Automated Self-Adaptation for Cyber-Defense - Pushing Adaptive Perimeter Protection Inward

Brett Benyo, Partha Pal, Richard Schantz, Aaron Paulos
Raytheon BBN Technologies
Cambridge, USA
{bbenyo, ppal, schantz, apaulos}@bbn.com

David J. Musliner, Tom Marble, Jeffrey M. Rye, Michael W. Boldt, Scott Friedman
SIFT LLC
Minneapolis, USA
{musliner, tmarble, jrye, mboldt, sfriedman}@sift.net

Abstract— **This paper presents a recently achieved incremental milestone on the long path toward more intelligently adaptive, automated and self-managed computer systems. We demonstrate the feasibility of integrated cyber-defense connecting anomaly detection and isolation mechanisms operating at different system layers with two complementary mediation policy adaptation techniques in service of automatic remediation against observed attacks and their future variants. We describe a number of experiments evaluating the relevance and effectiveness of the integrated cyber-defense operation.**

Keywords-adaptive defense, survivable application, resilience

I. Introduction

Traditionally, cyber-defense meant protecting perimeter routers and entry points at the network border. Defense also implied static preventive policy enforcement, for instance in the context of perimeter defense, filtering firewalls. The prevalence of preventive protection at the perimeter, and in some cases being the only defense available in the system, has led to the infamous *crunchy on the outside, soft inside* description of computer systems, implying that once an adversary compromises the outside barrier, he has free reign.

Like-minded researchers including ourselves, have been working on changing that perception on both counts. First, cyber defense is no longer confined to static and prevention focused security policy enforcement. Adaptive response plays a major role in an organization's cyber-defense strategy. Second, there is no single defensive boundary anymore; defensive layers permeate the entire system.

Major R&D efforts such as the DARPA CRASH and MRC programs are developing new defensive technologies that have the potential to become fundamental building blocks across all system layers, including the hardware, OS, programming languages and middleware-based execution management environments. To achieve better preparedness against the growing threat and increasing sophistication of cyber-attacks, the R&D trend has been to push the envelope on adaptive security—developing new types of adaptation mechanisms with deeper and wider scope, smarter and more effective management of defensive adaptation, as well as pushing the perimeters to be defended inward. Part of that has meant attempting to move from perimeter firewalls at the network border to distributed managed firewalls at individual hosts, and moving up the application stack in the form of application specific filtering.

We recently showed the feasibility and practicality of integrated components of a managed execution environment mounting automated post-incident responses to immunize a protected application against future attacks of the same or similar kind. Attacks are launched through the application's network interface and immunity is achieved by principled adaptation of its I/O mediation policy. Attack effects are detected by undesired I/O and execution behavior. The application is restored to a pre-attack state, a quick policy patch in the form of decision-tree classifiers is followed up by detailed fuzzing experiments to derive a more precise model of the input that triggers the exploited vulnerability. The resulting regular expression is then applied as a second policy patch. By utilizing the two complementary patch generation technologies, the resulting updated policy is able to block future I/O events that are similar in *content signature* (e.g., regular expression filter) or *in character* (i.e., classifier that separates the manifestation causing events). The post-incident workflow described above is implemented in the A3 execution management environment we are developing and accepts operator intervention as necessary. This technique covers novel attacks that eventually manifest a known undesired I/O or execution state, under the conditions that (for the current implementation) (1) at least a subset of the I/O events contributing to the manifestation occur after the last check pointed state, and (2) the attack does not exploit timing flaws.

