Available online at www.sciencedirect.com
ScienceDirect
Information and Software Technology 49 (2007) 381–397
INFORMATION AND SOFTWARE TECHNOLOGY
www.elsevier.com/locate/infsof

On the design of more secure software-intensive systems by use of attack patterns

Michael Gegick *, Laurie Williams

North Carolina State University, Department of Computer Science, 890 Oval Drive, Campus Box 8206, Raleigh, NC 27695, USA

Received 26 July 2005; received in revised form 5 June 2006; accepted 9 June 2006
Available online 4 August 2006

Abstract

Retrofitting security implementations to a released software-intensive system or to a system under development may require significant architectural or coding changes. These late changes can be difficult and more costly than if performed early in the software process. We have created regular expression-based attack patterns that show the sequential events that occur during an attack. By performing a Security Analysis for Existing Threats (SAFE-T), software engineers can match the symbols of a regular expression to their system design. An architectural analysis that identifies security vulnerabilities early in the software process can prepare software engineers for which security implementations are necessary when coding starts. A case study involving students in an upper-level undergraduate security course suggests that SAFE-T can be performed by relatively inexperienced engineers who are not experts in security. Data from the case study also suggest that the attack patterns do not restrict themselves to vulnerabilities in specific environments.

Keywords: Software and system safety; Patterns

1. Introduction

Retrofitting security implementations to a released software-intensive system or to a system under development may require significant architectural or coding changes. The cost associated with such change increases as a software development team progresses through the software lifecycle. For example, the cost associated with identifying and fixing a software problem after the release can be 100 times more expensive than if found in the requirements phase [3]. Thus, an early knowledge of what security vulnerabilities are present can reduce the high costs of fortifying (e.g., modifying and/or adding code) a product under development.

Unfortunately, many security practices in industry today do not start until late in the software process. Static analysis tools, white/black box testing, and red teaming are methods that usually take place once source code has already been written. If a vulnerability is found using any of these approaches, then securing the vulnerability may require many other changes to the software. Also, the lack of knowledge in security among software engineers often necessitates the involvement of a security expert/auditor to identify the vulnerabilities [22,24,13]. We propose a technique that would enable software engineers to build security in to the product from the start of the lifecycle.

This paper presents a systematic approach for identifying security vulnerabilities in software-intensive system designs to enable early mitigation of security vulnerabilities and software security fortification strategies by software engineers who may not be security experts. We propose attack patterns that are based on the software components involved in an attack and are tailored for identifying vulnerabilities in system designs. Most of the attack patterns are best fitted for design representations such as data flow diagrams that show components and the data flow between them, but are not restricted to data flow diagrams. When a potential vulnerability is found, the software engineer

* Corresponding author. Tel.: +1 919 513 4151.
E-mail addresses: mcgegick@ncsu.edu (M. Gegick), lawilli3@ncsu.edu (L. Williams).

0950-5849/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2006.06.002

either designs (e.g., changes the architecture of their system or inherits a new object-oriented model) for a safer system or is forewarned to make fortifications before implementing the system. Thus, security is integrated into the software as the software is being designed and built.

To develop an understanding of the kinds of vulnerabilities that software engineers frequently overlook during software design, we studied 244 vulnerabilities from four vulnerability databases (SecurityFocus,[1] Help Net Security,[2] Secunia,[3] SecurityTracker[4]) in late 2003 and early 2004. The vulnerability descriptions included the components[5] in the software system and the actions the attacker used to attack those components. The descriptions were abstracted into simple regular expressions to provide a generic representation of the vulnerability. The events in the regular expressions are symbolized by the component in the system that triggered the event, and we call these expressions *attack patterns*. The vulnerabilities we analyzed abstracted to 53 regular expression-based attack patterns. These attack patterns comprise our initial attack library (AL) of abstractions. Software engineers can then conduct a ***S**ecurity **A**nalysis **F**or **E**xisting **T**hreats* (SAFE-T) [9] by matching the attack patterns in our AL to their system design. In this paper, we show that doing so helps software engineers identify vulnerabilities before coding starts. SAFE-T involves the search for commonly overlooked threats because the vulnerabilities in the AL are known to exist in other applications.

Because SAFE-T was designed for software engineers who are security novices, we validated SAFE-T within the context of undergraduate computer science courses at North Carolina State University (NCSU). The studies were intentionally run with undergraduate computer science students because a research goal involved the development of a security vulnerability identification technique that could be used effectively by relatively inexperienced engineers without expert knowledge in security. The data were then analyzed to determine if the students could accurately map the attack patterns to vulnerabilities in the design.

In this paper, we provide a process for formulating an attack library and instructions on mapping the patterns to a system design. In Section 2, prior research is presented. In Section 3, the approach in this paper is explained. In Section 4, the results of the empirical studies are shown. Finally, in Section 5 a summary and discussion of future work is presented.

[1] http://www.securityfocus.com, owned by Symantec.
[2] http://net-security.org/, a privately owned company.
[3] http://secunia.com/, an IT security company.
[4] http://www.securitytracker.com/, owned by SecurityGlobal.net LLC.
[5] One of the parts that make up a system. A component may be hardware or software and may be subdivided into other components. The terms "module," "component," and "unit" are often used interchangeably or defined to be sub-elements of one another in different ways depending on the context [3].

2. Background and related work

In this section, we first define terms that are used throughout this paper and then we discuss past research on vulnerability identification. We then show classification schemes that organize vulnerabilities according to their characteristics.

2.1. Background

Before continuing, definitions (from seminal references, e.g., IEEE [14], as available) of essential words are provided to aid in understanding our approach:

- *Abstract-and-match security identification technique*. A technique for abstracting vulnerabilities into a generalized form so that they can be matched to a diverse set of systems.
- *Attack*. [An event that] occurs when an attacker has a motive or reason to attack and takes advantage of a vulnerability to threaten an asset [13].
- *Attack pattern*. A model that describes how to execute an attack [12]. Just as design patterns [8] show the foundational details of object-oriented designs that allow one to build new systems by inheriting the information captured in the pattern, attack patterns abstract the basic properties of an attack to identify where vulnerabilities may be present.
- *Design*. The architecture, components, interfaces and other characteristics of a system or component [14].
- *Error*. A human action that produces an incorrect result [14].
- *Exploit*. An exploit is synonymous with attack in this paper.
- *Threat*. A potential event that will have an undesired consequence (e.g., information leakage or denial-of-service) [13].
- *Vulnerability*. An instance of an error in the specification, development, or configuration of software such that its execution can violate the [implicit or explicit] security policy [of a software vendor] [15].

2.2. Abstract-and-match techniques

Several abstract-and-match security vulnerability identification techniques have been developed. With these techniques, known vulnerabilities are *abstracted* to a high-level representation, excluding details that make the vulnerability specific to the system or context in which it was found. Subsequently, software developers search for and *match* these generalized, abstracted vulnerabilities in the same or different systems from which they are originally found. In this section, we discuss the progression of abstract representations from text-based to graph-based techniques. Our approach builds upon these and represents vulnerabilities as regular expressions.

The abstract-and-match technique may have been first proposed by Carlstedt et al. [4]. In their work, they abstracted source code from resource-sharing operating systems such as Multics, TENEX, and EXEC-8. Vulnerable source code was paraphrased using abstract terms to generalize components in the system that were specific to the vulnerability. For example, they used "abstract cell" to represent any storage device such as a buffer, variable, or disk. Source code of the same or different operating systems was then examined for the presence of these representations to find a vulnerability. This approach requires code, suggesting that the technique is used relatively late in the software process.

Bruce Schneier [21] developed a graph-based representation of attack scenarios called *attack trees*. Attack trees show the steps an attacker can take to attack a system. Each root node in the attack tree represents a goal of an attacker, and the children of the roots are the sub-goals necessary to achieve the goals represented by their parents. The nodes of an attack tree can reference an action performed by the attacker on the software system or may indicate an action that an attacker performs outside of the software system (e.g., threat or bribe). The attack trees can be applied to system architectures to identify attack scenarios that may occur. This matching process may be achieved in the design, code, or testing phases of a software process. System designs do not typically show actions of an attacker and so matching the goals of an attacker to a system design is difficult.

