Matching Attack Patterns to Security Vulnerabilities in Software-Intensive System Designs

Michael Gegick, Laurie Williams
North Carolina State University, Department of Computer Science
{mcgegick, lawilli3}@ncsu.edu

ABSTRACT
Fortifying software applications from attack is often an effort that occurs late in the software development process. Applying patches to fix vulnerable applications in the field is a common approach to securing applications. Abstract representations of attacks such as attack trees and attack nets can be used for identifying potential threats before a system is released. We have constructed attack patterns that can illuminate security vulnerabilities in a software-intensive system design. Matching our attack patterns to vulnerabilities in the design phase may stimulate security efforts to start early and to become integrated with the software process. The intent is that our attack patterns can be used to effectively encode software vulnerabilities in vulnerability databases. A case study of our approach with undergraduate students in a security course indicated that our attack patterns can provide general descriptions of vulnerabilities. The students were able to accurately map the patterns to vulnerabilities in a system design.

General Terms
Security, Design

Keywords
Security, Design, Regular expression

1. INTRODUCTION
The cost associated with addressing software problems increases as the lifecycle of a project matures. The cost of finding and fixing a bug after a software product has been released can be 100 times more expensive than solving the problem in the requirements or design phase [3]. Thus, patching vulnerable software after release can be a costly way of securing applications. Furthermore, patches are not always applied by owners/users of the vulnerable software; patches can contain yet more vulnerabilities [20]. Another security technique is to rely on devices external to a software system such as intrusion detection systems and firewalls. These protection mechanisms can provide some security, but are only available when software development is complete. The rising concerns of software security suggest that security focuses should be applied throughout the software process to afford more cost effective solutions and provide opportunities for stronger, layered defenses, a best practice technique [10].

ICSE-SESS'05, May 15-16, 2005, St. Louis, Missouri, USA.

Many intrusions are based on a small number of known attacks, implying that the vulnerabilities are also the same or similar [13]. Furthermore, vulnerabilities can be attacked for years after their discovery [2]. Software engineers can make use of this pattern behavior of vulnerabilities. According to Christopher Alexander [1], a pattern describes a situation where a problem recurs and provides a solution that can be applied regardless of the environment in which the problem occurs. Alexander makes this claim in the context of architecting buildings, but his idea has also been adopted in the software community.

Gamma et al. [7] make use of patterns for object-oriented design and show how they can be effectively reused in many types of programs. Also, Schumacher and Roedig [18] make use of patterns in the context of security. They define a security pattern as a particular recurring security problem that arises in specific contexts and presents a well-proven generic scheme for its solution [18]. We use *attack patterns* to describe the components and events of a software system that are involved with the manipulation of common vulnerabilities. These attack patterns can be reused to identify vulnerabilities in software systems. While most patterns provide both a generic problem and a solution, our current research does not provide solutions to the attack patterns.

We propose a methodology for early identification of system vulnerabilities that we call **S**ecurity **A**nalysis for **E**xisting **T**hreats (SAFE-T) [8]. With SAFE-T, software engineers match attack patterns to system designs to identify potential vulnerabilities. SAFE-T is a means for security to start in the design phase of the software process and provides a taxonomy of attack patterns that can describe common security problems. Security awareness plagues software engineers today [10, 18, 20] and so we attempt to illuminate vulnerabilities with a software engineering approach.

In this research, we examined 244 previously-reported vulnerabilities from four web-based security vulnerability databases (SecurityFocus, Help Net Security, Secunia,

and SecurityTracker) to serve as a source of information for our study. The vulnerability descriptions included the components[1] in the software system that are susceptible to attack and the events that would occur in such an attack. The descriptions were abstracted into regular expressions to provide a general representation of an attack. 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 ability of the patterns to describe vulnerabilities was assessed in an experiment with in an undergraduate security course performing SAFE-T. Students were asked to match our attack patterns to a given system design. The students, novices to our approach, accurately detected vulnerabilities in two system design experiments. Our studies were specifically targeted at undergraduate students because a research goal involved the development of a security identification technique that could be used effectively by relatively inexperienced, non-security experts.

