Post–Patch Retraining for Host–Based Anomaly Detection

Michael E. Locasto
Dept. of Computer Science
Columbia University
locasto@cs.columbia.edu

Gabriela F. Cretu
Dept. of Computer Science
Columbia University
gcretu@cs.columbia.edu

Shlomo Hershkop
Dept. of Computer Science
Columbia University
shlomo@cs.columbia.edu

Angelos Stavrou
Dept. of Information and Software Engineering
George Mason University
astavrou@gmu.edu

Abstract

Applying patches, although a disruptive activity, remains a vital part of software maintenance and defense. When host-based anomaly detection (AD) sensors monitor an application, patching the application requires a corresponding update of the sensor's behavioral model. Otherwise, the sensor may incorrectly classify new behavior as malicious (a false positive) or assert that old, incorrect behavior is normal (a false negative). Although the problem of "model drift" is an almost universally acknowledged hazard for AD sensors, relatively little work has been done to understand the process of re-training a "live" AD model — especially in response to legal behavioral updates like vendor patches or repairs produced by a self-healing system.

We investigate the feasibility of automatically deriving and applying a "model patch" that describes the changes necessary to update a "reasonable" host-based AD behavioral model (i.e., a model whose structure follows the core design principles of existing host–based anomaly models). We aim to avoid extensive retraining and regeneration of the entire AD model when only parts may have changed — a task that seems especially undesirable after the exhaustive testing necessary to deploy a patch.

1 Introduction

Software systems have large, complex codebases that contain a number of vulnerabilities. Patching these errors in anticipation of or in response to attacks that exploit them remains one of the primary methods of defense for software systems. Patching, however, is a disruptive activity. Patches have the potential to change system behavior in unanticipated ways. Therefore, administrators and system owners must thoroughly vet patches before deployment: a potentially expensive and time–consuming activity. In essence, a patch affects not only the program that it is applied to, but also the environment (including other software, data, and defense systems) in which the program resides.

Defense systems that rely on measuring an application's behavior profile may be misled by the patch's deployment. For example, if a host–based anomaly detection (AD) system monitors an application, the AD must rebuild (*i.e., patch*) its behavioral model in response to changes the patch introduces. Even though most work on anomaly detection acknowledges the value of keeping a model updated, relatively little attention has been paid to the question of summarizing and communicating these changes to a sensor in a manner enabling precise and automated model updates.

This paper highlights the general problem of updating an application's environment in response to changes enacted by a self-healing *or* a reactive patching mechanism. In order to make the discussion concrete, however, we examine the feasibility of a procedure that amends an AD behavioral model in response to security–critical repairs.

1.1 AD Model Retraining

To the best of our knowledge, the two main current approaches to the problem of AD model retraining include: (1) fully retraining the AD sensor, and (2) incorporating a mechanism for gradual, online retraining into the AD algorithm itself. The first choice represents a significant increase in the length of the recovery process, and a patient attacker can exploit the latter. Instead, our goal is to harness the fact that patches, especially security–related ones, cause small, localized changes in the underlying AD model. Therefore, if we can provide an automated mechanism that efficiently incorporates these changes into the existing model, we can

avoid a lengthy and attack–prone retraining phase.

1.1.1 Complete Retraining

The first approach seems unsatisfactory because retraining the model may take significant amounts of time, and it represents an additional burden on sysadmins. Long training phases (usually necessary to capture the breadth of a complex application's behavior) impose additional delay on the patch deployment process (sometimes hours or days of downtime). If a patch is generated and deployed automatically (due to a self–healing or automatic defense mechanism), delays introduced by a long retraining period appear to defeat one of the main purposes of automated defense: the ability to respond with little or no human supervision at speeds comparable to that of the attack. Also, this phase may simply relearn large amounts of behavior that have *not* changed. Unfortunately, these problems may discourage sysadmins from employing an AD sensor in the first place.

