N-Version Architectural Framework for Application Security Automation (NVASA)

Majid Malaika, Southern Methodist University
Suku Nair, Southern Methodist University
Frank Coyle, Southern Methodist University

Abstract. In this paper, we expose application security issues by presenting the usage of N-Version programming methodology to produce a new architectural framework to automate and enhance application security. Web applications and cloud computing are dominating the digital world; therefore, our goal is to build resilient systems that can detect and prevent both known and zero-day application attacks. Automated process flow not only reduces security efforts during the Software Development Life Cycle (SDLC), but also enhances the overall application security. In addition, we propose compartmentalizing the application into separate components and applying the N-Version methodology to the critical ones to reduce the additional overhead introduced by the N-Version methodology.

Introduction

With the World Wide Web, online applications have become vital and more compelling than ever. Many financial transactions are made from home/office without the need to physically be present at the bank to finalize these transactions. Governments around the world are digitizing most of their services to make the process more convenient for their citizens. It is estimated that more than 60% of Internet users interact with government websites to perform tasks such as completing applications, renewals or inquiring for information. In fact, this number is increasing every year [1]. Online shopping has become more prevalent and convenient to customers than ever. In 2011, it is estimated that more than a trillion U.S. dollars were spent on online merchandise in the USA alone [2]. Rapid growth and huge improvements in information technology have raised many challenges. One challenge is application security.

The DoD relies heavily on software to deliver instantly accessible data to its users [3]. However, this makes remote malicious attacks a serious threat to DoD systems and users. Methods investigated and presented in this paper will complement DoD efforts in detecting and preventing these attacks through an application security framework (NVASA) that uses the N-Version programming methodology.

The Security Problem

Today, the main problem with system security is that it is viewed by enterprises as a commodity, where the usage of password patterns and the integration of anti-virus applications and firewalls promote a false sense of security. This is because most cyber-attacks target the application layer rather than the physical or network layer. Most annual security reports demonstrate insufficient application security measures taken by both enterprises and individuals [4, 5]. Some of the challenges in providing application security [6, 7] include: 1) System Complexity, applications today have the capability to interact automatically with users and/or other systems in a very complex fashion, therefore increasing the possibility of error injection during the SDLC. 2) Ubiquitous networking, more systems are connected to the Internet without appropriate security, thus becoming available online. 3) Built-in Extensibility, This is a desired feature in software engineering because it would enable the flexibility to add new components to the existing system in the future. Therefore, making it possible to inject malicious code in to the system. 4) Common platforms, While common platforms reduce cost and time when implementing new technologies or building an application, they also increase a malicious user's chances of exploiting more systems.

Most active attacks are carried out by exploiting existing vulnerabilities in the system. These vulnerabilities could be: 1) architectural design, 2) implementation, or 3) operational and platform vulnerabilities [8]. Human faults made during the design phase result in architectural design errors into the model's structure. Consequently, human faults made during the writing of the code would result in implementation vulnerabilities. Finally, faults in configuration files are operational and are considered platform vulnerabilities. The most dangerous vulnerabilities of all categories are the ones leading to immediate unauthorized access of the application [9, 10, 11, 12]. Providing high levels of application security is paramount in ensuring network, systems, and data security.

The remainder of the paper is organized as follows: Section II presents related work in the field of application security followed by a description of the methodology of N-Version programming. Section III presents the building blocks of our proposed N-Version Architectural Framework for Application Security Automation. In section IV, we present a prototype of the proposed NVASA framework and experimental results. In section V we introduce compartmentalization to reduce overhead. Finally, section VI concludes with a look at future research.

Related Work

Application security is often one step behind the latest cyber-attack schemes for reasons discussed in the previous section. The current emphasis on application security is to fix existing implementation errors that could be exploited by publicly known attacks such as Buffer Overflow, SQL Injection and Cross Site Scripting [9, 11, 12]. Other related work emphasizes extensive revisions to eliminate or reduce the injection of errors during the application's lifecycle.