In Section 2, we provide related work in abstracting vulnerabilities and vulnerability taxonomies. In Section 3 we discuss our contribution of attack patterns. In Section 4 the validation technique and results are presented and in section 5 we discuss the limitations of our approach. In Section 6 we conclude and summarize our findings.

2. RELATED WORK

In this section we discuss how vulnerabilities can be abstracted into a general representation. We then discuss taxonomies of vulnerabilities.

2.1 Abstracting vulnerabilities

Abstracting vulnerability information from flaws in software systems into patterns is not a new idea. In 1975, Carlstedt et al. [4] proposed abstracting specific system calls, data stores, and other entities in operating system source code into generalized patterns. These text-based patterns could then be applied to other operating systems that have different names of procedures, data stores, etc. for the same functionality. The patterns were found to be understandable by those with little experience in finding security flaws, suggesting that expert knowledge of security threats can be relayed to non-experts.

The process of finding security vulnerabilities in source code has been automated with static analyzers. Security vulnerabilities can be represented as finite state automata (FSA) to show a sequence of events that may result in an attack [6]. If a program is found to execute a sequence of commands that match the events in an FSA, then developers are warned that a vulnerability may exist in their code. Performing a 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 coding phase can be more costly than if the vulnerabilities were known in advance.

Graphical structures of abstract vulnerability representations have also been proposed. Attack trees [17] are tree diagrams (that can also be converted to text) whose nodes show the goals of an attacker. Attack trees show the point of view of an attacker completing steps toward his or her final objective. Software engineers can use attack trees to determine if such a sequence of objectives is possible in their software systems. Attack nets [15] are another graphical abstract representation of exposing vulnerabilities in software systems. Unlike the goal-oriented attack trees, attack nets show specific components of a system to identify precisely where in the system penetration testing should occur. Attack nets show more detail of a software system than attack trees because they are not strictly from the point of view of an attacker who may have little knowledge of the system. The graphical structures of the aforementioned abstract representations of attacks do not precisely mimic a system design. SAFE-T attempts to structure attack patterns that more closely correspond to system designs to facilitate vulnerability identification in the design phase.

2.2 Vulnerability taxonomies

A taxonomy of recurring vulnerabilities may be helpful for organizing the information needed to increase security awareness today. An advanced knowledge of vulnerabilities may be helpful for identifying potential attacks on software before it is released to customers. We examine three taxonomies:

- McGraw and Hoglund [9] 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. [14] provide a taxonomy that attempts to answer three questions about vulnerabilities: How did it enter the system? When did it enter the system? Where in the system is the vulnerability manifest?
- Krsul [12] provides four hierarchical classes in his taxonomy: Design Flaws, Environmental Flaws, Coding Flaws, and Configuration Flaws. We compared our taxonomy to the Environmental Flaws class, which is geared toward the identification of the assumptions that programmers make about their environment, and whose violation results in software vulnerabilities.

Describing and/or representing vulnerabilities in a straightforward manner is crucial for the effective use by others in the detection of vulnerabilities in their own system. We contribute a taxonomy tailored for detecting vulnerabilities in the design phase of the software process using attack patterns. We compare the three taxonomies to ours in Section 4.2.

3. RESEARCH APPROACH

Applying attack patterns to software designs early in the software process via SAFE-T may be useful for identifying security vulnerabilities while corrective action is still

[1] 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 [11].

relatively inexpensive. The attack patterns presented in this paper are expressed via regular expressions that begin with a "start" event, symbolized by the component that can be used to initiate the attack. Each major successive event in the attack is expressed by its associated component and is appended to the string that begins with the start component. Finally, the *threat target* [10] (that represents the final component of the attack) terminates the string of components to complete the illustration of the attack path. For example, the pattern,

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

describes a series of User (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 is 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. [16] to accompany the pattern to describe the attack pattern. In this example, the attack profile is as follows: *A user can exceedingly request from 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.* Our ultimate goal is to automate the matching of attack patterns to system designs thus we use regular expressions.

