Automatic Synthesis of Filters to Discard Buffer Overflow Attacks: A Step Towards Realizing Self-Healing Systems*

Zhenkai Liang, R. Sekar, and Daniel C. DuVarney
Department of Computer Science, Stony Brook University
{zliang,sekar,dand}@cs.sunysb.edu

Abstract

Buffer overflows have become the most common target for network-based attacks. They are also the primary propagation mechanism used by worms. Although many techniques (such as StackGuard) have been developed to protect servers from being compromised by buffer overflow attacks, these techniques cause the server to crash. In the face of automated, repetitive attacks such as those due to worms, these protection mechanisms lead to repeated restarts of the victim application, rendering its service unavailable. In contrast, we present a promising new approach that learns the characteristics of inputs associated with attacks, and filters them out in the future. It can be implemented without changing the server code, or even having access to its source. Since attack-bearing inputs are dropped *before* they corrupt the victim process, there is no need to restart the victim; as a result, recovery from attacks can be very fast. We tested our approach on 8 buffer overflow attacks reported in the past few years on `securityfocus.com` and were available with working exploit code, and found that it generated accurate filters for 7 out of these 8 attacks.

1 Introduction

Self-healing is emerging as an exciting new area within computer security. A key characteristic of approaches in this area is their ability to detect ongoing attacks, identify the underlying vulnerability being exploited, and adapt the system to "heal" the vulnerability. Once healed, the system becomes immune to subsequent attacks that exploit the same vulnerability. An important benefit of self-healing is that it avoids system resources from being spent on reactive defenses, such as system restarts, which can adversely impact system availability.

Although self-healing approaches have been studied in the context of spontaneous faults, they have just begun to receive attention in the context of computer and network security. We present a new approach that represents an important first step towards realizing *practical* defenses that employ self-healing. Our approach focuses on buffer overflows, which have become the most common target for network-based attacks. Among the COTS-related security advisories released by CERT Coordination Center in 2003 to 2004, 41 of the 51 were related to buffer overflows. Moreover, they are the primary mechanism used by worms in order to propagate.

The state-of-art in defenses against buffer overflows includes various *guarding* techniques [3, 4] for preventing execution from data segments, and *randomization* techniques [1, 2]. Although these techniques can detect attacks before vital system resources (such as files) are compromised, they cannot protect the victim process itself, whose integrity is compromised prior to the time of detection. For this reason, the safest approach for recovery is to terminate the victim process. With repetitive attacks, such as those due to worms, or other forms of automated attacks, these approaches will cause repeated server restarts, effectively rendering a service unavailable during periods of attack. In contrast, our self-healing approach can filter out attacks *before* process integrity is compromised, thereby enabling the service to continue without any interruption. Moreover, our approach doesn't require any user-supplied knowledge about the server, or access to its source code.

2 Overview of Approach

Our approach, called ARBOR (Adaptive Response to Buffer OveRflows), is designed to protect network server processes. It is based on the observation that attacks arrive via inputs to these processes. Figure 1 illustrates the architecture of ARBOR, which forms a protective layer between a process and the external environment by adaptively filtering out attack inputs. The adaptation is based on a feedback loop: inputs which don't trigger an intrusion report from the detector are allowed to pass through the filter unmodified, while inputs that trigger an intrusion report activate the feedback loop, causing the filter to be modified to block future attacks. The principal components of ARBOR are described below.

The *input filter* inspects all input entering the system and filters out (i.e., drops) those inputs that match existing filter rules. The filtering rules are generated (auto-

*This research is supported in part by an ONR grant N000140110967 and an NSF grant CCR-0208877.

