Practical Techniques for Regeneration and Immunization of COTS Applications*

Lixin Li Mark R. Cornwell E. Hultman James E. Just
Global InfoTek, Inc., Reston, VA, USA.

R. Sekar
Stony Brook University, NY, USA.

Abstract

In this paper, we describe RAMSES, a system for immunizing COTS applications from exploitation of common security vulnerabilities. RAMSES defends applications from many classes of attacks prevalent today. These attacks either have no effect on a RAMSES-protected application, or at worst, they cause an application crash. For attacks that do cause a crash, RAMSES learns from attack instances to generate improved responses that can then be deployed to protect application integrity as well as availability. Unlike many previous efforts in this area that required heavy-weight taint analysis, RAMSES achieves these benefits using light-weight techniques that typically introduce less than 5% overhead. Our experiments demonstrate that RAMSES is very effective for protecting a range of applications on the Microsoft Windows operating system. Thus, our techniques provide some of the benefits typically associated with vendor patches, while mitigating their drawback of timeliness.

1 Introduction

Experience has shown that most deployed applications contain a significant number of security vulnerabilities that continue to surface over their lifetime. Zero-day attacks that exploit previously undisclosed vulnerabilities are a source of serious concern for systems that are exposed to untrustworthy data, such as server systems. Even for those vulnerabilities that are first discovered by "good" folks, patch development to correct these vulnerabilities typically takes a long time, during which systems remain vulnerable. This window of vulnerability is further extended by the time needed for compatibility testing of patches before deployment.

The above factors have prompted end-users and system administrators to look for alternative defensive techniques that do not require active vendor cooperation. Unfortunately, the suite of defensive techniques available to a system administrator is quite limited. Some of these defenses require source-code access, which makes them largely useless in the context of proprietary software that dominates the Windows market. Other techniques introduce overheads that are too high to be acceptable for production systems. Finally, even when low overhead techniques are available, they can often convert an attack on integrity into one on availability, which is not an acceptable alternative for mission-critical systems.

An ideal solution to this problem would be one that (a) defends against most common types of attacks prevalent today, (b) protects application integrity without compromising its availability, (c) introduces low performance overheads and (d) works with COTS applications that are available only in binary code form. In this paper, we summarize our RAMSES (Regeneration And iMmunity SErviceS) system that is targeted at this ideal. RAMSES is inspired by the biological immune system in the sense that its initial responses may cause significant "collateral damage," analogous to the inflammatory response of the biological immune system. However, over time, those responses that cause too much collateral damage (e.g., application crash) will be replaced with more targeted responses that thwart attacks without causing significant harm to the host application. This learning ability mimics the (biological) acquired immune system, which is very effective in wiping out a pathogen without damaging the host organism. The key benefits of RAMSES are:

- *Protection from a large fraction of the most popular attacks,* including buffer overflows, format-string attacks, SQL injection, command injection, cross-site scripting, and so on.
- *Immunizing systems against zero-day attacks.* By "immune," we mean that attacks impact neither the integrity nor the availability of protected applications.
- *Protection against attack variants.* Our vulnerability-based approach to signature generation and application level filtering protect against variants of attacks.
- *Protection from brute-force attacks.* Some attacks may require guessing a random value or a key (or a password), e.g., guessing attacks on address-space randomization, and hence may require multiple attempts. By developing and deploying a filter, all but the first few attempts are blocked, thereby providing a good degree of protection from such attacks.

2 Related Work

The works most closely related to RAMSES are those based on taint analysis, and works on automated generation of (attack-blocking) signatures.

Taint-Tracking. A number of techniques have been developed that rely on fine-grained taint-tracking[1] for detecting

*This work was funded in part by Defense Advanced Research Project Agency (DARPA) under contract N00178-07-C-2005. Sekar's work was also supported by ONR grant N000140710928 and NSF grants CNS-0627687 and CNS-0831298. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, the Naval Surface Weapons Center, ONR, NSF, or the U.S. Government.

[1]These techniques indicate whether for each byte of program memory whether its content was derived from an untrusted source, i.e., it contains

memory corruption attacks [7] and script injection attacks [9, 15].