In Figure 1, a simplistic example of a system design is shown that reflects the log file vulnerability. The components in the system designs are numbered to provide a means of representing an attack path. There are two attack paths in Figure 1 that correspond to $(\texttt{User}^{+})(\texttt{Server}^{+})(\texttt{LogFile}^{+})(\texttt{HardDrive}^{+})$ attack pattern: 1-2-3-5 and 1-2-4-5. Upon the match, a security team now has a graphical representation of the attack path and the potential vulnerability.

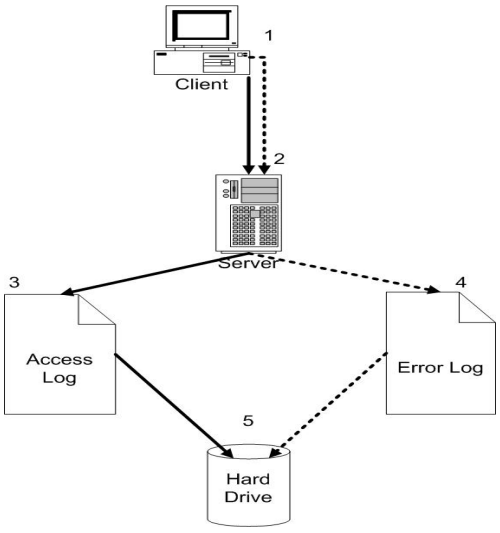

Figure 1. Sample system design.

Four regular expression operators are used in the attack patterns to further clarify the characteristics of an event (see Table 1).

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.
$\oplus$ (Exclusive OR)	The event to the left or right of the operator will occur, but not both.
$^{+}$ (superscript)	The event occurs at least once.
?	The event occurs zero times or once.

As just previously shown, the superscripted + can be used to show a series of one or more events. The Kleene closure can indicate where an event may or may not be required. The attack pattern,

$$(\texttt{User}^{+})(\texttt{Server}^{+})(\texttt{CPU}^{+})(\texttt{HardDrive*}),$$

can be read as a malicious `User` may submit at least one request, followed by the Server accepting at least one request, followed by the `CPU` running at least one process/thread, followed by the threads making zero or more disk writes. It is possible that each thread can write large amounts of information to the `HardDrive` causing a denial-of-service. The write operation may or may not occur and is thus characterized with the Kleene closure.

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

$$(\texttt{User})(\texttt{CommandLineArgumentEntry})(\texttt{ApplicationServer?})\ (\texttt{Application})\ (\texttt{CommandLineArgumentBufferWrite})(\texttt{Buffer}),$$

a `User` works on an application, and enters an excessively long `CommandLineArgument`, which is read by an `Application`, which may or may not be on `ApplicationServer`, followed by writing the `CommandLineArgumentBufferWrite`, followed by the data overflowing the `Buffer`. The ? allows the regular expression to show 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. Events such as these can be inserted into the regular expression to clarify operations that occur before or after the event that occurs at the component.

Lastly, the $\oplus$ (xor) 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,

$$(\texttt{User})\ (\texttt{Variable} \oplus \texttt{Filename} \oplus \texttt{Header})(\texttt{HTTPServer})(\texttt{PostMethod})\ (\texttt{BufferWrite})(\texttt{Buffer}),$$

a User 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 a `BufferWrite` of the `Variable` or `Filename` or `Header`, 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. This attack pattern demonstrates the ability of one attack pattern to generalize several different inputs that represent similar functionality.