1.1.2 Gradual Retraining

The latter approach (*i.e.,* online, gradual retraining) is most often employed to counter the problem of "model drift", in which the AD system makes an assumption that the monitored environment will display new normal behavior over its lifetime (*i.e.,* behavior that was not captured during the training period). For example, user behavior and access patterns can change in response to social demands not anticipated by the authors of the training phase. Thus, the AD algorithm continuously incorporates new data (*e.g.,* input data such as network traffic or data summarizing behavioral patterns, such as sequences of system calls) into the model of "normal." Consequently, a very patient attacker can gradually retrain the AD to accept data or behavior that would otherwise be considered malicious.

Rather than tackle this problem by automatically "cleaning" the input data or model [13] of this malicious influence (an important open problem), we propose altering the model only in response to legal, sanctioned, and (presumably) non–malicious changes in an application's behavior.

1.1.3 Spot Retraining

Sanctioned updates like patches are one cause of "model drift." AD systems need to distinguish between sanctioned and unsanctioned changes to the application they monitor. It seems as if most security–critical patches enact small changes to the system that only affect or invalidate correspondingly small parts of an application's behavioral model. If this hypothesis is correct, then it seems possible to construct an AD model update procedure that *derives the necessary changes from the text of a patch itself*[1]. The key challenge is to notify the AD about a patch in terms that it understands: changes in control and data flow. This challenge is the essence of automatic AD model retraining.

In this paper, we explore what can be done to automatically derive and deploy such a "model patch" by examining the data and control flow changes resulting from 11 security–critical patches. If the actions expressed within a patch can be captured in a standard summary of control and data flow changes, then that information can be used to update the AD model automatically. As a small example, if a patch changes the control flow so that a function `foo()` no longer calls function `bar()`, then an update procedure should remove predictive relationships involving `foo()` and `bar()` from the model.

We somewhat conflate the notion of a "repair" constructed by a self-healing mechanism with traditional source–level or binary–level patches. We do so to remain agnostic to the particular repair technique and not depend on an arbitrarily–chosen self–healing mechanism. Viewing self–healing repairs as patches helps make our techniques applicable to normal reactive patching mechanisms (*e.g.,* Patch Tuesday) as well[2]. An automated method for patching an AD sensor's behavioral model is ideal for use in self–healing and automatic update systems to reduce the amount of manual effort required to deploy repairs.

2 Related Work

AD sensors provide one mechanism for detecting the presence or activation of malware on a host by observing a shift in an application's execution profile. This conclusion rests on the assumption that malicious inputs rarely occur during normal operation. However, since a system can evolve over time, it is likely that new *non–malicious* inputs will be seen [16]. The seminal work of Hofmeyr, Somayaji, and Forrest [20, 26] helped initiate application behavior profiling at the system call level [12, 25, 17, 19] because that interface represents the services that malcode, once activated, must use to effect persistent changes and other forms of I/O. In particular, the malware may begin to use system services that the application has not previously invoked, or it may employ the set of already–used services in new ways (*e.g.,* via new arguments [23] to those calls). Such information is now easy to collect; the `strace` and `ltrace` tools for Unix perform exactly this task. As a basis for our work on patching an AD model, we propose a

[1] We can utilize the actual patching procedure to recover the context of the changes and a limited form of parsing or symbolic execution to gather information about data flow changes.

[2] Ironically, binary patches are often obfuscated to frustrate an attacker's attempts to reverse–engineer the vulnerability. Dealing with this type of patch is an interesting technical challenge. Section 4 limits our evaluation to source–level patches.

hybrid model that follows the form and content of this previous work. The model captures aspects of both control flow (via the execution context) and data flow (via return values and arguments).

Anomaly sensors [30, 21] classify network inputs as potentially malicious without relying on static signatures. Content–based approaches may work against slow and stealthy worms, but not all polymorphic ones, since the analysis is often fixed on specific byte patterns. Indeed, some work [15] has illustrated the evasion of anomaly–based classifiers, and Taylor and Gates suggest that anomaly detection (as originally conceived for host–based monitoring) is ill–suited for current network traffic [28].

