N-Version Programming for the Detection of Zero-day Exploits

Article · January 2006

CITATIONS
6

READS
514

3 authors, including:

William Allen
Florida Institute of Technology
55 PUBLICATIONS **410** CITATIONS

SEE PROFILE

FIT Computer Science Technical Report: CS-2006-04

N-Version Programming for the Detection of Zero-day Exploits

Lajos Nagy, Richard Ford and William Allen

Using N-Version programming techniques to increase software reliability is a well-explored field. In this paper, we extend the concept to the detection of new security vulnerabilities. Using our own N-Version arbiter, Judicare, we implement a simple auction web application, and demonstrate how our application is robust to the most common Web vulnerabilities as documented by OWASP. Finally, we discuss the implications of our results in the context of detection of Zero-Day attacks.

Contact: Dr. Richard Ford, rford@se.fit.edu
Department of Computer Sciences
Florida Institute of Technology

Invited paper, presented at:

The IEEE Topical Conference on Cybersecurity, Daytona Beach, FL, April 2006

N-Version Programming for the Detection of Zero-day Exploits

Lajos Nagy
Department of Computer Sciences
Florida Institute of Technology
Email: lnagy@fit.edu

Richard Ford
Department of Computer Sciences
Florida Institute of Technology
Email: rford@fit.edu

William Allen
Department of Computer Sciences
Florida Institute of Technology
Email: wallen@fit.edu

***Abstract*—**
Using N-Version programming techniques to increase software reliability is a well-explored field. In this paper, we extend the concept to the detection of new security vulnerabilities. Using our own N-Version arbiter, Judicare, we implement a simple auction web application, and demonstrate how our application is robust to the most common Web vulnerabilities as documented by OWASP. Finally, we discuss the implications of our results in the context of detection of Zero-Day attacks.

I. Introduction

Managing computer insecurity has become an increasingly large problem, with some estimates showing that security issues cost billions of dollars worldwide per year [1]. Typically, this insecurity has its roots in two different causes: software vulnerabilities and system misconfiguration.

While misconfiguration is somewhat difficult to solve technologically, there has been significant research directed to solving the problem of insecure software. Patching, Intrusion Detection Systems, and improved software development practices are but a few of the different countermeasures deployed; however, to understand software security issues in context, it is important to recognize the security lifecycle for vulnerabilities and exploits [2].

If we consider a system to be in a 'secure' state initially, when a vulnerability is discovered and disclosed it moves to a 'vulnerable' state. From here, the machine either returns to a 'secure' state when a patch is applied, or to an 'exploited' state when successfully penetrated. The time between the release of a vulnerability and the release of an exploit which leverages it is known as the number of 'Days of Risk' [3] and it is this figure which many consider to be a critical measure of security, as any gap between the availability of a solution and the availability of an exploit represents a time when machines are susceptible to attack.

A zero-day attack is an exploit which is released before there is widespread public knowledge of the underlying vulnerability. This case presents a special problem, as defenders are essentially tasked with detecting and parrying an as-yet unknown attack. In this paper we present the application of N-Version techniques to not only mitigate but also detect previously unknown exploits against component platforms.

The idea of N-Version programming is conceptually simple. One creates multiple versions of a particular application and compares the output of each version. If they agree, the system is deemed to be functioning correctly. While this technique is often touted for 'ultra-high reliability' systems, its uptake generally has been low, partly due to cost, and partly due to some concerns regarding independence of failure modes (see [4]).

In the security world, however, the goal is more than creating systems which have the property of high reliability. Simply knowing about the existence of a vulnerability can significantly help the task of defending a system. Thus, we have chosen to re-examine the subject of N-Version Programming from the perspective of exploit detection. In this context, many of the objections to N-Version techniques become void, as the problem space is quite different, and the potential value of the technique is greatly increased.

Our belief – supported by a thorough examination of the most common web application vulnerability causes – is that the conditions for software reliability are quite different to those required for security. For example, while Knight *et al.* have argued that different programming teams can make similar mistakes, it is more difficult to argue that the failure mode will be identical. Furthermore, security is often about exploitation of a vulnerability. When considering a buffer overrun, for example, it is *extremely* difficult to construct a buffer overrun which works on both Intel and SPARC architectures. Thus, even if there is some level of correlation in errors, leveraging these mistakes for anything other than denial of service is of extraordinary difficulty.