In general, attack patterns are also intended to be program language-independent so that coding vulnerabilities can be found regardless of the implementation. However, patterns such as

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

are specific to low-level software designs implemented in object-oriented programming languages where classes and methods are specified. The vulnerability that is captured by this expression is that a `Class` is extended to form a `Subclass`. The `Subclass` overrides secured methods of the parent `Class` to form its own `OverriddenSecuredMethods`. If the `OverriddenSecuredMethods` are not secured, then a vulnerability may exist in `Application`. An attacker may then attack the overridden secured methods in the application. This representation is limited to programming languages that allow a parent class to be extended, such as Java or C++. Other than the buffer overflow attack patterns we constructed, this pattern was the only code-level vulnerability found in the vulnerabilities analyzed for this study. The focus of this research is at higher, system-level vulnerabilities that can be examined before coding starts.

Finally, a pattern can indicate the multiple opportunities to prevent an attack to encourage layered defenses, a best practice technique [10]. The pattern,

$$(\texttt{User}^{+})(\texttt{HTTPServer}^{+})(\texttt{GetRequestRoutine}^{+})$$
$$(\texttt{Buffer} \oplus \texttt{CPU})$$

describes an attack where a `User` submits at least one large GET request, followed by the `HTTPServer` accepting the request at least once, followed by the `GetRequestRoutine` processing at least one request and writing it to a `Buffer` causing a buffer overflow. Alternately, if there are many of these large requests, then the `CPU` must process each one, which could consume the `CPU` cycles on the machine in which the server resides and cause a denial-of-service. The denial-of-service caused by the consumption of `CPU` cycles can be avoided if an implementation is provided to halt the `HTTPServer` from accepting a flood of requests. If this implementation fails, then a secondary defense could be a method that prevents the `HTTPServer` from accepting an unreasonably large GET request. In this way, the two defenses that secure the `CPU` from wasting cycles in a denial-of-service attack are restricting the number of requests and managing the size of the requests.

4.0 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 pattern, and the results of empirical studies examining how well the patterns can be matched to a system design.

4.1 Attack pattern validation

We analyzed 414 entries from the following vulnerabilities databases: SecurityFocus, owned by Symantec; Help Net Security, a privately owned company; Secunia, an IT security company; and SecurityTracker, owned by SecurityGlobal.net LLC). One hundred and seventy (41.1%) of these vulnerabilities were not used in our analysis to make a current set of attack patterns. Table 2 sums the type of vulnerabilities excluded from the study. Eighty-five (20.5%) of the vulnerabilities in SecurityFocus lacked descriptions with sufficient detail to form patterns.

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%)

There were 48 (11.6%) vulnerabilities that were specific to vendors and would not likely serve helpful in the general protection of typical software applications. For example, these included (1) Microsoft XP-specific problems, (2) Norton Antivirus crashing when scanning files in certain folders, and the ability for (3) Internet Explorer to capture user keystrokes. Retaining vulnerabilities that are specific to vendors would increase the number of attack patterns to consider and thus decrease the efficiency of matching the enumerated patterns to components in the system design in the general case.

Twenty-one (5.1%) of the observed vulnerabilities were inapplicable, meaning they 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 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 logic problems. Therefore, low-level 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 logic 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 and stored in tabular form which we call an attack library (AL). Sample entries of a AL are given in given in the Appendix; the full AL of the 53 patterns can be found in [8]. The right column shows the percentage of the 414 vulnerabilities that the attack pattern represents. The maximum number of vulnerabilities that a pattern mapped to was 83 and the minimum was one. On average, six vulnerabilities mapped to one pattern in the AL. This is evidence that the patterns can encapsulate several attacks to show multiple vulnerabilities in a system. The first column of the Appendix contains a regular expression followed by an attack profile. We then indicate where the attack pattern has also been published in the three studied taxonomies. The second column indicates the number of times that attack occurred in our sample from the vulnerability databases.

4.2 Taxonomy comparison

The attack patterns are now compared to the taxonomies of Hoglund and McGraw [9], Landwehr [14], and Krsul [12] (see Table 3). Each of the taxonomies presents a different way to represent a vulnerability classification. However, we show that some of the classifications among the different taxonomies refer to the same vulnerability. Sample mappings between our attack pattern-based taxonomy and these taxonomies can be found in the Appendix.

