Abstract

GEGICK, MICHAEL CHARLES. Analyzing Security Attacks to Generate Signatures from Vulnerable Architectural Patterns. (Under the direction of Dr. Laurie Williams.)

Current techniques for software security vulnerability identification include the use of abstract, graph-based models to represent information about an attack. These models can be in the form of attack trees or attack nets and can be accompanied with a supporting text-based profile. Matching the abstract models to specific system architectures for effective vulnerability identification can be a challenging process. This thesis suggests that abstract regular expressions can be used to represent events of known attacks for the identification of security vulnerabilities in future applications. The process of matching the events in the regular expression to a sequence of components in a system design may facilitate the means of identifying vulnerabilities. Performing the approach in the design phase of a software process encourages security to be integrated early into a software application. Students in an undergraduate security course demonstrated a strong ability to accurately match regular expressions to a system design. The identification of vulnerabilities is limited to known attacks of other systems and does not offer descriptions of what new attacks are possible to a future application. Extending the approach to incorporate new attacks is an avenue of future work.

Analyzing Security Attacks to Generate Signatures from Vulnerable Architectural Patterns

by

Michael Charles Gegick

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science
Raleigh
2004

APPROVED BY:

Committee Member

Committee Member

Chair of Advisory Committee

Biography

Michael has a BA in Biology from Washington and Jefferson College (1998) and a BS in Computer Science from North Carolina State University (2001). He enjoys swimming, biking, running, and rock climbing.

Acknowledgements

Thanks to the members of my thesis committee, Dr. Laurie Williams (chair), Dr. Annie Antón and Dr. Julia Earp for their support and suggestions during my research. Dr. Williams helped to steer me through the entire thesis and was always optimistic and amazingly patient. I would like to sincerely thank Dr. Gary McGraw (of Cigital), Dr. Michael Reiter (of Carnegie Mellon University), and Dr. Eugene Spafford, Dr. Pascal Meunier, and Rajeev Gopalakrish (each of Purdue University) for taking the time to understand my thesis approach and give opinions on its validity and practicality in the field of security engineering. I would also like to thank Nick Green and Eric Isakson for their suggestions on the approach when I first arrived at the concept. Additionally, Bruce Wieand was helpful in determining if regular expressions were a possible means of representing events that may occur in an attack and deserves much appreciation. Without Julie Starr, the teacher for CSC405 at North Carolina State University, the feasibility and validation studies would not have been possible. She was very cooperative and understanding with the incorporation of my assignment in her class. Many thanks to Penney Martin for helping to elicit the idea proposed in this thesis.

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

Table of Contents

LIST OF TABLES

LIST OF FIGURES

1.0 Introduction

Adding security late in the software development process, sometimes only after an actual penetration by an attacker, is a common practice in the industry setting. However, adding security to even a partially-completed software application is an insufficient means for securing against threats [20]. If many changes to a partially completed system are needed to fortify the code, then developers may need to make significant modifications to other parts of the system to accommodate the changes. If this is the case, then rewriting the code from scratch with security in mind may be less expensive in terms of time and effort and potentially be less error prone. A means of integrating security at the beginning of the software process is needed to overcome the consequences of starting security when it is too late. Incorporating a feasible method of security can help prevent weak security that leads to drastic consequences, such as endangering the integrity and prosperity of ecommerce and compromising the privacy of electronically-stored health information.

Organizations cannot completely rely on firewalls and cryptography that do not prevent all security attacks. Cryptography has provided many safe opportunities for the computing environment, but it is not a panacea. Fred Schneider's [16] analysis of CERT advisories shows that the majority of security vulnerabilities occur from "buggy code" and that applying cryptography would have prevented less than 15% of the vulnerabilities. Spafford contends that "using encryption on the Internet is the equivalent of arranging an armored car to deliver credit card information from someone living in a cardboard box to someone living on a park bench." [9] Thus, software engineering techniques should provide practical techniques for building secure software.

Viega and McGraw [20] assert that software engineers should begin early, know the security threats, design for security, and subject system design to thorough objective risk analyses and testing. The abundance of vulnerabilities that exist today can present a challenge for security experts to expose threats in a system. The efficacy of the security process depends on how early a vulnerability is found in the software process and removed. The longer any defect remains in a product the more expensive it is to fix it in terms of time, resources and money [4] , including security vulnerabilities. It is therefore essential to have a dependable means of identifying vulnerabilities in a software system early enough so that security can be built into the application from the start.

Most security attacks exploit known vulnerabilities in software systems, and thus old vulnerabilities cannot be ignored when building a new application [2]. This suggests that any system that has a vendor's application with a known vulnerability is highly susceptible to attack and implies that any product with the same functionalities as the vulnerable application is at risk by the same attack. Also, identical intrusions have been known to be repeated years later, suggesting that it is important to secure for even the oldest attacks that may appear forgotten [2]. Script kiddies, those individuals who execute scripts by written crackers, are the most likely candidates to execute these types of repeat attacks. Hence, basic security begins with a thorough knowledge of known attacks and with the examination of whether these same vulnerabilities can threaten a system under development.

My research objective is to create and validate abstract representations of known attack paths found in vulnerability databases to facilitate the identification of threats in a software system. The form of the proposed abstract representations is regular expressions that represent the components and the sequence of events that occurred in known attacks. Regular expressions can be used to show how data may flow from one component to another and represent where system components have accessed a resource in an illegal way. A matching of the events in the regular expression to a sequence of components in a system design can help identify potential attack paths for stakeholders of an application. A knowledge base of the regular expressions will serve as a means of identifying many known vulnerabilities to decrease the possibility of overlooking and, thus, perpetuating known vulnerabilities in a system design. Once the vulnerabilities are identified, a risk management process is needed to determine which vulnerabilities threaten the system the most. Software engineers can thus know which threats to secure in their code as they begin to build the system. In this way, security is built into the system at the start of the software process.

Four vulnerability databases (SecurityFocus[1], Help Net Security[2], Secunia[3], and SecurityTracker[4]) were studied for the purpose of collecting and analyzing the descriptions of previously-discovered vulnerabilities. The descriptions are used to determine the events that occurred in the attack and the components that were responsible for triggering the events. A total of 409 vulnerabilities were analyzed, and 53 regular expressions were produced. An initial twenty of these regular expressions were used in a feasibility study to test if advanced undergraduate students could map the expressions to the components of a system design. The results of the feasibility study motivated further study. Three hundred and fifty two vulnerabilities were additionally studied and 33 more regular expressions were developed. After the completion of this work, a validation study was conducted with advanced undergraduate students and 30 of the regular expressions. The results are discussed in this thesis and suggest that encapsulating known attacks with regular expressions for the identification of vulnerabilities in system designs may be a viable approach to finding security vulnerabilities in the design phase.

In this thesis, we examine the efficacy of using regular expressions to identify security vulnerabilities in software designs. The remainder of this thesis is organized as follows. Chapter 2 provides a background of related work on vulnerability identification. Chapter 3

[1] http://www.securityfocus.com
[2] http://net-security.org/
[3] http://secunia.com/
[4] http://www.securitytracker.com/

provides the methodology used to create and apply regular expressions. Chapter 4 presents the means in which vulnerabilities were collected. Chapter 5 shows the results of vulnerability collection. Chapter 6 presents the methodology and results of a feasibility study. Chapter 7 discusses the methodology and results of the validation study. Lastly, Chapter 8 presents a summary and gives ideas for future work.

2.0 Background

This chapter provides information on how vulnerabilities can be represented, how security can be a collaborative effort, where security can be integrated with the software process, and how risk management can be used to manage the assessment and implementation of security.

2.1 Vulnerability Representation

Once a vulnerability is found, it should be documented in a way that others can understand. If documented well, a person who did not find the vulnerability should be able to read the description and find the same vulnerability, if it exists, in their own system. Documenting vulnerabilities to find the same vulnerability in the same context is useful, but does not provide a scaleable approach for the many vulnerabilities among the different systems of today. If a vulnerability or attack can be represented abstractly to show a vulnerability in a different type of system or even a slightly different form, then the documentation becomes applicable to providing security in the general sense.

In 1975**,** Carlstedt et al. were perhaps the first researchers to generalize what they called "protection errors." [6] In their discussion, they made abstract representations of objects in the system. For example, any object such as memory, files, or variables was classified as an *abstract cell* that holds information. They excluded as much information about an error as possible while still instantiating it for specific objects in the system they assessed. Their abstractions are termed *error-patterns* and are represented by an enumerated list of the events that transpired among objects in the system. A *raw pattern* is an error pattern that describes a detailed error in a specific operating system as shown in the following example.

1. Load is called by Snap Dump to return the core address of IEAQADOA.
2. It is critical to Snap Dump that the module loaded is the actual system module IEAQADOA.

3. The identity of the module loaded is not verified by either Load or Snap Dump.

This raw pattern can be abstracted to higher-level components so it can be applied to a wider class of operating systems. The error pattern in its abstracted form looks like the following:

1. Supervisor procedure A is called by supervisor procedure B to return the core address of a procedure or data element C having name N.
2. It is critical to B that C is the bona fide system element named N.
3. The identity of C is not verified by either A or B. [5]

In the studies of Carlstedt et al., error patterns were applied to operating systems such as OS/360, Multics, TENEX and Exec-8. The authors did not include any numerical or statistical results from their study, but they qualitatively described searches that included the aid of error patters as effectively helping identify many more errors in a system when compared with blind searches that did not include the use of error patterns. In one case, two of the same errors were textually adjacent to each other. The first error was found, but the second was not found until weeks or months after the discovery using the error pattern that described the error. This implies that although a person is able to identify an error they may not be able to find other instances even if the errors are in the same vicinity as the found one.

In 1999, Schneier [17] developed the idea of attack trees to model different attack scenarios on the same target resource. An *attack tree* is a tree of nodes that represent events that an attacker can perform to achieve an attack; an example of an attack tree appears in Figure 1.

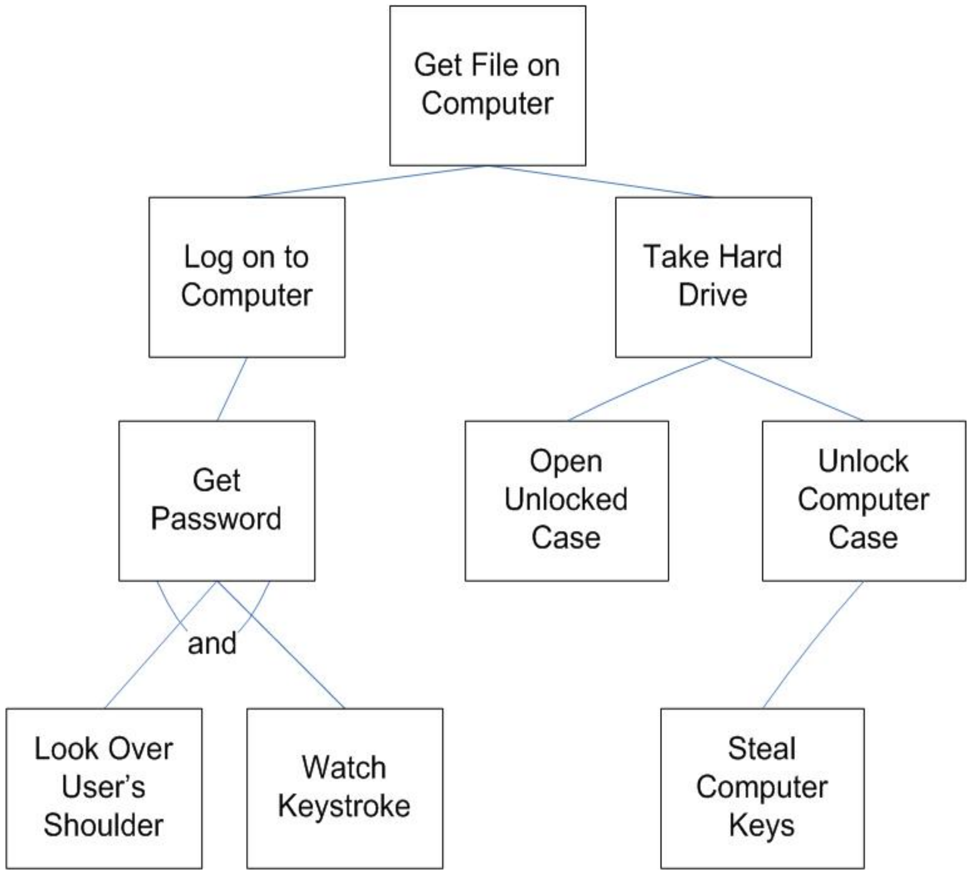


Figure 1: Get File Attack Tree. An attack tree that demonstrates possible scenarios for obtaining a file on a computer.

Each root node of the attack tree represents a goal of the attacker, and each leaf is a possible starting point of an attack. Each node under the root node is either an AND node or an OR node. AND nodes are those nodes that represent multiple goals an attack must accomplish to achieve the next goal. OR nodes are those nodes that an attacker can accomplish independently to reach his/her next goal. AND and OR nodes are distinguished by the insertion of "AND" directly below the parent that has at least two children nodes that represent events that must both be achieved to progress forward in the attack. All other nodes (except the root node) are OR nodes by default. Like the error patterns in the approach by Carlstedt et al. [6], attack trees can be applied to other systems where the same attack scenarios are possible. Therefore, attack trees are also reusable patterns that can be used for the identification of vulnerable components in a system. Unlike Carlstedt et al. [6] who used text-based descriptions, Schneier [17] uses a graph-based approach that shows an attack from the point of view of the attacker whereas the Carlstedt et al. approach uses a series of events in the machine instructions that results in an error.

Moore et al. [15] extend the idea of attack trees by creating attack patterns and attack profiles. Like attack trees, *attack patterns* contain the overall goal of the attack and the steps for achieving the attack. A list of pre-conditions for the attacker to attack the system is provided as well as the post-conditions, which are the results of a successful attack. Moore et al. [15] propose that an attack profile be associated with attack patterns. *Attack profiles* describe a common reference model for an analyst or designer to understand how the pattern can be applied to any architecture related to the vulnerability. Also included in the profile are a set of variants, a set of attack patterns, and a glossary of defined terms and phrases.

An attack profile should be generic enough to be applicable to any organization's architecture. Designers and analysts read the generic description and look for instances where they can apply the attack profile. Once the architecture is identified as potentially vulnerable, the attack patterns are used to demonstrate the attack scenarios associated with that architecture. The idea of using generic profiles facilitates the process of applying attack trees to different architectures, which aids in the extendibility and thus usefulness of attack trees. An attack tree alone does not provide the additional information that an attack profile and attack pattern contain and thus the approach set by Moore et al. may aid in the identification and understanding of an attack scenario.

In 2001, McDermott developed another graph-based approach to security called attack nets. *Attack nets* are based on Petri nets, Schneier's [17] attack trees. Attack nets are similar to attack trees in that they can have an attack tree structure and show different attack scenarios in a system; see Figure 2 for an example of an attack net.

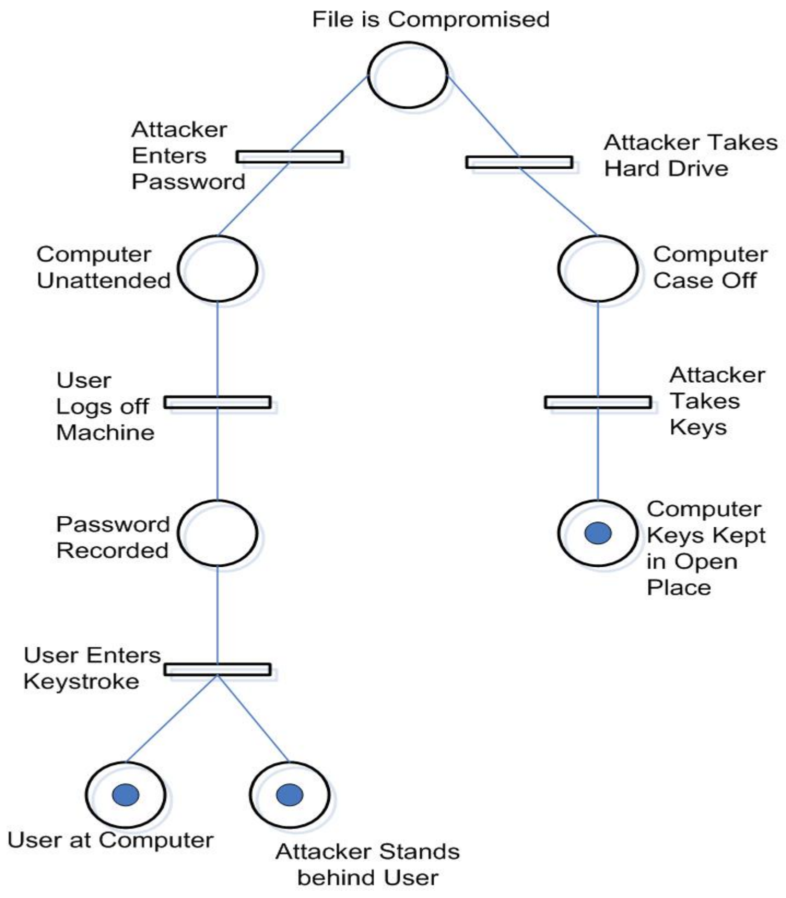


Figure 2: Get File Attack Net. An attack net that demonstrates possible scenarios for obtaining a file on a computer. The tokens at the leaves of the attack tree structure represent the progress of the attack.

Petri nets have different symbols/semantics than attack trees. Attack nets have *places* that are analogous to nodes in an attack tree. Places are connected by transitions which represent the actions of the attacker. Additionally, *arcs* are used to connect transitions to *places*. A *token* is used to represent the attacker's progress in the attack net as it moves from place to place via transitions. Like attack trees, attack nets are reusable for different systems with the use of generic places and transitions. However, the use of a token gives an advantage over attack trees because the sequence of events is shown via the token. Attack trees do not represent the sequence of events between multiple nodes at the same level of the attack tree. Also, attack nets can model inputs or commands at the transitions that attack trees cannot show. Petri nets need not take the form of an attack tree, but may also be cyclic to show an attack that can repeat the same sequences of actions. For example, a cyclic Petri net may show an attack logging on to different machines on the same network [19].

The supplementary information about the attack can be included in an accompanying document to convey a detailed description of the attack that helps users identify what scenarios are possible in their system. Both the attack tree and attack net scenarios show the view point of an attacker by indicating the attacker's location and their goal. The attacker's goal is not explicit in the text-based model used by Carlstedt et al. [6] and is thus an advantage of these graph-based approaches. Furthermore, a graph-based approach is likely to better illustrate an attack than a list of actions in a text format.

Another graphical approach is an extension of UML, UMLsec, [12] that was developed to facilitate the identification of security vulnerabilities in application designs. Stereotypes along with tagged values and constraints are used to encapsulate standardized security requirements in UML diagrams. Software engineers who design their systems with UMLsec can use pre-defined threat scenarios associated with the stereotypes to learn what attacks are possible in their system and what measures need to be taken to prevent them. For example, a client and sever communication can be represented with the <<Internet>> stereotype as opposed to the <<encrypted>> or <<LAN>> stereotype. The <<Internet>> stereotype has an associated threat profile that suggests an attacker can perform a delete, read, or insert in the communication. With this knowledge, software engineers are prompted with the standard security requirements are associated with each stereotype. This technique requires software engineers to know UML, which is a beneficial skill for applications that have static requirements at the beginning of the software process.

Bishop and Dilger [3] experiment 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. The specific class of TOCTTOU analyzed is the binding flaw where identifiers of an object are assumed to be true by the second event that carries out an action. The two types of identifiers for files in the UNIX operating system are path names and file descriptors. Path names are a path of pointers that start from the file system root and traverse through subdirectories down to the file. File descriptors are pointers that point to the memory address of the file without having to traverse the file system. File descriptors are bound to a file object making the naming scheme a more direct approach to identifying a file. An example of a TOCTTOU flaw can be demonstrated in a setuid to root program that first uses a file system call to check the access rights of the user before opening it. If the access rights allow the process to open it, then a second file system opens the file as it should. If, however, the filename used to identify the file changes between the first and second file system calls, then the program can open a file in which the user does not have privileges.

Bishop and Dilger [3] applied an analyzer, written in perl, to scan the code in the *sendmail* version 8.6.10 application for the *program intervals* of two file system calls to the same file. The *programming condition* for the race condition depends on how the file was referred. If both of the file system calls refer to the file by its path name, then an attacker can change the pointers in the construction of the pathname and thus alter the binding of the name to the file. If both file system calls refer to the file using file descriptors, then the race condition will not be possible. If only one of the file system calls uses the path name to refer to the file, then the race condition is possible. A human analyzer was used to determine if the programming condition was present for the flagged pairs of file system calls. Then, the environmental condition was manually inspected to determine if a race condition was possible for the file system calls. Their findings show that the analyzer found 24 programming intervals in the *sendmail* application. Only five of the 24 pairs met the programming condition that was susceptible to the race condition. The environmental conditions for each of the five pairs of file system calls permitted the race condition. One of the five pairs was a previously undiscovered flaw and was made known to the program vendor.

Bishop and Dilger [3] show that a pattern-directed approach for identifying vulnerabilities can be applied directly to source code. Their study suggests that different entities can be searched in a system and determined if their relationship can cause a vulnerability. Unlike Carlstedt et al. that set out to find many error types for an operating system, only a specific type of vulnerability was sought in the source of C programs. The specificity of the source code analyzer is also a differentiating factor in that attack trees and attack nets can be used to show different scenarios of an attack. This is the first approach mentioned in this thesis that attempts for an automated identification process. However, manual analyses are still required to determine if the cases returned by the analyzer are true positives or not.

The goal of intrusion detection systems (IDS) is to model the behavior of a completed/operational system during an attack [14]. This is unlike the objective of attack nets and attack trees, which is to show different scenarios of attacks in a system. Also, IDS cannot show access control violations such as those in TOCTTOU binding flaws. IDS represent the last line of defense against attacks since all development efforts have completed and the product is must be operational to be used with an IDS.

Kumar and Spafford [13] propose a generic pattern-matching technique as a method for the identification of attacks with IDSs. As with Carlstedt et al. [6], Moore et al. [15], Schneier [17], and McDermott [14], the IDS uses abstract representations of attacks to identify specific attacks in different contexts. There are three levels of abstractions used: Information layer, Signature layer and Matching Engine. The Information Layer is comprised of the raw data used in the analysis. This can either be in the form of an audit trail or network packet. The Signature Layer is a means of representing attacks by their characteristics, which needs to be generic for the IDS to be system-independent. Lastly, The Matching Engine is used to match patterns of actions of known attacks to the data in the Information Layer.

The signatures of attacks are a set of strings that are matched by a Colored Petri Automata (CPA). The CPA is based on a Colored Petri net [10] that uses colored tokens for an advanced knowledge of the path the token took from the start state to the end state. In the approach by Kumar and Spafford [13], tokens contained a local set of variables that were written to as the token progressed along the path. All strings that are defined in the Signature Layer are defined by scenarios that can be achieved in the CPA. Upon a match in an audit trail or network analysis, the IDS can signal an alarm to a system administrator that an attack is progressing in their system.

Kumar and Spafford [13] acknowledge that their approach has limitations. The IDS should be able to process large numbers of entries in the audit trail records or in network packet transactions. The amount of data to be analyzed and the number of strings to be matched against can be a time consuming process. If too much time is required to analyze the ongoing processes of the system, then the IDS may not be able to identify an attack before the attack has already exploited the system. However, their IDS does have the advantage that it is portable to different systems to detect the same errors, which follows the trend of generic identification of the previously mentioned efforts.

The process of matching abstract representations to the instances in a system can be a fatiguing process especially in large systems. Manually searching for errors in code via error types is possible, but depends on the motivation of the individual searching for the error. An automated process of searching is possible, but in the case of the TOCTTOU binding flaws, the search is specific to only one type of error. In the case of the IDS searches, many attacks can be identified, but these type of attacks are not data flow or access control oriented and are used only one the software product is released by the vendor. Applying attack trees to a system requires that either the design or code to be in place, but does not offer a facilitated process of matching the scenarios to the system. Also, attack nets were meant to be created by a brainstorming process when the design or doe of a system is analyzed. This thesis approach attempts to aid the matching process of abstract representations to design by explicitly indicating where in the design a vulnerability may exist. Thus, a stakeholder is not required to inspect a component in the design to determine if the attack is possible and repeat the same process for each remaining component in the system design.

2.2 Security Collaboration for non-experts

The number of vulnerabilities in a new system may be very large and the effort to find these vulnerabilities time consuming, making the security process difficult. The task of identifying a complete set of all of an organization’s system vulnerabilities is infeasible by one or a few security engineers. Furthermore, security engineers, especially if contracted, are not likely to know what digital assets are the most sensitive and should have the most precautions to assure secrecy. Thus, a security team may offer the most plausible way to answer these deficits. The Operationally Critical Threat, Asset, and Vulnerability Evaluation

(OCTAVE) Method [1] created at the Software Engineering Institute at Carnegie Mellon University is a method that encourages an interdisciplinary team of business, information technology, partners, contractors, service providers and end users to collaborate on the current state of the security, what risks are possible, and what security strategy should be taken to secure their system. The broad view of security enables the risk assessment and risk management to be effective as possible.

Without a formalized notation of attacks that is readable to those with no security experience or those unfamiliar to a vulnerability, security is limited to experts that may or may not be available. The practice of abstracting security vulnerabilities and attacks to high-level representations, such as was done by Carlstedt et al. [6], Schneier [17], Moore et al. [15], McDermott [14], Steffan and Schumacher [19], not only makes the vulnerability or attack portable to different contexts, but provides a means of enabling individuals without thorough security backgrounds to perform security analyses. This thesis also suggests a means for non-experts in the security field to contribute to the security process. The text-based vulnerabilities used in Carlstedt et al. [6] approach are at the level where non-experts can read the vulnerability and look for the vulnerability in the operating system code. However, Carlstedt et al. mention that the workers that perform the analyses must at least be familiar with the system to perform the pattern-directed approach. Nonetheless, in his observations Carlstedt [6] noted individuals without any previous experience with protection evaluation could find errors with the use of error patterns. Furthermore, Moore et al. [15] produced text-based attack profiles in which non-experts could read comprehensive descriptions of attacks and apply them to their architecture for the identification of attack scenarios.

The graph-based approaches may offer insight on an attack and thus may be a more effective method of identifying attacks. Schneier [17] comments that the attack tree is a difficult process that requires practice. However, once the attack tree is made, non-experts can apply the attack trees to their system for assessing attack scenarios. However, pictures alone cannot describe an attack. The information provided by transitions and tokens in attack nets may still be inadequate to describe an attack. Steffan and Schumacher [19] use an approach they term ATicki, which uses a WikiWikiWeb [7] to share information about attacks between security experts and non-experts. Conditions and transitions are hyperlinked to a WikiPage that contain background information, code samples and discussion threads about the attack. Thus, a non-expert can quickly learn the details of an attack by reading about what others have entered in the web pages. To further clarify an attack, a WikiPage specific to the context of the attack is also provided. This page describes the context of the attack and can also reference other contexts that serve as the base context of the attack. For example, if an attack net describes a Red Hat Linux attack scenario, then the context page may reference a corresponding Linux WikiWikiWeb scenario since Red Hat Linux inherits its functionality from Linux.

The authors used a ATiki describing PHP vulnerabilities to test the efficacy of sharing knowledge between experts and non-experts. Vulnerabilities were analyzed in SecurityFocus and other security portals to initiate the descriptions of PHP systems and their vulnerabilities. Their analyses were met with ambiguities and questions that were noted on the WikiPages. These gaps were filled in by the users of the WikiPages and eventually the descriptions of the PHP vulnerabilities were adequate to identify precisely what occurred in an attack. The time required to obtain the accurate descriptions and the users of the system were not specified in the study.

This thesis' approach attempts to allow non-experts of software security to contribute to the identification of attacks in system designs. A non-expert can use a regular expression to match the sequence of events explicitly represented by architectural components to a sequence of components in a system design. The abstract representations of events conceal the low-level details of a component that may be confusing to non-experts. A successful match of a regular expression to a sequence of components in the system design by a non-expert suggests an attack path can be found without prior knowledge of that attack. The attack profile that accompanies the regular expression aids the non-expert (or expert) to understand the context and characteristics of the attack. The thesis approach is similar to the use of attack profiles and attack patterns by Moore et al. [15], but a non-expert may not be able to map the text in the attack profile to the components in the system architecture. Also, the approach in this thesis and the approach by Moore et al. [15] have more opportunity for non-experts to identify places of attack in a system using text-based descriptions to explain the attack. The approaches just mentioned are dissimilar to IDSs and code scanners in that collaboration not a part of the technique. Furthermore, code scanners use low-level descriptions that are not readily known to non-experts and abstract signatures of IDSs are only useful when the system is under attack.

2.3 Integrating Security into the Software Process

Security practices can be injected throughout phases of the software process. Some practices are limited to a particular phase where others can be applied to multiple phases. Applying security techniques to applications under development is best done early in the software process. Starting in the design phase permits security to be designed into the application. Knowing what threats are possible to a system allows for a risk management to guide well-informed decisions on how to approach potential attacks. Software developers can start coding security fortifications based on the strategies provided by the risk management assessment at the beginning of the coding cycle. This section provides a description of different techniques for applying security in the software process.

UMLsec is one example of a security approach that encourages security to begin in the design phase. UMLsec designers can identify possible threat scenarios in their UML diagrams and warn coders of possible attacks with the use of standardized security requirements that designate what security measures should be implemented in their code. UMLsec is intended for security-critical systems and is thus best suited for systems with static requirements that are known at the beginning of the software process.

The idea of attack patterns and attack profiles proposed by Moore et al. [15] is intended for designers and analysts. Similar to UMLsec, their approach provides a method of integrating security in the design phase, but the process does not include standardized security requirements. Instead, the *attack profiles* suggest where the organization's architecture is subject to attack. Thus, *attack patterns* can be applied in the design phase, but can also be applied in any other software phase where there is enough information exists to apply the graphical attack model.

McDermott's [14] use of attack nets is intended for penetration testing, which can be applied to the operation, implementation, and design of a system. The penetration testing approach McDermott bases his attack nets on is the flaw hypothesis approach [21], an approach that is commonly used for penetration testing. The six steps that McDermott [14] recommends for penetration testing using attack nets are: (1) define goals (2) background study (3) attack net generation (4) hypothesis verification (5) flaw generalization and (6) flaw elimination. The first step is to determine what will be tested in the system. Secondly, research on the system and its artifacts is performed to know the characteristics of architecture. In step 3, the attack nets are used to model hypothetical vulnerabilities and attacks. This is achieved by a collaborative effort among penetration testers imagining any attack scenario possible as long as it can be represented by an attack net. The hypothetical attacks are then tested against source code to determine if the scenario is valid. If the tests indicate the attacks are valid, then a check in the system is performed to determine if an identical or similar vulnerability exists. Finally, the attack scenarios undergo risk management to determine what vulnerabilities will be addressed and with what priority to fix them.

The approach of this thesis is to attempt the integration of security as early as possible in the software process. The approach using regular expressions and system designs offers the same early start of UMLsec, but does not restrict itself to standardized security requirements nor UML diagrams. Attack profiles, attack patterns, and attack nets also offer the same benefit of starting security early. Approaches that start early in the software process offer security analysts and perhaps non-experts more time to find and fix vulnerabilities than do code scanners such as the one proposed by Bishop and Dilger [3] or searching techniques in code as done by Carlstedt et al. [6] Lastly, techniques to find vulnerabilities that can be applied in the development stages of the software process, unlike IDSs, enable developers more time to implement security before the product is released into the hands of customers.

2.4 Risk Management

No computer system is 100% secure [20]. Security vulnerabilities are always present because they are not found or there are not enough resources (e.g. time, people, expertise) to secure them. Therefore, security should be implemented through risk management where each threat is prioritized by its cost to the system. Those threats that are the most pernicious to the system are secured first [16]. Carlstedt et al. [6] proposed an economical process to search for the errors in operating systems. Those errors that give the greatest security relative to the effort required to develop effective search packages for patterns of these types should be secured first. Schneier [17] proposed a means of identifying which attack paths should have the highest priority of receiving security implementation. Each node in the attack tree can be assigned a value that represents the probability of being exploited, the cost to exploit it, or the cost that results when the node is achieved. Attack trees can then be prioritized based on the sum of values in the nodes. It may not be necessary to assign security implementations to the attack trees with the highest values. The organization can consider the profile of the attackers that are likely to attack their system. High profile attackers have the potential to attack the most costly attack paths whereas low profile attackers may not have the knowledge of how to exploit a technical vulnerability. McDermott [14] also recommends the prioritization of the list of attack scenarios generated from the brainstorming session in step 4 of his process. Some of the scenarios may not be as likely in the system and are ranked with low priority to allot more time for the implementations needed to fortify against more imminent attacks. Finally, Kumar and Spafford [13] implemented a method of assigning priorities to the signatures of their IDS. The token associated with each signature can be assigned a thread that runs in the system and the thread can then be prioritized to match the attacks that pose greater risk to the system. The approach in this thesis does not explicitly assign probabilities or costs to attack paths because the integrity, secrecy, and confidentiality of digital assets of all systems may be vary among the type of system. A risk management team, if available to an organization, can rank the attack paths identified in the matching process of regular expressions to sequences of components in the system design according to the probability and cost associated with the system.

3.0 Methodology

In this chapter a background of software security as related to this thesis is provided. Next, a discussion of the use of regular expressions to represent attack paths is given, followed by a section that shows how to apply the regular expressions to a system design. The application of regular expressions to a design is used to expose attack paths that an attacker can use to exploit a vulnerability. The use of a knowledge base to store regular expressions to supply a comprehensive means of identifying many different attack paths is also explained. A scenario of creating and applying regular expressions to system designs is then given to exemplify the process of identifying vulnerabilities. Finally, a risk management section describes what an organization should do once the vulnerabilities of a system design are identified.

3.1 Background

Generally, software systems contain components that access resources (e.g. hard drive, memory) to do useful work. In this thesis, we use the term *component* to be any object in the system that transfers or requests information to any object or resource. We use the term *resource* to be any object that contains digital assets of the system, including those components that act as resources for other components. Preparing an upfront inventory of what components and resources are in a software system and how they interact is beneficial for determining possible security vulnerabilities. Additionally, an organization should create a security policy that includes (1) the identification of what resources are to be protected; (2) what threats to protect the resources from and; (3) a risk analysis of the threats to permit vulnerability analysis [8]. The interaction of components and resources should be carefully managed to prevent unwanted access to the resources in the system. To do this, a formal description describing the access control rights can be established to describe the restraints that prevent inappropriate accesses, which may otherwise lead to software exploits. Herein lays the focus of software security, the enforcement of the rules that enforce a controlled environment. Any malicious or inadvertent access because a rule in the policy is not enforced or does not exist is a *security breech*. [17]

The security policy should not have generic descriptions that describe access control rights for specific components. Instead, each component and resource in the system should have an accurate and detailed list of rules stated in the policy that demonstrates how that object should be accessed. Schneier [17] claims that security should answers questions such as “Secure from whom?” and “Secure for how long?” In this thesis, the strategy is not only from whom or for how long, but accounts for the components in the system accessing a resource. That is, one should not secure the hard drive by asking “How do I protect the hard drive?” They must instead ask themselves “From what components do I protect the hard drive?”

Each component may have different usage frequencies, demands, and authorization levels of a resource thus providing for many different access types. Furthermore, a component that directly accesses a resource may work with other components, combinations of different components, or may be used in different environments. All of these differences may contribute to subtle, undetected accesses that result in security attacks. Thus, a sequence of component interactions leading to a resource can provide for an accurate description of what information is sent to or requested from the resource. A clearly-defined mapping between components and the resources they access in the security policy may enhance the ability to produce access-control requirements. The requirements can then be entered in a security requirements document to show software engineers where to implement security implementations.

3.2 Regular Expressions

In this research, regular expressions are used to model attacks on software systems. A regular expression is a grammar that determines the set of strings in a language. Software engineers can match a set of strings that represent attack paths to system designs during the design phase to identify potential security vulnerabilities. Incorporating vulnerability analyses in the design phase affords developers to design for security. The security coding efforts that are required are recognized early in the software process, which offers two advantages to software developers. First, more time can be allocated to complete the task of securing a vulnerability. Additionally, the appropriate security tactics can be implemented in conjunction with the coding phase instead of added on afterwards when efforts to secure the system may be infeasible.

In this research, regular expressions are used as signatures of previously-successful attacks by representing the events that transpire at each software component involved in the path chosen by the attacker. Each regular expression begins with a "start" event, symbolized by the component that was used to initiate the attack. Each major successive event in the attack is expressed with its associated component and is appended to the string starting with the start component. Finally, the resource the attacker maliciously accessed terminates the string of components to complete the illustration of the attack path. For example, the regular expression below represents a user running a process on a CPU.

`(User)(Server)(CPU)`

1. A `User` (the start component) can make an excessive number of requests;
2. The `Server` (resource) accepts the requests; and finally
3. A process/thread runs on the `CPU` (resource).

The third event represents that the CPU can be consumed by the excessive number of requests by a malicious user. The user may have the required access privileges to make requests, but the server should not be allowed to create the large numbers of threads/processes that run on the CPU. The regular expression only provides a means for searching for specified vulnerable sequences of components in a system design. It does not clearly inform a software engineer how the attack will occur, but shows that the potential for a vulnerability exists and should be handled appropriately. Thus, a description of the attack, or attack profile as used by Moore et al. [15] to elaborate abstract attack patterns can be used in conjunction with the regular expression to accurately describe the events occurring at each component in the attack.

The security requirements doctrine is a document that is created by requirements engineers that contains the rules of how data flows between components. The requirements should be checked against the security policy of an organization that describes which digital assets of a system are to be protected. If the security requirements comply with the descriptions in the security policy, then developers can know what implementations are needed to secure their code. We propose that regular expressions can be used to illuminate the data flow and access-control scenarios that lead to security exploits by showing known instances of where they occur in a system design.

As mentioned, access control is not limited to a binary setting that permits or denies access to a component/resource. Instead, access control is viewed as everything in-between and including the binary permissions. In the regular expression given above, there is the possibility of a malicious attacker sending excessive number of requests to a web server. The web server can potentially consume the resources of the CPU, causing a denial-of-service. A security breech that results from an access control violation occurs between the client and the server because the security policy states that the server should restrict the number of requests from a client in a given time increment. Also, the server needs to restrict its access to the CPU as to allow other processes and threads to have a chance to perform their duties. Another example is that a malicious client could send an excessive amount of data to a web server in a buffer overflow attack. Thus, the access-control violation is that too much data was written to the memory on the server. In this scenario, access rights cannot be simply turned off or on because there is no way to determine the nature of the client. Thus, observing how data may be un-sanitized in the component transfer can help indicate a noncompliance with the security policy.

There are four regular expression operators used in this thesis, and these are listed in

Table 1: Regular Expression Operators. Four regular expression operators were used to 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 occurs zero times or once.

Regular expression operators can be used to further clarify the characteristics of an event. Each of these four operators will be explained by example in this section. In the earlier example there existed no regular expression operators associated with the events. The absence of regular expression operators implies that each event must occur once and only once for a successful attack. Thus, the previous example can be clarified by the use of operators:

$$(\mathit{User}^{+})(\mathit{Server}^{+})(\mathit{CPU}^{+}).$$

The regular expression now suggests that the `User` must make at least one request, followed by at least one acceptance of the request by the `Server`, followed by at least one process/thread from the malicous user occupying some number of `CPU` cycles to cause a denial-of-service. It is difficult to know in advance exactly how many requests are required to perform the attack and hence the regular expression is intentionally ambiguous about how many requests need to be submitted to the `Server`.

The Kleene closure can indicate where an event may not be required. The same regular expression can be extended to look like

(User$^+$)(Server$^+$)(CPU$^+$)(HardDrive).*

A malicous `User` submits 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 disk and potentially consume the hard drive thus causing a denial-of-service. This event may or may not occur and is thus characterized with the Kleene closure.

Regular expressions can also make use of the ? operator. In this example,

(User)(CommandLineArgumentEntry)(ApplicationServer?)(Application)

(CommandLineArgumentBufferWrite)(Buffer),

a `User` works on an application, and enters an excessively long command line argument, which is read by an `Application`, which may or may not be on server, followed by writing the command line argument, followed by the data overflowing the buffer. This ? operator allows the regular expression to represent that a standalone or sever-based environment is susceptible to the same attack.

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

(User) (Variable + Filename + Header)(HTTPServer)(PostMethod)

(BufferWrite)(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 the `Variable` or `Filename` or `Header` written into a buffer, 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 exploit a small buffer on the vulnerable server.

The idea of expressing attacks with regular expressions includes the ability of portraying the same attack in different environments [11]. The components represented by the regular expressions are abstracted and expressed as generically as possible to accomplish a flexible usage in different systems. For example, a denial-of-service attack may occur when a client repeatedly makes a request to a web server. The regular expression for the attack can be represented as

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

which describes a series of `Client` 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 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 same vulnerability may occur in a database, FTP, or audit server environment. Therefore, the regular expression is generic enough to represent the database server with `Server`. Also, the same attack may occur if an error log file becomes large, and thus the event, `LogFile`, is generic enough to represent either access log or error log.

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

$$(\mathit{Class})(\mathit{Subclass})(\mathit{OverriddenSecuredMethods})(\mathit{Application})$$

are specific to low-level software designs where classes and methods are specified. The attack 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`. This representation is not overwhelmingly flexible in that it could only apply to programming languages that allow a parent class to be extended, such as Java or C++. This regular expression was the only code level vulnerability extracted from the analyses performed for this study and, thus, the focus of this thesis is at higher/system level vulnerabilities.

A security engineer should always anticipate that an attacker will either defeat or bypass a defense. Thus, a security measure should be implemented at each possible component in an attack path leading toward the target resource. A regular expression can indicate the multiple opportunities to prevent an exploit to encourage layered defenses, a best practice technique [11]. The regular expression,

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

describes an attack where a `User` submits at least one large GET request, followed by the `HTTPServer` accepting the request at least one request, followed by the `GetRequestRoutine` processing at least one request and writing it to a `Buffer` causing a buffer overflow. Also, 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 Server 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.

3.3 The System Design

A suitable format for the application of regular expressions is a system design. Designs show the components that are involved in the application and how they interact with one another. System-level designs typically have components connected by arrows that indicate data flow, as shown in Figure 3.