Our paper is constructed in the following form. In Section II we examine the ability of N-Version techniques to detect common vulnerabilities in web applications. In Section III we describe our application, and in Section IV we describe details of the implementation of Judicare, our transaction adjudicator. Finally, in Section V we present the conclusion of our research and its implications for further work.

II. Top Ten OWASP Vulnerabilities

We begin by examining the threat landscape our application is expected to execute in. For completeness, we have chosen to consider the most common vulnerabilities as reported by the Open Web Application Security Project (OWASP), a community project that is "dedicated to finding and fighting the causes of insecure software" [5]. OWASP has compiled a list

of the ten most critical web application security vulnerabilities. This list represents a "broad consensus" of industry members and security experts about the most important security flaws found in web applications. In this section we consider each vulnerability in turn and examine how our N-Version Honeypot Site, *SealedBid*, can help in finding novel attacks that fall into a particular category:

1) **Unvalidated Input** "*Information from web requests is not validated before being used by a web application. Attackers can use these flaws to attack backend components through a web application.*"
   The system can effectively discover these kind of attacks. This follows from the nature of the architecture, as there are actually *several* backend components and an attack that might work against one type of backend is unlikely to work identically on another one. For example, an SQL injection attack that works for MS SQL Server is unlikely to work without modification on a different DBMS.
2) **Broken Access Control** "*Restrictions on what authenticated users are allowed to do are not properly enforced. Attackers can exploit these flaws to access other users' accounts, view sensitive files, or use unauthorized functions.*"
   The system should discover most of these kind of attacks. In effect, the system cannot discover broken access control mechanisms at the *application level* because this is an application logic flaw that would be implemented unchanged throughout the different implementations of the application (indeed, design-level flaws are not ever truly addressed by N-Version techniques). It is worth pointing out that even in this case N-Version implementation might prevent exploitation of the vulnerability: if more then one developer has to implement the same specification the chances of discovering application logic flaws is higher. In addition, attacks that target broken access control mechanisms in one of the underlying system components of the independent implementations can be effectively discovered since, in most likelihood, the same attack will not work across different platforms and thus would yield different answers for the different implementations.
3) **Broken Authentication and Session Management** "*Account credentials and session tokens are not properly protected. Attackers that can compromise passwords, keys, session cookies, or other tokens can defeat authentication restrictions and assume other users' identities.*"
   The system should discover the majority of these kind of attacks. In this case a lot depends on how the HTTP Query Dispatcher handles user sessions across the different implementations. The current approach is to mandate that all implementations use HTTP Cookies as session information and the Dispatcher will make sure that each implementation receives the cookie that it set for the user. In this scheme the bulk of the burden of ensuring session security is placed upon the application designer and the system can only catch those attacks that try to exploit some platform-specific vulnerability of session handling.
4) **Cross Site Scripting (XSS) Flaws** "*The web application can be used as a mechanism to transport an attack to an end user's browser. A successful attack can disclose the end user's session token, attack the local machine, or spoof content to fool the user.*"
   The system cannot discover these kind of attacks *per se*. Cross site scripting flaws belong to application logic flaws though it is likely that some of these attacks will be detected due to different standards of parsing user-supplied input.
5) **Buffer Overflows** "*Web application components in some languages that do not properly validate input can be crashed and, in some cases, used to take control of a process. These components can include CGI, libraries, drivers, and web application server components.*"
   The system can effectively discover these kind of attacks by virtue of its multiple implementations using different operating systems, web servers, DBMSes, etc. It is unlikely – if not outright impossible – for one buffer overflow attack to work across multiple platforms. Given the overall importance of buffer overruns in the security world, the system's ability to reliably prevent attacks via this vector is a crucial benefit.
6) **Injection Flaws** "*Web applications pass parameters when they access external systems or the local operating system. If an attacker can embed malicious commands in these parameters, the external system may execute those commands on behalf of the web application.*"
   The system can effectively discover these kind of attacks. Attacks that fall into this category are generally highly platform-dependent, so even if they succeed in breaking one implementation all others are most likely to remain unaffected or simply unable to interpret the attack.
7) **Improper Error Handling** "*Error conditions that occur during normal operation are not handled properly. If an attacker can cause errors to occur that the web application does not handle, they can gain detailed system information, deny service, cause security mechanisms to fail, or crash the server.*"
   The system can effectively discover these kind of attacks. Although error handling can fall under the umbrella of application logic flaws the system is still able to intercept these attacks because unhandled errors tend to generate vastly different outputs on different operating systems/web servers/etc. By virtue of comparing the outputs of each implementation, the system can easily catch such attacks.
8) **Insecure Storage** "*Web applications frequently use cryptographic functions to protect information and credentials. These functions and the code to integrate them have proven difficult to code properly, frequently resulting in weak protection.*"
   The system might discover some of these kind of attacks. Again, those attacks that exploit flaws in application logic are hard to catch because they will be present throughout