Attack trees represent the point of view of an attacker that does not necessarily have detailed information about the software system. A detailed model of the system (e.g., a data flow diagram) may offer a more sufficient means of identifying more attacks than attack trees. Our research analyzes a system design for potential attacks based on the components in the system that makes those attacks possible. By associating the components in the system with attacks that have been performed on those components, a component-level perspective is created for the use of addressing security defenses. Once the vulnerable components (or sequences of components) are identified in a system design, attack trees may be used to identify the goals of an attacker and to further delineate possible steps in attack.

Another graph-based approach for securing applications involves the use of *attack nets* [18]. Attack nets are derived from Petri nets and include "places," which are analogous to nodes in an attack tree in that a place shows a state of an attack. Transitions occur between places to show what events are required to move from place to place. Arcs connect places and transitions to show the path the attacker takes. Attack nets can be used to show concurrency of events via multiple tokens progressing along the arcs and through the transitions to show which place is achieved at what time. Brief textual notes are included with the places and transitions to describe what must occur for the attacker to be successful. As with attack trees, attack nets can show what attack scenarios are possible in a system. Attack nets can be applied during penetration testing, but may also be used for vulnerability assessment in software designs. If an attack cannot be expressed in the form of a tree, then an attack tree or attack net may not be conducive for describing the attack.

Attack nets were advanced with text-based descriptions by Steffan and Schumacher [23]. These researchers included explicit pre- and post-conditions as textual descriptions to describe transitions and made them available through a WikiWikiWeb [7] called an ATiki. In an ATiki, conditions and transitions are hyperlinked to editable web pages where users can collaborate on describing what occurs during an attack. Furthermore, the context of an attack is also available on a different Wiki page, which describes the environment in which the vulnerability may occur. In Steffan and Schumacher's study of vulnerabilities with the PHP scripting language [23], accurate descriptions of attacks were created to support the graph-based attack nets with the help of individuals collaborating on the ATikis.

Moore et al. [20] proposed a formal description of attacks called *attack profiles*. Attack profiles consist of text- and graph-based descriptions of an attack. A designer or analyst reads the graph and text description in the attack profile and determines if the profile can be matched to their system architecture. If a match occurs, then attack patterns (in the form of attack trees) are applied to the place in the system that the attack profile describes. Including both text- and graph-based approaches in one technique may allow for more accurate identification of a vulnerability. Our research follows the idea that both a graphical and a textual representation to describe attacks can be beneficial for identifying vulnerabilities.

2.3. Automating the abstract-and-match technique

The process of matching vulnerabilities has been automated with static analysis tools. These tools do not require that the code be executing. Security vulnerabilities can be represented as finite state automata (FSA) to show a sequence of events that may result in an attack [6]. A static analysis tool can search for patterns based on the FSA and warn developers of potential vulnerabilities. Performing static analysis in the development phase of the software process has been shown to be an efficient means of verifying the existence of vulnerabilities [5]. However, fixing the vulnerabilities after the code is written can be more costly than if the vulnerabilities were known earlier.

Bishop and Dilger [2] experimented with an automated pattern-directed search on source code of C applications to identify race conditions on UNIX operating systems. The type of race condition studied is termed time-of-check-to-time-of-use (TOCTTOU) which describes a condition where a system first checks a characteristic of an object (e.g., a file) and then performs a second event that depends on the characteristic of the first event. They used an automated scanner to scan the code in the *sendmail* version 8.6.10 application for two file system calls to the same

file which would indicate a potential race condition. The analyzer found 24 instances of consecutive calls to the same file in the *sendmail* application. Only five of the 24 pairs were susceptible to the race condition. One of the five pairs was a previously undiscovered flaw and was made known to the program vendor. This evidence indicates that abstractions can be used to find both known and new vulnerabilities.

2.4. Vulnerability taxonomies

Many intrusions are based on a small number of known attacks, implying that the underlying vulnerabilities are also the same or similar [16]. Vulnerabilities can be attacked for years after their discovery [1] and thus should not be discounted based on their age. A taxonomy of these recurring vulnerabilities allows for previous knowledge to be applied to new attacks as well as providing a structured way to view such attacks [11]. We provide introductory information on three such taxonomies and will make a more detailed comparison with these taxonomies in Section 4.1.

- McGraw and Hoglund [12] describe 49 attack patterns in a text-based format. Specific instances of the attack patterns are given to exemplify how a vulnerability is attacked.
- Landwehr et al. [17] provide a three-tiered (Genesis, Time of Introduction and Location) taxonomy with multiple categories.
- Krsul [15] provides four classes in his taxonomy: Design Flaws, Environmental Flaws, Coding Flaws, and Configuration Flaws. Krsusl expands on the Environmental Flaws classification, which is geared towards the identification of the assumptions that programmers make about their environment and whose violation result in software vulnerabilities.

We contribute a taxonomy tailored for detecting vulnerabilities in the system design phase of the software process using attack patterns.

3. Research approach

We theorize that a vulnerability in a system could be represented by showing the events that transpire during an attack. We use regular expression-based attack patterns to denote the sequence of components that trigger the events in an attack. One can then match the components in the attack pattern to a corresponding sequence of components of a software-intensive system design.

To devise our regular expression-based attack patterns, we first analyzed online vulnerability databases to gather information about vulnerabilities from current software systems. The information supplied in the vulnerability databases was then encoded into attack patterns. In many instances, multiple similar attacks were abstracted into one pattern. In Section 3.1, we present how we gathered vulnerability information. In Section 3.2, we provide information on how we made our regular expression-based patterns. In Section 3.3, we discuss our limitations.

3.1. Vulnerability information

Four web-based security vulnerability sources (SecurityFocus; Help Net Security; Secunia; and SecurityTracker) were investigated to fuel the study with information about vulnerabilities. The vulnerabilities found in SecurityFocus are predominately posted by the individuals who discover the vulnerability, who may be customers or third-party researchers. The vulnerabilities posted at Help Net Security are, in general, reprinted from the BugTraq (also owned by Symantec) mailing list. Vulnerability information in Secunia is drawn from credible sources, and validations are performed as much as possible. SecurityTracker obtains most of its information from third-party researchers, and validations are also performed before publishing. The vulnerabilities posted on these web sites are generally discovered from "ethical hacking," where vulnerabilities are sought for learning, rather than malicious objectives. There is no clear knowledge of how many times these vulnerabilities have been maliciously exploited on live applications.

Analysis of the vulnerability descriptions focused on the events that occurred during the discovery of the vulnerability and on the components in the system that triggered those events. A useful description in the database included a reference to specific components and events that occurred at the component. If this detail was not present in the description, then an example of the attack that was sometimes given in the vulnerability descriptions could possibly illuminate the vulnerability. Sometimes example source code of the attack was made available to explain the procedure to attack the vulnerability. If a description did not adequately include these details, then the vulnerability was excluded from the study.

3.2. Attack patterns

Representing vulnerabilities as patterns can be useful for illuminating a problem that can occur in multiple and different software applications. Generic solutions to the problems represented by attack patterns should be abstract enough to encapsulate solutions for the same problem in many contexts, as is done with design patterns [8].

Our attack patterns are based on a set of components (or alphabet) that we observed in our vulnerability analyses. Appendix A is a subset of 76 symbols of our 103-symbol alphabet. The subset was chosen to correspond with our validation study which will be discussed in Section 4.3. The attack patterns begin with a "start" event, symbolized by the component (from alphabet found in Appendix A) that can be used to initiate the attack. Each major successive event in the attack is expressed by the component that triggered the event and is appended to the string that

begins with the start component. The *threat target* [13] that represents the final objective of the attack terminates the string of symbols to complete the illustration of the attack path. We used four regular expression operators in the attack patterns to further clarify the characteristics of an event (see Table 1).