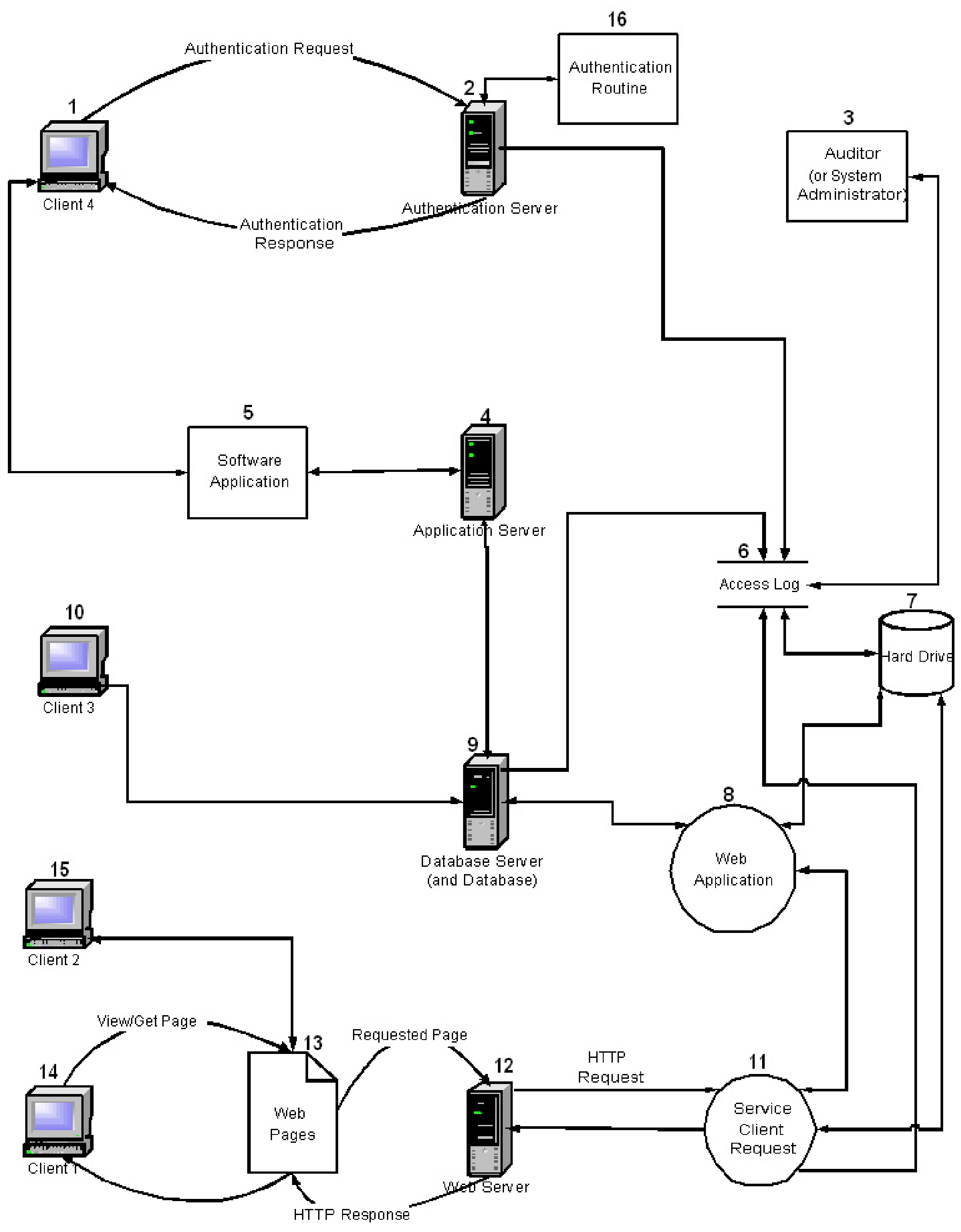


Figure 3: A Data Flow Diagram. A data flow diagram derived from Howard and LeBlanc [11] can show the flow of data between components, which can be captured by regular expressions.

Data flow diagrams were chosen for this research because most of the regular expressions represent high-level components involving data transfer. Also, high-level designs are not impacted as severely as low-level designs upon a change in requirements, which is common in the software life cycle. However, not all regular expressions are limited to system designs. For instance, the regular expression,

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

fits best with a low-level design, such as a UML diagram. Regardless of high-level or low-level, regular expressions are constructed to literally match with sequences of components in the design. Thus, a security analyst physically maps the regular expressions to sections of the design. A match between the regular expression and string of components signifies a potential vulnerability. For example, the regular expression

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

can be mapped to `Client 4` (component #1) making a request to the , `Authentication Server` (component # 2), which logs the request to the `Access Log` (component # 6), which are stored on the `Hard Drive` (component # 7). The sequence of components can be represented as “1-2-6-7” which shows a possible attack path in the system design.

3.4 Knowledge Base of Regular Expressions

No application should be a victim of the “oldest trick in the book.” Therefore, an organization should have a compilation of regular expressions and their corresponding profiles in a knowledge base. A knowledge base can be used to aid in the prevention of old and forgotten attacks from recurring. Also, if the newest attacks are stored in the knowledge base, then future applications may not suffer from these attacks either. Retaining the profiles and possibly methods of securing attacks in a knowledge base achieves the problem of losing tacit knowledge in an organization. This knowledge is especially useful in the case where a security expert has been temporarily contracted for an organization. The knowledge base thus serves as the collection of regular expressions that are used to examine system designs for potential vulnerabilities.

3.5 Methodology Scenario – Securing Applications from Enumerated Threats (SAFET)

The OCTAVE Method [1] encourages all stakeholders in a software project to be involved with the assessment of which threats are possible in the system. Customers, marketers, and information technology engineers each have a different perception of how the system will achieve its goals based on their background knowledge of their organization, customer's demands, and development strategies. For example, customers and business workers may have an advanced knowledge of the tendencies and limitations of users. Additionally, software engineers may have an advanced lower-level view of the system that grasps the intricacies of component functionalities. Bridging these specialized views by working together can help elicit the desired behaviors of the system and illuminate its potential threats.

Thus, a potentially useful format for representing security vulnerabilities for the diverse team is regular expressions. Regular expressions offer a high-level, human-readable model that abstracts operational and low-level detail to achieve a common view of the system among stakeholders. Using a structured format, stakeholders can collaborate on the identification of vulnerabilities in a system design. Furthermore, mapping the events that cause an attack to the components that are responsible for triggering the events narrows the scope of a search for the attack in a broad system design. Thus, stakeholders (technical or non-technical) can participate in the matching of a given set of attack paths, represented by regular expressions, to the sequence of components and resources in the system design. A team effort of exposing threats to a system can show different views of an attack and how they should be addressed. The security analysis in an organization’s design for known threats is given the reference Security Analysis for Existing Threats (SAFET).

A security engineer is beneficial for this approach because of their expert knowledge in attacks in software systems. They can help identify subtle attacks that may be overlooked by non-experts or determine if an attack path found by stakeholders is valid or invalid. However, the role of security engineer is not necessary if software engineers have enough expert knowledge about security. Also, a security engineer may not be available to a software team and thus they must make do with what they have. The approach proposed by this research provides a means of identifying security vulnerabilities to provide enough basic security to aid non-experts with important security advice so an application can be released with some confidence of withstanding attacks.

3.6 Risk Management

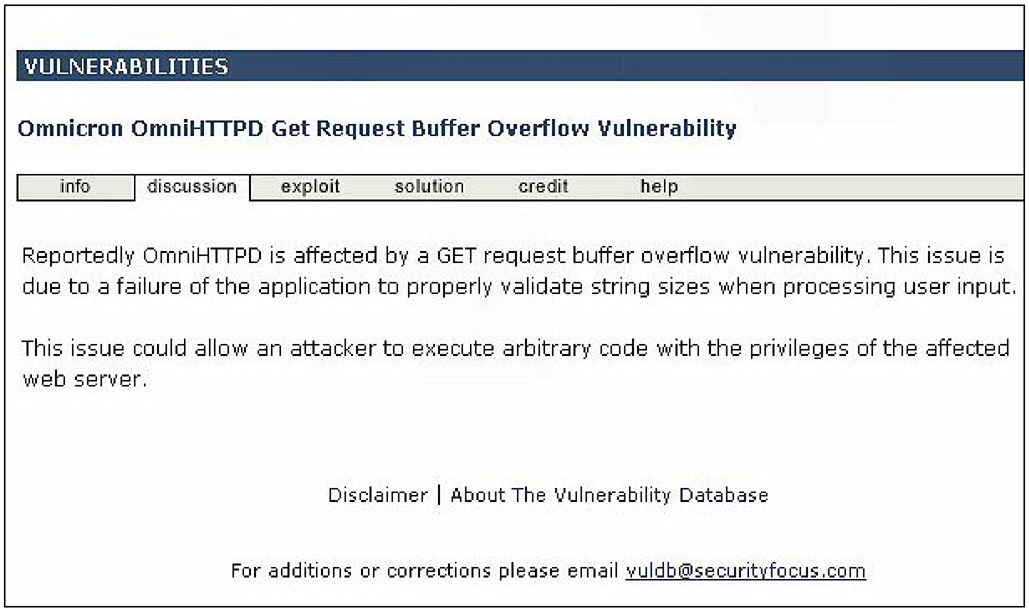
Many security vulnerabilities are possible in large complex systems, and it is infeasible to secure each one. A risk management team is necessary to prioritize the attack paths found based on how much each threatens the system. The security engineer should be a member of the risk management team is made of members. The vulnerabilities with the most risk are then assigned to software engineers so that security measures can be built into the system from the start. Each component should be analyzed for potential threats and then a numeric value of the risk involved should be calculated using the following Equation 1 [5].

$$\text{Risk} = (\text{probability}) * (\text{loss}) \quad \textbf{(1)}$$

In Equation 1, the risk of the attack is calculated by multiplying the probability of the attack occurring by a numeric value on what the organization stands to lose if the attack occurs. The risk assessment for each vulnerability should also include the risk of not meeting the project goals such as functionality, usability, efficiency, time-to-market because of the incorporation of security implementations in the application [20]. This equation allows risk analysts to rank the threats of the software system and thus identify the highest ranking threats as those that should be secured first. This tactic is often used because it is infeasible to address all security threats and so the most impending threats should be secured before addressing less costly threats.

4.0 Vulnerability Collection Methodology

Publicly-known vulnerabilities were collected from the following four full-disclosure vulnerability databases: SecurityFocus owned by Symantec, Help Net Security independently owned by Help Net Security, Secunia, an IT security company, and SecurityTracker owned by SecurityGlobal.net LLC. The SecurityFocus database was the most preferred database because of its ability to search for security attacks and the organization in which it used to display the attacks. An analysis was made of the "discussion" and "exploit" pages for each attack entry in the database (see Figure 4).



Omnicron OmniHTTPD Get Request Buffer Overflow Vulnerability

Reportedly OmniHTTPD is affected by a GET request buffer overflow vulnerability. This issue is due to a failure of the application to properly validate string sizes when processing user input.

This issue could allow an attacker to execute arbitrary code with the privileges of the affected web server.

Disclaimer | About The Vulnerability Database

For additions or corrections please email vuldb@securityfocus.com

Figure 4: SecurityFocus Example. A page from SecurityFocus that shows the different pages about a GET request vulnerability.

These pages usually contained the information needed to analyze the attack, but occasionally there was a script available to analyze the attack, too. The date, title, and SecurityFocus ID (see Appendix I) were recorded for the purpose of making a knowledge base of surveyed attacks. Each attack was investigated in the three other named vulnerability databases to confirm the accuracy and ensure the completeness of the attack information before continuing.

At the beginning of the vulnerability database analyses, I attempted to determine exactly how the attack occurred by performing searches on the Internet. The searching yielded hits that included informal web pages and message threads. The process of searching and studying the attacks could require many hours and sometimes there was not any additional information that proved useful. Furthermore, some searches did not yield any hits making the results of the effort inconsistent and thus hinder the ability to make accurate conclusions about the study. Therefore, information about the attacks was exclusively obtained from the four vulnerability databases to provide a consistent means of gathering information.

If enough information was available to understand the attack, then a regular expression was generated. Otherwise, the attack was marked as not having enough information. At the beginning of the study I had no experience with security vulnerabilities. For each vulnerability I had to research the terms and descriptions that were included in the attack. For example, I researched how cross-site scripting vulnerabilities occurred when I first encountered a cross-site scripting attack. When I next encountered a cross-site scripting error I was more likely to know how the attack occurred. However, some cross-site scripting errors had variations and required further study. The first week of the analyses resulted in understanding approximately ten vulnerabilities. By the end of the study, I could anticipate how an attack occurred before completely reading the description of the attack in the vulnerability database. Vulnerabilities that were either repeated or similar to previously listed vulnerabilities required less than 10-15 minutes to analyze. Vulnerabilities that were cryptic or lacked enough information still required approximately a half an hour to study and examine in the four databases.

The means of determining if a vulnerability had enough information to describe the attack was subjective. The ability to understand the attack was based on my experience in computer science. A security expert with excessive experience is more qualified to make assumptions about details that were not explicitly stated in the descriptions. Thus, for the vulnerabilities that I claimed did not have enough information to form regular expressions may actually be understood by a security expert who could produce a regular expression. The regular expressions in this thesis are not necessarily the regular expressions that should be used by an organization, but may provide ideas and motivation for those who create regular expressions.

The process of producing a regular expression starts with the identification of what events transpired in the attack. The events are represented by the components in the software system that triggered the event. Components are then abstracted to a readable term that can best represent any component of its type. For example, “web server” is abstracted to `Server` to represent any server (e.g. database, FTP, audit). The terms are then placed in parentheses for readability purposes. Some events that could help elaborate an attack were inserted between the events that are represented by components. For example, `BufferWrite` is an event that may occur during a buffer overflow attack and may be inserted between the process that performed the event and a buffer. Next, the components are logically arranged in the temporal order of the attack. If an event could be characterized by any of the regular expression operators (?, +, $^+$, *), then they were applied to term. A brief attack profile was also made to elaborate the abstract regular expression. A unique integer ID was then assigned to the regular expression and was entered into the regular expression knowledge base along with its corresponding attack profile. Such a knowledge base can be found in chapter 5.

Each successive attack examined was compared to the pre-existing regular expressions and attack profiles in the knowledge base. If the attack was a repeat, then it was given the ID of the already existing regular expression. If the attack was similar enough to be included in a pre-existing regular expression, then the two regular expressions were merged. For example, a buffer overflow may occur in a POST request from a large hidden variable, a large filename, or header field value as represented in regex4. Since these three methods of overflowing a buffer are similar they were combined into one regular expression to look like:

```
(User)(HTTPServer)(PostMethod)(Variable + Filename  +
                   (Header)(BufferWrite)(Buffer)
```

If the regular expression was too dissimilar to any of those in the knowledge base, then it was added as a new entry.

4.1 Limitations

The proposed approach of using regular expressions and designs is based on two assumptions. First, all components in the design have not already been created, that is no third party software that already has the vulnerability secured is used. It is presumed that the vulnerabilities will exist with the sequence of components/resources unless a security measure is taken. The second assumption is that stakeholders use a design in their software process. A high-level design is the minimum requirement for regular expressions to be used for a security analysis. A low-level design would work best if the requirements are well known and are static at the beginning of the software process.

The accuracy of the regular expressions is dependent on the precision of the information offered in the vulnerability databases. Each database allows anyone interested in security to subscribe to their mailing lists to receive the latest security advisories. SecurityFocus, Secunia and SecurityTracker allow anyone to post security vulnerabilities to their web pages.   Thus, the validity of this study is dependent on the accuracy of those posting the vulnerabilities. The vulnerability databases, especially SecurityFocus (or Bugtraq), have been referenced in many research papers and are likely legitimate sources of information for security studies.

5.0 Vulnerability Collection Results

At the initial stages of the research, the analysis approach was piloted to determine the viability of the approach for abstracting the variety of attacks present. At the time the feasibility study was conducted twenty-two vulnerabilities had been analyzed from the SecurityFocus database; these vulnerabilities were able to be abstracted by five regular expressions (see Table 2).

Table 2: The Initial Five Regular Expressions. An initial analysis of 22 vulnerabilities yielded five regular expressions. The frequency of the vulnerability occurring in the collection is also given.

Regular Expression	Profile	Frequency
`(Client`$^+$`)(Server`$^+$`)(Log`$^+$`)(Hard Drive`$^+$`)`	An attacker can exceedingly access server augmenting an access or error log file and eventually fill the hard drive causing the system to crash.	15 (68.2%)
`(User)(Machine)(SyslogFunction)(Log)`	It is possible to corrupt memory by passing format strings through the Syslog() function, a logging function. This may potentially be exploited to overwrite arbitrary locations in memory with attacker-specified values	4 (18.2%)
`(Client)(HTMLPage)(Server)(Hard Drive)`	A user may submit an excessive amount of data in an HTML page, thus filling up the hard drive	1 (4.5%)
`(Client)(Server)(GetMethod)(GetMethodBufferWrite)(Buffer)`	Writing an excessively long Get request into a small buffer will cause a buffer overflow	1 (4.5%)

Table2 (continued)

`(Client)(HTMLMessageBoard)(Server)(HTMLMessageBoard)(Client)`	An attacker may include hostile HTML and script code in posts to a message board. This code may be rendered in eh web browser the user who views the message.	1 (4.5%)

Of this sample, there was evidence that attackers commonly exploited vulnerabilities in the HTTP GET request routine. A search on “GET request” in the SecurityFocus database yielded 35 attacks since December 1998. This frequency gives evidence that the vulnerabilities in SecurityFocus represent a serious threat and are worthy of study.

The most recent attacks at the later stages of the study were obtained by searching for the latest posts to the SecurityFocus database. Three hundred and fifty-two vulnerabilities posted between 19 January 2004 and 9 March 2004 were analyzed and reviewed further in Help Net Security , Secunia , and SecurityTracker. Of these, one was a report and the other a duplicate of a previous vulnerability in SecurityFocus. Therefore, a total of 409 (including the 22 vulnerabilities from the initial study and the 35 vulnerabilities from the search in SecurityFocus) entries in SecurityFocus were studied and of these were 407 confirmed vulnerabilities[5]. These vulnerabilities were studied and subjected to the process of regular expression generation.

One hundred and seventy (41.8%) of the observed vulnerabilities were not used to make regular expressions. Table 3 sums the type of vulnerabilities excluded from the study.

[5] Please note that the results in this study may no longer agree with the data published in the online databases. These databases periodically update attack profiles and may include more attacks that had not been revealed at the time of the analysis

Table 3: Classes of Vulnerabilities Not Used. Six classifications of vulnerabilities were not used to make regular expressions.

Description	**Frequency**
Lack of information	85 (20.9%)
Specific to vendor	48 (11.7%)
Inapplicable	21 (5.2%)
Networking	14 (3.4%)
Encryption	1 (0.25%)
Hardware	1 (0.25%)

Eighty-five (20.9%) of the vulnerabilities in SecurityFocus lacked descriptions with sufficient detail to form regular expressions. Often, Help Net Security, Secunia, and SecurityTracker contained almost identical descriptions to each other and SecurityFocus suggesting the lack of detail made available to the public. The absence of detailed information is potentially due to the vendors not sharing data because the information could be used maliciously or for market advantage [18]. Furthermore, the description that is usually given is at a high level, thus preventing the ability to form a regular expressions to high-level abstractions. Therefore, using online vulnerability databases may not be a sufficient means for understanding attacks.

The scope of this study was limited to common software application coding problems. Therefore, networking and encryption vulnerabilities were excluded. Networking attacks made up 14 (3.4%) of the attacks found and included attacks at the packet level, network protocols, port scan, and switch vulnerabilities. One (0.25%) 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 regular expressions represent. Therefore, other techniques such are needed in tandem to support the wide variety of vulnerabilities.

Twenty-one (5.2%) 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. Regular expressions rely upon on the interaction of multiple components and thus cannot be used to abstract these types of attacks. Lastly, there were 48 (11.7%) attacks that were specific to vendors and would not likely serve helpful in the protection of typical software applications. For example, these included Microsoft XP specific problems, Norton Antivirus crashing when scanning files in certain folders, and the ability for Internet Explorer to capture user keystrokes. Retaining attacks that are specific to vendors would increase the size of a vulnerability knowledge base and thus decrease the efficiency of matching the enumerated regular expressions to components in the system design.

A total of 53 regular expressions were abstracted from the 237 remaining vulnerabilities. Table 4 shows the regular expressions and associated profiles. These regular expressions will be classified and explained further in subsequent discussion.

Table 4: Regular Expression Knowledge Base. Fifty three regular expression were found from 237 vulnerabilities.

Regular Expression ID	Regular Expression	Attack Profile	Frequency N (%)
Regex1	`(User`$^+$`)(Server`$^+$`)(Log`$^+$`)(HardDrive`$^+$`)`	A user 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 -- a denial-of-service attack (DoS). This may also occur on servers that log errors.	1 (0.25%)
Regex2	`(User)(Message)(Server)(Header`$^+$`)(MessageHeaderHandler)(Memory + CPU)`	A user may send a message with thousands of headers (e.g. MIME headers) to a server, causing a server memory/CPU DoS.	1 (0.25%)
Regex3	`(User)(HTTPServer)(GetMethod)(GetMethodBufferWrite)(Buffer)`	A user 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. The attacker may be able to escalate their privileges.	28 (6.9%)

Table 4 (continued)

Regex4	`(User)` `(Variable + Filename + Header)` `(HTTPServer)(PostMethod)` `(BufferWrite)(Buffer)`	A user 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). The attacker may be able to escalate their privileges.	7 (1.7%)
Regex5	`(User)(Server)(Message)` `(HeaderFieldBufferWrite)` `(Buffer)`	A user may submit an excessively long header field value causing a buffer overflow on the server (e.g. HTTP, email headers). The attacker may be able to escalate their privileges.	5 (1.2%)
Regex6	`(User)(HTTPServer)` `(HTTPMessageHandler)(Log)` `(SysAdmin)(LogEntryRead)` `(BufferWrite)(Buffer)`	A user that submits an excessively long message to the server can later induce a buffer overflow when viewed by a system administrator. It is possible for the attacker to escalate their privileges.	1 (0.25%)
Regex7	`(User)(HTTPServer)` `(PostMethod)` `(HTTPContent-LengthHeaderValue)` `(HTTPMessagePayloadLength)` `(ServerConnectionState)`	A user 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). (see regex37)	1 (0.25%)

Table 4 (continued)

Regex8	`(User)(UserNameEntry)(PasswordEntry)(Server)(AuthenticationRoutine)(BufferWrite)(Buffer)`	A user that submits an excessively long string of characters for either the username or password may cause a buffer overflow in the authentication routine. The attacker may be able to escalate their privileges.	4 (0.98%)
Regex9	`(User)(SQLInput)(Server)(WebApplication)(Database)(Data)`	Failure to sanitize user input (e.g. query string) can allow a user to submit an arbitrary SQL query, thus allowing for unauthorized access to data. This regex is too abstract to cover the many possibilities of invalid SQL input.	19 (4.7%)
Regex10	`(User)(SQLInputField)(Server)(WebApplication)(Database)(CPU)`	An attacker may submit a malicious SQL query (such as a Cartesian join of all tables) consuming the CPU.	1 (0.25%)
Regex11	`(User)(CommandLineArgumentEntry)(ApplicationServer?)(Application)(CommandLineArgumentBufferWrite)(Buffer)`	A user may submit an excessively long command line parameter causing a buffer overflow. The attacker may be able to escalate their privileges.	2 (0.49%)
Regex12	(User$^+$)(HTMLPage$^+$)(Server$^+$)(HardDrive$^+$)	A user may submit an excessive amount of data in an HTML page, thus filling up the hard drive on which the server resides.	1 (0.25%)

Table 4 (continued)

Regex13	`(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. This is called Cross Site Scripting (XSS).	47 (11.5%)
Regex14	`(User)(Machine)(SyslogFunction)(Log)(Memory)`	It is possible to corrupt memory by passing format strings through the Syslog(), a logging function. This may potentially be exploited to overwrite arbitrary locations in memory with attacker-specified values. The Syslog function is often improperly used and is thus a target of attacks. Machine is any computer that uses the syslog function.	1 (0.25%)
Regex15	`(User)(ReadUserInput)(EnvironmentVariableWrite)(Buffer)`	A user may submit an excessively long environment variable causing a buffer overflow in the application. The attacker may be able to escalate their privileges.	5 (1.2%)

Table 4 (continued)

Regex16	`(User)(GUI/Browser)(BookMarkSave)(BookmarkBufferWrite)(Buffer)`	A user 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.	1 (0.25%)
Regex17	`(User)(Application)(File)(FileRead)`	An application that reads a file may throw an exception or halt if the file is corrupt or has been tampered with by an attacker.	1 (0.25%)
Regex18	`(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). The attacker may be able to escalate their privileges.	83 (20.4%)

Table 4 (continued)

Regex19	`(Class)(Subclass)(OverriddenSecuredMethods)(Application)`	Overriding methods that have been secured in a super class may create a software vulnerability. In Netscape 4.0 the ClassLoader overrode the definition of built-in "system" types like java.lang.Class - applications usually subclass ClassLoader - a better example is from http://java.sun.com/j2se/1.4.2/docs/guide/security/jce/JCERefGuide.html - suggests to not override methods/constructors in CipherInputStream because the class takes into account many security considerations.	1 (0.25%)
Regex20	`(User)(Hyperlink)(Server)(HyperlinkBufferWrite)(Buffer)`	A user 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 user may take over the application.	2 (0.49%)
Regex21	`(User)(Server)(MessageHeaderHandler)(Server)`	A user 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.	5 (1.2%)

Table 4 (continued)

Regex22	`(UserInput)(PointerDereference)(Application)`	A user may fail to submit a username causing a DoS. This could be the result of a pointer that is dereferenced to obtain the username, but NULL is returned instead.	2 (0.49%)
Regex23	(User$^+$)(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.	3 (0.74%)
Regex24	`(UserInput)(IntegerEvaluationRoutine)(BufferWrite)(Buffer)`	A user that supplies an integer larger than the integer variable type expected may cause an exception/buffer overflow or DoS.	6 (1.47%)
Regex25	`(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.	27 (6.6%)
Regex26	`(User)(Server)(SearchString)(Information)`	A user may insert a directory traversal such as "../../" in a search string (e.g. CGI) and obtain private information.	2 (0.49%)

Table 4 (continued)

Regex27	`(User)(SearchString)(Server)(Data)(User)(BufferWrite)(Buffer)`	A user that requests data from an untrusted server may receive large data and result in a buffer overflow. Often happens in gaming environments.	4 (0.98%)
Regex28	`(Read)(FileHeader)(BufferWrite)(Buffer)`	A user 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.	1 (0.25%)
Regex29	`(User)(EmailHeader)(Firewall)(Buffer)`	A user can overflow a buffer in their firewall with a large email header to escalate their privileges (the user can attack their own company's LAN).	1 (0.25%)
Regex30	`(User)(MalformedDTD)(SOAPServer)(XMLParser)(CPU + Memory)`	A user that submits a malformed DTD may cause the XML parser of a SOAP server to consume the CPU/Memory.	1 (0.25%)
Regex31	`(User)(HTTPRequest)(ProxyServer)(BufferWrite)(Buffer)`	A user that submits an excessively long HTTP GET request to a proxy server may cause a buffer overflow. The attacker may be able to escalate their privileges.	3 (0.74%)
Regex32	`(User)(RequestMessage)(Router)`	A user 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.	1 (0.25%)

Table 4 (continued)

Regex33	`(User)(HTTPgetRequest)(Router)(EmbeddedServer)(Bufer*)`	A user that sends an excessively long GET request to a router may cause a DoS via a buffer overflow or CPU consumption.	2 (0.49%)
Regex34	`(User+)(HTTPServer+)(GetRequestRoutine+)(Buffer + CPU)`	A user may submit consecutive multiple long GET request URIs to either consume the CPU or overflow a buffer.	2 (0.49%)
Regex35	`(User)(HTTPgetRequest)(Router)`	A user may submit a malformed GET request (e.g. a blank (NULL)) request and cause a router to DoS.	1 (0.25%)
Regex36	`(User)((FTPCommand+MailCommand)+OSCommand)(FTPServer + MailServer))(BufferWrite)(Buffer)`	A user that submits an overly long OS command or FTP/Mail command may cause a buffer overflow in the FTP/Mail server. The attacker may be able to escalate their privileges.	8 (2.0%)
Regex37	`(User)(Socket)(Server)(ExceptionThrown*)(Server)`	A user may cause an exception to be thrown in the server and cause it to hang. (No data needs to be transferred) (similar to regex 7)	2 (0.49%)
Regex38	`(User)(UserNameEntry)(PasswordEntry)(AuthenticationServer?)(AuthenticationRoutine)`	A user that submits a malformed username or password for authentication may cause a DoS (e.g. format string specifier) or NULL as part of the name may bypass the authentication routine.	2 (0.49%)

Table 4 (continued)

Regex39	`(User)(FTPRequest)(FTPServer)(BufferWrite)(Buffer)`	A user may submit a long directory request (e.g. in the URL of a browser) by using long directory names or "/" can cause a buffer overflow or DoS in the FTP server. The attacker may be able to escalate their privileges.	7 (1.7%)
Regex40	`(User)(FTPRequest)(FTPServer)(GetRoutine)(Server)`	A user that requests a file that does not exist on the server may cause a DoS (e.g. Get <unavailable file>)	2 (0.49%)
Regex41	`(Metafile)(SizeField)(FileHeader)(FileRead)(BufferWrite)(Buffer)`	A user that specifies the "Size" field of a metafile to be less than the actual file may cause a buffer overflow.	1 (0.25%)
Regex42	`(Application)(DownloadMalicousFile)(PredictableFileLocation)(AttackerReference)(Information)`	A user that saves files to predictable locations especially where applications let you reference them may allow for information disclosure.	2 (0.49%)
Regex43	`(Application)(FileCreation)(System)`	If an application creates a file/directory that allows malicious users to write to them (makes them symbolic links or simply changes them), then attackers can escalate their privileges.	7 (1.7%)

Table 4 (continued)

Regex44	`(ApplicationRun)(Privileges)(System)`	An application that runs with SYSTEM privileges and lets a user execute another program such as CMD.EXE may grant themselves SYSTEM privileges.	2 (0.49%)
Regex45	`(User)(MessageHeader+QueryParam))(Server)(System)`	A user may insert shell commands into a message handler on a server (e.g. email server), which may allow the attacker run those commands on that system.	3 (0.74%)
Regex46	`(User)(Message)(Server)(System)`	A user that submits a message (command) to the server before authentication may cause a DoS (done in C code).	1 (0.25%)
Regex47	`(SourceFile)(IncludeFile)(EnvironmentVariable+ProgramVariable+URLparam)(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	8 (2.0%)
Regex48	`(User)(MalformedFTPCommand)(FTPServer)(BufferWrite)(Buffer)`	A user that submits an excessively long FTP command may cause a DoS or buffer overflow.	1 (0.25%)
Regex49	`(User)(InvalidRequest)(ErrorMessage)(System)`	A user that submits an invalid request may be returned with an error message that shows the installation path of the server.	2 (0.49%)

Table 4 (continued)

Regex50	`(User)(Application)(Subprocess)(System)`	An application that spawns a subprocess to handle a user command must ensure that the subprocess does not have elevated permissions.	1 (0.25%)
Regex51	`(WebBrowser)(CLSID)(Filename)(System)`	A user that embeds a CLSID in the filename of a malicious file can trick a web browser into opening the file with a different application than intended.	2 (0.49%)
Regex52	`(Server)(QueryString)(Command)(System)`	A user may insert a command for the value of a URL parameter and execute that command on the server (remote execution attack)	1 (0.25%)
Regex53	`(User)(URL)(Server)(Device)(System)`	A user that submits a URL with a device as part of the request may cause a DoS (e.g. http://[victim]/COM1)	1 (0.25%)

On average, six vulnerabilities in the collection of vulnerabilities mapped to one regular expression. The top five regular expressions with the highest frequency of representation in the vulnerability collection are shown in Table 5.

Table 5: Regular Expressions Occurring Frequently in the Knowledge Base.

Regular Expression ID	Regular Expression	Frequency N (%)
Regex3	`(User)(HTTPServer)(GetMethod)(GetMethodBufferWrite)(Buffer)`	28 (6.9%)
Regex9	`(User)(SQLInput)(Server)(WebApplication)(Database)(Data)`	19 (4.7%)
Regex13	`(User)(InjectionOfMaliciousHTMLTags/scriptInURL/Form)(Cookie*)(FormData*)(ServerVariables*)(Information)`	48 (11.8%)

Table 5 (continued)

Regex18	`(SocketRead)(SocketBufferWrite)(Buffer)`	83 (20.4%)
Regex25	`(User)(HTTPServer)(GetRequestRoutine) (Application + Information)`	27 (6.6%)

After analyzing the results and the associated vulnerabilities, regex9, regx13, and regex18 were found to be too abstract to accurately describe exactly how vulnerability may occur. Regex13 attempts to represent the many types of cross-site scripting vulnerabilities that may occur on a web page. Because there are many different methods for this attack to occur (e.g. hyperlinks, URLs, Cascading Style Sheet (CSS) tags) there were many vulnerabilities associated with this regular expression. The regular expression should precisely identify where vulnerabilities may occur in CSS tags, URLs and hyperlinks otherwise stakeholders may needlessly perform a test for each case.

The high degree of abstraction is also true for regex9, a representation for SQL injection attacks. SQL injections attacks include a large number of possibilities that can depend on the database administer who determines the rights for users and the type of data that resides in the database. For example, if passwords are kept in a database, then a crafty query may be able to extract the passwords. If, however, there are no passwords stored in the database, then the attack cannot occur. One regular expression cannot effectively represent the many possible attacks to the many possible databases that exist. Regex9 and regex13 were nevertheless included in the feasibility and validation studies because of their frequency (combined 16.5%) in the analyses. Lastly, regex18, which represents any buffer overflow occurring from a socket connection, could be further specified by the 19 (35.8%) regular expressions shown in Table 6.

Table 6: Further Specified Regular Expressions. Nineteen regular expressions are specific representations of regex18.

Regular Expression ID	Regular Expression	Frequency N (%)
Regex2	`(User)(Message)(Server)(Header`$^+$`) (MessageHeaderHandler)(Memory + CPU)`	1 (0.25%)
Regex3	`(User)(HTTPServer)(GetMethod)(GetMethodBufferWrite) (Buffer)`	28 (6.9%)
regex4	`(User) (Variable + Filename + Header) (HTTPServer)(PostMethod) (BufferWrite)(Buffer)`	7 (1.7%)
Regex5	`(User)(Server)(Message) (HeaderFieldBufferWrite)(Buffer)`	5 (1.2%)
Regex8	`(User)(UserNameEntry)(PasswordEntry) (Server)(AuthenticationRoutine) (BufferWrite)(Buffer)`	4 (0.98%)
Regex11	`(User)(CommandLineArgumentEntry)(Application) (ApplicationServer?) (CommandLineArgumentBufferWrite) (Buffer)`	2 (0.49%)
Regex12	`(User`$^+$`)(HTMLPage`$^+$`)(Server`$^+$`) (HardDrive`$^+$`)`	1 (0.25%)
Regex15	`(User)(ReadUserInput) (EnvironmentVariableWrite) (Buffer)`	5 (1.2%)
Regex16	`(User)(GUI/Browser)(BookMarkSave) (BookmarkBufferWrite) (Buffer)`	1 (0.25%)
Regex20	`(User)(Hyperlink)(Server) (HyperlinkBufferWrite)(Buffer)`	2 (4.9%)
Regex27	`(User)(SearchString)(Server)(Data) (User)(BufferWrite) (Buffer)`	4 (0.98%)
Regex28	`(Read)(FileHeader)(BufferWrite) (Buffer)`	1 (0.25%)
Regex29	`(User)(EmailHeader)(Firewall) (Buffer)`	1 (0.25%)
Regex31	`(User)(HTTPRequest)(ProxyServer) (BufferWrite)(Buffer)`	3 (0.74%)
Regex33	`(User)(HTTPgetRequest)(Router)(EmbeddedServer) (Bufer*)`	2 (0.49%)
Regex34	`(User+)(HTTPServer+) (GetRequestRoutine+)(Buffer + CPU)`	2 (0.49%)

Table 6 (continued)

Regex36	(User)((FTPCommand + MailCommand) + OSCommand) (FTPServer + MailServer))(BufferWrite)(Buffer)	8 (2.0%)
Regex39	(User)(FTPRequest)(FTPServer)(Buffer Write)(Buffer)	7 (1.7%)
Regex48	(User)(MalformedFTPCommand) (FTPServer)(BufferWrite)(Buffer)	1 (0.25%)

Regex18 was created at the onset of the study and appeared to have an adequate amount of abstraction for vulnerabilities involving large amounts of data that are transferred via sockets. However, as the study continued, it was found that there were 19 different ways this socket-based attack could occur. Furthermore, regular expression such as regex3 and regex26, already map to a relatively high number of vulnerabilities and should thus be represented as distinct vulnerabilities to accurately capture the corresponding attacks. Because regex18 should be more definitive toward the identification of vulnerabilities in a system design, it was not used in the validation study.

Regex3 and regex25 mapped to higher numbers than most other regular expression not because the regular expression was abstract enough to capture many different types of vulnerabilities, but because the same attack was repeated frequently in the SecurityFocus database. Both of these regular expressions represent vulnerabilities in the GET requests to an HTTP server. Regex3 captures the events that occur when an attack submits excessively long requests that results in buffer overflows. Regex25 identifies the different types of malformed requests that attackers use to exploit the vulnerabilities in the application processing the request. The combined percentage, 13.5%, of their representation in the vulnerability collection demonstrates attackers' attraction to the request process. Thus, the use of two regular expressions to represent likely attacks to a system with the associated vulnerable sequence of components may provide a swift means for the identification of the vulnerabilities in a design.

The regular expressions have been grouped into seven classifications to demonstrate the ability of representing different types of attacks. The seven classifications are based on the means in which the attacker exploited a vulnerability: (1) buffer overflows; (2) malformed data; (3) inserted commands; (4) excessive data or requests; (5) access privileges; (6) error messages; and (7) miscellaneous means.

The most common type of attack present in the analysis was the buffer overflow attack. Twenty-one (39.6%) of the regular expressions in Table 7 describe the data flow that can cause buffers to overflow.

Table 7: Regular Expressions Representing Buffer Overflows.

Regular Expression ID	**Regular Expression**	**Frequency**
Regex3	`(User)(HTTPServer)(GetMethod)(GetMethodBufferWrite)(Buffer)`	28 (6.9%)
Regex4	`(User)(Variable + Filename + Header)(HTTPServer)(PostMethod)(BufferWrite)(Buffer)`	7 (1.7%)
Regex5	`(User)(Server)(Message)(HeaderFieldBufferWrite)(Buffer)`	5 (1.2%)
Regex6	`(User)(HTTPServer)(HTTPMessageHandler)(Log)(SysAdmin)(LogEntryRead)(BufferWrite)(Buffer)`	1 (0.2%)
Regex8	`(User)(UserNameEntry)(PasswordEntry)(Server)(AuthenticationRoutine)(BufferWrite)(Buffer)`	4 (1.0%)
Regex11	`(User)(CommandLineArgumentEntry)(Application)(ApplicationServer?)(CommandLineArgumentBufferWrite)(Buffer)`	2 (0.5%)
Regex15	`(User)(ReadUserInput)(EnvironmentVariableWrite)(Buffer)`	5 (1.2%)
Regex16	`(User)(GUI/Browser)(BookMarkSave)(BookmarkBufferWrite)(Buffer)`	1 (0.2%)
Regex18	`(SocketRead)(SocketBufferWrite)(Buffer)`	83 (20.4%)
Regex20	`(User)(Hyperlink)(Server)(HyperlinkBufferWrite)(Buffer)`	2 (0.5%)
Regex24	`(UserInput)(IntegerEvaluationRoutine)(BufferWrite)(Buffer)`	6 (1.5%)

Table 7 (continued)

Regex27	(User)(SearchString)(Server)(Data)(User)(BufferWrite)(Buffer)	4 (1.0%)
Regex28	(Read)(FileHeader)(BufferWrite)(Buffer	1 (0.2%)
Regex29	(User)(EmailHeader)(Firewall)(Buffer)	1 (0.2%)
Regex31	(User)(HTTPRequest)(ProxyServer)(BufferWrite)(Buffer)	3 (0.7%)
Regex33	(User)(HTTPgetRequest)(Router)(EmbeddedServer)(Bufer*)	2 (0.5%)
Regex34	(User$^+$)(HTTPServer$^+$)(GetRequestRoutine$^+$)(Buffer + CPU)	2 (0.5%)
Regex36	(User)((FTPCommand + MailCommand) + OSCommand)(FTPServer + MailServer))(BufferWrite)(Buffer)	8 (2.0%)
Regex39	(User)(FTPRequest)(FTPServer)(BufferWrite)(Buffer)	7 (1.7%)
Regex41	(Metafile)(SizeField)(FileHeader)(FileRead)(BufferWrite)(Buffer)	1 (0.2%)
Regex48	(User)(MalformedFTPCommand)(FTPServer)(BufferWrite)(Buffer)	1 (0.2%)

The ability to overflow an unprotected buffer is not always a difficult task for the attacker. Thus, it may be one of the first vulnerabilities sought in an attack. The challenge to attackers lies in the ability to insert exploit code into the adjacent memory location. Currently, libraries in programming languages such as C are becoming more secure to buffer overflows to prevent an attacker from elevating his/her privileges [20]. However, the frequency in which these attacks occur suggests that buffer overflows are an important vulnerability to identify and secure.

The second largest classification in the vulnerability collection contains ten (18.9%) regular expressions and describes the use of malformed data to exploit vulnerabilities. Malformed data includes: format specifiers; NULL values; incorrect format (e.g. failing to supply a ':' between a header field and header field value); negative value; special characters; malicious use of Unicode characters; and directory traversals that are used to cause security breeches, such as a range of application errors, data theft, and escalation of privileges. Ten (18.9%) of the regular expressions are used to capture the different scenarios where malformed data can exploit susceptible vulnerabilities (see Table 8).

Table 8: Regular Expressions Representing Malformed Data.

Regular Expression ID	Regular Expression	Frequency
regex14	`(User)(Machine)(SyslogFunction)(Log)(Memory)`	1 (0.2%)
regex17	`(User)(Application)(File)(FileRead)`	1 (0.2%)
regex21	`(User)(Server)(MessageHeaderHandler)(Server)`	5 (1.2%)
regex22	`(UserInput)(PointerDereference)(Application)`	2 (0.5%)
regex25	`(User)(HTTPServer)(GetRequestRoutine) (Application + Information)`	27 (6.6%)
regex26	`(User)(Server)(SearchString)(Information)`	2 (0.5%)
regex30	`(User)(MalformedDTD)(SOAPServer)(XMLParser) (CPU + Memory)`	1 (0.2 %)
regex32	`(User)(RequestMessage)(Router)`	1 (0.2%)
regex35	`(User)(HTTPgetRequest)(Router)`	1 (0.2%)
regex38	`(User)(UserNameEntry)(PasswordEntry) (AuthenticationServer?) (AuthenticationRoutine)`	2 (0.5%)

Hence, regular expressions can capture attacks dealing with large amounts of data and with malformed data. Buffer overflows and malformed data are two types of attacks that are the most directly related to data flow between components suggesting that the approach this thesis proposes is especially useful in data flow diagrams.

Attackers can also take advantage of systems that do not properly handle executable data that is submitted by the user. Malicious data can be inserted in the format of HTML tags in web pages, shell commands in message handlers, or command arguments in URLs unexpectedly on the victim machine. Upon executing the commands, attackers can learn information about other users or gain privileges on the system. The six (11.3%) regular expressions in Table 9 attempt to warn stakeholders what vulnerable code makes these attack possible.

Table 9: Regular Expressions Representing Remote Executions.

Regular Expression ID	Regular Expression	Frequency
Regex13	`(User)(InjectionOfMaliciousHTMLTags/scriptInURL/Form)(Cookie*)(FormData*)(ServerVariables*)(Information)`	48 (11.8%)
Regex45	`(User)(MessageHeader + QueryParam))(Server)(System)`	3 (0.7%)
Regex46	`(User)(Message)(Server)(System)`	1 (0.2%)
Regex51	`(WebBrowser)(CLSID)(Filename)(System)`	2 (0.5%)
Regex52	`(Server)(QueryString)(Command)(System)`	1 (0.2%)
Regex53	`(User)(URL)(Server)(Device)(System)`	1 (0.2%)

Cross-site scripting vulnerabilities, represented by regex13, are commonly exploited on user machines when a victim visits an attacker's web page. The victim unknowingly executes the malicious code on his/her machine and potentially gives the attacker access to private information. These regular expressions show where malicious data can be placed in vulnerable places that are executed.

Four (7.5%) regular expressions are data flow related because each represents an attack that involves submitting excessive data that can consume memory or the CPU (see Table 10).

Table 10: Regular Expressions Representing Excessive Data Exploits.

Regular Expression ID	Regular Expression	Frequency
Regex1	$(\mathrm{User}^{+})(\mathrm{Server}^{+})(\mathrm{Log}^{+})(\mathrm{HardDrive}^{+})$	1 (0.2%)
Regex2	$(\mathrm{User})(\mathrm{Message})(\mathrm{Server})(\mathrm{Header}^{+})(\mathrm{MessageHeaderHandler})(\mathrm{Memory} + \mathrm{CPU})$	1 (0.2%)
Regex12	$(\mathrm{User}^{+})(\mathrm{HTMLPage}^{+})(\mathrm{Server}^{+})(\mathrm{HardDrive}^{+})$	1 (0.2%)
Regex23	$(\mathrm{User}^{+})(\mathrm{Server}^{+})(\mathrm{CPU}^{+})(\mathrm{HardDrive}^{*})$	3 (0.7%)

In general, these attacks are relatively simple compared with the other attacks. Some of these attacks are automated and send a plethora of requests or data to the victim machine. Or, the attack could involve a simple upload of a prodigious amount of data. Since each of these regular expression involved the same events occurring at least once, the superscripted + is used. To ensure the strictest security, software engineers must assume that a large load on the CPU can be normal and that the load may be just short of the maximum capacity of the CPU. Thus, it is possible, but not likely, that only one more user request may put the machine into a denial-of-service.

