improve the chances of avoiding detection. Incorporating this bias into the fitness function, and testing the resulting exploits on a misuse detection system, Snort, indicated that the resulting attacks were more effective at avoiding detection.

The principal objective of evolving overflow attacks is to assess critical applications against buffer overflow vulnerabilities. Recent work on vulnerability testing indicated that intrusion detection systems can detect a particular instance of an attack, but are unable to 'generalize' to the class of overflow attacks [1, 3, 4, 13, 15, 17, 18, 19]. The main contribution of this work is to automate the generation of successful malicious buffers. We consider this as a part of a wider 'white hat' framework where a co-evolutionary scheme is used to instigate an 'arms race' between exploits and detector. That is to say, detectors are only as good as our ability to provide signatures for new exploits. A framework based on co-evolution provides the basis discovering generic detectors for classes of attack by providing a sufficiently wide range of exploit behavior.

The remainder of the paper is organized as follows. Section 2 describes the buffer overflow concepts. Methodology is discussed in Section 3. Experimental results are presented in Section 4 and conclusions are drawn in Section 5.

2 Buffer overflow attacks

In programming languages such as C, data integrity checks are minimal for performance reasons; therefore the programmer is responsible for making sure that the memory allocated for a variable is sufficient. If these checks are omitted, a buffer overflow occurs, thus the contents spill over to other variables and overwrite the unallocated memory addresses. When a function is called within a program, the program uses a stack segment, which has a first-in-last-out structure, to push function variables. Moreover, a stack is used to remember which instruction is executed following the function call. This is called the return address and is stored at the bottom of the stack. Following the execution of a function, variables are popped from the stack and the return address is used to fetch the next instruction.

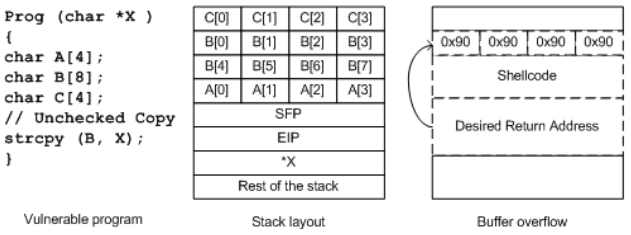

Figure 1. Example of a stack buffer overflow

Figure 1 shows three components: (i) A simple vulnerable program that omits the data integrity check; (ii) the stack layout after this program is started; and (iii) the malicious buffer overflow, which overwrites the return address with the attacker's desired return address. The size of 'char' is assumed 1 byte and the size of 'long' is assumed 4 bytes.

If the buffer overflow in the stack overwrites the return address, the execution of a program can be diverted to any arbitrary code. This is particularly dangerous if the program runs with super user privileges. In order to deploy a successful overflow, the attacker should create a malicious buffer, which (1) contains a shellcode and (2) overwrites the return address and gains control. Shellcode provides the assembly of instructions that spawn a root shell or adds a root privileged user.

Since the address of the unchecked variable is determined at runtime, estimating the address of the first instruction in shellcode is crucial (i.e. jumping elsewhere in the shellcode will have an undetermined outcome). To increase the chance of success of the malicious buffer, two supplementary components are added to the shellcode. First, the end of the shellcode is flooded with the desired return address. Attackers can approximate the desired return address by determining the address of the current stack pointer (ESP) and appending a suitable offset. Since the desired return address is an approximation and it is important to jump to the first instruction of the shellcode, the head of the malicious buffer is filled with a special purpose instruction called 'no operation' or NoOP, which is used to intentionally waste computational cycles. Sequences of NoOP instructions are referred to as the NoOP sled. As long as the desired return address is accurate enough to direct execution to the NoOP sled or the first instruction of the shellcode, the attack is successful. However, the NoOP sled, which usually manifests itself as a long sequence of 0x90 bytes, presents a very obvious detection signature. Therefore from attacker's point of view, shorter NoOP sleds are desirable, where this implies that the original stack pointer offset must be estimated more accurately.