This paper concentrates its contributions in three areas. First, we present a high-water mark working prototype for defense-oriented self-adaptation, demonstrating automation of complex multi-stage post-incident response, closing the loop between detection at the victim's execution state and patching at the processes I/O perimeter. This significantly reduces the time required to devise and deploy a patch, and in turn reducing the time the application remains unpatched and vulnerable. Second, we establish the effectiveness of pushing the I/O filtering perimeter defense inward to a

This work is being supported by the United States Air Force and DARPA under Contract Nos. FA8750-10-C-0242 andFA8650-10-C-7087. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

boundary point near an application, without interfering with other defenses or other computations running on the same host. Third, we demonstrate an application-centric open architecture platform for cyber-attack management, integrating synergistic capabilities from multiple external technologies like FuzzBuster's input fuzzing and the Weka machine-learning framework with A3 core services like replay and execution introspection to realize robust automated cyber-defense and resiliency objectives. We discuss initial experimental evaluation of the post-incident policy patching capabilities using real as well as injected attacks against an illustrative Apache-PHP application.

II. Building to the New Capabilities

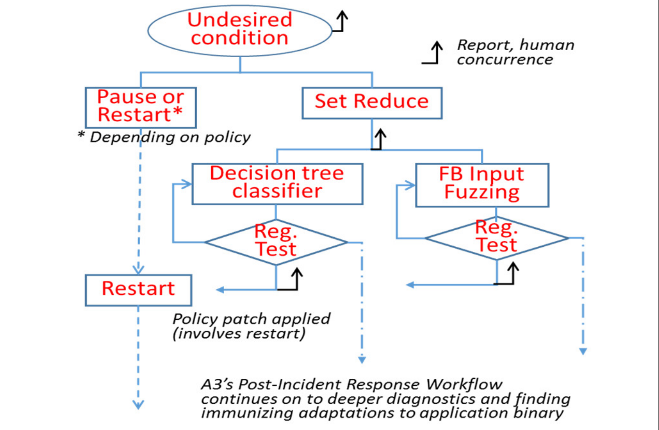
Our approach to automated adaptation of near-application perimeter protection is informed by our past experience in developing and demonstrating advanced adaptive cyber defense technologies. In the DPASA project [1] we divided a high-valued network into four distinct quads so that 4-fold Byzantine tolerant replicated services can have each replica in different quads, and used hardware voting to cut off suspected quads that showed abnormal behavior detected by IDSs and the BFT protocols. Beyond introducing a 2 layer hierarchy (i.e., the entire network vs. 4 independent quads), we also used managed distributed firewalls [2] running on the network interface cards of each host. The distributed firewalls enabled fine grained control of the inbound and outbound network traffic at each host, based on peer IP address, local and remote port and protocol. In the CSISM project [3] we advanced this type of adaptive and managed perimeter defense further by adding a sophisticated reasoning component to make blocking and quarantining decisions based on interpreting alerts and lifecycle events as accusations and evidence. Around the same time, the CORTEX [4] project developed a *taste-tester* framework to provide intrusion-tolerant database services. In response to detected malicious SQL requests, CORTEX would develop a generalized SQL filter that will block future SQL requests that attempt to exploit the same vulnerability. The FuzzBuster project has developed a more general approach to application-level protection, using fuzz-testing to refine vulnerability models and develop adaptations to prevent future attacks. AWDRAT and PMOP [8] demonstrated use of library reloading, alternative methods and rebuilding application data structures as a means to defend against future instances of observed attacks. These adaptations were implemented by wrappers interposed between system and library calls, and are similar in spirit to A3's I/O mediation.

Many of the precursor technologies, unless it is integrated with a specific application, such as the database in CORTEX, suffer from co-tenancy at the network enclave level and at the host. The subtle ways the co-tenants interact and interfere, knowingly or unknowingly, through shared system resources like the network and stable storage, make enforcement of application-specific policy at the network or host level, let alone dynamic adaptation of such policies, much more difficult and less effective. The policy enforcement points handle the aggregate interaction of all tenants and what is good for one is typically not so good for another. Even if a workable coarse grain policy is found after careful planning and profiling, flaws and gaps are inevitably found by novel and zero-day attacks, necessitating changes in the policy, entailing lengthy rounds of profiling and testing, during which the applications remains vulnerable.