The four (7.5%) regular expressions in Table 11 represent attacks on machines that allowed a user/application to assume elevated privileges in the system.

Table 11: Regular Expressions Representing Escalated Privilege Attacks.

Regular Expression ID	Regular Expression	Frequency
Regex9	`(User)(SQLInput)(Server)(WebApplication)(Database)(Data)`	19 (4.7%)
Regex43	`(Application)(FileCreation)(System)`	7 (1.7%)
Regex44	`(ApplicationRun)(Privileges)(System)`	2 (0.5%)
Regex50	`(User)(Application)(Subprocess)(System)`	1 (0.2%)

The attacks found in the databases were accomplished by submitting SQL commands to obtain user accounts; by applications spawning child processes with the same permissions as the parent; by applications creating files with elevated permissions; or by systems that give processes system privileges. These regular expressions indicate where access privileges are not enforced on processes or files. Unlike the previous three data flow-oriented classifications, this is the first example where access control is the primary objective of the attacker.

The three (5.7%) regular expressions in Table 12 are attacks that may either cause a denial-of-service or disclosure of sensitive information.

Table 12: Regular Expressions Representing Error Message Attacks.

Regular Expression ID	Regular Expression	Frequency
Regex37	`(User)(Socket)(Server)(ExceptionThrown*)(Server)`	2 (0.5%)
Regex40	`(User)(FTPRequest)(FTPServer)(GetRoutine)(Server)`	2 (0.5%)

Table 12 (continued)

Regex49	`(User)(InvalidRequest)(ErrorMessage)(System)`	2 (0.5%)

In the scenarios represented by these regular expressions, it is possible to cause the system to hang because an exception (error) was thrown in the process. If the process was a server, then the result is a denial of service. In another case, if the server encountered an error and returned the error to an attacker and the server root directory was disclosed with the error, then the attacker has gained important directory structure knowledge about the victim system. Thus, regular expression can warn stakeholders how error messages are harmful to their systems. These regular expressions represent another classification that are not primarily involved with data flow.

The remaining five regular expressions in Table 13 are not immediately related to any of the previous six groups or to each other.

Table 13: Miscellaneous Regular Expressions.

Regular Expression ID	**Regular Expression**	**Frequency**
regex7	`(User)(HTTPServer)(PostMethod)(HTTPContent-LengthHeaderValue)(HTTPMessagePayloadLength)(ServerConnectionState)`	1 (0.2%)
regex10	`(User)(SQLInputField)(Server)(WebApplication)(Database)(CPU)`	1 (0.2%)
regex19	`(Class)(Subclass)(OverriddenSecuredMethods)(Application)`	1 (0.2%)
regex42	`(Application)(DownloadMalicousFile)(PredictableFileLocation)(AttackerReference)(Information)`	2 (0.5%)
regex47	`(SourceFile)(IncludeFile)(EnvironmentVariable + ProgramVariable + URLparam)(System)`	8 (2.0%)

These five regular expressions are therefore grouped together in the miscellaneous category. Regex7 is a scenario where an attacker can specify the payload length of a message as less than the actual size of the payload, which caused a socket to stay open on the server. In this reported vulnerability, a clever attacker falsified the metadata of the information that was sent to the victim machine. Additionally, a query that is CPU-intensive such as a Cartesian join, can cause a denial-of-service, as represented with regex10. This is a case where legal commands should be monitored for their requests on the resources of the system. Regex42 is a case where an attacker knows the location of a downloaded file and takes advantage of this knowledge. Regex47 occurred eight times in the analyzed collection and represents attackers changing system variables in victim applications to point to attacker machines. If the variables include directories of applications, such as PHP, then the attacker can control the PHP program with the malicious source code. Finally, regex19 is a low-level regular expression that illuminates that the overridden methods of parent classes in source code must be secured from the same vulnerabilities as methods in the parent class. For example in Java, one must protect their overridden methods in the CipherInputStream class to ensure the stream remains encrypted otherwise the data may become available to an attacker. From the observations in this group and in the previous two groupings, regular expressions are not limited to data flow, but to a variety of attacks that have been recently exploited applications. Therefore, regular expressions in this study are valid sources of security information that stakeholders can use to protect their systems.

6.0 Feasibility Study

This chapter provides information on the methodology used to create the feasibility study and then discusses the results in the order of the metadata of the study, valid and invalid answers, unique attack paths, regular expression not in the design, and miscellaneous data.

6.1 Feasibility Study Methodology

To test the hypothesis of a regular expression mapping to a design and the ability of a human to perform the approach, a blind feasibility study (see Appendix II) was conducted in an upper-level undergraduate security course at North Carolina State University in Fall 2003. The course was chosen because it is an introductory security course that teaches concepts such as information security management and because the students represent individuals that have at best a limited background in security, which would validate if non-experts could identify attack paths in a system design. Also, the students in the class were not familiar with me as a student, teacher or teaching assistant, which further enabled the study to be blind. The paper-based study was given to the class as an addendum to an already-scheduled assignment. Students were given 14 days to complete the assignment and asked to work on it independently. The assignment was also posted on the course web page for those students who did not attend the lecture in which the assignment was handed out. Questions could be asked on an Internet message board specific to the class or by email.

At the time of the feasibility study, there were 20 regular expressions fabricated from the vulnerability databases and all were included in the assignment. Attack profiles containing a written synopsis were also given to describe the abstract regular expressions. Five regular expressions were not given attack profiles to determine if students needed the profiles to find the attack path. Also included in the assignment was a simple, high-level design consisting of 16 components that represented a hypothetical banking system which was derived from Howard and LeBlank's data flow diagram [11]. The design consisted of four clients and four commonly-known severs to illustrate the remote attacks described by the regular expressions. The design was limited to components that students most likely learned or used in classes because it was not known how students would react to the complexity and size of the assignment. The design was also limited to one page to prevent the thrashing of one page to another while looking for components and attack paths.

Given the design and knowledge base of attacks, students could enact the process of finding security vulnerabilities in the design phase of software application. Students were asked to match the components/events described by the regular expressions to the strings of components in the system design. Upon a match, the string of components were written down using the numbers associated with each component labeled in the design in a "<component number >-<component number>" template. In the simple illustration shown in Figure 5, a `User` can request information to a `Server`, which records accesses and errors in its log files that are stored on the `Hard Drive`.

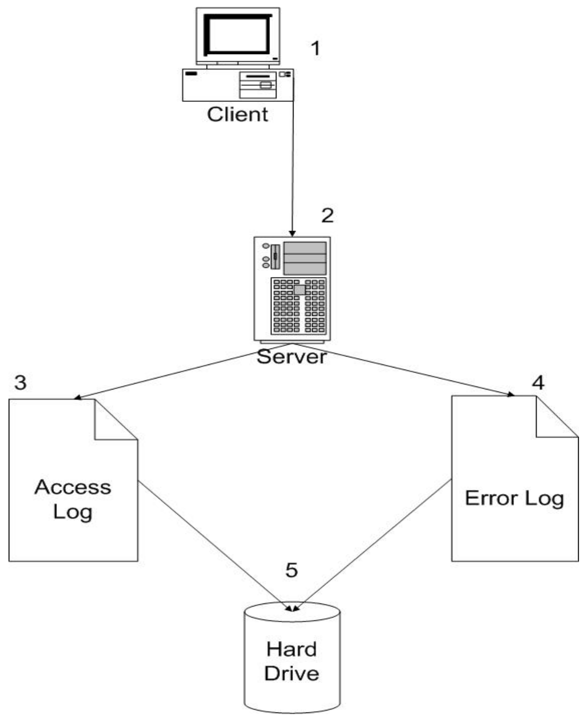


Figure 5: A Simple Design. A simple illustration of a data flow diagram used to show a vulnerable sequence of components.

The regular expression, *(Client$^+$)(Server$^+$)(LogFile$^+$)(HardDrive$^+$)*, has a profile that may look like

```
An attacker can exceedingly access the web server or
database server augmenting the access log file and
eventually filling the hard drive causing the system
to crash, a denial-of-service attack.
```

Using the profile to understand how the attack occurs, students can match the components in the regular expression to the components in the diagram. This regular expression yields two possible answers: 1-2-3-5 and 1-2-4-5. Students were told to match as many attack paths in the design to each regular expression as possible. They were allowed to make any assumptions necessary about how an attack could occur so that the maximum number of attack paths could be found. Two of the regular expressions were purposefully not shown in the design to validate that components in the design cannot be confused with the components in the regular expressions.

6.2 Feasibility Study Results

The intention of the feasibility study was to assess whether students can read a knowledge base of regular expressions and find their instances, if any, in a system design. Students were asked to follow the "<component number>-<component number>" template to make all answers look like the example attack path, "14-13-12-11-6-7," given in the homework assignment. This attack path represents a sequence of six components in the design; `Client 1` (component number 14), `Web Pages` (component number 13), `Web Server` (component number 12), `Service Client Request` (component number 11), `Access Log` (component number 6), `Hard Drive` (component number 7). The student's attack paths were organized based on the resource that was attacked and the path in which the attack occurred. For example, for regex1,

```
(Client+)(Server+)(Log+)(Hard Drive+),
```

students gave a combined total of five different attack paths as shown in Table 14.

Table 14: Student Answers for One Regular Expression. Students found five different attack paths in the system design that were represented by regex1.

Attack Path Number	Valid Attack Path	Number of Student Finds
1	14-13-12-11-6-7	43 (100%)
2	10-9-6-7	43 (100%)
3	1-2-6-7	11 (25.6)
4	1-5-4-9-6-7	1 (2.3%)
5	15-12-12-11-6-7	1 (2.3%)

The different attack paths submitted for each regular expression were arbitrarily numbered for identification purposes. Each student answer for an attack path is termed an "instance" of that attack path. So, since 43 each gave the answer "14-13-12-11-6-7," then there are 43 instances of the same attack path being found.

Students did not describe the same attack path with the same sequence of component numbers. Some students did not include all the components in the path, while others included extra components beyond the regular expression to elaborate how the attack occurred. For example, regex5,

`(Client)(Server)(HTTPMessage)(HeaderFieldBufferWrite)`,

included three different component sequences submitted by students that identified the same attack path. In the component sequence “15-13-12,” one student did not include the `Service Client Request` (component number 11) routine where the HeaderFieldBufferWrite would occur. However, the data flow arrows in the design make it obvious that the attack path would eventually lead to the `Service Client Request` process and so the answer was accepted. In the component sequence “15-13-12-11,” ten students included the `Service Client Request` process to accurately describe where the buffer write exists. One student appended `Hard Drive` to the sequence to obtain “15-13-12-11-7,” which elaborates where the buffer resides in the system. These inconsistencies were tolerated if the attack path was still obvious in showing how the attack occurred.

The attack paths students reported for each regular expression were checked for validity. Valid attack paths were based on the plausibility that the attack could occur along the sequence of components answered by the students. If the components in the design supported the actions necessary to achieve the attack, then the answer was considered valid. For example, in regex2,

`(Client+)(Server+)(HTTPMessageHeaderHandler+)`,

43 (100%) of the class entered “14-13-12-11” where `Client1`, makes a request via the `Web Pages` to the `Web Server`, which is handled by the `Service Client Request` process. The `Web Pages` component is not listed in the regular expression because a web page is not necessary for the vulnerability to be exploited. Students were instructed to include all intermediate components for clarification and thus `Web Pages` is included in their answers.

The attack path, “14-13-12-11,” is valid because it includes components that have been specified from the abstractions in the regular expression and because a client can make a request to a web server containing a large number of headers to cause consumption of the hard drive. At the time of the study, the `HTTPMessageHeaderHandler` was the target resource of an attacker and so terminated the string of components in the regular expression. After the feasibility study was performed, resources such as memory and the CPU were explicitly inserted in the regular expressions for clarity. All valid answers were considered equally viable because each attack path led to a vulnerable resource.

If an attack path seemed unreasonable, but an assumption was supplied that justified the attack, then the answer was accepted. For example, regex20,

`(Class)(Subclass)(OverriddenSecuredMethods)`,

is associated with two answers that would have been marked wrong if assumptions were not given. Both students entered the attack path “1-2-16” and wrote that they assumed the sequence involved a secured socket stream method being overridden. The assumptions are derived from the hint given in the class lecture that described the Java class, `CipherInputStream`, as having methods being overridden without proper security measures. Regex20 is a special case and is discussed later section 6.6.

If an attack was not obvious because the student did not clearly show how the components could achieve the attack profile, then the answer was classified as invalid. In regex2, one student answered “1-5-4-9-6-7-11” and another “1-5-4-9-7-11.” These answers were marked invalid because students did not write any assumptions about how the attack would occur. These paths do not explain why an HTTP message travels from `Client4` to the `Service Client Request` process. If the students had assumed that the `Application Server` (component 4) was a web application server, then the component sequences would be correct. The same applies to seven instances of “10-9-8-11” and one instance of “1-2-6-7-11.” None of the answers included assumptions about the `Authentication Server` or `Database Server` having the capability of handling an HTTP message. Also, answers that had only one component such as “15” were marked invalid because none of the attacks involve only one component.

The findings in this study are presented in the following order: metadata of the class, valid and invalid answers, unique paths, student answers to regular expressions that were not existent in the design, and miscellaneous data. The regular expressions discussed in these results refer to the regular expression in the KB for the assignment. Note that the regular expressions in the feasibility and validation study are numbered differently and are also numbered differently than the final knowledge base in Section 5.0. The regular expression knowledge base (for the feasibility study) is in the assignment in Table 1 of Appendix II.

6.3 Metadata

Approximately one-third of the 62 students registered for the course attended class the day the assignment was discussed in class. Therefore, approximately two-thirds of the class were not familiarized with the assignment background and instructions in-advance of partaking the assignment. Furthermore, only 43 of the 62 students completed the assignment. To ensure a blind study, a hardcopy of the assignment was distributed with unique IDs that replaced student names. Fortunately, students that did not receive the hardcopies entered arbitrary numbers on the assignment they downloaded from the course web site, which permitted the study to be performed blindly. Students were asked to record how much time they spent on the assignment; on average, they spent 53.7 minutes on the assignment based on the input of 41 of the 43 students that gave valid times. The two times entered they were not included were textual responses of “obviously, very little” and “too much." The range of time to complete the assignment was 15 minutes to 180 minutes (see Appendix III).

6.4 Valid and Invalid Answers

The 43 students responded with a sum of 937 valid answers that were grouped into 75 different valid attack paths. Students also entered 65 instances of where an abstract attack path was represented in the design and were organized into 36 invalid attack paths (see Appendix IV). These results represent an approximate 2:1 ratio of valid to invalid attack paths and approximately a 14:1 ratio of valid to invalid instances found. These results indicate that students were able to instantiate the abstract regular expressions into specific attack paths present in the system design.

One hundred percent of the participating students found the same four valid attack path for three regular expressions represented in the design (see Table 15).

Table 15: Attack Paths Found by 100% of the Participating Students. One hundred percent of the participating students found four attack paths represented by three regular expressions.

Regular Expression ID	Regular Expression	Valid Attack Path
regex1	`(Client`$^+$`)(Server`$^+$`)(Log`$^+$`)(Hard Drive`$^+$`)`	14-13-12-11-6-7
		10-9-6-7
regex2	`(Client`$^+$`)(Server`$^+$`)(HTTPMessageHeaderHandler`$^+$`)`	14-13-12-11
regex12	`(User)(CommandLineArgumentEntry)(Application)(ApplicationServer*)(CommandLineArgumentBufferWrite)`	1-5-4

The perfect agreement between classmates may have resulted from an easier correlation between the components in the system design and the components in the regular expression. Or, students may have more experience in their classes with these components and were better able to find them in the system design. The regular expressions do not have similar profiles thus they do not require similar/obvious attack paths.

An analysis to determine the number of valid attack paths found per regular expression was performed and is represented in Figure 6.

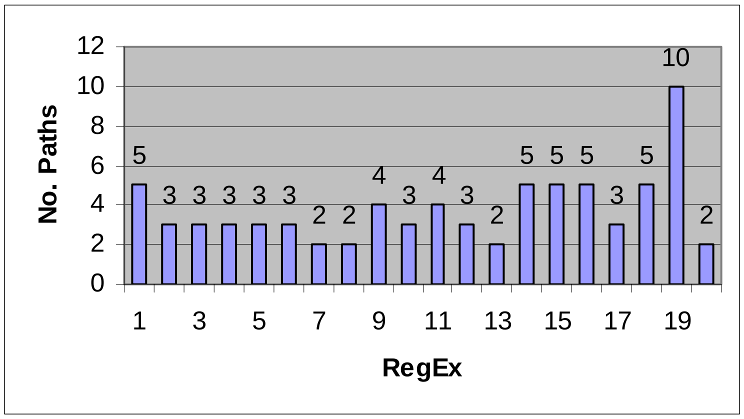


Figure 6: Number of Valid Attack Paths per Regular Expression.

The average number of valid attack paths per regular expression is 3.8. The regular expression with the highest number of valid attack paths is regex19,

`(SocketRead)(SocketBufferWrite)`,

with ten (see Table 16).

Table 16: Regular Expressions Further Specified. Regex19 is associated with the largest number of valid attack paths per regular expression.

Valid Attack Path	Number of Student Responses
14-13-12	9 (20.9%)
1-5-4	4 (9.3%)
13-12-11-2	1 (2.3%)
15-13-12-11	2 (4.7%)
10-9	3 (7.0%)
9-12	1 (2.3%)
9-6	1 (2.3%)
1-2-4	2 (4.7%)
12-11-8	1 (2.3%)
7-8	1 (2.3%)

Regex19 is considerably more abstract than most of the other regular expressions. It can apply to any socket connection and thus the students found many vulnerabilities in the design. Another factor that may have contributed to the large number of different attack paths is that there was no attack profile given with this regular expression. Students may not have understood the regular expression and were more creative with their answers.

The regular expressions with the lowest number of valid attack paths found are regex7, regex8, regex13, and regex20 with only two paths each (see Table 17).

Table 17: Regular Expressions with Least Number of Valid Attack Paths. The four regular expressions with the least number of valid attack paths are regex7, regex8, regex13, regex20.

Regular Expression ID	Regular Expression	Valid Attack Path
regex7	(Client)(Server)(PostMethod)(HTTPContent)(LengthHeaderValue)(HTTPMessagePayloadLength)(ServerConnectionState)	14-13-12-11
		15-13-12-11
regex8	(User)(UserNameEntry)(PasswordEntry)(AuthenticationServer*)(AuthenticationRoutine)	1-2-16
		10-9-4-5-1-2-16
regex13	(Client)(HTMLPage)(Server)(Hard Drive)	14-13-12-11-7
		15-13-12-11-7
regex20	(Class)(Subclass)(OverrridenSecuredMethods)	10-9-4-5
		1-2-16

This is due to the small number of vulnerabilities in the design that correlate to these attacks. Regex20 was not in the design and was expected to have zero valid attack paths. However, there were two attack paths given with valid assumptions and thus they were classified as valid attack paths. Also, Regex20 did not have a corresponding attack profile; consequently students may not have understood this expression well enough to realize it was not in the design.

Figure 7 shows the number of invalid attack paths that students answered for the regular expressions.

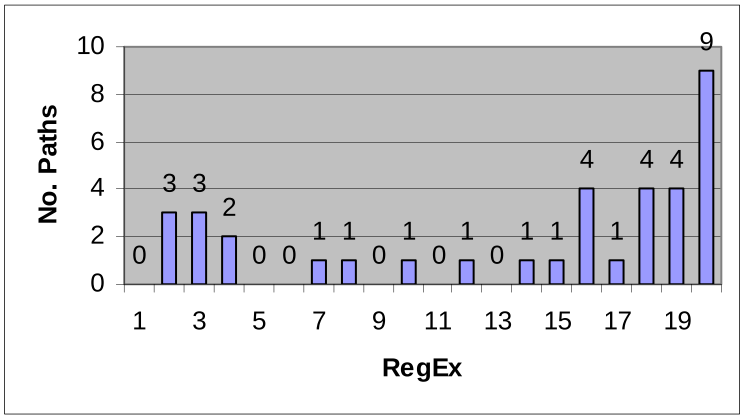


Figure 7: Number of Invalid Attack Paths per Regular Expression.

The average number of invalid attack paths per regular expression is 1.8. The regular expressions with the least number of attack paths, zero, are regex1, regex5, regex6, regex9, regex11, and regex13 (see Table 18).

Table 18: Regular Expressions without Invalid Attack Paths. Six regular expressions were associated with zero invalid attack paths.

Regular Expression ID	Regular Expression
Regex1	`(Client+)(Server+)(Log+)(Hard Drive+)`
Regex5	`(Client)(Server)(HTTPMessage)(HeaderFieldBufferWrite)`
Regex6	`(Client)(Server)(HTTPMessageHandler)(Log)(Sysadmin)(LogEntryRead)`
Regex9	`(Client)(SQLInputField)(Server)(WebApp)(Database)`
Regex11	`(Client)(HTMLForm)(WebApp)(Server)(cgihtml)(FileSystem)`
Regex13	`(Client)(HTMLPage)(Server)(Hard Drive)`

Thus, students did not identify any invalid paths for six (30%) of the twenty regular expressions. The regular expression with the most invalid attacks is regex20 with nine. Regex20, also having the least number of valid answers, is a special case and is discussed in detail in Section 6.6.

The largest number of invalid attacks was entered for regex2; three paths were found with a total of ten instances, as shown in Table 19.

Table 19: Regular Expression with the Most Number of Invalid Attack Paths. Regex2 had the most number of invalid attack paths with three.

Regular Expression	Invalid Attack Path	Number of Student Finds
`(Client`$^+$`)(Server`$^+$`)(HTTPMessageHeaderHandler`$^+$`)`	10-9-8-11	7 (16.3%)
	1-5-9-6-11	2 (4.7%)
	1-2-6-7-11	1 (2.3%)

The invalid attack paths were considered incorrect because the paths are unlikely using HTTP headers between the client and server. None of the students clearly indicated how an HTTP header could be sent to any of the authentication, application or database servers. If the application server was a web application server, then the attack could be considered valid, but there were no assumptions made. These results indicate that designs may not offer enough detail to determine if an attack may occur, thus assumptions are important to determining security attacks. The best answers used the web server component (component number 13), which clearly requires HTTP headers. Regex20 had nine invalid attack paths with 24 instances, but this is considered a special case and is discussed in Section 6.6.

The differences in the number of instances of each attack path per regular expression depends on how well the regular expression and attack profile describe the attack; the accuracy of the abstraction of the regular expression; the ability of students to make assumptions about components; and their desire to find attack paths that snake their way through the design. The attack paths with the most instances imply that they are the most obvious in the design. Attack paths with fewer instances suggest that more assumptions must be made about how the attack may occur and that the path may be more complex. For example, regex14,

`(Client)(HTMLMessageBoard)(Server)(HTMLMessageBoard)(Client)`,

produced one attack path that was “14-13-12-13-14.” Twenty-six students found this attack path and 12 more found “14-13-12-13-15”. The components in the attack paths correlate one-to-one with the components in the regular expression. One creative student found "15-13-12-11-8-9-4-5-1" as a sneaky attack path that traversed most of the components in the design. Extra components in the attack path suggest how subtle security attacks may be in a system. This valid attack path was not discovered by any other student and was not intentionally put into the design. This unexpected valid path indicates that regular expressions are flexible enough encapsulate many attack paths that threaten the system.

6.5 Unique Attack Paths

An analysis was done to determine how many attack paths had only one instance. Twelve (16.0%) of the valid attack paths found were found only once as shown in Table 20.

Table 20: Regular Expressions with Valid Unique Attack Paths. Students found 12 valid unique attack paths for seven regular expressions.

Regular Expression ID	Regular Expression	Valid Unique Attack Path
Regex1	`(Client+)(Server+)(Log+)(Hard Drive+)`	1-5-4-9-6-7
		15-13-12-11-6-7
Regex8	`(User)(UserNameEntry)(PasswordEntry)(AuthenticationServer*)(AuthenticationRoutine)`	10-9-4-5-1-2-16
Regex11	`(Client)(HTMLForm)(WebApp)(Server)(cgihtml)(FileSystem)`	1-5-4-9-8
Regex14	`(Client)(HTMLMessageBoard)(Server)(HTMLMessageBoard)(Client)`	15-13-12-11-8-9-4-5-1
Regex18	`(User)(File)(FileRead)(BufferWrite)`	3-6
Regex19	`(SocketRead)(SocketBufferWrite)`	9-12
		9-6
		12-11-8
		7-8
		13-12-11-12
Regex20	`(Class)(Subclass)(OverriddenSecuredMethods)`	10-9-4-5

Regex19 may have the highest number of unique paths because it is more generalized than the other regular expressions, and it can appear almost anywhere in the design. Also, there was no profile given for this regular expression because it was used as a determining factor on whether or not to provide a profile for the regular expressions in the validation study. Students may have been confused on what to do without a profile and thus been more creative in their answers. The low percentage of valid unique attack paths suggests that students usually find the same valid attack paths.

The number of unique answers is higher for invalid answers. Twenty six (72.2%) of the invalid attack paths were only found once and are show in Table 21.

Table 21: Regular Expressions with Invalid Unique Attack Paths. Twenty-six invalid unique attacks associated with eight regular expressions were submitted by students.

Regular Expression ID	Regular Expression	Invalid Unique Attack Path
Regex2	`(Client+)(Server+)(HTTPMessageHeaderHandler+)`	1-2-6-7-11
Regex3	`(Client)(Server)(GetMethod)(GetMethodBufferWrite)`	1-2-16-2-1-5
		1-2-16
Regex4	`(Client)(Server)((GetMethod)+(PostMethod))(PayloadValueBufferWrite)(WebApp)`	10-9-4-5
		1-5-4-9-8
Regex8	`(User)(UserNameEntry)(PasswordEntry)(AuthenticationServer*)(AuthenticationRoutine)`	4-2-16
Regex10	`(Client)(SQLInputField)(Server)(WebApp)(Database)`	11
Regex12	`(User)(CommandLineArgumentEntry)(Application)(ApplicationServer*)(CommandLineArgumentBufferWrite)`	14-13-12-11-7
Regex14	`(Client)(HTMLMessageBoard)(Server)(HTMLMessageBoard)(Client)`	14-13-12-11-8
Regex16	`(User)(ReadUserInput)(EnvironmentVariableWrite)`	10-9-6
		1-2-1-2
		10
Regex17	`(User)(GUI/Browser)(BookMarkSave)(BookmarkBufferWrite)`	10-9-8
Regex18	`(User)(File)(FileRead)(BufferWrite)`	1
		14
		10
		15
Regex19	`(SocketRead)(SocketBufferWrite)`	12-11-2
		6-7
		1

Table 21 (continued)

Regex20	`(Class)(Subclass)(OverriddenSecuredMethods)`	15
		14
		5
		8
		2-4-6-9
		10-9

Regex20 was not in the design, but many students believed it existed and so arrived at vague and creative answers. As with regex19, a profile was not given with regex20 possibly causing students to guess on where such a vulnerability may exist. The guesses likely caused a larger number of unique answers to be produced. The number of invalid unique attack paths is approximately twice that of the valid unique attack paths. The high percentage of invalid unique attack paths suggest that regular expressions are clear enough to students so that many students will not find the same wrong answer. Also, the percentages of valid and invalid unique answers represent that students are more likely to find invalid unique attacks than valid unique attacks.

6.6 Regular Expressions Not Represented in the Design

Two regular expressions, regex17 and regex20,

`(User)(GUI/Browser)(BookMarkSave)(BookmarkBufferWrite)` and

`(Class)(Subclass)(OverriddenSecuredMethods)`,

were purposefully inserted into the knowledge base that were not represented in the system design to determine if students accurately found components in the design that matched the components in the regular expression. Any answer that was left blank or was indicated explicitly as not being in the design was recorded. Two (4.7%) students indicated that regex17 was not in the design and 14 (32.6%) students indicated that regex20 was not in the design. Regex20 was associated with the most students who claimed that an instance of the regular expression was not present in the design. Furthermore, students also claimed that some of the regular expressions were not represented in the design when in actuality they were (see Figure 8).

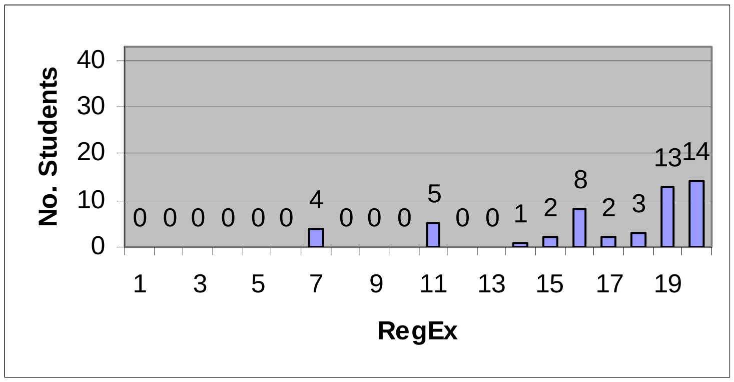


Figure 8: Number Students Indicating that the Regular Expression was not in the Design.

These data show that seven or 38.9% of the 18 of the regular expressions that were intended to be found by students were incorrectly marked as not present in the design (see Table 22).

Table 22: Regular Expressions not Found in the Design. Seven regular expressions were incorrectly identified as not being in the design by students.

Regular Expression ID	Regular Expression	Number Students Indicating not in Design
Regex7	`(Client)(Server)(PostMethod)(HTTPContent)(LengthHeaderValue)(HTTPMessagePayloadLength)(ServerConnectionState)`	4 (9.3%)
Regex11	`Client)(HTMLForm)(WebApp)(Server)(cgihtml)(FileSystem)`	5 (11.6%)
Regex14	`(Client)(HTMLMessageBoard)(Server)(HTMLMessageBoard)(Client)`	1 (2.3%)
Regex15	`(User)(Machine)(SyslogFunction)(Log)`	2 (34.9%)
Regex16	`(User)(ReadUserInput)(EnvironmentVariableWrite)`	8 (37.2%)
Regex18	`(User)(File)(FileRead)(BufferWrite)`	3 (7.0%)

Table 22 (continued)

Regex19	`(SocketRead)` `(SocketBufferWrite)`	13 (44.1%)

The average number of students incorrectly determining that the regular expressions were not in the design is 5.1. Thus, eleven (61.1%) of the regular expressions that were actually in the design were always correctly labeled as existing in the design.

The code-level regular expression,

`(Class)(Subclass)(OverriddenSecuredMethods)`,

denoted by regex20 was purposefully not shown in the high-level design. This vulnerability had the largest instance of students, 14 (32.6%), responding as not existing in the design. However, one student found the attack path "10-9-4-5" and assumed it was possible if a secured connection existed. Two students found the attack path "1-2-16" by making assumptions about a secured socket stream present in path. These three students either investigated the hint supplied with the regular expression that discussed the vulnerability occurring in a Netscape Java implementation, or they listened to the explanation of the regular expression in the lecture when the assignment was given. The valid answers were not expected because the components were not explicitly shown in the design. There were 24 instances of invalid answers grouped into nine different attack paths (see Table 23).

Table 23: Invalid Attack Paths for Regex20. Nine invalid attack paths were submitted by students for regex20, which did not exist in the design.

Invalid Attack Path	**Number of Student Responses**
1-5-4	7 (16.3%)
15	1 (2.3%)
14	1 (2.3%)
14-13-12-11	8 (18.6%)
14-13-15	3 (7.0%)
5	1 (2.3%)
8	1 (2.3%)
2-4-6-9	1 (2.3%)
10-9	1 (2.3%)

Four of the attack paths had only one component involved suggesting that students thought the attack would be initiated from the component itself. The other 20 paths were routes between clients and servers, but did not have any assumptions about how they may have had secured connections. For example, the path "1-5-4" (`Client4-Software Application-Application Server`) does not suggest that a secured method is overridden. As mentioned in Section 6.4, there were more invalid answers and invalid unique attack paths than with other regular expressions. It is concluded that students were eager to find attack paths that could possibly exist for an attack. Security attacks can often be subtle and the extra effort shows that regular expressions may entice students to find hidden vulnerabilities.

The regular expression

`(User)(GUI/Browser)(BookMarkSave)(BookmarkBufferWrite)`,

represented by regex17 was not explicitly shown in the design. However, 41 (95.3%) students responded with three attack paths (totaling 46 instances) that were evaluated as valid because GUIs and Internet browsers can be implied on the client components that connect to the servers (see Table 24).

Table 24: Valid Attack Paths for Regex17. Three attack paths were found for regex17, which did not exist in the design.

Valid Attack Path	Number of Student Responses
14-13-14	28 (65.1%)
15-13-15	10 (23.3%)
1-5-4	8 (18.6%)

This represents that students were able to look at components in a system design and creatively suggest how such an attack may occur even though it does not directly correspond with the regular expression. One unique answer, "10-9-8," attempts to show a bookmark being saved via a database server was marked as the only invalid answer because it is not clear how such an attack would occur. Two students (4.7%) said the regular expression was not represented in the design. No valid unique attacks paths were submitted for regex17.

The regular expression,

```
    (SocketRead)(SocketBufferWrite),
```

represented by regex19 existed in the design, but was not explicitly shown. Unlike regex17, sockets are an implied component in design, and thus the states in the regular expression can be assumed to occur. Thirteen (30.2%) students indicated that this attack was not possible in the design and thus was the regular expression that was represented in the design with the highest instance of “not in design” answers. Hence, although 95.3% of the class could determine that a bookmark save may exist in the system, 30.2% of the students did not find the buffer overflow occurring at a socket. The unexpectedly high number of students indicating that regex17 was in the design may have occurred because saving a bookmark is an action that could occur on the client, which in the directions was described as possibly being an Internet browser. The socket, a physical component, could have been something students wanted to see to make the attack seem possible. The high number of instances could also be due to the absence of the attack profile in the knowledge base.

6.7 Miscellaneous Data

Forty-one students responded to the question on the assignment that asked to rate the approach (see Appendix V). The possible choices were (1) Poor (2) Below Average (3) Average (4) Good and (5) Excellent. The rating that most students (36.6%) agreed upon was Good (4). Three students gave the approach a 1 and three gave the approach a 5. The Likert scale ratings chosen by the students suggest that most students thought the approach was a good approach to detecting security vulnerabilities in a system design via regular expressions. Students were also asked to evaluate how helpful the attack profiles were in determining if an attack was present. Forty-one students responded to the question (two students left the answer blank) and 23 (56.1%) responded with “yes” indicating that the attack profiles were useful. Therefore, attack profiles were considered to be helpful. As a result, attack profiles were used for each regular expression in the validation study, as will be discussed in Chapter7.

Student comments that were requested to support their rating of the approach were also collected in the assignment and are grouped together based on similarity. Thirty students gave comments and 11(35.5%) responded with complaints about not knowing enough about the functions of the components and how the system worked. This information was used to recognize that an important assumption must be made about this approach. One must assume that if the component sequences are present in the design, then the security attack is possible even in the absence of detailed information about the system and its components. Not all stakeholders will know the characteristics of components in the system, and thus some attack paths will be based on blind knowledge of the subject matter. Therefore, when one attempts to find an attack path, he or she must consider the components to be “black boxes” and assume that only the sequence of events is necessary for the attack to occur.

The final piece of data for this assignment was the number of questions about the assignment after the initial lecture. Only one question was asked and was asked on the student message board; the only question asked was “[f]or column 2, do we need to include all the component sequences we can find for a regular expression or will one possible path be fine?” The question was answered with “find as many attack paths as possible.”

7.0 Validation Study

This chapter provides information on the methodology used to create the validation study and then discusses the results in the order of the metadata of the study, valid and invalid answers, unique attack paths, regular expression not in the design, and miscellaneous data.

7.1 Validation Study Methodology

The validation study (see Appendix XII) was an advanced version of the feasibility study, done in a similar fashion. Validation was performed in the next semester (Spring 2004) in the same undergraduate security course with a different class of students. The assignment was done completely on the Internet using WebAssign (http://webassign.com) to host the questions and assignment description and a personal web page to display the system design. Students electronically submitted their answers once the 14 day deadline was reached.

During the time of the feasibility study, time records were used to track the amount of time required for the following phases of the study: vulnerability collection and analysis, assignment creation, collection and analysis of student answers. These times were used to scope the validation study and set a predetermined number of vulnerabilities to analyze and the number of regular expressions to subject to the study. The conclusions of the time records indicated that a subset consisting of 30 regular expressions should be used in the assignment. This value also reflected student patience and motivation that was observed in the comments of the study. Students seemed to become wary of looking for multiple paths for 20 regular expressions in a system design. The 30 regular expressions were selected on how well they fit a high-level design. Most of the regular expression abstracted from the databases could be represented with high-level system designs, and thus there were not enough regular expressions to apply to a possible second design that could show the other types of low-level vulnerabilities. Of the thirty regular expressions that were used, 16 were the same as the feasibility study. The regular expressions that were not in the design from the feasibility study were again tested under the same conditions. Also adding to the complexity of the assignment was an advanced version of the design from the feasibility assignment. The design included 14 more components to explicitly show processes, buffers, firewalls, routers, and to test the student's ability to find attack paths that are obscured with added complexity in the design. The instructions to the students were the same for the validation study. However, 11 (25.6%) students that partook in the feasibility study complained about n knowing the role of the components in the design. Thus, the students in the validation study were told to black box the components and assume that the attack was achievable if the components existed.

7.2 Validation Study Results

The intention of the validation study was to assess whether students can read a knowledge base of regular expressions and find their instances, if any, in a system design. Students were asked to follow the "<component number>-<component number>" template to make all answers look like the example attack path, "26-25-24-23-19-11-12," given in the homework assignment. This attack path represents a sequence of seven components in the design; `Client` (component number 26), `Web Pages` (component number 25), `Proxy Server + Firewall` (component number 24), `HTTP/FTP Server` (component number 23), `HTTP Message & Header Handler` (component number 19), `Access Log` (component number 11), `Hard Drive` (component number 12). However, some students instead replaced the hyphen with either spaces or commas to look like "16 15 14 12" or "26,25,24,23,19,21". The student's attack paths were organized based on the resource that was attacked and the path in which the attack occurred (see Appendix VIII). For example, for regex1,

$$\texttt{(Client}^{+}\texttt{)(Server}^{+}\texttt{)(Log}^{+}\texttt{)(Hard Drive}^{+}\texttt{)},$$

students gave a combined total of seven different attack paths as shown in Table 25.

Table 25: Student Answers for One Regular Expression. Students found seven different attack paths in the system design that were represented by regex1.

Attack Path Number	**Valid Attack Path**	**Number of Student Finds**
1	1-2-3-11-12	30 (52.6%)
2	16-15-14-11-12	37 (64.9%)
3	17-18-25-24-23-19-11-12	9 (15.8%)
4	17-18-25-24-27-28-21-19-11-12	1 (1.8%)
5	1-7-8-9-10-14-11-12	4 (7.0%)
6	26-25-24-23-19-11-12	29 (50.9%)
7	26-25-24-27-28-21-19-11-12	4 (7.0%)

The different attack paths submitted for each regular expression were arbitrarily assigned an Attack Path Number for identification purposes. Each student answer for an attack path is termed an "instance" of that attack path. So, since 30 students each gave the answer "1-2-3-11-12," then there are 30 instances of the same attack path being found.

Students did not describe the same attack path with the same sequence of component numbers. Some students did not include all the components in the path, while others included extra components beyond the regular expression to elaborate how the attack occurred. For example, regex1 included two different component sequences submitted by students that identified the same attack path. In the component sequence "26-23-11-12," one student did not include the intermediate components Web Pages (component number 25), Proxy Server + Firewall (component number 24), or `HTTP Message and Header Handler` (component number 19). However, the data flow arrows in the design make the excluded components obvious parts of the attack path. The remaining 44 students that found this attack path included all the components to look like “26-25-24-23-19-11-12.” These inconsistencies were tolerated if the attack path was still obvious in showing how the attack occurred.

The attack paths students reported for each regular expression were checked for validity. Valid attack paths were based on the plausibility that the attack could occur along the sequence of components answered by the students. If the components in the design supported the actions necessary to achieve the attack, then the answer was considered valid. For example, in regex1, 45 (78.9%) of the class entered “26-25-24-23-19-11-12” a `Client1`, makes a request via the `Web Pages, which goes through the` `Proxy Server + Firewall` to the `HTTP/FTP Server`, followed by processing in the `HTTP Message & Header Handler`, which enters the request to the `Access Log` and is saved on the `Hard Drive`. The `Web Pages, Proxy Server + Firewall and HTTP Message & Header Handler` components are not listed in the regular expression because they are not necessary for the vulnerability to be exploited. Students were instructed to include all intermediate components for clarification and thus `Web Pages` is included in their answers. The attack path, “26-25-24-23-19-11-12,” is valid because it includes components that have been specified from the abstractions in the regular expression, and because a client can make an excessive number of requests to a web server to cause consumption of the hard drive. All valid answers were considered equally viable because each attack path led to a vulnerable resource.

If an attack path seemed unreasonable, but an assumption was supplied that justified the attack, then the answer was accepted. For example, regex29,

`(Class)(Subclass)(OverriddenSecuredMethods)`,

is associated with 13 answers that would have been marked wrong if assumptions were not given. Each student assumed that the system was built in Java and that secured methods were overridden with methods that did not implement security measures. Regex29 is a special case and is discussed later Section 6.6.

If an attack was not obvious because the student did not clearly show how the components could achieve the attack profile, then the answer was classified as invalid. In regex15,

```
(User)(ReadUserInput)(EnvironmentVariableWrite)(Buffer),
```

four students answered “1-2-3-4-30,” two entered “1-2-3-4-30-4-3-1,” and one entered “1-3-4-30.” Each of the three paths are organized in the same group because they are the same attack path with varying degrees of included components. These answers were marked invalid because students did not write any assumptions about how the attack would occur. These paths do not explain why an Authentication Server would accept a username and password for environment variables. Also, answers that had only one component such as “3” were marked invalid because none of the attacks involve only one component.

The findings in this study are presented in the following order: metadata of the class, valid and invalid answers, unique paths, student answers to regular expressions that were not existent in the design, and miscellaneous data. The regular expressions discussed in these results refer to the regular expression in the knowledge base for the assignment. Note that the regular expressions are numbered differently and are also numbered differently than the final knowledge base in Section 5.0.

7.3 Metadata

There were 70 students in the class, but only 58 completed the assignment. One student’s answers were eliminated because the student only answered one of the 30 regular expression answers. Unlike the class in the feasibility study, there was approximately 100% attendance to the lecture in which the assignment was presented. Students were asked to record the amount of time they spent on the assignment. On average, students spent 114.2

minutes on the assignment, which was based on 55 students that gave valid times. The range of time to complete the assignment was 20 to 480 minutes (see Appendix VII). The average time difference between the feasibility and validation studies was 60.5 minutes. The increase time was due to the ten added regular expressions and 14 new components in the system design.

7.4 Valid and Invalid Answers

The students responded with a sum of 2067 valid answers that were grouped into 137 different valid attack paths. Students also entered 155 instances of where an abstract attack path was represented in the design and were organized into 45 invalid attack paths. These results represent an approximately 3:1 ratio of valid to invalid attack paths and a 13:1 ratio of valid to invalid instances found. The large difference between the number of valid/invalid attack paths and number of instances suggests that students are more likely to discover correct answers than incorrect answers. These differences are proportionally similar to that of the feasibility and thus provide further support that student can find a valid attack path more often than invalid attack paths. If there were large quantities of students finding wrong answers, then this would indicate the approach is not viable for finding security vulnerabilities. However, the large numbers indicate that a security engineer will face the challenge of analyzing a large number of attack paths and approximately 30% of them may be false positives. These data also show that regardless of attack path validity, students can effectively instantiate abstract regular expression into specific attack paths in the system design. Lastly, these data suggest that there is more potential for attacks as systems increase in complexity, and that students are more likely to find more attacks given a larger knowledge base of attack profiles.