the multiple implementations. Nevertheless, if application developers used cryptographic functions provided by the respective platform environments then the system might catch some attacks because it is unlikely that the same error was coded into different implementations of the same cryptographic function. Furthermore, the underlying storage is likely to be different on different platforms, making it difficult for the attacker to craft an exploit which behaves identically across all servers.

9) **Denial of Service (DoS)** *"Attackers can consume web application resources to a point where other legitimate users can no longer access or use the application. Attackers can also lock users out of their accounts or even cause the entire application to fail."*
The system might discover some of these kind of attacks. To be precise, those attacks that try to crash the whole site through some OS or web server exploit will probably be able to do this only for one implementation. This provides the defender with useful information regarding the vulnerability without allowing for system compromise.
Furthermore, it should be possible to extend the system to consider the overall responsiveness of the different implementations. If certain requests take significantly longer on one platform than another, it might be worth considering the possibility that a potential DoS vulnerability may exist in the code. Thus, we envisage extending the system to include performance data at some point in the future.
10) **Insecure Configuration Management** *"Having a strong server configuration standard is critical to a secure web application. These servers have many configuration options that affect security and are not secure out of the box."*
The system can effectively discover these kind of attacks. Different implementations use different web servers and it is unlikely that an attack will work identically across different web servers and web server configurations.

III. The Web Application

In order to demonstrate these claims, we implemented a simple auction site, *SealedBid*, with multiple users that can bid for goods. The rationale behind this decision is that we wanted an application that uses a relational database and some form of user authentication. The chosen bidding scheme was a sealed-bid second-price auction where a single indivisible item is being sold – a so-called Vickrey auction [6]. In this auction type bidders submit their bids in a 'sealed envelope' and the winner of the auction pays the *second largest* bid that arrived. It can be proven that this auction type encourages bidders to use the real value (as perceived by them, of course) of the item as their bid without actually forcing them reveal this value to their competitors.

In our implementation the auction site SealedBid plays the role of the auctioneer and thus it is important that users of the site have faith in its security. On the other hand, being the holder of sensitive information, e.g. the bid amounts of each bidder, the site is an ideal target for online attacks since a simple glance at the contents of the bid database could reveal a lot about the perceived value of the goods up for auction.

A. Web Site Functionality

The following is a list of the main pages of the auction site along with the brief description of the relevant functionality of the page in question. In the current implementation each user must be either a seller or a bidder, but cannot be both.

- *Login Page* Presents the usual *username* and *password* fields to the user and checks the supplied values against the database. If the authentication was successful the user is redirected to the *Bidder Page* or the *Seller Page*.
- *Seller Page* This page displays both ongoing and closed auctions that belong to the seller. For open auctions the user can see how many bids have arrived so far and how much time is left for the auction. When the auction is over it is closed and the seller can see the username of highest bidder and, in accordance with the auction rules, the second largest bid that the winner will have to pay for the item in question.
- *Bidder Page* This page displays all open auctions and closed auctions. The bidder can see his last bid for each auction that it bid for. As long as an auction is open the bidder is free to change his/her bid. Since it is a Vickrey auction, the bidder is not required to increase its bid. A bidder simply submits a new bid for the lot in question that reflects the bidders latest estimation of the 'real' value of the item. For closed auctions, the bidder can see who won the auction but (s)he cannot see the price the winner has to pay *unless* the auction was won by the current bidder.

B. N-Version Implementation Details

The decision was made to separate business logic from the database and use a three-tier architecture when implementing the site. All business data was to be kept in a relational database while the pages were to be generated with one of the popular web-scripting languages (ASP, JSP, PHP, etc.) In order to facilitate N-Version programming the functionality of each and every page had to be specified with meticulous detail. For example, the maximum and minimum length of each possible input field had to be specified in order to avoid inconsistencies that might result from one application accepting six character passwords and the others not doing so. In a sense, all parallel implementations were to perform their tasks in unison, with the overarching objective being to ensure that two implementations can only get out of sync due to external attack or code implementation errors. To achieve this objective the functionality of each page was kept to the bare minimum so that it would be easy to test across different implementations.