Most publications suggest updating the model after significant changes to the environment, data stream, or application. Little work, however, has been done to show the feasibility of such a process in any practical fashion beyond retraining the entire system from a known clean training set. It seems that anomaly detectors would benefit from an additional source of information that can confirm or reject the initial classification, and Pietraszek [24] suggests using human–supervised machine learning for such tuning. Although researchers [22, 11] suggest updating a network content–based anomaly sensor in a reactive fashion (that is, based on confirmation of an attack from a highly precise, host–based detector) little work to sketch an update procedure, experimentally validate the claim, or evaluate the utility of such action has been done.

3 Updating an AD Model

A variety of specific AD models, training algorithms, and testing algorithms exist. Many models, however, express the same basic structure and contain similar types of information. Despite this variety, our goal is to provide a mechanism that provides enough information to update these models given minor tweaks, translation proxies, or shim code to adjust for the syntactic differences between the model patch generator and the model in question.

Notwithstanding these differences, the key problem is translating from a static description of what behavior *could* occur (as expressed by the patch) to whatever dynamic description of events is contained in the model. In order to make this discussion concrete, however, we define a basic, straightforward context window anomaly model that contains aspects of both data and control flow.

The model employs a context of m function instances to predict the occurrence of other function instances. That is, the model can be logically represented as a table of entries of the form: $\{f_i(args_i, rval_i), \ldots, f_j(args_j, rval_j\} \rightarrow \{f_k(args_k, rval_k)\}$. The conditional probability of f_k occurring with a particular set of arguments $args_k$ and return values $rval_k$ is based on the proceeding context of m (which can vary) functions[3]. The simplest case is based on monitoring only sequences of system call names or numbers for each process. Modeling both the system calls and the arguments to those calls allows the model to improve its granularity. In addition, we can model library and application function calls and their parameters. One way to model the relationship between calls and arguments is to calculate the aggregated conditional probabilities between specific calls and arguments [27].

While it may be fairly straightforward to adjust control flow based directly on the information contained in a patch (*e.g.,* an insertion or removal of a function call), characterizing changes to the data sets representing the arguments or return values represents a more challenging task, and some pathological cases exist. For example, the dynamic behavior of a patch might be such that the application processes a completely different distribution of input data or produces radically different output data. Entries in the model for functions that process such data may now have outdated character distribution models or constraints for their arguments. Arbitrary and widespread behavioral changes will likely perturb the model beyond our ability to micro–patch it. In these cases, simply retraining by replaying a "clean" input archive may represent the best option.

As in Section 1, we make the simplifying assumption that security–critical patches do not widely perturb the model or constraints on data arguments. Our examination of patches in Section 4 bears this hypothesis out. However, in cases where dataflow does drastically change between "known" data distributions, we may be able to automatically or manually annotate the model patch with these change types. A model patch can be bootstrapped from the patch text, then improved manually or via symbolic execution — in this case, manually improving the model patch and applying it will still likely result in a faster update of the model rather than complete retraining.

4 Evaluation

Our evaluation contains two assessments. First, we review some anomaly and specification–based sensors to discover their supervised or unsupervised training time. We do so to confirm our hypothesis that such systems have relatively long training periods that occasionally require significant user input or supervision. In some cases, including Systrace [25] and modern PC firewalls that employ a user–driven training mode, supervision can span hours or weeks. Second, we summarize the changes in data and control flow enacted by a series of security–critical patches.

[3] Although we restrict our examination to a host–based model, examining the impact that patches have on *n*–gram based network content models is an interesting area for complementary work.

4.1 Cost of Training

The training phase represents a crucial component of most AD systems. The detection performance of an AD sensor depends significantly on the training data set's quality, labeling (or lack thereof), and size, as well as the particular learning method (supervised or unsupervised) in use. Much research on both network and host–based sensors indicates that some minimum training requirement (defined in terms of requests, runs, or packets) exists before the model quiesces and becomes usable to the sensor [14, 25, 18, 29]. In most cases, increasing the length of the training phase can boost detection performance.