The highest frequency of instances for a valid attack path reported by students was 55 (96.5%), which occurred for regex11,

```
(User)(CommandLineArgumentEntry)(Application)(ApplicationServer*)
                (CommandLineArgumentBufferWrite)
```

with the sequence “1-7-8-9-10-9-8-6”. Regex11 in the validation study is the same as regex12 in the feasibility study, which had 100% of the class finding the same attack path. In both studies, the students found the same path albeit the path in the validation study is technologically more advanced. Also in both studies, there were two alternative attack paths that students found, but these paths had less than four instances each. The scenario that the attack profile describes is probably more familiar to the students than other attacks, making it easier to distinguish among the many components in the design. Also, the components in the regular expression are not explicitly shown in the design (as with the feasibility study), representing that students can instantiate an attack path from an abstract description.

An analysis to determine the number of valid attack paths found per regular expression was performed and is represented in Figure 9.

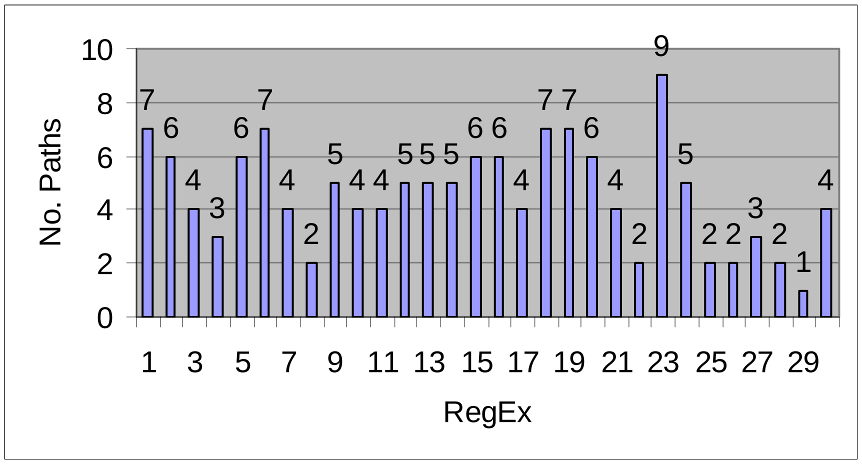


Figure 9: Number of Valid Attack Paths per Regular Expression.

The average number of attack paths per regular expression is 4.6. The regular expression with the most number of valid attack paths is regex23, `(Client)(SearchString)(Server)(Data)(Client)`,

with nine, which describes clients requesting information from servers (see Table 26).

Table 26: Regular Expression with the Most Number of Valid Attack Paths.

Valid Attack Path	Number of Student Responses
26-25-24-23-24-25-26	20 (35.1%)
16-31-15-14-15-31-16	18 (31.6%)
17-18-25-24-23-19-22-21-20-19-23-24-25-26	7 (12.3%)
17-18-25-24-23-24-25-18-17	7 (12.3%)
1-7-8-9-10-14-10-9-8-6-8-7-1	4 (7.0%)
1-7-8-9-10-14-15-31-16	3 (5.3%)
26-25-24-23-23-24-25-18-17	1 (1.8%)
16-31-15-14-10-9-8-7-1	1 (1.8%)
26-25-24-23-19-13-14-15-31-16	1 (1.8%)

This is the case because there are many combinations of clients potentially polling servers in the design.

The regular expression with the least number of valid attack paths is regex29,

`(Class)(Subclass)(OverriddenSecuredMethods)`.

Thirteen students assumed that the vulnerability may occur anywhere in the system that is implemented with Java and overrode secured methods. Regex29 was represented as regex20 in the feasibility study, which also had the fewest number of valid attack paths. Regex29 is a special condition where it was purposefully not put in the design. This is discussed further in Section 7.6.

Figure 10 shows the frequency of invalid attack paths that were produced for each regular expression.

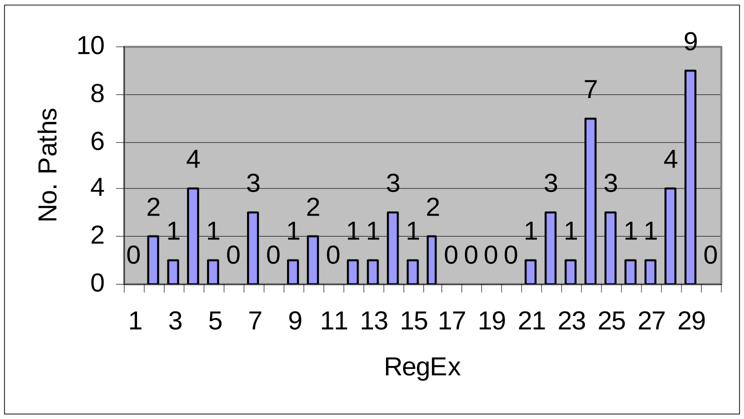


Figure 10: Number of Invalid Attack Paths per Regular Expression.

The average number of invalid attack paths per regular expression is 1.7. The regular expression with the largest number of invalid attack paths is regex29 with nine. Regex29 is identical to regex20 in the feasibility study, and continues the pattern of it being the regular expression having the highest invalid yields and lowest valid yields. This will be discussed in Section 7.6. The regular expressions with the least invalid attack paths, zero instances found, are shown in Table 27.

Table 27: Regular Expressions without Invalid Attack Paths. Six regular expressions had zero invalid attack paths.

Regular Expression ID	Regular Expression
Regex1	`(Client`$^+$`)(Server`$^+$`)(Log`$^+$`)(Hard Drive`$^+$`)`
Regex6	`(Client)(HTTPServer)(HTTPMessageHandler)(Log)(Sysadmin)(LogEntryRead)`
Regex8	`(User)(UserNameEntry)(PasswordEntry)(Server)(AuthenticationRoutine)`
Regex11	`(User)(CommandLineArgumentEntry)(Application)(ApplicationServer*)(CommandLineArgumentBufferWrite)`
Regex17	`(Client)(Hyperlink)(Server)`
Regex18	`(Client`$^+$`)(Server`$^+$`)(MessageHeaderHandler`$^+$`)`
Regex19	`(Client)(Server)(DaemonProcess)(Hard Drive)`
Regex20	`(UserInput)(IntegerEvaluationRoutine)`
Regex30	`(Client)(Application)(EnvironmentVariable + ProgramVariable + URLParam)(MaliciousIncludeFile)`

Thus, students did not submit any invalid answers for nine (30%) of the regular expressions. This is the same percentage of regular expressions not producing any invalid answers in the feasibility study. Only three regular expressions (regex1, regex6, regex9) are the same between the two experiments that had zero invalid answers. Therefore, with the appropriate degree of abstraction, regular expressions appear to be used to identify multiple vulnerable attack paths in system designs.

Regex28,

```
        (Client)((FTPCommand + MailCommand) + OSCommand)
                (FTPServer + Mailserver))(Buffer),
```

had the most instances of invalid attacks, 29, which were grouped into four invalid attack paths (see Table 28).

Table 28: Regular Expression with the Most Number of Invalid Attack Paths. Regex28 had 29 invalid attacks paths grouped into four attack paths.

Invalid Attack Path	Number of Student Responses
26-25-24-27-28-21	24 (42.1%)
1-7-8-9-10-14-11-3-4-30	3 (5.3%)
16 15 14 12	1 (1.8%)
16-31-15-14-13-19-21-29-23-24-27-28-21	1 (1.8%)

Twenty-four of these instances involved component number 27 (email server) instead of component number 23 (the FTP server). The FTP and email servers are located closely together and may have thus been confusing to students who gave a cursory glance to the design. This is especially likely for those students who performed the assignment online where viewing the design was more difficult than it was on paper. The remaining instances were infeasible unless assumptions were provided to clarify how an attack would occur along the specified components.

The average numbers of valid attack paths in both the feasibility and validation studies (3.8 and 4.6) are larger than the average numbers of invalid attack paths (1.9 and 1.7).

These findings suggest that students can instantiate abstract regular expressions into concrete attack paths in a system design. These data also represent that one regular expression can effectively encode multiple attack paths in a system design.

Several of the invalid answers are due to the complexity of the design. Some students thought that if they simply found a path of components that could be connected by arrows and started and ended with what was described in the regular expression, then the path was a valid attack path. This error is also due to the lack of experience and knowledge of the system. This was typically found when the attack paths involved the process required with server requests. For example, "26-25-24-27-28-21-20-19-11-12" shows that a client uses an email server, followed by a call to the GET request routine, and then to the `HTTP Message and Header Handler`. This sequence of events is not valid, but the student considered this because the attack mapped to the regular expression with extra intermediate components. The GET, POST and message header handler were all handled by one component in the feasibility study and thus the misconception did not occur with the simpler design. Therefore a flaw in approach proposed in this thesis is that the stakeholders must have a good understanding of how components interact to know how data flows and where attacks may occur.

7.5 Unique Attack Paths

In the validation study, 17 (12.4%) of the valid attacks paths had only one instance. That is, there was only one instance of an attack path for a particular regular expression (see Table 29).

Table 29: Regular Expressions with Valid Unique Attack Paths. Students found 17 valid unique attack paths for 12 regular expressions.

Regular Expression ID	Regular Expression	Valid Unique Attack Path
Regex1	`(Client`$^+$`)(Server`$^+$`)(Log`$^+$`) (Hard Drive`$^+$`)`	17-18-25-24-27-28-21-19-11-12
Regex3	`(Client)(HTTPSever)(GetMethod) (GetMethodBufferWrite)(Buffer)`	1-7-8-9-10-14-12-19-20-12
Regex5	`(Client)(Server) (HeaderFieldBufferWrite)(Buffer)`	1-7-8-9-10-9-8-6
Regex6	`(Client)(HTTPServer) (HTTPMessageHandler)(Log) (Sysadmin)(LogEntryRead)`	1-7-8-9-10-14-13-19-11
		26-25-24-27-28-21-29-23-19-11-5-11
Regex7	`(Client)(HTTPServer) (PostMethod)(HTTPContent-LengthHeaderValue) (HTTPMessagePayloadLength) (ServerConnectionState)`	16-15-14-13-19-23-29-21-19-22
Regex9	`(Client)(SQLInput)(Server) (WebApp)(Database)(Data)(Buffer)`	1-7-8-9-10-14-10-9-8-6
Regex10	`(Client)(SQLInputfield)(Server) (WebApp)(Database)(CPU)`	26-25-24-23-19-13-14
		17-18-25-24-23-19-13-14
Regex11	`(User)(CommandLineArgumentEntry) (Application) (ApplicationServer*) (CommandLineArgumentBufferWrite)`	1-2-3-4-30
Regex13	`(MaliciousClient) (Injection of malicious HTML tags, script in URL, Form) (Cookie*)(FormData*) (ServerVariables*)(VictimClient)`	1-2-3-1-7-8-9-10-14-13
		16-31-15-14-13-14
Regex17	`(Client)(Hyperlink)(Server)`	1-7-8-9-10-14
Regex20	`(UserInput) (IntegerEvaluationRoutine)`	17-18-25-24-27-28-21
Regex23	`(Client)(SearchString)(Server) (Data)(Client)`	26-25-24-23-23-24-25-18-17
		16-31-15-14-10-9-8-7-1
		26-25-24-23-19-13-14-15-31-16

Although regex23 had the most number of unique attacks, there were only three and is a result of creative students finding ways of one client attacking another client in the system design. Regex29, not present in the design, had no valid unique attack paths. Regex1 in the validation study is the same as regex1 in feasibility study, which had two instances of unique attack paths. The low percentage of valid unique attack paths suggests that students usually find the same valid attack paths.

Twenty-six (57.8%) of the invalid attacks paths were only found once for 14 regular expressions (see Table 30).

Table 30: Regular Expressions with Invalid Unique Attack Paths. Twenty-six invalid unique attack paths are associated with 14 regular expressions submitted by students.

Regular Expression ID	Regular Expression	Invalid Unique Attack Path
Regex2	`(Client`$^+$`)(Server`$^+$`) (MessageHeaderHandler`$^+$`) (Hard Drive`$^+$`)`	1-10-19-12
		1-2-3-11-12-19-12
Regex3	`(Client)(HTTPSever)(GetMethod) (GetMethodBufferWrite)(Buffer)`	23-25-24-23-19-20-21
Regex4	`(Client)(HTTPServer)(PostMethod) (Variable + Filename + Header) (Buffer)`	17-18-25-24-27-28-21-22-19-20-21
		1-7-8-9-10-14-13-19-22-21
		1-2-3-11-12-13-19-22-21
Regex7	`(Client)(HTTPServer)(PostMethod) (HTTPContent-LengthHeaderValue) (HTTPMessagePayloadLength) (ServerConnectionState)`	1-7-8-9-10-14-13-19-23-29-21-22-19
		26-25-24-27-28-21-29-23-19-22-19
Regex9	`(Client)(SQLInput)(Server)(WebApp) (Database)(Data)(Buffer)`	26 25 24 12
Regex10	`(Client)(SQLInputfield)(Server) (WebApp)(Database)(CPU)`	3 4 5 6
		16 15 14 28 24
Regex13	`(MaliciousClient)(Injection of malicious HTML tags, script in URL, Form) (Cookie*)(FormData*) (ServerVariables*)(VictimClient)`	17-18-25-24-23-29-12

Table 30 (continued)

Regex14	(User)(Computer)(SyslogFunction)(Log)	14-11
		1-7-8-6
Regex21	(Client)(HTTPServer)(GetRequestRoutine)	26-25-24-27-28
Regex22	(User)(GUI/Browser)(BookMarkSave)(BookmarkBufferWrite)	1-2-3-4-30
Regex24	(Client)(SearchString)(Server)(Data)(Client)	23-28-21
		14-19-21
		17-18-25-24-27-28-21
Regex27	(Client)(RequestMessage)(Router)(CPU)	16 15 14 12
Regex28	(Client)((FTPCommand+MailCOmmand)+(OSCommand)(FTPServer+Mailserver))(Buffer)	16 15 14 12
		16-31-15-14-13-19-21-29-23-24-27-28-21
Regex29	(Class)(Subclass)(OverriddenSecuredMethods)	13-19-22
		17-18-25-24-23-19-21
		19-20-19-22
		13-13-13

Regex29 may be associated with the largest number of invalid unique answers because it was not in the design. This was also true for regex20 in the feasibility study, which represented the same attack. However, regex22, also not shown in the design, had only one invalid unique path. Regex22 and regex29 will be discussed further in Section 7.6. The high percentage of invalid unique attack paths suggest that regular expressions are clear enough to students so that many students will not find the same wrong answer. Also, the percentages of valid and invalid unique answers represent that students are more likely to find invalid unique attacks than valid unique attacks.

The percentages of unique answers between the feasibility and validation assignments are similar; 16% and 12.4% for valid attacks and 72.2% and 57.8% for invalid attacks. This suggests that students are more likely to agree on valid attack paths than invalid attack paths. The security engineer's role is facilitated by the number of paths that stakeholders consistently find. If an attack path is unique, it is more probable that the attack path is incorrect, but his/her judgment must determine the validity. The number of occurrences of attack paths also depends on the number of stakeholders on the project. A larger number of stakeholders will generate a higher probability of attacks that are agreed upon and thus a higher probability of accuracy in finding attacks.

7.6 Regular Expressions Not Represented in the Design

Two regular expressions, regex22 and regex29,

`(User)(GUI/Browser)(BookMarkSave)(BookmarkBufferWrite)` and

`(Class)(Subclass)(OverriddenSecuredMethods)`,

were purposefully inserted into the knowledge base that were not represented in the system design to determine if students accurately found components in the design that matched the components in the regular expression. Answers that were left blank or were explicitly indicated as not existing in the design were analyzed. Figure 11 show the number of students that indicated a regular expression was not in the design.

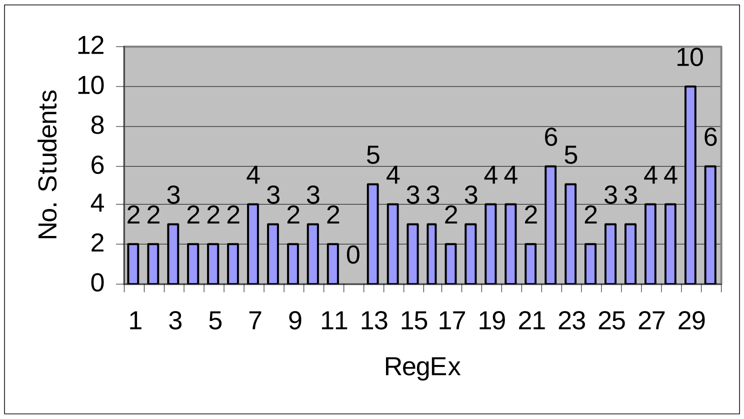


Figure 11: Number of Students Indicating that the Regular Expression was not in the Design.

Thus, only one regular expression, regex12,

`(Client)(HTMLPage)(Server)(Hard Drive)`, had no students saying it was not in the design. As mentioned in Section 7.4, there were large numbers of invalid attack paths for regular expressions not shown in the design and a relatively small number of valid attack paths that were accepted for the regular expressions not intended to be in the design. This trend is similar to that of the feasibility study.

The regular expression

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

was represented by regex29 was purposefully not shown in the high level design to determine if students accurately found components in the design that matched the components in the regular expression. Ten (17.5%) students reported the regular expression as not present in the design. As with regex20 in the feasibility study, 14 (32.6%) students was the maximum number of “not in design” responses of all the regular expressions. Therefore, although a low percentage of the class recognized that the regular expression was not represented in the design, it still has the highest incidence of all regular expression of students declaring it as being not in the design.

Regex29 produced the most number of unique invalid attacks, most number of invalid attacks, and was among the fewest valid attacks in both studies suggesting that when students do not understand a regular expression, they still try to find it in the design. Twenty-two invalid answers were found and placed into nine different attack paths for regex29 (see Table 31).

Table 31: Regular Expression with the Most Number of Invalid Attack Paths. Regex29 was associated with the most number (nine) of invalid attack paths.

Invalid Attack Path	**Number of Student Responses**
31-31-31	1 (1.8%)
13-19-22	1 (1.8%)
17-18-25-24-23-19-21	1 (1.8%)
19-20-19-22	1 (1.8%)
26-25-24-23-24-25-26	3 (5.3%)
16-31-15-14-13-12	7 (12.3%)

13-13-13	1 (1.8%)
10-9-8-6-8-9-10	3 (5.3%)
1-7-8-9-10	4 (7.0%)

There were five unique attack paths suggesting that students were uncertain about how this attack could occur in the design. However, ten (17.5%) students made generalized assumptions about how the regular expression could be applied to the system and thus their answers were accepted and all classified under one path. There were four invalid unique attacks and zero valid unique attack paths.

The regular expression

```
(User)(GUI/Browser)(BookMarkSave)(BookmarkBufferWrite)
```

represented by regex22 was another example of a regular expression not existing in the design. However, students found two valid attack paths with a total of 43 instances (see Table 32).

Table 32: Valid Attack Paths found for Regex22.

Valid Attack Path	**Number of Student Responses**
26-25-24-23-19-21	36 (63.2%)
17-18-25-24-23-19-21	7 (12.3%)

Similar to the feasibility study, where 95.3% of the students found attack paths, a regular expression not intended to be in the design was found by the majority of the students. This represents students were able to look at components in a system design and creatively suggest how such an attack may occur, even though it does not directly correspond with the regular expression. Three different invalid attack paths were found with a frequency of five, five and one (see Table 33).

Table 33: Invalid Attack Paths found for Regex22.

Invalid Attack Path	Number of Student Responses
16-31-15-14-13-19-21	5 (8.8%)
1-7-8-9-10-14-13-12-21	5 (8.8%)
1-2-3-4-30	1 (1.8%)

Six students (10.6% of the class) indicated that the regular expression was not represented in the design. There was one invalid unique attack path and zero valid unique attack paths.

Twenty-eight of the regular expressions were intended to be found in the design. Twenty seven of the 28 regular expressions (96.4%) were incorrectly labeled as not existing in the design and the average number of students indicating this was 3.0. For regex1 through regex10, two students had suspiciously similar data and did not answer the first ten questions that asked to find component sequences for the first ten regular expressions in the knowledge base, thus these answers were considered not in the design and presumable corrupted the data. Only one of the regular expressions, regex12, had no students saying it was not in the design. In both the feasibility and validation studies, there were not any instances of any regular expressions that existed in the design that were not found by students.

7.7 Miscellaneous Data

Fifty-one students responded to the question that asked to rate the approach (see Appendix IX). The possible choices were (1) Poor (2) Below Average (3) Average (4) Good and (5) Excellent. The rating that most students (39.0%) agreed upon was 3. The Likert scale ratings chosen by the students suggest that most students thought the approach was an average approach to detecting security vulnerabilities in a system design via regular expressions Student comments were also collected in the assignment and are grouped together based on similarity (see Appendix X). Six (10.5%) students remarked that there was not enough information to perform the assignment, which is approximately the same percentage as in the feasibility study. The approach attempts to allow non-experts contribute to the identification process and so conceals details about the components in the design, and thus formal descriptions about the components were not made available to the students. In industry, it is unlikely that all stakeholders will know all the details about components and thus they must research the component to know its identity.

Fourteen (24.6%) of the students wrote positive feedback on the approach, which is much more than the four (9.3%) students in the feasibility study. There was one student who did not understand regular expressions 28, 29 and 30 and so indicated this in his answer; there were not any students in the feasibility study that did not understand the regular expressions. The students also asked 13 questions regarding the assignment and can be seen in Appendix XI. Also, there were 49 paths that were entered as answers that were meaningless indicating that students were not interested in the assignment. This did not occur in the feasibility study.

8.0 Conclusions and Future Work

Security vulnerabilities were analyzed in the SecurityFocus, Help Net Security, Secunia and SecurityTracker databases to study what attacks occur today and the techniques used to exploit vulnerabilities. The information in the databases describes how a vulnerability could be exploited in a software application. An analysis of the attack descriptions in the databases reveals the events that transpired and what software components were used in the attack. The events of an attack were formalized by using regular expressions to encapsulate the steps that an attacker took in the exploitation of the software application. This thesis suggests that regular expressions can be used to represent signatures of known attack paths for the identification of security vulnerabilities in future applications. The method of identifying attacks is achieved via matching a sequence of components in a system design that permits the sequence of events in the regular expression to occur. If a match exists, then the vulnerability that was exploited in the known attack 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.

The development of a system design to identify vulnerabilities allows for a graph-based representation of an attack. A pictorial paradigm is consistent with the graph-based approaches proposed by Schneier [17] and McDermott [14] that illuminate different attack scenarios in a system. However, a diagram alone may not be sufficient to thoroughly describe the attack for those partaking in the security assessment. Therefore, text-based profiles are included with each regular expression to explain the events in the attack. Documentation to supplement graphs is also the idea of Moore et al. [15] with the inclusion of attack profiles associated with attack patterns and Steffan and Schumacher [19] with their contribution of a ATiki that allows for collaborative efforts in the description of places and transitions in attack nets. The availability of both a graph- and text-based description of an attack alleviates the requirements of a strong security background as evidenced by students, with at most a limited security background, finding excessively more valid attacks paths than invalid ones. Vendors that do not have a security engineer accessible to evaluate their systems have the alternative of using the approach in this thesis or the use of attack trees, attack patterns, attack profiles, attack nets, or ATikis to integrate security into the design phase of their software process. Each approach requires that a human read the architecture and then determine if the architecture matches an attack profile, attack tree, or attack net. The matching process of regular expressions to sequences of components in the design facilitates the identification of vulnerabilities for the stakeholders by indicating exactly what components need to be searched.

A total of 409 vulnerabilities from the SecurityFocus, Help Net Security, Secunia and SecurityTracker vulnerability databases were studied in this thesis. Of this sample, 237 could be used in the regular expression-oriented approach. Fifty-three regular expressions were created, and on average one regular expression was able to abstract six vulnerabilities. The regular expressions were classified into the following six categories: buffer overflows, malformed data, remote executions, CPU/hard drive consumptions, privilege escalation errors, and a miscellaneous category that contained dissimilar attacks. The results indicated that vulnerabilities can be abstracted to form regular expressions although the process of producing regular expression is bounded by the information supplied in the vulnerability databases.

A feasibility and validation study tested students' ability to match a regular expression to a system design. In both studies students gave more correct answers than incorrect answers of how an attack path could match with a regular expression. Also, students found many attack paths that other students found representing the high degree of correlation between a regular expression and a sequence of components in a system design. Most of the invalid answers resulted from students trying to find attack paths that were not in the design. This is contradictory evidence that regular expressions do accurately represent attack paths in the design. More elaborate profiles and perhaps WikiWikiWebs to allow for collaboration on attack paths as suggested by Steffan and Schumacher [19] may be necessary to prevent incorrect identification of attack paths represented by regular expressions.

There are several limitations that accompany this thesis' approach to applying security to software. First, the vulnerabilities studies pertain to software coding flaws excluding the configuration, encryption, and networking errors mentioned in Chapter 5. Thus, this approach is intended to be used in tandem with different approaches that address other dimensions of security. Secondly, the vulnerabilities studied are limited to known vulnerabilities and do not show what new types of attacks are possible nor vulnerabilities that may be specific to the system under security analysis. Furthermore, vulnerability identification is limited to the comprehensiveness of a knowledge base containing known attacks. A knowledge base with few attack entries does not have the same potential as a knowledge base of many attack entries. A comprehensive knowledge base that contains every vulnerability or attack ever discovered is an insurmountable task. However, stakeholders involved with the security of an organization can update their knowledge base by entering the new vulnerabilities described in security databases such as SecurityFocus. Also, security vulnerabilities found within an organization that are not in the knowledge base should also be added to the collection to retain the tacit knowledge of the individual(s) that discovered the attack. Ideally, knowledge bases among different organizations could be merged to account for more attack paths, but such a method of sharing is not likely because of the proprietary information that may be released in the description of the attack.

The accuracy of the regular expressions and profiles is dependent on the genuineness of the information in the four vulnerability databases studied. The validity of the data entered in the databases depends on thorough research and testing on vendor software by those who discover and enter the vulnerability. Due to most of the software applications being proprietary, the information that explicitly describes the vulnerabilities is confidential. Thus, the information in the vulnerability descriptions is, in general, limited to high-level descriptions limiting most regular expressions to be high-level representations of attack paths. Low-level detail in the regular expressions could aid software engineers in the identification of subtle flaws in their code that could be exploited.

The approach presented in this thesis relies on an organization to use designs at the beginning of the software process. Since not all organizations and software processes require designs, this approach can be quickly ruled out of any attempt for security assessment in those organizations. Also, software applications that have volatile requirements are likely to change the design, and hence the security evaluation may need repeated, especially for low-level regular expressions. However, regular expressions that represent high-level attack paths may endure requirements changes alleviating the need for a secondary security evaluation. And, the technique assumes that all components in the design are made from scratch. That is, third party software that will be used in the future application may already have security built into their applications. Applying regular expressions to fortified components may thus be a wasted attempt of identifying security vulnerabilities.

The identification of false positives in this study represent common problem with security approaches today. False positives were found in feasibility and validation studies because the regular expression was not accurate enough to describe an attack path and because stake holders did not find valid attack paths. False positives reduce the efficacy of the approach when all the attack paths submitted by stakeholders are submitted for review for a risk management team. Each path that is a false positive consumes time that could have been used to assess a true positive. A large number of false positives would certainly slow the security process and question the validity of the approach. Fortunately, the results from the feasibility and validation studies indicated the number of false positives is much less than the true positives. Not only are large numbers of false positives a threat to the system, but so are large numbers of regular expressions. It may seem infeasible to have a large number of regular expressions and an even higher number of attack paths that they map to. Perfect security is likely to be unattainable and thus an organization needs to determine what impending attacks will cause the most loss. The responsibility of securing a software application should thus be given to a risk management team that determines what vulnerabilities must be secured and what losses are acceptable.

The approach should further be validated 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, further work to is needed to expedite the identification process of finding vulnerabilities and reduce the number of false positives. Automating the matching of regular expressions to the components in the system design conserves the time for identifying vulnerabilities and grants more time for risk management. An automated process may also more accurately identify attack paths, thus decreasing the number of false positives in a security analysis. Lastly, test cases between each component in the system design need to be created to restrict the flow of data and provide access control to prevent the attacks. Successful implementations of the recommended future work may offer an expedient and accurate method of identifying security vulnerabilities in future software applications.

9.0 References

1. Alberts, C. and Dorofee, A. OCTAVE Criteria, Version 2.0, CMU/SEI, Pittsburgh, PA, 2001.

2. Arbaugh, W.A., Fithen, W.L. and McHugh, J. Windows of Vulnerability: A Case Study Analysis. *IEEE*, *3* (12). 52-59.

3. Bishop, M. and Dilger, M. Checking for Race Conditions in File Accesses. *Computing Systems*, *9* (2). 131-152.

4. Boehm, B. *Software Engineering Economics*. Prentice-Hall, New Jersey, 1981.

5. Boehm, B.. Tutorial: Software Risk Management, IEEE Computer Society Press, 1989.

6. Carlstedt, J., Bisbey II, R. and Popek, G. Pattern-Directed Protection Evaluation, USC Information Sciences Institute, Marina del Rey, 1975.
7. Cunningham, W. The WikiWikiWeb. (http://c2.com/cgi/wiki?WelcomeVisitors), 1994.

8. Fites, Philip E., Martin P. J. Kratz, and Alan F. Brebner, *Control and Security of Computer Information Systems*, W. H. Freeman/Computer Science Press, September. 1988.

9. Garfinkel, S. and Spafford, E. *Web Security & Commerce*. O'Reilly & Associates, Inc., Sebastapol, 1997.

10. Helmer, G., Wong, J., Slagell, M., Honavar, V., Miller, L. and Lutz, R. Software Fault Tree and Colored Petri Net Based Specification, Design and Implementation of Agent-Based Intrusion Detection-Systems.

11. Howard, M. and LeBlanc, D. *Writing Secure Code*. Microsoft Corporation, Redmond, 2003

12. Jürjens, J.: *UMLsec*: *Extending UML for Secure Systems Development*, Fifth International Conference on the Unified Modeling Language - the Language and its applications, 2002

13. Kumar, S. and Spafford, E., A Pattern Matching Model for Misuse Intrusion Detection. in *Proceedings of the 17th National Computer Security Conference*, (West Lafayette, 1994), Purdue University.

14. McDermott, J. Attack Net Penetration Testing. In The 2000 New Security Paradigms Workshop (Ballycotton, County Cork, Ireland, Sept. 2000), ACM SIGSAC, ACM Press, pp.15-22.

15. A. P. Moore, R. J. Ellison, and R. C. Linger. *Attack modeling for information security and survivability*. Technical Note CMU/SEI-2001-TN-001, CMU Software Engineering Institute, March 2001.

16. Schneider, F. *Trust in Cyberspace*. National Academy Press, Washington, DC, 1998.

17. Schneier, Bruce, Attack Trees: Modeling Security Threats. Dr. Dobb's Journal, CMP Media, Inc. December 1999.

18. Spafford, E., (http://www.landfield.com/isn/mail-archive/1998/Oct/0093.html) 1999.

19. Jan Steffan, Markus Schumacher, Collaborative Attack Modeling, Proceedings of the 2002 ACM Symposium on Applied Computing (SAC'02, Madrid, Spain), pp. 253-259, ACM Press, ISBN: 1-58113-445-2, 2002

20. Viega, J. and McGraw, G. *Building Secure Software How to Avoid Security Problems the Right Way*. Addison-Wesley, Boston, 2002.

21. Weissman, Clark (1995). Penetration Testing. In M. Abrams, S. Jajodia, and H. Podell (Eds.),*Information Security: An Integrated Collection of Essays*, pp. 269-296. Los Alamitos, CA:IEEE Computer Society Press.

Appendices

Appendix I

Vulnerability Collection

Table 34: Vulnerability Collection

Date Published	SecurityFocus Title of Vulnerability	Bugtraq ID	Associated Regular Expression
14-Feb-04	Microsoft IIS Unspecified Remote Denial Of Service Vulnerability	9660	1
3-Sep-98	Multiple Vendor MIME Header DoS Vulnerability	1760	2
29-Aug-99	Netscape Enterprise Server GET Request Vulnerability	1024	3
17-Jan-00	InetServ 3.0 WebMail Long GET Request Vulnerability	949	3
14-Dec-00	Watchguard SOHO Firewall Oversized GET Request DoS Vulnerability	2114	3
21-Feb-02	Nombas ScriptEase:WebServer Edition GET Request Denial of Service Vulnerability	4145	3
17-Apr-02	WebTrends Reporting Center GET Request Buffer Overflow Vulnerability	4531	3
8-Jul-02	MyWebServer GET Request Buffer Overflow Vulnerability	5184	3
16-Sep-02	PlanetWeb Long GET Request Buffer Overflow Vulnerability	5710	3
30-Sep-02	WN Server Malformed GET Request Buffer Overflow Vulnerability	5831	3
12-Oct-02	My Web Server Long Get Request Denial Of Service Vulnerability	5954	3
7-Nov-02	Macromedia JRun IIS ISAPI Filter GET Request Buffer Overrun Vulnerability	6122	3
12-Nov-02	Light HTTPD GET Request Buffer Overflow Vulnerability	6162	3
16-May-03	Snowblind Web Server HTTP GET Request Buffer Overflow Vulnerability	7619	3
23-Jun-03	Armida Databased Web Server Remote GET Request Denial Of Service Vulnerability	8017	3
14-Jul-03	Twilight WebServer GET Request Buffer Overflow Vulnerability	8181	3
25-Sep-03	Athttpd Remote GET Request Buffer Overrun Vulnerability	8709	3

Table 34 (continued)

8-Oct-03	Centrinity FirstClass HTTP Server Long Version Field Denial Of Service Vulnerability	8793	3
3-Nov-03	IA WebMail Server Long GET Request Buffer Overrun Vulnerability	8965	3
24-Jan-04	TinyServer Multiple Vulnerabilities	9485	3
26-Jan-04	InternetNow ProxyNow Multiple Stack and Heap Overflow Vulnerabilities	9500	3
28-Jan-04	Loom Software SurfNow Remote HTTP GET Request Denial Of Service Vulnerability	9519	3
28-Jan-04	Macromedia ColdFusion MX Form Fields Denial of Service Vulnerability	9522	3
17-Feb-04	Vizer Web Server Remote Denial of Service Vulnerability	9678	3
17-Feb-04	KarjaSoft Sami HTTP Server GET Request Buffer Overflow Vulnerability	9679	3
17-Feb-04	KarjaSoft Sami HTTP Server GET Request Buffer Overflow Vulnerability	9679	3
23-Feb-04	Avirt Voice HTTP GET Remote Buffer Overrun Vulnerability	9721	3
23-Feb-04	Avirt Soho Server HTTP GET Buffer Overrun Vulnerability	9722	3
23-Feb-04	Avirt Soho Web Service HTTP GET Buffer Overrun Vulnerability	9723	3
5-Mar-04	Seattle Lab Software SLMail Pro Remote Buffer Overflow Vulnerability	9809	3
27-Jul-02	D-Link Print Server Long Post Request Denial Of Service Vulnerability	5330	4
25-Nov-02	Pserv HTTP POST Request Buffer Overflow Vulnerability	6242	4
31-Jul-03	McAfee ePolicy Orchestrator Agent POST Request Heap Overflow Vulnerability	8316	4
22-Jan-04	McAfee ePolicy Orchestrator Agent HTTP POST Buffer Mismanagement Vulnerability	9476	4
9-Feb-04	Sambar Server Results.STM Post Request Buffer Overflow Vulnerability	9607	4
26-Feb-04	Dell OpenManage Web Server POST Request Heap Overflow Vulnerability	9750	4

Table 34 (continued)

26-Feb-04	Dell OpenManage Web Server POST Request Heap Overflow Vulnerability	9750	4
29-Oct-03	TelCondex SimpleWebserver HTTP Referer Remote Buffer Overflow Vulnerability	8925	5
1-Nov-03	BRS WebWeaver httpd `User-Agent` Remote Denial of Service Vulnerability	8947	5
18-Feb-04	Metamail Multiple Buffer Overflow/Format String Handling Vulnerabilities	9692	5
24-Feb-04	Apple QuickTime/Darwin Streaming Server DESCRIBE Request Remote Denial of Service Vulnerability	9735	5
27-Feb-04	UUDeview MIME Archive Buffer Overrun Vulnerability	9758	5
8-Nov-03	Liteserve Server Log Handling Buffer Overflow Vulnerability	0	6
24-Sep-03	NullLogic Null HTTPd Remote Denial Of Service Vulnerability	8697	7
16-Feb-04	RobotFTP Server Username Buffer Overflow Vulnerability	9672	8
27-Feb-04	Calife Password Heap Overrun Vulnerability	9756	8
1-Mar-04	Calife Local Memory Corruption Vulnerability	9776	8
23-Jan-04	QuadComm Q-Shop SQL Injection Vulnerabilities	9481	9
4-Feb-04	All Enthusiast ReviewPost PHP Pro Multiple SQL Injection Vulnerabilities	9574	9
6-Feb-04	Multiple Oracle Database Parameter/Statement Buffer Overflow Vulnerabilities	9587	9
9-Feb-04	PHP-Nuke Public Message SQL Injection Vulnerability	9615	9
10-Feb-04	PHPNuke Web_Links Module Remote SQL Injection Vulnerability	9630	9
11-Feb-04	BosDev BosDates SQL Injection Vulnerability	9639	9
16-Feb-04	YABB SE Quote Parameter SQL Injection Vulnerability	9674	9
18-Feb-04	Ecommerce Corporation Online Store Kit Multiple SQL Injection Vulnerabilities	9687	9

Table 34 (continued)

19-Feb-04	PunkBuster Database Remote SQL Injection Vulnerability	9697	9
28-Feb-04	Invision Power Board Search.PHP "st" SQL Injection Vulnerability	9766	9
1-Mar-04	IGeneric Free Shopping Cart SQL Injection Vulnerability	9771	9
1-Mar-04	YABB SE Multiple Input Validation Vulnerabilities	9774	9
3-Mar-04	SpiderSales Shopping Cart Multiple Vulnerabilities	9799	9
9-Mar-04	Confixx DB Parameter SQL Injection Vulnerability	9830	9
18-Jul-02	SQL Injection Legalities	Dev Archive	10
4-Mar-04	Sun Solaris Multiple Unspecified Local UUCP Buffer Overrun Vulnerabilities	9837	11
15-Jul-03	Citadel/UX Unlimited Biography Data Denial Of Service Vulnerability	8192	12
7-Nov-02	Summit Computer Networks Lil' HTTP Server pbcgi.cgi Cross Site Scripting Vulnerability	5211	13
31-Dec-03	GNU Mailman Admin Page Multiple Cross-Site Scripting Vulnerabilities	9336	13
21-Jan-04	Darkwet Network WebcamXP Cross-Site Scripting Vulnerability	9465	13
21-Jan-04	Mephistoles HTTPD Cross-Site Scripting Vulnerability	9470	13
22-Jan-04	Acme thttpd CGI Test Script Cross-Site Scripting Vulnerability	9474	13
23-Jan-04	Novell Netware Enterprise Web Server Multiple Vulnerabilities	9479	13
23-Jan-04	QuadComm Q-Shop Cross Site Scripting Vulnerabilities	9480	13
24-Jan-04	Oracle HTTP Server isqlplus Cross-Site Scripting Vulnerability	9484	13
26-Jan-04	IBM Net.Data db2www Error Message Cross-Site Scripting Vulnerability	9488	13
26-Jan-04	Herberlin BremsServer Cross-Site Scripting Vulnerability	9491	13
26-Jan-04	Cherokee Error Page Cross Site Scripting Vulnerability	9496	13
26-Jan-04	Xoops Viewtopic.php Cross-Site Scripting Vulnerability	9497	13
27-Jan-04	WebLogic Server and Express HTTP TRACE Credential Theft Vulnerability	9506	13

Table 34(continued)

27-Jan-04	Novell Groupwise Webaccess Cross Site Scripting Vulnerability	9508	13
28-Jan-04	BRS WebWeaver ISAPISkeleton.dll Cross-Site Scripting Vulnerability	9516	13
29-Jan-04	CPAN WWW::Form HTML Injection Vulnerability	9526	13
3-Feb-04	Summit Computer Networks Lil' HTTP Server URLCount.CGI HTML Injection Vulnerability	5115	13
3-Feb-04	PHPX Multiple Vulnerabilities	9569	13
4-Feb-04	RXGoogle.CGI Cross Site Scripting Vulnerability.	9575	13
9-Feb-04	PHP-Nuke 'Reviews' Module Cross-Site Scripting Vulnerability	9605	13
9-Feb-04	JShop E-Commerce Suite xSearch Cross-Site Scripting Vulnerability	9609	13
9-Feb-04	PHP-Nuke 'News' Module Cross-Site Scripting Vulnerability	9613	13
10-Feb-04	MaxWebPortal Multiple Input Validation Vulnerabilities	9625	13
12-Feb-04	JelSoft VBulletin Cross-Site Scripting Vulnerability	9649	13
13-Feb-04	JelSoft VBulletin Search.PHP Cross-Site Scripting Vulnerability	9656	13
18-Feb-04	WebCortex WebStores2000 Error.ASP Cross-Site Scripting Vulnerability	9693	13
19-Feb-04	LiveJournal HTML Injection Vulnerability	9700	13
23-Feb-04	EZBoard Font Tag HTML Injection Vulnerability	9725	13
23-Feb-04	LiveJournal CSS HTML Injection Vulnerability	9727	13
24-Feb-04	Seyeon Technology FlexWATCH Server Cross-Site Scripting Vulnerability	9739	13
26-Feb-04	CalaCode @mail Webmail System Cross-Site Scripting Vulnerability	9748	13
26-Feb-04	Symantec Gateway Security Error Page Cross-Site Scripting Vulnerability	9755	13
28-Feb-04	PHPBB ViewTopic.PHP "postorder" Cross-Site Scripting Vulnerability	9765	13
1-Mar-04	Invision Power Board Multiple Cross-Site Scripting Vulnerabilities	9768	13

Table 34 (continued)

1-Mar-04	IGeneric Free Shopping Cart Cross-Site Scripting Vulnerability	9773	13
1-Mar-04	Software602 602Pro LAN Suite Web Mail Cross-Site Scripting Vulnerability	9777	13
1-Mar-04	Software602 602Pro LAN Suite Web Mail Installation Path Disclosure Vulnerability	9781	13
2-Mar-04	NetScreen SA 5000 Series delhomepage.cgi Cross-Site Scripting Vulnerability	9791	13
3-Mar-04	SandSurfer Multiple Undisclosed Cross-Site Scripting Vulnerabilities	9801	13
5-Mar-04	VirtuaSystems VirtuaNews Multiple Module Cross-Site Scripting Vulnerabilities	9812	13
8-Mar-04	VirtuaSystems VirtuaNews Admin.PHP Cross-Site Scripting Vulnerability	9819	13
9-Mar-04	Invision Power Board Pop Parameter Cross-Site Scripting Vulnerability	9822	13
1-Mar-04	JFTPGW Remote Syslog Format String Vulnerability	10438	14
27-Jan-04	Apple Mac OS X TruBlueEnvironment Local Buffer Overflow Vulnerability	9509	15
27-Jan-04	IBM Informix Multiple Local Privilege Escalation Vulnerabilities	9511	15
5-Feb-04	SGI IRIX Libdesktopicon.so Local Buffer Overflow Vulnerability	9547	15
21-Feb-04	LGames Lbreakout2 Multiple Environment Variable Buffer Overflow Vulnerabilites	9712	15
27-Feb-04	xboing Local Buffer Overflow Vulnerabilities	9764	15
19-Feb-98	Netscape 4 DoS/Possibly exploitable buffer overflow	archive	16
20-Feb-04	Xfree86 Direct Rendering Infrastructure Buffer Overflow Vulnerabilities	9701	17
14-Jul-98	Malicious Java applet security flaw in ClassLoader Vulnerability	164	19
4-Feb-04	Multiple RealPlayer/RealOne Player Supported File Type Buffer Overrun Vulnerabilities	9579	20
7-Feb-04	The Palace Graphical Chat Client Remote Buffer Overflow Vulnerability	9602	20

Table 34 (continued)

