Mimicry Attacks Demystified: What Can Attackers Do To Evade Detection?

H. Güneş Kayacık, A. Nur Zincir-Heywood
Dalhousie University, Faculty of Computer Science,
6050 University Avenue, Halifax, Nova Scotia. B3H 1W5
{kayacik, zincir}@cs.dal.ca

Abstract

Mimicry attacks have been the focus of detector research where the objective of the attacker is to generate an attack that evades detection while achieving the attacker's goals. If such an attack can be found, it implies that the target detector is vulnerable against mimicry attacks. In this work, we emphasize that there are two components of a buffer overflow attack: the preamble and the exploit. Although the attacker can modify the exploit component easily, the attacker may not be able to prevent preamble from generating anomalous behavior since during preamble stage, the attacker does not have full control. Previous work on mimicry attacks considered an attack to completely evade detection, if the exploit raises no alarms. On the other hand, in this work, we investigate the source of anomalies in both the preamble and the exploit components against two anomaly detectors that monitor four vulnerable UNIX applications. Our experiment results show that preamble can be a source of anomalies, particularly if it is lengthy and anomalous.

Keywords

Information hiding, mimicry attacks, anomaly detection, vulnerability testing

1 Introduction

Over the past years, systematic methods for vulnerability testing have been proposed to analyze host based anomaly detection systems against blind spots and evasion attacks. The purpose of vulnerability testing is to locate vulnerabilities or holes in a detector before the attackers can exploit them. Previous research established that it is possible to evade anomaly detectors, namely Stide, by altering an attack to make it look like normal. To this end, mimicry attack notion was introduced [1] against anomaly detectors where the original exploit was crafted by producing a legitimate sequence of system calls while performing malicious actions, typically by making use of a template denoting a core attack. Consequently, the resulting mimicry attack remains within the normal operational limits and deploys undetected by the detector.

In particular, related detector blind spot testing [1, 2, 3] is mainly focused on the Stide detector [4] (or variants [5] thereof) and employed critical UNIX system applications. Undetectable exploits are developed by locating appropriate sequences of system calls that match the contents of Stide's normal behavior database whilst successfully reaching the behavioral objectives of the original exploit. Typically, a minimalist configuration of the anomaly detector is utilized, under the general observation that it is easier to make a strong detector if the alphabet of permitted instructions is small.

In a typical buffer overflow exploit, the first step is to corrupt the data types and local variables, which gives the attacker the control of the application. For example, in case of an Ftpd attack [9], attacker achieves this by logging onto the ftp server anonymously and issuing malformed commands such as *CWD ~{*. We call the actions taken by the attackers before they gain full control of the application as the preamble. During the preamble phase, the application is still operational and the attacker does not have the full control yet, hence the attacker may not be able to prevent the vulnerable application from generating anomalous behavior.

After the attacker gains control of the application, the second step is to execute arbitrary code or command to carry out a malicious action such as spawning a root shell or creating a super-user account. Commonly, this is achieved by injecting a shellcode. Shellcode is a short segment of an assembly program that aims to execute code on the vulnerable host. In case of the Ftpd attack, the shellcode spawns a UNIX shell with super-user privileges and binds it to a port so that the attacker can login without supplying a password. Attackers can modify the exploit components fairly easily, by changing the injected shellcode, to evade detection. On the other hand, modifying the preamble requires finding an alternative way to take advantage of the vulnerability or finding another vulnerability, therefore cannot easily be modified.

In previous work [1, 2, 3], the attack was said to be optimal if the exploit component raised no alarms. However, even the exploit raises no alarms, introducing the preamble can introduce alarms for both the preamble itself and the transition between the preamble and exploit. Therefore, in this work, we expand upon our previous work [11] and investigate the source of anomalies on both preamble and exploit components. To this end, we employ two anomaly detectors to monitor four UNIX applications with known vulnerabilities. We observe the anomaly rates for the original attacks that are downloaded from the SecurityFocus website [7, 8, 9, 10]. Furthermore, we employed a mimicry attack generation methodology to observe the change in anomaly rates. The methodology automates the design of exploits while utilizing only the anomaly rate without using internal data structures or algorithms specific to a detector.

Furthermore, since the anomaly rates reported for mimicry attacks in previous work was zero, locality frame count feature of anomaly detectors were not investigated in depth. Locality frame count keeps track of the anomalies over a given time period based on the assumption that security violations will tend to produce clustered anomalies more than legitimate errors [5]. This implies that, if the preamble produces a sufficiently large cluster of anomalies, the attack can be detected and stopped before the exploit deploys. To this end, we also investigate the effects of anomalies clustered together, in this work.

The remainder of the paper is organized as follows. Relevant work on mimicry attacks is discussed in Section 2. The anomaly detectors employed in our analysis are introduced in Section 3. The four UNIX applications with known, documented vulnerabilities that the anomaly detectors monitor are presented in Section 4. The mimicry attack generation methodology that we employed in our analysis is briefly discussed in Section 5. The results of our analysis detailing the anomaly rates and the source of anomalies in mimicry attacks are reported in Section 6 and the conclusions are drawn in Section 7.

2 Related Work

Wagner et al. [1] introduced the "mimicry attack" concept, where original attacks were modified to evade detection. They proposed three methods to avoid detection: (i) modifying system call parameters; (ii) inserting system calls that are irrelevant to the attack being deployed while minimizing the anomaly rate; and finally (iii) generating equivalent attacks by replacing the system calls that can easily be identified by the detector. An example for the last method is substituting an attack that spawns a UNIX shell with an attack that creates a super-user account. Both of these give the attacker super-user privileges. Mimicry attacks were generated for wuftpd service by manually modifying the detectable system call sequences. Normal behavior was generated by "running wuftpd on hundreds of large file downloads over a period of two days"[1]. Although Wagner et al. was aware of the preambles, they assumed that the attacker could silently take control of the application without being detected. In our work, we discuss that such an assumption may not always be the case.