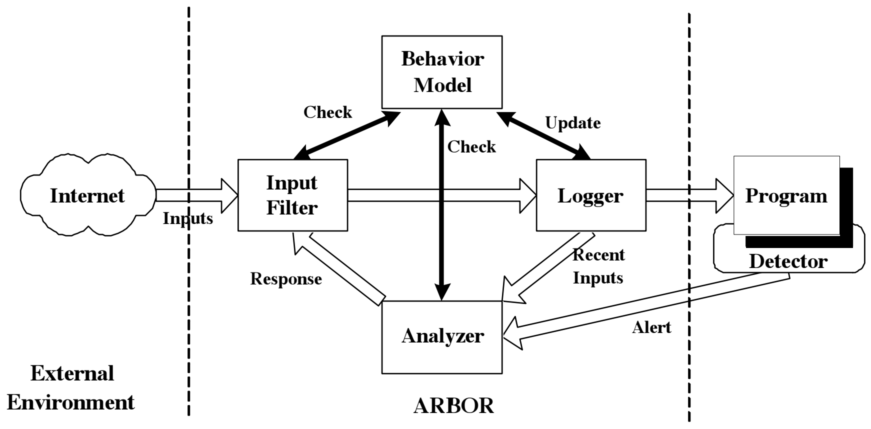

Figure 1: Architecture of our approach, which is a protective layer between the protected program and the external environment.

matically) by the *analyzer* component, and are intended to capture characteristics that distinguish attack-bearing inputs from benign ones.

The *behavior model* is a central component that is consulted by all components of ARBOR. It is constructed from events, such as system calls and other relevant library calls, intercepted by the logger. The model takes the form of a finite-state automaton that is similar to a control-flow graph of a program, except that (a) it records only those events reported by the logger, and not the internal program actions, such as assignments and jumps, and (b) it captures only those program paths that were actually traversed. The behavior model is based on our earlier work [7]. The behavior model provides contextual clues that form the basis of the filter generation logic in the analyzer component, as well as provides the tests made by the input filter.

The *logger*, like the rest of the components, is implemented using library call interposition. (Instead of system call interposition, we use library interposition because of its low performance overhead. Library interposition also provides adequate security for our purposes, since we are interested in program behaviors before it is compromised.) It records a log of input events that is used subsequently by the analyzer. In order to reduce space as well as time overheads due to logging, only a subset of data is sampled and logged.

The *detector* is external to ARBOR. Our current implementation uses address obfuscation [2] to build the detector. With the defense of address obfuscation, each buffer overflow attempt will be turned into a server crash. We intercept this crash event to trigger the feedback loop in ARBOR.

The *analyzer* is responsible for synthesizing filters that distinguish attack-bearing inputs from benign ones. The analyzer resides in a separate process from the protected process, collecting recent program behavior events from the logger. If the protected process is attacked, the analyzer will be notified by the detector. Upon receiving a notification, the analyzer examines recent inputs to the protected process to identify attack-bearing inputs. The process of analysis is described in greater detail in the following section.

2.1 Automatic Identification of Attack-bearing Inputs

Given the nature of buffer overflow attacks, a basic characteristic for filter generation come to mind — length of input data, as buffer overflow attacks are usually associated with receiving inputs that are longer than what is expected by the program. But length of input alone is not sufficiently powerful to identify inputs of typical buffer overflow attacks, because an attacker only needs to provide an input longer than the normal input for that buffer, not necessarily longer than all other inputs to the program. Therefore, the attack-bearing input may not be distinguishable among all the inputs. However, if we compare the inputs serving the same purpose, the length difference between attack-bearing inputs and benign inputs will likely become more obvious. In our approach, we use the program's execution context of input operations to provide us "hints" about the purpose of an input. The problem, then, is how to extract context information from a program's execution.

One of the key insights in this paper is that we can infer relevant context information by observing the actions of the protected process, e.g., the execution path taken by the program, the contents of runtime stack at the point of input operation, parameters to input operation. We treat a process as a state machine, which makes transitions from one state to another. The transitions are made on library calls. In our approach, we use the location in the program from which library calls are made to represent states of the state machine. The state machine provides a context of the process's current operation. In addition, we may rely on context information observed near the input operation in question. This leads to two major types of contexts: *current context* and *historical context.*

Current context is the program state at the point of input, which helps to distinguish a specific input operation from others made by the protected program. In our implementation, current context is defined by the location in the program from which the input operation was invoked, and a sequence of return addresses on the program's stack. The return addresses provides information about how the current operation is invoked. It is particularly helpful when the program uses a centralized input handler function that is called from multiple places in the code, and this function in turn invokes the actual input operation. In this case, the calling location for the input function may always remain the same, but the sequences of return address on the stack are different.