The A3 execution management environment addresses this problem by containerizing the protected application in such a way that (a) the protected application has an entire (virtual) host to itself and (b) all network and storage I/O is subjected to mandatory mediation, so that a tight application-specific policy (a) is feasible and (b) does not interfere with any other application. In this context an application means one or a collection of cooperating processes working to provide a single, well-defined service. The illustrative example we use in this paper is an Apache-PHP document management application that consists of multiple processes. The A3 environment offers a number of core services and an open architecture for integrating synergistic capabilities to facilitate self-adaptive post incident response. As we explain next, the A3 and FuzzBuster integrated operation is an example narrowly focused on adapting the application's I/O mediation, closing the loop between attack manifestations and patching the mediation policies.

III. System Under Test

Both A3 and FuzzBuster have been introduced and discussed in papers from the last AHANS workshop [5, 6]. Before describing the A3-FuzzBuster integrated operation, we briefly introduce A3's post-incident response handling. In this context, incidents are manifestation of known undesired conditions caused by potentially unknown exploits and vulnerabilities, for example, eventual detection of a remote network shell or segmentation fault. Figure 1 shows part of A3's post-incident response workflow, primarily focusing on the adaptive modification of I/O mediation policy. An observed manifestation of an undesired condition pauses or restarts the application from a past checkpoint. In parallel, A3 initiates *Set Reduce*, which partitions the recorded input events to "benign" and "malicious" sets by playing back recorded inputs in a binary-search like algorithm and watching which playback sequence reproduces the observed undesired manifestation. A policy patching step follows. If a


Undesired condition
Report, human concurrence
Pause or Restart*
Set Reduce
* Depending on policy
Decision tree classifier
FB Input Fuzzing
Reg. Test
Reg. Test
Restart
Policy patch applied (involves restart)
A3's Post-Incident Response Workflow continues on to deeper diagnostics and finding immunizing adaptations to application binary
Figure 1: The workflow to patch the IO mediation policy

patch is found, the application is restarted with the patched policy. The policy patch, acting as an I/O filter, buys time to perform a deeper diagnosis and find a more robust fix.

As shown in Figure 1, policy patching involves two complementary patch generation techniques: decision-tree classifiers and input fuzzing. The classifier generation process takes both the malicious and benign partitions of the recorded inputs, and *learns* the semantic differentiator between the two sets. The classifiers can result in a complex predicate involving attributes of the I/O events (i.e., this technique is aware of the application's I/O protocol) and their values. Input fuzzing starts with the malicious set and employs sophisticated fuzz testing techniques to find a regular expression filter. The regular expression filter is a generalized signature for the I/O payloads that trigger undesired manifestations similar to the members of the malicious set, and in that sense is an input-output model of the exploited vulnerability. These two techniques highlight the two prominent interaction patterns supported in A3's open integration architecture. The classifier generation step is performed using the Weka framework, which is invoked as an internal on-demand service by A3. In contrast, regular expression filters are generated by FuzzBuster, running as an external service independent of A3, using a standardized interface and interaction protocol, C3PO, designed to regularize exchanging fault handling information.