Tan et al. [2] employed four methods to manually change the behavior of the attack: (i) hiding an attack in the blind spot of the detector; (ii) modifying an attack so that it looks like a normal behavior; (iii) hiding an attack so it looks like a less dangerous attack; and (iv) modifying an attack so that it looks like a different attack. In their experiments, attacks against restore, tmpwatch and kernel/traceroute applications were employed. Normal behavior for Restore is obtained by "monitoring a regular user executing the restore system program to retrieve backup data from a remote backup server" [2]. Normal behavior for tmpwatch is generated by populating a short directory tree with files under */tmp* and executing tmpwatch program to clean files that are more than 5 days old. Normal behavior for kernel attack was not obtained since the vulnerability in the kernel was used to exploit another vulnerability in traceroute.

Stide detects foreign sequences that are not in the normal database. Thus, Tan et al. [3] investigated hiding in the detector's blind spots in more detail by developing variants of a core exploit with the objective of increasing the minimal foreign sequence length. They reported that if the foreign sequence length is greater than the sliding window size of Stide, an attack could evade detection. In their experiments they employed Stide on traceroute and passwd applications. For traceroute, normal behavior is obtained by "executing traceroute to acquire diagnostic information regarding the network connectivity between the localhost and nis.nsf.net"[3]. For the passwd application, normal data was obtained by executing the passwd without any arguments, which expires the old password and installs the new one provided by the user.

Using a categorization scheme, Gao et al. [13] divided anomaly detectors into three categories: black-box detectors [4] only make use of the system calls, whereas gray-box detectors [16, 17] use – in addition to system calls – runtime observations such as program counter and return addresses stored in the stack. White-box detectors, however also incorporate information from the source code, which makes it difficult to hide the attacks. The authors presented a systematic study, which showed the benefits and overheads of changing gray-box anomaly detector parameters such as (1) the amount of runtime information, (2) atomic unit that the detector monitors and (3) sliding window size. Experimental results indicated that expanding the model by using more information and increasing window size results would increase the mimicry attack length. In other words, attackers will need more code to hide their actions. Although it is more difficult to evade white-box detectors, authors discussed that they are platform dependent and they are not universally applicable [13]. In addition to the systematic study, authors present a methodology to forge the program counter information on statically linked executables so that the detector [4] does not detect an anomaly in the return addresses even though system calls are made by the attack code [13].

Kruegel et al. developed a methodology where the detection system is adaptive (i.e. Stide variants [4, 5]). This time, the ability to build exploits automatically was used to improve the operation of the detector [14]. However, in [14], automation is performed using a static tool at the Intel x86 assembly level to redirect control flow using symbolic execution.

Giffin et al. generated mimicry attacks against Stide by applying automatic model checking to prove that no reachable operating system configuration corresponds to the effect of an attack [15]. However, in their approach, the operating system model, application (program) model and system call specifications as well as the attack configuration are still generated manually.

Parampalli et al. [18] proposed a mimicry attack methodology against "powerful system call monitors". "Powerful system monitor" is defined as a detector that has full knowledge of the system call parameters as well as their roles in the execution of the system call. They introduced persistent interposition attack concept where the objective of the attacker is to modify the read and write system calls to deploy the attack. Their methodology is similar to man-in-the-middle attacks since the objective of the attack code is to intercept and modify the read and write system calls that the victim application makes. Their results on Apache web server showed that although the persistent interposition attacks are not powerful enough to obtain a rootshell, they can evade monitors that monitor system call arguments while achieving goals such as stealing financial information or impersonating web servers.

3 Anomaly Detectors

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. When a buffer overflow attack is deployed, a vulnerable privileged program is exploited to do something that it is not supposed to do. This implies that it is possible to observe a change in the program behavior. Anomaly based detectors are based on this assumption. Needless to say, if the attack is sufficiently similar to the normal behavior, it may not be detected. In this work, we employed Stide and pH anomaly detectors in our experiments.

3.1 Stide

Forrest et al. [4] employed a methodology motivated by immune systems. This characterizes the problem as distinguishing 'self' from 'non-self' (normal and abnormal behaviors respectively). An event horizon is built from a sliding window applied to the sequence of system calls made by an application during normal use. The sequences formed by the sliding window are stored in a table that establishes the normal behavior model. During the deployment (detection) phase, if the pattern from the sliding window is not in the normal behavior database it is considered a mismatch.

Input to the Stide detector takes the form of system call traces of an application for which the detector is trained. Specifically, Stide builds a "normal database" by segmenting the training data (of system call traces) into fixed length sequences [6]. To do so, a sliding window of *N* is employed over the training dataset and the resulting system call patterns are stored in the "normal database". During testing, the same sliding window size is employed on the data. Resulting patterns are compared against the "normal database" and if there is no match, a mismatch is recorded. Given a window size of *N* and system call trace length *M*, anomaly rate for the trace is calculated by dividing the number of mismatches by the number of sliding window patterns (i.e. $M - N + 1$). In our experiments, we employed the default training parameters for Stide listed in Table 1.

Table 1. Stide configuration parameters

Parameter	Setting
Sliding window length	6

3.2 Process Homeostasis (pH)

Process Homeostasis (pH) [5] is an anomaly detector based on Stide that employs a detection methodology similar to Stide. pH is implemented as an extension to Linux 2.2 Kernel. Therefore, pH monitors system calls more efficiently by capturing system calls directly at the kernel level as opposed to Stide that employs Strace to capture system calls. pH monitors the changes in short system call sequences by employing look ahead pairs. While employing the sliding window approach, pH does not store the sliding window patterns but records tuples, which consist of the current and the past system calls and the sliding window location. Somayaji [5] established that look ahead method is more efficient to store, could potentially converge to a normal profile quicker than the full sequence method. Additionally, tolerization and sensitization concepts were introduced. Tolerization allows pH to improve false alarm rates by leaving out minimal anomalies, which is likely to be slight changes in normal behavior. Sensitization prevents abnormal behavior from leaking into normal behavior database [5].