Application Security

Most current related work in the field of application security focuses on enforcing extensive security guidelines during the SDLC [13, 14]. These guidelines focus on providing the regulations needed to promote the development of secure applications through the SDLC. The initial approach to application security espoused manual audits to the source code [15]. This approach consists of reading the source code line by line to detect and fix existing vulnerabilities. Another method is "fuzzing" testing, which

is followed by inputting a distorted or illegal input to the application and monitoring the behavior of the application. Sulley and SPIKE frameworks [16] are two examples of "fuzzing" testing.

Runtime checking is another method for detecting and preventing the exploitation of existing vulnerabilities. This method of testing is followed by adding special checks within the source code to ensure the program behaves as desired. ProPolice framework [17] is a GNU Compiler Collection (GCC) extension developed by IBM to protect against stack smashing that uses runtime checking. Mudflap framework [18] is another GCC extension for pointer debugging that uses runtime checking to detect and prevent exploitation of vulnerabilities.

These solutions are limited and have major disadvantages. First, actual security vulnerabilities are triggered by a certain specific set of circumstances which makes it extremely hard to strike using random "fuzzing" testing. A second limitation is latent security vulnerabilities that are present within critical systems but concealed from the system's stakeholders. These latent vulnerabilities can damage an organization's reputation and could lead to financial loss. Further, it takes an average of 14 weeks to patch or fix an existing vulnerability after discovery, which opens a window for additional attacks leading to more damages [5]. A third limitation is the high cost of implementing a secure application because the enforcement of security training as well as hiring special security testers and purchasing specific security tools adds enormously to the total cost. A fourth limitation is the increased time-to-market since these solutions engage in extensive training and thorough testing of the code. Finally, following these extensive guidelines often results in the detection of known existing vulnerabilities. However, these extensive guidelines have no scheme for detecting or preventing latent vulnerabilities from being exploited.

N-Version Programming

The concept N-Version Programming was introduced in the late 1970's by Liming Chen and Algirdas Avizienis. It is defined as, "The independent generation of N≥2 functionally equivalent programs from the same initial specification" [19].

The aim of the N-Version programming methodology is to improve software reliability. It is introduced by the following proposal: "The independence of programming efforts will greatly reduce the probability of identical software faults occurring in two or more versions of the program" [19].

Therefore, to build a pure N-Version model as shown in Fig. 1, two policies must be adopted: 1) All versions must share the exact same initial specification. The purpose of the initial specification is to state the functional requirements that stakeholders want the application to perform. They must be clear and detailed oriented to eliminate any confusion during the development process. 2) Versions must be independently generated. This is achieved by choosing different algorithms and programming languages for each version, as well as the independent processes for generating the versions, which should be carried by N independent individuals or groups that have no interaction with each other. This isolation of design and process between groups, coupled with the diversity of choosing programming languages and algorithms, greatly reduces the probability of producing identical software faults in two or more versions [19, 20].

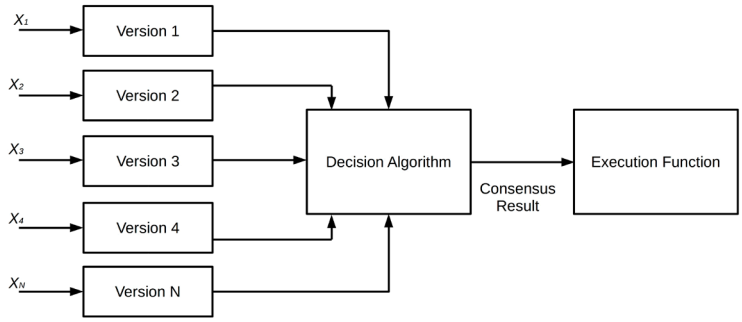

Figure 1. N-Version Model [21]

Building N-Versions Architectural Framework for Application Security Automation (NVASA)

To build the NVASA framework, we first collect the initial specifications and requirements from stakeholders. Second, we hand the specifications to N-Different programmers or groups of programmers. Each group develops a separate version using a different language and algorithm than the rest of the groups to satisfy the diversity of the N-Version methodology. Depending on the language and specifications outlined by stakeholders, one of the groups will develop the NVASA framework's layers mentioned below [21].

As shown in Fig. 2, the NVASA framework is constructed of four layers. The first layer is the N-Version routing layer, the second layer is the N-Version environment layer, the third layer is the N-Version decision layer, and the fourth layer is the backend application server layer.