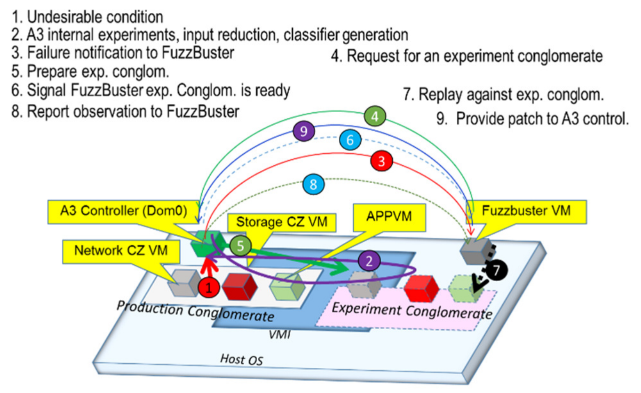

1. Undesirable condition
2. A3 internal experiments, input reduction, classifier generation
3. Failure notification to FuzzBuster
4. Request for an experiment conglomerate
5. Prepare exp. conglom.
6. Signal FuzzBuster exp. Conglom. is ready
7. Replay against exp. conglom.
8. Report observation to FuzzBuster
9. Provide patch to A3 control.
A3 Controller (Dom0)
Storage CZ VM
APPVM
Fuzzbuster VM
Network CZ VM
Production Conglomerate
Experiment Conglomerate
VMI
Host OS

Figure 2: A3-FuzzBuster integrated capability

Figure 2 shows the integrated setup combining A3 and FuzzBuster. The protected Apache-PHP application runs in its own dedicated AppVM. Network and storage interactions of the protected application are subjected to mandatory mediation at the Network and Storage crumple zones respectively, each realized as different VMs. A3 provides an onboard laboratory area where copies of the protected application (labeled experiment conglomerate) can be started. The experiment conglomerate is used for A3 internal purposes i.e., replay-based Set Reduce to partition the recorded inputs into benign and malicious sets or testing decision-tree classifiers, and is also exposed to external services for integrated operation e.g., FuzzBuster's fuzz tests. A3's VM introspection (VMI) service enables deep inspection of the memory state of executing processes, in both production and experiment conglomerates. A3 comes with a dashboard through which human operators interact with the environment controller (running in Dom 0). Only the Xen micro-kernel and Dom 0 services are trusted; the Dom U VMs are not. The mandatory mediation and deeper introspection of executing processes enable A3 to detect undesired states that the application should never be in.

Figure 2 highlights A3-FuzzBuster integrated operation, starting with the detection of a manifest undesired condition (1), isolating the failure causing inputs (2), notifying FuzzBuster (3), setting up an experiment conglomerate (4, 5, 6), FuzzBuster interaction with the experiment conglomerate for fuzzing runs (7) with A3 reporting observed undesired conditions back (8), and FuzzBuster providing a patch to A3 (9). The experiment conglomerate can be started and configured beforehand, minimizing the delay associated with the handshake in steps 4 and 6. The protected application and/or the experiment conglomerate may need to be restarted during fuzzing experiments, depending on the application and attack. For example, in the experiments and attacks considered in this paper, restarting the Apache application (instead of the entire experiment conglomerate) was sufficient for continuing the fuzzing runs. Restarting the application takes seconds, whereas restarting the conglomerate takes ~2 minutes.

The steps described above are part of the standardized C3PO protocol that enables A3 and FuzzBuster to interoperate with each other as well as other conformant external services. For A3, C3PO provides a standard way to designate a failure report to other components that may be able to produce a *patch*, while for FuzzBuster it offers a standard way to fuzz test externally managed applications. Configurable parameters of the C3PO protocol enable tuning the integrated behavior in a number of ways such as returning a patch within a deadline, incrementally refining previous patches, tolerating certain percentage of false positives or negatives etc.

To evaluate the adaptation of mediation policies as part of our post-incident response automation, we implemented seven known attacks against the Apache-PHP webserver application that targeted vulnerabilities in the Apache 2.2 source code or server configuration. These attacks lead to resource exhaustion, data exfiltration, and remote code execution. Five of these attacks can be launched using a single input message. Two require multiple message sequences, where the first message uploads a file to the webserver (exploiting a permissive file upload service), followed by sending a trigger message that exploits a vulnerability in conjunction with the uploaded file. Since fully implementing and weaponizing the attacks requires significant development, we instead used a fault injector module to inject the end-effects of the attacks. The fault injector triggers computation based on specific patterns in input messages, and the existence or contents of local configuration files used by Apache, emulating the execution the weaponized attack payload. Examples of injected attack effects include termination of the Apache request handling child processes, prolonged CPU over-utilization, memory exhaustion, access to files outside application's data folders, opening remote connections and listening sockets, etc.