3 Methodology

Our objective is to evolve programs that can craft buffer overflows to automate vulnerability testing. To do so, we employ Grammatical Evolution to discover the characteristics of a successful buffer overflow. Moreover, we utilize fitness sharing to encourage the evolution of different malicious buffers. For the purposes of this work, a simple (generic) vulnerable application was developed, which performs a data copy without checking the internal buffer size. Resulting attacks are passed through the Snort[1] intrusion detection system to assess the detection rate.

3.1 Grammatical evolution

Evolutionary Computation, and Genetic Programming (GP) in particular [9], provide several properties of utility to this work. The GP representation takes the form of a computer program, thus naturally fitting the objective of designing alternative malicious code. Performance is quantified using a fitness function, where there are no smoothness constraints on the form that such a function should take (unlike neural networks, where the cost function must typically be differentiable). As with other forms of Evolutionary Computation, GP is based on a 'population' of candidate solutions. In order to guide the 'evolution' of such a population, selection and search operators are based on the concepts of natural selection and genetics respectively. Specifically, a selection operator is used to define which individuals get to survive and reproduce, whereas the search operators define how new individuals (children) are introduced into the population. In this case, two parents are selected with uniform probability from the original population. Search operators are then stochastically applied to the parents, thus creating two children. The fitness (performance) of the parents and children is then compared using the *a priori* defined fitness function (sections 3.1.1 and 3.1.2). If the children perform better than the parents, they replace the parents. Otherwise, the parents are retained. Such a scheme results in a stochastic hill-climbing algorithm, where this forms the basis for the deterministic crowding method for multimodal optimization [11].

Where Grammatical Evolution (GE) [12] differs from GP is in the representation. Specifically, GE utilizes a separate genotypic and phenotypic representation. The genotypic representation is translated into the phenotypic form using a Context Free Grammar (CFG), typically of a Backus-Naur Form. The use of a CFG enforces the typing rules – syntax and semantics of the language – irrespective of the changes made by the search operators [12]. Unlike other structures typically employed to evolve computer programs, such as Tree or Linearly structured Genetic Programming; support for multi-typed languages is now straightforward. Moreover, we will also be able to make use of recent advances that combine GE with fitness

[1] A widely used open source intrusion detection system, Section 3.3.

sharing [11] to provide multiple solutions from the same population.

In this case a simple C grammar was developed for generating programs that assemble the malicious buffer exploits. The resulting C program is an individual, which approximates the desired return address and assembles the malicious buffer exploit. Each individual of the population represents a buffer overflow attack. The grammar specifies the offset, size of the NoOP sled, and the number of desired return addresses; hence GE alters these parameters to generate a wide variety of malicious buffers. The exploit contains a NoOP sled, a 46-byte shellcode that spawns a UNIX shell, and back-to-back desired return addresses. The first set of experiments with a basic GE (detailed in Section 4) showed that the population converges to one type of solution. In our second and third set of experiments, niching based on fitness sharing [2, 11] was used to encourage population diversity, that is, multiple types of attack. Thus, a fit individual can get a low 'shared' fitness, if many individuals find a similar solution. Fitness sharing is implemented to pressure individuals into utilizing different NoOP sled sizes and return addresses. The generic parameters employed in training of the individuals are summarized in Table 1.

Table 1. Training parameters

Parameter	Setting
Number of individuals	200
Number of generations	500
Probability of mutation	0.0
Probability of crossover	0.9
Replacement Strategy	Children replace the parents if their fitness is better
Number of niches	5
Training Time	Approximately 7 hours

3.1.1 Fitness function. The fitness function is used to express several characteristics that are related to achieving the overall objective. That is to say, basing the fitness function on a binary criteria – such as, does this individual successfully gain super user status – would not provide a sufficiently informative function space for the efficient evolution of exploits. In this work, six characteristics of a malicious buffer are utilized:

Existence of the shellcode ($\mu_{shellcode}$): A binary flag declaring whether the (root shell access) shellcode is inserted into the malicious buffer successfully. Thus, even if the overflow is successful, without the shellcode, the attack cannot succeed.

Success of the attack ($\mu_{success}$): The reaction of the application, i.e. a binary flag indicating whether the root shell was obtained.