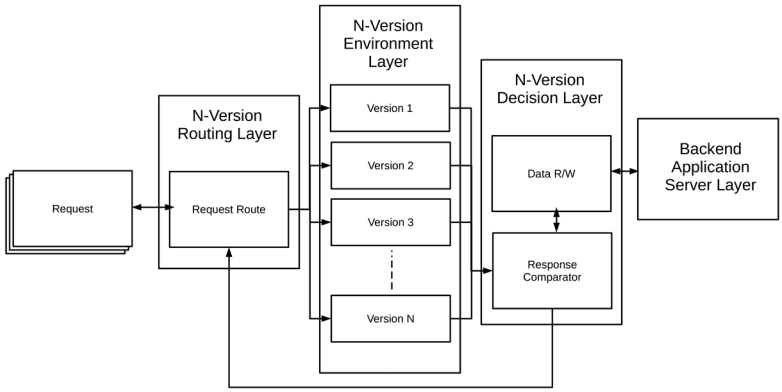

Figure 2. NVASA Framework Four Layers [21]

N-Version Routing Layer

The N-Version routing layer connects the framework with users/applications over the network. It is responsible for receiving requests from external users and routing the input to the N-Versions in the environment layer to be executed. The routing layer is also responsible for replying to requesters with the appropriate response after executing the request.

N-Version Environment Layer

The N-Version environment layer contains the N-Version applications where each version is designed and developed by an independent individual or group using a unique algorithm and/or programming language as mentioned in Section 2. Each version receives its execution command along with the identical input of

all N-Versions from the routing layer. All the versions then execute concurrently, which eliminates any reduction in performance. The N-Versions can be either loosely coupled running on different platforms in different physical locations or tightly coupled running on the same physical machine; this is decided based on the initial design specification of the NVASA framework.

Loosely coupled versions have many advantages: 1) The time overhead is reduced compared to tightly coupled versions running on one machine. 2) There is no single point of failure within the N-Version environment layer compared to tightly coupled versions. 3) Loosely coupled systems scale better than tightly-coupled systems. On the other hand, loosely coupled versions add more complexity to the development and testing phases of an application since messaging schemes must be implemented to connect layers and components. Additionally, the communication channels need to be protected through transport or network layer security protocols. Depending on the type of applications, communication between various layers and versions may cause additional time overhead.

N-Version Decision Layer

The N-Version decision layer is composed of two components: 1) The Response Comparator component, and 2) The Data Read and Write (R/W) Component. The response comparator component receives the N-Version outputs where a decision algorithm applies generic consensus rules to determine a consensus output. The role of the decision algorithm is to determine exploited versions by identifying conflicting output compared to the majority of the versions and removing the exploited and breached versions from the pool, thereby eliminating any malicious attempt to exploit the application. If necessary the response comparator component then passes the consensus output to the data read and write (R/W) component to generate the R/W command from the consensus output to be applied to the backend application server layer. Finally, the data read and write component generates the output or confirmation based on the initial request and passes it to the response comparator to be then sent to the request route component to reply to the requester.

Backend Application Server Layer

For applications that read or write data to or from a server or database, the backend application server layer is essential. It receives the consensus output from the data R/W component in the decision layer, preventing any direct communications between the backend application server layer and the N-Versions in the environment layer. This significant design requirement prevents any successful exploits from modifying or leaking the information by a malicious request.

Implementation and Results

A prototype NVASA framework was implemented and testing conducted to validate the protection provided by leveraging an N-Version service implementation in a distributed or web application architecture. In order to develop a practical prototype that can produce reliable results, we searched for pre-developed AES implementations online to be used as our N-Versions. These N-Versions must be written in different languages and by different developers in order to satisfy the diversity required in the N-Version programming methodology mentioned earlier. Three AES versions were located and used, the three were similar in structure and produced the same output. The first version was written in Java by Neal R. Wagner [22].The second version was written in C# by James McCaffrey [23]. And the third and final version was written in C/C++ by Niyaz PK [24].