With the current context, the analyzer synthesizes a filter rule matching the attack as follows. For each suspicious input, the analyzer first identifies its current context C, and then retrieves the input statistics under context C, which is maintained by the logger during the program's normal execution. If in this context, the size a of the suspicious input is significantly larger than the maximum size b_{max} that has ever been seen during normal execution, we report this input as an attack-bearing input. The synthesized filtering rule is simply one that flags an attack if the input size is larger than the geometric mean of a and b_{max}.

If the context of all input operations made by a program are identical, then the above approach may fail to distinguish between attack-bearing and benign inputs. In this case, we extend our approach to use *historical context*, which takes into account the program paths that were taken prior to the input operation.

2.2 Light-weight Recovery After Discarding Attack-bearing Inputs

After discarding input, it is necessary for the server process to take recovery actions, so that it can get ready to serve future requests. However, it is difficult to *automatically* discover the set of actions to be taken. To address this problem, we observe that networked servers expect and handle transient network errors, which can cause input operations to fail. We leverage this error recovery code to perform the necessary clean up actions. Specifically, whenever ARBOR drops attack-bearing input, it changes the return code of the input operation to an error code associated with a network error. This error code causes the server to invoke its recovery code, including freeing of resources allocated to process the client request and so on. We have found this approach to work successfully in all of our experiments.

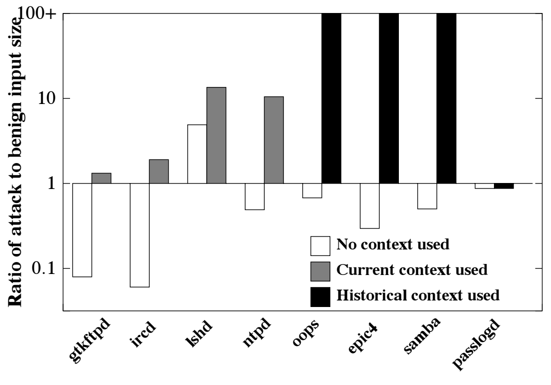

Figure 2: Effectiveness of ARBOR in synthesizing size-based filters against buffer overflow attacks. A value less than one indicates that ARBOR is not able to distinguish attack-bearing inputs from benign input.

3 Preliminary Results

We used ARBOR to defend several programs against remote buffer overflow attacks. Our focus was on "real" buffer overflow attacks, so we examined buffer overflow attacks reported on securityfocus.com during 2001–2003. We found eight attacks with working exploit code.

Figure 2 shows the results obtained with these programs, which are ratios of attack-bearing input size and maximum benign input size. A ratio less than one indicates that ARBOR cannot distinguish attack-bearing input from benign input under the program context used. The results are divided into three groups according to the context involved in the synthesized filters. In the first group (gtkftpd, ircd, lshd, ntpd), current context (specifically, the program location from where the input operation was called) was enough to generate an effective filter. The programs in the second group (oops, epic4, samba) received several types of inputs from a single location, so current context was not sufficient to synthesize filters. However, by using historical context, ARBOR was able to create an accurate filter that discarded subsequent attacks. The third group (passlogd) consists of a single program, for which ARBOR cannot successfully generate a filter. The buffer overflow in the third group is caused by part of the input, which cannot be identified by the overall length of the input.

Importance of context information As we can see from the figure, when context information is not involved, only 13% (1 out of 8) of attacks can be identified, in which the attack-bearing input is so huge that it is larger than all other inputs. After current context is involved, 50% of the attacks can be identified and filtered out. If both current context and historical context are used, ARBOR can generate effective filters for 88% of the attacks.