NoOP Sled Score: Based on the ratio of NoOP instructions to the overall size of NoOP sled prior the shellcode. If the execution jumps into the NoOP sled, any non-NoOP instruction in NoOP sled can have undesirable effects on the succeeding shellcode. The NoOP sled score is formulated as:

$$1 - \frac{\#\ of\ non\text{-}NoOP\ instructions}{NoOP\ sled\ length}$$

Back-to-back Desired Return Addresses Score: Similar to NoOP sled score; it is based on the ratio of correct desired return addresses to the total number of 4-byte return addresses following the shellcode. If the stack pointer is overwritten with a faulty desired return address, execution will not jump to the shellcode. Score can be calculated with:

$$1 - \frac{\#\ of\ faulty\ return\ addresses}{Total\ \#\ of\ return\ addresses}$$

Desired return address accuracy: The difference between the desired return address and the actual address of the variable. Small difference indicates that the approximation is accurate. Accuracy is formulated as:

$$\frac{1}{\left| address_{actual} - address_{desired} \right| + 1}$$

Score calculated on NoOP sled size: We consider this is the easiest characteristic of a buffer attack to detect, thus minimizing the size of the NoOP sled is considered to improve the chances of not being detected. This is implemented in the third set of experiments. Score on NoOP sled length is expressed as:

$$\frac{1}{1 + NoOP\ sled\ length}$$

The last 4 characteristics are incorporated into the fitness function with their respective weights. In our experiments, the weights are all equal and detailed in Table 2.

The fitness function provides the basis for directing the search for solutions. In this work the view is taken that a hierarchy of objectives exists. Thus, if the malicious buffer does not contain the shellcode (i.e. $\mu_{\text{shellcode}} = 0$), the individual is assigned the minimum fitness. If the attack is successful (i.e. $\mu_{\text{success}} = 1$), the individual is assigned fitness between 100 and 120 based on the size of its NoOP sled. The perfect individual should be successful with a small NoOP sled, or no NoOP sled at all. If the attack is not successful, it is assigned a fitness based on the error rate of the NoOP sled, desired return addresses and the accuracy of the approximation. The overall fitness function incorporating these properties (with NoOP sled minimization) has the following form:

$$fitness = \mu_{shellcode} \times \begin{pmatrix} \mu_{success} \times (100 + W_{NS} \times score(NoOP)) + \\ (1 - \mu_{success}) \times \begin{pmatrix} W_{NE} \times score(NoOPError) + \\ W_{RE} \times score(retError) + \\ W_{DA} \times score(dist) \end{pmatrix} \end{pmatrix}$$

Table 2. Weights of the four characteristics of a malicious buffer

Weight	Value
Error on NOOP Sled (W_{NE})	20
Error on desired return addresses (W_{RE})	20
Desired return address accuracy (W_{DA})	20
NOOP sled size score (W_{NS})	20

3.1.2 Fitness sharing. Although EC is a population based search algorithm, schema theory indicates that as the fitter individuals reproduce, the population diversity decreases, resulting in the population converging on a small region of the search space [10]. In order to encourage the same population to provide multiple unique solutions, we borrow the concept of Fitness Sharing from Genetic Algorithms. In this case the fitness of an individual is discounted in proportion to the similarity with others in the population. Such a scheme is introduced to encourage solutions with different NoOP sled lengths and number of desired return addresses. That is to say, given a successful attack, fitness sharing encourages the identification of additional variants of the same attack.

Shared fitness for an individual *i* is calculated based on the raw fitness of the individual divided by the niche count m_i:

$$f_{shared} = \frac{f_{raw}}{m_i}$$

where niche count, m_i, increases as the similarity of an individual to other individuals increases, and is calculated over the population of *N* individuals:

$$m_i = \sum_{j=1}^{N} sh(d_{i,j})$$

$d_{i,j}$ denotes the Euclidean distance between individual *i* and *j*, which is calculated on two dimensions formed by NoOP sled length and the number of desired return addresses:

$$d_{i,j} = \sqrt{\left(NoOP_i - NoOP_j\right)^2 + \left(ret_i - ret_j\right)^2}$$

If distance *d* is smaller than the determined radius σ, sharing function *sh(d)* returns a value between 0 and 1, which increases as the distance decreases. Hence the sharing function can be expressed as:

$$sh(d) = \begin{cases} 1 - \dfrac{d}{\sigma} & d < \sigma \\ 0 & otherwise \end{cases}$$

σ is estimated from the population by determining the current extremes. This takes the form of the minimum and maximum values of the NoOP sled length and number of desired return addresses. Given the number of niches *q*, σ can be calculated as follows [2]:

$$\sigma = \frac{\sqrt{\left(\max(NoOP) - \min(NoOP)\right)^2 + \left(\max(ret) - \min(ret)\right)^2}}{2\sqrt{q}}$$

As the population evolves, the boundaries formed by NoOP sled lengths and number of desired return addresses will also change. Since determining the boundaries requires a pass of the entire population, σ is calculated every 5 generations in our experiments.