As part of the AES framework implementation we exposed the C# and C/C++ versions as web services to be able to integrate them with the NVASA Framework. Each web service would accept a request with a 192-bit key and 128-bit clear text block and reply with the cipher text if the encryption was successful. Since the interface was developed using the Java language, a simple function call was sufficient to connect to the Java version which will also accept a request with a 192-bit key and a 128-bit clear text block. Fig. 3 shows the structure of our AES NVASA Framework implementation. We minimized our involvement in writing or modifying any code as much as possible to satisfy the diversity required in the N-Version programming methodology. Therefore, we used pre-developed implementations that matched each version's language to integrate each with the NVASA framework.

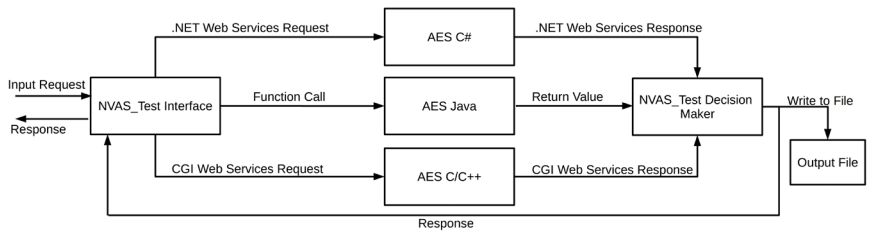

Figure 3. NVASA Framework AES Implementation

For this prototype we wrote a java class to act as the Routing and Decision Layer (Layer 1 and 3 From Fig. 3). The java class simply accepts the user input then executes all three versions simultaneously. Each version then produces its output which is then fed to the decision algorithm. The decision algorithm then runs to reach a consensus output. If a consensus is reached, the cipher text is written to a file and sent to the requester; otherwise, we have the choice of either dropping the request or replying back to the requester with a "Request Denied" message.

To test our NVASA implementation we developed a benchmark of a 100 test cases, each test case included a key length, key, and a clear text block. We executed each standalone version against each test case first, then executed the same set of test cases against the NVASA AES implementation. Table 1. shows a segment of the failed test cases and a comparison between the output of each version and our NVASA AES implementation.

Looking at table 1. we realize that in the first two test cases more than one version failed from a total of three versions, but providentially the NVASA recovered from such an exploit due to the fact that each version produced a different and unique output; therefore, the decision algorithm couldn't reach a consensus output; thus, dropped the request and never propagated it through the system. This step is essential because the decision algorithm will detect an attack and prevent it from propagating through the system even if it was successful in exploiting most of the versions.

This NVASA prototype shows how the new framework is able to improve Security by detecting and preventing known and potential zero-day attacks using the N-Version Programming methodology which revealed that the diversity of the languages and algorithm used greatly reduces the probability of having identical vulnerabilities in two or more versions. This was achieved without the need to modify the source code within the versions or install any patches.

Overhead Reduction

We have demonstrated that the NVASA framework is effective in improving security by detecting zero-day cyber attacks; however, the NVASA adds overhead to the development phase in the SDLC compared to the single version implementation. Therefore, compartmentalizing the application's architecture and applying the NVASA framework to the critical components in terms of security can reduce the overhead while improving security.

There are standards produced to assist in identifying the critical components within an application. The Common Criteria Information Technology Security Evaluation [25] is an international standard widely used to identify the critical security components. Other work has been done by Young, 1991. Also other work has been done by Andrew Rae, and Colin Fidge [26] to improve Young's approach by making it more efficient.

Rae's approach relies on the category of information manipulated by the component to prioritize and identify the critical components. The assumption here is that the application architecture model consists of components and connections. These connections carry the information from one component to another while the components can generate or manipulate the information to achieve the application's goal.

Based on the carried information, components and connections are categorized into two categories, First, Data Connection/Component where classified information is carried or manipulated by the Data component/connection. Classified data is defined as critical and sensitive data to the application or external world. Second, Control Connection/Component where non-classified information is carried by the Control connection/component. Unclassified information is defined as non-critical data to the application or external world. If a component/connection acts as both 'Data' and 'Control', then we classify it as 'data' to protect the classified information within.

Fig. 4 demonstrates N-Versioning the compartmentalized critical component highlighted in red. The nFork acts as the interface layer, it executes the N-Versions with the incoming parameters while the nJoin acts as the decision layer where it receives all N-Version outputs and executes the decision algorithm to come to a consensus.