10-Nov-02	Monkey HTTP Server Invalid POST Request Denial Of Service Vulnerability	6096	21
2-Dec-02	libSieve Header Name Buffer Overrun Vulnerability	6294	21
19-Dec-02	CUPS Negative Length HTTP Header Vulnerability	6437	21
21-Nov-03	Imatix Xitami Post Request Header Remote Denial Of Service Vulnerability	9083	21
4-Feb-04	Web Crossing Web Server Component Remote Denial Of Service Vulnerability	9576	21
3-Dec-03	GNU Zebra / Quagga Remote Denial of Service Vulnerability	9029	22
4-Feb-04	GNU Radius Remote Denial Of Service Vulnerability	9578	22
11-Jun-03	ArGoSoft Mail Server Multiple GET Requests Denial Of Service Vulnerability	7873	23
12-Feb-04	Crob FTP Server Remote Denial Of Service Vulnerability	9651	23
26-Feb-04	CalaCode @mail Webmail System POP3 Remote Denial of Service Vulnerability	9749	23
26-Jan-04	Gaim Multiple Remote Boundary Condition Error Vulnerabilities	9489	24
16-Feb-04	Microsoft Internet Explorer Bitmap Processing Integer Overflow Vulnerability	9663	24
24-Feb-04	RedStorm Ghost Recon Game Engine Remote Denial Of Service Vulnerability	9738	24
2-Mar-04	Coreutils DIR Width Argument Integer Overflow Vulnerability	9793	24
3-Mar-04	QMail-QMTPD RELAYCLIENT Environment Variable Integer Overflow Vulnerability	9797	24
8-Mar-04	Network Time Protocol Daemon Integer Overflow Vulnerability	9818	24
17-Dec-98	Microsoft IIS Malformed HTTP Get Request Denial Of Service Vulnerability	6789	25
29-Mar-00	Xitami Webserver empty GET request DoS Vulnerability	3511	25
11-Apr-01	Lotus Domino R5 Server GET Request DoS Vulnerability	2571	25

Table 34 (continued)

8-Jul-02	Working Resources BadBlue Get Request Denial Of Service Vulnerability	5187	25
29-Jul-02	Abyss Web Server HTTP GET Request Directory Contents Disclosure Vulnerability	5345	25
18-Nov-02	Perception LiteServe Malformed GET Request Buffer Overflow Vulnerability	6192	25
17-Mar-03	McAfee ePolicy Orchestrator HTTP GET Request Format String Vulnerability	7111	25
12-May-03	Pi3Web Malformed GET Request Denial Of Service Vulnerability	7555	25
12-Jun-03	WebBBS Pro Malicious GET Request Denial Of Service Vulnerability	7890	25
19-Jun-03	Power Server Remote GET Request Denial of Service Vulnerability	7983	25
19-Jan-04	Pablos FTP Server Unauthorized File Existence Disclosure Vulnerability	9443	25
20-Jan-04	Anteco Visual Technologies OwnServer Directory Traversal Vulnerability	9461	25
20-Jan-04	2Wire HomePortal Series Directory Traversal Vulnerability	9463	25
20-Jan-04	AIPTEK NETCam Webserver Directory Traversal Vulnerability	9456	25
20-Jan-04	Leif M. Wright Web Blog File Disclosure Vulnerability	9517	25
22-Jan-04	Netbus Directory Listings Disclosure and File Upload Vulnerability	9475	25
24-Jan-04	Borland Webserver for Corel Paradox Directory Traversal Vulnerability	9486	25
26-Jan-04	Herberlin BremsServer Directory Traversal Vulnerability	9493	25
29-Jan-04	PJ CGI Neo Review Directory Traversal Vulnerability	9524	25
30-Jan-04	PhpGedView Editconfig_gedcom.php Directory Traversal Vulnerability	9529	25
30-Jan-04	JBrowser Browser.PHP Directory Traversal Vulnerability	9535	25
12-Feb-04	Macallan Mail Solution Web Interface Authentication Bypass Vulnerability	9646	25
23-Feb-04	phpNewsManager Functions Script File Disclosure Vulnerability	9720	25

Table 34 (continued)

24-Feb-04	Apache Cygwin Directory Traversal Vulnerability	9733	25
24-Feb-04	GWeb HTTP Server Directory Traversal Vulnerability	9742	25
4-Mar-04	SmarterTools SmarterMail Multiple Vulnerabilities	9805	25
8-Mar-04	PWebServer Remote Directory Traversal Vulnerability	9817	25
16-Feb-04	ShopCartCGI Remote File Disclosure Vulnerability	9670	26
18-Feb-04	Owl's Workshop Multiple Remote File Disclosure Vulnerabilities	9689	26
22-Jan-04	EA Black Box Need For Speed Hot Pursuit 2 Game Client Remote Buffer Overflow Vulnerability	9473	27
16-Feb-04	Freeform Interactive Purge/Purge Jihad Game Client Remote Buffer Overflow Vulnerability	9671	27
1-Mar-04	Volition Red Faction Game Client Remote Buffer Overflow Vulnerability	9775	27
2-Mar-04	Volition Freespace 2 Game Client Remote Buffer Overflow Vulnerability	9785	27
18-Feb-04	Linux Kernel NCPFS ncp_lookup() Unspecified Local Privilege Escalation Vulnerability	9691	28
19-Feb-04	Zone Labs ZoneAlarm SMTP Remote Buffer Overflow Vulnerability	9696	29
20-Feb-04	Oracle 9i Application/Database Server SOAP XML DTD Denial Of Service Vulnerability	9703	30
17-Jun-03	Proxomitron Proxy Server Long Get Request Remote Denial Of Service Vulnerability	7954	31
20-Feb-04	PSOProxy Remote Buffer Overflow Vulnerability	9706	31
23-Feb-04	Proxy-Pro Professional GateKeeper Web Proxy Buffer Overrun Vulnerability	9716	31
31-Mar-03	Kerio WinRoute Firewall Malformed HTTP GET Request Denial of Service Vulnerability	7245	32
3-Dec-02	Multiple Linksys Devices GET Request Buffer Overflow Vulnerability	6301	33
28-Feb-03	USRobotics Broadband-Router GET Request DoS Vulnerability	6994	33

Table 34 (continued)

12-Oct-02	SurfControl SuperScout WebFilter Malformed GET Request DoS Vulnerability	5854	34
24-Apr-03	VisNetic ActiveDefense Multiple GET Request Denial of Service Vulnerability	7428	34
3-Dec-03	Linksys WRT54G Router Blank HTTP GET Request Denial Of Service Vulnerability	9152	35
4-Feb-04	TYPSoft FTP Server Remote Denial Of Service Vulnerability	9573	36
16-Feb-04	RhinoSoft Serv-U FTP Server SITE CHMOD Buffer Overflow Vulnerability	9675	36
20-Feb-04	TYPSoft FTP Server Remote CPU Consumption Denial Of Service Vulnerability	9702	36
26-Feb-04	RhinoSoft Serv-U FTP Server MDTM Command Time Argument Buffer Overflow Vulnerability	9751	36
27-Feb-04	ArGoSoft FTP Server Multiple Vulnerabilities	9770	36
28-Feb-04	Multiple WFTPD Vulnerabilities	9767	36
2-Mar-04	ProFTPD _xlate_ascii_write() Buffer Overrun Vulnerability	9782	36
2-Mar-04	1st Class Internet Solutions 1st Class Mail Server Remote Buffer Overflow Vulnerability	9794	36
23-Jan-04	Reptile Web Server Remote Denial Of Service Vulnerability	9482	37
17-Feb-04	TransSoft Broker FTP Server Denial of Service Vulnerabilities	9680	37
7-Feb-04	BolinTech Dream FTP Server User Name Format String Vulnerability	9600	38
1-Mar-04	Squid Proxy NULL URL Character Unauthorized Access Vulnerability	9778	38
24-Jan-04	RhinoSoft Serv-U FTP Server MDTM Command Stack Overflow Vulnerability	9483	39
5-Feb-04	XLight FTP Server Long Directory Request Remote Denial Of Service Vulnerability	9585	39
10-Feb-04	XLight FTP Server Remote Denial Of Service Vulnerability	9627	39
16-Feb-04	ACLogic CesarFTP Remote Resource Exhaustion Vulnerability	9666	39

Table 34 (continued)

16-Feb-04	XLight FTP Server Remote Send File Request Denial Of Service Vulnerability	9668	39
17-Feb-04	SmallFTPD Remote Denial Of Service Vulnerability	9684	39
12-Mar-03	Novell Netware FTPSERV.NLM FTP GET Denial Of Service Vulnerability	7073	40
20-Feb-04	Microsoft Windows XP explorer.exe Multiple Memory Corruption Vulnerabilities	9707	41
19-Feb-04	AOL Instant Messenger Buddy Icon Predictable File Location Weakness	9698	42
20-Feb-04	Multiple Outlook/Outlook Express Predictable File Location Weaknesses	9709	42
20-Jan-04	SuSE Multiple Scripts Insecure Temporary File Handling Symbolic Link Vulnerabilities	9457	43
27-Jan-04	IBM Informix Dynamic Server/Informix Extended Parallel Server Multiple Vulnerabilities	9512	43
28-Jan-04	Internet Security Systems BlackICE PC Protection Upgrade File Permission Vulnerability	9513	43
30-Jan-04	GNU LibTool Local Insecure Temporary Directory Creation Vulnerability	9530	43
12-Feb-04	Mailmgr Insecure Temporary File Creation Vulnerabilities	9654	43
22-Feb-04	Synaesthesia Insecure File Creation Vulnerability	9713	43
8-Mar-04	GNU Automake Insecure Temporary Directory Creation Symbolic Link Vulnerability	9816	43
16-Feb-04	Symantec AntiVirus Scan Engine For Red Hat Linux Insecure Temporary File Vulnerabilities	9662	44
22-Feb-04	Dell TrueMobile 1300 WLAN System Tray Applet Local Privilege Escalation Vulnerability	9714	44
20-Jan-04	PHPix Remote Arbitrary Command Execution Vulnerability	9458	45
31-Jan-04	Leif M. Wright Web Blog Remote Command Execution Vulnerability	9539	45
23-Feb-04	Confirm E-Mail Header Remote Command Execution Vulnerability	9728	45

Table 34 (continued)

23-Feb-04	RobotFTP Server Remote Pre-authenticated Command Denial Of Service Vulnerability	9729	46
26-Jan-04	Kietu Index.PHP Remote File Include Vulnerability	9499	47
26-Jan-04	Gallery Remote Global Variable Injection Vulnerability	9490	47
29-Jan-04	Third-party CVSup Binary Insecure ELF RPATH Library Replacement Vulnerability	9523	47
30-Jan-04	PhpGedView [GED_File]_conf.php Remote File Include Vulnerability	9531	47
30-Jan-04	Laurent Adda Les Commentaires PHP Script Multiple Module File Include Vulnerability	9636	47
11-Feb-04	VisualShapers ezContents Multiple Module File Include Vulnerability	9638	47
16-Feb-04	Voice Of Web AllMyPHP Remote File Include Vulnerabilities	9664	47
24-Feb-04	Opt-X header.php Remote File Include Vulnerability	9732	47
3-Mar-04	BolinTech Dream FTP Server FTP Command Format String Vulnerability	9800	48
2-Mar-04	Magic Winmail Server LDapLib.PHP Remote Installation Path Disclosure Vulnerability	9786	49
5-Mar-04	Invision Power Board Error Message Path Disclosure Vulnerability	9810	49
9-Mar-04	IBM DB2 Remote Command Server Privilege Escalation Vulnerability	9821	50
27-Jan-04	Microsoft Internet Explorer CLSID File Extension Misrepresentation Vulnerability	9510	51
11-Feb-04	Opera Web Browser CLSID File Extension Misrepresentation Vulnerability	9640	51
26-Jan-04	Antologic Antolinux Administrative Interface NDCR Parameter Remote Command Execution Vulnerability	9495	52
26-Jan-04	Mbedthis Software AppWeb HTTP Server Empty Options Request Denial Of Service Vulnerability	9494	53
29-Oct-02	Apache 2 WebDAV CGI POST Request Information Disclosure Vulnerability	6065	220

Table 34 (continued)

20-Jan-04	NetScreen Security Manager Insecure Default Remote Communication Vulnerability	9455	220
26-Jan-04	mIRC DCC Get Dialog Denial Of Service Vulnerability	9492	220
28-Jan-04	TRR19 Privilege Escalation Vulnerability	9520	220
28-Jan-04	Macromedia ColdFusion MX Security Sandbox Circumvention Vulnerability	9521	220
28-Jan-04	Internet Security Systems BlackICE PC Protection blackd.exe Local Buffer Overrun Vulnerability	9514	220
29-Jan-04	Kerio Personal Firewall Local Privilege Escalation Vulnerability	9525	220
30-Jan-04	ChatterBox Remote Denial of Service Vulnerability	9532	220
30-Jan-04	Sun Solaris PFExec Custom Profile Arbitrary Privileges Vulnerability	9534	220
31-Jan-04	BugPort Unauthorized Configuration File Viewing Vulnerability	9542	220
1-Feb-04	Eggdrop Share Module Arbitrary Share Bot Add Vulnerability	9606	220
3-Feb-04	Sambar Server Results.STM Post Request Buffer Overflow Vulnerability	9607	220
4-Feb-04	Apache mod_python Module Malformed Query Denial of Service Vulnerability	9129	220
4-Feb-04	Linux Kernel R128 Device Driver Unspecified Privilege Escalation Vulnerability	9570	220
4-Feb-04	Apache mod_digest Client-Supplied Nonce Verification Vulnerability	9571	220
4-Feb-04	FreeBSD NetINet TCP Maximum Segment Size Remote Denial Of Service Vulnerability	9572	220
5-Feb-04	Multiple Vendor libc DNS Resolver Information Leakage Vulnerability	6116	220
5-Feb-04	Netpbm Temporary File Vulnerabilities	9442	220
5-Feb-04	GNU LibTool Local Insecure Temporary Directory Creation Vulnerability	9530	220
5-Feb-04	IBM Cloudscape Database Remote Command Execution Vulnerability	9583	220
6-Feb-04	BSD Kernel SHMAT System Call Privilege Escalation Vulnerability	9586	220

Table 34 (continued)

6-Feb-04	Mambo Open Source Itemid Parameter Cross-Site Scripting Vulnerability	9588	220
6-Feb-04	Cactusoft CactuShop Lite Remote Arbitrary File Deletion Backdoor Vulnerability	9589	220
6-Feb-04	Apache-SSL Client Certificate Forging Vulnerability	9590	220
6-Feb-04	Joe Lumbroso Jack's Formmail.php Unauthorized Remote File Upload Vulnerability	9591	220
6-Feb-04	Linux VServer Project CHRoot Breakout Vulnerability	9596	220
7-Feb-04	Multiple Check Point Firewall-1 HTTP Security Server Remote Format String Vulnerabilities	9581	220
7-Feb-04	Brad Fears PHPCodeCabinet comments.php HTML Injection Vulnerability	9601	220
7-Feb-04	Apache mod_php Global Variables Information Disclosure Weakness	9599	220
8-Feb-04	OpenBSD ICMPV6 Handling Routines Remote Denial Of Service Vulnerability	9577	220
8-Feb-04	GNU Mailman Malformed Message Remote Denial Of Service Vulnerability	9620	220
9-Feb-04	CDE LibDTHelp DTHelpUserSearchPath Local Buffer Overflow Vulnerability	8973	220
9-Feb-04	Check Point VPN-1/SecuRemote ISAKMP Large Certificate Request Payload Buffer Overflow Vulnerability	9582	220
9-Feb-04	Multiple Nokia Object Exchange Protocol Message Remote Denial Of Service Vulnerabilities	9603	220
9-Feb-04	Nadeo Game Engine Remote Denial of Service Vulnerability	9604	220
9-Feb-04	ClamAV Daemon Malformed UUEncoded Message Denial Of Service Vulnerability	9610	220
9-Feb-04	Computer Associates eTrust InoculateIT For Linux Vulnerabilities	9616	220
9-Feb-04	Multiple Red-M Red-Alert Remote Vulnerabilities	9618	220
10-Feb-04	Microsoft Windows Internet Naming Service Buffer Overflow Vulnerability	9624	220

Table 34 (continued)

10-Feb-04	Multiple Vendor Bluetooth Device Unspecified Information Disclosure Vulnerability	9024	220
10-Feb-04	Microsoft Virtual PC For Mac Temporary File Privilege Escalation Vulnerability	9632	220
10-Feb-04	Microsoft Baseline Security Analyzer Vulnerability Identification Weakness	9634	220
10-Feb-04	Samba Mksmbpasswd.sh Insecure User Account Creation Vulnerability	9637	220
10-Feb-04	Platform Load Sharing Facility LSF_ENVDIR Local Command Execution Vulnerability	7655	220
11-Feb-04	Midnight Commander Virtual File System Symlink Buffer Overflow Vulnerability	8658	220
11-Feb-04	Apache Web Server Multiple Module Local Buffer Overflow Vulnerability	8911	220
11-Feb-04	Linux Kernel 32 Bit Ptrace Emulation Full Kernel Rights Vulnerability	9429	220
11-Feb-04	Gaim Multiple Remote Boundary Condition Error Vulnerabilities	9489	220
11-Feb-04	Novell Groupwise Webaccess Cross Site Scripting Vulnerability	9508	220
11-Feb-04	Util-Linux Login Program Information Leakage Vulnerability	9558	220
11-Feb-04	Linux Kernel R128 Device Driver Unspecified Privilege Escalation Vulnerability	9570	220
12-Feb-04	Sophos Anti-Virus Delivery Status Notification Handling Scanner Bypass Vulnerability	9650	220
12-Feb-04	AIM Sniff Temporary File Symlink Attack Vulnerability	9653	220
12-Feb-04	Libxml2 Remote URI Parsing Buffer Overrun Vulnerability	9718	220
12-Feb-04	Sophos Anti-Virus MIME Header Handling Denial Of Service Vulnerability	9648	220
16-Feb-04	Microsoft Outlook Express Arbitrary Program Execution Vulnerability	9673	220
16-Feb-04	Paul Daniels SignatureDB sdbscan Local Buffer Overflow Vulnerability	9661	220
17-Feb-04	YaBB Information Leakage Weakness	9677	220

Table 34 (continued)

17-Feb-04	APC SmartSlot Web/SNMP Management Card Default Password Vulnerability	9681	220
17-Feb-04	Microsoft Windows XP Help And Support Center Interface Spoofing Weakness	9685	220
18-Feb-04	Microsoft Windows NtSystemDebugControl() Kernel API Function Privilege Escalation Vulnerability	9694	220
18-Feb-04	Linux Kernel execve() Malformed ELF File Unspecified Local Denial Of Service Vulnerability	9695	220
19-Feb-04	Cisco ONS Platform Vulnerabilities	9699	220
20-Feb-04	Oracle9i Lite Multiple Unspecified Vulnerabilities	9704	220
20-Feb-04	Oracle9i Database Server Unspecified Security Vulnerabilities	9705	220
20-Feb-04	Singularity Software Team Factor Integer Handling Memory Corruption Vulnerability	9708	220
21-Feb-04	W3C Jigsaw Unspecified Remote URI Parsing Vulnerability	9711	220
23-Feb-04	Samhain Labs HSFTP Remote Format String Vulnerability	9715	220
24-Feb-04	Multiple Apple Mac OS X Local And Remote Vulnerabilities	9731	220
24-Feb-04	Gigabyte Gn-B46B Wireless Router Authentication Bypass Vulnerability	9740	220
24-Feb-04	Gamespy Software Development Kit Remote Denial Of Service Vulnerability	9741	220
25-Feb-04	Microsoft ASN.1 Library Multiple Stack-Based Buffer Overflow Vulnerabilities	9743	220
25-Feb-04	FreeChat Remote Denial Of Service Vulnerability	9744	220
27-Feb-04	Sun Solaris Unspecified Passwd Local Root Compromise Vulnerability	9757	220
27-Feb-04	Sun Solaris conv_fix Unspecified File Overwrite Vulnerability	9759	220
29-Feb-04	ignitionServer Global IRC Operator Privilege Escalation Vulnerability	9783	220
1-Mar-04	GNU Anubis Multiple Remote Buffer Overflow and Format String Vulnerabilities	9772	220

Table 34 (continued)

2-Mar-04	Hot Open Tickets Unspecified Privilege Escalation Vulnerability	9790	220
2-Mar-04	SureCom Network Device Malformed Web Authorization Request Denial Of Service Vulnerability	9795	220
4-Mar-04	Adobe Acrobat Reader XFDF File Handler Buffer Overflow Vulnerability	9802	220
4-Mar-04	HP Tru64 UNIX Unspecified IPsec/IKE Remote Privilege Escalation Vulnerability	9803	220
5-Mar-04	Seattle Lab Software SLWebMail Multiple Buffer Overflow Vulnerabilities	9808	220
9-Mar-04	IBM WebSphere Unspecified Security Vulnerability	9833	220
9-Mar-04	IBM AIX Rexecd Privilege Escalation Vulnerability	9835	220
19-Jan-04	Multiple JDBC Database Insecure Default Policy Vulnerabilities	9444	221
20-Jan-04	WebTrends Reporting Center Management Interface Path Disclosure Vulnerability	9460	221
20-Jan-04	DUware Software Multiple Vulnerabilities	9462	221
21-Jan-04	Cisco Voice Product IBM Director Agent Unauthorized Remote Administrative Access Vulnerability	9468	221
21-Jan-04	Cisco Voice Product IBM Director Agent Port Scan Denial Of Service Vulnerability	9469	221
21-Jan-04	Microsoft Windows Samba File Sharing Resource Exhaustion Vulnerability	9467	221
21-Jan-04	Apache mod_perl Module File Descriptor Leakage Vulnerability	9471	221
22-Jan-04	Native Solutions TBE Banner Engine Server Side Script Execution Vulnerability	9472	221
22-Jan-04	Sun Solaris modload() Unauthorized Kernel Module Loading Vulnerability	9477	221
23-Jan-04	Finjan SurfinGate FHTTP Restart Command Execution Vulnerability	9478	221
26-Jan-04	Microsoft Windows XP Explorer Self-Executing Folder Vulnerability	9487	221
29-Jan-04	Clearswift MAILsweeper For SMTP RAR Archive Denial Of Service Vulnerability	9556	221

Table 34 (continued)

30-Jan-04	FreeBSD mksnap_ffs File System Option Reset Vulnerability	9533	221
6-Feb-04	Linux Kernel do_brk Function Boundary Condition Vulnerability	9138	221
6-Feb-04	Linux Kernel do_mremap Function Boundary Condition Vulnerability	9356	221
9-Feb-04	Shaun2k2 Palmhttpd Server Remote Denial of Service Vulnerability	9608	221
9-Feb-04	Microsoft Internet Explorer LoadPicture File Enumeration Weakness	9611	221
9-Feb-04	Microsoft Windows XP HCP URI Handler Arbitrary Command Execution Vulnerability	9621	221
10-Feb-04	Nokia Bluetooth Device Unauthorized Access Vulnerability	9032	221
10-Feb-04	Microsoft Internet Explorer Shell: IFrame Cross-Zone Scripting Vulnerability	9628	221
10-Feb-04	Microsoft Internet Explorer Double-Null URI Denial Of Service Vulnerability	9629	221
12-Feb-04	XFree86 Unspecified Vulnerability	9655	221
12-Feb-04	XFree86 CopyISOLatin1Lowered Font_Name Buffer Overflow Vulnerability	9652	221
13-Feb-04	Microsoft Internet Explorer Unspecified CHM File Processing Arbitrary Code Execution Vulnerability	9658	221
17-Feb-04	Ipswitch IMail Server Remote LDAP Daemon Buffer Overflow Vulnerability	9682	221
18-Feb-04	Linux Kernel do_mremap Function VMA Limit Local Privilege Escalation Vulnerability	9686	221
18-Feb-04	Linksys WAP55AG SNMP Community String Insecure Configuration Vulnerability	9688	221
18-Feb-04	Linux Kernel Vicam USB Driver Userspace/Kernel Memory Copying Weakness	9690	221
21-Feb-04	Jabber Software Jabber Gadu-Gadu Transport Multiple Remote Denial Of Service Vulnerabilities	9710	221
25-Feb-04	Mozilla Browser Zombie Document Cross-Site Scripting Vulnerability	9747	221

Table 34 (continued)

25-Feb-04	MTools MFormat Privilege Escalation Vulnerability	9746	221
26-Feb-04	PerfectNav Malformed URI Denial Of Service Vulnerability	9753	221
26-Feb-04	eXtremail Authentication Bypass Vulnerability	9754	221
27-Feb-04	Microsoft Internet Explorer Cross-Domain Event Leakage Vulnerability	9761	221
27-Feb-04	FreeBSD Unauthorized Jailed Process Attaching Vulnerability	9762	221
1-Mar-04	Motorola T720 Phone Denial Of Service Vulnerability	9779	221
5-Mar-04	Norton AntiVirus 2002 ASCII Control Character Denial Of Service Vulnerability	9811	221
6-Mar-04	Norton AntiVirus 2002 Nested File AutoProtect Bypass Vulnerability	9814	221
6-Mar-04	Apple Safari Large JavaScript Array Handling Denial Of Service Vulnerability	9815	221
9-Mar-04	Microsoft MSN Messenger Information Disclosure Vulnerability	9828	221
9-Mar-04	Apache Mod_Access Access Control Rule Bypass Vulnerability	9829	221
9-Mar-04	Apache Mod_SSL HTTP Request Remote Denial Of Service Vulnerability	9826	221
9-Mar-04	Confixx Perl Debugger Remote Command Execution Vulnerability	9831	221
9-Mar-04	F-Secure SSH Server Password Authentication Policy Evasion Vulnerability	9824	221
9-Mar-04	WU-FTPD restricted-gid Unauthorized Access Vulnerability	9832	221
9-Mar-04	IBM DFSMS/MVS Tape Utility Unspecified Vulnerability	9834	221
9-Mar-04	Microsoft Outlook Mailto Parameter Quoting Zone Bypass Vulnerability	9827	221
5-Feb-04	SqWebMail Authentication Response Information Leakage Weakness	9541	222
5-Feb-04	Crossday Discuz! Cross Site Scripting Vulnerability	9584	222
11-Feb-04	HP-UX NLSPATH Environment Variable Format String Vulnerability	8985	222

Table 34 (continued)

27-Jan-04	TCPDump ISAKMP Decoding Routines Denial Of Service Vulnerability	9507	223
7-Feb-04	Multiple Vendor Network Device Driver Frame Padding Information Disclosure Vulnerability	6535	223
16-Feb-04	Computer Associates eTrust Antivirus Malicious Code Detection Bypass Vulnerability	9665	223
17-Feb-04	Snort Signature Mislabeling Weakness	9683	223
24-Feb-04	Digital Reality Game Engine Remote Denial Of Service Vulnerability	9736	223
25-Feb-04	Alcatel OmniSwitch 7000 Series Security Scan Denial Of Service Vulnerability	9745	223
26-Feb-04	Internet Security Systems Protocol Analysis Module SMB Parsing Heap Overflow Vulnerability	9752	223
27-Feb-04	Apple Mac OS X Apple Filing Protocol Client Multiple Vulnerabilities	9763	223
2-Mar-04	Nortel Wireless LAN Access Point 2200 Series Denial Of Service Vulnerability	9787	223
2-Mar-04	SonicWall Firewall/VPN Appliance Multiple ARP Request Handling Vulnerabilities	9789	223
2-Mar-04	BSD Out Of Sequence Packets Remote Denial Of Service Vulnerability	9792	223
4-Mar-04	Cisco Content Service Switch Management Port UDP Denial Of Service Vulnerability	9806	223
6-Mar-04	NFS-Utils rpc.mountd Denial Of Service Vulnerability	9813	223
9-Mar-04	Microsoft Windows Media Services Remote Denial of Service Vulnerability	9825	223
27-Jan-04	BEA WebLogic Operator/Admin Password Disclosure Vulnerability	9501	224
27-Jan-04	BEA WebLogic Server and Express SSL Client Privilege Escalation Vulnerability	9502	224
27-Jan-04	BEA WebLogic Server/Express Potential Administrator Password Disclosure Weakness	9503	224

Table 34 (continued)

27-Jan-04	BEA WebLogic Incorrect Operator Permissions Password Disclosure Vulnerability	9505	224
27-Jan-04	Bodington Uploaded File Disclosure Vulnerability	9528	224
28-Jan-04	Inlook Unauthorized User Password File Access Vulnerability	9527	224
30-Jan-04	JBrowser Unauthorized Admin Access Vulnerability	9537	224
31-Jan-04	SqWebMail Authentication Response Information Leakage Weakness	9541	224
31-Jan-04	Aprox Portal File Disclosure Vulnerability	9540	224
4-Feb-04	RealOne Player SMIL File Script Execution Variant Vulnerability	9378	224
7-Feb-04	OpenJournal Authentication Bypassing Vulnerability	9598	224
10-Feb-04	Caucho Technology Resin Source Code Disclosure Vulnerability	9614	224
10-Feb-04	Caucho Technology Resin Directory Listings Disclosure Vulnerability	9617	224
16-Feb-04	mnoGoSearch UdmDocToTextBuf Buffer Overflow Vulnerability	9667	224
23-Feb-04	Platform Load Sharing Facility EAuth Component Buffer Overflow Vulnerability	9719	224
23-Feb-04	Platform Load Sharing Facility EAuth Privilege Escalation Vulnerability	9724	224
24-Feb-04	Apple Mac OS X PPPD Format String Memory Disclosure Vulnerability	9730	224
24-Feb-04	Working Resources BadBlue Server phptest.php Path Disclosure Vulnerability	9737	224
2-Mar-04	Symantec Firewall/VPN Appliance Cached Plaintext Password Vulnerability	9784	224
4-Mar-04	DAWKCo POP3 with WebMAIL Extension Session Timeout Unauthorized Access Vulnerability	9807	224
9-Mar-04	LionMax Software Chat Anywhere User IP Address Obfuscation Vulnerability	9823	224
23-Feb-04	nCipher Hardware Security Module Firmware Secrets Disclosure Vulnerability	9717	225

Table 34 (continued)

28-Jan-04	OracleAS TopLink Mapping Workbench Weak Encryption Algorithm Vulnerability	9515	226
10-Feb-04	EvolutionX Multiple Remote Buffer Overflow Vulnerabilities	9631	11, 8
13-Feb-04	Sami FTP Server Multiple Denial Of Service Vulnerabilities	9657	39, 40
28-Jan-04	DotNetNuke Multiple Vulnerabilities	9518	9, 13
14-Feb-04	Multiple ASP Portal Vulnerabilities	9659	9, 13
16-Feb-04	EarlyImpact ProductCart Multiple Vulnerabilities	9669	9, 13
17-Feb-04	Ecommerce Corporation Online Store Kit More.PHP Multiple Vulnerabilities	9676	9, 13
23-Feb-04	XMB Forum Multiple Input Validation Vulnerabilities	9726	9, 13
4-Mar-04	Multiple Vendor HTTP Response Splitting Vulnerability	9804	a report
27-Jan-04	Multiple Apple Mac OS X Operating System Component Vulnerabilities	9504	see 9509
	220 = Not enough information		
	221 = specific to vendor		
	222 = Did not understand the attack		
	223 = networking		
	224 = Does not fit this approach		
	225 = hardware vulnerability		
	226 = encryption		

Appendix II

Feasibility Assignment

New attacks to software may not be predictable, but a software team should not permit recurrences of old attacks in new software. Many of the vulnerabilities in this assignment are repeated (in some cases dozens of times) in the Bugtraq database (http://www.securityfocus.com). By acknowledging that a system design contains the same vulnerable component sequences in previous software applications, developers can be warned about possible attacks in their future code. This assignment encourages security to be built into the software application instead of being added at the end of the software process as an afterthought.

The following terms and background information will be useful for the assignment.

Terms

1. Security policy - formal statement of the rules by which people who are given access to an organization's technology and information assets must abide [4].

2. Access policy (a subcomponent of a security policy) - access rights and privileges to protect assets from loss or disclosure by specifying acceptable use guidelines for users, operations staff, and management. It should provide guidelines for external connections, data communications, connecting devices to a network, and adding new software to systems [4].

3. Security – From the terms 1 & 2, security can be defined as enforcing the rules defined in a security policy that describes access to resources. A security breach occurs when a component maliciously or undesirably accesses a resource [1].

Background

The OCTAVE framework created by the Software Engineering Institute (SEI) at Carnegie Mellon University suggests that all stakeholders in a software application should be involved in evaluating security risks. It is therefore necessary for software vulnerabilities to be recognized by developers, security experts, marketers, and customers. In this way, non-experts of a system (including developers) should be able to quickly predict security vulnerabilities at the onset of the software process. One approach is to define error patterns that are used to find vulnerabilities in a system. This assignment will be used to evaluate the human-readable aspect of representing security vulnerabilities with regular expressions (regex for short). Regexs are utilized because they can be adapted into a programmable form for automation and because they attempt to be environment/language independent [2].

Directions

Twenty vulnerable component sequences have been abstracted into regular expressions to conceptualize the access paths used in known attacks. Each path contains at least one access control violation that risks the use of a resource in the system. **Determine what access control violation scenarios are *achievable* in the given system design.**

1. Search the system design for the string (i.e. the sequence of components) that is defined by each regular expression listed in Column 1 of Table 1.
2. Enter your findings in the blank table cells. In Column 2, write down the number associated with each component in the sequence path (e.g. 1-2-3) that corresponds to the regular expression (there may be more than one string in the design for a

given regular expression). Document any assumptions about the system you think are necessary to qualify your answer in Column 3. Your assumptions would, in practice, be considered in the access policy (not given). Not all strings may be present in the design and multiple attacks may exploit the same string. The last five cells of Column 5 are an optional, creative exercise that asks for a potential scenario of the access control violation. Use the information in Column 4 as a hint to answering Column 5.

Interpreting a Regular Expression
Each regex represents a sequence of events that are involved with data flow between components. For example, (Client$^+$)(Server$^+$)(Log$^+$)(Hard Drive$^+$) shows a series of client requests, followed by a series of server actions, followed by a series of log updates, followed by a series of disk writes.

Example
Servers often log each request a user makes into an access log. If the server is not properly implemented, an attacker could send millions of requests to the server and flood the hard drive with access log entries. The primary components involved in the attack are the client, server, access log and hard drive. The vulnerable components are represented by the regular expression (Client$^+$)(Server$^+$)(Log$^+$)(Hard Drive$^+$). The given system design shows that Client 1(component #14) can make requests to the Web Server (#12) via the web pages (#13). The Service Client Request routine (#11) is responsible for processing the HTTP headers and payloads of all the requests made of the server. The request routine enters the information into the Access Log (#6), which is stored on the Hard Drive (#7). Therefore, the sequence of components in the design that matches (Client$^+$)(Server$^+$)(Log$^+$)(Hard Drive$^+$) is #14, #13, #12, #11, #6, #7 (14-13-12-11-6-7). Components #13 and #11 are intermediate components not shown in the regex because they are only stepping stones involved in the attack. The database environment with components #10, #9, #6, #7 (10-9-6-7) reflect the same regular expression (thus the same vulnerability), but without the web environment specific components, #13 and #11. How about the component sequence 1-2-6-7? Is this feasible? If you make the assumption that a user (or script) can log into the authentication server millions of times without tripping an intrusion detection alarm and proper software checks are not in place, then the vulnerability may exist. Remember that this approach takes place in the design phase (before coding has started) so many assumptions can be made about how the system will be implemented.

Excessively
Vital Regex Notes:

1. The regexs are abstract and can therefore be fulfilled in different ways (e.g. Server implies Web Server or Database Server)
2. Some regexs will be straightforward and others will require assumptions. Software designs do not often show many important low level details, so document <u>any</u> assumptions necessary about how an attack may occur.
3. Regexs only show the main components involved in a vulnerability. You may assume that a vulnerability exists if there are intermediate components/processes in the design sequence. Include the intermediate components in your answers for Column 2.
4. One component may be responsible for <u>consecutive</u> multiple events in a regex (e.g. the Service Client Request routine may read a packet stream and write to a buffer).

You may represent the repetition as something like 1-2-2 or simply 1-2 (in this example, component #2 carries out two consecutive events.)

5. Review of regular expressions
    a. (A*) (Kleene closure) implies event A occurs 0 or more times.
    b. (A^+) implies event A occurs one or more times.
    c. (A + B) implies event A or event B occurs.

Vital System Design Notes:

1. Not all resources (e.g. hard drives, memory (including buffers)) are shown in the design because of page size constraints.
2. Arrows represent data flow.
3. Circles represent processes/methods.
4. “Client” and “user” are synonymous.
5. Assume the “Service Client Request” routine contains all the necessary methods for processing HTTP headers and Get & Post Methods.

Grading: Column 2 of Table 1 answers are 20 points (total). Partial credit will be awarded based on assumptions in Column 3. Feedback questions are 5 points (total).

Please send all comments/questions to mcgegick@ncsu.edu.

References

[1] J. Viega and G. McGraw, *Building Secure Software: How to Avoid Security Problems the Right Way*. Boston, MA: Addison-Wesley,
2002.

[2] M. Howard and D. LeBlanc, *Writing Secure Code*. Redmond, WA: Microsoft Press, 2003.

[3] http://securityfocus.com

[4] http://www.sei.cmu.edu/

Feedback

An assignment of this type has never been given before in CSC 405. Please provide feedback that will help us refine the assignment for next semester.

1. What parts of the assignment are not clear?

2. How much time did you spend answering the questions in Table 1?

3. The last five rows of Table 1 do not have explanations available in Column 5. Did you find the information in Column 5 necessary to determine an attack?

4. Rate the approach of identifying security vulnerabilities and give a brief explanation.

    1. Poor
    2. Below average
    3. Average
    4. Good
    5. Excellent

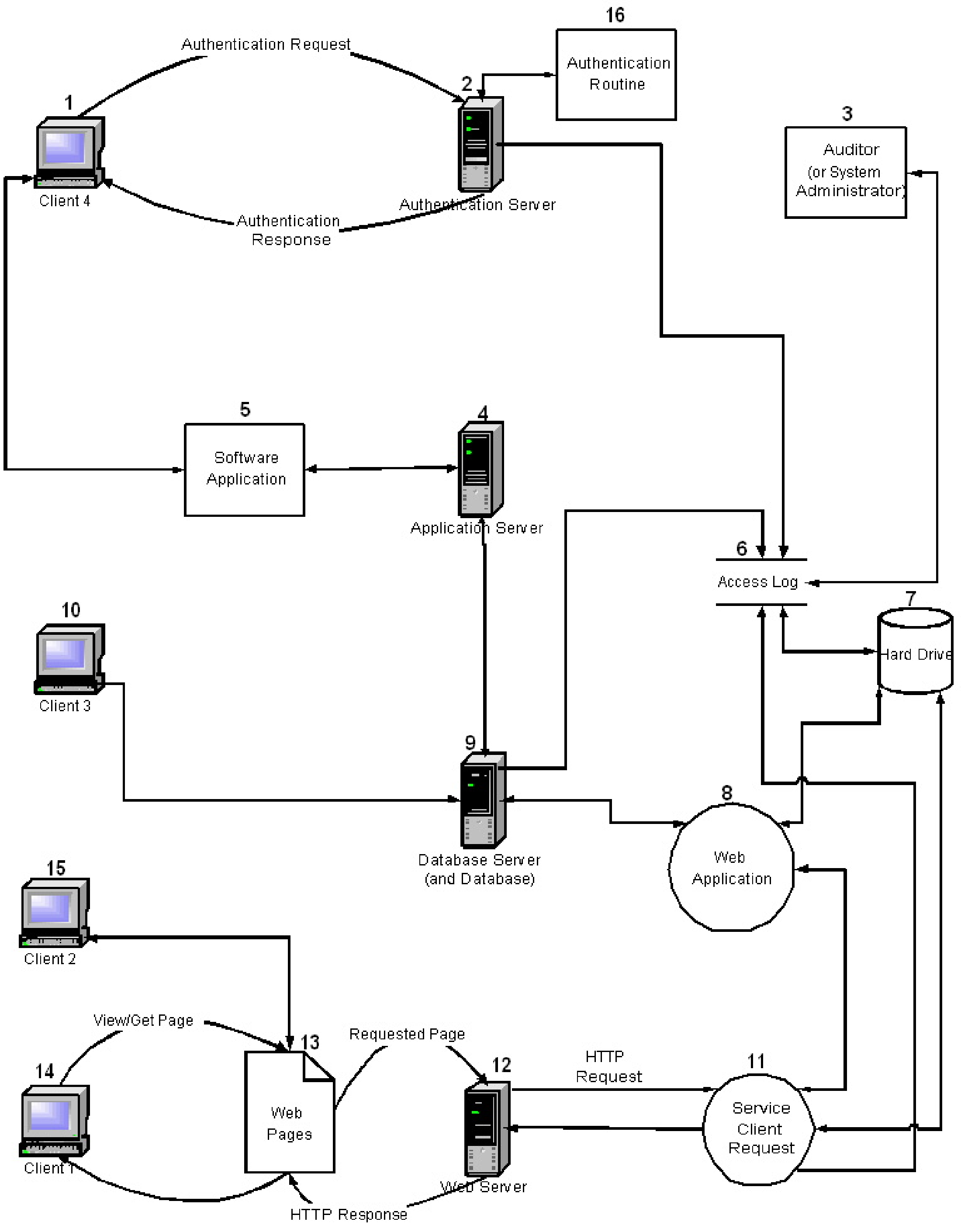

16
Authentication Routine
Authentication Request
2
Authentication Server
1
Client 4
Authentication Response
3
Auditor (or System Administrator)
5
Software Application
4
Application Server
6
Access Log
7
Hard Drive
10
Client 3
9
Database Server (and Database)
8
Web Application
15
Client 2
View/Get Page
13
Web Pages
Requested Page
12
Web Server
HTTP Request
11
Service Client Request
14
Client 1
HTTP Response

Table 1

Column 1	Column 2	Column 3	Column 4	Column 5
Regular Expression	**Component Sequence**	**Assumptions**	**Type of Attack**	**Attack Scenario**
(Client$^+$)(Server$^+$)(Log$^+$)(Hard Drive$^+$)	14-13-12-11-6-7 10-9-6-7	There is no check to determine how large the access log file becomes.	Boundary Condition Error	An attacker can exceedingly access either the web server or database server augmenting the access log file and eventually filling the hard drive causing the system to crash, a denial-of-service attack (DoS).
(Client$^+$)(Server$^+$)(HTTPMessageHeaderHandler$^+$)			Failure to Handle Exceptional Conditions	A client may send a message with thousands of headers, causing a denial-of-service.
(Client)(Server)(Get Method)(GetMethod BufferWrite)			Boundary Condition Error	Writing an excessively long Get Request into a small buffer will cause a buffer overflow.
(Client)(Server)((GetMethod)+(PostMethod))(PayloadValue BufferWrite)(WebApp)			Boundary Condition Error	Data (e.g params in a query string) that are sent to a software application are too large for their intended buffers.
(Client)(Server)(HTTPMessage)(HeaderFieldBufferWrite)			Boundary Condition Error	Excessive header field values written into small buffers will cause a buffer overflow.
(Client)(Server)(HTTPMessageHandler			Boundary Condition	A system administrator can

)(Log)(Sysadmin)(LogEntryRead)			Error	induce a buffer overflow when viewing an excessively long log entry.
(Client)(Server)(PostMethod)(HTTPContent-LengthHeaderValue)(HTTPMessagePayloadLength)(ServerConnectionState)			Failure to Handle Exceptional Conditions	Sending a value via the Post method in the Content-Length of the HTTP header not equal to the content-length may cause the socket to stay open (DoS)
(User)(UserNameEntry)(PasswordEntry)(AuthenticationServer*)(AuthenticationRoutine)			Boundary Condition Error	Writing an excessively long string of characters for either the username or password into a small buffer will cause a buffer overflow.
(Client)(SQLInputField)(Server)(WebApp)(Database)			Input Validation Error	Failure to sanitize user input can allow a user to submit any SQL query, thus allowing for unauthorized access to data.
(Client)(SQLInputField)(Server)(WebApp)(Database)			Failure to Handle Exceptional Conditions	An attacker may submit a malicious SQL query (such as a Cartesian join of all tables) consuming the CPU.
(Client)(HTMLForm)(WebApp)(Server)(cgihtml)(FileSystem)			Input Validation Error	A bug in cgihtml will allow a user's form-data to overwrite specified files in the victim's file system

(User)(CommandLineArgumentEntry)(Application)(ApplicationServer*)(CommandLineArgumentBufferWrite)			Boundary Condition Error	A user may enter excessively long command line parameters causing buffer overflows.
(Client)(HTMLPage)(Server)(Hard Drive)			Boundary Condition Error	A user may submit an excessive amount of data in an HTML page, thus filling up the server's hard drive.
(Client)(HTMLMessageBoard)(Server)(HTMLMessageBoard)(Client)			Input Validation Error	An attacker may exploit this vulnerability by including hostile HTML and script code in posts to a message board. This code may be rendered in the web browser of a user who views message.
(User)(Machine)(SyslogFunction)(Log)			Input Validation Error	It is possible to corrupt memory by passing format strings through the Syslog(), a logging function. This may potentially be exploited to overwrite arbitrary locations in memory with attacker-specified values.
(User)(ReadUserInput)(EnvironmentVariableWrite)			Input Validation Error	

(User)(GUI/Browser)(BookMarkSave)(BookmarkBufferWrite)			Boundary Condition Error	
(User)(File)(FileRead)(BufferWrite)			Boundary Condition Error	
(SocketRead)(SocketBufferWrite)			Boundary Condition Error	
(Class)(Subclass)(OverriddenSecuredMethods)			Access Validation Error	(Hint: Occurred in a Netscape java implementation that used java.lang.ClassLoader)

Descriptions of Errors

Boundary Condition Error

A boundary condition error occurs when:

1. A process attempts to read or write beyond a valid address boundary. (e.g. buffer overflow)
2. A system resource is exhausted.
3. An error results from an overflow of a static-sized data structure. This is a classic buffer overflow condition.