Although taint-tracking is very effective and highly accurate for detecting attacks, its use on production systems is seriously hampered by several factors. First, taint-tracking requires *intrusive instrumentation,* wherein every program statement is transformed to introduce additional statements that propagate taint. System administrators, concerned about the impact of such instrumentation on the stability and robustness of the target application, may be reluctant to deploy them on production systems. Second, taint-tracking techniques, especially those operating on C [15] or binary code [10, 11], have high overheads, often slowing down programs by a factor of two or more. Finally, taint-tracking techniques are generally *language-specific.* For instance, none of the previous techniques have been applicable to both Java and C.

In the RAMSES project, we overcame these drawbacks of traditional taint-tracking techniques with *taint inference,* a non-intrusive approach that can be easily applied to COTS applications without access to source-code. Our experiments indicate that it introduces low overheads of under 5%. In addition, our technique is truly language-independent, having been applied to programs written in Java, C and PHP.

Automated Signature Generation. The last few years have witnessed considerable interest in automated responses to attacks. The most popular approach in this context has been the development of automated techniques for filtering malicious inputs, typically called automated signature generation or filter-generation. Initially, this line of research was targeted at network worms, and relied on "content-based signatures," where the signature captured characteristics of the attack payload. The problem with such signatures is that they can often be evaded by polymorphic attacks. To address this problem, several researchers have focused on generating vulnerability-oriented signatures [6, 3, 1, 4, 14]. However, practical adoption of these techniques is hampered by one or more of the following drawbacks:

- *Reliance on heavy-weight analyses.* Many of these techniques rely on very heavy weight analyses, including taint analysis [3, 1] and symbolic execution [1, 2]. Other techniques [14, 4] rely on generating many attack variants and sending these as inputs to the protected application.
- *Requirement for accurate replay of attacks.* Due to their nature (use of heavy-weight analysis or the need for generating attack variants), online signature generation becomes impossible. Instead, off-line techniques are needed, where all inputs are first stored. When an attack is detected, recent inputs (and possibly their variants) need to be replayed (possibly on an instrumented version of the application). Accurate replay is a challenging problem for many applications since their response to any given input is a function of their state. To ensure accurate replay in general, the exact state of the application at the time of input processing needs to be reproduced — this in turn may require generation of checkpoints of some sort, which will again increase overheads, as it needs to be taken for each input.[2]
- *Limitation to narrow class of attacks.* Previous techniques [6, 3, 1, 2, 14, 4] have been developed in the context of memory corruption attacks, and do not address other attack classes such as SQL injection.

data that may be controlled by an attacker.

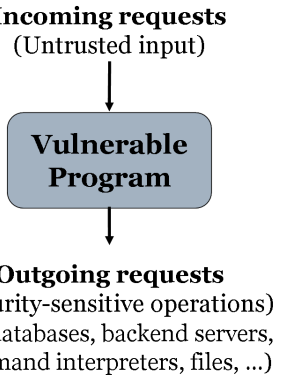

Figure 1. Protected Application.

In contrast, RAMSES project is aimed at developing *filters* that are applicable to a wide range of common exploits. These filters are generated online, without any need for attack replay or heavy-weight instrumentation/analysis.

3 Approach Overview

RAMSES is focused on defense against remote attacks intended to gain control of a target Windows-based host. The selection of Windows hosts forces us to address the issues of operating effectively without access to source code of either the OS or applications.

Figure 1 illustrates the context of RAMSES protection. The attack target is a program that mediates access to protected services (e.g., SQL server), subsystems (command interpreters), important resources (e.g., files or devices), as well as internal data structures (stack, heap, etc.). We assume that the goal of an attacker is to gain unintended access to protected subsystems/resources. Since a remote attacker has no direct access to protected resources, he can gain unintended access only indirectly: by crafting a malicious incoming request that subverts the vulnerable program into performing resource operations intended by the attacker. Thus, a successful attack relies on (a) the ability of an attacker to *exert control* over a sensitive operation, and

[2] If working exploits were available, then replay of attacks becomes easy. Unfortunately, in production settings, they are not available in many cases, e.g., zero-day attacks.

(b) whether this degree of control is *intended* by the programmer (or the administrator) of the protected application. Below, we provide an overview of our taint-inference technique (in Section 3.1) that addresses (a), and a policy-based output filtering technique (in Section 3.2) that addresses (b). In Section 3.3, we then summarize our input filtering technique, which forms the basis of immunizing applications.