3.2 The vulnerable application

Similar to the example given by Erickson [6], we developed a basic vulnerable application, but this time using four 500-byte arrays. Erickson [6] only employed one 500-byte array whereas our preliminary experiments with that application indicated that the NoOP sled is frequently too small to raise any alarms (detailed in Section 4). The vulnerable program has *setuid* bit enabled and runs with root privileges. It copies the first command line argument to the fourth array without checking the size. This means, a successful attack should deploy a malicious buffer that is long enough to overwrite the return address after exceeding 2,000 bytes.

3.3 Intrusion detection system

After initial experiments, the aforementioned bias towards a minimal NoOP sled was utilized to improve the chances of avoiding detection. To validate the enhancement, the attacks were passed through a misuse detection system, Snort, to observe the detection (or lack of detection) of the malicious buffers. Snort is one of the best-known lightweight IDSs, which attempts to balance (detection) performance, flexibility and simplicity. It represents a widely used open-source intrusion detection system, able to detect various attacks and probes including instances of buffer overflows, denial of service attacks and stealth port scans [14]. Snort 2.3.2 (build 12) was installed and patched with the latest signatures (Mar 9, 2005) from the Snort web site [16]. Since we are interested in the detection of shellcode attacks, all signatures are disabled except the shellcode signatures. There are 21 shellcode signatures, which mainly detect different encodings of NoOP instructions as well as other well-known instructions such as setting user or group ID to root. Other than the signature reduction, Snort is employed with default parameters.

For the IDS to detect an attack, the attack in question should be manifested in the event stream that the IDS monitors. Since Snort is a network based IDS, this means the shellcode should appear in the network traffic. To

make the shellcode apparent in the Snort event stream (i.e. the network traffic), the vulnerable application is altered to print the contents of a variable. To achieve this situation, the attacker connects to the target host via telnet and dispatches the malicious buffer. We assume that he/she has no way of suppressing the variable dump, which triggers the Snort signatures. Given the use of encrypted protocols such as SSH, we note that the shellcode may not always appear in the network traffic. However, our objective in employing Snort is not to observe the detection of the shellcode by a network based IDS, by itself. Instead, our objective is to determine the detection of the attack with a misuse detection system (especially since the NoOP sled length is being minimized) and Snort is one of the most widely used misuse detection systems. After each attack is deployed, the Snort log files are checked to determine how many alarms were raised. From the attacker's point of view, between two successful attacks, the one that raises fewer alarms is favored. A similar evaluation methodology was employed to test the detection capabilities of IDSs on service vulnerability attacks in Vigna *et al.* [18].

4 Results

In the initial set of experiments, fitness sharing is not utilized, the second set of experiments utilizes fitness sharing, whereas the third set of experiments incorporates the bias to encourage smaller NoOP sleds.

The results are expressed in terms of fitness of the individuals (attacks) and the number of alerts that Snort generates when they were executed. Moreover, we are naturally interested in identifying whether a subset of attack properties is more correlated with evolving successful buffer overflow attacks than others. To do so, three characteristics of a malicious buffer are observed: the NoOP sled size, the number of desired return addresses, and an assessment of buffer overflow. For the latter, four types of buffer overflow are considered:

- *Invalid Buffer*: The buffer does not contain the shellcode, hence has zero chance of success.
- *Valid Buffer*: The buffer has NoOP sled, shellcode and desired return addresses present.
- *Viable*: Over 10 trials, the buffer deploys successfully, obtaining a root shell.
- *Undetectable*: In addition to its success, Snort raises no alarms during its execution.