During training, a sliding window is employed over the training set and a "normal database" of three dimensional matrix is built where the dimensions are as follows: (1) current system call; (2) previous system call; (3) location of the previous system call on the sliding window. During testing, the same sliding window is employed on the test data. If a given sliding window sequence produced a look ahead pair that is not in the normal database, a mismatch is recorded. Similar to Stide, given a window size of *N* and system call trace length *M*, anomaly rate for the trace is calculated by dividing the number of mismatches by the total number of look ahead pairs.

Another important feature of pH is that it responds to attacks by slowing down the process. Delay is an exponential function of locality frame count. Locality frame count aims to identify the clusters of anomalies. To this end, pH simply maintains a count of how many of the past LF (usually 128) system calls were anomalous. Process delays can substantially delay the execution of a program when a cluster of anomalies is observed. In our experiments, we employed the default pH training parameters listed in Table 2.

Table 2. pH configuration parameters

Parameter	Setting
Look ahead pair window size	9
Locality frame window size	128
Delay Factor	1
Suspend execve after	10 anomalies
Suspend execve duration	2 days
Anomaly limit	30
Tolerize limit	12

4 Vulnerable Applications

In our experiments, we employed four Linux applications, which have known and documented vulnerabilities, namely Traceroute, Restore, FtpD, and Samba. These are also the vulnerable applications used in the mimicry attack literature [1, 2, 3]. Traceroute, and Restore vulnerabilities can be exploited locally whereas FtpD and Samba vulnerabilities can be exploited remotely. For each application, we developed normal use cases, which represent the scenarios of legitimate use.

4.1 Traceroute

Traceroute is a network diagnosis tool, which is used to determine the routing path between a source and a 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 [7]. The attack takes advantage of vulnerability in malloc chunk, and then uses a debugger to determine the correct return address to take control of the program. In order to analyze the traceroute behavior under normal conditions, we developed five use cases, Table 3; whereas in the previous research [3] only one normal use case was used for training, namely use case 1.

Table 3. Traceroute normal use cases

Use Case	System Calls
1. Target a remote server	736
2. Target a local server	260
3. Target a non existent host	153
4. Target localhost	142
5. Help screen	24

4.2 Restore

Restore is a component of UNIX backup functionality, which restores the file system image taken by the dump command. Files or directories can be restored from full or incremental backups.

Restore version 0.4b15 is vulnerable to an environment variable attack where the attacker modifies the path of an executable and runs restore. This results in executing an arbitrary command with super-user privileges, which leads to a root compromise. In the published attack [8], attacker spawns a root shell. Table 4 summarizes five normal use cases that we developed for Restore. As in the previous work [2], we have monitored a regular user executing the restore system program to retrieve backup data from a remote backup server. However, we have also repeated the case for different sizes of files and back-up types.

Table 4. Restore normal use cases

Use Case	System Calls
1. Restore a small file system dump from a full backup.	2256
2. Restore a small file system dump from an incremental backup.	1027
3. Restore a large file system dump from a full backup.	167207
4. Restore a large file system dump from an incremental backup.	68185
5. Help screen	53

4.3 Samba

Samba suite provides printer and file sharing for Windows clients and can run on most UNIX variants. Samba sets up printer and network shares that appear as Windows disks and printers under a Windows operating system.

Redhat 9.0 is shipped with Samba suite version 2.2.7a, which has a vulnerability [9] that can be exploited over the network to gain super-user privileges. The buffer overflow occurs when Samba service tries to copy user supplied data into a static buffer without checking. The published attack binds a root shell to a network port. Table 5 summarizes the six normal use cases that we developed for Samba.

Table 5. Samba normal use cases

Use Case	System Calls
1. Mount a samba share successfully	1156
2. Invalid password while mounting samba share	680
3. Unmount a samba share successfully	186
4. Find and edit a remote file. (Using commands: ls - cd - ls - pico)	254
5. Find and copy a 38MB remote file to a local directory (Using commands: ls - cd - cp)	65648
6. Change samba password remotely	1527

4.4 FtpD

Redhat 6.2 is shipped with Washington University Ftp Server version 2.6.0(1), which provides FTP access to remote users. WuFtpd 2.6.0(1) is susceptible to an input validation attack where the attacker can corrupt the process memory by sending malformed commands and overwrite the return address to execute his/her shellcode. Although the attack [10] is an input validation attack, the deployment is similar to a buffer overflow attack. Table 6 summarizes the ten normal use cases that we developed for FtpD. Use cases 7 through 10 represents the legitimate errors that a user can make during a normal FTP session. On the other hand, in the previous research [1] wuftpd was run on only large file downloads over a period of two days.

Table 6. FtpD normal use cases

Use Case	System Calls
1. Upload 10K data	2249
2. Upload 60M data	32912
3. Upload 650M data	334252
4. Download 10K data	2252
5. Download 60M data	32908
6. Download 650M data	334244
7. Three failed login attempts	2236
8. Help screen	2017
9. Attempt to access non-existent files and directories	2213
10. Type non-existent commands.	2017

5 Generating Mimicry Attacks

In previous mimicry attack generation research, typically a “white box” access to detector behavior is assumed. This has resulted in very efficient algorithms for designing mimicry exploits. In particular, such research has for the most part concentrated on the Stide open source host based anomaly detector [4] or its improved versions [5]. The design of exploits then boils down to locating sequences of system calls that both match the contents of an anomaly detector’s normal behavior database whilst successfully reaching the behavioral objectives of the original ‘core’ exploit. A minimalist configuration of the anomaly detector is utilized, under the general observation that it is easier to make a strong detector if the alphabet of permitted instructions is small. Thus, any weakness detected under these conditions will be exasperated when more realistic configurations of the detector’s normal behavioral database are employed (a larger alphabet of normal behavior will result in even more opportunities for defeating the detector). Such a problem was shown to be of a sufficiently focused form for exhaustive search algorithms to solve the problem in seconds with solutions returning the equivalent of a zero anomaly rate.