For example, the attack pattern,

$$(\texttt{Client}^{+})(\texttt{Server}^{+})(\texttt{LogFile}^{+})(\texttt{HardDrive}^{+})$$

is composed of four symbols from our alphabet: `Client`, `Server`, `LogFile`, and `HardDrive`. This attack pattern is read as a series of `Client` (the start component) requests, followed by a series of `Server` actions, followed by a series of log updates to the `LogFile`, followed by a series of disk writes to the `HardDrive` (the threat target). The access log records an entry for each request and if enough requests are made, then the hard drive can be consumed by the access log file. The abstracted terms do not clearly indicate what event occurs in an attack, but only show the components involved in the attack. Thus, we use a text-based attack profile derived from Moore et al. [20] to accompany the pattern for clarification purposes. In this example, the attack profile is a client can exceedingly access a server that logs accesses to the hard drive. If permitted, the log file may become large enough to fill the hard drive causing the system to crash. Upon matching an attack pattern to a vulnerability, an end-user of SAFE-T can verify that the match is authentic by reading the attack profile to assure such an attack is plausible.

Howard and LeBlanc [13] suggest representing vulnerabilities with regular expressions to provide an automated means of identifying a class of threats rather than one describing a specific problem. Our patterns are abstracted and expressed as generically as possible to accomplish flexible usage in multiple systems. The same generalized vulnerability may involve an attack on a database, FTP, or audit server environment. Therefore, the pattern is generic enough to represent each of these three different types of servers with one general `Server` symbol. Also, the same attack may occur if an error log file becomes large, and thus the symbol, `LogFile`, is generic enough to represent either access log or error log. Unlike attack trees, each event in our attack patterns is represented by a component; nodes of attack trees are represented by goals of the attacker.

In Fig. 1, we show a simplistic example of a system design that reflects the log file vulnerability. The components in our system designs are numbered to provide a means of representing an attack path. There are two attack paths in Fig. 1 that correspond to $(\texttt{Client}^{+})(\texttt{Server}^{+})(\texttt{LogFile}^{+})(\texttt{HardDrive}^{+})$ attack pattern: 1–2–3–5 and 1–2–4–5. Each attack path represents a string of symbols over the alphabet. Upon the match, a security team now has a graphical representation of the attack path and the vulnerability. Including a graphical representation follows the steps of previous work and aids those who think visually.

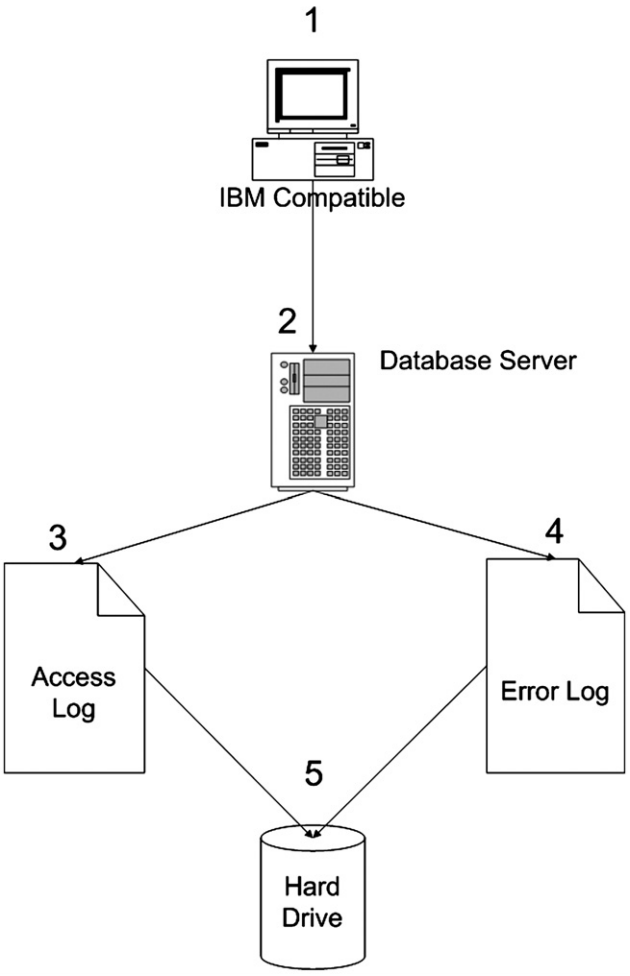

Fig. 1. Sample system design.

Regular expressions can make use of the ? operator which means the event may not happen or can happen at most once. In this example,

Table 1
The four regular expression operators that symbolize events in an attack path

Operator	Description
Kleene closure (*)	An event may occur zero or more times
⊕	The event to the left or right of the operator will occur, but not both
$^{+}$(superscript)	The event occurs at least once
?	The event zero times or once

(Client)(CommandLineArgumentEntry)
(ApplicationServer?)(Application)
(CommandLineArgumentBufferWrite)
(Buffer),

a Client works on an application, and enters an excessively long CommandLineArgument, which is read by an Application, which may or may not be on an ApplicationServer, followed by writing the CommandLineArgumentBufferWrite, followed by the data overflowing the Buffer. The ? allows the regular expression to represent that a standalone or sever-based environment is susceptible to the same attack (the presence of an ApplicationServer is not a requirement but is possible). Also, note that the events CommandLineArgumentEntry and CommandLineArgumentBufferWrite do not represent components in the system. These events are inserted into the regular expression to clarify operations that happen before or after the event that occurs at the component.

Lastly, the exclusive or, ⊕, operator can be used to show that either the event to left or to the right of the operator occurs, but not both. In this example,

(Client)(Variable ⊕ Filename ⊕ Header)
(HTTPServer)(PostMethod)(BufferWrite)
(Buffer),

a Client interacts with a web server, makes a POST request with either a long Variable or Filename or Header, followed by the HTTPServer accepting the request, followed by the PostMethod processing the request, followed by the Variable or Filename or Header BufferWrite, followed by the Buffer overflowing. Any of the Variable, Filename, or Header can be used to cause a buffer overflow. Only one of these events is needed to attack a small buffer on the vulnerable server.

Security testing can be achieved by basing test cases on attack patterns and abuse/misuse cases [19]. Testers can create malicious test cases that perform the sequence of events represented in our attack patterns. If the test successfully exploits a vulnerability, then the attack pattern accurately described the vulnerability in the system. If the test case fails to exploit the system, then either the attack pattern did not accurately describe a threat to the system, the test case was not properly implemented, or fortification are in place.

3.3. Limitations

Our attack patterns are limited to describing attacks where a sequence of events occur that lead to the compromise of a digital asset. Vulnerabilities that cannot be described with events, such as configuration errors, cannot be expressed with our attack patterns. For example, if a system administrator permits an attacker to read a privileged file because access right are not properly set, then the attack cannot be represented with an attack pattern. Our attack patterns can represent events where either one or many events occur in a sequence. Also, our attack patterns are limited to attacks where the sequence of events must occur in a specified order; our attack patterns cannot easily represent different temporal orders of a sequence of events.

Not all the symbols in the alphabet are atomic and can thus reduce the precision of effective pattern matching. For example, the Server symbol should be broken down into atomic components such as the read operations from the Client and the handling of the data from that Client. A more precise alphabet can lead to more accurate attack patterns and thus less false positives (see Section 4.3). Refining the attack patterns can serve as a means to perform mathematical (e.g., theorems about necessary or sufficient conditions for attacks to occur) or statistical (e.g., Given that a partial sequences of events has occurred, what is the probability that an attack can be completed?) analyses. Reducing each symbol in the alphabet will increase the number of symbols and thus increase the complexity and difficulty of SAFE-T. Each component would also have to be included implicitly or explicitly in the system design for a match to occur. There should also be a means to determine the order of what components and/or events can occur sequentially in the regular expression. This problem is analogous to regular expressions which define legal arithmetic expressions for the order of operands and operators. For example, regular expressions can prevent an instance of two operands occurring next to each other. Indicating which components can occur consecutively in an attack pattern may help for more definitive pattern matching.

4. SAFE-T: Regular expression-based attack patterns

We now discuss the development of 53 attack patterns, how we validated the efficacy of encoding vulnerability information in an attack patterns, and the results of empirical studies examining how well the patterns can be matched to a system design.