Table 3 details the assessment of buffer overflows for different experiments. In all three experiments, the C grammar (forms the program for assembling the malicious buffer) ensures that majority of the population is at least valid. Although niching reduces the number of viable buffers, it also encourages diversity in the population, which will be discussed later in this section.

Table 3. Malicious buffer types and counts for three experiments

	No Niching	Niching	Niching & NoOP min
Invalid	2	6	2
Valid	0	118	111
Viable	146	54	57
Undetectable	52	22	30

Figure 2 summarizes the population from the three experiments with NoOP size and number of desired return addresses plotted with fitness. As mentioned above, the vulnerable variable is approximately 2000 bytes away from the return address (EIP). Experiments with basic GE provided attacks that resulted in a range of return addresses. Introduction of the sharing function increased the diversity of all three parameters: fitness, NoOP size, and return address. This indicates that the attacks learned to overwrite the EIP with an approximated return address. In the third set of experiments, two attacks stand out from the rest of the population with a fitness value of 110. These attacks successfully deployed, whilst using a single one NoOP instruction, making them very difficult to detect using signatures targeting the NoOP code. Moreover, this was achieved without compromising the success of the attack itself.

Figure 3 shows the NoOP sled size and the accuracy of the desired return address. In all three experiments, NoOP sled size has a linear relation with the accuracy of the desired return address, i.e., the population appears on the far side of Figure 3. That is to say, as the accuracy gets better, the NoOP sled size gets smaller. Moreover in case of successful attacks, it is observed that NoOP sled size is always kept below 2,000 bytes.

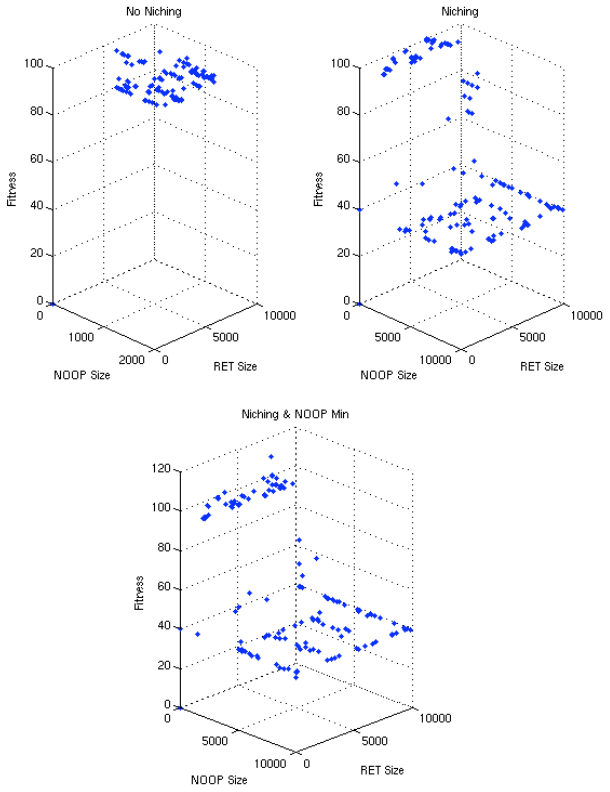

Figure 2. Fitness, NoOP sled size and the desired return address size of the population in the last generation