On the other hand Kayacik et al. [12] assumes a “black box” approach to mimicry attack generation. As such the feedback from the detector is limited to the anomaly rate alone, where this is provided as a part of normal operation. Hence the approach does not make use of the internal data structures or algorithms specific to a particular detector. The selected methodology [12] has the advantage of requiring only the anomaly rate from the detector, hence working with a black-box assumption whereas other related methods [1, 2, 3] require the internal knowledge of the target detector. The process of generating mimicry attacks was automated using Genetic Programming (GP). Throughout the search process, GP maintains a population of candidate solutions. Each candidate solution, or individual takes the form of a program. Programs were represented as a sequence of system calls where the set of permitted system calls was predefined by the user. The search process progresses through the iterative application of GP search operators. Performance (fitness) of an attack is evaluated using a fitness function, which ranks the population according to attack success and anomaly rate. In our analysis, we selected the mimicry attack that produces the least anomaly rate at the end of the search process and employed them in our experiments. We note that the method used for mimicry attack generation can be substituted with other methods without the loss of generalization as long as the resulting mimicry attacks produces fewer anomalies.

6 Analysis

Stide and pH are trained on the normal system call traces collected from executing the normal use cases detailed in Section 4. Therefore there exist two detector configurations (namely Stide and pH) for each vulnerable application. We employed the attacks that were made available for Traceroute, FtpD, Samba and Restore, which are called original attacks hereafter. We deployed the original attacks while detectors were monitoring the vulnerable processes and recorded the anomaly rates. We then employed our mimicry attack generation methodology to generate attacks with minimized anomaly rates.

Tables 7 and 8 detail the anomaly rates reported by Stide and pH respectively. Although the mimicry attacks succeed in minimizing the anomaly rates, they are above zero in case of all four applications. Mimicry attack against Samba application succeeds in minimizing the anomaly rate down to ~3% whereas the mimicry attack against Restore minimized the anomaly rate to ~46%.

Anomaly rates reported in Tables 7 and 8 include the anomaly rates from preamble and the exploit whereas Tables 9 and 10 detail the length and the anomaly rates of the exploit component of mimicry attacks. Results show that although the exploit produces anomaly rates close to the optimal 0%, anomaly rate for an attack can be substantially greater. For example, in Tables 9 and 10, although the exploits for Restore produces 0.1% and 0.4% anomaly rates, anomaly rate of the attacks are 46.14% and 48.52%, for Stide and pH respectively. The reasons behind this phenomenon are discussed in the next section

Table 7. Anomaly rates of original and mimicry attacks reported by Stide

	Original Attack	**Mimicry Attack**
Traceroute	60.67	20.59%
Restore	84.65%	46.14%
Samba	10.00%	2.94%
FtpD	22.69%	19.14%

Table 8. Anomaly rates of original and mimicry attacks reported by pH

	Original Attack	**Mimicry Attack**
Traceroute	65.44%	21.62%
Restore	87.47%	48.52%
Samba	15.84%	7.96%
FtpD	25.34%	15.92%

Table 9. Exploit characteristics of the mimicry attacks against Stide

	Anomaly Rate	**# System Calls**
Traceroute	16.66%	30
Restore	0.40%	996
Samba	0.50%	996
FtpD	57.14%	7

Table 10. Exploit characteristics of the mimicry attacks against pH

	Anomaly Rate	**# System Calls**
Traceroute	11.71%	111
Restore	0.10%	993
Samba	0.10%	993
FtpD	0.10%	993

6.1 A Closer Look at Preamble

The buffer overflow attacks generally aim to inject a shellcode in a vulnerable buffer and have the vulnerable application to execute the injected assembly program. The size of the vulnerable buffer is generally too short to inject an entire program therefore the injected shellcode executes system calls on the target host to spawn a UNIX shell or write to the password file of the host with super-user privileges.

In this work, we make the observation that there are two parts to each attack, the preamble and the exploit. The preamble is composed of the system calls executed during the phase where the attacker tries to gain control of the vulnerable application. On the other hand, exploit includes the system calls executed after attacker has the *full* control. We believe that differentiation is necessary since the attackers can alter the system calls executed after they have full control whereas during the preamble phase, where the attacker prepares the vulnerable application for the buffer overflow, the interaction between the attacker and the application may inevitably cause anomalous system calls. In other words, attackers can modify the exploit components fairly easily to evade detection. Modifying the preamble requires finding an alternative way to take advantage of the vulnerability or finding another vulnerability, therefore cannot easily be modified.

The boundary between the preamble and the exploit can be determined by locating the first action of the shellcode. All four attacks execute *execve('/bin/sh')* system call to spawn a UNIX shell with super-user privileges. Any system call including and after *execve('/bin/sh')* is a result of the spawned UNIX shell whereas the system calls before *execve('/bin/sh')* are executed while the attacker was corrupting the data types and variables to deploy the exploit. Table 11 details the ratio of mismatches reported by Stide and pH for preamble and exploit components separately for the four attacks. As discussed in Section 3.1, a mismatch is recorded, if the current observed behavior does not match any behavior in the "normal database". Anomaly rate is then calculated by dividing the number of mismatches by the total number of observations.