We can classify training cost based on two broad categories: supervised and unsupervised. An unsupervised training phase usually requires several thousand packets or requests. Moreover, some host–based detection systems impose an additional one–time overhead due to static analysis. Furthermore, when a host–based sensor employs dynamic analysis, there is also a per–request latency that can lead to several hours of offline training. Supervised AD systems require user input to drive the training process. In such systems, it is very difficult to quantify the effort required to train the system since it depends on user activity. It is clear, however, that such systems require input from multiple users over a long period of time [25, 18] before they can generate a normality model capable of differentiating between normal and abnormal behavior.

4.2 Security Patch Survey

A patch can affect a behavioral model by changing either or both the control and data flow. Examples of changes in control flow include updating, removing, or introducing new decision control structures; introducing a new child function; or inserting a new parent function (*e.g.,* a sanity check on input parameters). Changes in data flow include adding new variables or symbolic values; adding or removing arguments or function parameters; and modifications to the set of possible return values. We note that our examination is strictly static: it does not attempt to execute the patches or otherwise determine if the code contained in them is ever actually executed. In addition, we don't distinguish between macros and function calls.

Table 1 lists our results for a variety of applications, including stunnel [1], some web servers [2, 3, 4], linux [5], cvs [6] and fetchmail [7], as well as various vulnerabilities in libpng [8], Firefox [9], and Samba [10].

4.3 A Model Update Procedure

Most of the control flow changes we observed result from invocations of new functions as well as the insertion of new `if` statements or updates of `if` conditions. Most data flow changes involve new arguments to function calls, or new ways of wrapping those arguments, as well as new `return` statements that introduce new values. A majority of the patches we examined made very minor changes; for example, the patch to ghttpd substitutes the use of a "safe" library function and derives the value of a new argument for that call. The patch for nullhttpd introduces a new `if` statement and condition with a call to an application function to log an error (presumably, the dynamic behavior also involves the invocation of the library `printf()` family of functions and the `write()` system call). We can use a parsing and symbolic execution phase to learn and summarize these implicit changes. We envision generating model patches in a format similar to source code patches like those produced by `diff`. Model patches contain update summaries to the conditional probability entries in the model, along with changes to the format of the arguments and insertion and removal of functions from a call chain.

5 Conclusions

Applications contain vulnerabilities that, for the foreseeable future, can only be addressed by post–deployment reactive patching and self–healing strategies; we cannot excise all vulnerabilities prior to deployment. Patching and self–healing, whether automated or manual, are disruptive activities for the application and its environment, especially when the environment contains a host–based AD sensor. After a repair or patch, the sensor's model may be outdated.

We examine 11 security–critical patches to obtain an idea of how to summarize the data and control flow changes necessary to update a behavioral model. In follow–on work, we will define a "model patch" description language (MPDL), specify an algorithm for automatically deriving an MPDL script from the text of a patch, and build a tool that executes the AD update procedure by running an MPDL script. We expect that the end–to–end automation of AD model updates can ease the workload on overburdened sysadmins when deploying a patch or self–healing repair.

References

[1] http://www.securityfocus.com/bid/3748.
[2] http://www.securityfocus.com/bid/5960.
[3] http://www.securityfocus.com/bid/5774.
[4] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-3747.
[5] http://www.sfu.ca/~siegert/linux-security/msg00047.html.
[6] http://www.us-cert.gov/cas/techalerts/TA04-147A.html.
[7] http://fetchmail.berlios.de/fetchmail-SA-2005-01.txt.

Table 1. *Survey of Patches*. We list the vulnerable version of an application, the size of a patch in lines (including comments), and the changes in data and control flow introduced by the patch, as listed above. The magnitude of the difference between the changes and the application's total size supports the notion that patches introduce relatively confined model updates.

Application	Patch Size (lines)	control flow Δ	data flow Δ
Linux-2.4.19	20	3	1
ghttpd-1.4	16	4	5
nullhttpd-0.5.0	12	2	1
stunnel-3.21	29	0	3
libpng-1.2.5	98	10	12
cvs-1.11.15	81	1	2
Apache-1.3.24	11	0	1
fetchmail-6.2.0	183	1	5
Samba (CVE-2004-0882)	65	0	7
Samba (CVE-2004-0930)	386	99	39
Firefox-2.0.0.3	22	8	0

[8] `http://www.us-cert.gov/cas/techalerts/TA04-217A.html`.