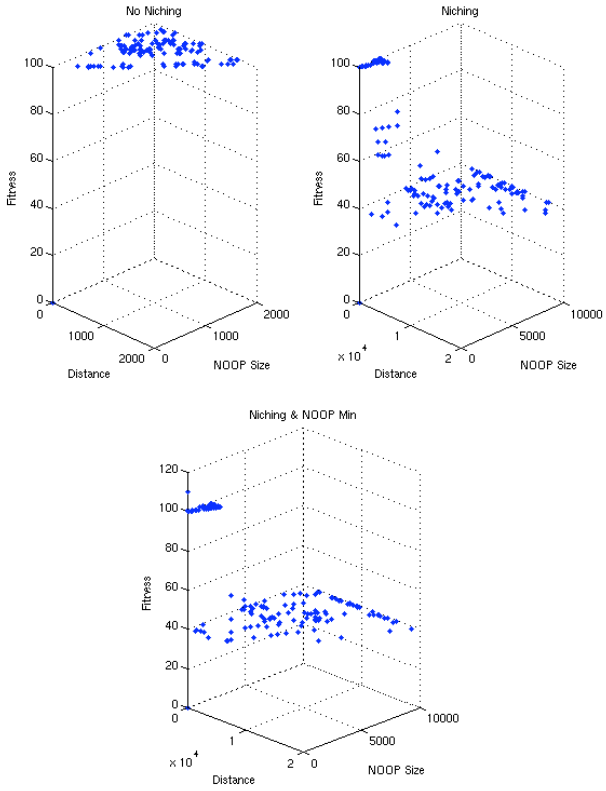

Figure 3. Fitness, NoOP sled size and the accuracy of the desired return address of the population in the last generation

Figure 4 details the mean fitness of the population over 500 generations. In all three experiments, populations converged to a solution after approximately 100 generations. In the niching experiments, mean fitness of the population is lower because attacks that generate valid buffers while maintaining diversity have a shared fitness comparable to the shared fitness of the attacks that generate viable buffers with similar parameters.

Figure 5 shows the change of NoOP sled length over generations. As indicated before, in all three experiments NoOP sizes are reduced below 200 to deploy viable buffer overflows. In Figure 4, we demonstrated that the population converges after 100 generations. In the experiments without NoOP minimization, after a few hundred generations, the mean NoOP sled length stops decreasing; whereas in the NoOP minimization experiments the fitness function continues to minimize NoOP sled length even if the buffer overflow deploys successfully.

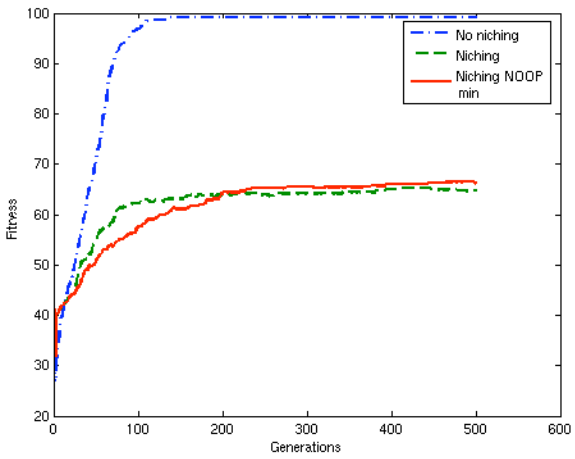

Figure 4. Mean raw fitness of the population over 500 generations

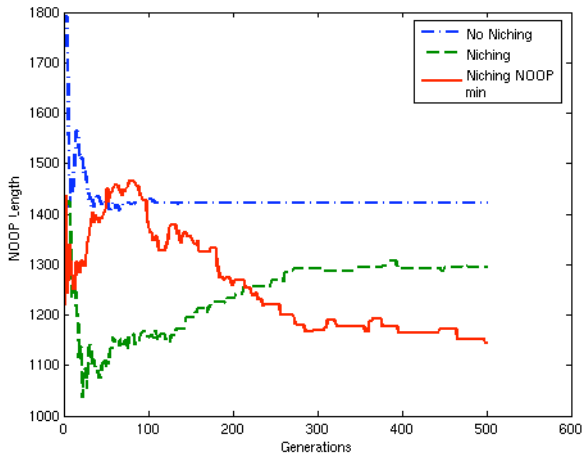

Figure 5. Mean NoOP size of the viable and undetected attacks over 500 generations

In Figure 6, buffer overflows are plotted with NoOP sled sizes, generated alerts and the fitness. Since population without niching converged with less diversity, the number of alerts is 0, 1 or 2. In case of niching, alert count ranges between 0 and 10 (greater diversity in NoOP sled length). Signature analysis showed that the Snort NoOP signature (shown below), which monitors the existence of large blocks of 0x90, triggered all alerts.

alert ip $EXTERNAL_NET $SHELLCODE_PORTS ->$HOME_NET any (msg:"SHELLCODE x86 NOOP"; content:"\|90 90 90 90 90 90 90 90 90 90 90 90\|"; depth:128; reference:arachnids,181; classtype:shellcode-detect; sid:648; rev:7;)

Figure 7 details the average alert count for viable attacks. Table 3 showed that the basic GE managed to produce the most undetectable attacks. However, it is also apparent that in terms of the average alert count of the population, niching with NoOP minimization produced the least alerts. Moreover niching with NoOP minimization resulted in two attacks with only one NoOP instruction each, effectively undetectable.