4.1. Attack pattern development

A total of 414 vulnerabilities were collected from the four online databases. One hundred and seventy (41.1%) of these vulnerabilities did not meet the selection criteria (see Section 3.1) required for creating attack patterns. Table 2 summarizes the type of vulnerabilities excluded from the study. Eighty-five (20.5%) of the vulnerabilities lacked descriptions with sufficient detail in SecurityFocus to form patterns. Often, Help Net Security, Secunia, and SecurityTracker contained the similar levels of detail suggesting the lack of information made

Table 2
Classes of vulnerabilities not used

Description	Frequency
Lack of information	85 (20.5%)
Specific to vendor	48 (11.6%)
Inapplicable	21 (5.1%)
Networking	14 (3.4%)
Encryption	1 (0.2%)
Hardware	1 (0.2%)

available to the public. Therefore, using online vulnerability databases alone may not be a sufficient means in providing information for research.

There were 48 (11.6%) vulnerabilities that were specific to vendors, such as Norton Antivirus crashing when scanning files in certain folders, and would not likely serve helpful in the general protection of typical software applications. Retaining vulnerabilities that are specific to vendors would increase the size of a pattern AL and thus decrease the efficiency of matching the enumerated patterns to components in the system design in the general case. Furthermore, proprietary code makes validating an attack more difficult because of the lack of information made available to the public.

Twenty-one (5.1%) of the observed vulnerabilities could not be represented as a sequence of events triggered by the components in a system. These included: (1) the failure to secure permissions to a file; (2) file upload ability that allowed users to open files on a server: (3) passwords kept in plaintext, (4) timed attacks to steal passwords, and (5) configuration errors. Our attack patterns rely upon on the interaction of multiple components, and thus cannot be used to abstract these single-component types of attacks.

The scope of this study was limited to common software application coding problems. Therefore, networking vulnerabilities were excluded. Networking attacks made up 14 (3.4%) of the vulnerabilities found and included vulnerabilities at the packet level, network protocols, port scan, and switch vulnerabilities. One (0.2%) vulnerability was a hardware problem that allowed an attack to obtain secret keys in a module's run-time memory. There was also one encryption vulnerability that existed because the encryption was too weak to secure user passwords. These classes of attacks are valid and detrimental to software systems, but do not fit the software coding schemes that attack patterns express. Therefore, other techniques are needed in tandem to support the wide variety of vulnerabilities.

A total of 53 patterns were abstracted from the 244 remaining vulnerabilities. An extensive sample of 30 of the attack patterns and associated profiles of the AL is provided in Appendix B. This subset is provided to correspond with our validation study discussed in Section 4.3. The right column shows the percentage of the 414 vulnerabilities that the attack pattern represents. The full set of attack patterns can be found in [9].

4.2. Taxonomy comparison

We compared our attack patterns to the taxonomies of Hoglund and McGraw [12], Landwehr [17], and Krsul [15] (see Table 3) to determine if our attack patterns could identify vulnerabilities from well-established authors in the security field, and go beyond what has been done before. Approximately two-thirds of our attack patterns described vulnerabilities abstracted by Hoglund/McGraw and Krsul and 100% with Landwehr. The specifics of the mapping between our attack pattern-based taxonomy and these taxonomies are described with each attack pattern definition in Appendix B.

Four of our 21 buffer overflow vulnerabilities mapped directly to specific buffer overflow vulnerabilities published by Hoglund and McGraw [12]. Our remaining 17 buffer overflow vulnerabilities can map into their general content-based buffer overflow category. Twelve more of our attack patterns can also be mapped to specific attack patterns in their work. Thus, 16 (30.2%) of our vulnerabilities map directly to those by Hoglund and McGraw [12], and a net 33 (62.2%) of our regular expression-based attack patterns can describe 14 (28.6%) of their attack patterns. The patterns proposed by Hoglund and McGraw [12] do not offer a graphical representation that may aid in the understanding of a vulnerability. With SAFE-T, a user creates the graphical representation by drawing a path (implicitly or explicitly) in the system design to show what components are involved in an attack and where the vulnerability occurs.

The taxonomy provided by Landwehr et al. [17] consists of three major categories: Genesis, Time of Introduction, and Location. The Genesis classification organizes vulnerabilities into 13 categories based on whether or not those vulnerabilities entered into the system via accident or were introduced maliciously. The Time of Introduction classification attempts to classify in which of the five software phases of the software process the vulnerability is introduced. Lastly, the Location classification distinguishes vulnerabilities into 11 categories based on whether they are found in the software (e.g., application or operating system) or in the hardware. The taxonomy describes vulnerabilities at a higher level of abstraction relative to our attack patterns, which allows for 100% of our patterns to be mapped their taxonomy (see Table 4). The degree of abstraction also permitted our attack patterns to map to multiple categories within the taxonomy. Our attack patterns consist of components that specify where a potential vulnerability exists in a system design.

Table 3
Taxonomy comparison

Source	No. of SAFE-T attack patterns matching (%)	Total number of attack classifications
Hoglund, McGraw	33 (62.2%)	49
Landwehr	53 (100%)	29
Krsul	35 (66.0%)	54

Table 4
A mapping between Landwehr et al. and SAFE-T patterns

Category	Division	No. of SAFE-T attack patterns matching (%)
Genesis	Boundary Condition	25 (47.2%)
	Validation error	17 (32.1%)
	Other exploitable logic error	11 (21.0%)
Time of introduction	Source code	53 (100%)
Location	Hardware	5 (9.4%)
	Privileged utilities	1 (1.9%)
	Application	47 (88.7%)

Table 5
A mapping between Krsul's classification and SAFE-T patterns

Division	No. of SAFE-T attack patterns matching (%)
2-2-1-1 (user input content is at most x)	20 (37.8%)
2-2-1-3 (user input for file content matches regular expression)	8 (15.1%)
2-2-1-4 (user content is free of shell metacharacters)	2 (3.8%)
2-7-2-5 (user input file content is of a known type)	2 (3.8%)
2-5-1-1 (user input command line content length is at most x)	1 (1.9%)
2-3-2-3 (environment variable content matches regular expression)	1 (1.9%)
2-8-4 (Directory Permissions Mode)	1 (1.9%)

Table 6
Student answers

	Implication	Feasibility	Validation
False negatives	Vulnerability persists (high risk)	2% ($N = 23$)	4% ($N = 84$)
True positives	Vulnerability identified	91% ($N = 937$)	90% ($N = 2067$)
False positives	Time wasted (low risk)	7% ($N = 65$)	6% ($N = 155$)

Table 7
Non-existent vulnerabilities

	Feasibility	Validation
(`Class`)(`Subclass`)(`OverriddenSecuredMethods`)(`Application`)		
True positive	3	15
False positive	24	22
Marked as not present in the design	14	10
(`Client`)(`GUI/Browser`)(`BookMarkSave`)(`Bookmark BufferWrite`)(`Buffer`)		
True positive	46	43
False positive	1	11
Marked as not present in the design	2	6

The data suggest that all of the vulnerabilities we analyzed are Source Code vulnerabilities. If the attack patterns were known at the time of the design phase, then early warnings could alert developers before coding starts. Our study indicates that the most likely method of introducing a security flaw into a software system is via improper bounds checking evidenced by 25 (47.2%) buffer-related attack patterns. Lastly, the 47 (88.7%) of the attack patterns are involved with software applications as opposed to other areas of attacks such as networks.