The biggest difficulty was making sure that output generated by different implementations are the 'same' within certain limits (i.e. variations in the position and amount of whitespace, comments, etc.) Since we are talking about a web application,

the output from each application is simply an HTML document. Fortunately, there are several tools available that can put HTML documents into a 'canonical' form that facilitates their comparison based on the 'essential' contents of the documents (disregarding accidental differences in prelude, whitespace, comments, case, attribute ordering, etc.). To further reduce implementation efforts, we started out by creating an HTML template for each page of the site and used the exact same templates for each implementation. Naturally, the templates had to be modified in each particular case, but still, our experience is that this approach helped enormously in ensuring that different implementations generated the same output for the same input.

At one point during the design process, we considered implementing a shared relational database preceded by a proxy. The rationale of this is that it would greatly simplify implementation, by ensuring that the databases of different implementations would never get out of sync. In this 'single database' setting a separate 'query arbiter' would have been need in front of the database that would have compared the relational queries submitted by each implementation and would only relay the query to the DBMS if all implementation agree. Obviously, this setting would ensure that the contents of the database cannot get out of sync. On the other hand, a single database would seriously limit the ability of the whole system to act as a Honeypot because certain types of attacks, i.e. those that target some particular DBMS implementation, could not be caught at all. Because of this, the decision was made that each implementation would use its own database. As a consequence, in the 'multiple databases' arrangement we had to make sure that we have a way of synchronizing the different database instances, the repositories of permanent state in our web application. We chose the simplest solution for now: there is a 'reset button' that simply resets the contents of each database to some initial state.

IV. Implementation Details

The architecture of the system consists of two main types of components: the HTTP query dispatcher and the different implementations. The general idea is that an incoming HTTP request is received by the HTTP query dispatcher, which relays the query to each implementation. Once each implementation is done processing the query and has generated some reply, the dispatcher then compares the replies and if they all match then returns one of them to the client. If there is a mismatch between any two replies or any of the implementation's timeouts then the dispatcher is instructed to suspect that an attack just took place and logs all relevant data of the query for further analysis.

In addition, to conceal its Honeypot nature, in the case of a reply mismatch the dispatcher will display some neutral error message, take the site off line, and notify the system administrator. As long as there are no successful attacks, from the point of view of its clients, the system operates as a site with a single implementation behind it.

OS	Web Server	App. Logic	DBMS
Debian Linux	Apache	PHP	MySQL
Solaris 9	Tomcat	JSP	Postgres
Windows 2000	IIS	ASP	MS SQL

TABLE I
IMPLEMENTATION PLATFORMS

The underlying assumption is that any external attack will cause one or more implementations to generate output that either reflects that the attack succeeded, or, which is more likely, that the particular implementation was not able to interpret the 'malformed' query. Either way, the likelihood that multiple implementations generate *exactly* the same output for an external attack is highly unlikely.

For the current project we chose to implement the SealedBid site on three different platforms. Table I summarizes the characteristics of each platform. In effect, we tried to make them 'as different as possible' in order to minimize the possibility of a single attack working across different implementations. To provide similar hardware environment for each platform, and to decrease development efforts, we decided to use a virtual PC environment provided by **VM**ware™.

A. Debian GNU/Linux

The Linux implementation was done first, took the least amount of time and was the easiest to set up – no wonder the Apache/PHP/MySQL triumvirate is so popular! During implementation there were some concerns that certain PHP specific 'tricks' might turn out to be hard to implement on the other platforms so it was decided that PHP 'trickery' would be kept to minimum and features of the scripting language used in implementation were chosen judiciously, always keeping an eye on the other two platforms.

B. Solaris 9

Due to the relative unfamiliarity of the platform to the developers, setting up the Solaris environment took considerable amount of time. The main difficulty was that Solaris has its own unique layout for 'standard' UN*X configuration files and it took a while to learn where to look for what. After installation of the operating system the Solaris package manager was a welcome surprise and greatly simplified the installation of the Jakarta Tomcat JSP Engine/Web Server and the Postgres DBMS. The development was done in JSP, using the already existing Linux/PHP implementation as a guideline. Due to the convergence of web scripting languages the JSP implementation could closely follow the structure of the PHP scripts which saved a some development time.