5 Conclusion

Grammatical Evolution was investigated within the context of vulnerability testing. Specifically, the evolved C program performs three tasks (1) approximating the address of the vulnerable variable, (2) determining the length of the NoOP sled and the number of desired return addresses (3) assembling the malicious buffer in the light of the characteristics established in first two tasks. As indicated before, between two attacks that deploy successfully, the one that raises fewer alarms is preferred. Hence, every 100 generations, the population is tested against Snort to determine the detection (or lack of detection) of the attacks. Results indicated that Snort detects buffer overflow attacks based only on the NoOP sled size.

Three sets of experiments were performed, namely the basic GE, GE with niching which encourages a population to maintain diversity, and GE with niching and NoOP minimization since longer NoOP sleds are easily detected. Although all three experiments produced comparable results, basic GE produced the best mean fitness and the most viable attacks. On the other hand niching produced programs that can craft a malicious buffer with different NoOP sled sizes and number of desired return addresses. Furthermore, NoOP minimization produced smaller mean NoOP sled lengths and fewer alerts per population, which are desirable from an attacker's point of view. Results also showed that in order an attack to be successful, the return address (EIP) should be overwritten with an accurate desired return address that directs the execution to a point in the NoOP sled or the first instruction of the shellcode. NoOP sled length decreases as the accuracy of the desired return address increases.

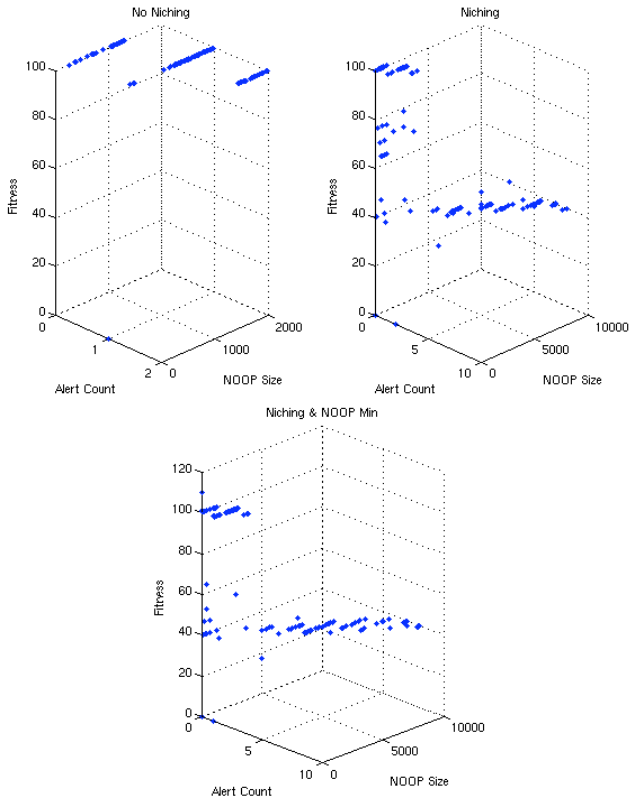

Figure 6. Fitness, NoOP size and alert counts of the population in the last generation

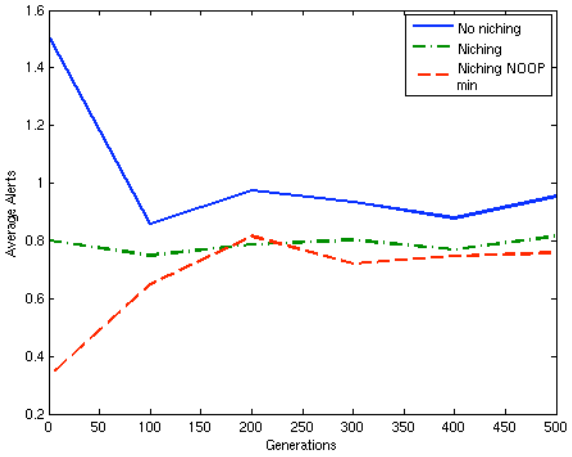

Figure 7. Average alert count of viable and undetectable attacks for three sets of experiments

Future work will mainly focus on attack obfuscation to generate variant buffer overflows for IDS blind spot testing and the implementation of buffer overflows for a 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 exploits will provide generic detectors that do not rely on third parties first labeling previously unseen exploits.