We developed a set of benign clients using the system under test in a variety of usage patterns, from periodic single file downloads, to bulk batch downloading, random heavy bursts of traffic, and periodic index requests. With benign clients providing background traffic and load, we start a malicious client that waits a random interval and sends the attack sequence. When the exploits trigger, the fault injector module injects the intended attack effects. This leads the protected application to an undesired state. When such states are detected by the A3 VMI-based detectors or I/O mediation policy it automatically kicks off the workflow described in Figure 1, beginning with replay-based set reduction, followed by derivation of an appropriate adaptation of the I/O mediation policy using decision-tree classifiers and regular expression filters.

IV. Results and Discuission

To evaluate the adaptation of mediation polices, we considered response time (the time taken to generate an enforceable filter), and patch quality (susceptibility to false positives (FP) or false negatives (FN)). The time from detectable manifestation to installation of an effective filter is the time the application remains vulnerable. In the experiments, we measured this as the elapsed time between detection of the undesired condition and the availability of a patch. Details about this interval for our two patch generation approaches are explained throughout this section.

By design generated classifiers and regular expression filters have no FNs or FPs based solely on repeats of previously seen inputs from the last checkpoint. However, since attackers can and will try again if they are blocked, and new benign clients will continue to access the application it is important to understand whether the generated filters can produce FPs or FNs going forward. Potential FNs can arise if a filter keys off a portion of the input that happened to be different for a malicious input set, but was irrelevant to actually triggering the failure. A clever attacker could modify his attack by changing the irrelevant section and exploit the same vulnerability. An FP can likewise occur if a derived filter is too general based on observed inputs and a later benign request from a new client happens to match the in retrospect inadvertently blocked section.

We evaluated the potential for emergent FPs or FNs for a generated filter through a manual step after the classifier or regular expression filters are generated. Manual analysis of the generated filter against the attacks tells us whether FP or FN is possible, in which case we generated specific new input messages to cause such events. Figure 1 workflow is invoked again, adding the new false positive to the benign set, and rerunning the filter generation algorithms.

A. *Mediation Policy Adaptation Using Input Classifiers*

Since the malicious set size is generally small, often one element, there are many potential decision trees that correctly classify it. Our algorithm produces as many decision trees as possible that satisfy size constraints and correctly classify all the training data. These trees are ranked by the information gain present in the root node, as determined by the C4.5 algorithm [7]. Trees with the same information gain are ranked using a non-optimal lexicographic ranking based on feature name. Our host controller allows a human operator to assign weights to the various features based on knowledge of the defended application and the HTTP protocol, which would override the standard information gain ranking. We did not use operator assigned rankings for the evaluation avoiding having to account for human intervention in response time.

Trees with high information gain in the root node have likely found a specific feature that classifies the malicious set as different from the benign set, whereas trees with low information gain in the root node are likely over-fitting the data. When multiple decision trees with the same ranking are generated, there is a decision to be made as to which trees to enforce as input filters. Using a single tree reduces potential FPs, but increases the potential for FNs, if the chosen tree uses extraneous filters that are essentially noise instead of the features that are the underlying culprit. Alternatively, using all of the trees reduces FN potential, but increases potential FPs. Since FNs are usually more problematic than FPs and we wanted to avoid accounting for human involvement in response time, we used all trees with the highest information gain ranking in the current evaluation. In an operational environment, a long running application will have built up a large set of benign inputs. Larger benign input sets will reduce the potential for FPs, since an FP requires a new benign input with a specific feature value that hasn't been seen before. The experiments we performed used a small set of benign clients (~50 inputs), to present close to a worst case scenario for the classifier algorithm.