[9] `http://www.mozilla.org/projects/security/known-vulnerabilities.html`.

[10] `http://us4.samba.org/samba/history/security.html`.

[11] K. G. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. D. Keromytis. Detecting Targeted Attacks Using Shadow Honeypots. In *Proceedings of the* 14th *USENIX Security Symposium.*, August 2005.

[12] S. N. Chari and P.-C. Cheng. BlueBoX: A Policy–driven, Host–Based Intrusion Detection System. In *Proceedings of the* 9th *Symposium on Network and Distributed Systems Security (NDSS 2002)*, 2002.

[13] G. F. Cretu, A. Stavrou, S. J. Stolfo, and A. D. Keromytis. Data Sanitization: Improving the Forensic Utility of Anomaly Detection Systems. In *Workshop on Hot Topics in System Dependability (HotDep)*, 2007.

[14] H. H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly Detection Using Call Stack Information. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, May 2003.

[15] P. Fogla and W. Lee. Evading Network Anomaly Detection Systems: Formal Reasoning and Practical Techniques. In *Proceedings of the* 13th *ACM Conference on Computer and Communications Security (CCS)*, pages 59–68, 2006.

[16] S. Forrest, A. Somayaji, and D. Ackley. Building Diverse Computer Systems. In *Proceedings of the* 6th *Workshop on Hot Topics in Operating Systems*, pages 67–72, 1997.

[17] D. Gao, M. K. Reiter, and D. Song. Gray-Box Extraction of Execution Graphs for Anomaly Detection. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2004.

[18] D. Gao, M. K. Reiter, and D. Song. Behavioral Distance for Intrusion Detection. In *Proceedings of the* 8th *International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 63–81, September 2005.

[19] J. T. Giffin, D. Dagon, S. Jha, W. Lee, and B. P. Miller. Environment-Sensitive Intrusion Detection. In *Proceedings of the* 8th *International Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2005.

[20] S. A. Hofmeyr, A. Somayaji, and S. Forrest. Intrusion Detection System Using Sequences of System Calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[21] C. Kruegel, T. Toth, and E. Kirda. Service Specific Anomaly Detection for Network Intrusion Detection. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, 2002.

[22] M. E. Locasto, K. Wang, A. D. Keromytis, and S. J. Stolfo. FLIPS: Hybrid Adaptive Intrusion Prevention. In *Proceedings of the* 8th *International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 82–101, September 2005.

[23] D. Mutz, F. Valeur, G. Vigna, and C. Kruegel. Anomalous System Call Detection. *ACM Transactions on Information and System Security*, 9(1):61–93, February 2006.

[24] T. Pietraszek. Using Adaptive Alert Classification to Reduce False Positives in Intrusion Detection. In *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2004.

[25] N. Provos. Improving Host Security with System Call Policies. In *Proceedings of the* 12th *USENIX Security Symposium*, pages 207–225, August 2003.

[26] A. Somayaji and S. Forrest. Automated Response Using System-Call Delays. In *Proceedings of the* 9th *USENIX Security Symposium*, August 2000.

[27] S. J. Stolfo, F. Apap, E. Eskin, K. Heller, S. Hershkop, A. Honig, and K. Svore. A Comparative Evaluation of Two Algorithms for Windows Registry Anomaly Detection. *Journal of Computer Security*, 13(4), 2005.

[28] C. Taylor and C. Gates. Challenging the Anomaly Detection Paradigm: A Provocative Discussion. In *Proceedings of the* 15th *New Security Paradigms Workshop (NSPW)*, pages 21–29, September 2006.

[29] K. Wang, J. J. Parekh, and S. J. Stolfo. Anagram: A Content Anomaly Detector Resistant to Mimicry Attack. In *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2006.

[30] K. Wang and S. J. Stolfo. Anomalous Payload-based Network Intrusion Detection. In *Proceedings of the* 7th *International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 203–222, September 2004.