Access Validation Error

An access validation error occurs when:

1. A subject invokes an operation on an object outside its access domain.
2. An error occurs as a result of reading or writing to/from a file or device outside a subject's access domain.
3. An error results when an object accepts input from an unauthorized subject.
4. An error results because the system failed to properly or completely authenticate a subject.

Input Validation Error

An input validation error occurs when:

1. An error occurs because a program failed to recognize syntactically incorrect input.
2. An error results when a module accepted extraneous input fields.
3. An error results when a module failed to handle missing input fields.
4. An error results because of a field-value correlation error.

Failure to Handle Exceptional Conditions

1. An error manifests itself because the system failed to handle an exceptional condition generated by a functional module, device, or user input.

[4]

Appendix III

Time Spent on the Feasibility Study

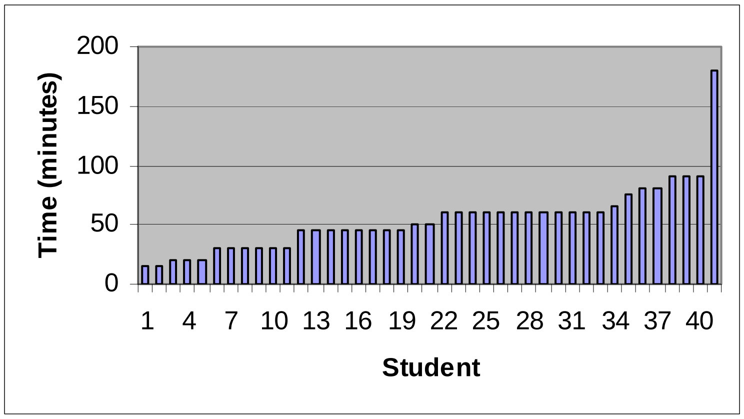


Figure 12: Time Spent on the Feasibility Study

Data Table

Table 35: Time Spent on the Feasibility Study.

Time (minutes)	Number Students
15	2
20	3
30	6
45	8
50	2
60	12
65	1
75	1
80	2
90	3
180	1

Appendix IV

Valid and Invalid Attack Paths for the Feasibility Study

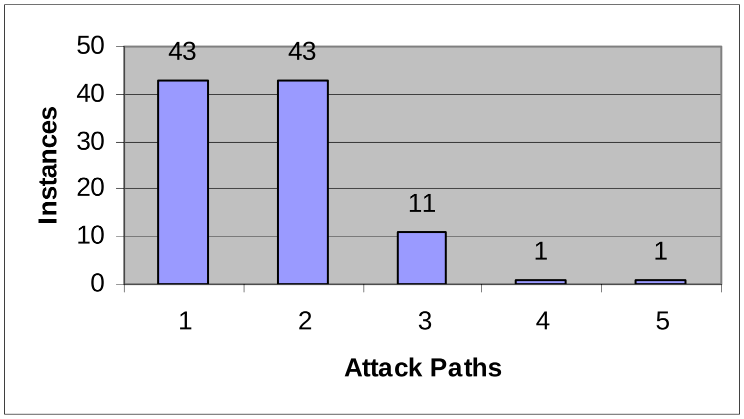


Figure 13: Valid Attack Paths for Regex1.

(No invalid answers reported)

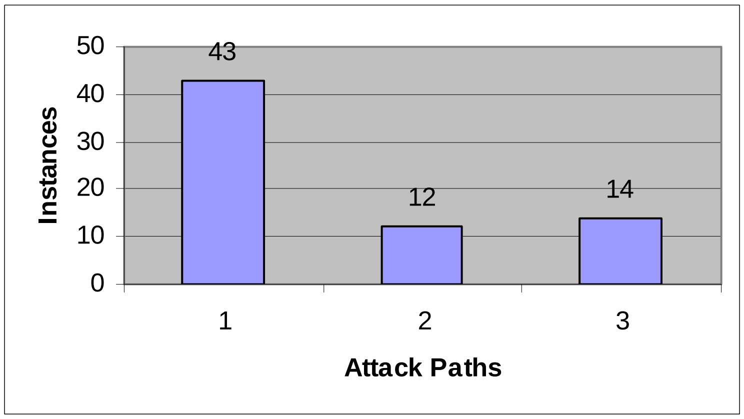


Figure 14: Valid Attack Paths for Regex2.

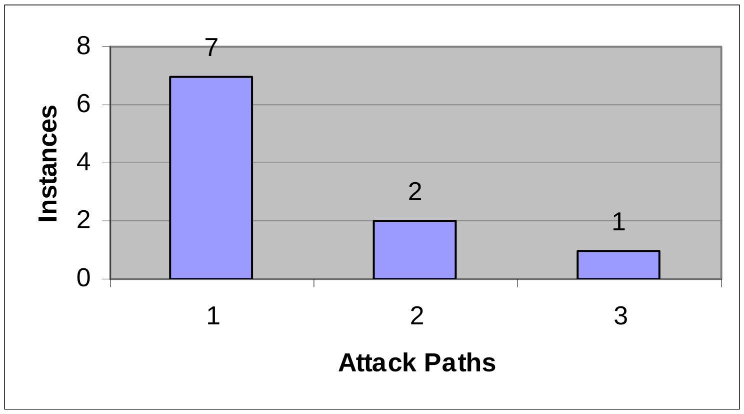


Figure 15: Invalid Attack Paths for Regex2

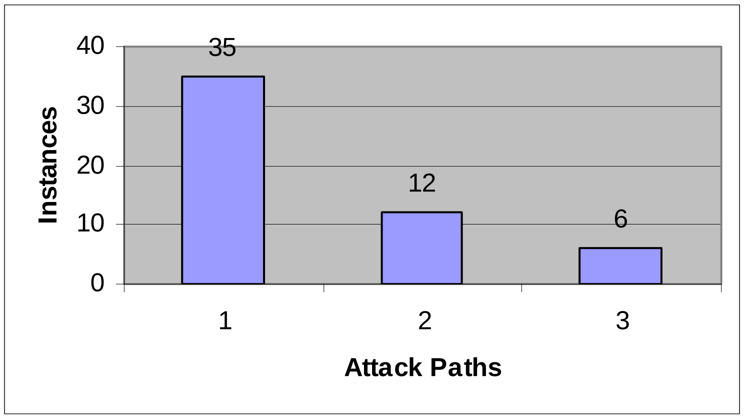


Figure 16: Valid Attack Paths for Regex3.

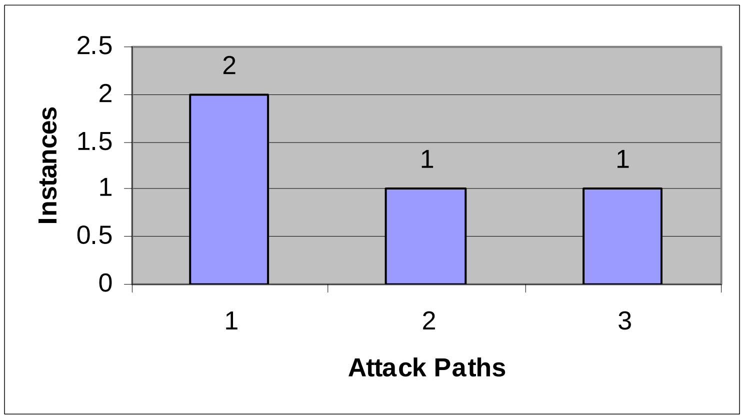


Figure 17: Invalid Attack Paths for Regex3.

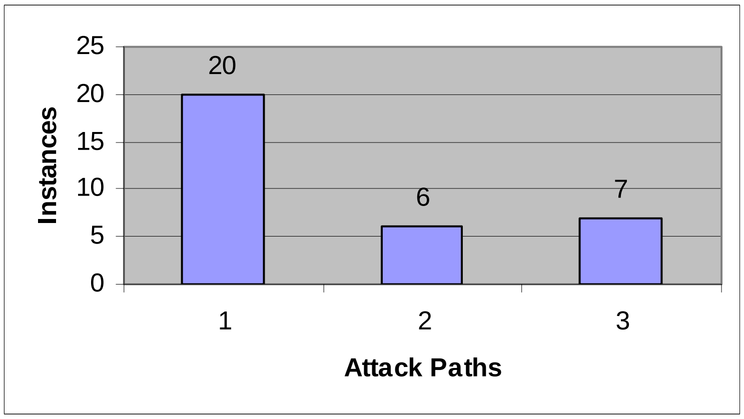


Figure 18: Valid Attack Paths for Regex4.

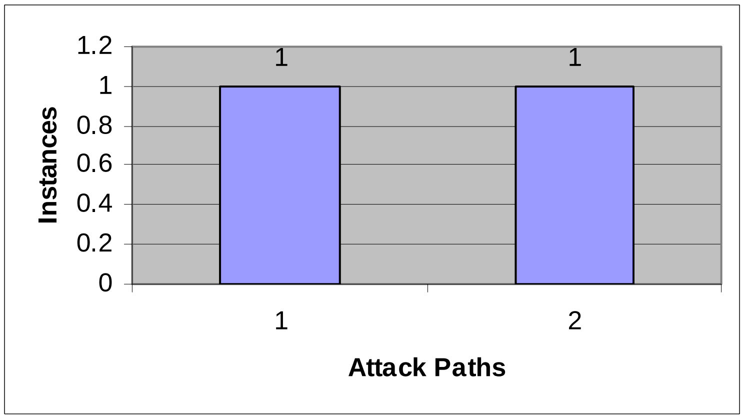


Figure 19: Invalid Attack Paths for Regex4.

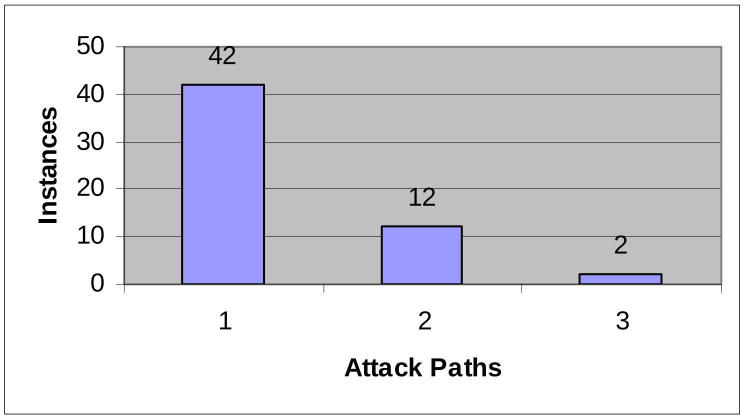


Figure 20: Valid Attack Paths for Regex5.

(No invalid answers reported)

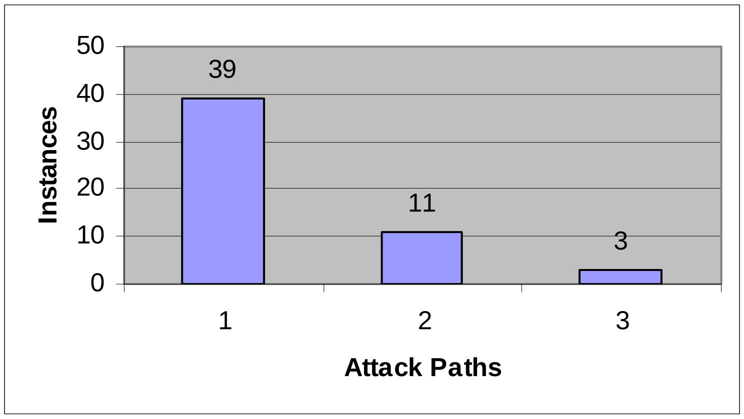


Figure 21: Valid Attack Paths for Regex6.

(No invalid answers reported)

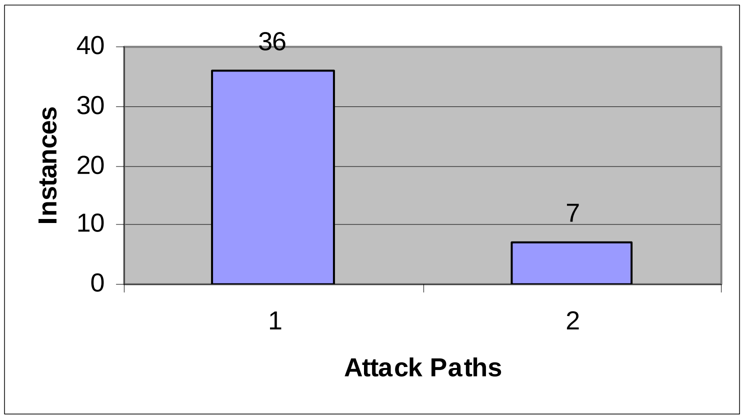


Figure 22: Valid Attack Paths for Regex7.

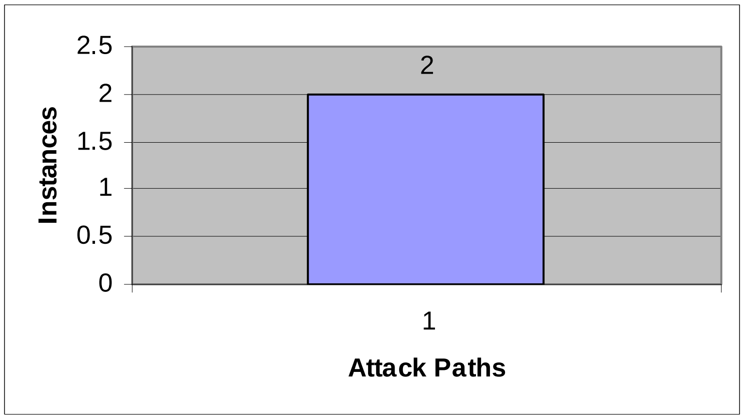


Figure 23: Invalid Attack Paths for Regex7.

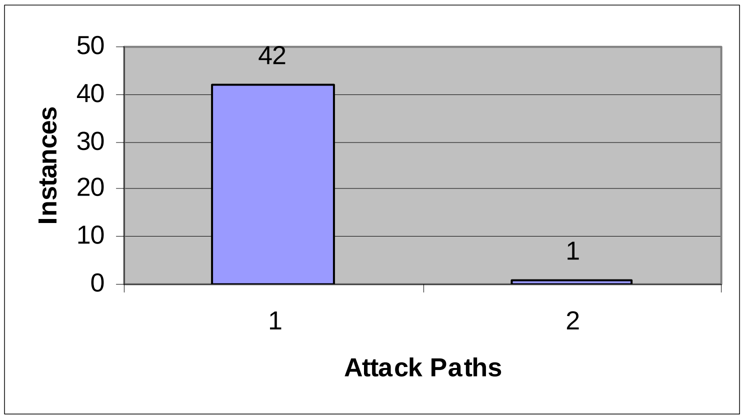


Figure 24: Valid Attack Paths for Regex8.

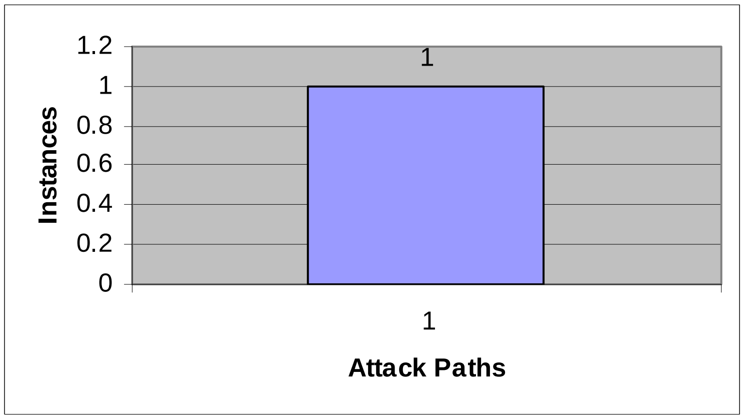


Figure 25: Invalid Attack Paths for Regex8.

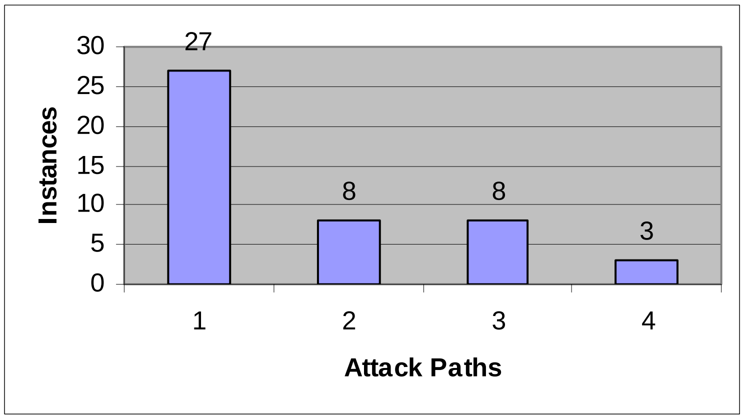


Figure 26: Valid Attack Paths for Regex9.

(No invalid answers reported)

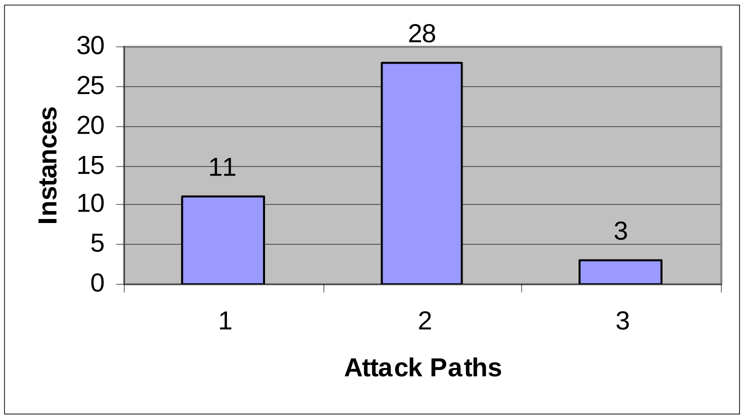


Figure 27: Valid Attack Paths for Regex10.

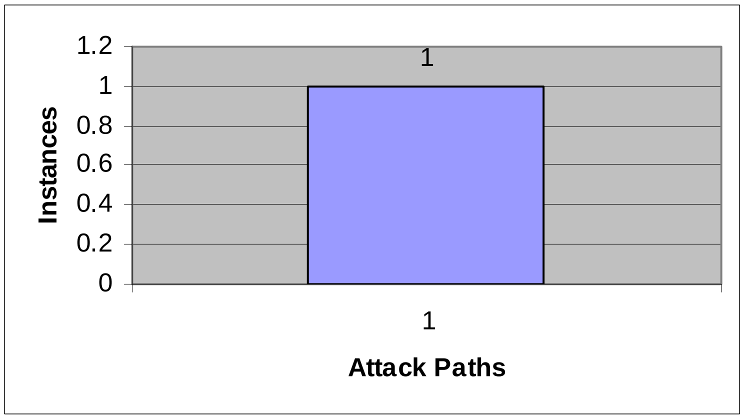


Figure 28: Invalid Attack Paths for Regex10.

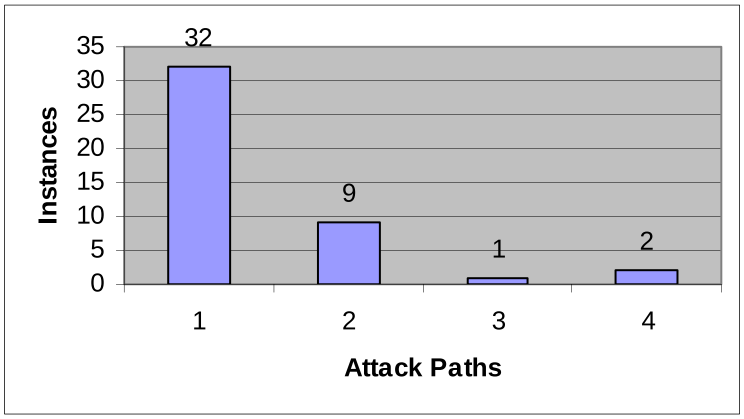


Figure 29: Valid Attack Paths for Regex11.

(No invalid answers reported)

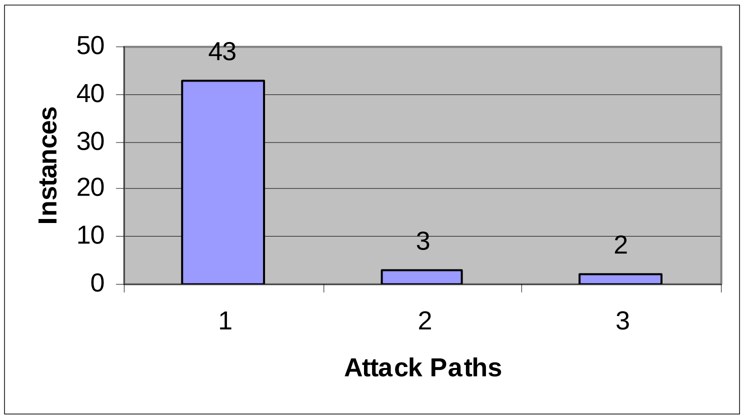


Figure 30: Valid Attack Paths for Regex12.

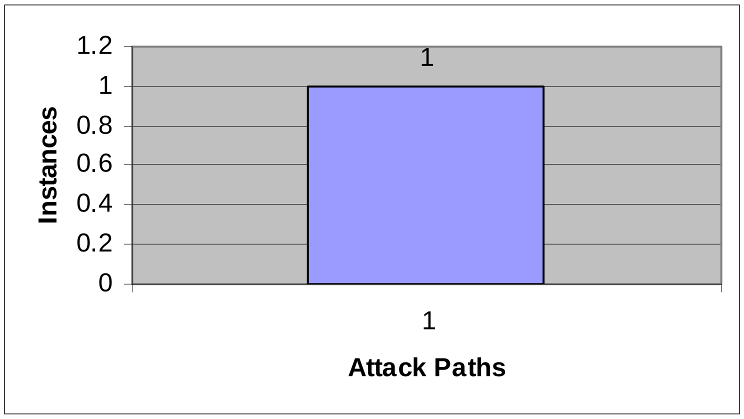


Figure 31: Invalid Attack Paths for Regex12.

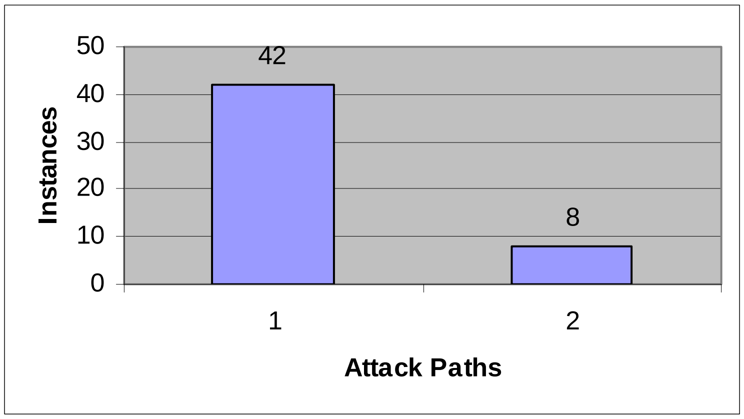


Figure 32: Valid Attack Paths for Regex13.

(No invalid answers reported)

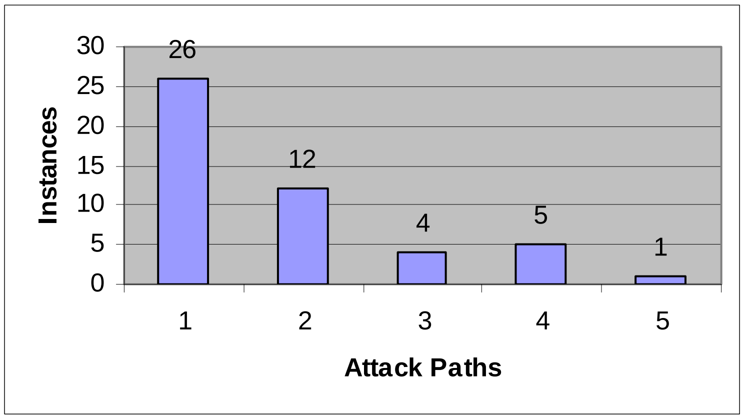


Figure 33: Valid Attack Paths for Regex14.

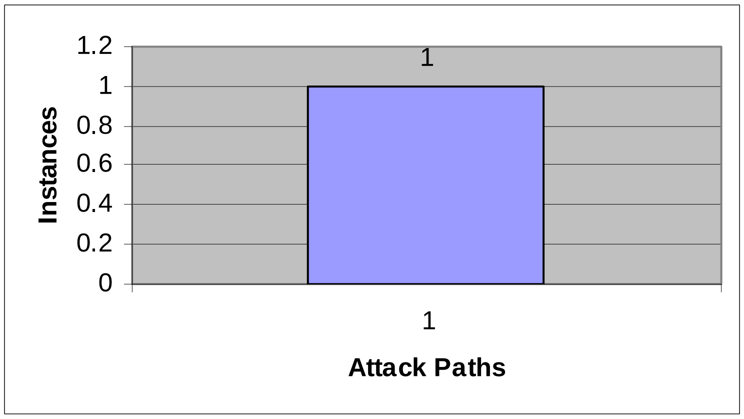


Figure 34: Invalid Attack Paths for Regex14.

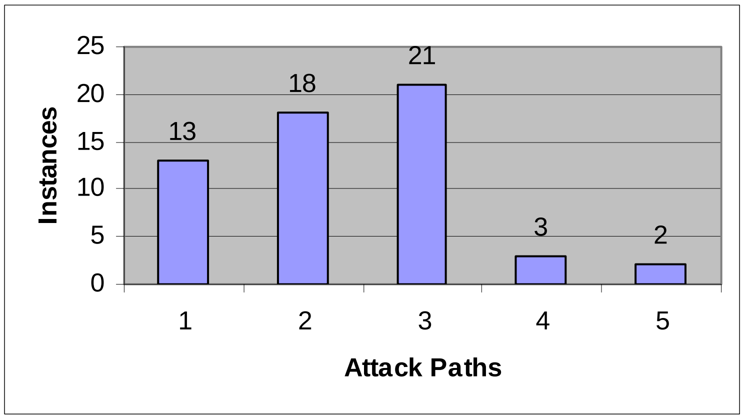


Figure 35: Valid Attack Paths for Regex15.

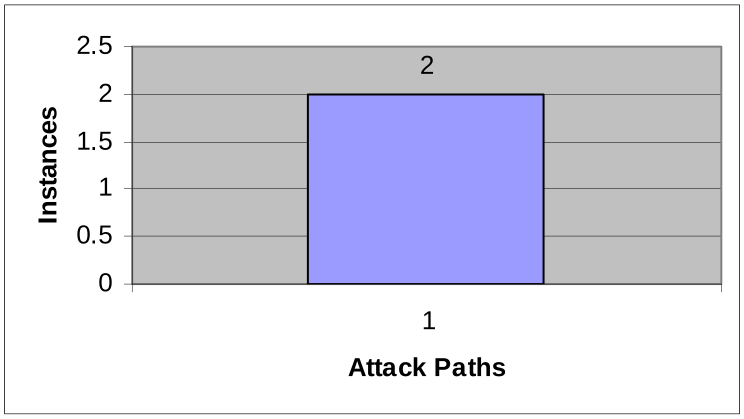


Figure 36: Invalid Attack Paths for Regex15.

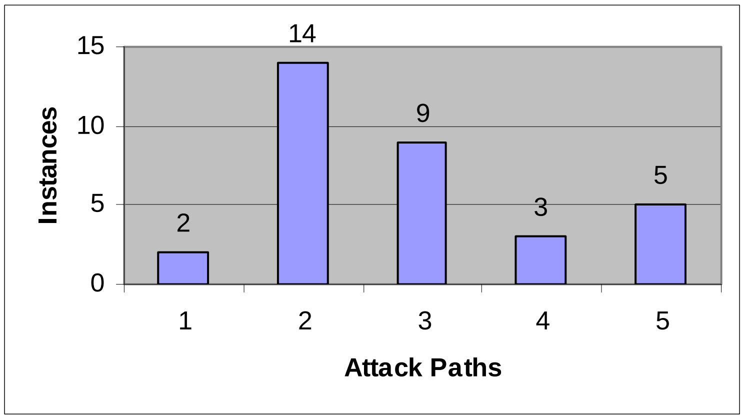


Figure 37: Valid Attack Paths for Regex16.

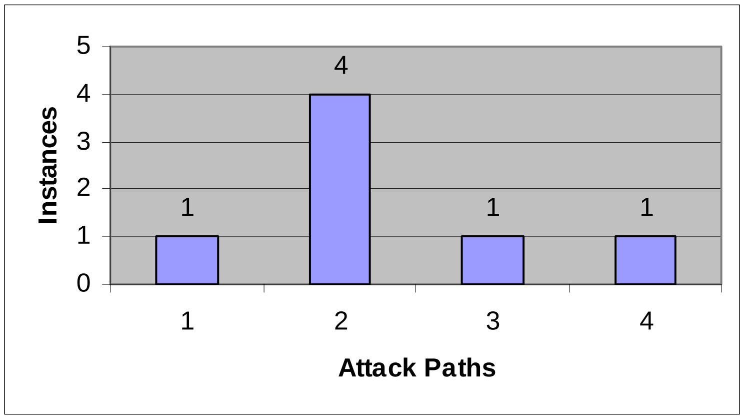


Figure 38: Invalid Attack Paths for Regex16.

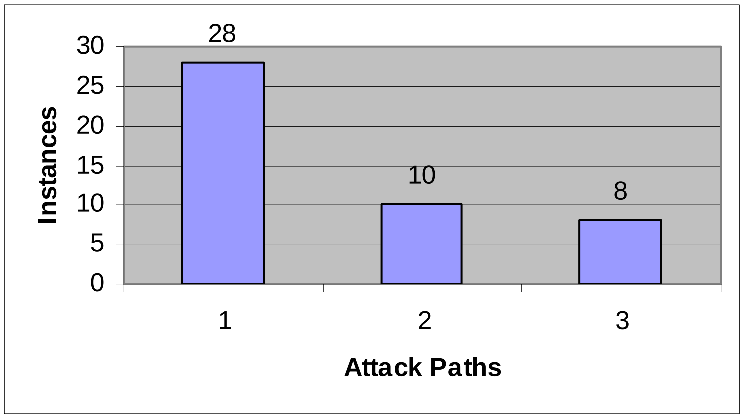


Figure 39: Valid Attack Paths for Regex17.

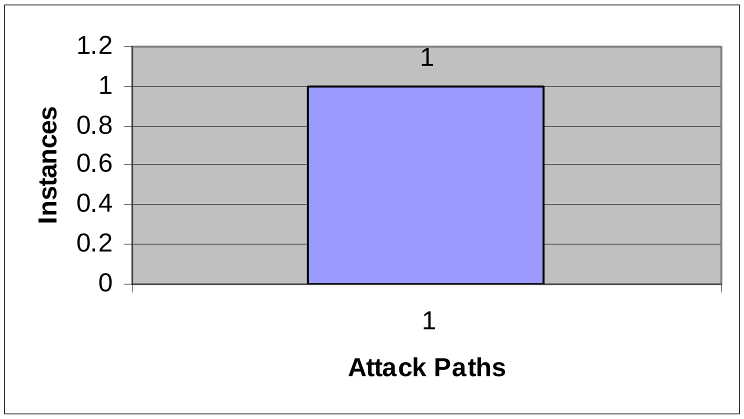


Figure 40: Invalid Attack Paths for Regex17.

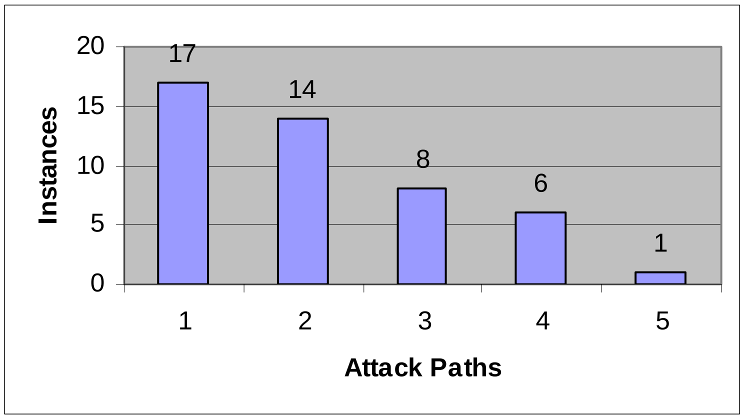


Figure 41: Valid Attack Paths for Regex18.

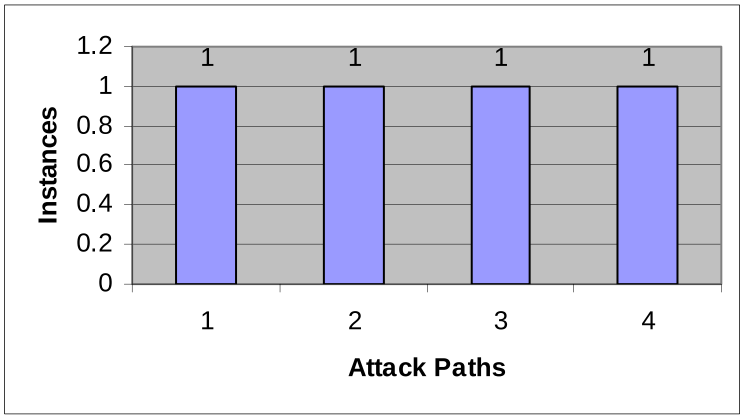


Figure 42: Invalid Attack Paths for Regex18.

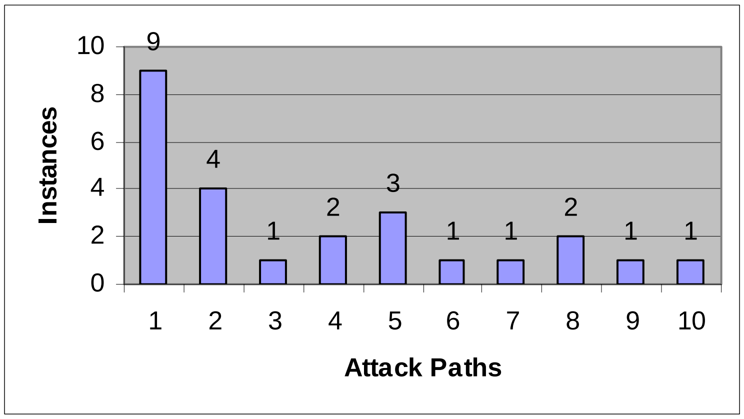


Figure 43: Valid Attack Paths for Regex19.

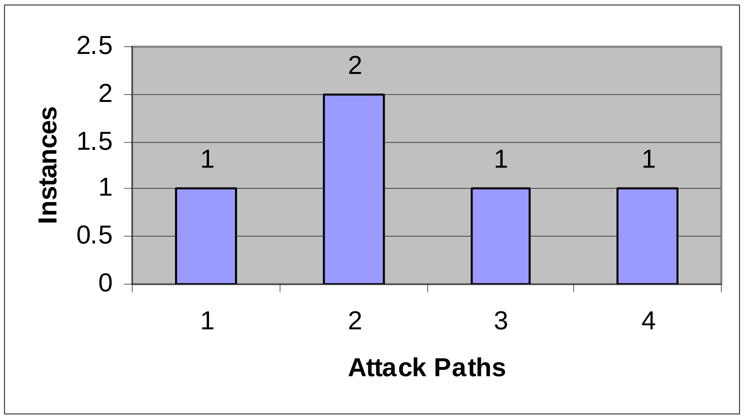


Figure 44: Invalid Attack Paths for Regex19.

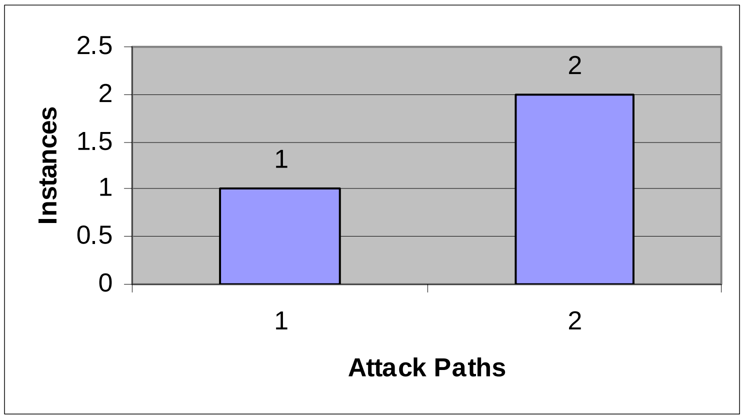


Figure 45: Valid Attack Paths for Regex20.

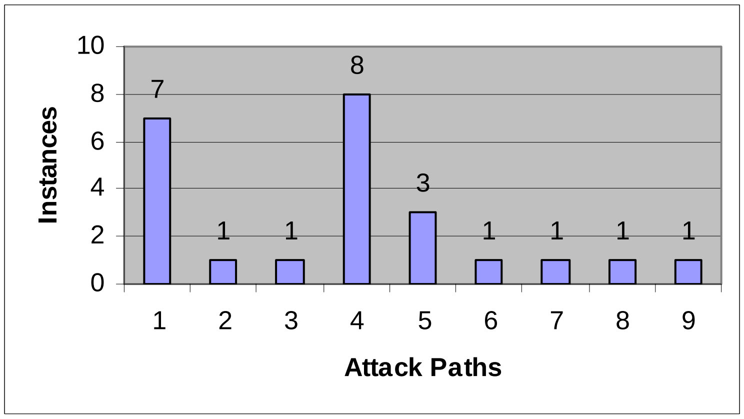


Figure 46: Invalid Attack Paths for Regex20.

Appendix V

Feasibility Study Likert Scale

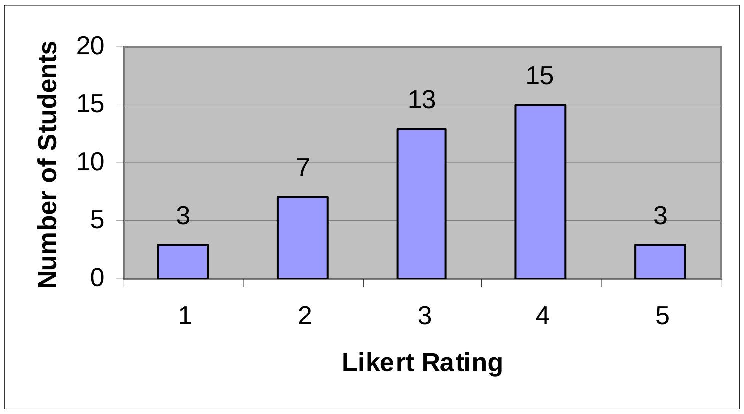


Figure 47: Feasibility Study Likert Scale

Appendix VI

Student Comments on the Feasibility Study

The Approach is Bad

1. The expression scan apply to many scenarios, but in this example the terminology is confusing and does not represent the diagram very well. If this approach was automated it could be quite effective, but doing it by hand is tedious, especially in large networks.
2. This was annoying.
3. Method seems like its trying too hard to be rigorous (using regexs) when the issue is too grey to be as precise

Students are Not Qualified to Determine the Validity of the Approach.

1. Don't know enough about industry to understand the effectiveness. Not really sure I know what other approaches are.
2. I have nothing to compare it against

The Approach is Good

1. It was a good idea of how network components work together and how to exploit vulnerabilities.
2. Because it is based upon everyone's past experience
3. Because anyone can use this approach to find security vulnerabilities
4. I would say that based on the given path to look for they were not all that hard to visualize based on the given diagrams.

The Approach has Limitations

1. Good attempt, and the graph help a lot to see the whole picture and identify risks that may be overlooked. And the regular expressions help direct the attack. However, it still seems like a system that misses the goal of use throughout the development process since deployment of products is rarely, if ever, static.
2. Diagrams of real systems will be much more complicated and dynamic. There are many more paths that would have to be checked. Plus, even after taking the incredible amount of time to find all possible vulnerable paths in a system, you would still have to go and traverse each of those paths to determine if there is actually a vulnerability. Some paths may be difficult to travel.
3. It seems to do well with what has been identified, but a more thorough, systematic approach (start with one point and explore all reasonable paths) might be more beneficial.
4. Seems good, but it only finds known vulnerabilities.

The Approach is Too Hard

1. This assignment seems a bit too complicated for 405.

The Approach is Advantageous to Software Engineering

1. If exercised early in design process, you can prevent errors before they happen, later it is just as valuable to hack the problems out.
2. This approach sees like it would be extremely powerful if time were made for it in the design cycle. Also, latter releases would be able to incorporate the lessons learned from the previous release.
3. This is a good way of identifying security holes because it's designed in such high level that everyone can get involved in security issues.

There is Not Enough Information to Understand the Approach

1. If I had prior knowledge of networking via a networking class, then this exercise would've been more effective. My lack of network knowledge limits what I take from this assignment.
2. It's a good idea but takes a great deal of predefined knowledge about the system. For instance, some of the portions of the regular expression are unclear as to where exactly in the network they occur.
3. Without specific occurrences and solid understanding of the diagram + regex it is very difficult to identify security vulnerabilities.
4. It is a good approach but it needs to have better definition for regular expression and what all the servers do.
5. We need a better understanding of each component in the system to actually find these vulnerabilities.
6. Too many assumptions are required. Students who don't have experience with this will have a difficult time. Clearer directions are needed.
7. The approach would be food/excellent if I knew more specifics of what the individual components did.
8. I'm sure this is a real good method of determining vulnerabilities if we only knew what each of the objects in the regular expressions meant.
9. It is hard to identify vulnerabilities without listing the specific capabilities of each machine.
10. If it was labeled better.
11. Excellent approach but poor implementation (the diagram is poorly labeled).

Students Did Not Understand the Approach

1. Not completely clear of the whole picture and exactly what is happening. A little confusing… Structure of homework very poor. Need to arrange things in a better/clearer order as to what is what and explain (exactly) what is to be done.
2. This is definitely a great approach to finding security vulnerabilities, but more detailed examples would have helped me deliver more of what you wanted. Explain your two examples more.