3.1 Taint Inference

Taint-tracking can be used to answer question (a). It involves marking untrusted inputs as "tainted," and as the program uses this data, copying the taint labels together with data values. However, as mentioned earlier, taint-tracking has several drawbacks. We have therefore developed the technique of *taint-inference* which operates by observing inputs and outputs, but without making any changes to the protected application. These observations could be made on the network in most cases, but our implementation relies primarily on intercepting calls to standard libraries.

If input undergoes arbitrary transformations in the program, then it becomes difficult to infer the relationship between inputs and outputs. However, in practice, many applications do not make arbitrary transformations, but rely on certain standard transformations, e.g., base64 or URL encoding, gzip compression, etc. If knowledge about these standard transformations were incorporated into the taint inference algorithm, then it becomes feasible to infer data propagation by comparing inputs and outputs. In our implementation, we intercept inputs after inputs are decoded, and outputs before they are encoded, and hence avoid problems created by these standard transformations.

To understand taint inference in a bit more detail, consider web applications. Incoming requests to web applications use the HTTP protocol, with standardized ways of encoding parameter names and values. Web applications typically retrieve these parameter values, apply simple sanitization or normalization operations on them, and finally use their values within an outgoing request sent to a back-end system. As a result, data flows can be identified by comparing input parameter values against (all possible) substrings of outgoing requests.

One of the key features of our taint inference algorithm is that it relies on approximate (rather than exact) substring match so as to be able to identify taint in the presence of simple sanitization or normalization operations. Such operations are often used by web applications, e.g., addition of quote characters, replacement of spaces with underscores, removal of certain characters, etc.

To illustrate taint inference, consider a command injection vulnerability in version 1.4.0 of SquirrelMail (a popular web application for email access) with GPG plug-in version 1.1. The vulnerable URL is `/squirrelmail-1.4.0/plugins/gpg/gpg_encrypt.php`. In the exploit we studied, there were about dozen parameters, all of which were parsed and extracted. The one that is of interest is the "send_to" parameter, which had the value

```
alice, bob; touch /tmp/GotYou
```

Based on these input values, SquirrelMail generated the following shell-command:

```
echo '···' | /usr/bin/gpg ··· -r
   alice@ -r bob;touch /tmp/GotYou@ 2>&1
```

Some parts of the command that are irrelevant for the attack have been replaced with "···". In addition, characters copied from the send_to parameter have been highlighted in italics. Note that the input text has gone through a few changes before use: in particular, the recipient list has been separated into its component names, and each of these names prefixed by a "`-r`" and postfixed with an "`@`" symbol. As a result, an exact substring matching algorithm will not identify that the send_to parameter appears in the shell-command, but an approximate substring matching algorithm can detect this with a high degree of confidence.

The goal of this paper is to provide a high level overview of the RAMSES system. Full details about the taint inference algorithm can be found in Reference [12], which also provides a full discussion of attack detection on web applications.

3.2 Output Filters

Taint inference enables us to determine whether an output is controlled by an attacker. Next, we need to determine whether this control is *intended* by the programmer and/or the administrator of the protected application. This is typically done using *security policies*. The technique of combining taint with security policies is well-established now, having been the subject of numerous papers [7, 9, 13, 15].

An important insight contained in many of these papers is that string-based injection attacks, including SQL injection, command injection and format-string attacks, involve alterations to the syntactic (or lexical) structure of output requests (see Figure 1) that result due to tainted data. For instance, in the SquirrelMail example described earlier, the attack changes the structure of the shell command: by introducing semicolons, additional shell commands are being introduced into the output request. The following policy will block this attack: tainted data must not span multiple words in the output. The popularity of taint-based defenses stems partly from the simplicity and generality of such policies. We have shown [12] that just 3 policies are enough to detect shell command injection, PHP command injection, SQL injection, and (reflected) cross-site scripting.