Krsul contributes a tree-based taxonomy of software vulnerabilities that can be expressed by numbers separated by hyphens (e.g., 2–2–1–2). At the top level (represented by the first number) of the tree, there are four questions that determine the basic category of the software vulnerability. All of our software vulnerabilities fall into question number two, which asks "Is the vulnerability the result of the implementer making simplifying assumptions about the environment in which the program was going to be run, and if this assumption were to be true the vulnerability would not exist?" [15] Thus, each of our attack patterns map to a vulnerability in the second of four categories of Krsul's taxonomy and so begin with "2." Each successive level of the tree (represented by a number) further classifies the type of vulnerability. We mapped 35 (66%) of the attack patterns presented in this research to seven (13%) of the classifications proposed by Krsul [15] (see Table 5). Krsul's taxonomy shows that the 35 attack patterns are derived from malicious

Table 8
A subset of 76 symbols in our alphabet

Component/event	Description
Application	A software product
ApplicationServer	A server that serves an application
AuthenticationRoutine	A routine responsible for authenticating a client on a software system
BookmarkBufferWrite	Writing a bookmark to a buffer
BookMarkSave	Saving a bookmark to disk
Buffer	Storage area in memory for data
BufferWrite	Writing data into a buffer
Class	A blueprint for an object at the source code level
Command	An instruction executed by system
CommandLineArgumentEntry	A command is entered into the shell of a operating system
CommandLineArgumentBufferWrite	Writing the command entered in the shell into a buffer
Cookie	A text file on a client's machine that stores information related to that client
CPU	Central processing unit
Data	Any type of information
Database	A data structure used for storing information
EmailHeader	Header information stored for emails
EnvironmentVariable	Variable that represents info about environment in which application running
EnvironmentVariableWrite	Writing an environment variable into a buffer
FileHeader	A header that supplies metadata about a file
Filename	The name of a file
Firewall	A device that prevents network traffic from entering into network
FormData	Data that is entered into a form on a web page
FTPCommand	A command (e.g., PUT) in a FTP system
FTPRequest	A client's request to a FTP server
FTPServer	A server that handles FTP requests
GetMethod	Method in a server responsible for getting the information a client has requested
GetMethodBufferWrite	Information retrieved from the GET method is written to a buffer
GUI/Browser	A web browser or graphical user interface for browsing pages on the Internet
HardDrive	Stores information permanently on a machine
Header	Header information in a message or software component
HeaderFieldBufferWrite	Writing header information that describes the data (e.g., message) into the buffer
HTMLPage	A page that can be viewed on the web
HTTPContent-LengthHeaderValue	The value that represents the size of the HTTP content that is sent to a server
HTTPMessageHandler	The routine in HTTP server responsible for handling messages to/from client
HTTPMessagePayloadLength	The actual length of the payload in a HTTP packet
HTTPRequest	A request from a client to a HTTP server
HTTPServer	A server that handles HTTP requests
Hyperlink	An HTML tag used for linking documents on the Internet
HyperlinkBufferWrite	Writing the information stored in the hyperlink into a buffer
IncludeFile	A file that is included in source files such as C that contains information needed for the source file to compile
InjectionOfMaliousHTMLTags/scriptURL/Form	A clients inserts HTML tags or script (e.g., JavaScript) into a web page
IntegerEvaluationRoutine	The routine on a software system responsible for handling (e.g., read/write/compute) an integer
Log	A file that keeps track of events
LogEntryRead	Reading an entry in a log
Machine	A computer
MailCommand	A command for an email system
MailServer	A server that handles email
Memory	Random access memory
Message	Data sent from one component to another
MessageHeaderHandler	A method that is responsible for processing the headers of a message
Metafile	A file that contains information describing information

(*continued on next page*)

Table 8 (*continued*)

Component/event	Description
OSCommand	A command (e.g., ls or dir) for an operating system
OverriddenSecuredMethods	The methods of a subclass that override the methods of its parent class
PasswordEntry	A client enters their password
PostMethod	The method on a server that is responsible for saving the client's information
ProgramVariable	A variable in a program
ProxyServer	A server that caches requests from a user. Can also be integrated with a firewall system.
QueryParam	A parameter that is part of a query (e.g., In the form of a URL ⟨URL⟩?param=arg)
QueryString (search string)	A string of parameters and values in a URL. Can transmit the information entered in an HTML form.
ReadClientInput	Reading a client's input
RequestMessage	A message sent to a component that requests a service
Router	A device on a network responsible for routing network packets
Server	A software process that handles requests from clients
ServerConnectionState	The connection between a server and a client
ServerVariables	Variables a server uses to organize web page information (e.g., URL, sessionID)
SizeField	A filed in the metafile that describes the size of the file
SOAPServer	Simple Object Access Protocol, resp. for remote procedure calls over HTTP
SourceFile	A file that contains source code for an application
SQLInput	Input for a structured query language system
SQLInputField	A text box in an HTML page responsible for transmitting the data entered into a SQL database
Subclass	A class that inherits attributes and members from a parent class
SysAdmin	System administrator
SyslogFunction	The software routine that implements the syslog utility. This is responsible for logging events in systems such as UNIX
UserNameEntry	A client enters their username
Variable	A variable passed from the client to the server
WebApplication	An application that is available to clients on the Internet

user input. The largest mapping falls into the 2-2-1-1 (see Table 5 for explanation of category) category because 20 of our attack patterns are involved with buffer overflows, which is a very common tactic for attackers.

We have shown that approximately one third of our taxonomy offers new classifications of vulnerabilities not mentioned by any of Hoglund and McGraw's or Krsul's taxonomy. Contributing to the difference in taxonomies is the time in which the studies occurred. In the six years, since Krsul published his taxonomy, new vulnerabilities have appeared and old ones have become better understood. Hoglund and McGraw's taxonomy focuses on Internet-based applications and may not have included the same sample of vulnerabilities as ours. Also, our attack patterns show more detail than those provided by Landwehr et al. and may provide more information for software engineers to detect a vulnerability. Our goal was for our attack patterns to be more similar to the structure of system designs so that they may be more useful for identifying vulnerabilities in the system design phase of a software process.

4.3. SAFE-T validation

We ran two studies (described fully in [9] and briefly summarized in [10]) to determine if personnel with relative inexperience in security could perform a SAFE-T by matching a regular expression-based attack pattern to a vulnerability in a system design. The artifacts used in this study are documented in [9].

The first study, a feasibility study, was run with 43 students in an upper-level undergraduate security class at NCSU. The study consisted of 20 attack patterns seeded into a system design containing 16 components. The results of this initial study indicated the students were able to match the attack patterns to a system design. Thus, in the following semester a validation study was performed in the same course, but with 58 students, and 30 attack patterns (documented in Appendix B) and a more advanced design (see Appendix C) with 14 more components. The results of both studies are now discussed.

Student submitted answers of an attack path represented by numbered components in the system design that corresponded to an attack pattern, as exemplified in Fig. 1 of Section 3.2. A student answer was considered a false negative if he or she provided no attack path/attack pattern when in reality the vulnerability existed. False negatives imply that vulnerabilities will go undetected in the software process. Student answers were categorized as true positives if they provided an attack path/attack pattern that corresponded with a seeded vulnerability. If a student provided