Acknowledgments

This work was supported in part by Discovery grants from the Natural Sciences and Engineering Research Council of Canada, and the CFI New Opportunities program. All research was conducted at the NIMS Laboratory, http://www.cs.dal.ca/projectx/.

Bibliography

[1] Christodorescu M., Jha S., "Static analysis of executables to detect malicious patterns", Proceedings of the USENIX Security Symposium, 2003.

[2] Deb K., Goldberg D. E., "An Investigation of Niche and Species Formation in Genetic Function Optimization" Proceedings of the third international conference on Genetic algorithms, pp 42 - 50, 1-55860-006-3, 1989.

[3] Detristan T., Ulenspiegel T., Malcom Y., Underduk M. S., "Polymorphic shellcode engine using spectrum analysis", Phrack Online Magazine, 61, 2003.

[4] Dozier, G., Brown, D., Cain, K., Hurley, J., "Vulnerability analysis of immunity-based intrusion detection systems using evolutionary hackers," Proceedings of the Genetic and Evolutionary Computation Conference, Lecture Notes in Computer Science, LNCS 3102, pp 263-274, 2004.

[5] Eiben A.E., Smith J.E., "Introduction to Evolutionary Computing", Springer, ISBN 3-540-40184-9, 2003.

[6] Erickson J., "Hacking: The Art of Exploitation", No Starch Press, ISBN 1-59327-007-0, Ch. 2, 2003.

[7] Foster J.C., Osipov V., Bhalla N., Heinen N., "Buffer Overflow Attacks: Detect, Exploit, Prevent", Syngress Publishing, ISBN 1-932266-67-4, Ch.5, 2005.

[8] Goldberg D.E., Deb K., "A Comparative Analysis of Selection Schemes Used in Genetic Algorithms," in Foundations of Genetic Algorithms, G.J.E. Rawlins (ed.), Morgan Kaufmann, ISBN 1-55860-170-8, 1991.

[9] Koza J.R., "Genetic Programming", MIT Press, 1992.

[10] Langdon W.B., Poli R., "Foundations of Genetic Programming", Springer-Verlag, IBSN 3-540-42451-2, 2002.

[11] Miller B.L., Shaw M.J., "Genetic Algorithms with dynamic Niche Sharing for Multimodal Function optimization", University Of Illinois at Urbana-Champaign, Dept. General Engineering, IlliGAL Report 95010, 1995.

[12] O'Neill, M., Ryan, C.: "Grammatical Evolution", IEEE Transactions on Evolutionary Computation, Vol. 5, No. 4, pp 349-358, 2001.

[13] Marti R., "THOR: A tool to test intrusion detection systems by variations of attacks", Master's Thesis, Swiss Federal Institute of Technology, March 2002.

[14] Roesch, M., "Snort - lightweight intrusion detection for networks", Proceedings of Thirteenth Systems Administration Conference – LISA, pp 229-238, 1999.

[15] Rubin S., Jha S., Miller B.P., "Automatic Generation and Analysis of NIDS Attacks", 20th Annual Computer Security Applications Conference - ACSAC, pp 28-38, 2004.

[16] Snort Web Site - www.snort.org, last accessed Mar 2005.

[17] Tan, K.M.C., Killourhy, K.S., Maxion, R.A., "Undermining an Anomaly-based Intrusion Detection System using Common Exploits", 5th International Symposium on Recent Advances in Intrusion Detection - RAID, Lecture Notes in Computer Science, LNCS 2516, pp 54-73, 2002.

[18] Vigna, G., Robertson, W., Balzarotti D., "Testing Network Based Intrusion Detection Signatures Using Mutant Exploits", ACM Conference on Computer Security, 2004.

[19] Wagner, D., Soto, P., "Mimicry Attacks on Host-based Intrusion Detection Systems", ACM Conference on Computer Security, pp 255-264, 2002.