			Standalone			
Number	**Test Case**	**Test For**	**C/C++**	**Java**	**C#**	**NVASA**
1	"73204483711"	Integer Over flow in Key Length	×	✓	×	✓
2	"zz...." in the plaintext or key field	Input Injection (Non HEX) in the plaintext and/or key field	×	×	×	✓
3	Null Input (Command Line or File)	Exception Handling with plaintext and/or key	✓	×	✓	✓
4	"Ree" instead of an Integer	Type Injection in the Key length field	×	✓	✓	✓
5	Normal HEX 192 Key and 128 Key	Normal Input	✓	✓	✓	✓
6	Larger key than specified (200 bits instead of 192 bits)	Input Validation Key Field	✓	✓	×	✓

✓ Program Passed Test Case × Program Failed Test Case

Table 1. Results of Testing NVASA Implementation Vs. the Standalone Versions

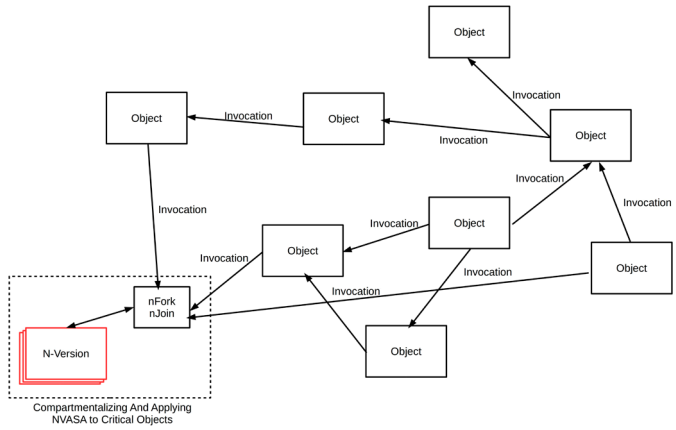

Figure 4. Applying the NVASA framework to the Critical Components

Conclusion and Future Work

The NVASA framework achieves better application security for critical web services, online, and cloud computing applications and improves the overall system security by using the N-Version programming methodology that comes to a consensus based on all the N-Versions outputs before applying any of these commands to the Backend Application Server Layer. This prevents any exploitation of the system which would lead to the destruction, modification or leakage of the confidential information to malicious users.

We showed the effectiveness of the NVASA architecture via the AES implementation, and how the decision layer is not only detecting attacks but also preventing the propagation of the effects of an attack. We showed with regards to some test cases how the decision algorithm was able to identify the irregularity among the N-Versions outputs, this was true even when all the N-Versions failed. This was achieved because each version failed uniquely producing a distinct output. Hence, enabling the decision algorithm to detect and prevent any of the malformed outputs from propagating further through the system.

In addition, we demonstrated compartmentalizing the application and applying the NVASA framework to the critical components as a method to reduce the added overhead associated with the implementation of the N-Version methodology.

Future work involves dealing with attacks targeting the state of the machine, for example backdoors etc. To that end we are further investigating the compartmentalization of components and its effects to real world applications in conjunction with automated code generation.

ABOUT THE AUTHORS

Majid Malaika received his B.S degree in Computer Science from King Abdul Aziz University. He received his M.S degree in Computer Engineering and his Doctorate Degree in Software Engineering from Southern Methodist University. Currently, he is an application security analyst at Amplify, a leading educational technology company in New York, USA. Majid's prior engagements were security consulting for multinational financial firms in New York. His work experience includes architecture risk analysis, risk management, code review, and penetration testing.

E-mail: majid.malaika@gmail.com

Suku Nair received his B.S. in Electronics and Communication Engineering from the University of Kerala, India. He received his M.S. and Ph.D. in Electrical and Computer Engineering from the University of Illinois at Urbana in 1988 and 1990, respectively. Currently, he is a Professor and Chair of the department of Computer Science and Engineering SMU at Dallas where he held a J. Lindsay Embrey Trustee Professorship in Engineering. His research interests include Cyber Security and Software Defined Networks.