Table 9
A subset of 30 attack patterns

Attack pattern and profile	*N* (%)
Buffer overflow attacks (21 patterns, 21.8% of the attack library) **(Client)(HTTPServer)(GetMethod)(GetMethodBufferWrite)(Buffer)** A client that submits an excessively long HTTP GET request to a web server may cause a buffer overflow. Either the requestURI or HTTP version may be too long for the buffer Hoglund & McGraw: Content-Based Buffer Overflow Landwehr et al.: Genesis – Boundary Condition Violation Time of introduction – Source Code Location – Application Krsul: 2-2-1-1	28 (6.8%)
(Client)((FTPCommand ⊕ MailCommand) ⊕ OSCommand)(FTPServer ⊕ MailServer)(BufferWrite)(Buffer) A client that submits an overly long OS command or FTP/Mail command may cause a buffer overflow in the FTP/Mail server Hoglund & McGraw: Content-Based Buffer Overflow Landwehr et al.: Genesis – Boundary Condition Violation Time of Introduction – Source Code Location – Application Krsul: 2-2-1-1	8 (1.9%)
(Client)(Variable ⊕ Filename ⊕ Header)(HTTPServer)(PostMethod)(BufferWrite)(Buffer) A client that submits an excessively long POST request via a Variable, Filename or Header, may cause a buffer overflow on the server. The POST request may be in the form of a hidden variable, filename or header Hoglund & McGraw: Content-Based Buffer Overflow Landwehr et al.: Genesis – Boundary Condition Violation Time of Introduction – Source Code Location – Application Krsul: 2-2-1-1	7 (1.7%)
(Client)(IntegerEvaluationRoutine)(BufferWrite)(Buffer) A client that supplies an integer larger than the integer variable type expected may cause an exception/buffer overflow or DoS Hoglund & McGraw: Content-Based Buffer Overflow and Arithmetic Errors in Memory Management Landwehr et al.: Genesis – Boundary Condition Violation Time of Introduction – Source Code Location – Application Krsul: 2-2-1-1	6 (1.4%)
(Client)(Server)(Message)(HeaderFieldBufferWrite)(Buffer) A client may submit an excessively long header field value causing a buffer overflow on the server (e.g., HTTP, email headers) Hoglund & McGraw: Content-Based Buffer Overflow Landwehr et al.: Genesis – Boundary Condition Violation Time of Introduction – Source Code Location – Application Krsul: 2-2-1-1	5 (1.2%)
(Client)(ReadClientInput)(EnvironmentVariableWrite)(Buffer) A client may submit an excessively long environment variable causing a buffer overflow in the application Hoglund & McGraw: Buffer Overflow with Environment Variables Landwehr et al.: Genesis – Boundary Condition Violation Time of Introduction – Source Code Location – Application Krsul: 2-2-1-1	5 (1.2%)
(Client)(UserNameEntry)(PasswordEntry)(Server)(AuthenticationRoutine)(Bu fferWrite)(Buffer) A client that submits an excessively long string of characters for either the username or password may cause a buffer overflow in the authentication routine Hoglund & McGraw: Buffer Overflow in Local Command-line Utilities Landwehr et al.: Genesis – Boundary Condition Violation Time of Introduction – Source Code	4 (1.0%)

(*continued on next page*)

Table 9 (*continued*)

Attack pattern and profile	*N* (%)
Location – Application Krsul: 2-2-1-1	
(Client)(QueryString)(Server)(Data)(Client)(BufferWrite)(Buffer) A client that requests data from an untrusted server may receive large data that results in a buffer overflow Hoglund & McGraw: Content-Based Buffer Overflow Landwehr et al.: Genesis – Boundary Condition Violation Time of Introduction – Source Code Location – Application Krsul: 2-2-1-1	4 (1.0%)
(Client)(HTTPRequest)(ProxyServer)(BufferWrite)(Buffer) A client that submits an excessively long HTTP GET request to a proxy server may cause a buffer overflow Hoglund & McGraw: Content-Based Buffer Overflow Landwehr et al.: Genesis – Boundary Condition Violation Time of Introduction – Source Code Location – Application Krsul: 2-2-1-1	3 (0.7%)
(Client)(CommandLineArgumentEntry)(ApplicationServer?)(Application)(Comm andLineArgumentBufferWrite)(Buffer) A client may submit an excessively long command line parameter causing a buffer overflow Hoglund & McGraw: Buffer Overflow in Local Command-line Utilities Landwehr et al.: Genesis – Boundary Condition Violation Time of Introduction – Source Code Location – Application Krsul: 2-5-1-1	2 (0.5%)
(Client)(Hyperlink)(Server)(HyperlinkBufferWrite)(Buffer) A client may make an excessively long hyperlink on a webpage and cause a buffer overflow on a server. If the hyperlink is used to connect to a session, then the malicious client may take over the application Hoglund & McGraw: Overflow Variables and Tags Landwehr et al.: Genesis – Boundary Condition Violation Time of Introduction – Source Code Location – Application Krsul: 2-2-1-1	2 (0.5%)
(Client)(HTTPServer)(HTTPMessageHandler)(Log)(SysAdmin)(LogEntryRead)(Bu fferWrite)(Buffer) A client that submits an excessively long message to the server can later induce a buffer overflow when viewed by a system administrator.12 Hoglund & McGraw: Content-Based13 Buffer Overflow Landwehr et al.: Genesis – Boundary Condition Violation Time of Introduction – Source Code Location – Application Krsul: 2-2-1-1	1 (0.2%)
(Client)(GUI/Browser)(BookMarkSave)(BookmarkBufferWrite)(Buffer) A client may save an excessively long bookmark and cause a buffer overflow. The bookmark may be written by the attacker or come from a long web page title. The attacker may be able to escalate their privileges Hoglund & McGraw: Content-Based Buffer Overflow Landwehr et al.: Genesis – Boundary Condition Violation Time of Introduction – Source Code Location – Application Krsul: 2-2-1-1	1 (0.2%)
(Read)(FileHeader)(BufferWrite)(Buffer) A client may label a file with an excessively long filename and cause a buffer overflow in the process reading the file. This occurred in an operating system context Hoglund & McGraw: Content-Based Buffer Overflow Landwehr et al.: Genesis – Boundary Co Time of Introduction – Source Code Location – Application Krsul: 2-2-1-1	1 (0.2%)

Table 9 (*continued*)

Attack pattern and profile	*N* (%)
(Client)(EmailHeader)(Firewall)(Buffer) A client can overflow a buffer in their firewall with a large email header to escalate their privileges Hoglund & McGraw: Content-Based Buffer Overflow Landwehr et al.: Genesis – Boundary Condition Violation Time of Introduction – Source Code Location – Application Krsul: 2-2-1-1	1 (0.2%)
(Metafile)(SizeField)(FileHeader)(FileRead)(BufferWrite)(Buffer) A client that specifies the ''Size'' field of a metafile to be less than the actual file may cause a buffer overflow Hoglund & McGraw: Content-Based Buffer Overflow Landwehr et al.: Genesis – Boundary Condition Violation Time of Introduction – Source Code Location – Application Krsul: 2-2-1-1	1 (0.2%)
Remote Execution Attacks (6 patterns, 13.4% of attack library) **(Client)(InjectionOfMaliciousHTMLTags/scriptInURL/Form)(Cookie*)(FormDat a*)(ServerVariables*)(Data)** A client may inject malicious scripts/tags (SCRIPT, OBJECT, APPLET, EMBED, FORM) or variables (e.g., JSP, ASP, search string) in a web page, msg. board, email, message (e.g., IM), Script in URL, URL parameter or HTML/CSS TAG, or HTML injection in HTML tag to obtain access to information such as cookies Hoglund & McGraw: Simple Script Injection Landwehr et al.: Genesis – Validation Error Time of Introduction – Source Code Location – Application Krsul: (no match)	48 (11.6%)
Malformed Data Attacks (10 patterns, 10.2% of the attack library) **(Client)(HTTPServer)(GetMethod)(Application ⊕ Data)** A malformed URL (e.g., excessive forward slashes, directory traversals, special chars such as '*', Unicode chars, format string specifier, NULL) may cause a DoS or in case of directory traversal the user may obtain private information Hoglund & McGaw: Postfix, Null Terminate and Backslash and Unicode Encoding Landwehr et al.: Genesis – Validation Error Time of Introduction – Source Code Location – Application Krsul: 2-2-1-3	27 (6.5%)
(Client)(Server)(MessageHeaderHandler)(Server) A client may send a negative, NULL, or invalid value (e.g., not include '':' between header name/value) in a header field resulting in a DoS on the server Hoglund and McGraw: (no match) Landwehr et al.: Genesis – Validation Error Time of Introduction – Source Code Location – Application Krsul: 2-2-1-3	5 (1.2%)
(Client)(Machine)(SyslogFunction)(Log)(Memory) It is possible to corrupt memory by passing format strings through the Syslog(), a logging function. This may potentially be attacked to overwrite arbitrary locations in memory with attacker-specified values. The Syslog function is often improperly used and is thus a target of attacks Hoglund & McGraw: String Format Overflow in syslog() Landwehr et al.: Genesis – Validation Error Time of Introduction – Source Code Location – Application Krsul: (no match)	1 (0.2%)
(Client)(Message)(Router) A client that submits malformed headers (e.g., failing to supply expected headers) may cause a DoS. Also, NULL as a header value may cause a DoS Hoglund and McGraw: (no match) Landwehr et al.: Genesis – Validation Error Time of Introduction – Source Code Location – Application Krsul: (no match)	1 (0.2%)