Table 3: Resemblance of Taxonomies

Source	Number of SAFE-T Patterns Matching to Other Taxonomy Classifications (%)	Total Number of Attack Classifications
Hoglund, McGraw	33 (62.3%)	49
Landwehr et al.	53 (100%)	29
Krsul	35 (66.0%)	54

Four of the 21 buffer overflow vulnerabilities mapped directly to specific buffer overflow vulnerabilities published by McGraw and Hoglund [9]. 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.

The taxonomy provided by Landwehr et al. [14] 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. Twenty-one (21.8%) of our attack patterns are buffer overflow attacks and match to the Boundary Conditions category, a subcategory of Genesis. The Time of Introduction classification attempts to classify which of the five software phases of the software process the vulnerability was introduced. According to Landwher's taxonomy, all of our attack patterns are introduced in the Source Code phase of the software process. 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. Our comparison shows that 47 (89%) of our attack patterns are involved with the Application category. Lahdwehr's taxonomy is at a higher level of abstraction relative to our attack patterns, which allows for 100% of our patterns to be mapped their taxonomy. Our attack patterns are more specific to what components can pinpoint a vulnerability in a system design.

Lastly, we were able to map 35 (66%) of the attack patterns presented in this research to seven (13%) of the classifications proposed by Krsul. Krsul's taxonomy shows that the 35 attack patterns are derived from malicious user input. The largest mapping falls into the 2-2-1-1 category, which describes that developers make assumptions about user input is of a length of x. Twenty of our attack patterns are involved with buffer overflows, which is a very common tactic for attackers.

Our taxonomy of vulnerabilities offers a means to identify vulnerabilities in system designs. We have shown that approximately one third of our taxonomy offers new classifications of vulnerabilities not mentioned in Hoglund and McGraw's taxonomy 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 is primarily Internet-based 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. that may provide more information for software engineers to detect a vulnerability. Compared to the three taxonomies presented in this paper, our attack patterns are more similar to the structure of system designs, suggesting that they may be more useful for identifying vulnerabilities in the design phase of a software process. As mentioned, detecting vulnerabilities early in the software process may allow for strengthened defenses and require less effort from a development team.

4.3 SAFE-T validation

We ran two studies to determine how seamless the matching process is between an attack pattern and a vulnerability in a software-intensive system design. In both studies, students were given an AL of attack patterns/profiles and had to match the patterns to vulnerabilities in a system design that was seeded with vulnerabilities. The artifacts used in these studies are documented in [8].

The first study, a feasibility study, was run with 43 students in an upper-level undergraduate security class at North Carolina State University. The exercise consisted of 20 attack patterns seeded into a system design consisting of 16 components. The results of the 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

patterns and an advanced design with 14 more components. The results of both studies are now discussed.

Student answers were categorized as:

- False negatives if the student missed a vulnerability, implying that the vulnerability will go undetected in the software process.
- True positives if a student chose components in the design that supported the actions necessary to achieve the attack.
- False positive if a student submitted an instance of a pattern that did not have a vulnerability. 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 4. The student answers in the feasibility study showed approximately a 2:1 ratio of true to false positives. The results of the validation study represent an approximately 3:1 ratio of true to false positives and thus support the findings in the feasibility study. 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 students can apply a generalized attack pattern to different scenarios in a system design to identify multiple vulnerabilities. However, the students' answers suggest that one-third of their findings are inaccurate, which can lead to wasted time during risk analyses.

Table 4: 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)

5.0 LIMITATIONS

SAFE-T is predicated on the assumption that the software process used calls for a system design. A high-level system design is the minimum requirement for the patterns to be used for a security analysis. Also, many of the attack patterns may match to a system design thus creating an overwhelming number of warnings to developers. Requiring developers to fortify against each possible attack can disrupt the balance of implementing security and functionality resulting in a slower development process. A risk analysis can be employed to manage the number of vulnerabilities to include only those with the greatest risk.