RAMSES output filters operate by blocking policy-violating outputs from being sent on the output interface shown in Figure 1. Specifically, note that output requests

are usually made by calling a function. Our filters intercept this call, and instead of allowing the call to go through, an error code is returned to the application. For string injection attacks, e.g., SQL injection, the application is typically able to process the error code and recover from the error. In those cases, output filtering provides sufficient protection. But there can be instances where recovery fails. For instance, with memory corruption attacks, process memory is already corrupted by the time of attack detection, and hence the best course of action is to terminate the program. Recovery may also fail in the case of attacks such as SQL injection if the application was not written carefully, and it ignored (some) error returns. In those cases, it is necessary to generate *input filters* (see Section 3.3) rather than relying on output filters.

In RAMSES project, we showed that taint-induced alteration of output structure was characteristic of not only string-based injection attacks, but also memory corruption attacks. In particular, when data is copied into a buffer, we expect that the copied data will be confined to a single array or structure, but a successful attack violates this condition. This observation enabled us to unify the detection of memory corruption and string injection attacks, and led to the development of a uniform technique for defending against both. In particular, we were able to unify the development of input filters for both types of attacks.

3.3 Input Filters

Automated signature generation techniques are aimed at detecting inputs that lead to attacks. One of the main challenges in signature generation is that of keeping false positives very low: benign input should not be filtered out, or else the signature generation system can inflict a DoS attack on the protected system. Another challenge relates to the generality of signatures. Ideally, a signature generated on the basis of an exploit would not only stop that exploit, but also its variants — ideally, all variants that exploit the same underlying vulnerability should be blocked by the signature.

We have developed a novel approach for producing generalized input filters that take advantage of taint-inference and the policies that are used to detect attacks at the output[3].

Specifically, we view the protected application as a function that maps input requests (and parameters) to corresponding outgoing requests and their parameters. Our approach is based on "learning" this function. (This approach works under the same assumption as taint inference: the protected application does not make arbitrary transformations on the input, but primarily relies on copying, with small changes.) There are two steps involved in this process.

- Given a particular outgoing request O, derive the function $f : I \longrightarrow O$ that maps the incoming request I to the corresponding output request O.
- Given an input I, identify the set O of output operations that could potentially result due to this input. Since all our security policies are based on tainted outputs, we limit ourselves to the subset of output operations that have some tainted parts.

Given that many of the applications of interest operate on strings, one of the obvious choices for the first task are finite-state string transducers (FSTs), which are similar to finite-state machines, except that they distinguish between input and output symbols. While FSTs are a reasonable starting point, we found them to be quite fragile for our purposes, since the languages we deal with at the input and output points are quite complex. We observed that a more robust learning algorithm could be developed by viewing the transformation as operating on parse trees of outputs, rather than a (flat) string representation. We then leveraged the approximate string-matching algorithm to compute the transformations that take place on input parameters before they are used in an outgoing request. In particular, this algorithm can be used to identify if a subtree of the output is derived from a specific input parameter, and to approximate the sanitization operations that may have been applied to this input.

In the next stage, we collect (I, O) pairs across a large number of training runs. We then use an algorithm to correlate input parameter values with outputs observed in these training runs. The purpose of this algorithm is to discover which input parameters play a role in determining the output operations invoked by the protected application. This is basically a classification problem. In particular, we construct a decision tree that has branches based on input parameter values, and each leaf identifies the set of output operations that may be made by the application when an input satisfying the conditions on the path from the root to this leaf is received by it. Note that in order for the classifier to converge, it will need to apply certain generalizations on the outputs. In our implementation, we simply compute the maximal common prefix of the set of outputs contained in any leaf of the decision tree. This means that for a given input, we may be able to predict only some parts of the output, while the rest of the parts are unknown. Hopefully, those parts of O that are necessary for checking output policies would be defined.

Now, the decision tree is used as follows to construct input filters. Given an input, the decision tree is used to predict the output. Attack detection policies are applied on this predicted output, and if there is a match, then the input

[3]For memory corruption attacks, attack detection relies on the address space randomization (ASR): in the presence of ASR, a memory corruption exploit would, with a high probability, cause a crash. A post-crash analysis of process memory is used to confirm that a memory corruption was the most likely reason for the crash.