(*continued on next page*)

Table 9 (*continued*)

Attack pattern and profile	*N* (%)
Access Control Attacks (4 patterns, 7.0% of the attack library) **(Client)(SQLInput)(Server)(WebApplication)(Database)(Data)** Failure to sanitize client input (e.g., query string) can allow a client to submit an arbitrary SQL query, thus allowing for unauthorized access to data Hoglund and McGraw: (no match) Landwehr et al.: Genesis – Validation Error Time of Introduction – Source Code Location – Application Krsul: 2-2-1-3	19 (4.6%)
Stress-based Attacks (4 patterns, 1.3% of attack library) **(Client^{+})(Server^{+})(CPU^{+})(HardDrive*)** A script that make an excessive number of connections to the listening daemon process of a server may cause a DoS. This script need only make connections – further I/O may not be necessary with the connections Hoglund and McGraw: (no match) Landwehr et al.: Genesis – Boundary Condition Error Time of Introduction – Source Code Location – Hardware Krsul: (no match)	3 (0.7%)
(Client^{+})(Server^{+})(Log^{+}) (HardDrive^{+}) A client can exceedingly access a server that logs accesses to the hard drive. If permitted, the log file may become large enough to fill the hard drive causing the system to crash Hoglund and McGraw: (no match) Landwehr et al.: Genesis – Boundary Condition Error Time of Introduction – Source Code Location – Hardware Krsul: (no match)	1 (0.2%)
(Client)(Message)(Server)(Header^{+})(MessageHeaderHandle r)(Memory ⊕ CPU) A client may send a message with thousands of headers (e.g., MIME headers) to a server, causing a server memory/CPU DoS Hoglund and McGraw: (no match) Landwehr et al.: Genesis – Boundary Condition Error Time of Introduction – Source Code Location – Hardware Krsul: (no match)	1 (0.2%)
(Client^{+})(HTMLPage^{+})(Server^{+}) (HardDrive^{+}) A client may submit an excessive amount of data in an HTML page, thus filling up the hard drive on which the server resides Hoglund and McGraw: (no match) Landwehr et al.: Genesis – Boundary Condition Error Time of Introduction – Source Code Location – Hardware Krsul: (no match)	1 (0.2%)
Miscellaneous (5 patterns, 3.1% of the attack library) **(SourceFile)(IncludeFile)(EnvironmentVariable ⊕ ProgramVariable ⊕ QueryParam)(System)** An attacker can change/influence an environment, program, or URL variable to point to a remote machine. If the variable points to an ''include'' directory, then the attacker's include file can be executed on the target system Hoglund & McGraw: Make Use Of Configuration File Search Paths Landwehr et al.: Genesis – Other Exploitable Logic Error Time of Introduction – Source Code Location – Application Krsul: 2-3-2-3	8 (2.0%)
(Client)(HTTPServer)(PostMethod)(HTTPContent-LengthHeaderValue)(HTTPMessagePayloadLength)(ServerConnectionState) A client may submit a value via the POST method that specifies the Content-Length of the HTTP header be less than the content-length of the message, thus causing the socket to stay open (DoS) Hoglund and McGraw: (no match) Landwehr et al.: Genesis – Other Exploitable Logic Error	1 (0.2%)

Table 9 (*continued*)

Attack pattern and profile	*N* (%)
Time of Introduction – Source Code Location – Application Krsul: (no match)	
(Client)(SQLInputField)(Server)(WebApplication)(Database)(CPU) An attacker may submit a malicious SQL query (such as a Cartesian join of all tables) consuming the CPU Hoglund and McGraw: (no match) Landwehr et al.: Genesis – Other Exploitable Logic Error Time of Introduction – Source Code Location – Hardware Krsul: 2-2-1-3	1 (0.2%)
(Class)(Subclass)(OverriddenSecuredMethods)(Application) Overriding methods in a subclass that have been secured in a super class may create a software vulnerability Hoglund and McGraw: (no match) Landwehr et al.: Genesis – Other Exploitable Logic Error Time of Introduction – Source Code Location – Application Krsul: (no match)	1 (0.2%)

an attack path/attack pattern that did not correspond with a seeded vulnerability, then the answer was categorized as a false positive. A false positive while performing SAFE-T implies that the effort in discovering the attack path, and the effort applied in a risk assessment of the vulnerability is wasted.

The student answers are quantified in Table 6. Students were largely successful in identifying vulnerabilities in the system design. In the feasibility study, 937 (91%) of the student answers were true positives and 2067 (91%) were true positives for the validation study. Only 65 (7%) of the answers in the feasibility study were false positives and 155 (6%) were false positives in the validation study. If the number of vulnerabilities reported is large after SAFE-T is performed, then a significant number of these are false positives, which lead to wasted time. On average, students reported 3.8 attack paths per pattern in the feasibility study and 4.6 in the validation study. These results suggest that the attack patterns are generalized enough to match against multiple vulnerabilities. The multiplicity of answers suggests that not only can an attack pattern match to different environments (e.g., web, database, etc.), but can match to different scenarios within an environment.

Two attack patterns,

(Class)(Subclass)(OverriddenSecuredMethods)
(Application)

and

(Client)(*GUI/Browser*)(BookMarkSave)
(BookmarkBufferWrite)(Buffer),

appeared in the AL but were not seeded in the provided system design of the feasibility and validation studies. The presence of these two patterns tested whether the students could determine if vulnerabilities were not present in the system design. We did not expect any true positives, but students made unforeseen assumptions that justified the vulnerability represented by the attack pattern (see Table 7).

In both studies, the data indicated that for all attack patterns, the first attack pattern was associated with the highest frequency of students marking the vulnerability as not present in the design. This attack pattern was also associated with the largest number of false positives in both studies. These data suggest that performing SAFE-T on a design that does not have the vulnerabilities in common with the AL may yield a large number of non-existent threats that may hinder security efforts. Students read the attack profile and attempted to justify that a vulnerability represented by the second attack patterns could exist in the system design. The student's justifications and assumptions were unexpected, but after consideration we decided that the vulnerability could exist in the design. One answer that was accepted was 16-31-15-14-13 (see system design in Appendix C), and the student assumed that if the application was a Java application and secured methods were overridden, then the vulnerability could occur. Twenty-two students gave nine different false positives, the maximum in the study, for this attack pattern. These data suggest that personnel may fabricate scenarios that cannot exist if they cannot find a vulnerability from an attack pattern. More detail in the attack patterns and system design will likely provide indisputable evidence for vulnerability identification.

5. Summary and future work

Security vulnerabilities were analyzed in the SecurityFocus, Help Net Security, Secunia and SecurityTracker databases to study what vulnerabilities appear today and the techniques used to attack the vulnerabilities. An analysis of the descriptions in the databases reveals the events that transpire and what components are used to attack the vulnerability. The events were abstracted

and formalized by using regular expressions to encapsulate the steps that can be used to attack the software application. This research suggests that abstract attack patterns in the form of regular expression can identify security vulnerabilities in future applications. The method of identifying vulnerabilities is achieved via matching a sequence of components in a system design to the symbols in a regular expression that permits the sequence of events in the attack pattern to occur. If a match exists, then the vulnerability may exist in the application being analyzed. Performing the matching in the design phases increases security awareness at the beginning of the software process and encourages risk management to begin early so a security team can determine how to fortify their application.

Continuing the research for vulnerability information beyond the scope of vulnerability databases and into open source references would make for more accurate vulnerability descriptions. Also, the approach should be validated further with professionals, computer science graduate students, and business students. The results will show if the approach is effective for those individuals with security expertise, computer science backgrounds, and if the approach can be performed without a background in computer science. Also, a study comparing what vulnerabilities and how many can be identified by SAFE-T, attack trees and attack nets would be useful in determining which approach can best increase security awareness for future systems. An analysis should also be included to determine when each approach is applicable and give a measure of its effectiveness.