6. SUMMARY AND FUTURE WORK

Security vulnerabilities were analyzed in four vulnerability databases to determine which vulnerabilities appear today and the techniques used to exploit the vulnerabilities. An analysis of the descriptions in the databases reveals the events that transpire and what components are used to exploit 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 can identify security vulnerabilities in future applications. The method of identifying vulnerabilities is achieved via matching a sequence of components in a system design 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.

Our validation is based on students that are relatively inexperienced in software security. 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. Finally, 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.

7. ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under CAREER award 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.

7. REFERENCES

[1] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksadahl-King, and S. Angel, *A Pattern Language*. New York: Oxford University Press, 1977.

[2] W. A. Arbaugh, W. L. Fithen, and J. McHugh, "Windows of Vulnerability: A Case Study Analysis," *IEEE*, vol. 3, 12 pp. 52-59, 2000.

[3] B. Boehm, "Industrial Metrics Top 10 List," *IEEE Software*, vol. 4, 5 pp. 84-85, 1987.

[4] J. Carlstedt, R. Bisbey II, and G. Popek, "Pattern-Directed Protection Evaluation," USC Information Sciences Institute, Marina del Rey ISI/RR-75-31, June 1975.

[5] H. Chen and J. Shapiro, "Using Build-Integrated Static Checking to Preserve Correctness Invariants," *ACM*, 2004.

[6] H. Chen and D. Wagner, "MOPS: an Infrastructure for Examining Security Properties of Software," *ACM CCS*, 2002.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*. Boston: Adison-Wesley, 1995.

[8] M. Gegick, "Analyzing Security Attacks to Generate Patterns from Vulnerable

Architectural Patterns",MS, NCSU, Raleigh. July, 2004
[9] G. Hoglund and G. McGraw, *Exploiting Software*. Boston: Addison-Wesley, 2004.
[10] M. Howard and D. LeBlanc, *Writing Secure Code*, 2nd ed. Redmond: Microsoft Corporation, 2003.
[11] IEEE, "ANSI/IEEE Standard Glossary of Software Engineering Terminology," 1990.
[12] I. Krsul, "Software Vulnerability Analysis",PhD, Purdue University, West Lafayette.
[13] S. Kumar and E. Spafford, "A Pattern Matching Model for Misuse Intrusion Detection," Proceedings of the 17th National Computer Security Conference, West Lafayette, 1994.
[14] C. Landwehr, A. Bull, J. McDermott, and W. Choi, "A Taxonomy of Computer Program Security Flaws, with Examples," *ACM Computing Surveys*, vol. 26, 3, 1994.
[15] J. P. McDermott, "Attack Net Penetration Testing," *ACM SIGSAC*, pp. 15-21, 2000.
[16] A. P. Moore, R. J. Ellison, and R. C. Linger, "Attack Modeling for Information Security and Survivability," Carnegie Mellon University March 2001.
[17] B. Schneier, "Attack Trees: Modeling Security Threats," *Dr. Dobb's Journal*, 1999.
[18] M. Schumacher and U. Roedig, "Security Engineering with Patterns," *8th Conference on Pattern Languages of Programs*, 2001.
[19] J. Steffan and M. Schumacher, "Collaborative Attack Modeling," *Proceedings of the 2002 ACM Symposium on Applied Computing (SAC'02, Madrid, Spain)*, pp. 253-259, 2002.
[20] J. Viega and G. McGraw, *Building Secure Software How to Avoid Security Problems the Right Way*. Boston: Addison-Wesley, 2002.

APPENDIX

Attack Pattern and Profile	N (%)
(SocketRead)(SocketBufferWrite)(Buffer) A user may submit an excessively long stream to a socket and cause a buffer overflow. This is true for handling any connection on the internet (e.g. GET request). 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	83 (20.0%)
(User)(InjectionOfMaliciousHTMLTags/scriptInURL/Form)(Cookie*)(FormData*)(ServerVariables*)(Information) A user 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%)
(User)(HTTPServer)(GetRequestRoutine)(Application ⊕ Information) 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%)