C. Windows 2000

Microsoft prides itself on the user-friendliness of its products and in fact the setting up our final platform of implementation, the Windows 2000/IIS/MS SQL combination, was only slightly slower than setting up the Debian GNU/Linux environment. With two implementations already under our

belt, it was relatively easily to write the application using ASP based on the JSP and PHP implementations.

D. The HTTP Query Dispatcher

The Query Dispatcher plays a central role in our architecture. This is the component that actually listens for incoming requests, logs all events, dispatches the request to each implementation, compares responses, and decides whether an attack likely took place. A chain is as strong as its weakest link, so in any realistic setting the Query Dispatcher has to be placed on a platform whose security is very unlikely to be compromised.

For the actual implementation of the Dispatcher we decided to use IBM's Web Intermediaries (WBI) framework that is developed by IBM's Almaden Research Center [7]. WBI is a Java framework that allows developers to build sophisticated 'assembly lines' from basic components that process HTTP requests and response in some way. The Query Dispatcher is implemented as HTTP proxy using the WBI Java Development Kit.

As the main tool of discovering novel attacks is extensive logging, the Dispatcher does exactly that. Each incoming request is logged and each server response is also logged. If a mismatch between the backend server outputs occurs the Dispatcher marks the request that caused the disturbance along with the outputs that were generated in response to that request.

Session management almost deserves a section in its own right because of the invasive changes our session management makes to the data transmitted over HTTP. However, as the Query Dispatcher plays an important role in it, we have included the discussion here. In order to maintain the illusion that each implementation operates as a stand alone web application something has to be done with the HTTP cookies that are set by each implementation. The problem is that the client sees just *one* session while actually there are three separate session in the background. This is where the Dispatcher steps in: it analyzes the session cookies from each implementation and maintains a separate session ID database and only sends a *single* session cookie to the client. When the client provides a session cookie to the Dispatcher, the Dispatcher checks its session ID database and if there is a match it sends each implementation its own session cookie. Basically the Dispatcher performs a seamless back and forth mapping between session cookies so the client will think that it is communicating with a single web application.

It follows that because of this manipulation of session cookies the backend implementations cannot be given the client HTTP request verbatim. In effect this reduces the usefulness of the system as a Honeypot since certain attacks might depend on specially formatted HTTP headers, which special formatting might be lost when the Dispatcher tinkers with session cookies.

V. Conclusions and Future Work

The idea of using N-Version techniques for resilience is well-known but often cost-prohibitive. In this paper, we have illustrated the potential for using the technique as a Honeypot in order to offset the high development costs of N-Version implementation.

We note that our approach will detect most new attacks within the OWASP framework with a high degree of reliability and a low false positive rate. Correctly specified, all differences either relate to a bug in the code, or a security issue. For security-sensitive systems, both of these faults are likely to be critical. Thus, the system can be used both to improve fault tolerance (and therefore availability) and also to detect unknown vulnerabilities in both the application and the underlying operating system components.

We would envisage the number of applications which leverage Judicare to be few; however, there is most certainly a role for the technology. We envisage deploying several Honeypots which use an N-Version design over the next 24 months, and will report on the security benefits and vulnerabilities discovered.

References

[1] L. Gordon, M. Loeb, L. Lucyshyn, and R. Richardson, *2004 CSI/FBI Computer Crime Survey*. CSI Publications, 2004.

[2] W. Arbaugh, W. Fithen, and J. McHugh, "Windows of Vulnerability: A Case Study Analysis," *IEEE Computer*, vol. 33, pp. 52–59, 2000.

[3] L. Koetzle, C. Rutstein, N. Lambert, and S. Wenninger, "Is Linux More Secure Than Windows?" *Forrester Research*, 2004.

[4] J. Knight and N. Leveson, "A reply to the criticisms of the Knight & Leveson experiment," *ACM SIGSOFT Software Engineering Notes*, 1990.

[5] OWASP Community, "The Ten Most Critical Web Application Security Vulnerabilities," *http://www.owasp.org/documentation/topten.html*, 2006.

[6] D. Lucking-Reiley, "Vickery Auctions in Practice: From Nineteenth Century Philately to Twenty-first Century E-Commerce," *Journal of Economic Perspectives*, 2000.

[7] IBM Research, "Web Intermediaries (WBI)," *http://www.almaden.ibm.com/cs/wbi/*.