Attack	Time (ms)	Filters	Optimal Tree Present?	Eventual Optimal Tree
CVE-2011-3192	968	2	No (FP)	Yes (1)
CVE-2011-3368	1490	8	No (FN)	Yes (2)
CVE-2012-0053	441	7	Yes	Yes
CVE-2011-0419	550	11	Yes	Yes
CVE-2012-0021	1251	2	Yes	Yes
CVE-2011-3607	617	6	No (FN)	No (FN)
Shellcode	1203	7	No (FN)	Yes (1)

Table 1: Decision Tree Results

Table 1 gives the results of the decision tree classifiers on the seven attacks. The Time column is the time to generate a patch. The Filters Generated column indicates the number of decision trees converted to input filters, which is the number of trees with the same information gain as the top ranked tree. An optimal decision tree is one that if implemented as NCZ filters results in no false negatives (the filter will block any attempt to exploit the same vulnerability), and no false positives (benign clients will not be blocked) as determined by manual analysis. The Optimal Tree column indicates "Yes" if the optimal tree was present in the set generated, "FP", if the best tree present has potential false positives, and "FN" if the best tree present had potential false negatives. The Final column indicates whether or not the extended manual tests (where experimenter generated false negative or false positives are injected) eventually resulted in an optimal decision tree. The number in parentheses indicates the number of iterations of generating a false negative or positive and rerunning the classifier algorithm.

For six of the seven attacks, the decision tree classifiers are eventually able to produce an optimal decision tree. The CVE-2011-3368 attack involves the attacker accessing private internal servers through a URL rewriting configuration error. The optimal decision tree effectively blocks the path to the internal server being attacked. If the application has access to multiple internal servers, the classifiers would need to see one attack for each internal server before blocking all such attacks. For this experiment, we assumed three internal servers, implying that the patch resulting from the observed attack will have two FNs. Note however, that a filter that allows FNs still slows the attacker down, since the attacker would need to spend some time and luck determining exactly what message attributes are triggering the block to successfully utilize the FN opening. The CVE-2011-3607 attack involves uploading a file, adding a header defined in that file to a subsequent input, and putting shellcode as the value of that header. The decision trees can block an individual header, but the attacker could upload a new file with a new header, and a new trigger input using that new header. The shellcode attack is similar to CVE-2011-3607, however under this attack the shellcode injected into a specific header field leads to an optimal tree that blocks any use of that field. Initially that optional tree is ranked lower than blocking the specific seen shellcode, requiring a second attack (FN) or manual experimenter modification of the feature weights to bring it to top ranking.

In all but one of the attacks (CVE-2012-0021), however, the defended application is either still vulnerable after the initial attack to a modified attack, or could delay benign clients with false positive blocked requests.

B. Mediation Policy Adaptation Using FuzzBuster

For the current evaluation, Fuzzbuster was configured to return an initial patch after two minutes of operation. The initial patches are susceptible to false negatives, since they contain extraneous information from the initial malicious input that has yet to be reduced out of the patch expression. Fuzzbuster then continued to refine the patch eventually producing a final patch when it had run to quiescence.

Attack	Time (min)	Inputs Tested	FP or FN Possible
CVE-2011-3192	96	1165	FN possible
CVE-2011-3368	53	627	No
CVE-2012-0053	30	366	FN possible
CVE-2011-0419	Timed out		FN possible
CVE-2012-0021	36	444	No
CVE-2011-3607	Timed out		FN possible
Shellcode	32	1131	No

Table 2: FuzzBuster Results

Fuzzbuster results on the seven candidate attacks are given in Table 2. The first column lists the total time from initial fault detection to final patch generation. The Inputs tested column indicates the total number of synthesized input messages tested. The final column indicates whether the final patch was susceptible to FPs or FNs as determined by manual analysis of the regular expression and the attack.

To compare the patch generation time of the two techniques, we measured the time FuzzBuster takes to generate the regular expression filters. Column 1 of Table 2 shows the FuzzBuster compute time, which does not include the time spent in coordinating with A3 during fuzz testing. The major component of the additional coordination delay is the ~5 second time required to revert the experiment conglomerate to a check pointed state, if and when needed (depending on the attack being tested).