Appendix VII
Time Spent on the Validation Study

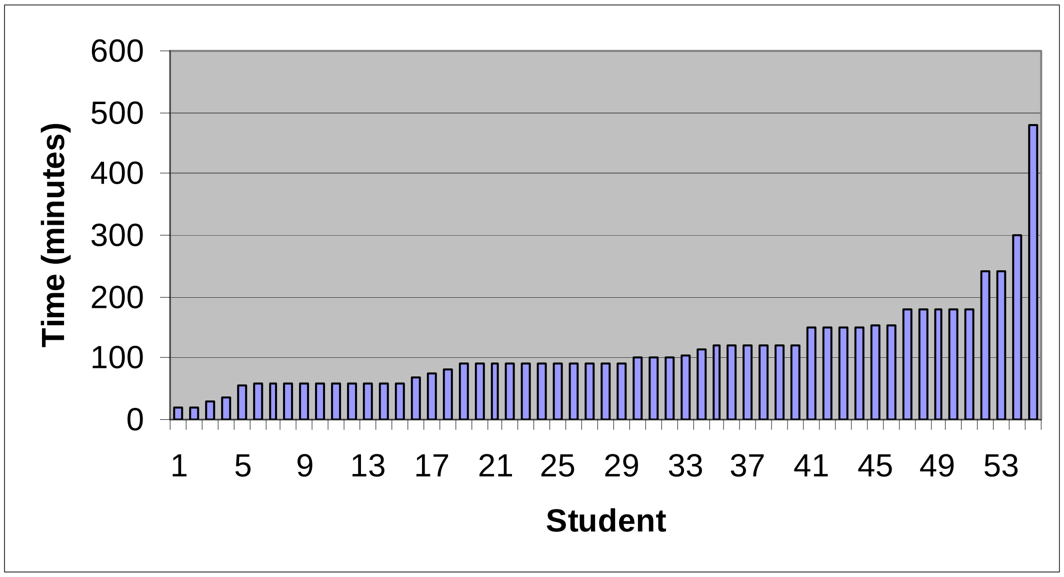


Figure 48: Time Spent on the Validation Study

Data Table

Time (minutes)	Number Students
20	2
30	1
35	1
55	1
60	10
70	1
75	1
80	1
90	11
100	3
105	1
115	1
120	6
150	4
154	2
180	5
240	2
300	1
480	1

Figure 49: Time Spent on the Validation Study

Appendix VIII

Valid and Invalid Attack Paths for the Validation Study

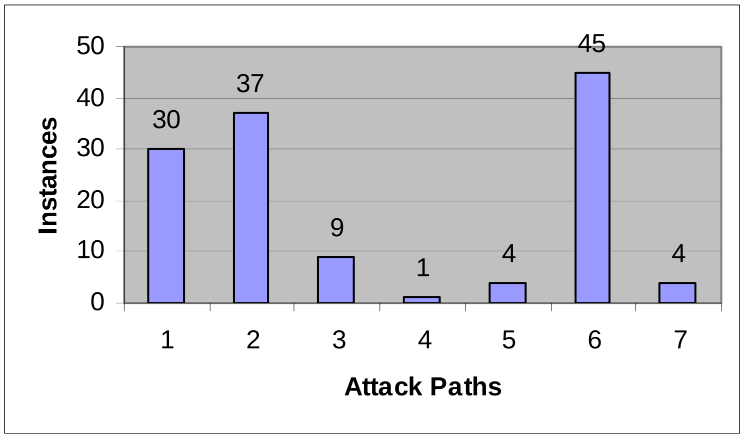


Figure 50: Valid Attack Paths for Regex1.

(No Invalid Attack Paths Reported)

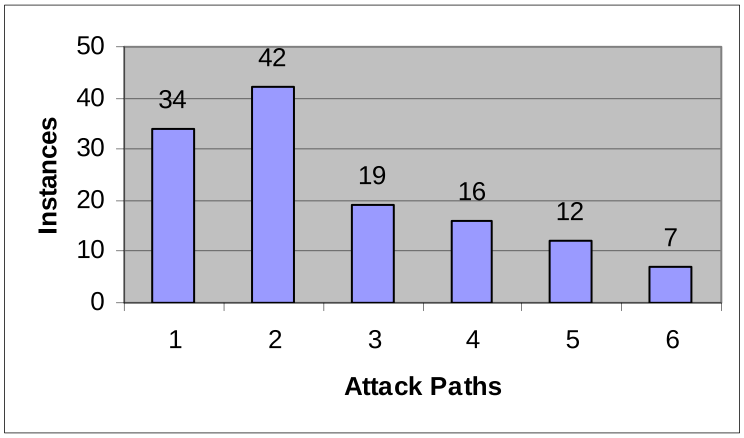


Figure 51: Valid Attack Paths for Regex2.

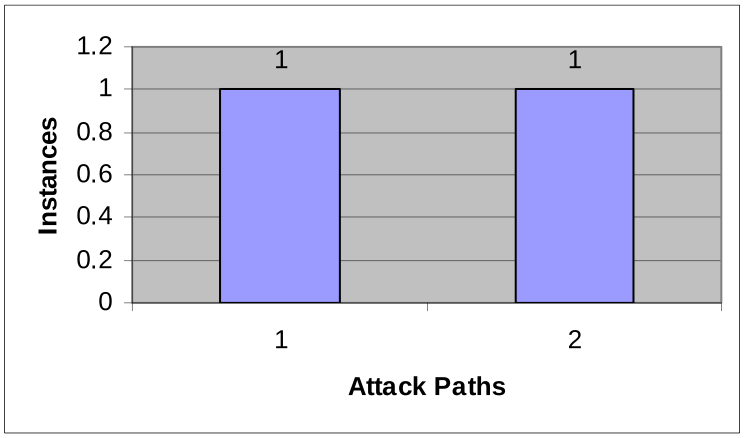


Figure 52: Invalid Attack Paths for Regex2.

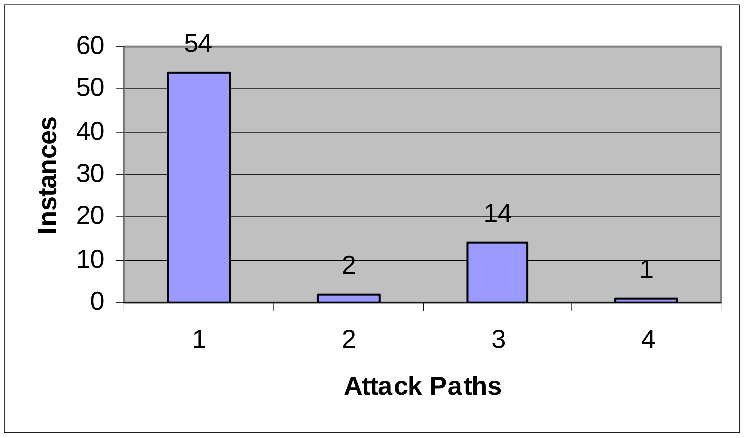


Figure 53: Valid Attack Paths for Regex3.

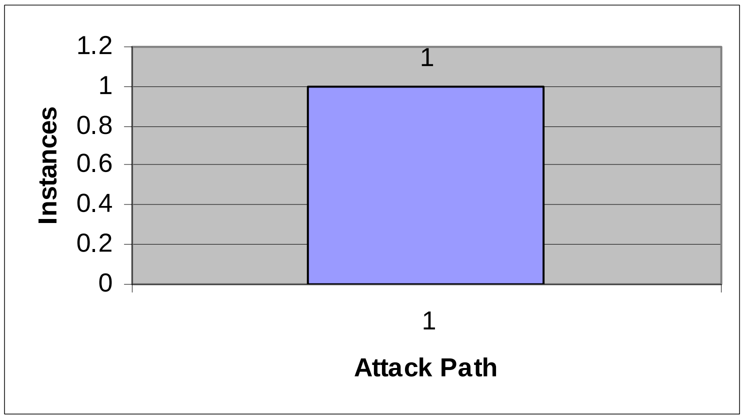


Figure 54: Invalid Attack Paths for Regex3.

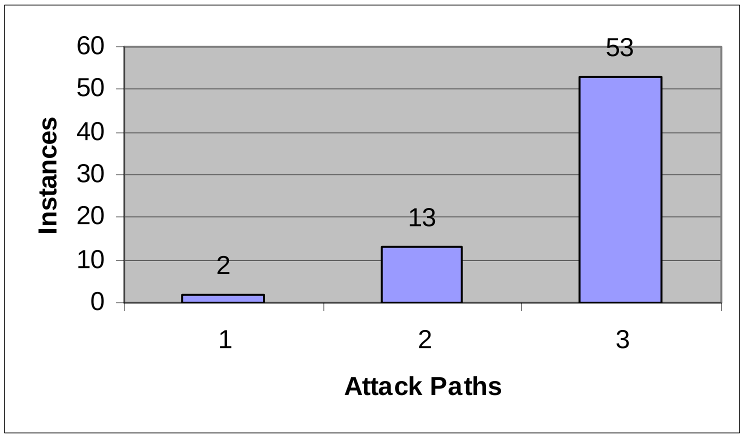


Figure 55: Valid Attack Paths for Regex4.

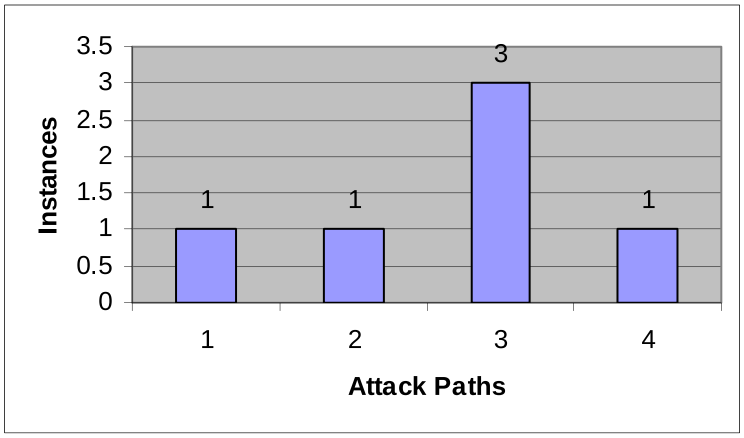


Figure 56: Invalid Attack Paths for Regex4.

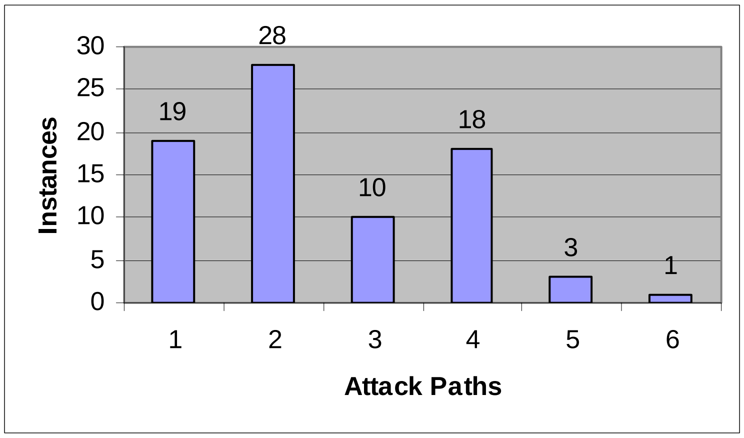


Figure 57: Valid Attack Paths for Regex5.

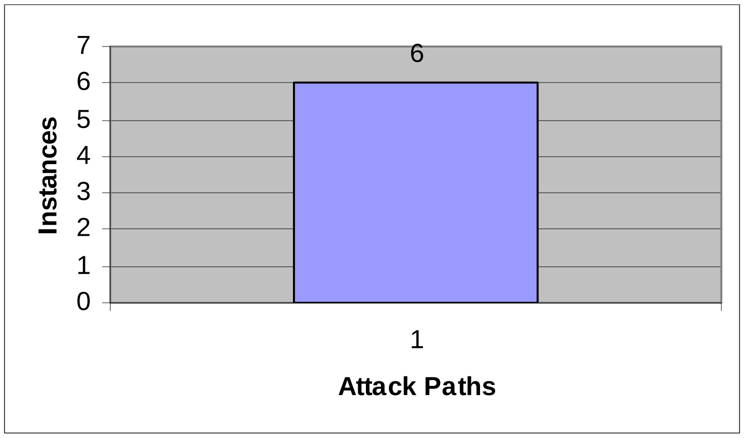


Figure 58: Invalid Attack Paths for Regex5.

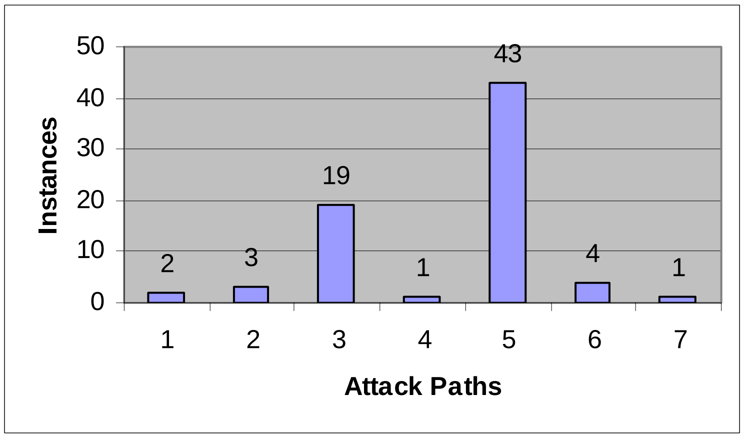


Figure 59: Valid Attack Paths for Regex6.

(No Invalid Attack Paths Reported)

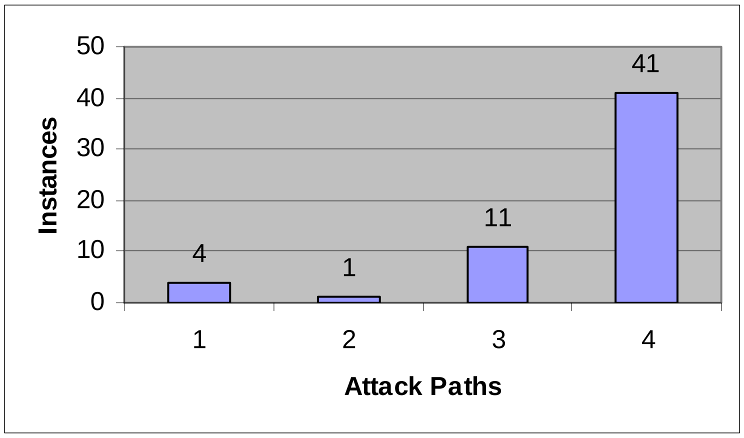


Figure 60: Valid Attack Paths for Regex7.

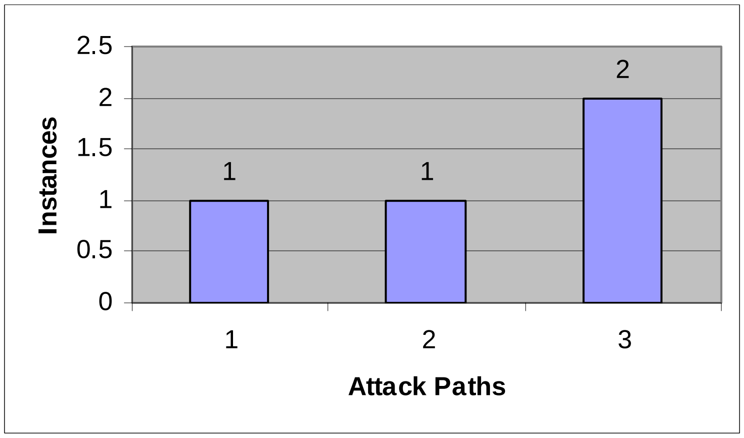


Figure 61: Invalid Attack Paths for Regex7.

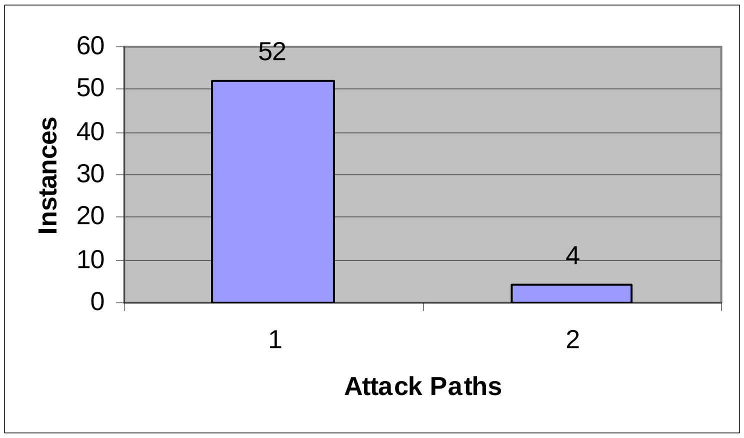


Figure 62: Valid Attack Paths for Regex8.

(No Invalid Attack Paths Reported)

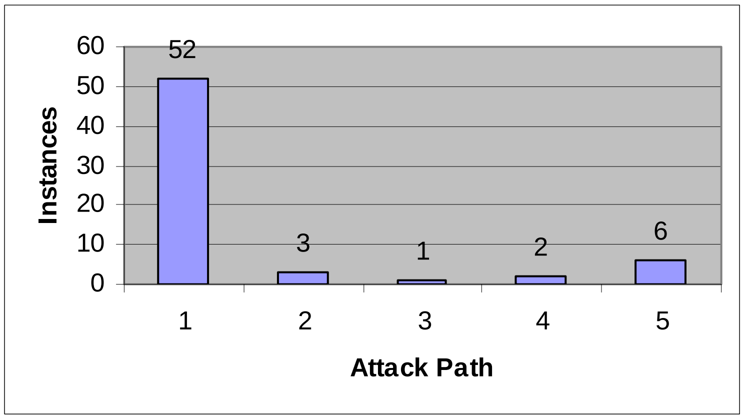


Figure 63: Valid Attack Paths for Regex9.

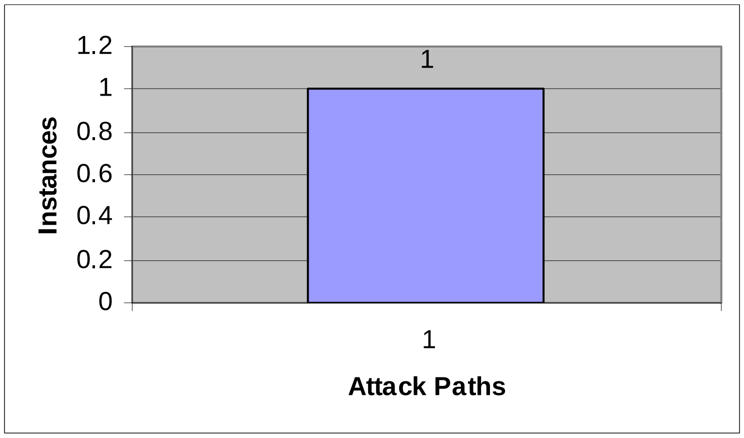


Figure 64: Invalid Attack Paths for Regex9.

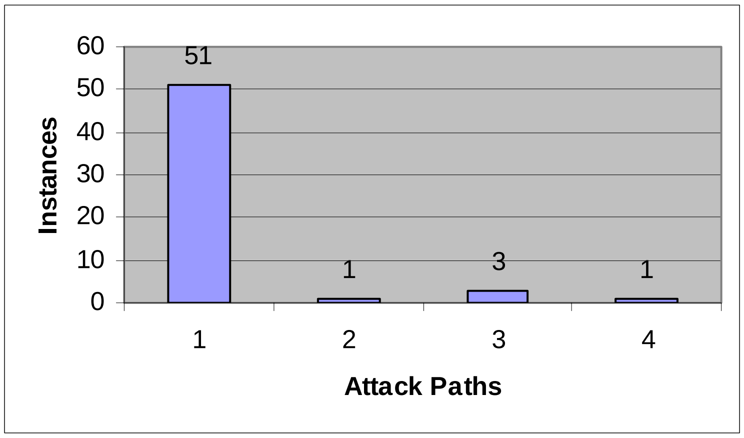


Figure 65: Valid Attack Paths for Regex10.

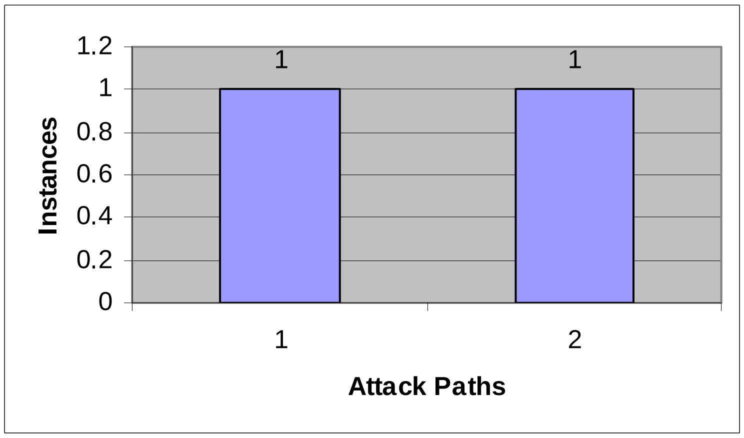


Figure 66: Invalid Attack Paths for Regex10.

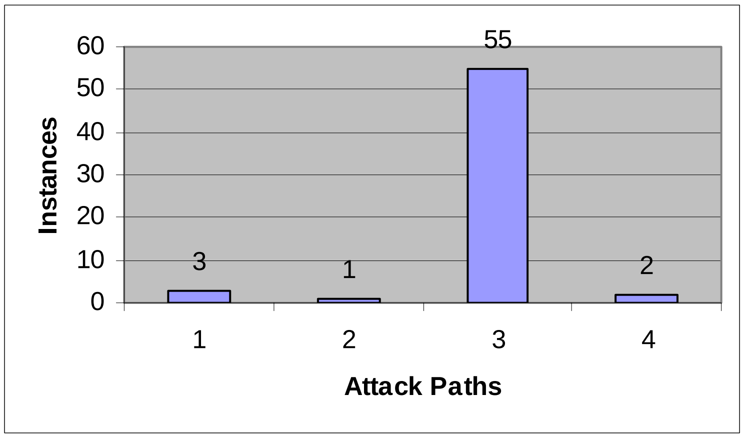


Figure 67: Valid Attack Paths for Regex11.

(No Invalid Attack Paths Reported)

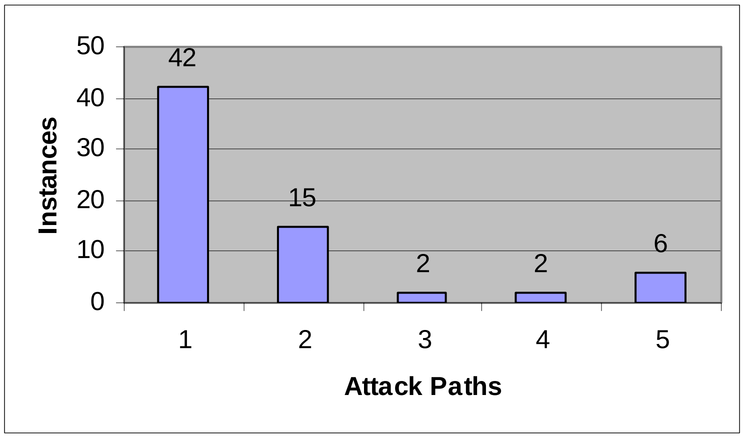


Figure 68: Valid Attack Paths for Regex12.

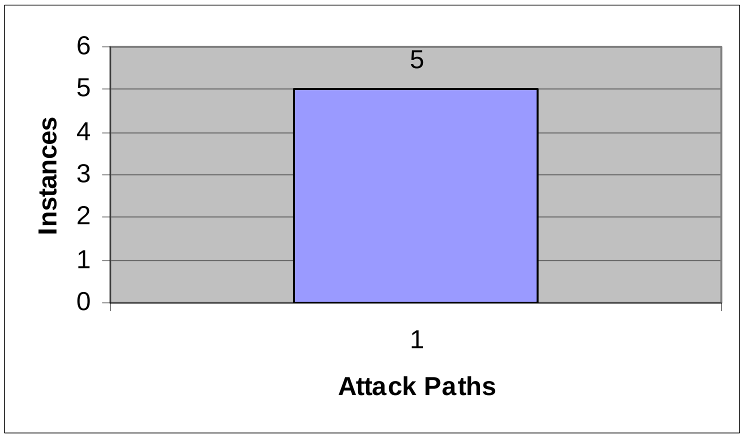


Figure 69: Invalid Attack Paths for Regex12.

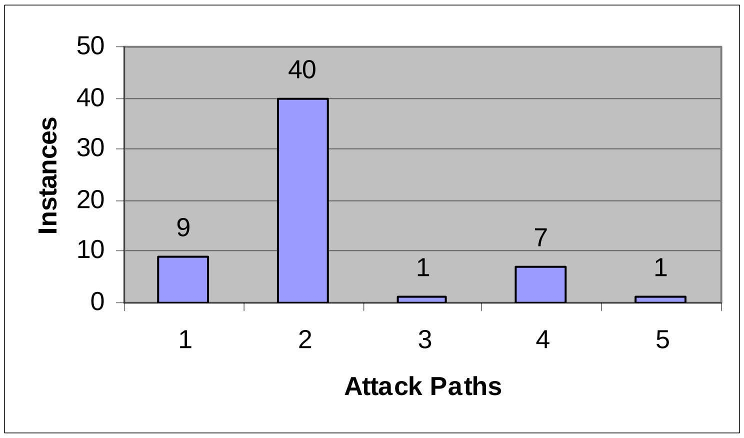


Figure 70: Valid Attack Paths for Regex13.

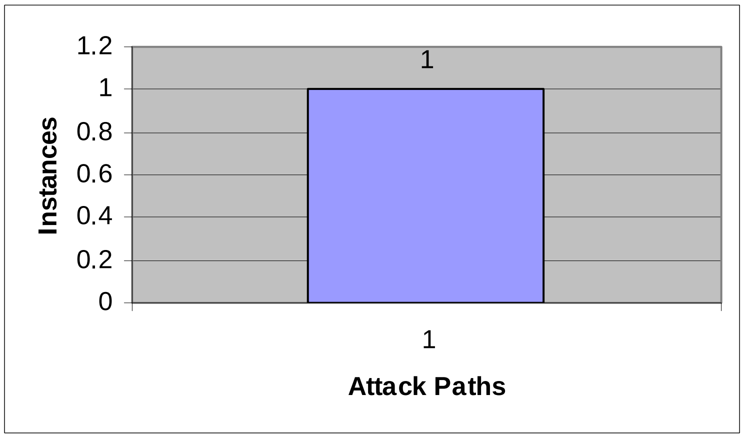


Figure 71: Invalid Attack Paths for Regex13.

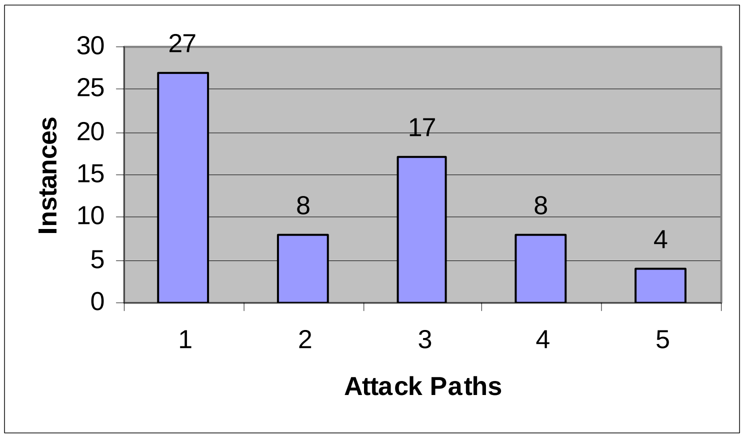


Figure 72: Valid Attack Paths for Regex14.

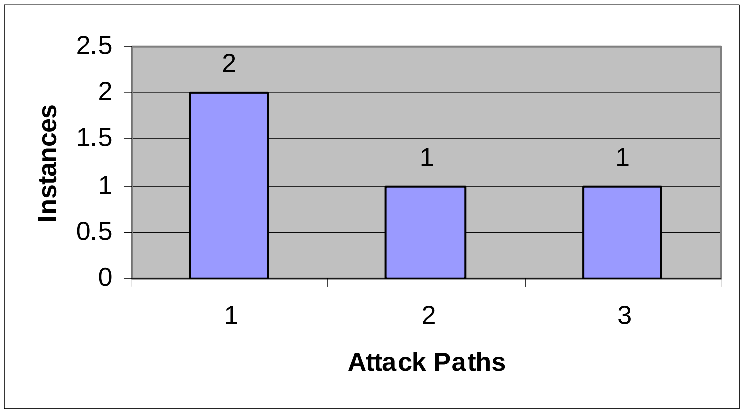


Figure 73: Invalid Attack Paths for Regex14.

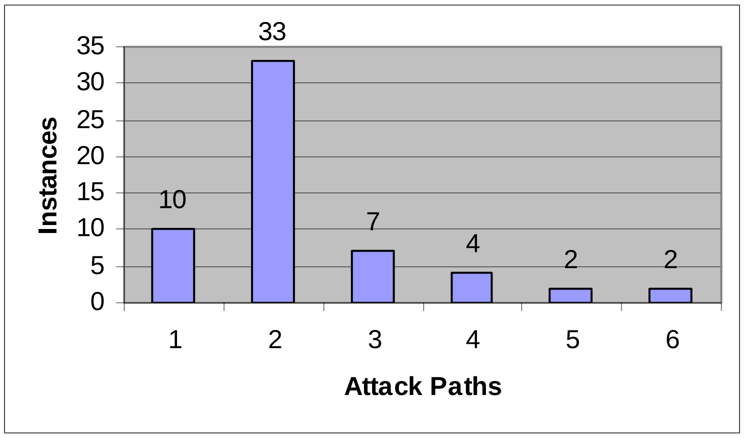


Figure 74: Valid Attack Paths for Regex15.

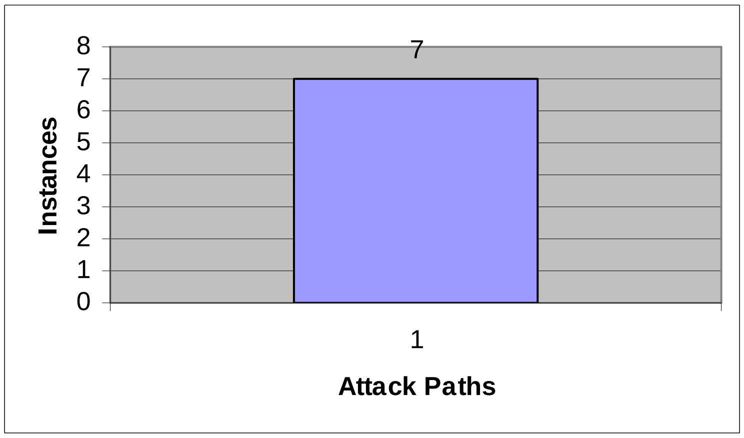


Figure 75: Invalid Attack Paths for Regex15.

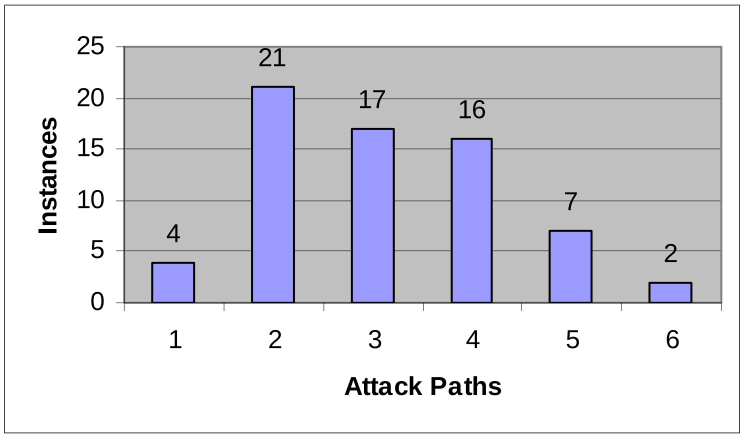


Figure 76: Valid Attack Paths for Regex16.

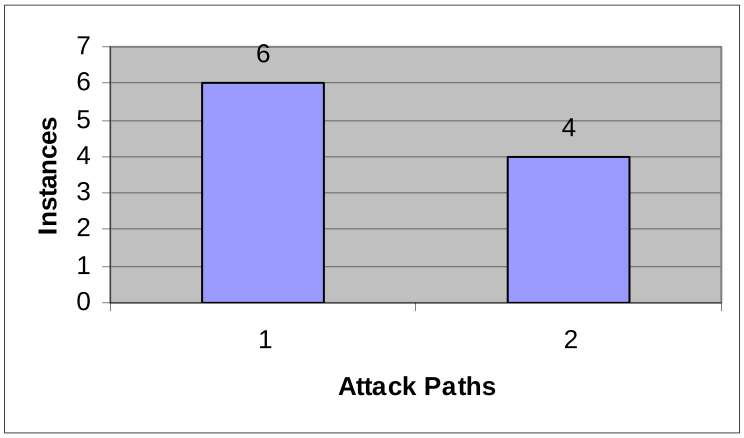


Figure 77: Invalid Attack Paths for Regex16.

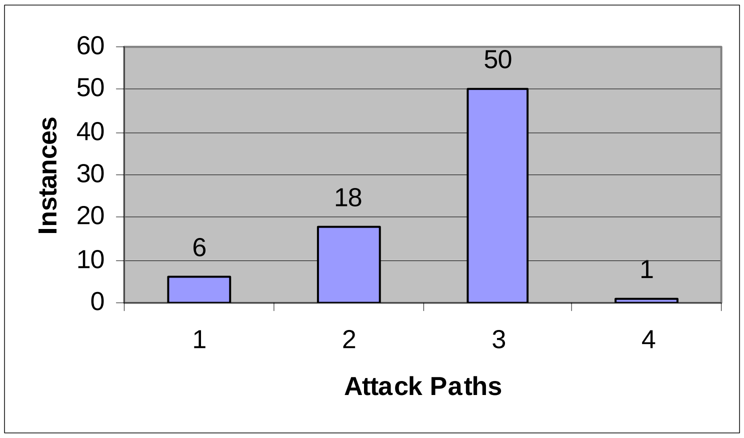


Figure 78: Valid Attack Paths for Regex17.

(No Invalid Attack Paths Reported)

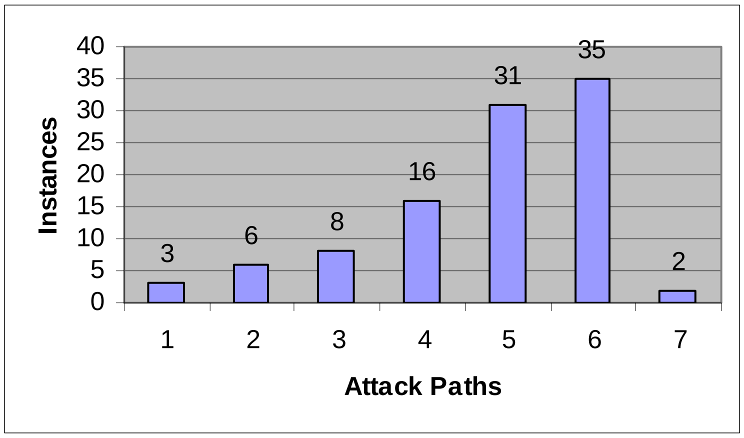


Figure 79: Valid Attack Paths for Regex18.

(No Invalid Attack Paths Reported)

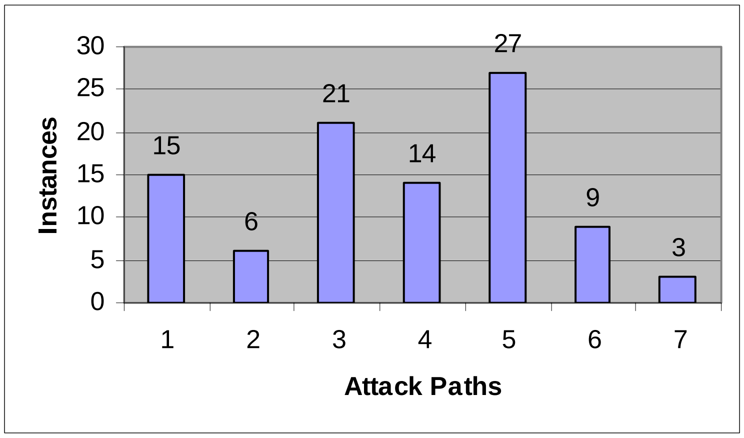


Figure 80: Valid Attack Paths for Regex19.

(No Invalid Attack Paths Reported)

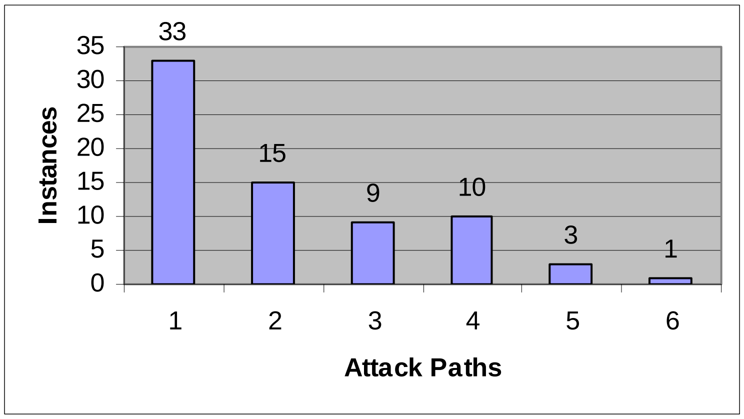


Figure 81: Valid Attack Paths for Regex20.

(No Invalid Attack Paths Reported)

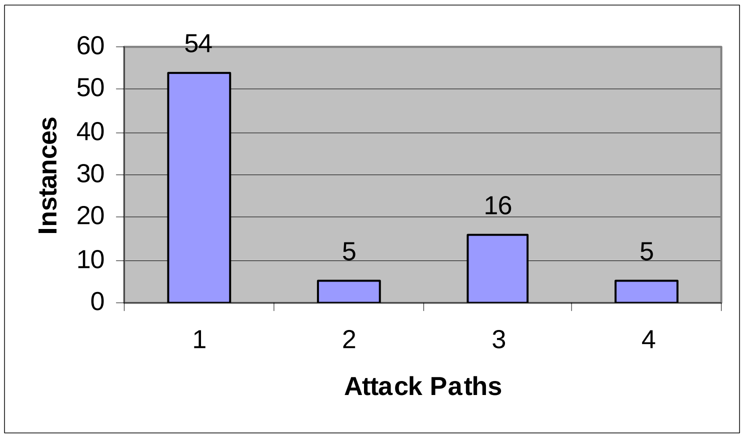


Figure 82: Valid Attack Paths for Regex21.

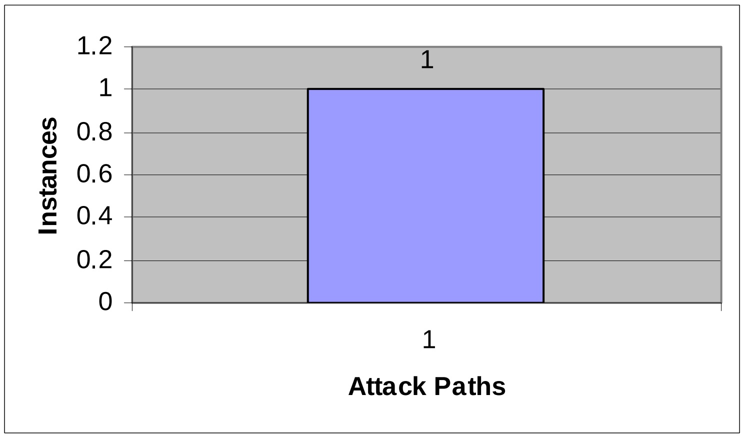


Figure 83: Invalid Attack Paths for Regex21.

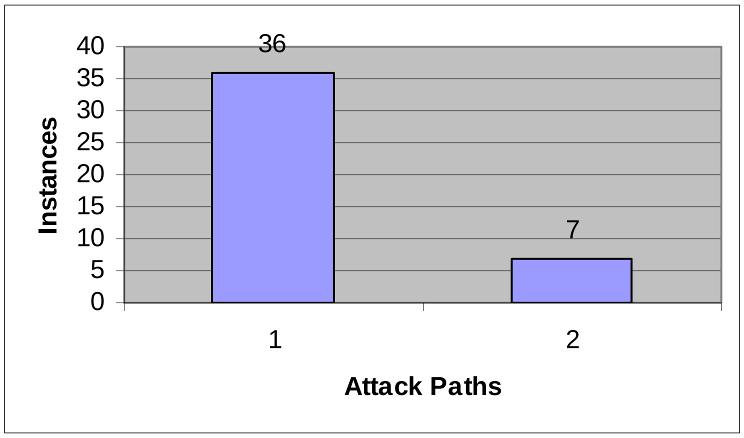


Figure 84: Valid Attack Paths for Regex22.

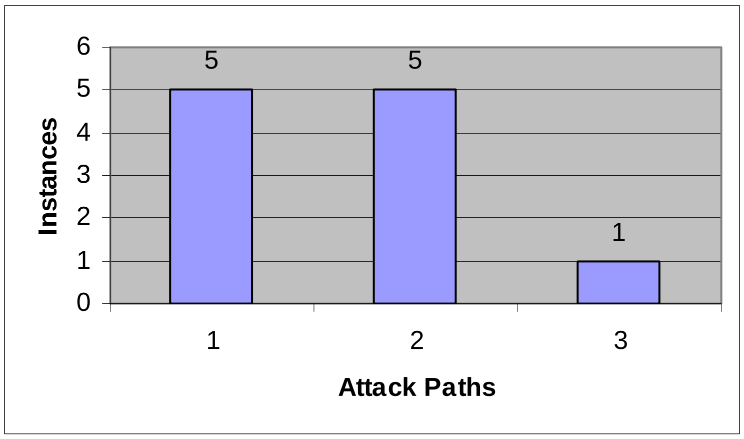


Figure 85: Invalid Attack Paths for Regex22.

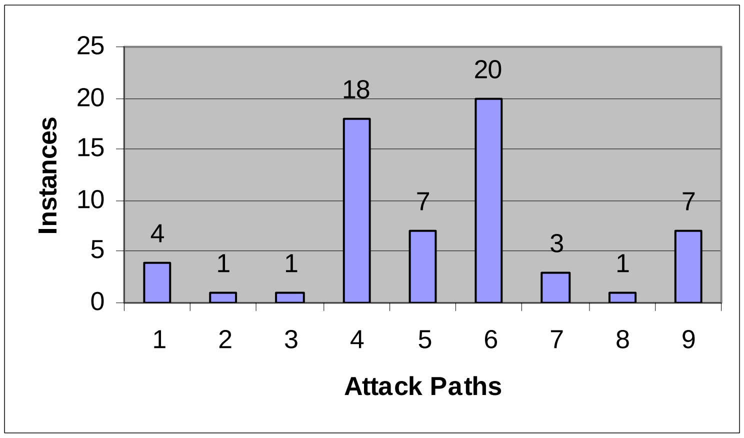


Figure 86: Valid Attack Paths for Regex23.

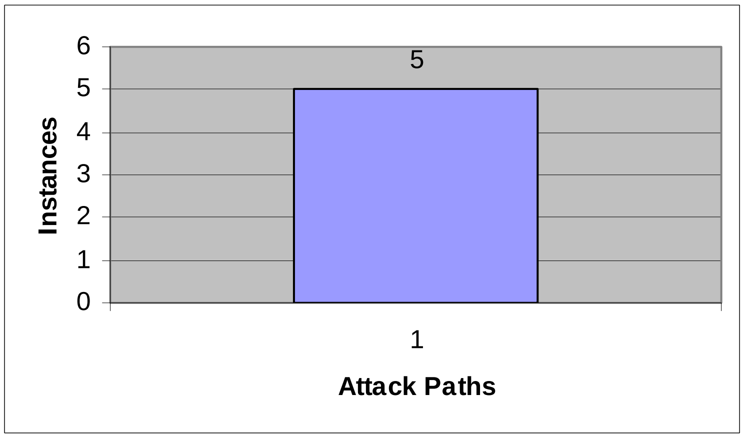


Figure 87: Invalid Attack Paths for Regex23.

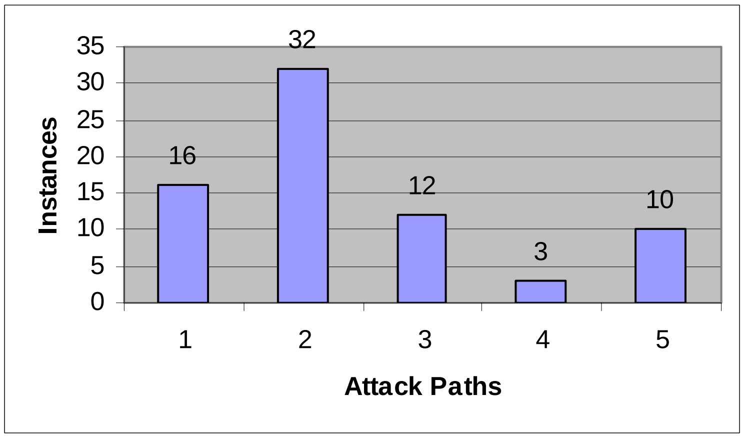


Figure 88: Valid Attack Paths for Regex24.