Table 11. Ratio of mismatches (%) for attacks reported by Stide for the preamble and exploit components separately

Traceroute			
	Preamble	**Exploit**	**Total**
System Calls	24%	76%	344
Mismatches (Stide)	8.04%	91.96%	199
Mismatches (pH)	12.62%	87.38%	214
Restore			
System Calls	32%	68%	4454
Mismatches (Stide)	30.83%	69.17%	3613
Mismatches (pH)	30.89%	69.11%	3881
Samba			
System Calls	88%	12%	4396
Mismatches (Stide)	31.64%	68.36%	433
Mismatches (pH)	55.62%	44.38%	694
FtpD			
System Calls	86%	14%	3024
Mismatches (Stide)	73.20%	26.80%	679
Mismatches (pH)	75.20%	24.80%	762

In total, Traceroute attack executed 344 system calls of which 24% belongs to the preamble component and 76% to the exploit. The attack as a whole (i.e. preamble and exploit combined) produced 199 mismatches 91.96% of which was generated by the exploit. Therefore, in the case of the Traceroute attack, an attacker can alter his exploit and substantially reduce the anomaly rate. Similar observations can be made for Restore attack. On the other hand, FtpD attack executed 3024 system calls, of which 86% belongs to the preamble component and 14% to the exploit. The attack as a whole produced 679 mismatches, 73.20% of which was generated by the preamble. Consequently, modifying the exploit would have less impact on the anomaly rate for FtpD. Samba attack exhibits similar properties however the preamble produced a smaller portion of the overall anomaly rate compared with the FtpD attack.

Moreover, we analyze the progression of anomaly rates for the original and the mimicry attacks against Stide and pH in Figures 1 and 2, respectively. This is achieved by recording the anomaly rate as the detectors monitor new system calls. The boundary between the preamble and the exploit is marked with a vertical line. From Figure 1 and 2 and Table 11, it is apparent that the preamble produces anomalies. A spike in the anomaly rate indicates a cluster of anomalies whereas a decreasing trend indicates that the anomaly rate for the current region is lower or zero, hence reducing the overall anomaly rate. Preamble component of the original attacks produce anomalies hence continuing to raise the anomaly rate, whereas the exploit component of the mimicry attacks produce zero or very few anomalies hence reducing the anomaly rate.

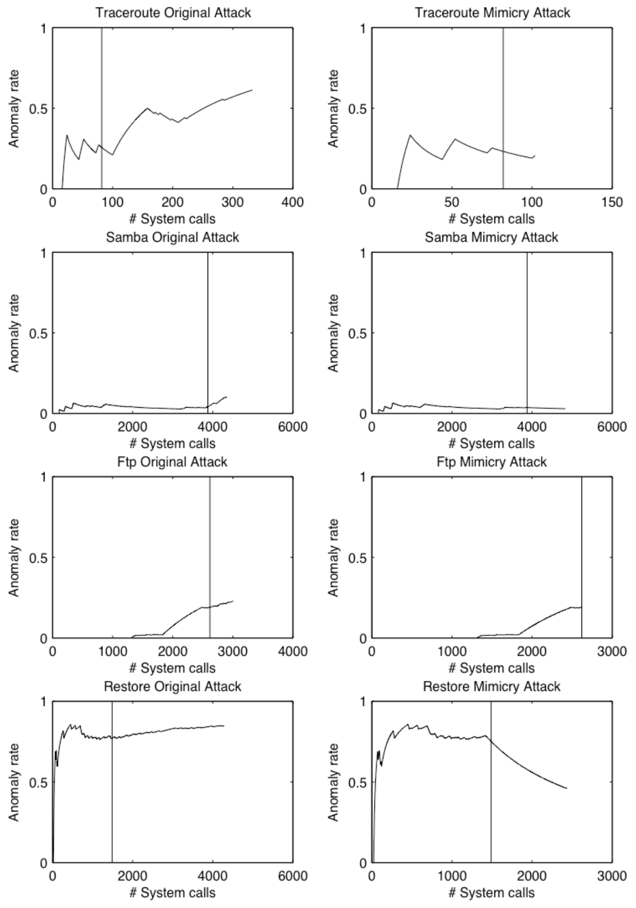

Figure 1. Anomaly rates of the original and mimicry attacks reported by Stide

The length of the preamble gains importance when determining the operational limits of Stide and pH. Specifically, if the preamble is short and if the attacker manages to modify his exploit accordingly (e.g. instead of spawning a root shell, create a super-user account), the anomaly rate of the attack as a whole can be substantially reduced. However if the preamble is long, there will be a higher likelihood of raising alarms no matter what type of exploit is being used. Figures 3 and 4 details the ideal anomaly rates of attacks on different exploit length conditions with the optimal (zero percent) exploit anomaly rates. Figures 3 and 4 indicate that the attacker needs longer exploits for Restore, Samba and FtpD.