FuzzBuster timed out after four hours on the attacks involving larger input messages, with the final patches blocking the attack, but susceptible to a FN from a modified attack. For CVE-2011-3607, the generated regular expression was closing in on an optimal solution of blocking just the specific triggering bytes wherever they appear, leading us to believe that future refinements to the fuzzing algorithms can result in an optimal filter.

Attacks that depend on specific sequences of characters in the input messages, such as CVE-2011-3368 and CVE-2012-0021, are blocked perfectly by the regular expression filters, allowing no false negatives or false positives. In attacks such as CVE-2011-3192 and CVE-2012-0053, however, the content of the message is irrelevant; the length of a specific header is the cause of the vulnerability. For these the resulting final patch still contains extraneous data, exploitable to generate a false negative.

C. Discussion

We first consider the differences between the decision tree and regular expression filters. Figure 3 shows one of the highest ranking decision trees generated in response to the CVE-2011-3368 attack, allowing the attacker to access a private server that should not be accessible. The "Yes" nodes of the decision tree indicate a member of the malicious

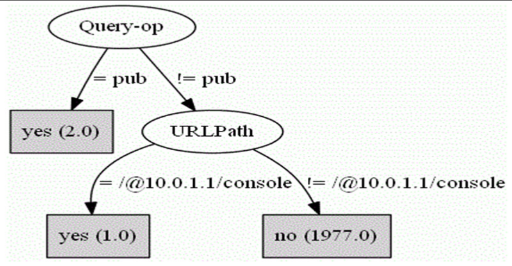


Figure 3: Decision tree filter example (CVE-2011-3368)

set. Any input request that uses the op=pub query parameter or any request that accesses the specific URL "/@10.0.1.1/console" is classified as malicious. Therefore, the policy that mediates input requests to the protected application is modified with a blocking rule with the condition `((op=pub) OR (URLPath= /@10.0.1.1/console))`. Note that the patch is aware of the HTTP protocol that the application is using for its network I/O. In contrast, the FuzzBuster generated regular expression filter for the same attack looks as follows: `".*?`

/\@10\.0.*?\n". Although not aware of any protocol, the blocking rule using this regular expression effectively blocks any use of the "/@10." character sequence in the request URL, which is the trigger that allows access to a private internal server via a URL rewrite rule misconfiguration, not just the "console" as in the case of the classifier. This regular expression filter blocks any message that matches this signature, blocking any attempts to exploit this same vulnerability to access a different private internal server, or a different section of the same one.

The run time of the entire decision tree classifier learning algorithm is on the order of one second, providing a patch almost immediately. The short response time enables the application to continue operating and serving benign requests while the vulnerability is further examined. The response time for regular expression filters on the other hand, is tens of minutes, although it can obtain a partial patch earlier, on the order of minutes. The seconds to minutes difference in response time is due to the fact that the classifier generation involves a one shot computing with all its inputs provided at the beginning, resulting in the decision trees at the end. In contrast, the process of generating a regular expression filter involves testing numerous variations of malicious inputs to zero-in on triggering the vulnerability causing the observed manifestation. FuzzBuster uses a suite of fuzz testing tools to synthesize input messages that explore how a vulnerability can be triggered. Each synthesized input is tested by sending it to a copy of the application and waiting a specified time to see if the undesired condition is reproduced. This iterative stop and go process is inherently slower, even when the remote interaction overhead is not considered in the response time. The wait time to see if an input succeeds in reproducing the undesired condition is a configurable parameter (1 second for our experiments). To understand the impact of the wait time, note that for the CVE-2012-0021 attack, 444 inputs were tested, so the 36 minute response time includes up to ~7.4 minutes of wait time.