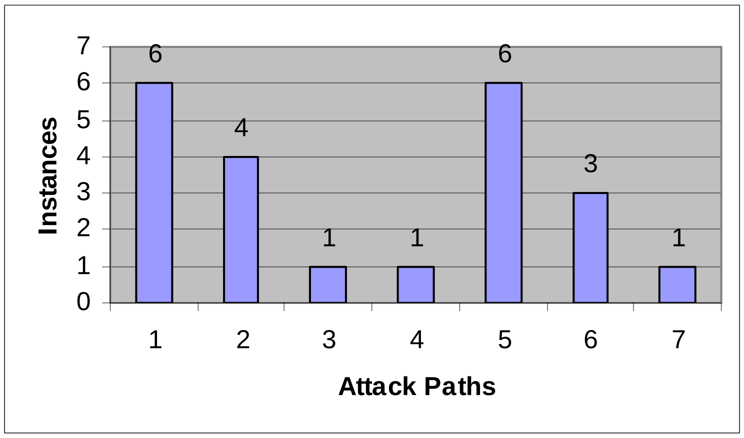


Figure 89: Invalid Attack Paths for Regex24.

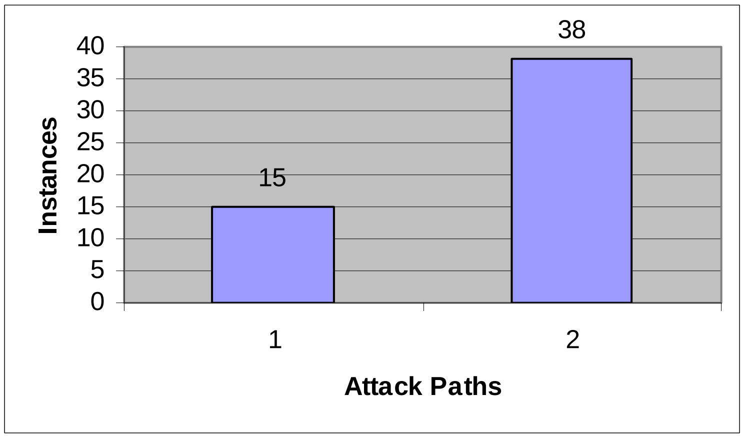


Figure 90: Valid Attack Paths for Regex25.

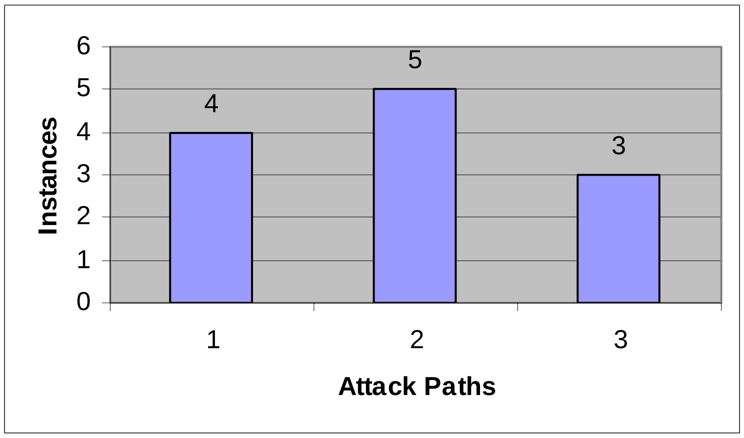


Figure 91: Invalid Attack Paths for Regex25.

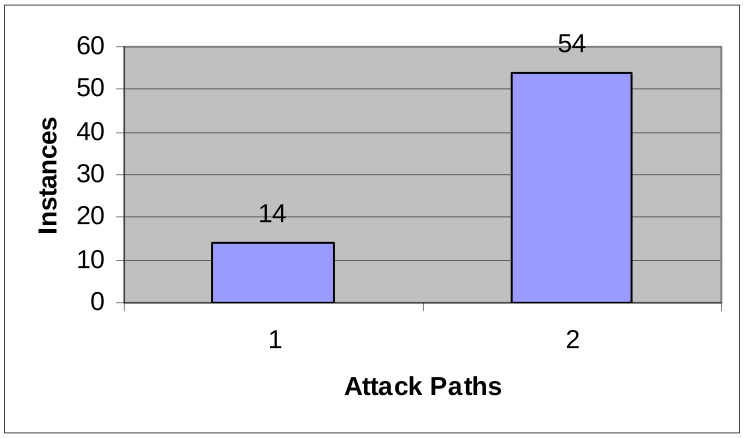


Figure 92: Valid Attack Paths for Regex26.

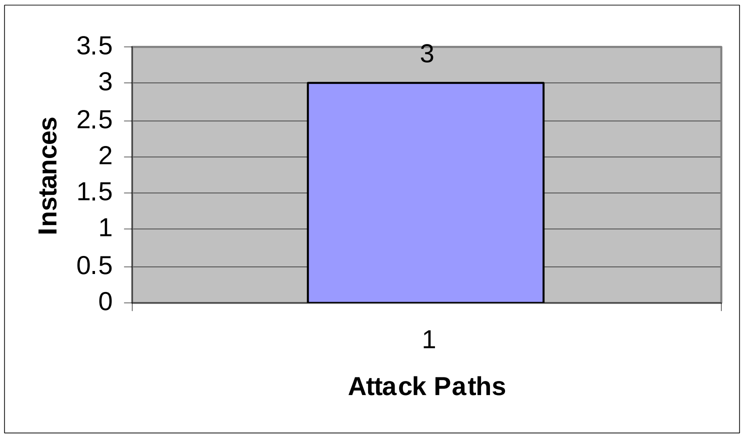


Figure 93: Invalid Attack Paths for Regex26.

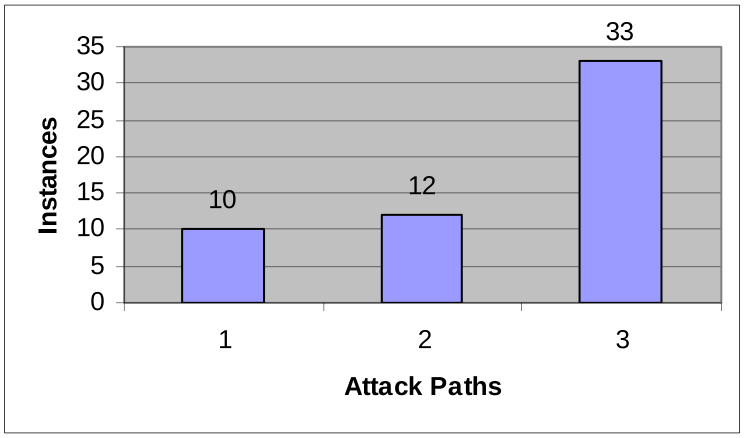


Figure 94: Valid Attack Paths for Regex27.

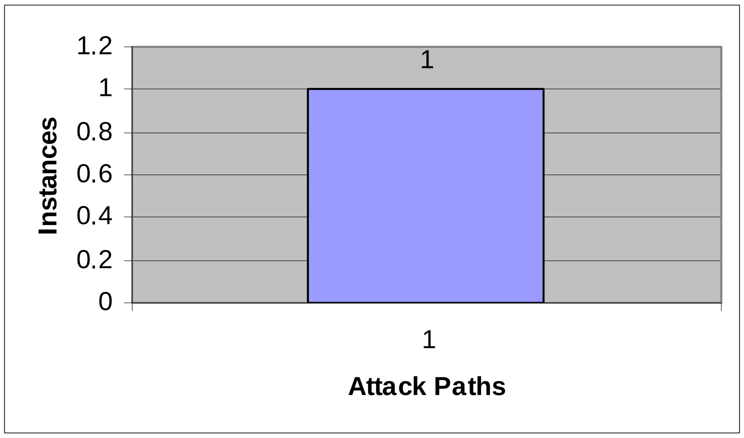


Figure 95: Invalid Attack Paths for Regex27.

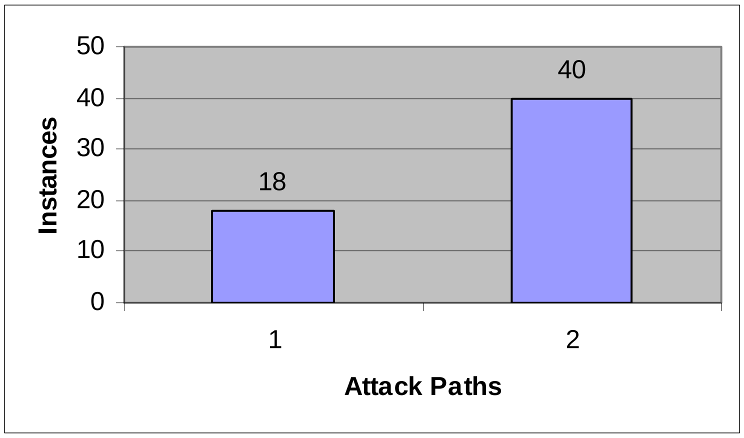


Figure 96: Valid Attack Paths for Regex28.

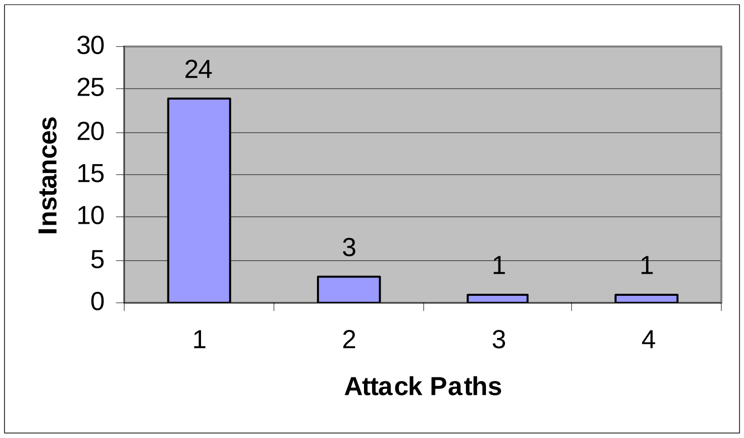


Figure 97: Invalid Attack Paths for Regex28.

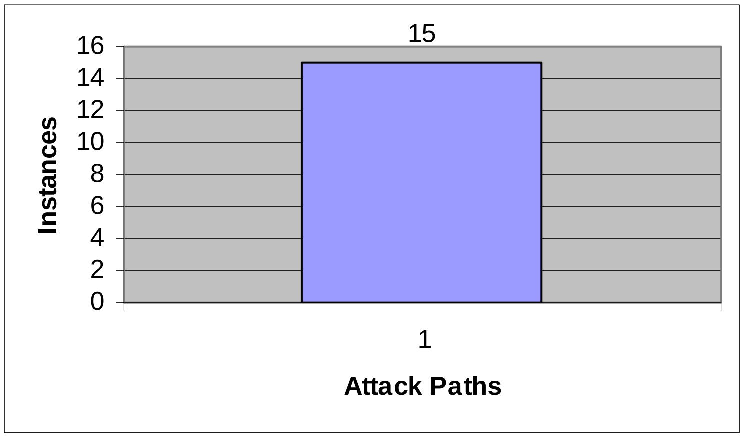


Figure 98: Valid Attack Paths for Regex29.

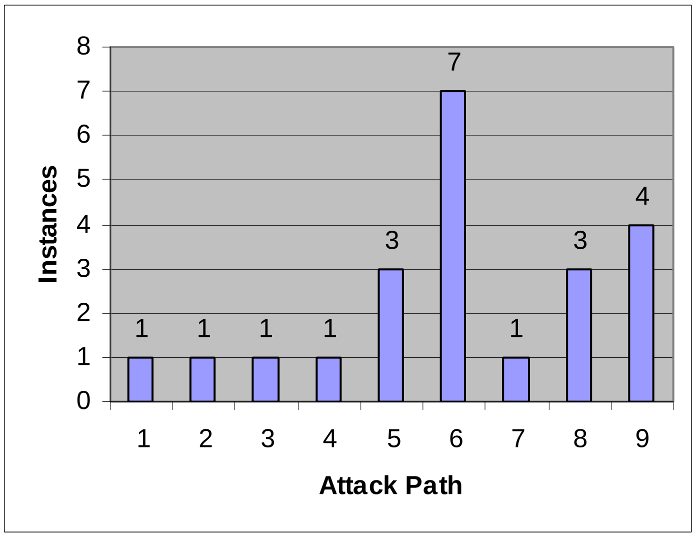


Figure 99: Invalid Attack Paths for Regex29.

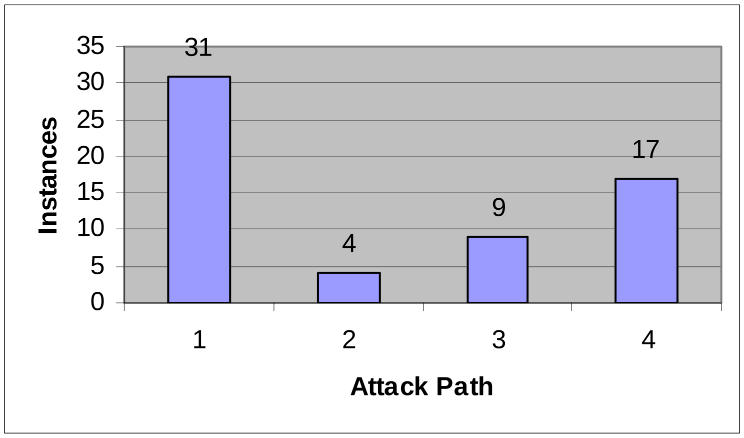


Figure 100: Valid Attack Paths for Regex30.

(No Invalid Attack Paths Reported)

Appendix IX

Validation Study Likert Scale

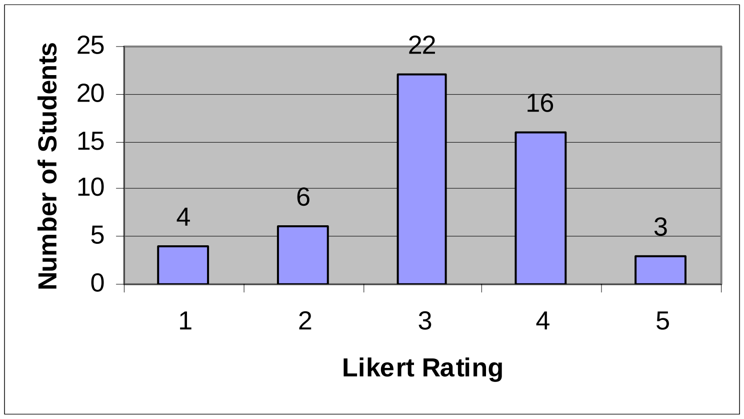


Figure 101: Validation Study Likert Scale

Appendix X

Student Comments for the Validation Study

Does not Scale

1. Like other testing methods, limited to how much it can really test (29^29 paths in this single example).
   -Within each path also lies a number of testing parameters as well; testing the range of a single integer along one point in a single path makes a complexity of (2^31)(29^29) ... simply not feasible.
2. It seems to me that this is not scalable to large environments. There are too many application level vulnerabilities and physical vulnerabilities to take into account all possibilities. It seems novel in approach, but I am not sure it will hold up in a robust environment.

The Approach is Bad

1. It takes a complicated system and shows 100's of potential ways it can be vulnerable. Way to many ways for all of them to be usefully examined. And all of them rely on incomplete information and guesswork. I hope that this was either a "trick" assignment or a joke of some kind.
2. it felt like a paint by numbers with a network
3. Doesn't really help the student understand what's going on, doesn't translate well from diagram analysis to actually using it to identify vulnerabilities.
4. I really didn't learn much more than simple patterns
5. not terribly informative. Mostly time-consuming and repetitive.
6. below average. While I do understand how this helps, I think most of the obvious flaws I already knew and the other ones I didn't understand enough to know if I did it right.
7. this assignment was pretty monotonous and i didnt feel like i learned alot
8. Students who know, let's say, what component has "cookies" might rate it to 5. But I'm sorry, it was not much help for me.
9. i don't feel this is how it's done in the real world and thus it isn't relavent
10. This exercise does a decent job of showing you the routes that attackers might take when conducting a malicious activity. Since there is no real interaction/experience, this exercise is limited in its effectivcness. I learn best with hands-on activities.
11. I don't like the approach really, but understand that it is not the easiest subject to bring across though. I understand the need for the design to be abstracted to be general in case, but the nature of the profiles in comparison to the design is so "out of whack" in many instances that the question became useless to me and I simply read the description of the attack in order to learn something. Given all of this, the time involved with this homework in terms of value of learning is wasted in comparison to our other homeworks. I spent less time on all of those (the pgp key, the picture, the key logic, etc) combined than I did on this and learned substantially more

The Approach has Limitations

1. This approach is good for identifying KNOWN weaknesses. That makes this way of acknowledging security risks a good start. However, there must be an effort to come up with new hacks in order to make this an excellent approach.
2. when it's all drawn out, it's easy to find the vulnerabilities, but i believe more are out there not detectable by an architectual diagram
3. if the design is incomplete or subject to change, this method will be a waste of time.
4. if a complete design is available, i suppose this can be a useful tool.
5. As long as you identify all the parts that are secruity vulenabilties. There could be vulenabilties or components that are vulenrable in this system that aren't show which makes it hard to indentify them.
6. unless highly trained, it seemed like a difficult task to do.
7. Such a system for identifying architectural security vulnerabilites is only good if the documentation and material supporting it is also good. Since there isn't much explanation on how to use the system, wading through the design is too complicated (and designs should be simple).
8. I would say average if you completely understand the process, but it seems vague to me. It is a good high level set up though.
9. I guess it would work good if there were paths that were used many times, so i would presume that those would be the most vulnerable paths.

The Approach is Advantageous to Software Engineering

1. It's not bad, but not as good as it could be. I think it'd be best to have all the people involved with the project to sit down together and try to brainstorm possible security risks. This would be a good start, and they could build off of it, but more than just this would be needed.
2. I think it is good to approach security design early on, however, there are many possible paths of attack here and this method makes it very difficult to check that you have covered every path.
3. From a good viewpoint, organize the system and being able to trace pathways visually can help deter common attacks and trace sources/restore systems quicker.
4. such an approach can make the program more secure by covering the domain of likely attacks. Also, it is good to take care of these things at the design stage so that the desired changes can be made and that too at a low cost.

The Approach is Good

1. Between average and above average since it takes into account a huge number of combinations a hacker could use to attack the system. Besides the system to analyze include a really good number of devices that can be present in a real system.
2. I think its a pretty neat approach as by acknowledging that a system design contains the same vulnerable component sequences as previously attacked software applications, developers can be warned about possible attacks in their future code.
3. It's okay, maybe a 4. It's simplistic, which makes it both good and bad...
4. I was interesting and learned a lot from it. It was a good idea.
5. software on top of the architecture could stop malicious activitiy, and looking at just the hardware wouldn't show that.

6. I like this way of identifying vulnerabilities. Instead of considering the vulnerabilities of each system independently, we get to see how systems can interact, sometimes poorly, which may cause vulnerabilities.
7. Excellent. This makes sure we go through an array of situations thus exposing vulnerabilites.
8. I give it an "Above Average" rating as it is a good way of flagging possible vulnerabilities. It is a fairly good method, but it doesn't mean that secure programs don't proceed on the same paths, as well. It is a way, it seems, to focus concern for possible security vulnerabilities.
9. An excellent first attempt to automated a security review of a design. However, some of the paths seem unrealistic to me.
10. It is very comprehensive and may even be too much so! Very good for high security considerations (governement/classified networks
11. While this method of securing a system should be used along side other approaches for the maximum level of security, it still does an amazingly good job of letting you visualize the attacks, and to identify obscure paths that otherwise might go unnoticed.
12. it gives the network admin an overview of the whole system. this can be used to pin point any weaknesses in the system. also, it can help identify any attacks that might occur within the network.
13. it was a cool little assignment, though some i didn't understand it was neat to see how some of these exploits snaked their way through
14. This is a great idea i give it a 5. Because you can find similar attack patterns and prevent it in the future.

Did Not Understand Assignment

1. It's good but it's not excellent because I had a hard time understanding what am i suppose to do and were to start but ones i figured out it was not that hard..
2. It would be better if there we're more examples given covering some of the more complex regular expressions used.

There is Not Enough Information to Understand the Approach

1. By looking at the high-level design we can clearly figure out the possible attacks, but its hard to figure out with having implemented that design, so that we know what the low level components are. With out know the low level components its hard to figure out ALL the possible attacks.
2. Not sure what i was doing. Just went with the flow. Need more explaination*
3. If greater detail was given – eg. code vulnerability samples it would be more effective in my opinion.
4. The system design needs to be much more detailed to accurately predict vulnerabilities.
5. It seems to be too vague. Instead of 30 regular expressions, maybe there should be less, more detailed, expressions that would give a better understanding of the material.
6. Some of these vulnerabilities are easy to find but for most part where there are variables and reads and writes involved, it is very hard to detect attacks.

Approach Should be Automated

1. This approach is probably helpful in detecting attacks especially since the same attacks are repeated over and over. But, there should be a mechanize to check for any new attacks or unusual behavior.
2. I guess this is a good approach, especially if it is automated. It formalises the design of the system, which makes it easy to see all the interconnections. Of course, it is still a hard homework assignment.

Profiles Needed?

1. I like the examples listed with each attack
2. profile does not have to be explored to understand these concepts.

Appendix XI

Student Questions in the Validation Study

Threads from the online message board.

1. **Student Question**
We don't have to include assumptions right?

Also, am I just looking at this wrong, can't for number 2, you list these which are all pretty close to the same:

26-25-24-23-19-12
26-25-24-23-19-20-12
26-25-24-23-19-22-21-12
26-25-24-23-19-22-21-20-12

Or is this going overboard. and just the first one is sufficient?

Are we suppose to list every possible path it can take to achieve the RegEx, or just the most obvious ones?

My Response
Assumptions are *not* required. Only put them in your answer if you think you need them.

For #2, the attack deals with sending thousands of headers to the server. Since the GET/POST requests are not necessarily involved in the attack you don't need to include components #20, #22 (thus your first answer is the best one)

2. **Student Question**
is data in hard drive? if yes, does user have to access log

My Response
Component # 14 is botha database and a database server so assume there is a hard driver on component 14. This will make it simpler than going to the access log to get to the hard drive. You may assume each server has a CPU and Hard Drive and you can stop there in your component sequence unless you need to get to that access log. Good question.

3. **Student Question**
can packet go from database server --> web application --> http message & header handler

My Response
Yes, it is certainly possible depending how you see the system being *implemented*. Include your assumption in your answer

4. **Student Question**
Can this path exist 17-18-25-24-23-19-11-12

My Response

Sure. It seems possible that the HTTP Message Handler could log the messages.

I posted this for clarify the assignment.
Since the system design is an abstraction, not all the little components will be shown in the design. However, the major components will be represented and you can simply assume in your answer that the little components do exist.

For example, these are just a few components not shown in the design are
CPU (there will always be a CPU somewhere in a system)
hyperlink (implied to be on the web pages component #25)
HeaderFieldBufferWrite (assume this to be taken care of by the HeaderHandler #19)
The “variable”, “Filename”, “Header” components in RegEx #4 can be assumed to be passed in on the URL from a client. There is no component that represents this, but you can assume that a client may make a POST request like this since there is a POST request handler (#22)
ServerConnectionState – this is the state of the server/client connection, that is not shown, but you know it must exist for any communication to exist.

5. Student Question
There is a CPU somewhere but what component number I need to include or just assume there is a CPU inside database server for question 10

My Response
Just assume there is a CPU inside the servers.

Questions received from email
1.

1) You give A* A+ and A+B examples. What about the ones with no +, *, or A+B? They can only happen once, correct? If so, do they HAVE to happen? That is to say, an object without any of these, such as #3:
"(Client)(HTTPServer)(GetMethod)(GetMethodBufferWrite)(Buffer) ", all of these can happen only once, but they MUST happen once?

2) In your example, the 3rd ansewr you give states an "assumption". Why do we need to state the assumption for that path? Beccause they are going through a firewall? Would the answer be wrong without the assumption? It
looks right to me, so long as they can break through the firewall.

3) You ask "What attack paths are possible for regular expression ##?" Does that mean we have to list all possible attack paths for that regular expression or just a single one?

4) As for answers, we ONLY need to answer with a number sequence, and not explain what function each performs? ie., in your regex notes, #4, you talk about the client performing 2 functions. If we put 1-2-2, is that
sufficient? My understanding was that the number sequence was enough.

You say that some components may not exist in the design. If this is the case, how do we represent them in our answer if there is no number to associate with them? i.e., you have components such as PostMethod,
Variable, Filename, GetMethod, HTTPContent-Length, CommandLineArgumentEntry, etc. How are we to know how to represent these or what path requires you to access them?

2.

I'm having a hard time to do this homework. I don't know much about this stuff. Can you please help?

Can I use the answers in the example to answer question 1?

Can this path exist 17-18-25-24-23-19-11-12?
Is there a proxy firewall from 24 to 27?

Can this path exist 16-15-14-13-19-12?

What component is HeaderFieldBufferWrite

3.
Hi, i'm kind of lost with this assignment and i don't know for sure if what i did is ok or not. I don't know if you can take a look to the submition i did already and tell me if at least i got the idea or i am completely wrong. That would be great. If you can not do that i would appreciate if you can guide my in some of the cases like number 29, 13, 16 & 24.

Does (A+B) mean event A or event B occurs only one time; or does it mean event A or B occurs at least one time?

In other words if I have (A) does that mean A occurs only one time or at least one time?

The example given in http://www4.ncsu.edu/~mcgegick/example.html
looks like the answer for problem 1 of hw 4.

I am correct or NOT.????????????

26-25-24-23-19-11-12
16-15-14-11-12

1-2-3-11-12
Assume a user (or script) can excessively log into the authentication server without tripping an intrusion detection alarm and proper software checks are not in place, then the vulnerability may exist.

1010101010101010101010
I just want to make sure that I am understanding what I am doing. Just to build my confidence on how to do the hw.

Would this be how the answers for problem 2 of HW4 supposed to look like.
26-25-24-27-28-21-20-19-11-12
26-25-24-27-28-21-20-19-13-12
26-25-24-27-28-21-22-19-11-12
26-25-24-27-28-21-22-19-13-12
26-25-24-27-28-23-19-11-12
26-25-24-27-28-23-19-13-12

This is the profile (Client+)(Server+)(MessageHeaderHandler+)(Hard Drive+)

10101010101010101010
I can NOT figure out this
[(Client)(HTTPServer)(GetMethod)(GetMethodBufferWrite)(Buffer)] regex in the design.

Any hints.

4.
Regex for problem 3=
(Client)(HTTPServer)(GetMethod)(GetMethodBufferWrite)(Buffer)
What do you do when you get to the buffer?
This what I came up with 26-25-24-23-19-20-21

The source code below looks like a flavor of the virus W32.NetSky.P@mm

I do NOT understand how is this helpfull. Can you give me some enlightment of this. Thx.

5.
or problem 4 the only thing I see that matches the regex:
(Client)(HTTPServer)(PostMethod)(Variable + Filename + Header)(Buffer)
is the following:
26-25-24-23-19-22-21

I am not sure of anything else.

6.
I was wondering if we have to give every path possibility for each set of regular expressions.

Appendix XII

Validation Study Assignment

Designing for Security

About this assignment

New attacks to software may not be predictable, but a software team should not permit recurrences of old attacks in new software. Many of the vulnerabilities in this assignment are repeated in the Bugtraq database (http://www.securityfocus.com). By acknowledging that a system design contains the same vulnerable component sequences as previously attacked software applications, developers can be warned about possible attacks in their future code. This assignment encourages security to be built into the software application instead of being added at the end of the software process as an afterthought.

Background
Software engineering approaches are being investigated for building secure software. Viega and McGraw claim that "[t]he fundamental technique [of security] is to begin early, know your threats, design for security, and subject your design to thorough objective risk analyses and testing" [1]. One approach is to define attack patterns that are used to find vulnerabilities in a system. This assignment will be used to evaluate the human-readable aspect of representing security vulnerabilities with regular expressions (regex for short). Regexs are utilized because they can be adapted into a programmable form for automation and because they attempt to be environment/language independent.

Directions

Thirty vulnerable component sequences have been abstracted into regular expressions to conceptualize the access paths used in known attacks. Each path contains at least one access control violation that risks the use of a resource in the system. Determine what component sequences in the system design can be represented by the regular expressions.
1. Search the system design for the string (i.e. the sequence of components) that is defined by each regular expression listed in the regular expression profiles .
2. Enter your findings in the essay formatted answer boxes in WebAssign. Write down the number associated with each component in the sequence path (e.g. 1-2-3) that corresponds to the regular expression (there may be more than one string in the design for a given regular expression). You may document any assumptions about the system you think are necessary to qualify your answer after your answer. Not all strings may be present in the design and multiple attacks may exploit the same string.

Interpreting a Regular Expression
Each regex represents a sequence of events that are involved with data flow between components. For example, (Client+)(Server+)(Log+)(Hard Drive+) shows a series of client requests, followed by a series of server actions, followed by a series of log updates, followed by a series of disk writes.

Use this example to guide you through the assignment.

Vital Regex Notes:
1. The regexs are abstract and can therefore be fulfilled in different ways (e.g. Server implies WebServer, Database Server, etc.)
2. Some regexs will be straightforward and others will require assumptions. Software designs do not often show many important low level details, so document any assumptions necessary about how an attack may occur.
3. Regexs only show the main components involved in a vulnerability. You may assume that a vulnerability exists if there are intermediate components/processes in the design sequence. Include the intermediate components in your answers.
4. One component may be responsible for consecutive multiple events in a regex (e.g. the Service Client Request routine may read a packet stream and write to a buffer). You may represent the repetition as something like 1-2-2 or simply 1-2 (in this example, component #2 carries out two consecutive events.)
5. Review of regular expressions
a. (A*) (Kleene closure) implies event A occurs 0 or more times.
b. (A+) implies event A occurs one or more times.
c. (A + B) implies event A or event B occurs.

Vital System Design Notes:
1. Arrows represent data flow.
2. Circles represent processes/methods.
3. "Client" and "user" are synonymous. A client can either be a browser such as Netscape Navigator or a user.

Please send all comments/questions to mcgegick@ncsu.edu.

References

[1] J. Viega and G. McGraw, Building Secure Software: How to Avoid Security Problems the Right Way. Boston, MA: Addison-Wesley, 2002.

1. Question 1 [340089] What attack paths are possible for regular expression #1?

[3]

2. Question 2 [340090] What attack paths are possible for regular expression #2?

[3]

3. Question 3 [340091] What attack paths are possible for regular expression #3?

[3]

4. Question 4 [340097] What attack paths are possible for regular expression #4?

[3]

5. Question 5 [340098] What attack paths are possible for regular expression #5?

[3]

6. Question 6 [340099] What attack paths are possible for regular expression #6?

[3]

7. Question 7 [340100] What attack paths are possible for regular expression #7?

[3]

8. Question 8 [340101] What attack paths are possible for regular expression #8?

[3]

9. Question 9 [340103] What attack paths are possible for regular expression #9?

[3]

10. Question 10 [340104] What attack paths are possible for regular expression #10?

[3]

11. Question 11 [340124] What attack paths are possible for regular expression #11?

[3]

12. Question 12 [340105] What attack paths are possible for regular expression #12?

[3]

13. Question 13 [340106] What attack paths are possible for regular expression #13?

[3]

14. Question 14 [340107] What attack paths are possible for regular expression #14?

[3]

15. Question 15 [340108] What attack paths are possible for regular expression #15?

[3]

16. Question 16 [340109] What attack paths are possible for regular expression #16?

[3]

17. Question 17 [340110] What attack paths are possible for regular expression #17?

[3]

18. Question 18 [340111] What attack paths are possible for regular expression #18?

[3]

19. Question 19 [340112] What attack paths are possible for regular expression #19?

[3]

20. Question 20 [340113] What attack paths are possible for regular expression #20?

[3]

21. Question 21 [340114] What attack paths are possible for regular expression #21?

[3]

22. Question 22 [340115] What attack paths are possible for regular expression #22?

[3]

23. Question 23 [340116] What attack paths are possible for regular expression #23?

[3]

24. Question 24 [340117] What attack paths are possible for regular expression #24?

[3]

25. Question 25 [340118] What attack paths are possible for regular expression #25?

[3]

26. Question 26 [340119] What attack paths are possible for regular expression #26?

[3]

27. Question 27 [340120] What attack paths are possible for regular expression #27?

[3]

28. Question 28 [340121] What attack paths are possible for regular expression #28?

[3]

29. Question 29 [340122] What attack paths are possible for regular expression #29?

[3]

30. Question 30 [340125] What attack paths are possible for regular expression #30?

[3]

31. Question 31 [340237] How much time did you spend on this assignment? Give your answer in minutes.

[3]

32. Question 32 [341495] What parts of the assignment are not clear?

[3]

33. Question 33 [341496] Rate the approach of identifying architectural security vulnerabilities and give a brief explanation. 1. Poor 2. Below Average 3. Average 4. Above Average 5. Excellent

[3]

Submit for Testing

Knowledge Base

Key	RegEx	Profile
1	(Client+)(Server+)(Log+)(Hard Drive+)	An attacker can exceedingly access either the web server or database server augmenting the access log file and eventually filling the hard drive causing the system to crash, a denial-of-service attack (DoS).

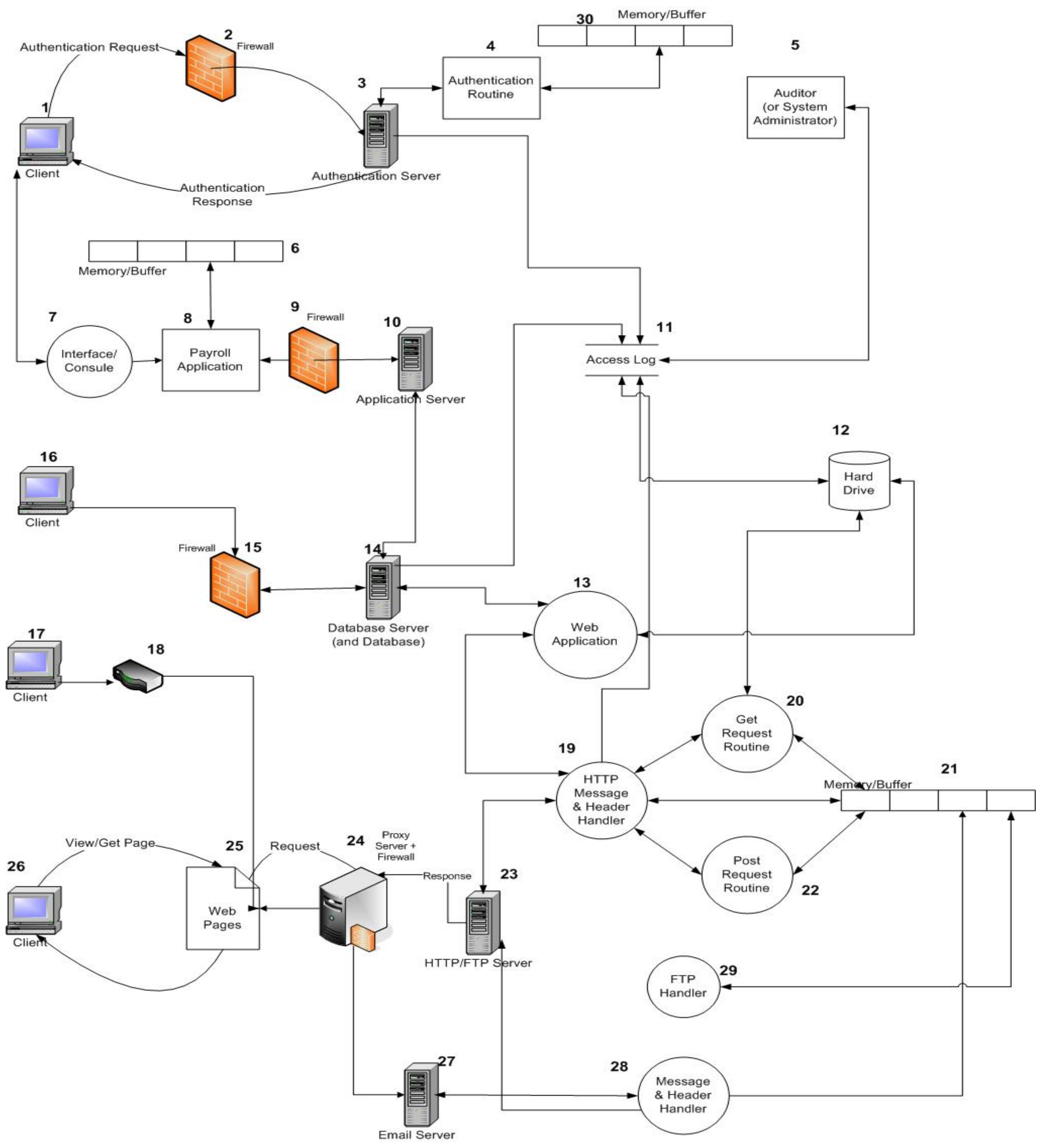
Key	RegEx	Profile
2	(Client+)(Server+)(MessageHeaderHandler+)(Hard Drive+)	A client may send a message with thousands of headers to a server (HTTP, email, etc), causing a denial-of-service.
3	(Client)(HTTPServer)(GetMethod)(GetMethodBufferWrite)(Buffer)	Writing an excessively long Get Request into a small buffer will cause a buffer overflow . Either the requestURI or HTTP version is too long for the buffer. Privileges can be escalated or a DoS may occur. See http://downloads.securityfocus.com/vulnerabilities/exploits/sp-samihttpddos.c as an example of how this may be implemented.
4	(Client)(HTTPServer)(PostMethod)(Variable + Filename + Header)(Buffer)	Excessively long Post requests can cause a buffer overflow. The attacker may send an excessively long variable name, filename, or header.
5	(Client)(Server)(HeaderFieldBufferWrite)(Buffer)	Excessive header field values (HTTP headers, email headers, etc.) written into small buffers will cause a buffer overflow.
6	(Client)(HTTPServer)(HTTPMessageHandler)(Log)(Sysadmin)(LogEntryRead)	A system administrator can induce a buffer overflow when viewing an excessively long log entry. Notice that the attack is against a sys. admin. (as opposed to a typical user)
7	(Client)(HTTPServer)(PostMethod)(HTTPContent-LengthHeaderValue)(HTTPMessagePayloadLength)(ServerConnectionState)	Sending a value via the Post method in the Content-Length of the HTTP header less than the content-length of the message may cause the socket to stay open. This has potential to cause a DoS in some systems.
8	(User)(UserNameEntry)(PasswordEntry)(Server)(AuthenticationRoutine)	Writing an excessively long string of characters for either the username or password into a small buffer will cause a buffer overflow.
9	(Client)(SQLInput)(Server)(WebApp)(Database)(Data)(Buffer)	A long query string submitted to the database may overwrite a buffer and eventually escalate privileges of the attacker.
10	(Client)(SQLInputField)(Server)(WebApp)(Database)(CPU)	An attacker may submit a malicious SQL query (such as a Cartesian join of all tables) consuming the CPU.
11	(User)(CommandLineArgumentEntry)(Application)(ApplicationServer*)(CommandLineArgumentBufferWrite)	A user may enter excessively long command line parameters causing buffer overflows.
12	(Client)(HTMLPage)(Server)(Hard Drive)	A user may submit an excessive

Key	RegEx	Profile
		amount of data in an HTML page (e.g. in an HTML form), thus filling up the server's hard drive
13	(MaliciousClient)(Injection of malicious HTML tags, script in URL, Form)(Cookie*)(FormData*)(ServerVariables*)(VictimClient)	Injecting 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/CSSTAG can give an attacker cookie-based authentication data, HTML form data, or server information. A simple example can be illustrated by the following: http://www.example.com/<script>alert('XSS')</script> Where any malicious script code can be inserted between the script tags.
14	(User)(Computer)(SyslogFunction)(Log)	It is possible to corrupt memory by passing format strings through the Syslog(), a logging function. This may potentially be exploited to overwrite arbitrary locations in memory with attacker-specified values.
15	(User)(ReadUserInput)(EnvironmentVariableWrite)(Buffer)	A user can read in a malicious environment variable that exceeds a buffer and thus possibly escalate the privileges of the attacker.
16	(User)(File)(FileRead)(BufferWrite*)	A file that is corrupted may cause an exception to be thrown in application as it is read, thus causing a DoS. Also, if the file is made longer than what is expected a buffer can be overrun and privileges can be escalated.
17	(Client)(Hyperlink)(Server)	Supplying excessively long data into the hyperlink can cause a buffer overflow. If a hyperlink is used to connect to a session, then the malicious client can take over the application.
18	(Client+)(Server+)(MessageHeaderHandler$^+$)	Client can send a negative, NULL, invalid value (e.g. not include '':' between header name/value) for a header field and cause a DoS. This may occur on any server with messages that have headers (email server e.g. to, from, subject)
19	(Client)(Server)(DaemonProcess)(Hard	A script that makes an excessive

Key	RegEx	Profile
	Drive)	number of connections in a small period of time to the listening daemon process of a server may cause a DoS.
20	(UserInput)(IntegerEvaluationRoutine)	A user that supplies an integer larger than the variable type expected may cause an exception/buffer overflow or DoS.
21	(Client)(HTTPServer)(GetRequestRoutine)	A malformed URL (as well as the in search string) may be NULL, contain Unicode chars, or format string specifiers and cause a DoS. Also, the URL may contain directory traversals in which an attacker can view files on the file system. An example of a directory traversal may look like this: http://www.example.com/../../../../etc/passwd. A search string (or query string) is the data that follows the '?' in a URL (e.g. http://www.example.com/path?param1=arg1&param2=arg2)
22	(User)(GUI/Browser)(BookMarkSave)(BookmarkBufferWrite)	A user maliciously saves a long bookmark in a web browser can cause a buffer overflow and escalate their privileges. This normally happens in custom made Help browsers that allow users to read the HTML documentation describing their product.
23	(Client)(SearchString)(Server)(Data)(Client)	A client that requests data from an untrusted server may receive large data and result in a buffer overflow. This most often happens in online gaming environments.
24	(Read)(FileHeader)(Buffer)	Reading the length of a long file name into a buffer may cause a buffer overflow.
25	(Client)(EmailHeader)(Firewall)(Buffer)	An attacker may send an email with excessively long headers to overflow buffers in a firewall to escalate their privileges.
26	(Client)(HTTPRequest)(ProxyServer)(Buffer)	A user can submit a long HTTP GET request to the proxy server and cause a buffer overflow.
27	(Client)(RequestMessage)(Router)(CPU)	Malformed headers of a message (e.g. failing to supply expected headers) may cause a DoS in a network router.
28	(Client)((FTPCommand +	A user that submits an overly long

Key	RegEx	Profile
	MailCommand) + OSCommand)(FTPServer + Mail server))(Buffer)	OS command or FTP/Mail command in an FTP program can cause a buffer overflow in the FTP/Mail server.
29	(Class)(Subclass)(OverriddenSecuredMethods)	When extending or subclassing a class (e.g. in Java) developers must make sure that the methods they override remain secure otherwise a security vulnerability will be possible. This is a source code oriented vulnerability.
30	(Client)(Application)(EnvironmentVariable + ProgramVariable + URLparam)(MaliciousIncludeFile)	An attacker can change/influence an environment or program variable to point to an "include" directory on a remote machine. The attacker's include file will then executed on the target system. This attack often occurs to PHP scripts.

The System Design



Authentication Request
2
Firewall
30
Memory/Buffer
4
Authentication Routine
5
Auditor (or System Administrator)
3
1
Client
Authentication Server
Authentication Response
6
Memory/Buffer
7
Interface/ Consule
8
Payroll Application
9
Firewall
10
Application Server
11
Access Log
12
Hard Drive
16
Client
Firewall
15
14
Database Server (and Database)
13
Web Application
17
Client
18
20
Get Request Routine
19
HTTP Message & Header Handler
21
Memory/Buffer
View/Get Page
25
Request
24
Proxy Server + Firewall
Response
23
26
Client
Web Pages
Post Request Routine
22
HTTP/FTP Server
29
FTP Handler
27
28
Message & Header Handler
Email Server