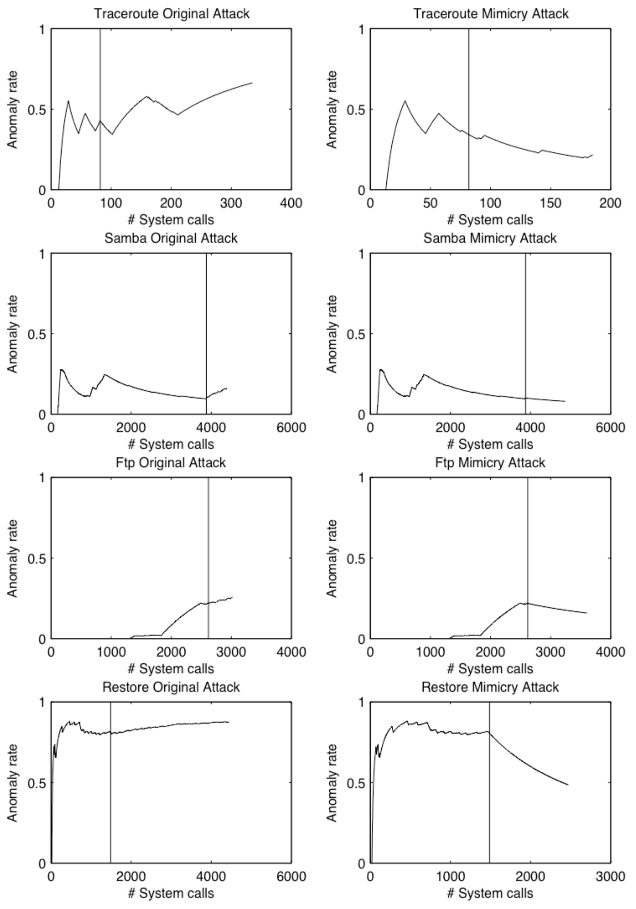

Figure 2. Anomaly rates of the original and mimicry attacks reported by pH

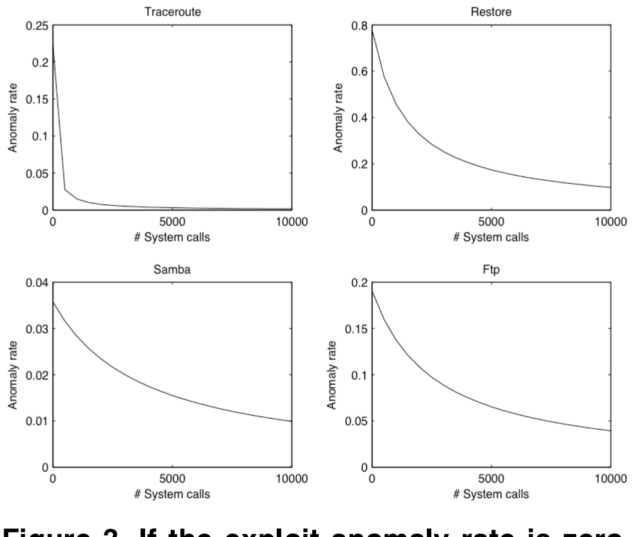

Figure 3. If the exploit anomaly rate is zero, the optimal anomaly rate of an attack on different exploit length conditions against Stide

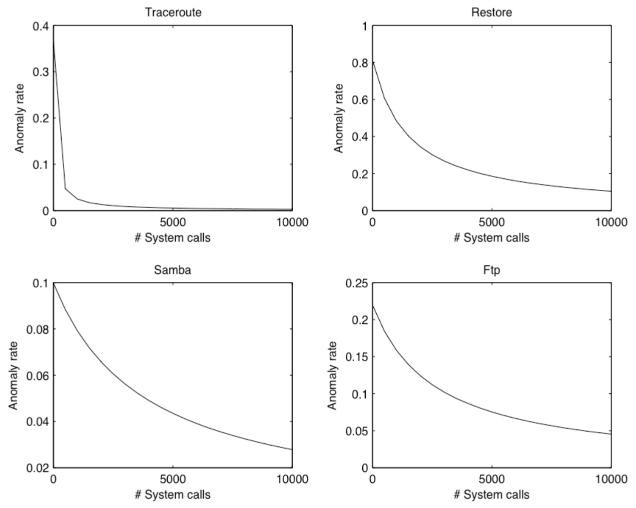

Figure 4. If the exploit anomaly rate is zero, the optimal anomaly rate of an attack on different exploit length conditions against pH

Our analysis indicate that anomaly rate returned for the exploit alone does not represent the anomaly rate returned for the entire attack since the activities associated with gaining the control of the application (preamble) raises alarms. Furthermore, the ratio of the preamble to the exploit and the anomaly rate from the preamble plays an important role in the overall anomaly rate of an attack. It is evident that anomaly rate of an attack can be better reduced where the exploit is relatively longer than the preamble, even though the exploit itself raises some alarms, Figures 3 and 4.

In previous work [1, 2 3], authors successfully developed mimicry attacks against Stide with 0% anomaly rates. However, the previous work did not take the preamble into consideration. In Table 12, we present the actual anomaly rate that Stide would report if the original exploits were replaced by equivalent exploits of the same size that raised no alarms. It is crucial to note that mimicry attacks can reduce the anomalies below the values in Table 12 by employing longer exploits hence changing the preamble to exploit ratio.

It is evident that even the exploit raises no alarms, the preamble will still cause anomaly rates between ~2% and ~25%. Furthermore, in previous work, an attack was considered optimal, if the exploit never generated any mismatches against the Stide database. Stide counts mismatches between the candidate trace and sequences of normal behavior in the detector database. That is to say, a sliding window comparison is made between the database and the candidate trace, in case of an attack preamble plus exploit. Therefore, even though exploit raises no alarms, introducing the preamble will return mismatches (alarms) for both the preamble itself, and at the transition between the preamble and the exploit. Thus, for a predefined preamble, the best that a mimicry attack can do is to minimize the contribution from the exploit and the transition from the preamble to the exploit.

Table 12. If Exploit Anomaly rate = 0%, the anomaly rate associated with the Preamble component of the original attacks

	Anomaly Rate (from preamble only)
Traceroute	4.72%
Restore	25.04%
Samba	3.16%
Ftpd	16.46%

6.2 Locality Frame Count and Process Delays in pH

Although Stide keeps track of locality frame count, pH employs the locality frame count to delay the processes. Locality frame count keeps track of the mismatches over a given time period (by default 128 system calls). Therefore, a cluster of mismatches produces high locality frame counts whereas the same number of mismatches distributed over the attack produces smaller locality frame count values.

pH responds to attacks by slowing down the process based on the observed locality frame count. The delay associated with the current system call can be expressed as [5]:

$$delay\_factor \times 0.01 \times 2^{LFC}$$

where higher *delay_factor* values produce longer process delays and the LFC signifies how many of the past 128 system calls were anomalous.