Considering the patch quality and response time together, the decision tree classifiers produce a patch within seconds, but are limited by what inputs have been already observed. FuzzBuster can take longer to produce a patch, on the order of tens of minutes, but can produce more general regular expressions that can block variations on the attack that have not been attempted yet. The test results confirm that the combination of classifier and regular expression filters complement each other providing a patch in seconds that may result in FP or FN, and buying time to find a more robust patch, either by the operator assisted optimal decision-tree or extensive FuzzTesting, that can block future attack variants more accurately.

V. CONCLUSION AND SOME NEXT STEPS

We have demonstrated effective, automated self-adaptation embedded within an application-centric perimeter defense concept. This represents substantial and important progress on two fronts: adaptive cyber-defense with quasi-realtime responsiveness to novel attacks that eventually manifest in detectable system conditions, and software engineering of self-improving software systems that integrates multiple synergistic technologies such as deep execution introspection, decision-tree classifier generation and targeted fuzz testing in the context of an execution management environment. While the initial evaluation is encouraging, further research would improve the quality of the defense against novel attack, and the scope and effectiveness of the self-management mechanisms. For example, the combination of faster but less precise protocol-aware adaptation followed by a generalized signature combination may be suitable for some but not all protected applications. We are expanding the set of remediation options available, both in depth and breadth to have more options as well as more synergistic combinations. Examples based on our current techniques include making FuzzBuster's regular expression filters protocol-aware and formulating the classifiers in a protocol independent manner for a more effective mix against a larger class of use cases. Simultaneously, we are working on expanding the scope of available adaptation responses from current I/O mediation policies to variants of executable binaries [10] as well as automated source code modifications [9], pushing the frontier of adaptive defense even deeper.

We are also working on enhancing the core A3 capabilities. One aspect is extending the scope of closed-loop adaptation to cover attacks that may have timing dependency using deterministic replay. Another is augmenting the concentration on reactive adaptation strategies with proactive approaches. Using FuzzBuster's meta-control to intelligently share the A3 laboratory area for proactively discovering and remediating vulnerabilities even before they are exploited is an example.

REFERENCES

[1] P. Pal, F. Webber, R. Schantz, "The DPASA Survivable JBI- A High-water Mark in Intrusion Tolerant Systems," Proc. EuroSys Workshop on Recent Advances in Intrusion Tolerant Systems, Mar 2007

[2] C. Payne, T.Markham, "Architecture and Applications for a Distributed Embedded Firewall," Proc. 17th ACSAC, Dec. 2001

[3] D. Benjamin, P. Pal et al., "Using a Cognitive Architecture to Automate Cyberdefense Reasoning," Proc. ECSIS Symp. on Bio-Inspired Learning and Intelligent Systems for Security, Aug 2008

[4] Cortex: Mission-Aware Cognitive Self-Regeneration Technology. Final Report, US AFRL Contract No. FA8750-04-C-0253, Mar 2006.

[5] P. Pal, R. Schantz, A. Paulos et al., "A3:An Environment for Self-Adaptive Diagnosis and Immunization of Novel Attacks," Proc. SASO workshop on Adaptive Host and Network Security, 2012

[6] D. Musliner, J. Rye, T. Marble, "Using Concolic Testing to Refine Vulnerability Profiles in FUZZBUSTER," Proc. SASO workshop on Adaptive Host and Network Security, 2012

[7] J. Quinlan, "C4.5: Programs for Machine Learning," Morgan-Kaufmann, San Fransico, USA, 1993

[8] H. Shrobe, R. Laddaga, B. Balzer, et al., "Self-Adaptive Systems for Information Survivability: PMOP and AWDRAT," Proc. SASO Conference, Boston, MA, 2007

[9] W. Weimer, et al., "Automatically Finding Patches Using Genetic Programming." In Proc. ICSE 2009

[10] T. Jackson, et al.,"Compiler-Generated Software Diversity;" in S. Jajodia et al (Eds.), Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats; Springer, September 2011