As a final point, we note that we restricted ourselves to length-based filters as a way to stress our approach for inferring and using program context in filtering rules. By using other criteria, such as input character distributions, in conjunction with length-based and context-based filtering rules, the approach can be made even more effective.

4 Related Work

Shield [9] is also aimed at filtering out network-based attacks on servers. Whereas our approach synthesizes filters automatically for a subclass of attacks (buffer overflows), Shield is based on manually developed filtering rules to address a broader range of attacks.

Automatic patch generation [8] attempts to deal with rapidly propagating worms by automatically generating a patch to fix the vulnerability being exploited by the worm. The primary differences with our approach are that [8] uses a more complex generate-and-test approach to diagnose the vulnerability exploited in an attack, requires source code of the protected server, plus an isolated, sandboxed duplicate of the protected server to test the correctness of the patch.

The *HACQIT* project [5] takes an alternative approach that combines software diversity with content-filters deployed at the network level. Attacks are suspected when two implementations of the same software yield different results on the same input. A rule-based algorithm is used to learn characteristics of inputs suspected of containing attacks, and generate a filter to discard such requests in the future at the firewall. This algorithm has been shown to work against Code Red worm. However, it is not clear how the algorithm can be generalized to deal with all types of buffer overflow attacks.

Failure-oblivious computing [6] is a source code transformation approach that can also recover quickly from attacks. It detects out-of-bounds write accesses, and directs them to a different (free) memory area. Subsequent out-of-bounds reads to the same area return the data that was previously written. Other out-of-bounds reads return carefully chosen values, e.g., all zeroes. The main strength of this approach is that it reliably detects the root cause of the problem. Its weaknesses are the high overheads required for memory error detection, often exceeding 100%; and the possibility that processing the attack input may cause the program to fail and/or crash, although their experiments indicate that is atypical.

5 Discussion

Our preliminary results demonstrate the feasibility of developing self-healing approaches to protect servers from buffer overflow attacks. Unlike previous approaches that were focused mainly on preventing a system compromise, our approach is able to provide complete immunity, in the sense that attacks don't even have a performance impact on the victim server.

While the approach is very effective on existing attacks, it does have some weaknesses that may be exploited in attacks specifically designed to fool it. For instance, attacks may be delivered through a sequence of small packets, causing each input operation to return a small amount of data. We are developing techniques to deal with this problem by assembling together the data returned by successive input operations, and developing filters based on this assembled data rather than the data returned by individual input operations.

A second problem concerns attacks where the buffer overflow is triggered by a field in the request, and not the entire request. Therefore, the length of input is not a characteristic of the attack. In this case, we need to identify the vulnerable field in the input, and use the identification information to increase the accuracy of program contexts. We are currently investigating techniques to handle such attacks.

References

[1] The pax team. http://pax.grsecurity.net.

[2] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of 12th USENIX Security Symposium*, August 2003.

[3] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of 7th USENIX Security Conference*, January 1998.

[4] H. Etoh and K. Yoda. Protecting from stack-smashing attacks. Published on World-Wide Web at URL http://www.trl.ibm.com/projects/security/ssp/main.html, June 2000.

[5] J. C. Reynolds, J. Just, L. Clough, and R. Maglich. On-line intrusion detection and attack prevention using diversity, generate-and-test, and generalization. In *Proceedings of the 36th Hawaii International Conference on System Sciences*, 2003.

[6] M. Rinard, C. Cadar, D. Roy, and D. Dumitran. A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors). In *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC)*, December 2004.

[7] R. Sekar, M. Bendre, P. Bollineni, and D. Dhurjati. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of IEEE Symposium on Security and Privacy*, May 2001.

[8] S. Sidiroglou and A. D. Keromytis. Countering network worms through automatic patch generation. Technical Report CUCS-029-03, Columbia University Department of Computer Science, 2003.

[9] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of the 2004 conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, August 2004.