Figure 5 details the locality frame counts observed for the original and mimicry attacks as they progress. In case of the original attacks, locality frame counts are either increasing or remaining the same, whereas mimicry attacks manage to reduce the observed locality frame counts. This is particularly apparent for traceroute where the exploit of the mimicry attack is sufficiently long to reduce the locality frame count below 20, whereas the original traceroute exploit increases it above 100.

Figure 6 shows the total delay observed for each attack as the attack progresses. For both original and mimicry attacks, it is apparent that the delays are expressed in days therefore locality frame count is an effective way to stop the attacks. Even a quick increase in locality frame count is sufficient to stop an attack since its effects are exponential and the value remains high until the locality frame moves to a segment with few anomalies. As an example, the sharp increase in locality frame count for the original ftp attack (Figure 5) causes a 0.01 x 2^{100} second delay which is roughly 1.47 x 10^{23} days. Therefore, once the locality frame count passes a certain limit, pH effectively "freezes" the attack hence preventing the successful execution of the exploit. This implies that even an attack with 0% anomaly rate exploit can be detected and stopped by focusing on the preamble alone. Thus, this shows the importance of the preamble and the results we obtained through the analysis discussed above.

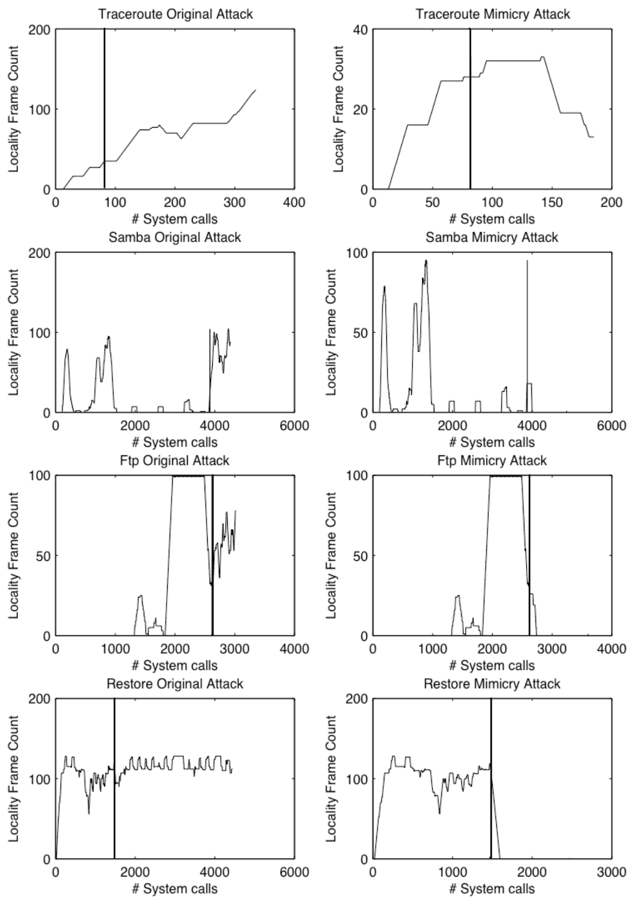

Figure 5. Locality Frame Count of the original and mimicry attacks for pH

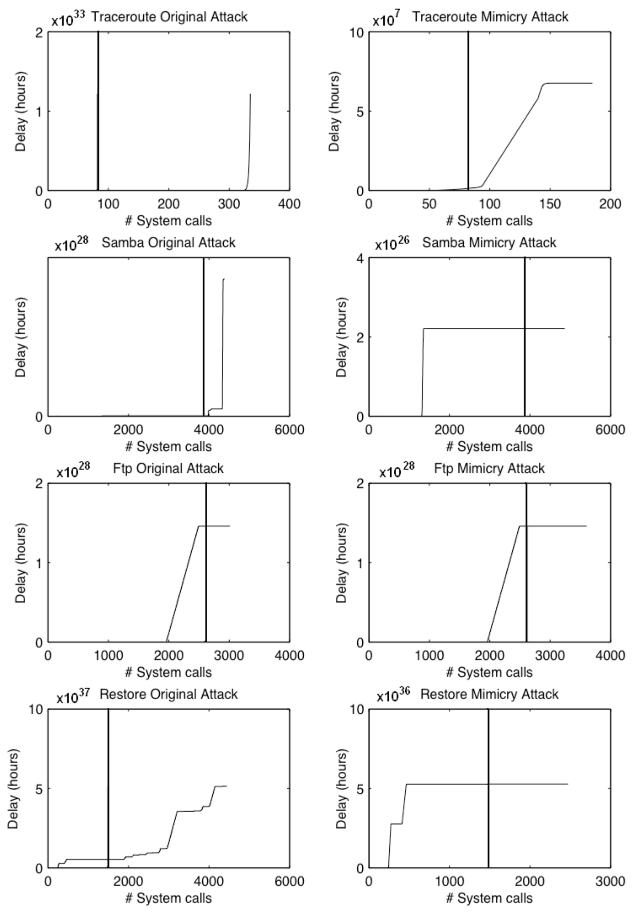

Figure 6. Delay associated with the original and mimicry attacks for pH

7 Conclusion

In this paper, we discussed that there are two components to every attack, namely the preamble and the exploit. Previous work in mimicry attack generation reported 0% anomaly rates, however they focused on the exploit alone. Anomaly rates for preamble and exploit components should be analyzed together, since it is not possible to execute an exploit without a break in to the system. To this end, we deployed Stide and pH anomaly detectors to monitor four UNIX applications that have known vulnerabilities. We then deployed the original attack to establish a baseline anomaly rate and deployed the mimicry attack while observing the anomaly rate and locality frame count changes as the attacks progress.