Acknowledgements

We thank the North Carolina State University (NCSU) Software Engineering reading group for their helpful suggestions on this paper. In particular, we thank Mattias Stallmann for his insight into regular expressions. This material is based upon work partially supported by the National Science Foundation under CAREER Grant No. 0346903. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Appendix A. Alphabet used in attack library

See Table 8.

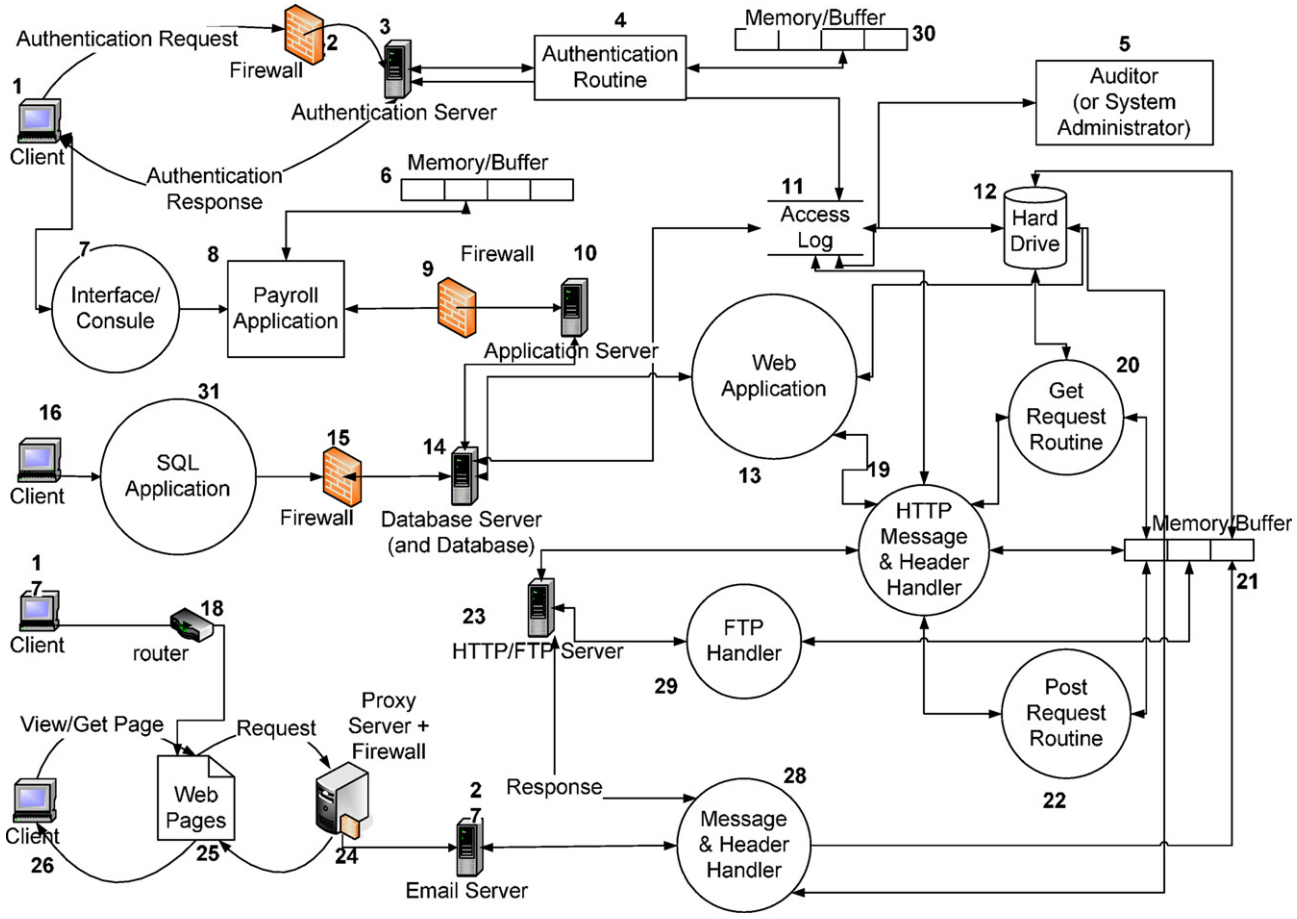

Fig. 2.

Appendix B. Attack library

The following is a subset of our attack library and is organized into seven general categories: Buffer Overflow Attacks, Remote Execution Attacks, Malformed Data Attacks, Access Control Attacks, Error Message Attacks, Stressed-based Attacks, and Miscellaneous. The first column shows the attack pattern, attack profile and a mapping (where possible) to the three taxonomies discussed in this paper. The heading for each category shows the number of attack patterns for each category and the percentage of the 53 attack patterns. The second column shows the number of instances of each attack pattern that occurred in the 244 vulnerabilities studied.

See Table 9.

Appendix C. System design

See Fig. 2.

References

[1] W.A. Arbaugh, W.L. Fithen, J. McHugh, Windows of vulnerability: a case study analysis, IEEE 3 (12) (2000) 52–59.
[2] M. Bishop, M. Dilger, Checking for race conditions in file accesses, Computing Systems 9 (2) (1996) 131–152.
[3] B. Boehm, V. Basili, Software defect reduction top 10 list, Computer 34 (1) (2001) 135–137.
[4] J. Carlstedt, R. Bisbey II, G. Popek, Pattern-directed Protection Evaluation, USC Information Sciences Institute, Marina del Rey, 1975.
[5] H. Chen, J. Shapiro, Using build-integrated static checking to preserve correctness invariants, ACM (2004).
[6] H. Chen, D. Wagner, Mops: an infrastructure for examining security properties of software, ACM CCS (2002).
[7] W. Cunningham, "Wikiwikiweb," 1994.
[8] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns, Addison-Wesley, Boston, 1995.
[9] M. Gegick, Analyzing security attacks to generate patterns from vulnerable architectural patterns, Computer Science, vol. MS, NCSU, Raleigh, 2004.
[10] M. Gegick, L. Williams, Matching attack patterns to security vulnerabilities in software-intensive system designs, International Conference on Software Engineering—Software Engineering for Secure Systems, ACM Press, 2005.
[11] S. Hansman, R. Hunt, A taxonomy of network and computer attacks, Computers and Security 24 (1) (2005) 31–43.
[12] G. Hoglund, G. McGraw, Exploiting Software, Addison-Wesley, Boston, 2004.
[13] M. Howard, D. LeBlanc, Writing Secure Code, Microsoft Corporation, Redmond, 2003.
[14] IEEE, Ansi/ieee standard glossary of software engineering terminology, 1990.
[15] I. Krsul, "Software vulnerability analysis," Computer Science, vol. Ph.D., Purdue University, West Lafayette, 1998.
[16] S. Kumar, E. Spafford, A pattern matching model for misuse intrusion detection, Proceedings of the 17th National Computer Security Conference, Purdue University, 1994.
[17] C. Landwehr, A. Bull, J. McDermott, W. Choi, A taxonomy of computer program security flaws, with examples, ACM Computing Surveys 26 (3) (1994).
[18] J.P. McDermott, Attack net penetration testing, ACM SIGSAC (2000) 15–21.
[19] G. McGraw, Software Security: Building Security In, Addison-Wesley, Boston, 2006.
[20] A.P. Moore, R.J. Ellison, R.C. Linger, Attack Modeling for Information Security and Survivability, Carnegie Mellon University, 2001.
[21] B. Schneier, Attack trees: Modeling security threats, Dr. Dobb's Journal (1999).
[22] M. Schumacher, U. Roedig, Security engineering with patterns, Proceedings of the 8th Conference on Pattern Languages of Programs, 2001.
[23] J. Steffan, M. Schumacher, Collaborative attack modeling, Proceedings of the 2002 ACM Symposium on Applied Computing (SAC'02, Madrid, Spain) 2002, pp. 253–259.
[24] J. Viega, G. McGraw, Building Secure Software How To Avoid Security Problems the Right Way, Addison-Wesley, Boston, 2002.