Application	Language	Size (lines)	Environment	Attacks	Comments	**Detection**	**False Positives**
phpBB 2.0.5	PHP/C	34K	IIS, Apache	SQL injection	CAN-2003-0486	Yes	None
SquirrelMail 1.4.0	PHP/C	42K	IIS, Apache	Shell command injection	CAN-2003-0990	Yes	None
SquirrelMail 1.2.10	PHP/C	35K	IIS, Apache	XSS	CAN-2002-1341	Yes	None
PHP/XMLRPC	PHP/C	2K	IIS, Apache	PHP command injection	CAN-2005-1921	Yes	None
AMNESIA [5] (5 apps)	Java/C	30K (total)	Apache/Tomcat	SQL injection	21K attacks, 3.8K legitimate	100%	0%
WebGoat [8]	Java		Tomcat	HTTP response splitting Shell command injection		Yes Yes	None None

Figure 2. Applications Used in Experimental Evaluation.

can be blocked. To avoid false positives, an input filter is deployed only if it is verified not to match a set of previously collected benign inputs (i.e., inputs that do not lead to an attack being detected).

4 Preliminary Results

The RAMSES system has been implemented on Windows, and has been evaluated using a number of applications. Some components of RAMSES (e.g., taint inference and output filters) have been more thoroughly evaluated than other components (e.g., input filters), and we summarize these results here.

4.1 Taint Inference

We have shown that taint inference works very well on web applications. For a variety of web applications written in Java, PHP and C, we showed that taint inference is able to infer taint propagation accurately enough to block all attacks. In addition, we showed that its performance is very good – in particular, it introduces overheads of less than 5% across the applications we studied.

Taint inference may fail to detect dependences between inputs and outputs for applications that perform significant transformations on inputs. It should be noted that the most common transformations arise due to various encodings used in HTTP, but these are already handled by our approach. Other than these, the most common transformations performed by web applications result in small changes to the input, and are hence detected by our approximate matching technique. However, if a web application makes extensive use of application-specific encodings or input-to-output transformations, taint inference may lead to significant false negatives and is hence inappropriate for such applications.

4.2 Output Filters

Output filters were evaluated using the applications shown in Figure 2. Of these, phpBB and SquirrelMail are very popular PHP applications, while PHP/XMLRPC is a popular library. AMNESIA dataset consists of many medium-sized realistic Java applications that have been used by previous works on SQL injection detection. This figure shows that RAMSES was very effective in producing accurate filters, using just a handful of policies. Additional details about these applications can be found in Reference [12].

In addition to the above applications, RAMSES output filters were evaluated by an external Red Team. This Red Team developed their own custom web application in PHP that was seeded with vulnerabilities by them. RAMSES was able to block all attacks on this application with the exception of one. This attack involved persistent data: i.e., attacker provided data was first stored into the database and subsequently retrieved, and SQL injection attack resulted when the retrieved data was used in constructing another SQL query. This is a known limitation of most existing taint-based techniques, including RAMSES. (It can be addressed by extending taint into persistent storage, i.e., recording the taint bits in the database along with the original data.)

4.3 Input Filters

Input filters have been evaluated primarily in the context of memory corruption attacks. Our evaluation considered a synthetic application (MEVS) that contained a variety of memory corruption vulnerabilities, as well as a few real-world vulnerabilities. The results are summarized in Figure 3.

As is evident from the table, our technique is quite effective in generating successful signatures for memory corruption exploits. This stems partly from our use of approximate string matching for taint inference. This enables us to deal with one of the common problems that arise in taint-based signature generation, namely, corruption of data that may happen between a buffer overflow and control-flow hijack. (Recall that we use ASR for detecting memory corruption attacks. As a result, attacks are not detected until the time of control-flow hijack, which typically causes a memory access violation.) For instance, if there is a buffer overflow on stack-allocated array, it is possible that local variables may be stored after the end of the array. Between the array overflow and the time the function returns, these local variables may be updated. As a result, originally tainted data is overwritten with untainted data, thus making it difficult to trace the data back to an attack input. These "gaps" in tainted