Our analysis results show that it is highly difficult to evade an anomaly detector with 0% anomaly rate. In the past, where such results were achieved, the anomaly rate was only calculated by counting the mismatches over the length of the exploit part, ignoring the contribution from the preamble. However, in practice buffer overflow attacks have two stages: (i) the break in, which we call as the preamble in this work; and (ii) the exploit itself. Even though it may be possible to achieve a 0% anomaly rate on an exploit alone, overall it will still have a non-zero anomaly rate associated with the preamble and the transition from the preamble to the exploit. The effect of the preamble and the transition from the preamble to the exploit is emphasized more when the size of the preamble part of an attack is greater than the size of the exploit part, as in the case of FtpD. Indeed, one can try to change this ratio by artificially increasing the length of the exploit but even then it is highly difficult to make it 0% (Figures 3 and 4) due to the effect and limitations on the total attack length i.e. as in finite buffer sizes. Thus, for a predefined preamble, the best that a mimicry attack can do is to minimize the contribution from the exploit and the transition from the preamble to the exploit.

Furthermore, experiment results showed that a delay associated with locality frame count is effective in preventing an attack to be deployed successfully. Specifically, an attack that achieves a low anomaly rate may be delayed for weeks if the anomalies are clustered together hence increasing the locality frame count. We believe that future mimicry attack research should move from focusing on the anomaly rate of the exploit alone to investigating multiple characteristics of an attack such as the anomaly rate of the preamble and the locality frame count.

Finally, future work will consider the analysis of different anomaly detectors to understand the effect of preamble in more detail. Moreover, a framework, which includes the effect of the preamble in the vulnerability/penetration testing, will be investigated.

Acknowledgments

The authors gratefully acknowledge the support of CFI New Opportunities, NSERC Discovery, MITACS and SwissCom Innovations Inc. The first author is a recipient of a Killam pre-doctoral scholarship. All research was conducted at Dalhousie NIMS Laboratory, http://www.cs.dal.ca/projectx/.

References

[1] D. Wagner, P. Soto: Mimicry attacks on host based intrusion detection systems, ACM Conference on Computer and Communications Security, pp. 255-264, 2002.

[2] Kymie M. C. Tan, John McHugh, Kevin S. Killourhy: Hiding Intrusions: From the Abnormal to the Normal and Beyond, Information Hiding, pp. 1-17, 2002.

[3] Kymie M. C. Tan, Kevin S. Killourhy, Roy A. Maxion: Undermining an Anomaly-Based Intrusion Detection System Using Common Exploits, RAID, pp. 54-73, 2002.

[4] Forrest S., Hofmeyr S. A., Somayaji A., Longstaff T. A.: A sense of self for Unix processes, IEEE Symposium on Security and Privacy, pp. 120--128, 1996.

[5] Somayaji, A. B.: Operating System Stability and Security Through Process Homeostasis. Doctoral Thesis. UMI Order Number: AAI3058952., The University of New Mexico. 2002.

[6] Kymie M.C. Tan, Roy A. Maxion: “Why 6?" Defining the Operational Limits of stide, an Anomaly-Based Intrusion Detector, IEEE Security and Privacy, pp. 188-201, 2002.
[7] SecurityFocus Vulnerability archives: LBNL Traceroute Heap Corruption Vulnerability, http://www.securityfocus.com/bid/1739
[8] SecurityFocus Vulnerability archives: RedHat Linux restore Insecure Environment Variables Vulnerability, http://www.securityfocus.com/bid/1914/
[9] SecurityFocus Vulnerability archives: Samba 'call_trans2open' Remote Buffer Overflow Vulnerability, http://www.securityfocus.com/bid/7294
[10] SecurityFocus Vulnerability archives: Wu-Ftpd Remote Format String Stack Overwrite Vulnerability, http://www.securityfocus.com/bid/1387/
[11] Kayacik H. G., Zincir-Heywood A. N.: On the Contribution of Preamble to Information Hiding in Mimicry Attacks, Proceedings of the 3rd IEEE International Symposium on Security in Networks and Distributed Systems, 2007.
[12] Kayacik H. G., Zincir-Heywood A. N., Heywood M. I.: Automatically Evading IDS Using GP Authored Attacks, Proceedings of the IEEE Computational Intelligence for Security and Defense Applications, 2007.
[13] Gao, D., Reiter, M. K., and Song, D.: Gray-box extraction of execution graphs for anomaly detection. In Proceedings of the 11th ACM Conference on Computer and Communications Security CCS '04.
[14] Kruegel C., Kirda E., Mutz D., Robertson W., Vigna G.: Automating mimicry attacks using static binary analysis, USENIX Security Symposium, pp. 717-738, 2005.
[15] Giffin J. T., Jha S., Miller B.P.: Automated Discovery of Mimicry Attacks, RAID, 2006.
[16] Sekar R., Bendre M., Dhurjati D., Bollineni P.: A Fast Automation-based Method for Detecting Anomalous Program Behavior, IEEE Security & Privacy Symp., 2001.
[17] Feng H., Kolesnikov O., Fogla P., Lee W., Gong W: Anomaly detection using call stack information. In IEEE Symposium on Security and Privacy, Oakland, California, May 2003.
[18] Parampalli, C., Sekar, R., and Johnson, R.: A practical mimicry attack against powerful system-call monitors. In Proceedings of the 2008 ACM Symposium on information, Computer and Communications Security. ASIACCS '08.