E-mail: nair@lyle.smu.edu

Frank Coyle is a Senior Lecturer at SMU. He received his BS degree from Fordham College, his MS from Georgia Tech and his PhD from Southern Methodist University. His areas of interest are software engineering, software security, and engineering education. He maintains a technology blog at www.drczone.com.

E-mail: coyle@lyle.smu.edu

Department of Computer Science and Engineering at Southern Methodist University
P.O.Box 750122 Dallas, TX 75275-0122
Phone: 214-768-3083
Fax: 214-768-1192

REFERENCES

1. Executive office of the President office of Management and Budget Washington, D.C. 20503 "E-Government Strategy" February 27, 2002.
2. U.S. Department of Commerce, Washington, D.C. 20233, "Quarterly Retail E-Commerce Sales 1st Quarter of 2011".
3. Mr. Jeff Hughes, Dr. Martin R. Stytz, Ph.D. "Advancing Software Security– The Software Protection Initiative".
4. IC3, "2008 Internet Crime Report" <http://www.nw3c.org/downloads/2008_IC3_Annual%20Report_3_27_09_small.pdf>
5. Networks and WhiteHat Security Solution, "Vulnerability Assessment Plus Web Application Firewall (VA+WAF)" June 2008. F5
6. John Viega & Gary McGraw. "Building Secure Software: How to Avoid Security Problems the Right Way" (Chapter 1, Pages 9-13) Addison-Wesley Professional Computing Series, Addison-Wesley, New York. 2001
7. Schneier, Bruce "Secrets and Lies" (Chapter 23, Page 358), New York: John Wiley & Sons, 2000.
8. Cowan, C. Wagle, F. Calton Pu Beattie, S. Walpole, J., "Buffer overflows: attacks and defenses for the vulnerability of the Decade" 2000.
9. Guy-Vincent Jourdan, "Command Injection" 2005.
10. Chris Anley, "Advanced SQL Injection In SQL Server Applications" 2002.
11. scut / team teso, "Exploiting Format String Vulnerabilities" 2001.
12. President's Information Technology Advisory Committee. "Cyber Security: A Crisis of Prioritization19": Report to the President. National Coordination Office for Information Technology Research and Development, February 2005.
13. DISA for the DoD, "Application Security and Development" STIG, V2R1.
14. Richard Kissel & Kevin Stine & Matthew Scholl & Hart Rossman & Jim Fahlsing & Jessica Gulick, "Security Considerations in the System Development Life Cycle" NIST Special Publication 800-64 Revision 2, 2008.
15. Alexander Ivanov Sotirov, "Automatic Vulnerability Detection Using Static Source Code Analysis Thesis", 2005.
16. Michael Sutton & Adam Greene & Pedram Amini, "Fuzzing: Brute Force Vulnerability Discovery", 2007.
17. Etoh, Hiroaki, & Kunikazu Yoda, "Protecting from stack-smashing attacks" <http://www.trl.ibm.com/projects/security/ssp/main.html>
18. Frank Eigler, "Mudflap: Pointer use checking for C/C++" 2003.
19. A.A. Avizienis, "The Methodology of N-version Programming", Software Fault Tolerance, edited by M. Lyu, John Wiley & Sons, 1995, pp. 23-46.
20. Michael R. Lyu, Algirdas Aviz˘ienis , "Assuring Design Diversity in N-Version Software: A Design Paradigm for N-Version Programming".
21. Majid Malaika, Suku Nair, and Frank Coyle "Application Security Automation for Cloud Computing" CloudComp 2010.
22. Neal R. Wagner, "The Laws of Cryptography: Java Code for AES Encryption" 2001. <http://www.cs.utsa.edu/~wagner/laws/AESEncryptJava.html>
23. James McCaffrey, "Keep Your Data Secure with the New Advanced Encryption Standard" 2003. <http://msdn.microsoft.com/en-us/magazine/cc164055.aspx>
24. Niyaz PK, " Advanced Encryption Standard (AES) " <http://www.hoozi.com/posts/advanced-encryption-standard-aes-implementation-in-cc-with-comments-part-2-decryption/>
25. "Common Criteria Information Technology Security Evaluation", 1999
26. Andrew Rae, Colin Fidge, "Identifying Critical Components During Information Security Evaluation"