Vulnerability Type	Attack Target	Payload Type	Attack Description	Buffer Identified?	Signature Generated?
Synthetic	MEVS	Working exploit	Stack buffer overflow to overwrite return address	Yes	Yes
Synthetic	MEVS	DoS	Stack overflow to overwrite return address	Yes	Yes
Synthetic	MEVS	Working exploit	Stack overflow	Yes	Yes
Synthetic	IIS ISAPI extension DLL	Working exploit with payload encoded	Stack overflow to overwrite SEH	Yes	Yes
Synthetic	MEVS	DoS with original input compressed	Stack overflow with decompressed input to overwrite return address	No	No
Synthetic	MEVS	Working exploit	Heap Overflow Freelist[00] to trigger double pointer unlink	Yes	Yes
Synthetic	MEVS	DoS	Heap Overflow [1 - 127] to trigger double pointer unlink	Yes	Yes
Synthetic	MEVS	Working exploit	Heap Overflow Lookaside list to trigger double pointer unlink	Yes	Yes
Synthetic	MEVS	DoS	Heap Overflow triggering blocks coalesce and double pointer unlink	Yes	Yes
Synthetic	MEVS	DoS	Heap overflow with original input reversed before overflow	No	No
Real world (CVE-2004-1134)	IIS5	Working exploit	Overflow stack buffer in w3who.dll to overwrite SEH	Yes	Yes
Real world (OSVDB-20909)	FreeFTPd 1.08	Working exploit	Overflow stack, overwrite SEH	Yes	Yes

Figure 3. Generation of Input Filters.

data are handled naturally by the use of approximate string matching. As a result, successful signatures are generated in spite of gaps. This contrasts with previous techniques such as [6] that rely on exact string matching, and hence may fail to generate signatures in the presence of such gaps.

In addition, the input filter generation algorithm was tested on an SQL injection attack on phpBB, and it successfully generated an accurate signature. The implementation of input filter generation has not been fully completed, and hence we have not evaluated it on other attacks.

Our input filter generation techniques inherit the limitations of taint inference mentioned earlier. In addition, note that input filtering is inherently harder than output filtering: with output filtering, we need only decide whether a particular output violates a policy; in contrast, with input filtering, we need to predict if a certain input can lead the application to produce an output that would violate this policy. Since such prediction is hard, one would expect that input filters would suffer from a higher rate of false positives and false negatives as compared to output filters.

Input filter generation has been motivated in part by our desire to protect an application from DoS attacks. However, we have not systematically evaluated our technique against DoS attacks.

References

[1] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 2–16. IEEE Computer Society Washington, DC, USA, 2006.

[2] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: Securing software by blocking bad input. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 117–130. ACM New York, NY, USA, 2007.

[3] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. *ACM SIGOPS Operating Systems Review*, 39(5):133–147, 2005.

[4] W. Cui, M. Peinado, H.J. Wang, and M.E. Locasto. Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing. In *IEEE Symposium on Security and Privacy, 2007. SP'07*, pages 252–266, 2007.

[5] William Halfond. SQL injection application testbed. On the web at http://www.cc.gatech.edu/ whalfond/testbed.html.

[6] Z. Liang and R. Sekar. Fast and automated generation of attack signatures: A basis for building self-protecting servers. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 213–222. ACM New York, NY, USA, 2005.

[7] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Network and Distributed System Security Symposium (NDSS)*, 2005.

[8] OWASP. Owasp webgoat project. On the web at http://www.owasp.org/index.php/Category:OWASP_WebGoat_Project.

[9] Tadeusa Pietraszek and Chris Vanden Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection (RAID)*, 2005.

[10] Feng Qin, Cheng Wang, Zhenmin Li, Ho seop Kim, Yuanyuan Zhou, and Youfeng Wu. LIFT: A low-overhead practical information flow tracking system for detecting general security attacks. In *IEEE/ACM International Symposium on Microarchitecture*, December 2006.

[11] Prateek Saxena, R. Sekar, and Varun Puranik. Efficient fine-grained binary instrumentation with applications to taint-tracking. In *IEEE/ACM Conference on Code Generation and Optimization (CGO)*, April 2008.

[12] R. Sekar. An Efficient Black-box Technique for Defeating Web Application Attacks. In *Network and Distributed System Security Symposium (NDSS)*, 2009.

[13] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 372–382, New York, NY, USA, 2006. ACM.

[14] X.F. Wang, Z. Li, J. Xu, M.K. Reiter, C. Kil, and J.Y. Choi. Packet vaccine: Black-box exploit detection and signature generation. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 37–46. ACM New York, NY, USA, 2006.

[15] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *USENIX Security Symposium*, August 2006.